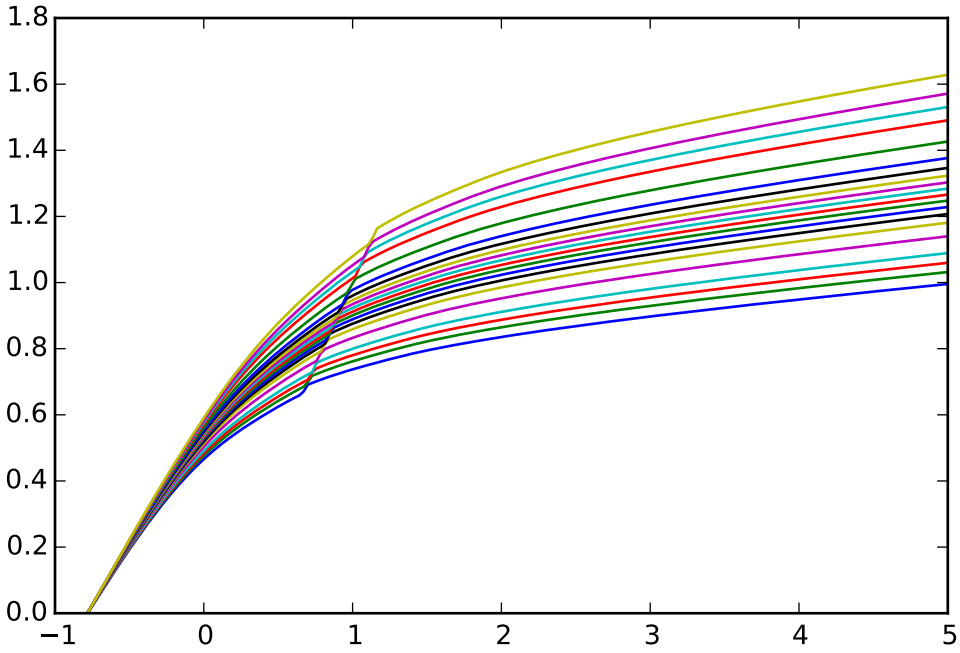
A User's Guide for HARK: Heterogeneous Agents Resources and toolKit

June 10, 2016



Contents

1 Introduction **3**
- 1.1 Getting Started . . . 4
- 1.2 Structure of HARK . . . 5
- 1.3 Other Resources . . . 6

2 General Purpose Tools **7**
- 2.1 HARKcore . . . 7
- 2.2 HARKutilities . . . 8
- 2.3 HARKinterpolation . . . 9
- 2.4 HARKsimulation . . . 9
- 2.5 HARKestimation . . . 10
- 2.6 HARKparallel . . . 10

3 Microeconomics: the AgentType Class **10**
- 3.1 Attributes of an AgentType . . . 11
- 3.2 A Universal Solver . . . 12
- 3.3 The Flow of Time and Other Methods . . . 12
- 3.4 Sample Model: Perfect Foresight Consumption-Saving . . . 13

4 Macroeconomics: the Market Class **15**
- 4.1 Down on the Farm . . . 16
- 4.2 Attributes of a Market . . . 16
- 4.3 Sample Model: FashionVictim . . . 18

5 Contributing to HARK **19**
- 5.1 What Does HARK Want? . . . 20
- 5.2 Git and GitHub . . . 20
- 5.3 Submitting Code . . . 20
- 5.4 Naming Conventions . . . 20
- 5.5 Documentation Conventions . . . 20

6 Future of HARK **20**
- 6.1 Future Tools . . . 20
- 6.2 Future Models . . . 20
- 6.3 Bounty Hunting . . . 20
- 6.4 All Aboard the Ark . . . 20

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. 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 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] Further, we seek to establish (with input from the community) standards for the description and specification of objects like discrete approximations to continuous distributions and interpolated function approximations, so that numeric methods can be quickly swapped without ugly "patching."

We hope that HARK will make it much easier and faster for researchers to develop solution and estimation methods for new models. The open source nature of HARK will make it easier for other researchers to audit and verify new models and methods, and to collaborate on correcting deficiencies when found. As HARK expands to include more canonical models and more tools and utilities, we can all spend less time managing numerical minutiae and more time fretting about identification arguments and data accuracy.

[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.

1.1 Getting Started

If you want to get started using HARK right away, this section provides a very easy quickstart guide for getting HARK up and running on your computer in just a few minutes. More information can be found in the `README.txt` file in the HARK repository (step 4), but the quick version for those who want to jump right in:

1. **Download Anaconda:** Go to https://www.continuum.io/downloads and download Anaconda for your operating system; be sure to get the version for Python 2.7.

2. **Install Anaconda:** Follow the easy instructions on that page to install Anaconda.

3. **Add extra packages:** (optional) If you want to use HARK's multithreading feature, you need to add two packages that aren't part of the default Anaconda distribution. Simply open a command prompt and do `conda install joblib` and `conda install dill`, accepting defaults to install.

4. **Download HARK:** Go to https://github.com/econ-ark/HARK, the home of the HARK repository. Click on the lime green button labeled "Clone or download" toward the upper right of the main panel, and select "Download ZIP".[2]

5. **Unzip HARK:** Unpack the `HARK.zip` file using any archive utility, like Peazip. Put the files anywhere you'd like, maintaining the internal directory structure.

6. **Run Spyder:** Open a command prompt and do `spyder`. Spyder is an interactive development environment (IDE) for iPython, a slightly prettier, more interactive flavor of Python.

7. **Open a HARK module:** Go to the directory where you put HARK and open any file with the `.py` extension; we recommend `/ConsumptionSaving/ConsIndShockModel.py`.

8. **Run the module:** Click on the green arrow "play" button toward the right side of the toolbar of icons at the top of the Spyder window (accept defaults if a dialogue box pops up). Congratulations! HARK is now up and running on your computer.

If you followed our recommendation to try `/ConsumptionSaving/ConsIndShockModel.py`, you should see graphical representations of the solution to a few consumption-saving models with idiosyncratic shocks to permanent and transitory income (see OTHER DOCUMENT for details of these models). If you chose a different module, you might get some examples of some other model, results of a structural estimation, or just a polite message that this particular module doesn't do much on its own. See the `README.txt` for a full list of modules from which you can expect non-trivial output.

[2] This is the fastest way to get HARK running on a computer. Later, if you want to make your own contributions to HARK by adding code (or fixing ours!), you'll need to clone the repository on your computer. See section 5.2 for more.

1.2 Structure of HARK

HARK is written in Python, an object-oriented programming language that has experienced increasing popularity in the scientific community in the past several years. A significant reason for the adoption of Python is the `numpy` and `scipy` packages, which offer a wide array of mathematical and statistical functions and tools; HARK makes liberal use of these libraries. Python's object-oriented nature allows models in HARK to be easily extended: more complex models can inherit functions and methods from more fundamental "parent" models, eliminating the need to reproduce or repurpose code.

As implied in the previous section, we strongly encourage HARK users to use the Anaconda distribution of Python, which includes all commonly used mathematical and scientific packages, an interactive development environment for iPython (Spyder), and a package manager that allows users to quickly install or update packages not included in the default distribution (conda).

Python modules (files with the `.py` extension) in HARK can generally be categorized into three types: tools, models, and applications. **Tool modules** contain functions and classes with general purpose tools that have no inherent "economic content", but that can be used in many economic models as building blocks or utilities; they could plausibly be useful in non-economic settings. Tools might include functions for data analysis (e.g. calculating Lorenz shares from data, or constructing a non-parametric kernel regression), functions to create and manipulate discrete approximations to continuous distributions, or classes for constructing interpolated approximations to non-parametric functions. Tool modules generally reside in HARK's root directory and have names like `HARKsimulation` and `HARKinterpolation`; they do not necessarily do anything when run.

Model modules specify particular economic models, including classes to represent agents in the model (and the "market structure" in which they interact) and functions for solving the "one period problem" of those models. For example, `ConsIndShockModel.py` concerns consumption-saving models in which agents have CRRA utility over consumption and face idiosyncratic shocks to permanent and transitory income. The module includes a class for representing "types" of consumers, along with functions for solving (several flavors of) the one period consumption-saving problem. When run, model modules might demonstrate example specifications of their models, filling in the model parameters with arbitrary values. When `ConsIndShockModel.py` is run, it specifies an infinite horizon consumer with a particular discount factor, permanent income growth rate, coefficient of relative risk aversion (etc), who faces lognormal shocks to permanent and transitory income each period with a particular standard deviation; it then solves this consumer's problem and graphically displays the results.[3] Model modules generally have `Model` in their name.

Application modules use tool and model modules to solve, simulate, and/or estimate economic models *for a particular purpose*. While tool modules have no particular economic content and model modules describe entire classes of economic

[3] Running `ConsIndShockModel.py` also demonstrates other variations of the consumption-saving problem, but their description is omitted here for brevity.

models, applications are uses of a model for some research purpose. For example, `/SolvingMicroDSOPs/StructEstimation.py` uses a consumption-saving model from `ConsIndShockModel.py`, calibrating it with age-dependent sequences of permanent income growth, survival probabilities, and the standard deviation of income shocks (etc); it then estimates the coefficient of relative risk aversion and shifter for an age-varying sequence of discount factors that best fits simulated wealth profiles to empirical data from the Survey of Consumer Finance. A particular application might have multiple modules associated with it, all of which generally reside in one directory.

1.3 Other Resources

In the incredibly unlikely scenario in which this document does not fill all of the gaps in your knowledge and answer any questions you might have while reading it, here is a collection of potentially helpful other resources.

- A tutorial on Python 2.7, straight from the source.
- Wikipedia article on object-oriented programming, not the subject of an edit war.
- QuantEcon, a collection of lectures on quantitative economic modeling from a couple of no-name economists, which in no way influenced HARK.
- A tutorial on git, the repository management and tracking system used by the HARK project, created by Linus Torvalds.
- A tutorial on GitHub, a website that provides a useful framework for git that makes it significantly more user-friendly, despised by Linus Torvalds.
- Wikipedia article on Linus Torvalds, just in case.
- The homepage for NumPy, a numerical package extensively used by HARK.

2 General Purpose Tools

HARK's root directory contains six tool modules,[4] each containing a variety of functions and classes that can be used in many economic models– or even for mathematical purposes that have nothing to do with economics. Some of the tool modules are very sparely populated at this time, while others are quite large. We expect that all of these modules will grow considerably in the near future, as new tools are "low hanging fruit" for contribution to the project.[5]

2.1 HARKcore

A key goal of the project is to create modularity and interoperability between models, making them easy to combine, adapt, and extend. To this end, the `HARKcore` module specifies a framework for economic models in HARK, creating a common structure for them on two levels that can be called "microeconomic" and "macroeconomic".

Microeconomic models in HARK use the `AgentType` class to represent the agents with an intertemporal optimization problem. Each model in HARK specifies a subclass of `AgentType`; an instance of the subclass represents agents who are *ex ante* homogeneous– they have common values for all parameters that describe the problem. For example, `ConsIndShockModel` specifies the `ConsumerType` class, which has methods specific to consumption-saving models with idiosyncratic shocks to income; an instance of the class might represent all consumers who have a CRRA of 3, discount factor of 0.98, etc. The `AgentType` class has a `solve` method that acts as a "universal microeconomic solver" for any properly formatted model, making it easier to set up a new model and to combine elements from different models; the solver is intended to encompass any model that can be framed as a sequence of one period problems. For a more complete description, see section 3.

Macroeconomic models in HARK use the `Market` class to represent a market (or other aggregator) that combines the actions, states, and/or shocks (generally, outcomes) of individual agents in the model into aggregate outcomes that are "passed back" to the agents. For example, the market in a consumption-saving model might combine the individual asset holdings of all agents in the market to generate aggregate capital in the economy, yielding the interest rate on assets (as the marginal product of capital); the individual agents then learn the aggregate capital level and interest rate, conditioning their next action on this information. Objects that microeconomic agents treat as exogenous when solving (or simulating) their model are thus endogenous at the macroeconomic level. Like `AgentType`, the `Market` class also has a `solve` method, which seeks out a dynamic general equilibrium: a "rule" governing the dynamic evolution of macroeconomic objects such that if agents

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

[5] That is, as the foundational, building-block elements of HARK, new tools are not difficult to program and do not require extensive integration with many moving parts.

believe this rule and act accordingly, then their collective actions generate a sequence of macroeconomic actions that justify the belief in that rule. For a more complete description, see section 4.

Beyond the model frameworks, `HARKcore` also defines a "supersuperclass" called `HARKobject`. When solving a dynamic microeconomic model with an infinite horizon (or searching for a dynamic general equilibrium), it is often required to consider whether two solutions are sufficiently close to each other to warrant stopping the process (i.e. approximate convergence). It is thus necessary to calculate the "distance" between two solutions, so HARK specifies that classes should have a `distance` method that takes a single input and returns a non-negative value representing the (generally dimensionless) distance between the object in question and the input to the method. As a convenient default, `HARKobject` provides a "universal distance metric" that should be useful in many contexts.[6] When defining a new subclass of `HARKobject`, the user simply defines the attribute `distance_criteria` as a list of strings naming the attributes of the class that should be compared when calculating the distance between two instances of that class. For example, the class `ConsumerSolution` has `distance_criteria = ['cFunc']`, indicating that only the consumption function attribute of the solution matters when comparing the distance between two instances of `ConsumerSolution`.

2.2 HARKutilities

The `HARKutilities` module carries a double meaning in its name, as it contains both utility functions (and their derivatives, inverses, and combinations thereof) in the economic modeling sense as well as utilities in the sense of general tools. Utility functions included at this time are constant relative risk aversion and constant absolute risk aversion. Other functions in `HARKutilities` include some data manipulation tools (e.g. for calculating an average of data conditional on being within a percentile range of different data), functions for constructing discrete state space grids, convenience functions for retrieving information about functions, and basic plotting tools using `matplotlib.pyplot`.

The module also includes functions for constructing discrete approximations to continuous distributions (e.g. `approxLognormal()` 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). As a convention in HARK, continuous distributions are reframed as finite discrete distributions when solving models; an N-dimensional random variable is formatted as a length $N+1$ list of 1D arrays, with the first element representing event probabilities and all other elements are realizations of the N component RVs. This both simplifies solution methods (reducing numeric integrals to simple dot products) and allows users to easily test

[6] Roughly speaking, the universal distance metric is a recursive supnorm, returning the largest distance between two instances, among attributes named in `distance_criteria`. Those attributes might be complex objects themselves rather than real numbers, generating a recursive call to the universal distance metric.

whether their chosen degree of discretization yields a sufficient approximation to the full distribution. See LINK TO SPHINX for full documentation.

2.3 HARKinterpolation

The `HARKinterpolation` module defines classes for representing interpolated function approximations. Interpolation methods in HARK all inherit from a superclass such as `HARKinterpolator1D` or `HARKinterpolator2D`, wrapper classes that ensures interoperability across interpolation methods. For example, `HARKinterpolator1D` specifies the methods `__call__` and `derivative` to accept an arbitrary array as an input and return an identically shaped array with the interpolated function evaluated at the values in the array or its first derivative, respectively. However, these methods do little on their own, merely reshaping arrays and referring to the `_evaluate` and `_der` methods, which are *not actually defined in* `HARKinterpolator1D`. Each subclass of `HARKinterpolator1D` specifies their own implementation of `_evaluate` and `_der` particular to that interpolation method, accepting and returning only 1D arrays. In this way, subclasses of `HARKinterpolator1D` are easily interchangeable with each other, as all methods that the user interacts with are identical, varying only by "internal" methods.

When evaluating a stopping criterion for an infinite horizon problem, it is often necessary to know the "distance" between functions generated by successive iterations of a solution procedure. To this end, each interpolator class in HARK must define a `distance` method that takes as an input another instance of the same class and returns a non-negative real number representing the "distance" between the two. As each of the `HARKinterpolatorXD` classes inherits from `HARKobject`, all interpolator classes have the default "universal" distance method; the user must simply list the names of the relevant attributes in the attribute `distance_criteria` of the class.

Interpolation methods currently implemented in HARK include (multi)linear interpolation up to 4D, 1D cubic spline interpolation, (multi)linear interpolation over 1D interpolations (up to 4D total), (multi)linear interpolation over 2D interpolations (up to 4D total), linear interpolation over 3D interpolations, 2D curvilinear interpolation over irregular grids, and a 1D "lower envelope" interpolator. See LINK TO SPHINX for full documentation.

2.4 HARKsimulation

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. As an example of their use, these tools are used in the consumption-saving models of `ConsIndShockModel.py` to simulate permanent and transitory income shocks as well as unemployment events. See LINK TO SPHINX for full documentation.

2.5 HARKestimation

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 included in `scipy.optimize`. Future functionality will include more robust global search methods, including genetic algorithms, simulated annealing, and differential evolution. See LINK TO SPHINX for full documentation.

2.6 HARKparallel

By default, processes in Python are single-threaded, using only a single CPU core. The `HARKparallel` module provides basic tools for using multiple CPU cores simultaneously, with minimal effort.[7] In particular, it provides the function `multiThreadCommands`, which takes two arguments: a list of `AgentType`s and a list of commands as strings; each command should be a method of the `AgentType`s. The function simply distributes the `AgentType`s across threads on different cores and executes each command in order, returning no output (the `AgentType`s themselves are changed by running the commands). Equivalent results would be achieved by simply looping over each type and running each method in the list. Indeed, `HARKparallel` also has a function called `multiThreadCommandsFake` that does just that, with identical syntax to `multiThreadCommands`; multithreading in HARK can thus be easily turned on and off.[8] The module also has functions for a parallel implementation of the Nelder-Mead simplex algorithm, as described in Wiswall and Lee (2011). See LINK TO SPHINX for full documentation.

3 Microeconomics: the AgentType Class

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;[9]

[7] `HARKparallel` uses two packages that aren't included in the default distribution of Anaconda: `joblib` and `dill`; see step 3 of the instructions in section 1.1 for how to install them.

[8] In the future, `HARKparallel` might be absorbed into `HARKcore` and `HARKestimation`, particularly if `joblib` and `dill` become part of the standard Anaconda distribution.

[9] 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.

3.1 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:

- `solveOnePeriod`: 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 `solveOnePeriod` 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_next`). 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 `solveOnePeriod` 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 `'solveOnePeriod'` is an element of `time_vary`.[10]

- `solution_terminal`: An object representing 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 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 repeats 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.

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

- `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 named 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).

3.2 A Universal Solver

When an instance of `AgentType` invokes its `solve` method, 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 `solveOnePeriod`. The time-varying inputs are updated each period, including the successive period's solution as `solution_next`; the same values of time invariant inputs in `time_inv` are passed to the solver in every period. The first call to `solveOnePeriod` uses `solution_terminal` as `solution_next`. 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 `solveOnePeriod` 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. Each solution 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 or value function in the solutions. If the solution class is defined as a subclass of `HARKobject`, it automatically inherits the default `distance` method, so that the user must only list the relevant object attributes in `distance_criteria`.

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.

3.3 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.
- `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. One final method bears mentioning: the `__call__()` method points to `assignParameters()`, a convenience method for adding or adjusting attributes (inherited from `HARKobject`). 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.

ADD PARAGRAPH ABOUT PRESOLVE AND POSTSOLVE

3.4 Sample Model: Perfect Foresight Consumption-Saving

To provide a concrete example of how the `AgentType` class works, consider the very simple case of a perfect foresight consumption-saving model. The agent has time separably additive CRRA preferences over consumption, discounting future utility at a constant rate; he receives a particular stream of labor income each period, and knows the interest rate on assets that he holds from one period to the next. His decision about how much to consume in a particular period can be expressed in Bellman form as:

$$
\begin{aligned}
V_t(M_t) &= \max_{C_t} \mathrm{u}(C_t) + \beta \not{\mathrm{D}}_t \mathbb{E}[V_{t+1}(M_{t+1})], \\
A_t &= M_t - C_t, \\
M_{t+1} &= \mathsf{R} A_t + Y_{t+1}, \\
Y_{t+1} &= \Gamma_{t+1} Y_t, \\
\mathrm{u}(C) &= \frac{C^{1-\rho}}{1-\rho}.
\end{aligned}
$$

An agent's problem is thus characterized by values of ρ, R, and β plus sequences of survival probabilities $\not{\mathrm{D}}_t$ and income growth factors Γ_t for $t = 0, \cdots, T$. This problem has an analytical solution for both the value function and the consumption function.

The `ConsIndShockModel` module defines the class `ConsumerType` as a subclass of `AgentType` and provides solver functions for several variations of a consumption-saving model, including the perfect foresight problem.[11] A HARK user could specify and solve a ten period perfect foresight model with the following commands:

```
MyConsumer = ConsumerType(time_flow=True, cycles=1)
MyConsumer.CRRA = 2.7
MyConsumer.Rfree = 1.03
MyConsumer.DiscFac = 0.98
MyConsumer.LivPrb = [0.99,0.98,0.97,0.96,0.95,0.94,0.93,0.92,0.91,0.90]
MyConsumer.PermGroFac = [1.01,1.01,1.01,1.01,1.01,1.02,1.02,1.02,1.02,1.02]
MyConsumer.solveOnePeriod = solvePerfForesight
MyConsumer.solve()
```

The first line makes a new instance of `ConsumerType`, specifying that time is currently "flowing" forward[12] and that the sequence of periods happens exactly once. The next five lines set the time invariant (`CRRA`, `Rfree`, `DiscFac`) and time varying parameters (`LivPrb`, `PermGroFac`), and the following one sets the appropriate one period solver.[13] After running the `solve` method, `MyConsumer` will have an attribute called `solution`, which will be a list with eleven `ConsumerSolution` objects, representing the period-by-period solution to the model.[14] Each `ConsumerSolution` has several attributes:

- `cFunc`: Optimal consumption function for this period as a function of (normalized) market resources $m_t = M_t/Y_t$. Can be called like any other function: `MyConsumer.solution[0].cFunc(5)` evaluates the consumption function at $m_t = 5$.

[11] As the name implies, the module is mostly concerned with models with idiosyncratic shocks to income.

[12] This is relevant for the intepretation of the time-varying inputs `LivPrb` and `PermGroFac`.

[13] By default, a new instance of `ConsumerType` has a solver for a model with risky income.

[14] The last element of `solution` represents the terminal period solution, where the agent consumes all available resources because there is no future.

- `vFunc`: Value function (over market resources m_t); can be evaluated like `cFunc`.
- `mNrmMin`: Minimum value of normalized market resources m_t such that `cFunc` and `vFunc` are defined.
- `hNrm`: Normalized human wealth– the PDV of future income (ignoring mortality) divided by current period income Y_t.
- `MPC`: The constant marginal propensity to consume (linear consumption function).

To solve a version of this problem in which the sequence of ten periods happens three times (yielding a `solution` attribute with thirty-one elements), the user can do:

```
MyConsumer(cycles = 3)
MyConsumer.solve()
```

To solve an infinite horizon problem in which the agent experiences the same one period problem indefinitely, the user can do:

```
OtherConsumer = ConsumerType(cycles=0, CRRA=3.5, Rfree=1.02, DiscFac=0.95,
  LivPrb = [0.99], PermGroFac = [1.01], solveOnePeriod = solvePerfForesight
OtherConsumer.solve()
```

This instance is specified as being infinite horizon by setting `cycles=0`. Note that the time-varying inputs are still specified as (one element) lists, even though they take on the same value in every period; this is because an infinite horizon model might consist of a multi-period sequence repeated indefinitely, rather than just one repeated period. The `solution` attribute of `OtherConsumer` will be a list containing one instance of `ConsumerSolution`, representing the solution to every period of the model.

4 Macroeconomics: the Market Class

The modeling framework of `AgentType` is deemed "microeconomic" because it pertains only to the dynamic optimization problem of agents, treating all inputs of the problem as exogenously fixed. In what we label as "macroeconomic" models, some of the inputs for the microeconomic models are endogenously determined by the collective states and controls of agents in the model. In a dynamic general equilibrium, there must be consistency between agents' beliefs about these macroeconomic objects, their individual behavior, and the realizations of the macroeconomic objects that result from individual choices.

The `Market` class in `HARKcore` provides a framework for such macroeconomic models, with a `solve` method that searches for a dynamic general equilibrium. An instance of `Market` includes a list of `AgentType`s that compose the economy, a methods converting microeconomic outcomes (states, controls, and/or shocks) into macroeconomic outcomes, and a method for converting a history or sequence of macroeconomic outcomes into a

new "dynamic rule" for agents to believe. Agents treat the dynamic rule as an input to their microeconomic problem, conditioning their optimal policy functions on it. A dynamic general equilibrium is a fixed point dynamic rule: when agents act optimally while believing the equilibrium rule, their individual actions generate a macroeconomic history consistent with the equilibrium rule.

4.1 Down on the Farm

The `Market` class uses a farming metaphor to conceptualize the process for generating a history of macroeconomic outcomes in a model. Suppose all `AgentTypes` in the economy believe in some dynamic rule (i.e. the rule is stored as attributes of each `AgentType`, which directly or indirectly enters their dynamic optimization problem), and that they have each found the solution to their microeconomic model using their `solve` method. Further, the macroeconomic and microeconomic states have been reset to some initial orientation.

To generate a history of macroeconomic outcomes, the `Market` repeatedly loops over the following steps a set number of times:

1. `sow`: Distribute the macroeconomic state variables to all `AgentType`s in the market.
2. `cultivate`: Each `AgentType` executes their `marketAction` method, likely corresponding to simulating one period of the microeconomic model.
3. `reap`: Microeconomic outcomes are gathered from each `AgentType` in the market.
4. `mill`: Data gathered by `reap` is processed into new macroeconomic states according to some "aggregate market process".
5. `store`: Relevant macroeconomic states are added to a running history of outcomes.

This procedure is conducted by the `makeHistory` method of `Market` as a subroutine of its `solve` method. After making histories of the relevant macroeconomic variables, the market then executes its `calcDynamics` function with the macroeconomic history as inputs, generating a new dynamic rule to distribute to the `AgentType`s in the market. The process then begins again, with the agents solving their updated microeconomic models given the new dynamic rule; the `solve` loop continues until the "distance" between successive dynamic rules is sufficiently small.

4.2 Attributes of a Market

To specify a complete instance of `Market`, the user should give it the following attributes:[15]

[15] For some purposes, it might be useful to specify a subclass of `Market`, defining `millRule` and/or `calcDynamics` as methods rather than functions.

- `agents`: A list of `AgentType`s, representing the agents in the market. Each element in `agents` represents on *ex ante* heterogeneous type; each type could have many *ex post* heterogeneous agents.

- `sow_vars`: A list of strings naming variables that output from the aggregate market process, representing the macroeconomic outcomes. These variables will be distributed to the textttagents in the `sow` step.

- `reap_vars`: A list of strings naming variables to be collected from the `agents` in the `reap` step, to be used as inputs for the aggregate market process.

- `const_vars`: A list of strings naming variables used by the aggregate market process but *do not* come from `agents`; they are constant or come from the `Market` itself.

- `track_vars`: A list of strings naming variables generated by the aggregate market process that should be tracked as a history, to be used when calculating a new dynamic rule. Usually a subset of `sow_vars`.

- `dyn_vars`: A list of strings naming the variables that constitute a dynamic rule. These will be stored as attributes of the `agents` whenever a new rule is calculated.

- `millRule`: A function for the "aggregate market process", transforming microeconomic outcomes into macroeconomic outcomes. Its inputs are named in `reap_vars` and `const_vars`, and it returns a single object with attributes named in `sow_vars` and/or `track_vars`. Can be defined as a method of a subclass of `Market`.

- `calcDynamics`: A function that generates a new dynamic rule from a history of macroeconomic outcomes. Its inputs are named in `track_vars`, and it returns a single object with attributes named in `dyn_vars`.

- `act_T`: The number of times that the `makeHistory` method should execute the "farming loop" when generating a new macroeconomic history.

- `tolerance`: The minimum acceptable "distance" between successive dynamic rules produced by `calcDynamics` to constitute a sufficiently converged solution.

Further, each `AgentType` in `agents` must have two methods not necessary for microeconomic models; neither takes any input (except `self`):

- `marketAction`: The microeconomic process to be run in the `cultivate` step. Likely uses the new macroeconomic outcomes named in `sow_vars`; should store new values of relevant microeconomic outcomes in the attributes (of `self`) named in `reap_vars`.

- `reset`: Reset, initialize, or prepare for a new "farming loop" to generate a macroeconomic history. Might reset its internal random number generator, set initial state varibles, clear personal histories, etc.

When solving macroeconomic models in HARK, the user should also define classes to represent the output from the aggregate market process in `millRule` and for the model-specific dynamic rule. The latter should have a `distance` method to test for solution convergence; if the class inherits from `HARKobject`, the user need only list relevant attributes in `distance_criteria`.

4.3 Sample Model: FashionVictim

To illustrate the `Market` class, consider a simple example in the emerging economic subfield of aesthemetrics, the `FashionVictimModel`.[16][17] This module defines a subclass of `AgentType` called `FashionVictimType`. Each period, fashion victims make a binary choice of style s: to dress as a jock (0) or punk (1). They receive utility directly from the outfit they wear and as a function of the proportion of the population who *just wore* the same style; they also pay switching costs (c_{pj},c_{jp}) if they change styles rather than keep the same as the previous period. Moreover, they receive an idiosyncratic T1EV preference shock to each style in each period. Defining the population punk proportion as p and the conformity utility function as $f : [0, 1] \to \mathbb{R}$, the current period utility function is thus:

$$u(s_t; s_{t-1}, p_t) = s_t f(p_t) + (1-s_t) f(1-p_t) + s_t U_p + (1-s_t) U_j - c_{pj} s_{t-1}(1-s_t) - c_{jp}(1-s_{t-1}) s_t.$$

Fashion victims are forward looking and discount future utility at a constant rate of β per period. To simplify the analysis, we assume they believe that the population punk proportion in the next period is a linear function of the punk proportion in the current period, subject to a uniformly distributed shock. No restrictions are put on the function f; fashion victims might be conformists who like to dress the same as others ($f'(p) > 0$) or hipsters who like to style themselves in the minority ($f'(p) < 0$).[18] A fashion victim's problem can be written in Bellman form as:

$$V(s_{t-1}, p_t) = \mathbb{E}\left[\max_{s_t \in \{0,1\}} u(s_t; s_{t-1}, p_t) + \eta_{s_t} + \beta \mathbb{E}\left[V(s_t, p_{t+1})\right]\right],$$

$$p_{t+1} = a p_t + b + \pi_{t+1}, \qquad \pi_{t+1} \sim U[-w, w], \qquad \eta_0, \eta_1 \sim T1EV.$$

An instance of `FashionVictimType` is thus characterized by values of U_p, U_j, c_{pj}, c_{jp} and a function f, as well as beliefs about p_{t+1} as a function of p_t (summarized by slope a, intercept b, and uniform shock width w). Given this information, a `FashionVictimType`'s infinite horizon microeconomic model can be solved by backward induction in a few lines; the "one period solver" is given by `solveFashion`. However, while individual agents treat the dynamics of p_t as exogenous, they are in fact endogenously determined by the

[16] This model is inspired by the paper "The hipster effect: When anticonformists all look the same" by Jonathan Touboul.

[17] For a more traditional macroeconomic model, the `cstwMPC` module includes a consumption-saving model with both idiosyncratic and aggregate shocks, in which individual asset holdings are aggregated into total productive capital, endogenizing the (dynamic) interest and wage rate.

[18] In practice, f is parameterized as the beta distribution for convenience.

actions of all the fashion victims in the market.[19] A dynamic general equilibrium of the "macroeconomic fashion model" is thus characterized by a triple of (a, b, w) such that when fashion victims believe in this "punk evolution rule" and act optimally, their collective fashion choices exhibit this same rule when the model is simulated.

The search for a dynamic general equilibrium is implemented in HARK's `Market` class with the following definitions:

`sow_vars = ['pNow']` (macroeconomic outcome is p_t)
`reap_vars = ['sNow']` (microeconomic outcomes are s_t for many agents)
`track_vars = ['pNow']` (must track history of p_t)
`dyn_vars = ['pNextSlope','pNextIntercept','pNextWidth']` (dynamic rule (a, b, w))
`millRule = calcPunkProp` (aggregate process: average the style choices of all agents)
`calcDynamics = calcFashionEvoFunc` (calculate new (a, b, w) with autoregression of p_t)
`act_T = 1000` (simulate 1000 periods of the fashion market)
`tolerance = 0.01` (terminate solution when (a, b, w) changes by less than 0.01)

The `agents` attribute has a list of 22 `FashionVictimTypes`, which vary in their values of U_p and U_j, and their f functions. The `marketAction` method of `FashionVictimType` simulates one period of the microeconomic model: each agent receives style preference shocks η_0 and η_1, sees the current proportion of punks p_t (sown to them as `pNow`), and chooses which style to wear, storing it in the binary array `sNow`, an attribute of `self`.

The `millRule` for this market is extremely simple: it flattens the list of binary arrays of individual style choices (gathered in the `reap` step) and averages them into a new value of p_t, to be tracked as a history and `sown` back to the `agents` to begin the cycle again. Once a history of 1000 values of p_t has been generated with the `makeHistory` method, we can calculate a new dynamic fashion rule with `calcFashionEvoFunc` by regressing p_t on p_{t-1}, approximating w as twice the standard deviation of prediction errors.[20] The new fashion rule is an instance of the simple FashionEvoFunc class, whose only methods are inherited from `HARKobject`.

When the `solve` method is run, the solver successively solves each agent's microeconomic problem, runs the `makeHistory` method to generate a 1000 period history of p_t, and calculates a new punk evolution rule based on this history; the solver terminates when consecutive rules differ by less than 0.01 in any dimension.

5 Contributing to HARK

blah blah

[19] The market might consist of agents all of the same type, or might have several types with *ex ante* heterogeneity.

[20] This is an arbitrary way to calculate w, but accuracy is not important in a silly example.

5.1 What Does HARK Want?

blah blah

5.2 Git and GitHub

blah blah

5.3 Submitting Code

blah blah

5.4 Naming Conventions

blah blah

5.5 Documentation Conventions

blah blah

6 Future of HARK

blah blah

6.1 Future Tools

blah blah

6.2 Future Models

blah blah

6.3 Bounty Hunting

blah blah

6.4 All Aboard the Ark

blah blah