Ferrall, Christopher

Working Paper

Object oriented (dynamic) programming: Replication, innovation and "structural" estimation

Queen's Economics Department Working Paper, No. 1432

Provided in Cooperation with:
Queen's University, Department of Economics (QED)

Suggested Citation: Ferrall, Christopher (2020) : Object oriented (dynamic) programming: Replication, innovation and "structural" estimation, Queen's Economics Department Working Paper, No. 1432, Queen's University, Department of Economics, Kingston (Ontario)

This Version is available at:
https://hdl.handle.net/10419/230585

Queen's Economics Department Working Paper No. 1432

Object Oriented (Dynamic) Programming: Replication, Innovation and "Structural" Estimation

Christopher Ferrall

Department of Economics
Queen's University
94 University Avenue
Kingston, Ontario, Canada
K7L 3N6

6-2020

Object Oriented (Dynamic) Programming: Replication, Innovation and "Structural" Estimation

Christopher Ferrall
Queen's University, Kingston, Canada
ferrallc@queensu.ca

June, 2020 [Current]

Abstract

This paper discusses how to design, solve and estimate dynamic programming models using the open source package `niqlow`. Reasons are given for why such a package has not appeared earlier and why the object-oriented approach followed by `niqlow` seems essential. An example is followed that starts with basic coding then expands the model and applies different solution methods to finally estimate parameters from data. Using `niqlow` to organize the empirical DP literature may support new research better than traditional surveys. Replication of results in several published papers validate `niqlow`, but it also raises doubt that complex models solved with purpose-built code can ever be independently verified.

Keywords: Dynamic Programming, Computational Methods, Replication Studies

1. Introduction

Since the early 1980s fields such as macro, labor, and industrial organization have estimated discrete choice, discrete time, dynamic programs.[1] A barrier to empirical DP is the need to write computer code from scratch without benefit of tools tailored to the task. When such computing tools emerge they ease verification, replication and innovation in the area. For example, in applied econometrics, widespread adoption of Stata and R has replaced low-level programming with high-level scripts that are portable and easy to adapt. Dynare plays a similar role solving dynamic stochastic general equilibrium (DSGE) models based on Matlab.

Why has empirical DP not benefited from development of a similar platform, even as applications and new solutions methods continue to be published? Although researchers shared their code nearly all published empirical DP work uses purpose-built code. Rust's distribution of Gauss code for Rust (1987) in the 1990s is the closest attempt to create a common platform. There are good reasons to doubt a platform for empirical DP is feasible. The models are complex, the details vary greatly across fields, and their use involves multiple layers of computation (nested algorithms). Perhaps all they have in common are tools provided by mathematical languages, such as matrix algebra, simulation, and numerical optimization.

Does no empirical DP platform exist because it is impossible? Or, is it possible, but for some reason a common platform has not emerged from purpose-built code? This paper introduces the software package `niqlow` to support the latter explanation. The package's design, and how it differs from purpose-built code, suggests why in nearly forty years no platform for empirical DP emerged.

`niqlow` replaces low-level purpose-built coding with high level tools to design, build and estimate empirical DP models. To demonstrate this claim a simple model is defined and coded with essentially a one-to-one correspondence between the mathematical elements and coding statements. The model is then extended and estimated from data without rewriting code or programming any standard aspects of empirical DP directly.

Starting with Eckstein and Wolpin (1989) and continuing through at least Keane et al. (2011), reviews of the empirical DP literature have attempted to standardize notation with no reference to computing. Because the math does not map directly to code, these frameworks offer limited help to someone developing their own first model. Using `niqlow` concepts as an intermediate translation between math and working code creates a new way to describe and organize the DP literature. When compared to starting from scratch or "hacking" existing code, new empirical DPs are easier to develop and verify using standardized concepts.

For example, many published models contain a binary action a and a state variable s whose transition is $s' = s + a$. That is, s counts how many times $a = 1$ has been chosen in the past. Until now all researchers code s independently from scratch. In `niqlow`, however, s is simply an "action counter" that can be added to any DP model with one or two user-written statements. Solution algorithms are built into `niqlow`, and two or more methods to solve a model can be compared without changing any low-level coding. Previously this required duplicating code specific to each method.

A standard section of most structural estimation articles derives the estimation objective step by step. This leaves the impression that the econometrics is specific to the model and the data, which is indeed the case when using purpose-built code. But customized econometric objectives is not a deep feature of other models. For example, a Stata user need not provide a function that returns the log-likelihood for their panel-probit data set. Stata can compute it using details provided by the user. Automation of econometric objectives for empirical DP also emerges in `niqlow`.

The second half of the paper conducts the first replication study of empirical DP results. Half a dozen papers published before 2000 are targeted. The replications help validate `niqlow` and build out its capabilities. The exercise also makes suggests that most published empirical DP results are essentially impossible to verify independently and will remain so until at least one common platform emerges.

`niqlow` uses the object-oriented programming (OOP) paradigm to provide menus for state variables, solution methods, and econometric calculations. This paper briefly explains OOP and why it seems fundamental to creating a platform for empirical DP. It also proffers an answer to why such a platform emerged so late compared to other areas of applied economics.

To promote collaborative development, `niqlow` is open source software housed on `github.com` under a GPL License. New solution methods and replications can be added to the platform. Solution methods can now be compared on different models not simply those chosen by authors proposing a new approach.[2]

1.1 A Tale of Two Papers

The divergence over time in the toolboxes available for empirical work can be traced starting from two early "structural" estimation papers: MaCurdy (1981) and Wolpin (1984). The former estimates a lifecycle labor

supply model on panel data using an approximation to the structural model. That is, the Lagrange multiplier on a lifecycle budget constraint in the MacCurdy model has a closed form in the estimated specification. That form could be imposed while estimating other parameters, but it would have to be computed on each iteration of the econometric objective. Instead, MaCurdy (1984) approximates the multiplier as a function of constant characteristics of the person This approximate model can be estimated using two-stage least squares and instrumental variables.

The latter paper, Wolpin (1984), estimates a lifecyle model of fertility on panel data using maximum likelihood. Its abstract defines the approach as a nested solution algorithm that imposes all restrictions of the model on the estimated parameters.[3] Although many things have changed since then, many if not most empirical DP papers follow the same basic strategy.

Given resources available at the time, both MaCurdy (1981) and Wolpin (1984) required extensive original programming and significant computations. Fast-forward to today and MaCurdy's procedure has been reduced to a single line of Stata code, roughly:

```
. xtivreg lnh `xvars' (lnw = exper exper2 L2.wage), first fd
```

That is, the procedure was a panel IV regression on first difference of log hours using experience and lagged wages as instruments for current wages. Stata's syntax allows the list of exogenous variables `` `xvars' `` to be defined elsewhere. On the other hand, nearly all empirical DP models continue to require purpose-built programs for *the* model. Any change requires re-coding. Certainly a single line in a Stata script cannot define an empirical DP model.

There are many reasons why panel IV estimation would have received more attention from software developers than maximum likelihood estimation of discrete choice dynamic programming models. The puzzle is not the difference in relative attention. Instead it is the absolute level of development: since Wolpin's purpose-built code there has been essentially zero infrastructure developed for empirical DP models. Something has blocked progress in the empirical DP toolbox that did not block IV panel regression code from evolving into single commands in popular packages.

Two claims are made here about this block in developing a platform. First, object-oriented programming (OOP) appears essential for removing barriers to a more general economics toolkit. This claim is based on `niqlow` and the absence of a non-OOP alternative to it. Without shifting to OOP code there was no way to avoid custom coding empirical DP. Second, an explanation is given for why empirical DP did not shift to OOP until `niqlow`. It is discussed in Section (6.5) after demonstrating the `niqlow` approach.

2. OOP VERSUS PP

Consider the task of creating a computing platform to be used by others to solve their own problems. Call the original coder the *programmer* and the one using the platform the *user*. OOP can be compared to the more straightforward procedural programming (PP) approach that has produced most published empirical DP results, in which the programmer and user are essentially the same person or team. Since this paper argues computer programming paradigms have affected the development of economic research, the two relevant paradigms are briefly described here. Readers familiar with OOP can skip this section.[4]

The difference between PP and OOP is how data are stored and processed. In procedural programming (PP) data stored in vectors or other structures are passed to *procedures* (aka functions or subroutines) to do the work.[5] The procedure sends the results back to the program through a return value or arguments of the procedure. The programmer of a platform would write functions that the user would call in their own program sending their data to the built-in functions.

OOP directly connects (binds) data and the procedures to process them. It does this by putting them together in a *class*. This brings new syntax and jargon. A class is a template from which *objects* are created during execution of the program. A class and objects created from it have variables (*members*) and functions (*methods*) that process the data stored in the class members.

OOP has three key features that are difficult to code using PP alone. First, a class can be *derived* from a base class while adding or modifying components. In other words, a class can *inherit* features from a parent class. The programmer may define child classes from a derived parent to handle different situations the user may confront. Each child in turn might have derived grandchildren. The user can also create their own derived class that inherit only the features of the ancestor classes. Inheritance is a *downstream* connection between classes created by the programmer for the user.

Second, all objects of the same class can share member data and methods while having their own copies of other members as designed by the programmer. Shared members are sometimes called *static* because additional storage for them is not created dynamically as objects are created during execution. Suppose a user's program creates one thousand objects of a class. All of them need access to a common parameter q. Then q can be stored in a static member of the class shared by all. If another value x is specific to each object then it is in a dynamic member and each object will have its own version of x. Static or shared members is a *horizontal* connection between objects while a program executes.

Third, there is an *upstream* connection between classes. In OOP jargon this is called a *virtual method*. Suppose the parent class marks `profit()` a virtual method. As with all ancestor features, a child class can access `profit()`. However, the child can define its own method `profit()`. Since the parent class labels the method virtual it allows the child version to replace the parent version when used by other parent code. That is, the programmer has given the user the option (or the requirement) to inject their own code into the base code. If `profit()` is not virtual then the user can still create their own version but it will not replace the parent version inside the parent code.

These downstream, horizontal, and upstream connections between data and the functions that process them can be implemented without using objects in procedural programming. However, as argued above, for more than 30 years almost no PP platform for empirical DP has been attempted, and none have succeeded in freeing users from writing the basic functions themselves. This suggests the complex environment of empirical DP requires OOP.

2.1 An Example

To illustrate differences between OOP and PP, consider a package written by a programmer to be used by economists (users): first using the PP paradigm only and then using OOP. The package, named `Marshall`, solves for Marshallian demand for a consumer with utility $U(x)$ defined on a vector x and a given price vector p and income m :

$$x^\star(p,m;U) \quad \equiv \quad \arg\max_{x:px\leq m} \quad U(x).$$

Most readers have probably coded an objective and then called a built-in optimization procedure to optimize that function. `Marshall` is a specialized version of that general problem.

The PP package documentation explains how users should code $U()$ in order to interact with tools in the package. The user codes `u(x)`, and sends it to a built-in procedure of the form `demand(u,p,m)`. That procedure uses algorithms to compute $x^\star(p,m)$. Suppose the user wants to use the Cobb-Douglas function, $U(x) = \sum \ln x_i$. Using "pseudo-code" the key parts of the user's program might look like:

```
#uses Marshall
u(x) {
    return sum_i ln(x[i])
    }
qdemand = demand(u,prices,income)
print("x* = ",qdemand)
```

Now consider the OOP version of Marshall. The programmer might define a class for a consumer:

```
class Consumer {
    members
                xstar, p, m
    methods
                demand()
                budget(p,m)
        virtual u(x)
    }
```

The syntax is pseudo code similar to actual OOP languages, including Ox. The method `u(x)` belongs to the `Consumer` class and does the work of computing utility. The budget parameters are stored as members of the class. These will be set by passing them to the `budget()` method. The method `demand()` is the same as the PP procedure above, but it will get the information it needs from the data members rather than from arguments. It stores the result in the member `xstar`.

The package comes with the Cobb-Douglas function set as the a default to demonstrate the package without any coding. Now $u()$ is coded as a method of the `Consumer` class:

```
Consumer::u(x) {
    return sum( log(x) )
    }
```

Unlike an ordinary function, the code for `u()` has the prefix of the class it belongs to. Code for the other methods would also be part of the package. User code to create a `Consumer` object, set the budget to already-defined values, and solve for $x^\star$ might look like this:

```
#uses Marshall
    ⋮
agent = new Consumer()
agent -> budget(prices,income)
agent -> demand()
print("x* = ",agent.xstar )
```

Here the `new` operator will make a copy of the template for `Consumer` in memory and store it in the variable `agent`. The code above would use the built-in utility and compute quantity demanded at the prices and income sent to the budget operator. The syntax `object -> method()` is a common way to invoke a method for a particular object. That is, instead of sending u to `demand()` in the PP approach, the data specific to `agent` is automatically available to the `demand()` method belonging to `agent`.

The code so far uses a built-in utility. Suppose the user wants to use a CES function. To do so, the user creates a class derived from `Consumer`.

```
class CES : Consumer {
    members  a
    methods  u(x) CES(ina)
    }
CES::CES(ina) {     a = ina    }
CES::u(x)     {     return sum_i (x[i]^a)^(1/a)    }
⋮
agent = new CES(0.2)
agent -> demand()
⋮
```

The first line shows that `CES` is a child of `Consumer`, whereas `Consumer` was a base class not derived from something else. Tjee new class does not declare its own `demand()` method, `CES` inherits the version from `Consumer`. The user also provides a method that is called to cretae a new object. As with Ox, this *constructor* has the same name as the class in the pseudo code. The parameter is passed to it and stored in `CES.a`, ready to be used by utility.

When `demand()` is invoked, the user-provided $u()$ will be called *not* the default version written by the programmer even though the user has not changed and perhaps cannot even see the code for `demand()`. This is because `Consumer::u()` was marked as `virtual`. By declaring `u()` virtual the programmer gives the user a controlled ability to change the underlying code.

In the PP packaged this injection of code was accomplished by passing `u` to the demand function. However, when many functions need to be replaced and many different consumer problems are solved by the user's program the PP approach can become unwieldily. The OOP approach has fixed costs of setup, but it scales more efficiently for both the programmer and the user as the problem increases in complexity. It is easier to ensure the right data and the right functions are being used within the package.

The programmer can create a taxonomy of classes for the user to choose from. In this simple case, `Marshall` might not just have a single `Consumer` class. It might have child classes for different classes of utility. The user can then start with one of those classes to either specialize or extend it for their model. An OOP package can provides the user with a structured menu of options, which is an important part of the `niqlow` approach to DP.

3. Empirical Dynamic Programming

This section defines key elements of dynamic programming that appear in empirical applications. This framework is then combined with a simple example implemented in `niqlow` before extending it to account for multiple problems and parameter estimation.

3.1 A Single Agent Problem

3.1.1 The Primitives

The symbols used to define a single generic DP model and explained in this section are in order:

$$\theta \in \Theta \qquad \alpha \in A(\theta) \qquad P(\theta';\alpha,\theta) \qquad U(\alpha,\theta) \qquad \delta \qquad \zeta \qquad \psi. \tag{1}$$

The first element is the *state* θ, a vector of state variables: $\theta = (\,s_0 \quad \ldots \quad s_N\,)$. A state is an element of the *state space* Θ. Second, at each state an *action* α is chosen, a vector of action variables: $\alpha = (\,a_0 \quad \ldots \quad a_M\,)$. The action is chosen from the *feasible choice set* $A(\theta)$.[6] Third, the next state encountered in the program, denoted θ', follows a semi-Markov *transition* that depends on the current state and action, $P(\theta';\alpha,\theta)$.

When making decisions at θ, the agent's objective involves the one-period payoff or utility $U(\alpha,\theta)$. In `niqlow` it is treated as a vector-valued function of the feasible action set, so it will be written $U(A(\theta),\theta)$. The objective is additive in values of possible states next period discounted by δ. The values of actions include a shock ζ_α contained in the vector ζ. These shocks often appear in empirical DP to smooth the solution.[7]

Finally, parameters that determine the other primitives are collected in the structural parameter vector ψ. All the other primitives listed above are implicit functions of ψ. When the empirical DP includes agents solving different problems, exogenous (demographic) data define different problems and they also interact with ψ. The roles of parameters and data are made explicit later.

3.1.2 Bellman's Equation

The *value* of an action takes the form[8]

$$v_\zeta(\alpha,\theta) = U(\alpha,\theta) + \zeta_\alpha + \delta E_{\alpha,\theta} V(\theta'). \tag{2}$$

The final term in the action value (2) is the endogenous expected value of future decisions. Optimal state-contingent choices and their value are defined as

$$\begin{aligned} \alpha^\star_\zeta(\theta) &= \arg\max v_\zeta(\alpha,\theta) \\ V_\zeta(\theta) &= \max_{\alpha\in A(\theta)} v_\zeta(\alpha,\theta) \\ V(\theta) &= \int_\zeta V_\zeta(\theta) f(\zeta) d\zeta. \end{aligned} \tag{3}$$

Value at θ integrates over optimal value conditional on ζ. For a model with no ζ the integral collapses to $V() \equiv V_\zeta()$.

The expected value of next period's state further sums over the transition of the discrete states:

$$E_{\alpha,\theta} V\left(\theta'\right) = \sum_{\theta' \in \Theta} V\left(\theta'\right) P\left(\theta'; \alpha, \theta\right). \tag{4}$$

Two assumptions about ζ are built into this expression. First, future shocks are built into $V(\theta')$ which is not affected directly by the current shock because ζ is IID over time. Second, the transitions of other state variables can be influenced by ζ only through the action α. These conditions form Rust's (1987) conditional independence (CI) property.

Bellman's equation, also known as the *Emax* operator, imposes the conditions (3) at all states simultaneously:

$$\forall\, \theta \in \Theta, \quad V(\theta) = \int_{\zeta} \left[\max_{\alpha \in A(\theta)} U\left(\alpha, \theta\right) + \zeta_{\alpha} + \delta E_{\alpha,\theta} V\left(\theta'\right)\right] f\left(\zeta\right) d\zeta. \tag{5}$$

3.1.3 Conditional Choice Probabilities: Three Flavors

The agent conditions choice on all available information, and $\alpha^{\star}$ in (3) is the set of feasible actions that maximize value at a state. This leads to the first notion of choice probability: from the agent's perspective. In particular, non-optimal choices have 0 probability of occurring. If the optimal choice is unique then the agent chooses it with probability 1. If ζ has a full-rank continuous distribution with unbounded support then there is zero probability of ties in action values and $\alpha^{\star}$ will be unique. To account for models with no ζ, and without loss of generality, the first flavor of choice probability assigns equal probability to all optimal actions:

$$\text{CCP1:} \qquad P_{\zeta}^{\star}\left(\alpha; \theta\right) = \frac{I\left\{\alpha \in \alpha_{\zeta}^{\star}(\theta)\right\}}{\#\alpha_{\zeta}^{\star}(\theta)}, \tag{6}$$

where $I\{\,\}$ is the indicator function and $\#B$ is the cardinality of a set B.

The choice probability in (6) is not continuous in the parameter vector ψ because it includes an indicator function. For example, suppose a small change in a parameter induces a small change in utility. This can shift an action α from optimal to not optimal and vice versa. $P_{\zeta}^{\star}\left(\alpha; \theta\right)$ jumps in value which in turn makes an econometric objective built on it discontinuous.

This issue is fixed by treating ζ as private to the agent. Now choice probabilities based on public information are continuous because they integrate over ζ, leading to the second notion of choice probability:

$$\text{CCP2:} \qquad P^{\star}\left(\alpha; \theta\right) = \int_{\zeta} P_{\zeta}^{\star}\left(\alpha; \theta\right) f(\zeta) d\zeta. \tag{7}$$

For example, when ζ is extreme value we get the familiar McFadden/Rust form of CCP:

$$P^{\star}\left(\alpha; \theta\right) = \frac{e^{v(\alpha,\theta)}}{\sum_{a \in A(\theta)} e^{v(a;\theta)}}. \tag{8}$$

The choice probability in (7) is relevant to the empirical researcher but not to the agent who conditions choice on ζ. (The integrated value $V(\theta')$ in (3) is relevant to the agent, because they also take an expectation of arriving at state θ' and then observing realized values of ζ'.)

If ζ is excluded from the model, CCPs may still need to be continuous in the parameter vector ψ. This leads to the third notion of conditional choice probability: *ex post* smoothing using a kernel ($K()$) over all feasible action vectors:[9]

$$\text{CCP3:} \quad P_K^{\star}(\alpha;\theta) = K[\, v(A(\theta);\theta)\,]. \tag{9}$$

CCP3 adds trembles to CCP1. Equations(7) and (9) differ because, in the former, the shocks enter the value function and affect the expected value of future states. In the later, the smoothing takes place separately from the value function. So a logistic kernel is the same functional form as the McFadden/Rust CCP in(8), but the values of the actions are not the same.

The three CCP flavors, un-smoothed as in(6), *ex-ante* smoothed as in (7), and *ex-post* smoothed in (9), are all part of `niqlow`. Within the smoothed classes, the functional or distributional form is a further part of the specification. `niqlow` provides options for standard functional forms and gives the user the possibility of adding alternatives.

Once solved, the DP model generates an endogenous state-to-state transition:

$$P(\theta';\theta) = \sum\nolimits_{\alpha\in A(\theta)} P^{\star}(\alpha;\theta)\, P(\theta';\alpha,\theta). \tag{10}$$

This transition sums over all feasible actions. Computing (10) is unnecessary in ordinary Bellman iteration, because an agent following the DP will make a choice at each θ they reach. (10) does play a role in some solution methods and predictions as discussed later in Section (5.4).

A final component of a single agent empirical DP model is the set of initial conditions from which data are generated. In non-stationary (lifecycle) models there are often natural initial values of state variables when the agent first makes a choice. If the environment is stationary then there may be a stationary or ergodic distribution over states, denoted $f_\infty(\theta)$. Treating the transition in (10) as a matrix, the stationary distribution satisfies

$$P(\theta';\theta)\, f_\infty(\theta) = f_\infty(\theta). \tag{11}$$

It is often assumed that data are drawn from the ergodic distribution as the initial condition for estimation or prediction.

3.2 Building a DP in `niqlow`

Empirical DPs have typically been solved using purpose-built programs with hard-coded loops to span the state space Θ of *the* model. Different tasks (model solution, prediction or simulation, etc.) use a different nests of loops that must be kept synchronized with the model's assumptions. Introducing another action or state variable requires re-coding and re-synching at the lowest level of the code.

A platform to build and solve *any* DP model cannot start with this code structure. In particular the hard-coding of the platform by the programmer cannot be model-specific. Instead, the work to build the state space, solve the model, and use it must be constructed from the user's code. The platform must offer standard choices and "plug-and-play" tools for building the model. Algorithm 1 summarizes how a user would use `niqlow` following this approach.

Algorithm 1. User Coding Steps in `niqlow`

A. *Declare* a new class (template) for the model

B. *Create* the DP

1. Initialize (call `Initialize()`)
2. Build the model (add action and state variables, etc.)
3. Create state and action spaces (call `CreateSpaces()`)

C. Code $U()$ and other functions related to the model.

D. *Solve* the DP

Apply a solution method to compute $V(\theta)$ and $P^{\star}(\alpha;\theta)$

E. *Use* the Solution

Simulate data, predict outcomes, estimate parameters, etc.

These steps solve a single agent DP once and use it somehow. Empirical DP almost always requires solving multiple problems multiple times using a nested solution method. In this case the *Use* and *Solve* steps above are intertwined (nested). How `niqlow` handles this is discussed in Section (6).

None of the steps in Algorithm 1 include low-level tasks such as: "Code loops to iterate on the value function and check for convergence." These tasks are done for the user based on the high-level elements their code provides. The top-level elements may appear in the code in a different order, but the numbered steps in part B must be executed in that order. Steps B.1 and B.3 each correspond to the specific named function in `niqlow`. The amount of coding that other elements in Algorithm 1 involve depends on the model and its purpose.

3.3 Example: Lifecycle Labor Supply

Consider a simple discrete choice lifecycle labor supply model. The agent lives 40 periods with the objective of maximizing discounted expected value of working ($m = 1$) or not ($m = 0$) each period. Earnings (E) come from a Mincer equation that is quadratic in actual experience M. Earnings are subject to a discrete IID shock e. Six equations describe the model:

$$
\begin{aligned}
\text{Objective:} \quad & E\sum_{t=0}^{39} 0.95^t \left[U\left(m_t; e_t, M_t\right) + \zeta_m\right] \\
\text{V shocks:} \quad & F(\zeta_m) = e^{e^{-\zeta_m}} \\
\text{Experience:} \quad & M_t = \sum_{s=0}^{t-1} m_s;\ m_0 = 0 \\
\text{E Shocks:} \quad & e_t \overset{iid}{\sim} dZ(15) \\
\text{Earnings:} \quad & E(M, e) = \exp\left\{\beta_0 + \beta_1 M + \beta_2 M^2 + \beta_3 e\right\} = \exp\{x\beta\} \\
\text{Utility:} \quad & U(m; e, M) = mE(M, e) + (1 - m)\pi.
\end{aligned}
\tag{12}
$$

The earnings shock follows a discretized standard normal distribution taking on 15 different values, hence the notation $dZ(15)$. The parameter π is the utility of not working. The coefficients in the earnings equation are elements of the vector β that includes the standard deviation of the earnings shocks, β_3.[10] The vector x in the earnings function is constructed at each state based on the current values of the state variables.

As a bridge to understanding `niqlow` code to implement (12), first translate the model into terms used in `niqlow`:

The Labor Supply Model Using `niqlow` Concepts

Element	**Value**	**Category**	**Params / Notes**
Clock	t	Ordinary Aging	T=40.
CCP	ζ	ExtremeValue	$\rho = 1$.
Actions	$\alpha = (m)$	Binary Choice	
States:	$\theta = (M)$	Action Counter	N=40
	$\epsilon = (e)$	Zvariable	N=15
Choice Set	$A(\theta) = \{0, 1\}$		for all θ
Utility	$U(\,) = \begin{pmatrix} \pi \\ E(\theta) \end{pmatrix}$		$E(\theta)$ defined in (12).

This intermediate translation of the math is a new way to summarize the DP literature and a strategy to reduce the fix cost of building new models. Using these essential elements we can write the code corresponding to each abstract step in Figure 2. The shock e could be placed in θ with M. Why it is efficient to place in ϵ is explained in Section (3.6.2).

A. Declare the template for θ

With line numbers added and Ox keywords in bold the code is:

[A]
```
    class LS : ExtremeValue {
1.      static decl m, M, e, beta, pi;
2.             Utility();
3.      static Build();
4.      static Create();
5.      static Earn();
    }
```

The class is named `LS` and is derived from the built-in class for extreme value shocks (`ExtremeValue`).[11] The declaration is not executed. Instead, it is the template for creating each point in the state space while the program executes (step B.2 of the algorithm). Most variables and methods needed by `LS` are already defined by its ancestor classes. Only details that `niqlow` cannot know ahead of time need to be added to `LS`. Line 1 of [A] declares a member for each element of the model, except t and δ, which are stored internally. All the new members are "static" as briefly explained in Section (2) as a horizontal connection between objects. An object of the `LS` class is created for each point in the state space Θ, but there is only one copy of the static members shared by each object.[12]

Create, Solve and Use

The code executed to solve the labor supply model appears in the class's methods as well as other parts of the user's program. Like C, an Ox program always contains a procedure named `main()` which is where execution begins. `LS` is designed so that the user's main procedure can be short:

[B]
```
#include "LS.ox"
     main() {
1.        LS::Create();
2.        VISolve();
3.        ComputePredictions();
         }
```

These three lines of code correspond to steps B-D in Algorithm 1. The sub-steps of B are combined into the `Create()` method. A user could put all the code inside `main()` rather than isolating some of it in `Create()`. Line 2 solves the value function and computes choice probabilities (step C). Solution methods are described in Section (4). Line 3 generates average values of all the variables based on the solution, as an example of using the solved model (step D).

B. Create the Model

The three sub-steps in part B are put together in `Create()`:

[C]
```
    LS::Create() {
1.      Initialize(1.0,new LS());
2.      Build();
3.      CreateSpaces ();
        }
```

Any DP model must include lines 1 and 3 of [C]. Items like state variables and actions must be added to the model in between those two statements. `Build()` contains the code specific to the labor supply model. As with

`main()`, `Create()` could contain more lines of code rather than placing them in `Build()`. The reason for this becomes apparent when the simple model is extended later on. The version of `Initialize()` and `CreateSpaces()` invoked by this code depends on which class `LS` was derived from.

B.2 Build the Model

In `niqlow` the action and state vectors are not hard-coded. They are built dynamically as the program executes by adding objects to a list. These tasks must happen in the Build sub-step of Algorithm 1. In the labor supply code they are placed in `Build()`:

[D]

```
   LS::Build() {
1.      SetClock(NormalAging,40);
2.      m = new BinaryChoice ("m");
3.      M = new ActionCounter ("M",40,m);
4.      e = new Zvariable ("e",15);
5.      Actions (m);
6.      EndogenousStates (M);
7.      ExogenousStates (e);
8.      SetDelta (0.95);
9.      beta = <1.2 ; 0.09 ; -0.1 ; 0.2>;
10.     pi = 2.0;
       }
```

Line 1 of [D] uses `SetClock()` to specify the model's clock using one of `niqlow`'s built-in clocks, `NormalAging`. The parameter for normal aging is the horizon T, here 40 years. If the user wanted to solve an infinite horizon model the code would simply change to `SetClock(InfiniteHorizon)`. These choices dictate how storage is created and how Bellman's equation is solved, but from the user's perspective it is simply a different choice of clock. Because model time t is essential it is an implicit element of θ and is stored internally.

Line 2 of [D] creates a binary action and stores it in `m` (declared a member of the `LS` class in [A]). Line 5 sends `m` to `Actions()` which adds it to the model. The two state variables are created on lines 3 and 4. Already mentioned in the introduction, `ActionCounter` is a built-in class derived from the base `StateVariable` class.[13] M needs to know which action variable it is tracking, so m is sent when the counter is created on line 3 along with the number of different values to track (from 0 to 39).

Line 4 creates the earnings shock as object of the `Zvariable` class. Creating state variable objects do not automatically add them to the model. Lines 6 and 7 do this. Since e is IID it can be placed in the ϵ vector by sending it to `ExgoenousStates()`, explained below. However, M is endogenous because its transition depends on its current value and the action m. It must be placed in θ by sending it to `EndogenousStates()`.

The last three lines of `Build()` set the value of model parameters. Because they do not affect the state space these statements can be placed elsewhere as long as they are executed before the model solution is started. Further, there is no need to formally define ψ and make these elements of it because this model is not going to be estimated. This will happen in the extension below. The discount factor δ is set by calling `SetDelta()` and is stored internally like the clock t was.

B.3 Creating Spaces

Line 3 in [C] sets up the model by creating the state space, the action spaces and other supporting structures. Selected output from the `niqlow` function is given in Figure 1. The summary echoes the model's class and its ancestors back to the `Bellman` class. It echoes the clock type and then list state variables and the number of values they take on. Note that t was added to θ by `SetClock()`, and it is the second right-most variable in θ. Several state variables are listed that take on 1 value and were not added to the model by the user code. These are placeholder variables for empty vectors explained below. Next, the report lists the size of the state space Θ and some other spaces. The difference values are discussed later.

Figure 1. `CreateSpaces()` Report for the Labor Supply Model

```
0. USER BELLMAN CLASS:    LS | Exteme Value  | Bellman
1. CLOCK:                 3. Normal Finite Horizon Aging
2. STATE VARIABLES
              |eps   |eta |theta -clock        |gamma
                e    s21      M      t     t''     r      f
       s.N     15      1     40     40      1      1      1
3. SIZE OF SPACES
                      Number of Points
    Exogenous(Epsilon)               15
    Endogenous(Theta)                40
    Times                            40
    EV()Iterating                    40
    ChoiceProb.track               1600
    Total Untrimmed               24000
5. TRIMMING AND SUBSAMPLING OF THE ENDOGENOUS STATE SPACE
                           N
    TotalReachable       820
         Terminal          0
     Approximated          0
```

C. Code Utility and Other Functions

Step C in Algorithm 1 says to code utility and other functions. This code is written before running the program and creating spaces, but utility is not involved in that step and only gets called once a solution method begins. The user's utility replaces a virtual utility called inside `niqlow` algorithms. There are several other virtual methods that the user might need to replace, and some examples appear below.

To match the model specification in (12), and in anticipation of empirical applications, earnings and utility are coded as separate functions:

[E]

```
    LS::Earn() {
       decl x;
1.     x = 1 ~ CV(M) ~ sqr(CV(M)) ~ AV(e);
2.     return exp( x*CV(beta) ) ;
       }
    LS::Utility() {
3.     return  CV(m)*(Earn()-pi) + pi;
       }
```

The Mincer equation has been written in matrix form as $E = \exp\{x\beta\}$. The expression for `x` on line 1 uses Ox-specific syntax to construct the vector. The presence of `CV()` and `AV()` in these expressions is specific to `niqlow` and is fundamental enough to merit some explanation.

If M were an ordinary counter taking on values between 0 and 39 then the statement `D=M;` sets `D` to the current value of `M`. However, `M` is an object of a class so it is not the same as its value. Instead, the *current* or *counter value* of M is a member of the object. Classes in `niqlow` that represent DP variables have a member `v` that holds the counter or current value of the variable. The internal code sets the value of `v` to correspond to the current state θ before code such as `Utility()` is called. The function `CV(M)` returns `M.v`. So either `D=M.v;` or `D=CV(M);` is how user code would set `D` to the value of M at the current state.

Recall that utility is treated as a vector valued function corresponding to the feasible set $A(\theta)$. Since m is an action variable its current value is not a scalar at θ. In this simple one-choice model the current value is always the same: $CV(m) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. If other actions were added, or if constraints on feasible actions were imposed on choice, the current value of m would be a different length because the number of distinct actions α would change.

Since M is a simple count variable, it takes on values like a loop counter or index. The earnings shock e, is also like a loop counter, so its Its counter value ranges from 0 to 14. However, e is a discretized normal random variable and the counting values are associated with both positive and negative real numbers, i.e. quantiles of the standard normal distribution, such as -1.282, the 10th percentile of $N(0,1)$. The user's code can carry out these transformations of the integer value `e.v`, but `niqlow` can track *actual values* of variables for the user. The actual values of an object are stored as vector member `actual`. Thus the actual value at any point is `actual[v]`. The current value is an index into the actual vector. The function `AV()` function retrieves this value, so when $CV(e) = 3$ $AV(e)$ might equal -1.282. For M the actual vector is `(0 1 … 39)` and `actual[v]=v`. That is, the default is that `AV(s)=CV(s)`. Only in the case like a discretized normal will there be a difference. The user can set actual values for their state and action values and can make them dependent on structural parameters.

The parameter π stored in `pi` is set to a real number on line 10 in [D]. It is not an object of a class, so `pi` is its own value. The vector `beta`, created on line 9 is also its own value, but line 2 of [E] sends it to `CV()`. This is unnecessary (at this point), but is still correct. That is because `CV()` is code so that when an ordinary scalar or matrix is passed it simply returns the value: thus `CV(beta)=beta`. The reason to do this becomes apparent when discussing estimation of parameters in Section (5.4.)

D-E. Solve and Use the Solution

Line 2 of `main()` in [B] calls a function that will solve the DP model. Solution methods are discussed Section (4). `ComputePredictions()` is also part of `niqlow` and is called in the main program on line 3. It uses the solved model and integrates over the random elements and optimal choice probabilities to produced predicted outcomes at each t. How predictions are computed and used to estimate in GMM estimation is discussed in Section (5.4).

The output of the prediction listed in Figure 2 shows the agent works with probability 0.3982 in the first period. This integrates over the discrete distribution of earnings shocks and the continuous extreme-value choice smoothing shocks as well as optimal decisions. In the last period of life a large sample of (homogeneous) people would work 13% of the time and will have accumulated 7.52 years of experience.

Figure 2. Predictions for the Labor Supply Model

t	m	M	t	m	M
0	0.3982	0.0000	20	0.1540	4.9016
1	0.3761	0.3982	21	0.1508	5.0556
2	0.3537	0.7743	22	0.1480	5.2064
3	0.3319	1.1279	23	0.1456	5.3545
4	0.3111	1.4598	24	0.1434	5.5000
5	0.2918	1.7709	25	0.1414	5.6434
6	0.2741	2.0628	26	0.1397	5.7848
7	0.2580	2.3369	27	0.1382	5.9246
8	0.2434	2.5949	28	0.1369	6.0628
9	0.2303	2.8383	29	0.1358	6.1997
10	0.2186	3.0686	30	0.1348	6.3355
11	0.2082	3.2872	31	0.1339	6.4703
12	0.1989	3.4954	32	0.1332	6.6042
13	0.1907	3.6944	33	0.1325	6.7373
14	0.1834	3.8850	34	0.1320	6.8699
15	0.1769	4.0684	35	0.1316	7.0019
16	0.1712	4.2453	36	0.1313	7.1335
17	0.1661	4.4165	37	0.1310	7.2648
18	0.1615	4.5825	38	0.1309	7.3958
19	0.1575	4.7441	39	0.1308	7.5267

The definition of the labor supply model corresponds roughly 1-to-1 with user code in `niqlow`. There has been no previous attempt to embed empirical discrete dynamic programming in a higher-level coding environment for even a simple class of models. It is true that one-time code for what has been shown so far is not complicated. Extensions of the model are shown which can be implemented with one or two lines in `niqlow` that would otherwise involve rewriting of the one-time code. Before discussing them, consider a side benefit of using a platform rather than purpose-built code: efficiency.

3.4 Efficient Computing

Dynamic programming suffers from the curse of dimensionality: the amount of work to solve a program depends on the size of the state space Θ which grows exponentially in the dimensions of the state vector θ. How big Θ can be before it is too big depends on many factors, including processor speed, solution methods and code efficiency.

The labor supply model is a small problem, and a novice coder can implement it with straightforward code. However, efficient code is not necessarily simple or intuitive to write. As a novice builds on the small problem inefficiencies in their code can invoke the curse of dimensionality before necessary. When this happens they need to discover the inefficiencies and rewrite their code. These inflections points, where a novice's progress slows down while rewriting code, may determine when a project stops. That is, the point can be reached where the marginal cost of finding and eliminating inefficiencies exceeds the marginal value of additional complexity.

This section discusses three ways efficiency is automatically accounted for in `niqlow`. Many other strategies to increase efficiency, and to balance storage and processing requirements, are encoded in `niqlow`. Models developed with `niqlow` do not avoid the curse of dimensionality, but they can delay it. Further, code development becomes more linear compared to a naive approach which includes inflection points that dramatically slow progress.

3.4.1 Time (and Memory) is of the Essence

An important distinction among dynamic programming models is whether the model's horizon, denoted T, is finite or infinite. More precisely, the issue is whether any regions of the state space are ergodic. If Θ is ergodic, t simply separates today from tomorrow and the value of all states can affect the value at any current state. Bellman's equation then implies a fixed point in the value function. If, at the other extreme, the agent is subject to aging and $t' = t + 1$, then only future states affect the value of states at t. Bellman's equation can then be solved backwards starting at $t = T - 1$. In between these extremes there are mixed clocks that include stationary and non-stationary stages as well stochastic clocks such as the risk of early death.

A solution method could always assume that the DP problem is ergodic. The reason for not doing this is practical: past states would enter calculations unnecessarily. A practical DP platform exploits the reduced storage and computation implied by a non-stationary clock. It also must exploit methods to find fixed points quickly in stationary models.

As seen in the labor supply code (Line 1 of [D]), the clock is set by `SetClock()`. The clock controls how Bellman's iteration proceeds. If t is a stationary phase (so it is possible that $t' = t$) then a fixed point condition must be checked before allowing time to move back to $t - 1$. If, on the other hand, t is a non-stationary phase then no fixed point criterion must be satisfied.

Further, empirical dynamic programming involves multiple stages which process (span) the state space, not just Bellman iteration. For example, once the value function has been computed the model can be used for simulation, prediction or estimation. These processes involve all values of time whereas Bellman's iteration only involves "today" and possible states "tomorrow". This creates another complication. While iterating on Bellman's equation the transition $P(\theta'; \alpha, \theta)$ should map into only the possible next time periods. But when simulating outcomes the transitions must relate to model time.[14]

`niqlow` addresses all these issues for the user without re-coding. It accounts for differences in how backward iteration proceeds and what future values are required to compute current values. It also uses different linear mappings from state values into points of the state space depending on whether it is accessing the value function or tracking model time.

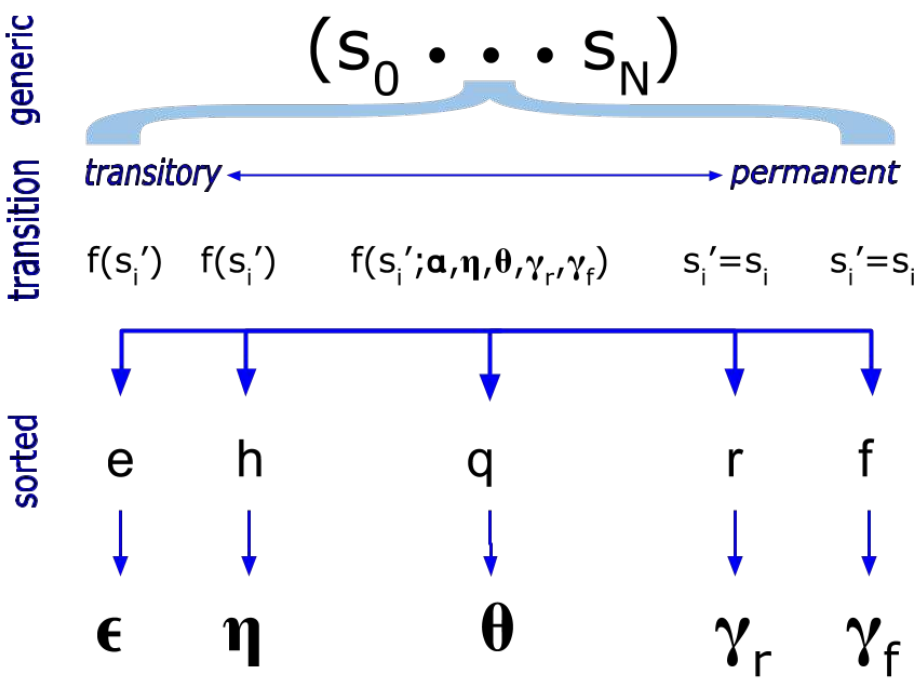
3.4.2 Not all State Variables are Equally Endogenous

A state variable has a transition that determines its value in the next period depending on the current state of the program and the action chosen. Empirical DP models contain some state variables that follow simple or

specialized transitions, such as the earnings shock e in the labor supply model. The agent conditions their choice on the realized value of e, but e has no direct impact on future states, including its own value which is IID. This means less information needs to be stored about e than, say, M. `niqlow` handles this by letting the user place state variables in different vectors (really lists of state variable objects).

In the labor supply model e was added to the "exogenous" vector and M was added to the "endogenous" vector. Figure 3 illustrates the full set of options for classifying state variables. It begins at the top where the state vector contains generic state variables, s_i. Each is filtered into one of five vectors based on its transition. The two leftmost vectors contain variables for which the transition can be written $f(s_i')$ because they are IID. The two rightmost vectors contain variables that are fixed for the agent because they do not transit at all: $s_i' = s_i$. In the middle is the vector that contains state variables that may be neither transitory nor permanent. This is the endogenous vector θ. Naive code treats all state variables generically at the cost of wasted computation or storage.

Figure 3. Sorting State Variables into Separate Vectors



There are two vectors to each side of θ in the figure because specialized state variables differ in a dimension other than their own transitions. The two vectors to the right of θ in Figure 3 distinguish between random and fixed effects and are discussed later. The general expression for the transition of variables in θ is $f(q;\alpha,\eta,\theta,\gamma_r,\gamma_f)$. The transition can depend on the action and all other state variables *except* those in ϵ. By definition, a variable placed in ϵ cannot directly affect other transitions. If an IID state variable can directly affect the transitions of other state variables it can be placed in η. That is, ϵ (and ζ) satisfy conditional independence but η does not.[15]

The basic notation of a DP model involving α, ζ, and θ appearing in (2)-(5) must be extended to account for the option to sort state variables into the five vectors:

$$\begin{array}{rcl} \text{Basic} & \Rightarrow & \text{Extended} \\ U(\alpha;\theta) & \Rightarrow & U(\alpha;\epsilon,\eta,\theta,\gamma) \\ v_\zeta(\alpha,\theta) & \Rightarrow & v_\zeta(\alpha;\epsilon,\eta,\theta,\gamma) \\ P(\theta';\alpha,\theta) & \Rightarrow & P(\theta';\alpha,\eta,\theta,\gamma) \\ E_{\alpha,\theta}V(\theta') & \Rightarrow & E_{\alpha,\theta,\eta,\gamma}V(\theta',\gamma) \end{array} \tag{13}$$

That is, $U()$ potentially depends on everything except ζ which only affects $v_\zeta()$. The primitive transition depends on everything except ζ and ϵ, so the same is true of the expectations operator over future states. The new categories of state variables allow `niqlow` to economize on storage and computing in large-scale DP projects. By the same token, these extra vectors can be ignored when solving a small DP.

The minimum required information that has to be stored at θ is a matrix of dimension $\#A(\theta) \times \#\eta$, where $\#\eta$ means the number of distinct values of the semi-exogenous vector. This space holds the value of actions $v(\alpha;\eta,\theta)$. The full IID vector ϵ is summed out at each value of η. This requires only temporary storage for utility $U()$.[16]

Once $V(\theta)$ is finalized this matrix can be rewritten with the conditional choice probability matrix $P^\star(\alpha;\eta,\theta)$. This re-use of the same matrix cuts storage in half. Further, the presence of group variables implies looping over the state space Θ repeatedly. However, `niqlow` re-uses Θ for each value of γ which can drastically cut storage compared to code that stores each DP problem simultaneously.

Almost not other information about the DP problem is duplicated at each point θ. Otherwise large scale DP problems would take up more memory than necessary. A key example of this conservation of information: only one object of each state variable in the model is created. There is not a different object for a state variable associated with each state. The value of a state variable s at the current state is the member `s.v` discussed earlier.

3.4.3 Not All States Are Reachable

In the labor supply model the agent begins with 0 years of experience. States at $t = 0$ with positive values of M are irrelevant to any application of the problem to data. There is no need to solve for $V()$ at these states, although doing so causes no harm. At $t = 1$ the only reachable states are $M = 0$ and $M = 1$. The other 38 values of M are irrelevant to the problem in the second period. The terms *reachable* and *unreachable* are used for this distinction instead of *feasible* and *infeasible*. Whether a variable's value is reachable depends on the type of clock and the initial conditions not just the set of feasible actions $A(\theta)$. If the clock were stationary, or if initial conditions allowed for other initial values of M, then these states would become reachable.

Dedicated code for spanning the state space for the labor would account for unreachable states by changing the main loop, similar to this pseudo-code:

```
for ( t=39; t>=0; --t ) {
    for (M=0; M<=t ; ++M) {
        ⋮
       }
    }
```

Note the limits on the inner loop is `t` not 39. This is an example of hard-coded procedural programming for a

particular kind of state variable appearing a specific model. To enforce reachability as the model is changed requires inserting, deleting or modifying these loops. Since naive code has multiple nested loops to handle different tasks the chance of mistakes or inefficiencies is always present.

`niqlow` accounts for unreachable states for many state variable classes when `CreateSpaces()` is and it builds Θ. For example, if the model has a finite horizon clock and includes an action counter like M, then only points that satisfy $M \leq t$ are created. The user can override this if, for example, $M = 0$ is not the initial condition. This one-time cost of deciding which states are reachable reduces storage requirements and lowers the ongoing cost of each state space iteration that may occur thousands of times during estimation.[17]

3.4.4 Adding up Inefficiencies in Naive Code

Three possible code inefficiencies have been explained: generic transitions, duplicate storage of action value and choice probability, and unreachable states. How large are these issues in the simple labor supply model? The output in Figure 2 computes the savings and reports them to the user.

First, the naive state space would include $40 \times 40 \times 15 = 24,000$ states. A value function for all $24,000$ states must be computed. However, `niqlow` would instead average the 15 values of the IID earnings shocks and store only a single value at each θ. Next, `niqlow` would reduce Θ to $40 * 41/2 = 820$ states through trimming of unreachable states. And based on the finite horizon clock it would only store the value function for $2 \times 40 = 80$ points while iterating (one vector for $t+1$ and one for t). Finally, a naive solution might store $96,000$ numbers, $48,000$ for action values and $48,000$ for choice probabilities at each state (whether reachable or not). Meanwhile, `niqlow` would store $820 \times 15 \times 2 = 24,600$ values, overwriting $v(\alpha, \theta)$ with $P^{\star}(\alpha; \theta)$.

The transition $P(\theta'; \alpha, \theta)$ is computed from the transitions of the state variable objects the user has added to the model. A simple solution would express the transition at each θ as a possibly sparse matrix: one column for each point in Θ for θ' and a row for each feasible action at the current state. A naive approach would require a $2 \times 24(15)$ matrix at each θ. This is not totally naive because it uses the fact that all transitions are to the next time period $t+1$. Since e is IID its transition does not depend on θ, and since M is deterministic only two values have non-zero probabilities in the transition. This means that computing EV naively is a potentially a large matrix calculation that includes mainly zeros. Instead, `niqlow` detects the endogenous transition in the labor supply model involves only two states since the next state is either M or $M+1$.[18]

3.5 Extensions

Suppose the labor supply model is just an initial attempt that the user wants to build on. They can add/modify elements of `LS` or they can create a new class derived from `LS` keeping the base untouched. Some extensions are discussed here showing the changes needed to effect them. The new derived class will be called `LSext` in each case.

First, suppose the earnings shocks should change from IID to correlated over time:

$$e_{t+1} = 0.8e_t + z_{t+1}. \tag{14}$$

If the model were hard-coded in loops, this change would require a major rewrite. In `niqlow` it requires two simple changes to `Build()`. First, replace `Zvariable()` on line 4 with `e = new Tauchen("e",15,3.0,<0.0;1.0;0.8>);`. Now e contains an object that implements Tauchen's approach to discretizing a correlated continuous shock. As before, 15 discrete values will be used. The remaining arguments set the options of the discretization, including a correlation of $\rho = 0.8$ that appears in a vector of normal parameters. Second, since e is no longer IID it is placed in θ along with M (as on line 6 above in `Build()`). Those two changes complete the modifications.

Most DP applications involve heterogeneous agents solving related but different dynamic programs. In `niqlow` different DPs create different points in the "group space" denoted Γ. A single DP problem is a point γ in this space. This was illustrated in Figure 1 which showed γ to the right of θ. There are two types of permanent heterogeneity, corresponding roughly to fixed and random effects in econometrics models. Fixed effect variables are *observed* permanent differences in exogenous variables placed in the γ_f sub-vector. Random effect variables are *unobserved* permanent differences placed in γ_r. Memory is economized by reusing the state space for each group. That is, Θ is shared for all values of γ. [19]

In a second extension the model will include gender and race variables and a discrete unobserved skill. Three more lines of code in the build segment expands the model:

[F]
```
    LSext::Build() {
1.      Initialize(1.0,new LSext());
2.      LS::Build();
        ⋮
3.      x = new Regressors({"female","race"},<2,3>);
4.      skill= new NormalRandomEffect ("skill", 5);
5.      GroupVariables(skill,x);
        ⋮
        CreateSpaces();
    }
```

The template for `LSext` (not shown) would add static members for the new variables `x` and `skill`. Since `LSext` is derived from `LS` they common elements are already available. However, `Initialize()` on Line 1 must receive a copy of `LSext` to clone over the state space. It cannot be sent a copy of the base `LS` class as was done in the base model. `CreateSpaces()` can only be called once, and the new group variables must be added to the model before it is called.

This is why `LS::Build()` did not include calls to `Initialize()` and `CreateSpaces()`. They were placed in [C]. Now on line 2 `LSext::Build()` can reuse `LS::Build()` in [D] to set up the shared elements. If the user expects to create further extensions built on `LSext::Build` then the first and last statements in [F] would be moved to a different function so that derived classes can use it but initialize and create spaces for its class.

The `Regressors` class on line 3 holds a list of objects that act like a vector and can be used in regression-like equations such as earnings. The columns can be given labels and the number of distinct values are provided (in this case 2 and 3, respectively). The `NormalRandomEffect` on line 4 is like `Zvariable()` except it is a permanent value rather than an IID shock. Earnings and utility would be modified to these group variables (not shown).

Next, the user wants to add a choice to attend school or not (s) and a state variable to track accumulated schooling: $S' = S + s$. The agent cannot attend school and work in the same period, so the choice vector and feasible set are now

$$\alpha = (\, s \quad a \,) \in A(\theta) \equiv \{\alpha : \; s * a = 0\}\,.$$

Although initialization and space creation can only occur once, in between new variables can be added to the vectors more than once. So the extended build is simply:

[G]
```
LSext::Build() {
    ⋮
    LS::Build();
    s = new BinaryChoice("att");
    S = new ActionCounter("sch",8,s);
    Actions(s);
    EndogenousStates(E);
    ⋮
    }
```

The base version added `m` to the action vector and this adds `s` to it. `S` is an action counter like `M` but limited to 8 years of additional schooling to reduce the size of the state space.

The agent must tell `niqlow` to impose the condition that the agent can either work or study but not both. This creates an additional trimming of unreachable states: $M + S \leq t$. In this approach the agent has to impose this extra condition on reachable states. Later on a different approach is shown that will do this automatically. The user replaces two built-in virtual methods with their version:

[H]
```
LSext::FeasibleActions() {
    return CV(m) .* CV(s) .== 0;
    }
LSext::Reachable(){
    return CV(M) + CV(S) <= I::t;
    }
```

The first returns a vector of ones and zeros that indicates whether an action α is feasible at the current state θ. It says the product of each row of the action vector must be 0. Ox syntax allows the expression to closely match the definition of $A(\theta)$. The second returns a scalar 0 or 1 to indicate whether the current state is reachable from initial conditions. It needs to now what the current value of t is which up until now was not required. Since the clock is stored internally `niqlow` places its current value in the `I` class, so `I::t` is always available as well as other indices of the current state.

4. Solution Methods

In `niqlow` a DP solution method is coded as a class from which objects can be created then applied to the problem. The base `Method` class iterates over (or spans) the group space Γ and the state space Θ by nested calls to objects to iterate over parts of the spaces down to iteration over the exogenous state vectors at each endogenous state θ. These procedures are equivalent to the usual nested loops in purpose-built DP code.

4.1 Bellman Iteration

Solution methods can be categorized in different ways. One distinction is between brute force and clever methods. Brute force methods, such as Wolpin (1984), iterate on Bellman's equation (5) to solve the model while estimating parameters in ψ.. Bellman iteration is implemented by the `ValueIteration` class derived from `Method`. The function `VISolve()` used in [B] is a short cut that creates an object of the `ValueIteration` class, calls its solution function, prints out the results and deletes the object.

Bellman iteration itself depends on details of the model, most notably the model's clock. Algorithm 2 describes the algorithm and how allows properties of the clock to determine the calculations.

The form of the Emax operator itself also depends on the smoothing method, related to the presence of ζ in choice values. If no smoothing terms are present, Emax is simply the maximum of $v(\alpha;\epsilon,\theta)$ over $\alpha \in A(\theta)$. In general, the solution method relies on code related to the base class the DP model to handle it.[20]

Algorithm 2. Backward Bellman Iteration

Let Θ_t denote the subset of states with the clock set to t. Let `SetP` be a binary flag. Let V_1 and V_0 be two vectors of equal size that depends on the maximum size of Θ_t and how many different values t' can take on for the clock.

A. Initialization

1. Set t = T-1.
2. Set $V_1(\theta') = \vec{0}$.
3. Span the state space to compute and store $P(\theta';\alpha,\theta)$ at each θ.
4. Set `SetP= TRUE` if $T-1$ is a non-stationary phase.

Notes. If the clock is stationary t starts at 0. Final values from a previous solution can be stored in V_1 instead of re-initialized.

B. Iteration

1. If $t = -1$ STOP. (Convergence has occurred and choice probabilities have been computed over Θ.)
2. Visit each point in Θ_t.
    a. Compute $U(\alpha; \epsilon, \theta)$ and $v(\alpha; \epsilon, \theta)$ for each action and each IID vector ϵ.
    b. Compute EV (or Emax) defined in (5), averaging over ϵ. Set $V_0(\theta) = EV$ for each θ.
    c. If `SetP=TRUE`, replace the matrix holding $v(\alpha)$ with the choice probability $P^{\star}(\alpha; \theta)$ in equation (7).
3. Check convergence by calling the Clock's `Update()` function
4. Swap V_0 and V_1. Return to step 1.

C. Update (clock specific).

If `SetP=TRUE`

Set $t = t - 1$. (Convergence was achieved on the last iteration)

Otherwise

Compute $\Delta_t \equiv |V_1 - V_0|$

If $\Delta_t < \epsilon_{VI}$ then `SetP=TRUE`.

In purpose-built code, working backwards in t and spanning Θ_t would involve nested loops. In a language such as FORTRAN the depth of the nest would depend on the number of state variables in θ. Adding or dropping states requires inserting or deleting a loop.[21]

`niqlow` relies on a fixed depth of nesting independent of the length of vectors. The difference in the update stage is handled by a method of the clock. One segment of code works for all types of clock. Further, the same code handles tasks other than Bellman iteration. Each task, such as computing predictions, is a derived class with its own function that carries out the inner work at each state. New methods can be implemented without duplicating loops in different parts of procedural programming code.

4.2 Variations on Value Iteration

Several alternatives to Bellman iteration algorithm are currently implemented in `niqlow`. This section briefly discusses some key ones. Summaries of the algorithms appear in the Appendix.

Most methods try to reduce calculations relative to brute force methods. Rust (1987) used Newton-Kantorovich (NK) iteration, summarized in Algorithm A1. This strategy applies to a model with an ergodic clock so the fixed point can be expressed as the root of a system of equations. It relies on the state-to-state transition defined in (10) which is not needed for ordinary value iteration. A number of initial Bellman iterations reduce Δ_t, defined in Algorithm 2, below a threshold. Then NK starts to update according to a Newton-Raphson step. Since NK is a class derived from `ValueIteration` it inherits the ordinary method but also contains the code to switch.

Hotz and Miller (1993) introduced the use of external data to guide the solution method as described in Algorithm A2. It is somewhat related to NK, because it delays or even skips value function iteration completely. This class of methods is referred to as CCP (conditional choice probability) methods, because it uses observed choices to obtain values of $V(\theta)$ in one step without Bellman iteration. The Aguireibiria and Mira (2002) extension to the one-step Hotz-Miller technique is summarized in Algorithm A8. This effectively swaps the nesting in Wolpin (1984): likelihood maximization changes utility parameters which are fed to $P^{\star}$ and then updates V.

The Keane and Wolpin (1994) approximation method, described in Algorithm A3, is also derived from the basic brute force algorithm. It splits iteration over the state space into two stages. At the first stage it visits a subsample of states to compute Emax defined in (5). Information is collected to approximate the value function on the sample. At the second stage the remaining states are visited to extrapolate $V(\theta)$ from the first stage approximation using information that is much less costly to compute than the full Emax operation. An example of applying Keane Wolpin approximation is described in the replication section.

A user coding the Keane-Wolpin algorithm from scratch faces extensive changes to all nested loops. One issue is that, as Keane and Wolpin (1994) report, the approximation can save substantial computational time but is not particularly accurate. It can be useful to get a first set of estimated parameters more quickly and then either increase the subampling proportion or simply revert to brute force. So a novice coder would likely copy their brute force loops and then "hack" them to carry out the two stages. Now the code has two nested loops that need to be kept in synch as the model changes or the brute force approach is abandoned altogether.

In `niqlow` the two algorithms can be compared by adding three lines of code while being ensured they are solving the same model:

[I]

```
    ⋮
    vi = new ValueIteration();
    kw = new KeaneWolpin();
1   vi->Solve();
2   SubSampleStates ( 0.1, 30 , 200 );
3   kw -> Solve();
```

On Line 1 ordinary value iteration is used. Then on Line 2 the user creates a 10% subsample of reachable states with a minimum of 30 and a maximum of 200 states at each t. Line 3 then applies the KW approximation. Output of the two solutions can be compared, and routines are available to compare the value function results between two algorithms.

Another common problem combines Bellman iteration with the calculation of reservation values of a continuous variable Z, where $Z \sim G(z)$ and is IID over time. Like the choice-specific value shocks ζ, values of Z are neither stored nor represented as an element of the ordinary state vector. Instead, $G(z)$ affects equations in the solution method and ultimately choice probabilities. With a few changes shown below the labor supply model can be converted to a reservation value problem in `niqlow`.

Finding reservation values in `niqlow` is described in Algorithm A4. To use t, several conditions must hold. First, α must to be one-dimensional (only one variable added to it); no smoothing shock ζ is included; no state variables are placed in the ϵ and η vectors; and choice values $v(\alpha,\theta)$ must the single-crossing property. The user must parameterized the model to enforce the last condition. The other conditions are enforced by `niqlow` by requiring the reservation value method be applied to a model of the `OneDimensionalChoice` class. This special class also creates dynamic (non-static) space for $z^\star$. Any IID state variables must stay in θ so that the value of $z^\star$ can be stored conditional on their values as well.

In some models not all states involve reservation wages. For example, if no offer arrives in a period of a search model no reservation value is defined. Such states still have a value that enters the value of other states. This is a condition determined by the model and denoted $CC(\theta)$ for "continuous choice." The agent is making a discrete choice at each value of continuous state variable. This is coded by finding reservation values for when the continuous choice jumps. By default, all states of a `OneDimensionalChoice` model involves reservation values, so $CC(\theta)=1$.. As with reachable states discussed above, the user can provide a replacement `Continuous()` method to indicate whether or not $z^\star$ is computed at θ.

The Appendix explains how `niqlow` represents reservation value models and as an example converts the discrete shock in the labor supply model into a continuous shock.

5. Parameters, Data, and Estimation

5.1 Overview

In estimation, parameters contained in ψ are chosen to match the external data. As usual, the current estimates, $\hat{\psi},\psi$ are used as if they were the true parameters. Most empirical DP publications contain a section that constructs the sample log-likelihood function or the GMM objective. The econometric objective seems totally specific to the model and difficult or impossible to automate across models. However, `niqlow` automates the computation of objectives built on economic models for a various structural techniques. It integrates an OOP package for static optimization and root-finding algorithms with the DP methods already discussed.

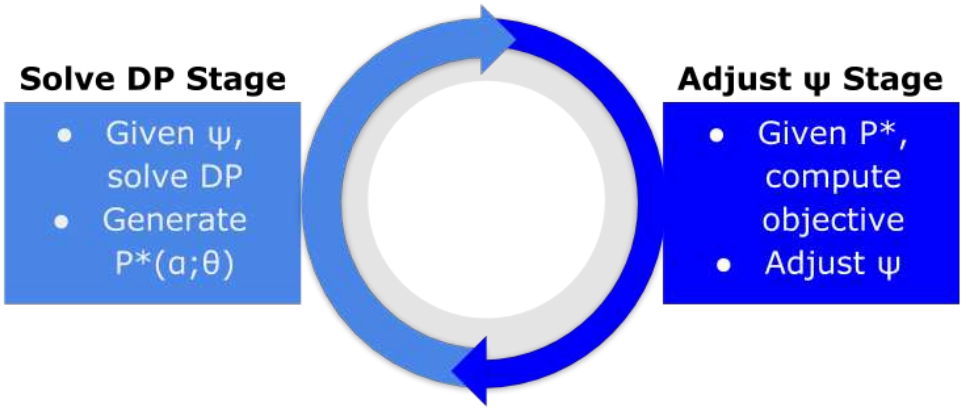
Most empirical DP uses variants of the nested algorithm introduced in Wolpin (1984) and illustrated as a two-sided feedback loop in Figure 4. Given ψ the DP model is solved and outcomes (CCPs) produced. Parameters are changed by a numerical optimization routine. It is this feedback that MaCurdy (1981) avoided by approximating the Lagrangian rather than computing its value based on the current value of the regression coefficients.

Figure 5 illustrates more layers of dependency. The top (or outer) level is an optimization algorithm that controls ψ. The parameters are tied to an econometric objective at the next level down. Levels 1 and 2 correspond to the two sides of Figure 4 which hides the layers below.

The objective relies on a data set (level 3) which must be organized to match the model to external data allowing for issues such as unobserved states and measurement error. Model outcomes and predictions use the DP solution method (level 4). Finally, the base of the pyramid is one or more DP models.

Each level of Figure 5 must interface with the adjacent levels. The DP model must also access the structural vector ψ controlled at the top level. `niqlow` provides classes for each level and the connective tissue between them. This integrated approach then supports automated construction of estimation problems. That is, the user need not write code for higher levels and can focus on the code at the base.

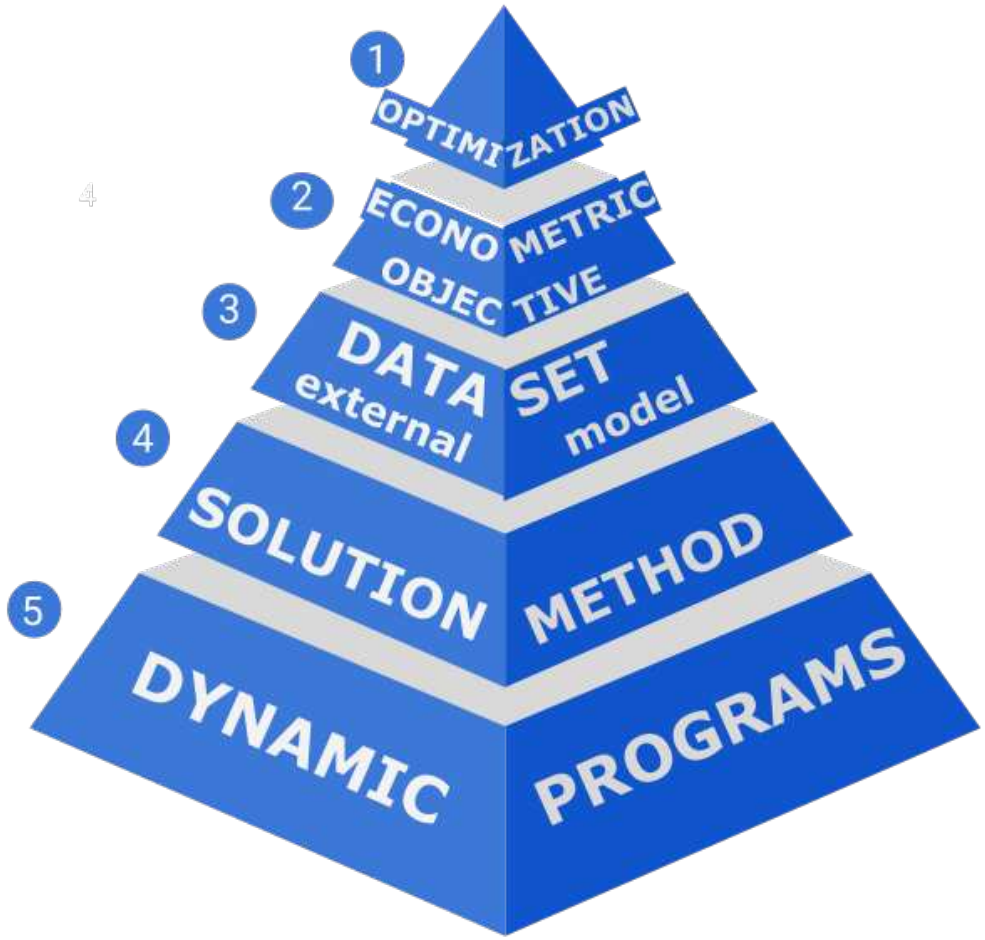
Figure 4. DP Estimation as a Two-Stage Cycle



Estimation methods such as Aguireibiria and Mira (2002)'s pseudo MLE and Imai et al (2009)'s MCMC estimation avoid computations within stages and swap the order of levels in Figure 5. Dedicated purpose-built code to implement them looks very different than Wolpin nested solution code. However, in `niqlow` the levels of Figure 5 are modular. They can be re-ordered as long different connections between objects representing each level have been written.

Previous sections used the labor supply model to demonstrate how to build up a DP as objects derived from the `Bellman` class, which corresponds to Level 5 of the pyramid. DP solution methods represent level 4. This section discusses how Levels 2 and 3 are represented in `niqlow`.

Figure 5. Levels of Dependency in Empirical DP



5.2 Outcomes and Predictions

Data related to a dynamic program are represented by the `Data` class which has two built-in child classes. Data can be based on either *outcomes* or *predictions*. An outcome corresponds to the point when all randomness and conditional choices have been realized at a state. The complete outcome is a 6-tuple written as a function of parameter vector ψ, :

$$Y(\psi) \equiv \begin{pmatrix} \alpha & \zeta & \epsilon & \eta & \theta & \gamma_r & \gamma_f \end{pmatrix}. \tag{15}$$

Some elements of Y are never observed in external data. In particular, the continuous shock vector ζ and the random effects vector γ_r are treated as inherently unobserved. Only the agent knows these values at the point they choose α.

Empirical work requires outcomes to be defined that mask components of the full outcome. Outcomes must also be put in a sequence to create the path of an individual agent. Unlike reduced-form econometrics empirical DP data cannot generically be represented as a matrix or even a multi-dimensional array of numbers. Instead, in `niqlow` they are stored as *linked-lists* of objects of the `Outcome` class. The next section builds on outcomes to define "generic" likelihood functions. The `niqlow` code is shown to estimate the simple labor supply model without the user coding the likelihood.

A *prediction*, on the other hand, is the expected value of outcomes conditional on some information. The basic prediction integrates over all contemporaneous randomness conditional on θ and the permanent variables:

$$E\left[Y\left(\theta;\gamma_r,\gamma_f\right)\right] \equiv \sum_{\eta}\sum_{\epsilon}\sum_{\alpha\in A(\theta)} f_{\eta}(\eta) f_{\epsilon}(\epsilon) P^{\star}(\alpha;\epsilon,\eta,\theta,\gamma_r,\gamma_f) Y(\psi). \tag{16}$$

This is what an agent expects to happen (in a vector sense) given that θ has been realized but the IID state variables have not. The continuous shock ζ is integrated out by the choice probability $P^{\star}$.

As with outcomes, the information not available in the external data will not always match up with the conditional information in (16). And predictions must be put in sequence to create a path. After discussing likelihod Section (5.5) returns to prediction and how they are used to construct GMM objectives.

5.3 Likelihood

There are three types of likelihood built into `niqlow` depending on what aspects of the model are observed in the data. The types are labeled F, IID and PO. Each is a version of the full outcome Y in which more information is masked or unobserved than in the previous version. The type of outcome must match up to the external data which is denoted $\hat{Y}$.

`niqlow` can determine which likelihood to apply automatically from the data read into an `OutcomeDataSet` object. The appropriate formula is computed without further user coding. Since each level of information relaxes the previous one, the PO algorithm could be applied to the other forms. Using the more restricted forms when possible speeds computation.

5.3.1 F: Full Likelihood

Define Y^F as the full information outcome from the econometrician's point of view. That is, it includes everything but these elements of Y:

$$Y^F(\psi) \equiv \left(\quad \alpha \quad \cdot \quad \epsilon \quad \eta \quad \theta \quad \cdot \quad \gamma_f \right), \tag{17}$$

where $\cdot$ denotes missing information.

To build a likelihood function using full information, first assume there are no random effects variables so it is irrelevant that γ_r is missing in Y^F. Then the likelihood for a single single observed outcome is the probability that the observed action would be taken:

$$L^F\left(\, Y(\hat{\psi}) \,;\, \hat{Y}\right) = P^{\star}(\hat{\alpha};\hat{\epsilon},\hat{\eta},\hat{\theta},\cdot,\hat{\gamma}_f). \tag{18}$$

This compact expression requires some explanation. The superscript has shift to L^F and the model outcome is shown with no superscript to avoid duplicate notation. On the right hand side is the conditional choice probability. This takes one of the forms in (6)-(9).

Since the observed outcome includes the action and all state vectors, their data values are inserted into the conditioning values. `niqlow` integrates data and predictions so it inserts the observed values for the user. The model and external versions of the data illustrated in Level 3 of Figure 5 correspond to $Y(\psi)$ and $\hat{Y}$, respectively. Calculations such as (18) then enter an overall objective at Level 4.

5.3.2 All But IID Likelihood

Data rarely contain the full information outcome of a model as defined here. More common is the next case in which ϵ is unobserved. For example, in the labor supply model the earnings shock e would probably be unobserved. However, it is typical to observe additional function(s) of the full outcome to help identify parameters of the model. In that case, earnings is observed in the agent worked. Extra information of this form is sometimes referred to as "payoff-relevant variables." In `niqlow` they are referred to as *auxiliary outcomes*, and they are all placed in a vector y.

This leads to the third outcome type:

$$Y^{IID}(\psi) \equiv \begin{pmatrix} y & \alpha & \cdot & \cdot & \eta & \theta & \cdot & \gamma_f \end{pmatrix}. \tag{19}$$

The auxiliary outcomes have been added to the front of the outcome. The agent has more information than y so it is unnecessary for them to condition choices or transitions on y. On the other hand, when information in the data is less than the agent's, y can contribute to the likelihood.

Both the smoothing shock ζ and the IID state ϵ affect the DP transition only through the choice of α. This means the IID outcome is sufficient to predict the next outcome as the agent would, namely using $P(\theta';\alpha,\theta)$. A likelihood for a sequence of outcomes for and individual can be constructed that integrates only over the IID elements. Note that the other IID vector, η, does not satisfy this condition because it can directly affect the transitions of other state variables.

The likelihood of an outcome when ϵ is unobserved is an expectation of L^F :

$$L^{IID}(\, Y(\hat{\psi})\,,\, \hat{Y}\,) = \sum\nolimits_{\epsilon} f(\epsilon \,|\, y=\hat{y})\, L^F\left(Y,\hat{Y}\right). \tag{20}$$

The choice probability component is weighted by the conditional probability of ϵ given that it generates the observed value of y. In this case (20) can be discontinuous in ψ. In the case of the labor supply model only observed earnings on one of the 15 points of support for e would have positive likelihood. Further, if the set of discrete values of ϵ consistent with the model changes with a small change in ψ it causes a jump in likelihood.

It is standard to add measurement error to observed auxiliary values (and other states or actions) in order to smooth out the IID likelihood. The measurement error is *ex post* to the agent's problem so it enters only at this stage:

$$L\left(\, Y(\hat{\psi})\,;\, \hat{Y}\,\right) = P^{\star}(\hat{\alpha};\cdots) f(\epsilon) L_y(\hat{y},y). \tag{21}$$

Here $L_y()$ is the likelihood contribution for the data given the model's predicted value for the auxiliary vector y. As shown below, `niqlow` includes built-in classes to add normal linear or log-linear noise to any outcome's likelihood contribution. Other types of measurement or classification error can be created by creating a class derived from the built-in `Noisy` class.

5.3.3 Path and Panel Likelihood

So far we have considered a single decision point in the DP model. In general, a *path* of outcomes for an agent is observed. The likelihood must account for stochastic transitions from one state to the next along the path. Denote a single outcome on the path as Y_s and the path itself as

$$\{Y\} = \left(Y_0, \ldots, Y_{\hat{T}} \right). \tag{22}$$

The index s is not necessarily the same as model time t, and since t is an element of θ it does not need to be specified separately. The corresponding observed path is $\{\hat{Y}\}$ with $\hat{T}+1$ decisions observed. Random effects in γ_r create permanent unobserved heterogeneity along a path which must be accounted for now.

Let $g(\gamma_r)$ be the probability distribution of the random effects vector. The distribution can depend implicitly on γ_f and ψ. Multiple paths with the same fixed effects are stored in a "fixed panel." These fixed panels are concatenated across γ_f in a "panel." A panel might hold simulated data only, but if external data will be read in then it is represented in `niqlow` by the `OutcomeDataSet` class.

Suppose the information available at a single decision is either full (F) or everything-but-IID as defined above. Let $\tau \in \{F, IID\}$ indicate which type of data are observed. Then the likelihood for a single agent's path is

$$\begin{aligned} &L\left(\{Y\}\left(\hat{\psi}\right), \{\hat{Y}\} \right) = \\ &\quad \sum\nolimits_{\gamma_r} g(\gamma_r) \prod\nolimits_{s=0}^{\hat{T}} \left\{ L^{\tau}(Y_s, \hat{Y}_s) \times \left[P\left(\hat{\theta}_{s+1}; \hat{\alpha}_s, \hat{\eta}_s, \hat{\theta}_s \right) \right]^{I\{s<\hat{T}\}} \right\}. \end{aligned} \tag{23}$$

The rightmost term is the model probability for observed state-to-state transitions which only applies before the last observation on the path.

Efficient computation and storage of both the DP solution and this likelihood requires some coordination. And the placement of the "nested" solution algorithm must be exact. When there are fixed effects the model must be solved for each combination of γ_r and γ_f. Let there be G different combinations of groups. As discussed earlier, the endogenous state space Θ is not duplicated for each combination of permanent values. Otherwise storage requirements would multiply G-fold. On other hand, this means each combination must be fully processed for a given structural vector ψ before proceeding to the next group. The fixed panel must initialize a vector to contain $L()$ for each of its members and then store partial calculations for γ_r. Only when the outer summation in (23) is complete can the log of the path likelihood be taken and summed across paths to form the log-likelihood. This must then be repeated for each fixed effects vector.

5.3.4 2-Stage Estimation

A version of ML estimation commonly known as two-stage estimation is available in `niqlow` (Algorithm 6). Since it was used in Rust (1987) two-stage estimation has been used to reduce the computational burden of maximum likelihood estimation. Parameters in the structural vector ψ must be marked whether they *only* affect the transition of endogenous states or not. Consistent estimates of those parameters can be found without imposing Bellman's equation using the observed transitions.

More generally, in two-stage estimation each parameter is given one of the three markers by the user. The full parameter vector then contains three lists (sub-vectors):

$$\psi = \begin{pmatrix} \psi_0 & \psi_p & \psi_u \end{pmatrix}. \tag{24}$$

The first, ψ_0, contains parameters held fixed throughout estimation, including weakly identified parameters or ones set through "calibration." The second vector, ψ_p, includes parameters that affect only transitions, $P(\theta'; \alpha, \theta)$. The final vector, ψ_u, contains parameters that affect utility and the discount factor δ if it is estimated.

At the first stage only ψ_p is estimated using the limited information path likelihood:

$$L^{1st}(\, Y(\hat{\psi})\,,\, \hat{Y}\,) = \prod_{s=0}^{\hat{T}} 1 \times P(\hat{\theta}_{s+1}; \hat{\alpha}_s, \hat{\theta}_s)^{I\{s<\hat{T}\}}. \tag{25}$$

This is the same as (23) except "1" replaces the model's generated CCP. The likelihood conditions on the observed choice can be computed without solving Bellman's equation for $P^\star$. The rest of the model setup is still required to compute the outcome-to-next-state transitions along the path. After estimating ψ_p they are fixed at those values and ψ_u is made variable. The likelihood reverts to (23).

Algorithm 5. 2-stage Estimation.

1. Set weakly-identified or other fixed parameters (contained in $\hat{\psi}_0$).
2. Set initial values of $\hat{\psi}_p$ and ensure the optimizing algorithm does not vary other elements of the overall vector. Do not iterate on $V()$ while maximizing the partial likelihood (25) by varying $\hat{\psi}_p$.
3. Fix all elements of $\hat{\psi}$ except the u vector. Iterate on $V(\theta)$ to compute CCPs and use the full path likelihood (23).
4. [Optional] Free elements of $\hat{\psi}_u$ as well. Iterate on the full likelihood to gain precision and compute correct standard errors.

As the replication of Rust (1987) in Section (6.2) demonstrates, `niqlow` implements 2-Stage Estimation without modifying the underlying code. The user declares the objective as two-stage and sends lists of estimated parameters for ψ_p and ψ_u. The user code sets the stage and estimates accordingly.

5.3.5 Type PO: Partial Observability

Implicit in (23) are two assumptions. First, the initial conditions for the agent's problem are known up to γ_r. Second, the full action and endogenous states in θ are observed so that the likelihood only integrates and sums over IID values, ζ and ϵ. Calculation of L can then proceed forward in time and the path, starting with $s = 0$.

More generally, partial observability of the outcome creates a difficulty for computing the path likelihood forward in time. Let q be a state variable in the endogenous vector θ that is unobserved in the data at outcome Y_s. It could missing systematically as a hidden state or incidentally for this outcome alone. It has a distribution conditional on past outcomes that can be computed moving forward. So the contribution to likelihood at s can be computed by summing over possible values of q weighted by its conditional distribution. However, the distribution at $s+1$ depends on the distribution of q at s. So the distribution must be carried forward in the calculation. Each unobserved value creates more discrete distributions to sum over.

Even if all the unobserved states are tracked and their joint distribution across missing information computed, this can still be inadequate to compute the correct likelihood. With q missing at Y_s, suppose also that the future point $s+k$ it equals $q^\star\star$ in the data. However, that value happens to have zero probability of occurring if $q = q^\star$ at s. For example, if a stock was only at $q^\star$ at s it may be impossible for the stock to grow to $q^\star\star$ by $s+k$.

Since q was unobserved at s the forward likelihood included $q^\star$ in the sum and the distribution. Now the contributions of paths that start at $q^\star$ must be eliminated between s and $s+k$. The likelihood must sum over only unobserved states conditional on past observed outcomes and future *consistent* outcomes. In general there is no way to calculate the likelihood forward when endogenous states or actions are unobserved.[22]

Ferrall (2003) proposed Algorithm 6 to handle partial observability. Instead of computing the likelihood of a path going forward in time, it is computed backwards starting at the last observed outcome $\hat{T}$. At any point s the likelihood computed so far is not a single number. Instead, likelihood is a number attached to every state at s that is consistent with the observed data from s forward. This avoids the trap of following paths forward that end up inconsistent with later outcomes.

A one-dimensional vector of likelihoods, denoted L_s^{PO}, contains the information to track likelihood contributions farther in the future. If, at a particular s, the IID-level outcome is observed then this vector collapses to a single point. Encountering missing values for a smaller s will expand the L_s^{PO} to vector again. The algorithm works for the special cases of Full or IID observability as well, but moving backwards is slower than algorithms that can move forward in s. When data are read into `niqlow` the user flags which variables are observed, and all other variables are treated as unobserved. As the data on the observed variables is read in incidental missing values are also detected. Then each observed path can be assigned one of the three tags $\tau \in \{F, IID, PO\}$ and the most efficient path likelihood is then computed.

Algorithm 6. Partial Observability Path Likelihood

Initialization

Set $s = \hat{T}$. Initialize $L_{s+1}^{PO} = 1$. Define sets of outcomes Υ_0 and Υ_1, initialized as empty.

Iteration

1. Construct Υ_0 as outcomes consistent with the current observation on the path, $\hat{Y}_s$:

$$\Upsilon_0 \equiv \left\{ Y : Y \in \hat{Y}_s \right\}.$$

Here "$\in$" means that the model outcome has the same values as the data.

2. For all $Y \in \Upsilon_0$, define

$$P(Y'; Y) = \begin{cases} \sum_{\theta \in Y} \sum_{\alpha \in Y} P^{\star}(\alpha; \theta) L_{s+1}^{PO}(Y') & \text{if } Y' \in \Upsilon_1 \\ 0 & \text{otherwise.} \end{cases}$$

Transitions inconsistent with observations later in the realized path are zeroed out.

3. Transition probabilities are multiplied by the conditional likelihood: $\forall Y \in \Upsilon_0$,

$$L_s^{PO}(Y) \equiv L^{IID}\left(Y, \hat{Y}_s\right) \sum_{Y'} P\left(Y'; Y\right).$$

4. Decrement s. Swap Υ_0 and Υ_1.

5. If $s \geq 0$, return to step 1. Otherwise, handle initial conditions.

Initial Conditions

a. With simple aging $s = 0$ corresponds to the first observed decision period, t_{min}. If $t_{min} > 0$ then the same process as above is continued but all outcomes are consistent with the data (because there is no data for $t < t_{min}$). If $t_{min} = 0$ and Υ_1 is a singleton, then its likelihood is the path likelihood. Otherwise, average the likelihoods of outcomes in Υ_1 to collapse the path likelihood to a scalar.

b. If the clock is ergodic then optionally weight initial outcomes on path with the stationary distribution defined in (11).

5.4 Estimating the Labor Supply Model

Consider using external data to estimate the parameter vector β of the labor supply model. For simplicity other parameters are held fixed so we can set structural vector as $\psi = \beta$. Data for individual i is a path of the form:

$$\left\{ \hat{Y} \right\}^i = \left\{ \ (m_s, M_s^o, E_s^o) \ \right\}_{s=0,\ldots,\hat{T}^i}^i.$$

All work decisions are observed and there are no gaps in model time ($t_{s+1} = t_s$). Actual earnings, E^a, depend on the observed work choice:

$$E^a = \begin{cases} E() & \text{if m=1} \\ . & \text{if m=0.} \end{cases}$$

The earnings shock e is unobserved when not working. It could be inferred from E^a except observed earnings, E^o, include log-linear measurement error:

$$E^o = e^{\nu} E^a, \quad \nu \sim N(0, \sigma^2).$$

The auxiliary vector y appearing in (19) contains E^o. The observed path does not necessarily start at $t = 0$. If it does, then observed experience M^o equals actual experience because it can be computed from past work decisions before sending the data to `niqlow`. Otherwise, we'll assume that M_0^o equals initial experience (acquired perhaps through retrospect questions). In this case (23) is the likelihood for one observation with type $\tau = IID$.

To bring the simple labor supply model to this data, derive from `LS` a new class:

[J]
```
class LSemp : LS {
 static decl obsearn, dta, lnlk, mle, vi;
 static ActualEarn();
 static Build();
 static Estimate();
    }
```

Since `LS` is the parent class all its components are also in `LSemp`. `LSemp` does not declare `Utility` because its utility is the same as `LS`. The model is built using the code already included in the base `LS` class:

[K]
```
    LSemp::Build() {
1.      Initialize(1.0,new LSemp());
2.      LS::Build();
3.      obsearn = new Noisy(ActualEarn);
4.      AuxiliaryOutcomes(obsearn);
5.      CreateSpaces();
        }
```

The required `Initialize()` function is called on line 1. The only difference with the earlier call is that the state space Θ consists of `LSemp` objects not `LS` objects. The empirical version of the model shares the same set-up as the earlier version, so `LS::Build()` can be called on line 2. Observed earnings is created as an object on line 3. The `Noisy` class adds measurement error to the argument, actual earnings E^a coded as a static function:

[L]
```
LSemp::ActualEarn() {
    return m->myEV() ? Earn() :.NaN;
    }
```

Line 4 of [K] adds `obsearn` to the auxiliary outcomes, which makes it an element of the y vector introduced in (19).

The function for actual earnings requires some explanation. It uses Ox's `.NaN` code for missing values when not working ($m = 0$). This is determined by `m->myEV()` instead of, say, $CV(m)$, which was explained earlier as returning the current value of `m`. That is used during Bellman iteration when m is taking on hypothetical values.

At the estimation stage, however, the DP model must use the external value of actions to provide a predicted value of earnings. This was seen in the likelihood (18) where the observed action $\hat{\alpha}$ enters the choice probability. `niqlow` handles this through its `Data`-derived classes that set values shared with the DP model. `myEV()` is another method of action and state variables. It retrieves the realized value of the variable if called during a post-solution operation such as likelihood calculation.

In ordinary econometrics, such as panel IV techniques, an endogenous variable is its own value. Only the value in the data is relevant. The need for `CV()` and `myEV()` reflects the complexity of nested estimation algorithms handled by `niqlow` and illustrated in Figure 5. At the solution stage variables must take on hypothetical values to solve the model. At the estimation stage values from the external data set must be inserted into the contingent solution values such as the CCP.

`Estimate()` contains the code to compute the MLE estimates of β:

[M]

```
    LSemp::Estimate() {
1.      beta = new Coefficients("B",beta);

2.      vi = new ValueIteration();

3.      dta = new OutcomeDataSet("data",vi);
4.      dta -> ObservedWithLabel(m,M,obsearn);
5.      dta -> Read("LS.dta");

6.      lnlk = new DataObjective("lnlk",dta,beta);

7.      mle  = new BHHH(lnlk);
8.      mle -> Iterate();
        }
```

On line 1 β is created as an object derived from the `Parameter` class designed to be manipulated by optimization algorithms. Like state variable objects, the value of the parameter is the `v` member of the class. Different classes constrain parameters in ranges of real numbers and vectors of related parameters like β.

In the basic model β was an ordinary vector. Its value was set inside `LS::Build()` in [D]. In this version the goal is to estimate β from data. So on line 1 its value is replaced. It now holds an object of the `Coefficients` class. This is designed to hold a vector of freely varying parameters like regression coefficients. A constant vector is needed as default for starting values. Since `beta` contains a vector before this line, its current value can be sent as the initial vector. By the end of line 1 that vector is stored internally in the object and `beta` now contains an object not a vector.

The code for the true earnings function `LS::Earn()` in [E] already included `CV(beta)`. So when this model is estimated it will retrieve the values from the parameter object under the control of an optimization algorithm.

In the first use of the labor supply model the simple `VISolve()` function carried out Bellman iteration once. Its use of a solution method object was hidden from the user. Now that the solution method is nested within a likelihood calculation it is not enough. The `vi` member holds the value iteration object (line 2). Somehow this object must be embedded (nested) within the estimation procedure.

Lines 3-5 in [M] create the data set object. As illustrated in Figure 5 this handles both external data and model predictions. The dataset object also needs to know which model outcomes are in the external data (implying other variables should be treated as unobserved). Line 4 says that m, M, and observed earnings are in the data and will have the same labels as their objects. Now the data can be read from a Stata file on line 5. A column in the external data must hold the ID of each path and the value of t so that a panel of paths can be created. Since they were not set explicitly in the code, the default labels will be used and must appear in the data file to avoid an error. As `Read()` reads in the data it determines whether there are missing values along the path and what type of variable is missing (an element of ϵ versus other vectors that imply partial observability.) By line 6 the class of likelihood function for each path in the data has been determined.

The DP solution method created on line 1 of [M] has already been embedded at line 3, because `vi` was sent as the second argument when `dta` was created. Whenever a new likelihood evaluation is needed `vi->Solve()` will be called to re-solve the model. This is the "nesting" of the solution algorithm inside the sample likelihood.

Line 6 creates the econometric objective. `DataObjective` expects a data set such as `dta` to be sent to it. It has a built in method to compute the log likelihood. The objective is the "home" of the parameters to be estimated, β.

Lines 7 creates the algorithm to carry out the outer iteration of maximizing the likelihood. Note that the DP model is analogous to the econometric objective `lnlk`, and the Bellman solution method `vi` is analogous to the optimization algorithm. One difference is that the user's code must send `lnlk` to the optimization algorithm, whereas the value function method does not take an object.

This asymmetry is caused by the fact that only one DP model can exist at any point in the execution of the program and there is not a dynamically declared object that encapsulates it. Earlier it was discussed that `niqlow` relies on `static` variables in order to delay the curse of dimensionality in terms of storage/memory required by the state space Θ. Thus, `vi` does not need to be told which DP problem to solve because there is always only one. The dynamic objects are the points in the state space each of the user's Bellman-derived class.

However, the base class for an objective like `lnlk` uses no static variables so multiple objectives may be defined if necessary, including nesting of objectives within other objectives. But `mle` must know its job is to maximize `lnlk` and not some other objective.

This example uses the BHHH algorithm (line 7), which is Newton's method except the outer product of the likelihood's Jacobian matrix is used to approximate the Hessian. This happens without any special coding from the user because any `Objective` object has two methods for evaluating itself. One returns a scalar. The other returns a vector to be aggregated into the scalar. In this case, `lnlk` returns the vector of log-likelihoods for each observation. In turn `BHHH` uses the vector version to compute the matrix of partial derivatives with respect to ψ and then the outer product.

Line 8 in [M] calls the `Iterate()` method of the BHHH object to maximize the log-likelihood starting at the initial values of β. Values read in from a file can supplant the hard-coded starting values of `beta`. This is analogous to `Method` classes for different DP solution methods, each having a `Solve()` function to carry out the task on demand.

5.5 Summary of the Estimation Code

An additional 14 lines of executable code were used to estimate the simple labor supply model. The chains of interactions in the code segment above matches the stacking illustrated in Figure 5. The DP model itself (the base level) is implicit, because only one state space can be defined. Next comes the solution method (`vi`) for solving the DP. Then a data set object (`dta`) to store the panel of external data and to interface with the solution method.

Next, an econometric objective (`lnlk`) holds parameters (`beta`) and acts as an interface between `dta` and the optimization algorithm (`mle`). When evaluating the likelihood the external side of the data is inserted into the automatically generated objective based on conditional choice probabilities and transitions. The `myEV()` function used in `ActualEarn()` matches the data and its theoretical counterpart and handles integration over unobserved components when necessary.

The algorithm `mle` is at the top of the pyramid. Its `Iterate()` method modifies parameters of the objective which in turn will call the solution method to update the DP's prediction and compare them to the data using the log-likelihood function as the metric. The algorithm does not know how the parameters it is choosing enter the problem, but the user's code ensured that their current value would enter the DP problem by using `CV(beta)` on line 2 of [E] and passing `beta` to the objective on line 6 of [M]

The user's code could substitute different classes at each level of the process without changing any other code. The user's code can create different solution methods, load different data sets, or compare different optimization algorithms on the same objective. None of this requires changing the underlying code for the DP problem as long as `CV()` and other preparations discussed above have been used to make the code ready for alterations.

5.6 Moments and Predictions

A basic prediction denoted $E[Y]$ was defined in (16) that integrates current IID randomness and conditional choices. Predictions always integrate over permanent random effects (γ_r) since they are treated as unobserved. Predictions always condition on permanent fixed effects (γ_f) since they are treated as observed. Thus there is a single predicted path for each γ_f whereas multiple outcome paths (used for MLE and data simulation) can be generated and/or observed in data for each γ_f. Concatenating prediction paths across different fixed effects produces a prediction panel.

One reason to use predictions (GMM) instead of outcomes (MLE) is limited access to confidential data on individuals. The confidential data may be housed on a secured data network that lack the computing power the empirical DP requires. If the researcher is allowed to take averaged data out of the secure system these moments can be used to estimate ψ.

5.6.1 Sequence of Predictions

To simplify notation, the description here will focus on only the endogenous state vector θ. Begin with an initial distribution over states, denoted $Q_0(\theta)$. For example, if the initial state is given as θ_0 then this would simply be a vector of 0s and one 1: $Q_0(\theta) = I\{\theta = \theta_0\}$. Or, in a stationary environment the initial conditions might be the stationary distribution $Q_0(\theta) = f_\infty(\theta)$ defined in (11).

Now consider the distribution of θ after s decisions, $Q_s(\theta)$. This distribution starts from $Q_0()$ and then accounts for the distribution of actions at 0 and the subsequent states at 1 and so forth until s actions have been taken. The prediction is the expected outcome over optimal conditional choice probabilities:

$$E_s[Y] = \sum\nolimits_{\theta\in\Theta}\sum\nolimits_{\alpha\in A(\theta)} P^\star(\alpha;\theta)Q_s(\theta)EY(\alpha,\theta). \tag{26}$$

Computationally this involves a loop over the state space and an inner product of the outcome and conditional choice probability vectors. Within the same loop, the distribution over states in the next period can be computed recursively.

First, Q_{s+1} is initialized as a vector of zeros. Then it is updated with:

$$Q_{s+1}\left(\theta'\right) = Q_{s+1}\left(\theta'\right) + \sum\nolimits_{\alpha\in A(\theta)} P\left(\theta';\alpha,\theta\right) Q_s\left(\theta\right). \tag{27}$$

Once the prediction and next stage's distribution are computed, s is incremented and the process repeated to form a path of predicted outcomes of a desired length.

Unlike path likelihood, which must deal with missing information along the path, Y can be treated as the full outcome. Figure 2 displays one element of $E_s[Y]$ vector, namely the action m, along the path prediction for the simple labor supply. The full vector would also include the state M. The exogenous states would by definition have time-invariant predictions and are not recorded in $E[Y]$. However, they can enter auxiliary outcomes which would be included in the prediction.

The external data may have missing information. Continue to define $\hat{Y}_s$ as the data but now is not an individual path but an average across *ex-ante* identical agents sharing a value of γ_f. It includes only including observed components of the outcome. To match empirical moments (averages) to the model prediction $E[Y]$, a *masking* matrix M_s selects columns of the full outcome that are observed at stage s. Then the differences between the predicted moment and the empirical moment is

$$\Delta Y_s = \left(\hat{Y}_s - M_s\hat{E}_s\left[Y\right]\right). \tag{28}$$

Masking is applied to the predicted moment only to make it conform to the external data vector.

5.6.2 Calculation or Simulation?

When the state space is large, computing the prediction is costly, mainly because of the need to compute Q_s by looping over all states. Simulation avoids computing Q_s. Further, simulation code is fairly simple and fast: draw a θ_0 from $Q_0(\theta)$ and an α_0 from $P^\star\left(\alpha;\theta_0\right)$. Compute $Y_0 = Y(\alpha_0,\theta_0)$ as an unbiased estimate of $E_0[Y]$. Then draw a simulated realization $\theta_1 = \theta'$ from $P\left(\theta';\alpha_0,\theta_0\right)$. Repeat over the length of the observed paths. The vector of realized outcomes is one simulation matching to the observed data. Repeat the path simulation multiple times and average the outcomes to increase precision of the simulated estimate of $E_s[Y]$.

As with simulated MLE, simulated moments can be discontinuous in estimated parameters. For example, suppose a structural parameter (an element of ψ) affects the transition of unobserved states. Then $P(\theta'; \alpha, \theta)$ will change in response to the parameter. On the next simulation a small change in this parameter will either change the simulated next state or not, which creates a small jump in the objective. If, on the other hand, the full distribution Q_s is computed then predicted outcomes remain continuous in estimated parameters as long as $P(\theta'; \alpha, \theta)$ and $P^\star(\alpha; \theta)$ are continuous in ψ.

As `niqlow` includes tools to simulate realized paths it is possible to construct simulated method-of-moment objectives to estimate ψ. However, the base code uses direct computation of Q_s to ensure continuity at the cost of greater execution time.

5.6.3 GMM Objectives

Returning to the differences between averaged data (moments) and predictions defined in (28), concatenate them along an observed path:

$$\Delta\{Y\} = \begin{pmatrix} \Delta Y_0 \\ \Delta Y_1 \\ \cdots \\ \Delta Y_{\hat{T}} \end{pmatrix}. \tag{29}$$

To form an econometric objective the vector of differences is aggregated into a scalar value. There are 3 types of GMM objectives built into `niqlow`. The simplest case is a weighted sum of all differences:

$$J_0 = -\sum_{s=0}^{\hat{T}} \left\| \sum_{j=1}^{J} \omega_{sj} \Delta Y_s^j \right\|. \tag{30}$$

The negative sign appears because objectives are maximized. The user specifies the weights ω_{sj} to place on each observed moment j at each point s, including an option to set all weights equal. Otherwise these weights can be ad hoc "importance" weights. Further, the number of observations averaged at each s can be read in so that weights also account for precision in the empirical moments. If the model includes different observed group γ_f there is a summation over their separate values of in (30) and the subsequent forms.

The next option is to specify a matrix or matrices to account for contemporaneous correlations between observed moments:

$$J_1 = -\sum_{s=0}^{\hat{T}} \|\Delta Y_s \; \Omega_s \; \Delta Y_s\|. \tag{31}$$

Finally, efficient GMM requires that a weighting matrix for the full path of moment differences:

$$J_2 = \Delta\{Y\} \; \Omega \; \Delta\{Y\}. \tag{32}$$

The efficient matrix Ω can be computed from individual data if available and the empirical moments are computed from them. However, this is not helpful if GMM is used because only moments over individual paths are available. `niqlow` includes procedures that will simulate individual paths and then compute Ω from them using a first-stage (consistent but inefficient) parameter vector.

The previous section discussed all the code involved in estimating the labor supply model using MLE. Much of that code would remain if GMM were used instead. The key differences would be a single statement:

```
momdta = new PredictionDataSet(UseLabel,"avgdata",vi);
```

The first line creates a prediction data set and specifies that `vi` is the nested solution method object to be called whenever the objective will be re-computed. This also specifies the default weighting scheme in (30). The statements to match outcomes to data columns, create the data objective and read in the data are essentially the same as with MLE.

Suppose parameters have been estimated using (30) as first-stage consistent but not efficient estimates. Then this line:

```
momdta -> SimulateMomentVariances(1000);
```

uses 1000 simulated individual paths of the predicted outcomes (using the first stage parameters) to compute the variance matrix of all observed moments along the path, Ω, for each γ_f. These matrices are stored in a file. (If the individual level data are available then the empirical covariance matrices of outcomes being averaged in the moments/predictions can be computed directly.) Then, for the second stage the data set would now specify intertemporal weights:

```
momdta = PredictionDataSet(UseLabel,"avgdata",vi,0,INTERTEMPORAL);
```

As with MLE example in the previous section, `niqlow` can construct GMM objectives automatically for a wide class of problems with no user coding beyond defining what is observed in the data.

5.7 Roadblocks

Return to the question posed earlier: What does the main econometric work of MaCurdy (1981) now take one Stata command but Wolpin (1984) still requires purpose-built code? MaCurdy (1981) found ways to keep data, model predictions, and estimated parameters from having multifaceted roles. Wolpin could not do this, in part because the binary choice he analyzed does not concentrate future value into a Lagrange multiplier on a budget constraint. And estimated parameters are not just coefficients on observables. They enter the value function and therefore choice probabilities for all choices over the lifetime. The choice probabilities are contingent, and the observed choices determine which one enters the likelihood. In nearly 40 years of published research no software emerged to assemble a DP model and derived its empirical content from pre-defined components when faced with these complex interactions. Instead, all or nearly all work continued to use purpose-built code within the procedural programming (PP) paradigm.

For techniques such as panel IV used in MaCurdy (1981), the data and estimated parameters do not interact with model predictions in the same way. The observed hours of work, while a choice within the lifecycle model, really only enters the model as an observed data point. There is no need to track contingent other choices that would alter future states as in a discrete DP model. Thus, hard-wired code for panel IV and other similar methods could move towards an OOP approach as followed by Stata, R and other current platforms.

Why did a similar platform for empirical DP not arise from PP code? One reason outlined above is that elements of DP models, such as state variables, encode more information than a current value. A state variable also has a transition which potentially depends on actions, other states and estimated parameters. That information has to pass from the user's code to built-in routines. Further, efficient computation dictates that details of the state transitions should alter the loops that solve Bellman's equation on the state space. For example, only states reachable or relevant to time t should enter value calculations at t. In addition, a single object, such as an action vector α, plays multiple roles. One role is in computing utility at a state while solving the model. Another a role is helping to determine the transition of state variables affected by the actions. The action vector takes on a third role when, for example, the observed choice determines which CCP enters the likelihood.

Shortest-path PP code leads to "hard-wiring" these complex elements. Changing aspects of the model involves rewriting the hard-wired code. So one particular model cannot be turned into a template for any model. `niqlow` demonstrates that the OOP paradigm can account for these multi-faceted roles without requiring the user to re-program the innermost code.

There are several reasons why researchers coding empirical DP never adopted OOP in the past so that perhaps a more general platform might emerge from a single model. First, to this day, empirical DP remains "compute bound," meaning processor speed is a limiting factor to the scale of the model. For reduced-form estimation researchers could move to Stata even when it was restricted in many ways compared to, say, FORTRAN. They could do so because by the 1980s these techniques were not compute bound.Meanwhile empirical DP continued to rely on FORTRAN or other PP languages to produce fast but single-purpose hard wired nested loops. And once estimation was complete there is little incentive to return to the code and make it general, which would be difficult anyway for the reasons give above. So when options to use OOP in other languages became available, researchers comfortable with OOP would have seen little added value for their purpose-built code.

To overcome the roadblock `niqlow` could not generalize purpose-built code. Instead it required an OOP representation of required elements of a DP model before solving a single model. Classes encode features of each element independent of the rest of the model. This and other aspects of the package create computational overhead compared to, say, nested loops in FORTRAN. However, as discussed in Section (3.3), efficiencies in the design are coded once-and-for-all rather than re-coding hard-wired loops only when inefficiencies of simple code become a limitation. Further, integrated data and optimization tools, as illustrated in the estimation of the simple labor supply model, further reduces specialized coding.

The higher fixed cost of carrying out a single "structural" estimation limits entry and keeps published results opaque to most readers. Meanwhile, published reduced form work is often accompanied by Stata do files that can be tweaked easily to carry out related work or eventually converted to native commands. Adoption of `niqlow` is still challenged by computational concerns. Custom-built and expertly-tuned code for a single problem will always finish quicker than equivalent code in `niqlow` *if* the starting line for the comparison is the start of program execution. When the starting line is pushed back to the first attempt to explain some economic behavior as a dynamic program, then time-to-completion is likely to even out and perhaps favor `niqlow`. Novice researchers appear to grossly underestimate the time required to produce empirical DP results even when carried out by experienced researchers. If their forecasts were more accurate the short-term gains from hard-wired code may seem less important. Further, as argued above, the inflection points in adding complexity to the model are straightened out by `niqlow`, so projects may have fewer footnotes stating that obvious extensions were not made because of computational cost.

The final component of the roadblock is not building on the past. A researcher who has published an empirical DP project can share their custom and finely tuned code for their problem. This code may not run on the novice coder's system, and the code is unlikely to include documentation or instructions for adopting the code for other problems. This roadblock is lowered by `niqlow` because it is open source, fully documented, and implemented in a free mathematical language. Learning from past work directly by modifying code that replicates existing DP results becomes possible.

6. REPLICATIONS

Replicating published papers in `niqlow` helps verify aspects of the code and demonstrates it is a viable platform for building on existing work. As Hammermash (2007) argues, systematic replication exercises in econometrics have proven to be difficult. They receive mainly lip service. Replication of empirical DP papers poses some unique challenges and at least one advantage compared to other fields. The main challenge is reconstructing the layers of computation illustrated in Figure 5.

One advantage for replicating empirical DP compared to "reduced form" empirical papers is that most empirical DP papers report in-sample and counterfactual policy predictions that do not require the data to repeat. Clear descriptions of the model, values of all structural parameters, and explanations of what outcomes are being reported are sufficient to attempt a replication. On the other hand, replicating reduced form analysis usually requires the original data (or at least cross products of the estimation sample). Recent reduced-form results are easier to replicate than older ones because data and code in standard platforms such as Stata and R are now commonly provided.

For empirical DP models, code may be provided for recent work, but determining whether it contains mistakes is difficult by simply reading and executing it. Further, published work pushes the boundary of feasible computations so the cost of attempting to replicate current work is very high. On the other hand, results produced at the boundary of feasible computations twenty years ago are now cheap to compute. Not only are computers orders of magnitude faster, but the original analysis required many iterations and specification changes to optimize the econometric objective. Overall, replicating or verifying old reduced-form is harder than current reduced form analysis, but the opposite is typically true for empirical DP results.

The replications discussed here focus on pioneering work covering now-standard approaches to empirical dynamic programming. Each replication is describe following the template used for the lifecycle labor supply example. Key aspects of the model are defined using `niqlow`'s terms. Some `niqlow` code is shown when it illustrates a new and general point. In some cases original symbols are changed in order to avoid conflict with `niqlow` notation. When discrepancies between the replication and original results remain after efforts to eliminate them some possible reasons are given.

6.1 Wolpin's Fertility Model

As discussed earlier, Wolpin (1984) models fertility choices given random child mortality and non-random lifecycle income. The replication solves the model differently than the original in order to use the same approach as the simple lifecycle labor supply model example. The original solution method is the reservation value approach followed in the replication of Wolpin (1987) in Section (6.4).

Wolpin (1984) Model Summary (replication version)

Element	**Value**	**Category / Params / Notes**	
Clock	t	Ordinary Aging	$T = 30$.
CCP		Bellman (Unsmoothed)	
Actions	$\alpha = (n),$	Binary Choice	a birth
States:	$\epsilon = (e),$	Zvariable	N=15
	$\theta = (M)$	RandomUpDown	N=19.
Choice Set	$A(\theta) = \begin{cases} \{0,1\} & \text{if } t < 20 \\ \{0\} & \text{otherwise.} \end{cases}$	fecund years	
Utility	$RU\,(n, M) = (\alpha_1 + e)M' + \alpha_2(M')^2 + \beta_1 X + \beta_2 X^2 + \gamma_1 M' X + \gamma_2 M'$	X : income net of child-rearing costs.	
	$EU = (1-\pi_d)RU(1, M+1) + \pi_d U(1, M)$	Exp. utility of surviving children	
	$U\,(\,) = \begin{pmatrix} RU(0, M) \\ EU \end{pmatrix}$		

Wolpin solves for the reservation value of e for having a child, but in the replication e is simply discretized and optimal choices are computed without smoothing. Since parameters are not being estimated it does not matter that the choice probabilities are not smoothed. A newborn child dies during the year before utility is realized, thus the one-period utility includes an expectation over the random outcome. The mortality probability π_d is specified as $\pi_d = \Lambda(\pi_0 + \pi_1 t)$ where $\Lambda()$ is the logistic distribution function.

The stock of children, M, is therefore not a simple action counter we have encountered several times already. Instead, it is a special case of built-in class for state variables that moves up or down one value or stays the same. The user provides an expression that returns the state-contingent vector of probabilities and the `RandomUpDown` handles the transition.

The parameter estimates reported in the published version are slightly different than the ones reported in the working paper version, Wolpin (1982). The working paper reports more predictions based on the estimates than the published version. Thus, the working paper results are the target of the `niqlow` replication. In particular, the focus is on the average value of $P^{\star}(1;\theta)$, the probability of a birth, by age of the mother. This requires summing over realized states of surviving children, a calculation within the realm of niqlow's `Prediction` class. The paper also reports changes in lifetime children born based on hypothetical changes in income and child mortality rates.

Table 1 reports a comparison of the reported and replicated predictions. Replicated birth probabilities by year are within .02 of the reported probabilities at each year. Each row of Panels B and C replicate behavioral responses to different hypothetical changes in income and child mortality. Looking at total fertility (column N) the replicated values are all within about 2% of the reported values. The relative changes are all in the same direction as well. The difference in solution methods is not a large part. Reported figures are based on 200 values of the discretized normal shocks and sensitivity of the replicated values to changing that is small compared to the discrepancy with the reported values. Changes in numerical precision in computing over 40 years are probably sufficient to explain the discrepancy.

Table 1. Replicating Wolpin (1984) Results

A. Birth Probabilities by Age

t	Original	Replicated	t	Original	Replicated
0	.138	0.13930	10	.400	0.39801
1	.360	0.37811	11	.376	0.36816
2	.552	0.54229	12	.345	0.33831
3	.543	0.53234	13	.313	0.30630
4	.530	0.52239	14	.281	0.27363
5	.515	0.50746	15	.250	0.23881
6	.498	0.49254	16	.216	0.20701
7	.477	0.47264	17	.182	0.17413
8	.454	0.44776	18	.149	0.14428
9	.428	0.42289	19	.122	0.11473

B. Hypothetical Income Changes on Fertility

ORIGINAL				REPLICATED		
N1-5	N6-10	N	|	N1-5	N6-10	N
2.124	2.369	7.047	|	2.1144	2.3433	7.0211
2.117	2.362	7.024	|	2.1095	2.3380	7.0058
2.121	2.367	7.039	|	2.1144	2.3433	7.0197
2.134	2.380	7.086	|	2.1294	2.3578	7.0605
2.154	3.402	7.158	|	2.1443	2.3728	7.1242
2.126	2.376	7.146	|	2.1244	2.3532	7.0651

Each row is based on different values of parameters in the income process Y (changes in values not shown). Columns are periods 1-5, 6-10, and overall. Reported values are expected number of births in the range. (Columns for 11-15 and 16-20 are not shown here.)

C. Hypothetical Child Survival and Fertility

ORIGINAL			REPLICATED		
N1-5	N6-10	N	N1-5	N6-10	N
2.124	2.369	7.047	2.1144	2.3433	7.0211
2.078	2.326	6.801	2.0939	2.3163	6.9418
2.032	2.282	6.555	2.0647	2.2883	6.8454
1.979	2.230	6.274	2.0380	2.2587	6.7561
1.927	2.175	6.001	2.0100	2.2331	6.6611
2.003	2.356	7.018	2.1443	2.3775	7.1292

Each row is based on different values of parameters in the child mortality logit (changes in values not shown). Columns and reported values same as Panel B.

Source of Original: Wolpin 1982. A. Table 5, page 43. B. Table 8, page 45. C. Table 9, page 47.

6.2 Rust (1987)

Rust 1987 is a model of bus engine replacement and is probably the most replicated paper within this class, because the model is influential and fairly simple. Rust released documented code and the data in the 1990s that is still available, although how many researchers have used that code is unknown. The package was not designed to be a general purpose platform and applications would have likely been very similar to the bus replacement model. As with the labor supply example shown earlier, we begin with replicating the model's predictions at the estimate parameters. Then, following the example further, a derived class is used to incorporate data and replicate some econometrics results.

6.2.1 Solution

The original article estimates many different specifications on different bus engine types. The replication is based on Model 11, a case emphasized in the original. It includes a linear cost of bus maintenance estimated on group 4 engines and fixing the number of discretized odometer readings to 90.

Rust (1987) Model Summary

Element	Value	Category	Params / Notes
Clock	t	Ergodic	
CCP	ζ	ExtremeValue	$\rho = 1.$
Actions	$\alpha = (d),$	Binary Choice	replace engine.
States:	$\theta = (x)$	Renewal	$N = 90$, discrete mileage
Choice Set	$A(\theta) = \{0, 1\}$		for all θ
Utility	$U(\,) = \begin{pmatrix} -\theta_1 x \\ -RC \end{pmatrix}$		Linear cost.

`niqlow` has a pre-defined `Rust` class which includes the first three components above as pre-defined features. The user simply adds their own state variables to the model. It also provides `Renewal` state variable class. The user specifies the increment probabilities and the binary action that renews the process. The basic code (not showing all statements):

[N]

```
struct Zurcher : Rust {
    static decl x;
    Utility();
    }
Zurcher::Run() {
    Initialize(new Zurcher());
    EndogenousStates(x = new Renewal("x",NX,d,theta3) );
    CreateSpaces();
    SetDelta(0.9999);
    VISolve();
    }
```

Figure 3 in the paper compares predicted values of $P^{\star}(1;x)$, probability of replacement given mileage, for discount factors of 0 and 0.9999. Figure 6 compares the original predictions and the replications. The additional information in the original figure is the empirical hazard rate. The replication is very close as can be determined when looking at the axis values at key points. For example, the curves both cross at about $x = 63$ and P^{star} at 0.03.

Figure 6. Replicating Rust Figure 3

Original

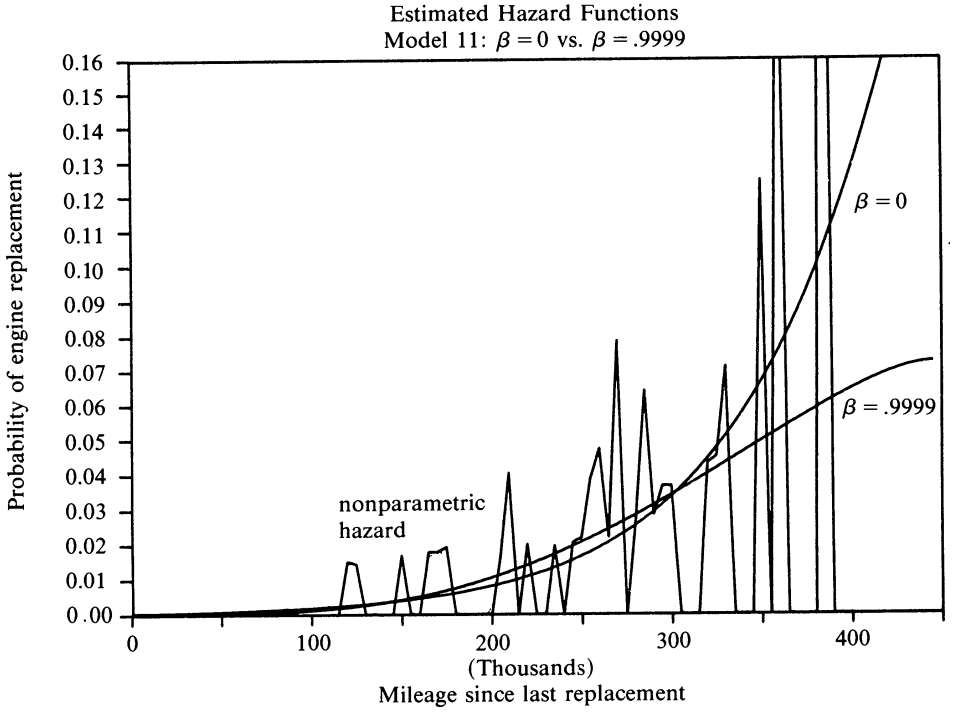
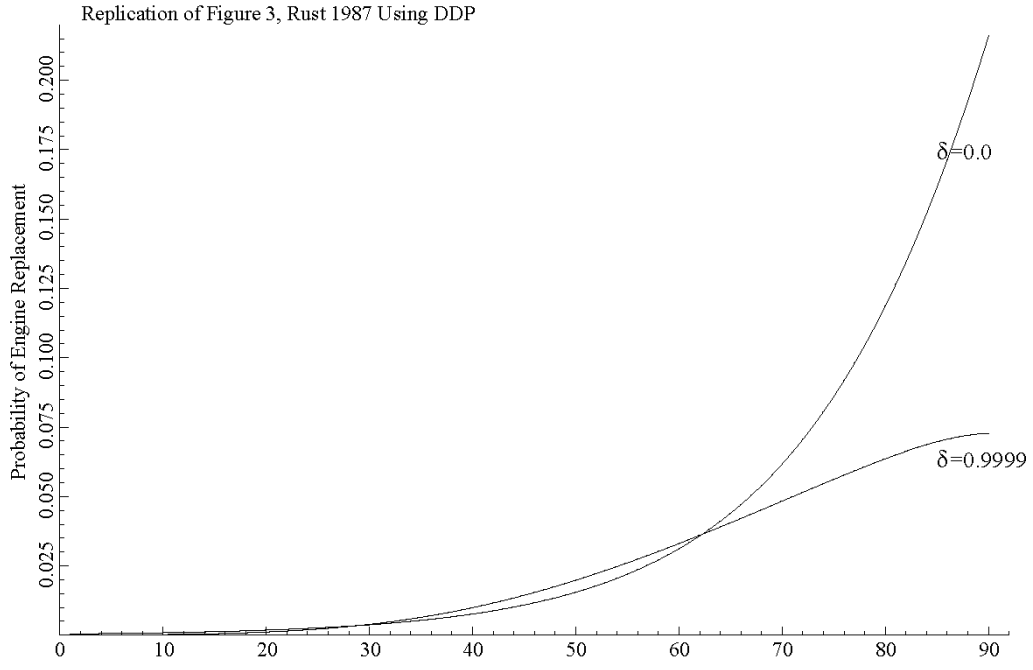


FIGURE 3

Replicated



6.2.2 Estimation

Having verified that `niqlow` replicates the solution at estimated parameter values, consider the estimation stage. The bus data was read into to an `OutcomeDataSet`. A new class `EZ` derived from `Zurcher` accounts for parameters, data and estimation. As with the labor supply example discussed above, `Zurcher::Utility()` is reused in the derived class. Rust 1987 used the two stage estimation procedure in Figure 4. The key statements used to replicate Rust's complete estimation procedure are

[O]

```
1.     plist = EZ::SetUp(0);
2.     EMax  = new ValueIteration();
3.     buses = new BusData(EMax);
4.     nfxp  = new DataObjective("ZurcherMLE",buses,plist[0]);
5.     mle   = new NelderMead(nfxp);
6.     nfxp  -> TwoStage(plist[One],plist[Two]);

7.     nfxp  -> SetStage(0);
8.     mle   -> Iterate(0);

9.     nfxp  -> SetStage(1);
10.    mle   -> Iterate(0);
```

As with the labor supply estimation code in Section (5.5), the statements mirror the pyramid in Figure 5. `EZ::SetUp()`, which is not shown, returns three lists of parameters: the full list ψ and the subvectors ψ_p and ψ_u defined in (24). The argument 0 sets the first row to replicate, which fixes $\delta = 0$ to estimate the myopic model. `EMax` defined on line 2 holds the same solution method which is based to the bus data object on line 3.

The likelihood is defined as a `DataObjective` on line 4, which expects a data object and the full parameter vector is sent to it. The objective is declared as two-stage and the subvectors are sent on line 6. Setting the stage to 0 on line 7 ensures a flag has been set to skip iteration on Bellman equation. Only transitions enter the likelihood (25). The renewal transitions are optimized when iteration is started. Then the second stage is declared, which sets the no-iteration flag to false so that choice probabilities enter the likelihood (23) and only utility-related parameters are varied.

Model 11 in Rust (1987) was estimated for two discount factors. The final values of the likelihood were collected and likelihood ratio statistics computed. Table 2 summarizes the reported and replicated values. The likelihood values are within 1% of the original values. Nonetheless the likelihood ratios are different enough that computed p-values go from 0.053 in the original to 0.089 in the replication. Attempts to explain the discrepancy failed, but as with Wolpin (1984) it seems quite possible that software and hardware differences are the primary cause. More importantly, the replication verifies the automated construction of the likelihood functions inside `niqlow`.

Table 4. Replicating Empirical Results in Rust 1987

δ	Original lnL	Replicated lnL
.9999	-3304.155	-3309.692
0	-3306.028	-3311.143
Like Ratio:	3.746	2.902
p-value	0.053	0.089

Source of Original: Rust (1987), Table IX Group 4 Estimates

6.3 Wolpin (1987): Reservation Values

Wolpin (1987) estimates a search model by solving for the reservation wage $z^\star$ in a finite horizon. As discussed above, solving for reservation values is an extension to basic Bellman iteration described in the Appendix. To solve for $z^\star$ the user code must provide the utility vector at the current guess of the reservation wage z, denoted $U(z)$. To work backwards the user must also provide the expected utility of each option given the reservation wage $z^\star$, EU. In this model there are states at which no choice is feasible. In this case the user code provides utility as in the standard discrete choice models.

The clock is a finite horizon with ordinary aging, but different ranges of t represent different stages in the transition from full-time schooling to full-time work. Reservation wages are computed for search in a school and for a fixed number of weeks if leaving school without a job. For weeks beyond that the agent must accept any offer so the reservation wage is 0 (or in general the lowest possible offer) and no solution is required. Finally, the model is closed out by a large number of periods in which the wage is received each week.

Wolpin (1987) Model Summary

Element	Value	Category / Params / Notes	
Clock	t	Ordinary Aging. T=615. $t < 61$ in school	
CCP	$\ln z \sim N(\tilde{w}, \sigma^2)$	OneDimensionalChoice	
Actions	$\alpha = (a)$	Binary Choice	accept offer.
States:	$\theta = (h, m)$	`h`: IIDBinary	has offer
		`m`: LaggedAction	accepted in $t-1$.
Choice Sets	$CC(\theta) = (!m)h \,\&\, t < 115$	condition to solve for $z^\star$	
	$A(\theta) = \begin{cases} \{0\} & !h \\ \{0,1\} & CC(\theta) \\ \{1\} & (!m)h \,\&\, t \geq 115 \end{cases}$	no offer has offer, can choose must accept offer	
Utility	$PDV(z) = \frac{z+\delta_v^{T-t}}{1-\delta_v}$	lifetime value of offer	
at z	$U(z) = \begin{pmatrix} -c \\ PDV(z) \end{pmatrix}.$		
Exp. Utility	$E_w = \tilde{w}\exp\{\sigma^2/2\}$	Expected wage offer	
	$EU = \begin{pmatrix} c \\ PDV(E_w \lambda/p) \end{pmatrix}$	see (42)	
Utility	$U() = (1-m)(-(1-h)c$ $+hPDV(E_w))$	Non-choice state	

Although h, the indicator for having an offer in hand, evolves independent of other choices it is not identically distributed since its distribution depends on time. The IIDBinary class allows the user to specify a probability that $h = 1$, which in the model is

$$P(h=1) = \begin{cases} 0.01 & \text{if } t \leq k \\ \Phi\left(-2.08 - 0.0025(t-k)\right) & t > k. \end{cases}$$

Even if h were truly IID it must be included in θ since `niqlow` does not allow for exogenous state variables within reservation value models.

The estimates use a discount rate of $\delta = 0.999$, but the original text states for the long post-search period a "annual discount rate of 5 percent" was used. In weekly terms this results in a discount factor of $\delta_v = e^{\ln(.95)/52} = .99901408$. The searcher gets the full present value of wages upon acceptance. This is consistent with the model since there are no further decisions after a wage is accepted. If further decisions that depend on the accepted wage were made then a discrete approximation to its distribution would have to be tracked as a discrete state variable.

The `niqlow` code for this model combines elements of the discrete and continuous earnings shocks version of the labor supply model. Two elements are worth noting to show how they are handled. First, m is included in the state vector to indicate that a job was accepted the previous period and decision making ends. To mark a value $m = 1$ as terminal simply requires:

```
m -> MakeTerminal(1);
```

This must happen in the Build code segment such as [D]. Future values are not computed at terminal states, so the value of a terminal state is return as the utility by the user's code.

Second, as discussed earlier in Section (4.2), a state involves a reservation value if it satisfies

$$CC(\theta) = (t < T + k)\ \&\ h\ \&\ (1 - m).$$

At other states the searcher has no choice: they either have no offer to reject or they must accept any offer. This is achieved by providing a replacement for the virtual `Continuous()` method with one that returns the logical condition:

```
SchToWork::Continuous() {
    return (I::t < T+k) && CV(h) && (1-CV(m));
    }
```

The target of the replication are predicted hazard rates reported in Wolpin (1987). A hazard rate is the same as averaging $P^{\star}(1)$ conditional on not terminating yet ($m = 0$). The paper grouped the hazard by weeks which requires some additional code. The table of predicted hazards in the original paper and the replicated results are given in Table 3. The results are substantially different. Starting from the bottom, we see the replicated hazard is about one-fifth that of the reported values. The difference falls as we move back in time to the point that the initial replicated hazard is only 10% below the reported value. This first hazard includes in-school search. The computed reservation wages are for the most part negative. Given log-normal offers this implies all offers are accepted. And after week 54 all offers must be accepted if still searching (by assumption). Together this means that, except near week 1 in the table, the declining hazards simply reflect the falling offer probability not a change in rejection rates.

The reported replication values include improvements from close readings of the text and modifying the code (such as the exact calculation of PDV above). However the differences remain unresolved with Table 3 the best attempt. The original text (Wolpin 1987 p. 812) notes negative reservation values. Given the need to calculate expected values entering this period while iterating backwards it is important to handle negative values carefully. That is, when $w^{\star}$ is on the support of offers it is possible to use $w^{\star}$ to compute $v(0)$, the value of rejection. This should receive 0 weight in the calculation because $Prob(w \leq w^{\star}) = 0$ when offers are log-normal and $w^{\star} < 0$. Numerical issues in these computations from the 1980s may explain the discrepancy with modern calculations.

Table 5. Replication of Wolpin (1987) Table IV, p. 813

Weeks i	Reported	Replicated
1	.313	.2773175
2-13	.141	.08758
14-26	.135	.0825763
27-39	.127	.0690007
40-52	.117	.0577042
53-65	.105	.0545755
66-78	.097	.0445571
79-91	.090	.0355048
92-104	.083	.028323
105-117	.076	.0228554
118-130	.070	.0186378
131-143	.064	.0153897
144-156	.059	.0127247
157-166	.054	.0107464

Predicted hazard rates out of job search at estimated parameters.

6.4 Aiyagari (1994)[23]

The primary focus of `niqlow` is estimation of dynamic programming models. Replication of the calibrated equilibrium model of Aiyagari (1994) demonstrates that the tools are in place for other nested DP applications, as well as the combined estimation of equilibrium models. In the case of Aiyagari (1984) a DP model of households feeds into an equilibrium condition on prices through an aggregate production function. The `Equilibrium` type of objective in `niqlow` handles this class of problem.

Aiyagari (1994) DP Model Summary

Element	Value	Category / Params / Notes	
Clock	t	Ergodic	
CCP		Bellman (unsmoothed)	
Actions	$\alpha = (s)$	assets t'	N=?. actual v's uneven
States:	$\theta = (z \quad S)$		N=(7,?)
		`z` : : Tauchen	labor shock
		`S`: LaggedAction	current assets
Choice Sets	$A(\theta) = \{0, \ldots, ?\}$	for all θ	
Utility	$C() = S(1+r) + we^z - s,$	r = interest rate; w = wage.	
	$U(\,) = \begin{cases} \frac{C(\alpha;\theta)^{\mu-1}-1}{\mu-1} & C \geq 0 \\ -\infty & \text{else} \end{cases}$	Dynamic restriction on s	

Outside the agent's problem an aggregate production determines prices:

$$f(K,L) = K^{\alpha}L^{1-\alpha}, \tag{33}$$

where K and L are per-capita capital and labor, respectively. Equilibrium r and w satisfy value marginal product conditions:

$$\begin{array}{ll} \text{Quantities} & \text{Prices} \\ K = \sum_{\theta} f_{\infty}(\theta)S & r = f_1(K,L) - \psi \\ L = E[e^z] & w = f_2(K,L) = (1-\alpha)[\alpha/(r+\psi)]^{\frac{\alpha}{1-\alpha}} \end{array} \tag{34}$$

where ψ is the depreciation rate of capital and f_{∞} is the stationary distribution over states in (11).

Aiyagari (1994) reduces the system from 2 equations down to 1 by writing w as a function of r. The root of the equation in r is solved by bracket-and-bisection. To compute quantities in (34) Aiyagari simulates 10,000 draws of z and averages the value of S from the agent's optimal decisions. Labor supply is computed using the closed form for the expected value of a lognormal variable (already used in the reservation earnings example).

When derived from the `Equilibrium` class the quantities of inputs supplied come from DP model predictions discussed several times above. Unlike the original code, which uses analytic derivatives to compute the equilibrium conditions, `Equilibrium` uses numerical gradients. The user provides the production function as an object of the `Objective` class, the same class from which `DataObjective` and `Equilibrium` itself are derived. Some standard forms such Cobb-Douglas are provided in `niqlow`. `Equilibrium` also takes a parameter list for the system which are prices.

To create the equilibrium system takes 10 statements:

[P]

```
    AiyagariEQ::AiyagariEQ(){
1     aggF = new CobbDouglas("F",MPco,1.0,AYG::Flabs);
2     deprec = zeros(Two,One); deprec[KK] = Kdeprec;
3     price = new array[Two];
4     price[KK] = new Bounded(AYG::Plabs[KK],-Kdeprec,1.5*lam,lam);
5     price[LL] = new Determined(AYG::Plabs[LL],Wage);
6     [stnpred,Qcols] = AiyagariAgent::Build(price,KK);
7     Equilibrium("Eq",1,price);
8     SetOneDim(KK,KK);
9     alg = new OneDimRoot(this);
      }
    AiyagariEQ::Wage() {
10      return alM1*( alpha/(CV(price[KK])+Kdeprec) )^(alpha/alM1);
      }
```

When an `Equilibrium`-derived object is created it sets `aggF` as an objective, `CobbDouglas` in Aiyagari (1992). Line 2 sets that capital depreciates but labor does not (remains 0). Lines 3-5 create an array of prices. The interest rate r is constrained between two theoretical bounds. As mentioned above, the wage w is computed by a closed form consistent with equilibrium and the current value of r. The return value on line 10 using the price of capital and technology parameters set elsewhere is the closed from for w.

The household's problem, `AiyagariAgent`, is nested within the equilibrium calculations. Line 6 in [P] calls a `Build()` function (not shown here because it is similar to ones discussed earlier). This function receives prices so they can be passed on to the agent's problem. In turn, it creates and returns a `PathPrediction` object stored in `stnpred`. `Build()` has also nested the value iteration method within the prediction object it sends back. Prediction start from the ergodic distribution, $f_\infty(\theta)$. It tracks asset holdings S and labor supply e^z in the prediction. Equilibrium needs to know which columns of the prediction matrix contain these outcomes. These indices are also returned by `Build()` and stored in `Qcols` on line 6.

By Line 7 all the required elements of the `Equilibrium` class have been created so now the base creator function is called. This adds `price` to the parameter list of the system. Attempts to solve the 2x2 system failed so line 8 concentrates the system from 2 dimensions down to 1 as done in Aiyagari (1994). This means a one-dimensional root-finding algorithm can be used (line 9). This must be paired with defining the wage as a `Determined` parameter so the system varies only one price, r. This change in strategy is achieved by simply modifying lines 8 and 9 and changing the class used on line 5.

The system's value is computed by the `vfunc()` routine. The base `Equilibrium` class contains an automated `vfunc()` method, similar to the automatic econometric objectives explained above. It computes the system of equations to solve using the components defined by the user's code. That is, `AyagariEQ` does not need to provide the code. The internal code illustrates how the elements from the user code are used:

[Q]

```
    Equilibrium::vfunc() {
1       stnpred -> Predict(T,Zero);
2       Q = stnpred -> GetFlat(T,Qcols)';
3       aggF -> Encode(Q);
4       foc = aggF -> Gradient()' - CV(deprec) - vcur.X;
5       return foc;
        }
```

Line 1 computes stationary predictions for the household problem, which will itself call a value iteration method to resolve the agent's problem at the current prices. Line 2 gets the values for aggregate supply of inputs using `Qcols` to select the right columns. Line 3 then sends these values to the aggregate production function. Line 4 takes the numerical first order condition for profit maximization. Prices are stored internally in `vcur.X` for reasons not explained here. Note that Line 5 returns a 2x1 vector of first order conditions, but line 9 of [P] made internal adjustments so that only the capital equation will be used and seen by the solution algorithm.

Some details are missing in Aiyagari (1994). In particular, the grid on assets is not described completely. Further, it appears that the upper bound U and the set of points were varied as parameter values were changed. Table 4 reports the original and replicated values of r for some sets of parameters reported in the original. The top set of parameters were the main focus of the replication and the output is essentially the same.

The main issue left unexplained by the original is the number and location of grid values for S. These were modified in order to match the first outcome. As parameters varied gaps emerge between the replication and the original values in Table 4. This may be due to Aiyagari modifying the grid along with the parameter values. Some attempt was made to do this in the replication with mixed success.

One issue the replication raises is the lack of smoothing. The asset choice s is discrete, so for each S and labor supply shock z there is a unique optimal value of s. This results in limited variation in the transition probabilities near the top and bottom of the range of S. Indeed, the stationary distribution becomes unstable or non-existent for some parameter values. The finite simulation used by Aiyagari (1994) may have been a poor approximation to the stationary distribution in these cases. A small amount of ex-post smoothing of the choice probabilities would help ensure an ergodic distribution exists regardless of parameter values. In `niqlow` this simply requires a change in the base class for the agent model to one which provides either type 2 or type 3 smoothed choice probabilities.

Table 6. Replication of Aiyagari (1984)

Parameter Values			Equilibrium interest rate r		
σ	**ρ**	**μ**	**Original**	**Replicated**	**Rel. Diff**
0.2	0.0	1	4.1666	4.1668	4.3439e-05
0.2	0.0	3	4.1456	4.1580	0.0029849
0.2	0.0	5	4.0858	4.1423	0.013824
0.2	0.3	3	4.0432	4.1284	0.021078
0.2	0.6	5	3.5857	3.9045	0.088913
0.4	0.0	1	4.0649	4.1096	0.011003
0.4	0.0	3	3.7816	3.9332	0.040084
0.4	0.0	5	3.4177	3.7609	0.10043
0.4	0.3	1	3.4188	3.7562	0.098704
0.4	0.3	5	2.8032	3.3374	0.19055
0.4	0.6	1	3.7567	3.9110	0.041074
0.4	0.6	3	1.8070	2.4476	0.35450
0.4	0.9	1	3.3054	3.5640	0.078233

Source of Original: Aiyagari 1994. Table II A and B, page 678.

6.5 Rosenzweig and Wolpin (1993)

Rosenzweig and Wolpin (1993) estimates a lifecyle model of consumption, farm production and asset accumulation. Compared to the first generation of empirical DP, the model includes a larger set of state variables, more complex feasible sets, and non-linear utility in consumption.

Each year the farmer chooses to buy and sell bullocks, breed bullocks, and purchase an irrigation pump, a one-time choice. Calves must mature before they are productive. Bullocks can die, and bad weather shocks can push farm profits and therefore consumption negative. Accumulating bullocks beyond what is productive provides partial insurance for these shocks. Farmers are risk averse, negative consumption is infeasible so instead farmers hit an exogenous positive consumption floor which requires they sell-off bullocks.

Rosenzweig Wolpin (1993) Model Summary

Element	Value	Category	Params / Notes
Clock	t	Ordinary Aging	$T > 50$
CCP		Bellman (unsmoothed)	
Actions	$\alpha = (b \quad m \quad c)$	`b`: buy/sell bullocks	N=(5,2,3)
		`m`: purchase pump	actual: $\{-2, 1, \ldots, 2\}$
		`c`: birth calves	
States:	$\theta = (M \quad L_1 \quad L_2 \quad L_3 \quad B)$		N=(2,3,3,3,3)
		`M`: PermanentChoice	has pump
		`L`$_k$: LaggedAction	calves born $t-k$
		`B`: custom	Bullock stock
	$\epsilon = (e \quad w)$		N=(?,2)
		`e`: Zvariable	profit shock
		`w`: IIDBinary	weather shock
Choice Sets	$A(\theta) = \{\alpha : 0 \le b + B \le B^M$	stock	
	$\& \; c \le B$	1 calf per bullock	
	$\& \; M + m \le 1\}$	at most 1 pump	
Utility	$\Pi = x\beta_\pi$	Profit	
		$x = \left(1 \; I_{B=1} \; I_{B\ge 2} \; M \; w \; Age \; Age^2 \; e\right)$	
	$C^\star = \Pi - p_m m - p_c c - p_b b$	Gross Consumption	
	$S = (b = -B)\&(1-c)\&(1-m)$	Selloff condition	
	$C = \begin{cases} C^\star & C^\star > C_{floor} \\ C_{floor} & C^\star \le C_{floor} \& S \\ . & \text{else} \end{cases},$	Net Consumption	
	$U() = \frac{1}{\gamma}\left(C - C_{floor}\right)^\gamma$		

Consider just the endogenous state elements, the clock (item 1) and θ (item 3). In direct coding this model already requires at least 6 nested loops to span the state space, not counting the two additional exogenous variables in item 4. This is now a common size for empirical DP models, but compared to work published before 1992 the number of dimensions is a rapid growth in complexity. Many of the interactions between state variables are non-trivial. For example, because a pump lasts forever M stays at 1 for all t after $m = 1$. Further, the feasible number of bullock transactions depends on the current stock. However, in `niqlow` the state space Θ is still set up with three statements:

[R]
```
M     = new PermanentChoice("M",m);
lcalf = KLaggedAction(calf,BullMaturity);
B     = new Bullocks();
```

The first two state variable types are built into `niqlow`. The third is particular to this model and requires a new transition B'. Its transition is similar to other standard state variables.

Two policy experiments based on the estimated model are summarized as figures in the original text. The baseline predictions (not the responses to policy changes) are replicated in `niqlow`. Figure 7 shows predicted consumption over the farmer's lifecycle. The graphs begin at age 30 and end at age 90. The effect of insurance (which eliminates the effect of weather shocks) is compared to simulated paths with periodic shocks that result in selling of bullocks for self-insurance.

The replicated graph below the original does not match the consumption level. Average consumption is approximately 1700 rupees more in the replication than the original. The lifecycle profiles have similar shapes except the replicated version shows the end-of-life effect of selling off bullocks because they are not needed for smoothing. The decision horizon is not stated in the text but it appears it was set greater than the ages shown in order to eliminate this effect. Larger values of T were used in the replication to see if that could explain the differences in levels, but the age-by-age differences remained similar.

Figure 7. Replicating RW Figure 1--Consumption Profile

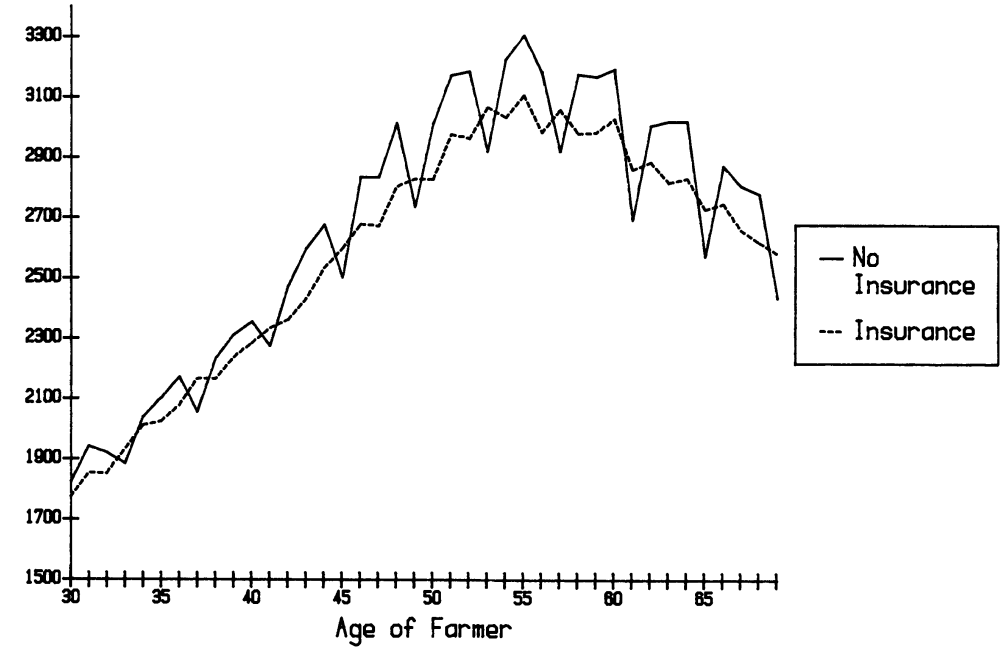


FIG. 1.—Effects of (actuarially fair) weather insurance on life cycle consumption

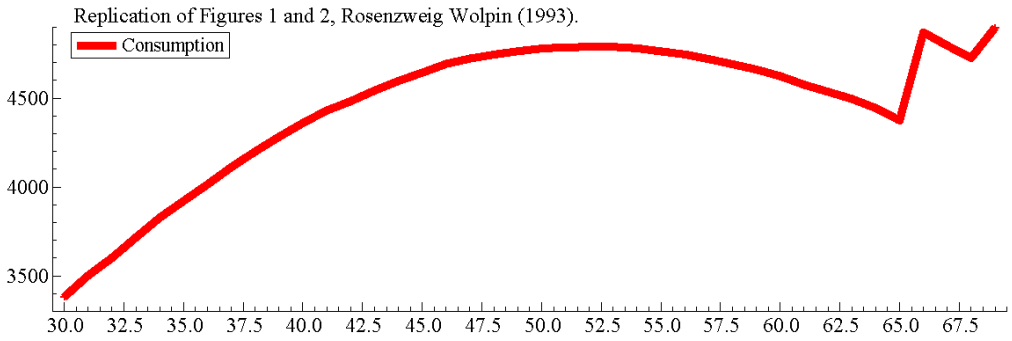


Figure 8 showed the average stock of bullocks by age under the baseline and two different policies. Again, the level in the replication is not the same. At age 30 the original baseline average is approximately .6 bullocks. In the replication it begins over .8. The original baseline stock reaches about 1.2 bullocks per farmer then declines (driven by the age profile in farm profits). The replicated result never reaches 1.0 before declining. These differences may be an artifact of the approach to simulation which has not been repeated in the `niqlow` version. They may also be caused by other differences that cannot be eliminated based on the information available.

Figure 8. Replicating RW Figure 2--Bullocks

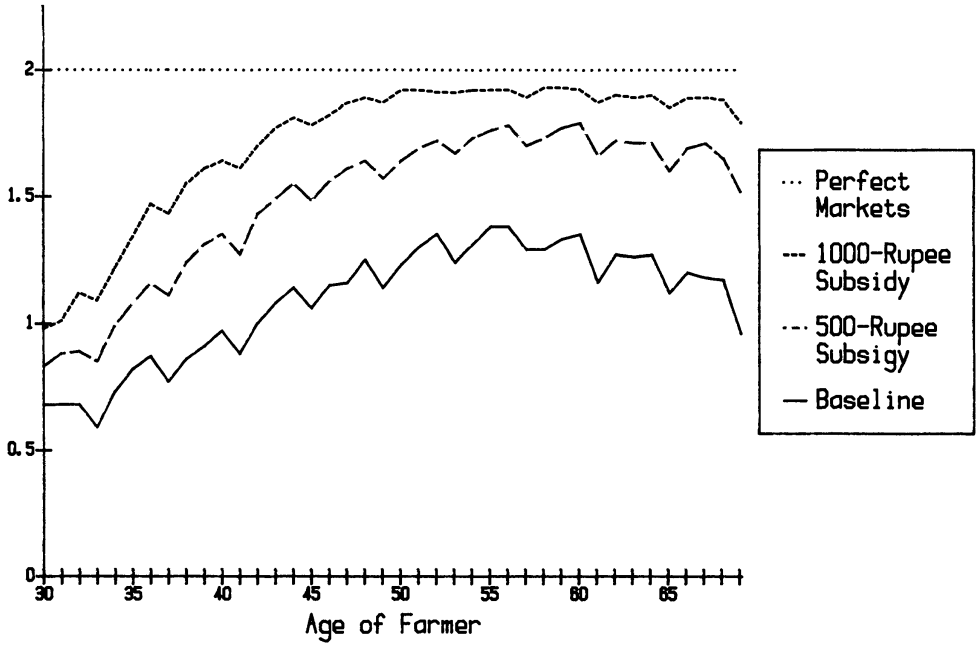
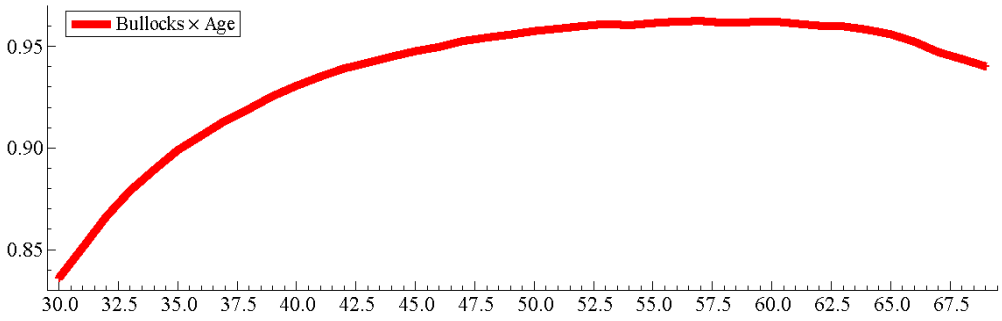


FIG. 2.—Effects of income subsidies on life cycle bullock accumulation compared with actual (baseline) and perfect markets environments.



6.6 Keane and Wolpin (1997)

Keane and Wolpin (1997) can be viewed as a multifaceted extension of the basic labor supply model coded earlier. Instead of a binary action, the agent has 5 options: work in one of three sectors, attend school, or stay home. In turn, experience in (and previous selection of) four of the options helps determine current and future wages. The one-dimensional IID discrete earnings shocks in the basic labor supply model becomes a vector of five shocks (IID over time but correlated across sectors). There is a single random effect that takes on 4 values and determines an endowment of sector-specific skill. A binary fixed effect controls for years of completed schooling by age 16, the start of the program.

Keane and Wolpin (1994) estimates three models: a static ($\delta = 0$) model, a basic model, and an extended model to explain more features of the data. The replication targets the basic model, although features in the extended model are handled by built-in features of `niqlow`. The state space in the basic model is large even by current standards. The Keane-Wolpin approximation method discussed in Section (4.2) was created to make estimation of the model feasible. However, in the original paper models were estimated using both the full brute-force and Keane-Wolpin approximation method, with an emphasis on the full solution.

Keane and Wolpin (1997) Model Summary

Element	**Value**	**Category / Params / Notes**
Clock	t	Ordinary Aging T=50.
CCP	ExPostSmoothing	Logit Kernel $\rho = 1/500$.
Actions	$\alpha = (d)$	N=5; sector WC, BC, Mil, Sch, Home
States:	$\theta = (x_0 \quad x_1 \quad x_2 \quad x_3)$	ValuesCounters
	$\epsilon = (e_0 \quad e_1 \quad e_2 \quad e_3 \quad e_4)$	MVNvectorized $Var(e) = \Omega$
Groups	$\gamma_r = (k),$	RandomEffect N=4; skill endowment
	$\gamma_f = (S_0)$	FixedEffect N=2; initial schooling
Choice Sets	$A(\theta) = \begin{cases} \{0,1,2,3,4\} & t < S \\ \{0,1,2,4\} & \text{else} \end{cases}$	School feasible for first S years
Utility	$h_d = X\alpha_d + E(k,d) + \alpha^2 x_d^2$ $-I_{d=3}[x_3. > D_y]\beta$	$d = 0,\ldots,4$ college & grad. sch. tuition
	$U(\,) = \begin{pmatrix} \exp\{h_0 e_0\} \\ \exp\{h_1 e_1\} \\ exp\{h_2 e_2\} \\ h_3 + e_3 \\ h_4 + e_4 \end{pmatrix}$	

The state variables are created as follows:

[S]
```
1.  offers = new MVNvectorized("eps",M,N,{zeros(M,1),vech(chol)});
2.  xper   = ValuesCounters("x",accept,mxcnts);
3.  k      = new RandomEffect("k",Ntypes,kdist);
4.  isch   = new FixedEffect("Is",NIschool);
```

One way to extend the labor supply model would replace the earnings shock with a five-dimensional vector. If

used here it would create a lattice of points representing the five-dimensional vector. Using just 3 discrete values for each shock results in $3^5 = 243$ exogenous points at each endogenous point. Instead, a one-dimensional state variable is used that vectorizes ϵ. `MVNvectorized` on line 1 represents ϵ using `Noffers` pseudo-random draws of IID standard normal vectors of length 5 are used. `Noffers` is much smaller than 243. It takes a vector of means and a vectorized version of the lower triangle of the Cholesky decomposition of Σ. Keane and Wolpin estimated Σ using a vector of standard deviations and a correlation matrix which were combined to produce the Cholesky representation.

`ValuesCounters` is not a class (hence `new` does not appear on line 2). Rather, it is a function that creates and returns a list of action counter objects for different values of a single action. So `xper` is a list of five state variables all ready to be added to θ. The argument `mxcnts` is a vector of maximum counts. For "home" section ($d = 4$) the max count is 0 because it does not enter X. This will result in a state variable that takes on only one value (thus expending the length of θ but not the size of the state space). The state variables inherently compute reachability as a function t so that the user code does not need to do this.

The finite-mixture random effect k has a distribution that depends on `isch`. This is handled by sending a static function on line 3 which returns a column from a matrix of type proportions, `kprob`. In the replication code the matrix is constant. When estimating it would contains sets of simplex parameter vectors. When the model predictions or likelihood is to be evaluated the distribution of the random effects variables is used to integrate over types.

The first step in computing utility is constructing a linear index vector:

$$h = X\alpha_d + E(k,d) + \alpha^2 x_d^2 \tag{35}$$

X is a matrix constructed from the experience levels in all sectors. The coefficients are choice-specific, allowing own-experience to have a different return in the sector that other experience (again noting that schooling is simply experience in the school sector). In addition, own-experience squared enters h as does the type- and sector-specific skill endowment. Initial schooling at age 16 is added to x_3 before constructing X and x_3^2, and schooling beyond 12 and 16 years (D_y) requires paying level-specific tuition.

Based on h, the base utility is a vector for current actions:

$$U_\theta = \begin{cases} e^h & d < 3 \\ h & d \geq 3 \end{cases} . \tag{36}$$

Note that U_θ equals utility at the *median* shock vector, $\epsilon = 0$. It is not expected utility because the normal errors enter employment utility multiplicatively. Keane and Wolpin used the true mean adjusted for the variance in the wage shocks to compute "maxE." U_θ only needs to be computed once for all discrete shock vectors. Decomposing this way reduces computations at each point in the state space that is sample for complete solution. It is the vector required at non-sampled points in the approximation.

With some setup of the parameter vectors, 5 lines of code compute U_θ :

[T]
```
    KWJPE97::ThetaUtility() {
1.      x = CV(xper);
2.      x[school]+=School0[CV(isch)];
3.      Er = alph0*(1~x[:school])' + ownsqr.*sqr(x)' + kcoef[][CV(k)];
4.      Er[school] -= bet*(x[school].>YrDeg);
5.      Er[:military] = exp(Er[:military]);
       }
```

`niqlow` always calls `ThetaUtility()` before it loops over exogenous variables. If the user does not supply this function the virtual method does nothing. `KWJPE97` provides a replacement both because it saves calculations and because it is required when using the Keane-Wolpin approximation method. The variable `Er` can be static, meaning all points share one location not different locations for each θ. Otherwise, storing `Er` at each θ would quintuple the storage required by Θ. As discussed earlier, the fact that the state space is reused across group variables avoids multiplying storage eight-fold across values of `k` and `isch`.

Finally, utility conditional on the shocks builds from the base utility vector:

[U]
```
KWJPE97::Utility() {
    rr = AV(offers)';
    rr[:military] .*= Er[:military];
    rr[school:] += Er[school:];
    return rr;
    }
```

`AV(offers)` returns the 1x5 vector of errors as if individual normal shocks were included. The base `Er` created by `ThetaUtility()` is added or multiplied to offers depending on the type of sector.

Code for computing both the brute force and Keane-Wolpin approximation results for the estimated basic model is available in the `niqlow` package. The replication targets choice probabilities reported in Figures 1-5 in Keane and Wolpin (1997) for the basic model. In the original they were produced by simulating paths and aggregating choices. In the replication the predicted paths are computed not simulated.

However, replications have failed to match the figures closely. Among the reasons for this are some ambiguities in the details of the original code, especially concerning the estimated variance matrix of the earnings shocks and the method to smooth choice probabilities. Choice probabilities are sensitive to guesses for these details. This itself is an obstacle to learning from previous research when no common platform exists. Further, the model is large enough that computing both the value function and predictions is costly. Elements of the replication code are validated from smaller test code, so it seems likely that either parameters are misreported in the original or a description is misinterpreted.

7. Conclusion

This paper introduces an approach to empirical dynamic programming that eliminates the need for customized coding of each project. `niqlow` is an open-source platform that uses object oriented programming (OOP) to provide its user a set of tools to design, solve and estimate a model. The platform recognizes that empirical DP involves multiple phases including: computing the value function to determine choice probabilities; evaluating the likelihood for a data set, finding consistent parameter estimates; simulating the effects of policy

experiments. The user's code consists primarily of high-level statements that select off-the-shelf components of standard models and accomplish these phases without re-inventing the wheel for their particular problem.

Using `niqlow` this paper replicates results from several seminal papers in the literature. In some cases the results are similar to the original. In other cases discrepancies remain unexplained. Results replicated here were generated before the turn of the century. What does this exercise say about current research and whether it may be verified independently? The prospects are dim until a platform such as `niqlow` is used to produce new empirical DP results.

Replication of these older papers takes anywhere from a few seconds up to 30 minutes on a laptop *given correct code has been written.* Until results match up to the published output the source of the problem must be tracked down. It could be the description in the paper is wrong or is being misinterpreted or there really was a bug in the original code. Of course the bug may be in `niqlow`, but since it is not purpose-built its output is compared to test problems with known answers as well as replication that have been successful. When a bug is discovered in `niqlow` relevant to one of the replications the code is run again to see if it discrepancies go away. But at some point an unsuccessful replication attempt simply stops.

To replicate more recent work than shown here requires deciphering descriptions of much more complex models. Even if independent replication code can be written, producing the results are likely to take non-trivial amounts of computer power. This takes as given the reported estimates. To verify that they satisfy the extremum conditions for consistency requires the original data and orders of magnitude greater computing power. As Hammermash (2007) and others have lamented, replicating existing results is valued insufficiently to justify months of coding let alone days or weeks of computing to verify results from a single publication.

There is little evidence from the literature that sharing purpose-built code for a published paper has promoted verification or direct extension of already published empirical DP work. Referees of empirical DP models are also rarely in a position to replicate results reported in manuscripts. Perhaps some referees review code submitted by the authors, but since all code is customized it seems likely that only errors in transcribing output could be caught. Finding bugs in purpose-built code for a large empirical DP model is never going to be practical. At best, referees can point to discrepancies in output that might be caused by mistakes that the authors can work to resolve.

Incorrect but unused results are not particularly harmful. If the results do not affect policies there is no need to verify them. Results generated from empirical DP papers have played at best an indirect role in shaping policy. One reason for this is skepticism among researchers not performing empirical DP themselves. For them results are opaque, not independently verified, and rarely subject to robustness checks. Thus, a conundrum exists: empirical DP results are not trusted in part because they are based on purpose-built code that will never be practical to verify. Because the results are not trusted they rarely play any role in policy debates. Since they do not influence policy verifying the results stay unverified.

Going forward, the only hope to improve verification is to adopt open standards for describing, coding, and computing models. As seen in other areas of economics, more than one standard can be adopted. Once at least one standard has been established, results produced to no standard at all should be valued less than those that can be verified within an open-source framework.

What is valued most in economics is not applications of existing methods but the creation of new methods that potentially improve on existing ones. Typically when a new empirical DP method is published it is applied to a simple model and compared to an existing method using purpose-built code. The method is not "plug-and-play." A researcher wishing to compare the approaches for themselves must "hack" code provided by the method's developers. On the other hand, plug-and-play is an increasingly accurate description of techniques coded in platforms such as R and Stata.

As with applications, publication of new methods should require they be plug-and-play on a common platform, if one exists. If required, other researchers and perhaps referees could compare the new method to old ones for other models and along dimensions not considered by the developers. This may in turn focus the literature on methods that truly improve on existing ones.

NOTES

[1] Recent reviews of the literature include Aguirregabiria and Mira (2010) and Keane et al. (2011). Recent advances in solution methods include Imai et al (2009), Arcidiacono and Miller (2011), Kasahara and Shimotsu (2012) and Aguirregabiria and Magesan (2013).

[2] As Dynare is built on Matlab, `niqlow` is written in the mathematical language Ox, which is free for research purposes and runs on most systems. Current `niqlow` syntax is used here and the code is included in the examples in `niqlow` distribution. The current version has no graphical user interface or menu system, but the OOP approach makes it straightforward to build an interface that would produce the code for the user based on menu selections.

[3] If Wolpin (1984) had followed MaCurdy (1981) it would have relied on a panel probit model. However, unlike Euler equation based model, the forward-looking factors in a discrete choice model cannot be isolated to a single Lagrange multiplier. An "approximate" structural approach in Wolpin (1984) that avoided a nested solution would have likely been a poor approximation and possibly more costly to compute than the exact solution.

[4] Statistical packages such as Stata rely on OOP in the underlying code, but users are somewhat sheltered from OOP concepts in using them. No other platform I am aware is designed as a general platform for doing model-based empirical economics, whether object-oriented or not.

[5] In the context of solving economic models, "data" refers here not just to the observations in a statistical analysis but also parameter values, prices, choices, state variables, etc. These are the quantities that the program is processing in order to solve and estimate a model.

[6] The emphasis in `niqlow` is placed on discrete actions and discrete states, but some elements of the core code includes continuous state variables and continuous choices can be incorporated. Methods for continuous time models can be added as well.

[7] The shock vector ζ can be multiplicative not additive. The additive form is more common and is the default in `niqlow`.

[8] It is possible to describe dynamic programming without defining $v(\alpha,\theta)$. However, empirical DP explains probabilistic choices, usually by integrating over the addition shock. The values of individual choices are needed to compute the choice probabilities beyond defining or solving the DP.

[9] For example, Keane and Wolpin (1997) smooth choice probabilities *ex post* because the jointly normal shocks are discretized. This is not obvious to the reader because the text refers to Keane and Wolpin (1994) and the explanation is given in footnote 23. The kernel used is not stated directly appears to be the Logit kernel used in McFadden (1989).

[10] Another approach would keep the earnings shock continuous and solve for a reservation value for $m=1$. This framework is described later and used in the replication of Wolpin (1997).

[11] Currently `niqlow` also includes model classes based on normally distributed additive shocks, both *ex ante* and *ex post*.

[12] This use of static variables to replace action and state variables is critical to memory management. If they were not static, each point in the state space would have its own version of the variables duplicated across the state space Θ. As static members they do not increase memory requirements along with the state space. The state-specific (non-static) members which expand with the size of Θ are kept to a minimum.

[13] An action counter has a deterministic transition. We can also express this as a stochastic process $P(s') = I_{s'=s+I_{a=k}}$. Further, this is what is called an *autonomous* state variable: its transition does not depend on the value of other state variables, simply the current value s and some aspects of the action vector α. `niqlow` allows for blocks of state variables for which the transition must be jointly determined. Each state variable class has a `Transit()` method which returns a pair of values: a vector of feasible integer values next period and a matrix of transition probabilities corresponding to feasible actions (rows) and feasible state values next period (columns). Because M was added to the model its `Transit()` function will be called at each state along with all other transitions. The transitions for all state variables are combined to form $P(\theta';\alpha,\theta)$ which ends up as a vector of feasible state indices and a matrix of probabilities.

[14] In between the extremes, Θ may include both ergodic and non-ergodic subspaces. For example, a model may have a sequential phase during t matters but eventually reach a stationary phase:

$$t' = \begin{cases} t+1 & \text{if } t < T-1 \\ T-1 & \text{otherwise} \end{cases}$$

In an ordinary lifecycle model $T-1$ is a final period, but under this clock the agent stays in the final period forever. Lifecycle models can often incorporate early mortality:

$$t' = \begin{cases} t+1 & \text{prob. } 1-\lambda(\theta) \\ T-1 & \text{prob. } \lambda(\theta) \end{cases} \tag{37}$$

The last period $T-1$ is death which may have an intrinsic value (such as bequests). The current value function depends on values for two different future times, $t+1$ and $T-1$. In `niqlow` the clock is a combination of t and another state labelled t''. For the most part t'' stays hidden from the user. Its job is to track which values of t' are feasible next period. Its role can be explained by reference to different types of clocks. In both ergodic and simple finite horizon models, t' only takes on one value: $t' = t$ if ergodic or $t' = t+1$ if simple finite horizon (normal aging). So for these clocks $t'' = 0$ always. But in mixed clocks or stochastic clocks more than one value of t' can occur tomorrow and t'' tracks those.

[15] An example of an IID process that could be sorted into η but not ϵ is a wage offer h with on-the-job search. If the offer is accepted it determines earnings this period also the existing wage next period. The existing wage is then state variable placed in θ because its transition depends on choices. Next period a new IID outside shock h is realized as well, but current h affected the transition beyond its influence on the action α so it must be placed in η not ϵ.

[16] In addition, `niqlow` stores the transition $P(\theta';\alpha,\theta)$ at each state because they are needed for simulation and prediction. The transitions are stored for each θ using a sparse method that tracks only feasible new state indices and the vector of probabilities conditional on choices. The transitions of the IID vectors ϵ and η are stored once and combined with the θ transition to determine the full state-to-state process.

[17] In complex models there are other ways states become unreachable. How the user specifies this is illustrated in Section (3.5).

[18] `niqlow` uses a feature of Ox that allows indexing a vector with a vector of integers. Using an interpreted language such as Ox includes overhead, but features of the syntax such of this can result in fast as well as simple and general code.

[19] Since outcomes and predictions require solutions are available for the problem, the algorithms must solve each group's problem and process it before the next one. The simple `VISolve()` function can only be used with heterogeneity to solve the problems. Use of the solved solutions requires nesting the a solution method within the use of the solution as discussed Section (5.4).

[20] Other important qualifiers include the presence of IID state variables and terminal states at which Bellman iteration is not applied. `niqlow` allows the user to control these and other details of the model.

[21] DP models coded in FORTRAN and relying on an array of indices to represent the state vector run into a hard limit of 7 subscripts or counts of state variables. `niqlow` avoids this altogether by mapping a multidimensional space into one dimension. That is, if θ is a vector of state variable indices, then the state's one-dimensional index is $I\theta$, where I is a row vector of offsets that depend on the number of values each state variable takes on. In addition, as in other interpreted languages, the length of a vector such as θ can be set dynamically during runtime in Ox.

[22] One approach for computing likelihood with unobserved states is to use simulation of outcomes based on optimal choices. This is an effective way to calculate the likelihood for a given set of parameters. The complication is ensuring that the simulated value is continuous in estimated parameters.

[23] This replication was developed with Nam Phan. Code for replicating the results in the Matlab packaged described in Kirby (2017) is available from `https://github.com/vfitoolkit/vfitoolkit-matlab-replication`.

[24] `niqlow` has a `LiquidAsset` state variable that generalizes a lagged action to account for difference actual values and approximate continuous states. That is, both a and S are thought of as discrete approximations to continuous variables. The actual values are not necessarily evenly spread out in order to have higher resolution in some segment of the distribution. If a and S take on the same actual values then S reduces to a simple `LaggedAction` variable. Otherwise, the resolution in s may be different than S, so the grid points do not match up. In this case S will have a random transition, taking on its grid values above and below `AV(s)` with probabilities equal to where s falls in the interval. This is a stochastic equivalent to a piecewise linear approximation for a continuous state variable.

APPENDIX: SOLUTION ALGORITHMS

These algorithms are explained in Section (4.2). Aguirregabiria and Mira (2011) review several methods in more detail.

Algorithm A7. Newton-Kantorovich Iteration

Initialization

Initialize as in Bellman iteration above. Set a threshold ϵ_{NK} for switching to N-K iteration. Since the model is stationary `SetP` is initially FALSE. A new flag, `SetPtrans` also starts as FALSE.

Iteration

Begin with Bellman iteration, inserting a check for $\Delta_t < \epsilon_{NK}$.

When this occurs. set `SetP=TRUE` and `SetPtrans=TRUE` to compute the state-to-state transition (10) on each iteration.

Replace step c in the Bellman Iteration that computes $V_0 = Emax(V_1)$ with:

c. Compute

$$\begin{aligned} \vec{\Delta}_t &\equiv Emax(V_1) - V_1 \\ g &\equiv \left(I - \delta P(\theta';\theta)\right)^{-1} \left[\vec{\Delta}_t\right] \\ V_0 &= V_1 - g \end{aligned}$$

Algorithm A8. Hotz-Miller Inversion

Initialization

Construct empirical conditional choice probabilities, $\hat{P}(a;q)$, where a and q are empirical observations of each combination of (α, θ).

Inversion

At each point θ construct

$$Q(\theta) = \hat{P}(\alpha;\theta)\left[U(\alpha;\theta) + \gamma_E - \ln\left(\hat{P}(\alpha;\theta)\right)\right] \tag{38}$$

Compute the vector of values consistent with empirical CCPs

$$\begin{aligned} g &\equiv \left(I - \delta P(\theta';\theta)\right)^{-1} \left[\vec{\Delta}_t\right] \\ V_0 &\equiv g * Q \end{aligned} \tag{39}$$

Algorithm A9. Keane and Wolpin Approximation

Initialization

Create a random subsample of states for each t, denoted Θ_t^S. This sampling can be controlled so no sampling takes place for some t and minimum and maximum counts are enforced.

Iteration

Follow these steps at each t.

1. For all $\theta \in \Theta_t$:

   Store action values at a single ϵ_0 (the median or mean vector): $v_0(\alpha, \theta) = v(\alpha; \epsilon_0, \theta)$ and `maxE` $= V_0(\theta) = \max v_0(\alpha, \theta)$.

   if $\theta \in \Theta_t^S$, compute `Emax` $= V$, averaging over all values of the IID state vector ϵ:

   $$V(\theta) = \sum\nolimits_{\epsilon} \left[\max\nolimits_{\alpha \in A(\theta)} U(\alpha; \epsilon, \theta) + \delta E_{\alpha,\theta} V(\theta')\right] f(\epsilon).$$

2. Approximate V on Θ_t^S as a function of values computed at ϵ_0. The default runs the regression:

   $$\hat{V} - V_0 = X\beta_t = \left((V_0 - v_0) \quad \sqrt{V_0 - v_0}\right)\beta_t.$$

   That is, the difference `Emax-maxE` is a non-linear function of the differences in action values at the median shock.

3. For $\theta \notin \Theta_t^S$, compute $v_0(\theta)$, extrapolate the approximation:

   $$V(\theta) = \max\{V_0(\theta), V_0(\theta) + X\hat{\beta}_t\}.$$

   Carry out update conditions for t as in Algorithm 2.

Notes: The user can provide a replacement for the regression specification or the approximation method by replacing virtual method of the `KeaneWolpin` class.

Algorithm A10. Aguirregabiria Mira Iteration

Initialization

Estimate transition-specific parameters using step 1 of Two-Stage Estimation.

Carry out Hotz-Miller inversion to compute initial CCPs and value function, denoted $\hat{P}_0$ and V_0. These are the stage $k = 0$ of AM iteration.

Iteration

Estimate utility-specific parameters (ψ_u) using the full path likelihood.

At stage $k > 0$, compute $\hat{P}_k$ as $P^\star$ from (7) to evaluate the likelihood. Current values of ψ_u enter $U()$ and interact with prior values of V_k and $\hat{P}_k$ enter $v(\alpha, \theta)$.

Iteration is complete when $\|P_k - P_{k-1}\|$ is less than tolerance. Otherwise, use the Hotz-Miller inversion in (39) to update V_k based on the new choice probabilities.

Reservation Values

For a binary choice a, there is a single $z^\star$ at each (implicit) state that solves $v(1, z^\star) - v(0, z^\star) = 0$. Let EV_i denote $EV(\theta' \mid a = i)$. After some rearranging $z^\star$ satisfies

$$U(1, z^\star) - U(0, z^\star) = \delta [EV_0 - EV_1]. \tag{40}$$

The differences between current and future values balance, and (40) are solved as a non-linear equation. The condition that future expected values cannot depend on current z, avoids a more complicated condition. Each additional choice beyond binary adds another equation between adjacent values to be solved simultaneously.

Once $z^\star$ has been found, Bellman iteration requires calculation of the expected value of arriving at the (now explicit) state θ :

$$\begin{aligned} EV(\theta) = G(z^\star) &\left(E[U|a = 0, z < z^\star] + \delta EV_0\right) \\ + (1 - G(z^\star)) &\left(E[U|a = 1, z \geq z^\star] + \delta EV_1\right). \end{aligned} \tag{41}$$

These conditions are illustrated in Figure A1. Conditions (40)-(41) include three additional objects that the user's code must provide. The first condition for $z^\star$ needs $U(A(\theta), z)$, a vector of utilities for candidate values of z while solving for $z^\star$. The second includes $F(z^\star)$ and the vector of expected conditional utilities, $E[U(0) \mid z < z^\star]$ and $E[U(1) \mid z \geq z^\star]$. To use the `ReservationValues` solution method the user derives their model from the `OneDimensionalChoice` class. This class adds two virtual functions that other classes do not. the two functions are `Uz()` and `EUtility()`. At $z^\star$ the difference in current utility equals the discounted difference in future values. The user codes `Uz(z)` to return utility of all choices at z. The reservation value solution method uses that to solve for $z^\star$. Backward induction requires the expected utility of each option conditional given the optimal value $z^\star$, hence the need to provide `EUtility()`.

Algorithm A11. Reservation Wage Iteration

Initialization

Categorize each θ as an element of Θ_Z if reservation values are needed there. The default is yes unless the user provides a `Continuous()` method. If so, create storage for $z^\star$ at θ.

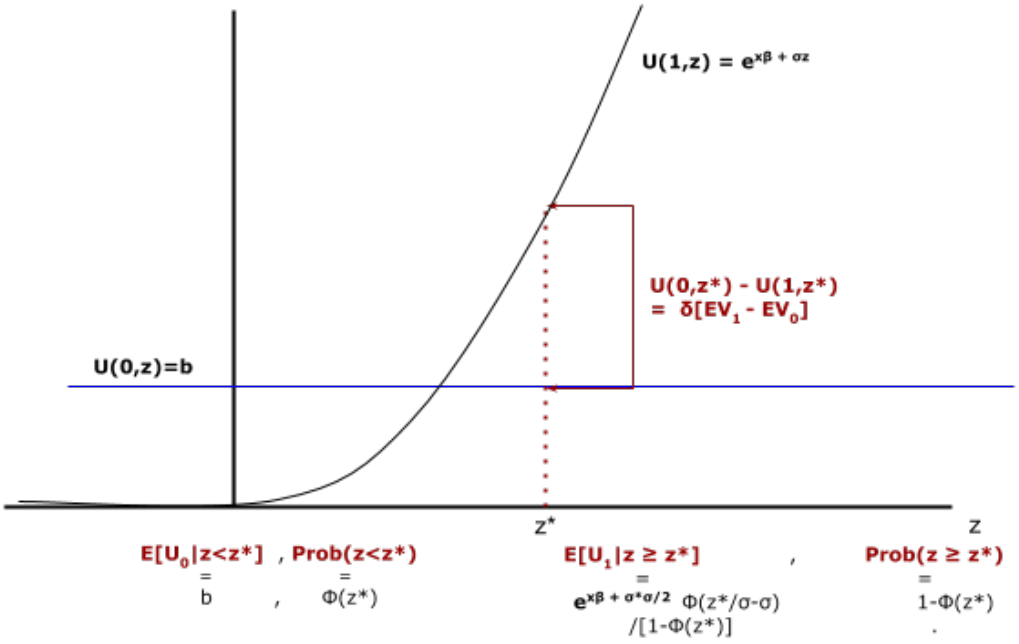
Iteration

Follow these steps at each t.

1. At each $\theta \in \Theta_Z$ initialize z solve for $z^\star$ based on the user-provided $Uz(A(\theta); z)$. When completed store $z^\star$ at θ. Use `EUtility()` that provides $F(z^\star)$ and $E[U|A(\theta), z^\star]$ to compute $V(\theta)$ based on (41).
2. For θ not in Θ_Z compute $V(\theta) = U(\alpha; \theta) + \delta EV(\theta')$ as in Bellman iteration but with a single action available.

Carry out update conditions for t as in Algorithm 2.

Figure A1. Conditions for Reservation Values



Converting the labor supply model to a reservation value model

For the basic labor supply model, the earnings shock e is a discretized normal. Change that assumption so it is a continuous standard normal random variable, re-labeled z. The class created for this version will be named `LSz`. Earnings are now written

$$E = \exp\{\beta_0 + \beta_1 M + \beta_2 M^2 + \beta_3 z\}.$$

Since E exceeds the value of not working (π) for large enough values of z there is a reservation value for the choice $m = 1$. Since `LSz` must be a one-dimensional choice model it cannot be derived from `LS` as was done earlier with `LSext`.

However, code from the basic model can be reused because most elements of `LS` are *static*. The only non-static member was `Utility()`. By using the static elements a change to `LS` will still be reflected automatically in `LSz`. In particular, `LSz::Uz()` can use `LS::Earn()` to utility at z. First set the value of `LS::e` to `z` then calls `LS::Earn()`.

In the model expected utility of not working is simply π. For working, expected utility involves $E[e^{\beta_3 z}|z > z^\star]$. The Mills-ratio formula for log-normality gives expected utility conditional of acceptance as

$$E[U|m = 1, z > z^\star] = \exp\left\{\beta_0 + \beta_1 M + \beta_2 M^2 + \frac{\beta_3^2}{2}\right\} \frac{\Phi\left(z^\star/\beta_3 - \beta_3\right)}{\Phi\left(z^\star\right)}. \tag{42}$$

The constant factor includes $\beta_3^2/2$, because it is coefficient on the normal random variable z (and hence the variance of the model shock). The ratio of $\Phi()$ values is new to the continuous specification and can't be borrowed from the discretized `LS`.

However, note that only the constant term includes the Mincer equation, and it is the same as the original earnings function if $e = \beta_3/2$. So again the base `LS::Earn()` function can be used by setting the value of `e` first. Thus, even though `LS` was coded for a completely different approach its specification is still synchronized with the reservation wage version. Only elements specific to the new version need to be coded.

This code segment converts the labor supply model to a continuous choice reservation value problem.

[V]

```
#include "LS.ox"

class LSz : OneDimensionalChoice {
    static Run();
    EUtility();
    Uz(z);
    }
LSz::Run() {
    Initialize(new LSz());
    LS::Build(d);
    CreateSpaces();
    RVSolve();
    ComputePredictions();
    }
LSz::Uz(z) {
    LS::e = z;
    return LS::pi | LS::Earn();
    }
LSz::EUtility()    {
    decl pstar = 1-probn(zstar),
         sig   = LS::beta[3];
    LS::e = sig/2;
 return {  ( LS::pi | LS::Earn()*probn((zstar/sig-sig)/pstar))
             , (1-pstar)~pstar
           };
 }
```

REFERENCES

[1] Aguirregabiria, Victor and A. Magesan 2013. "Euler Equations for the Estimation of Dynamic Discrete Choice Structural Models", *Advances in Econometrics 31*, 3-44.

[2] Aguirregabiria, Victor and Pedro Mira 2002. "Swapping The Nested Fixed Point Algorithm: A Class Of Estimators For Discrete Markov Decision Models," *Econometrica 70*, 4, 1519-43.

[3] ---- 2010. "Dynamic Discrete Choice Structural Models: A Survey," *Journal of Econometrics*, 156, 1 (May), 38-67.

[4] Aiyagari, S. Rao 1994. "Uninsured Idiosyncratic Risk and Aggregate Saving," *Quarterly Journal of Economics 109*, 3 (Aug), 659-684.

[5] Arcidiacono, Peter and Robert A. Miller 2011, "Conditional Choice Probability Estimation of Dynamic Discrete Choice Models With Unobserved Heterogeneity," *Econometrica 79*, 6, 1823-1867.

[6] Eckstein, Zvi and Kenneth Wolpin 1989. "The Specification and Estimation of Dynamic Stochastic Discrete Choice Models: A Survey," *Journal of Human Resources 24*, 4, 562-598.

[7] Ferrall, Christopher 2003. "Estimation and Inference in Social Experiments," manuscript, Queen's University working paper .

[8] Ferrall, Christopher 2005. "Solving Finite Mixture Models: Efficient Computation in Economics Under Serial and Parallel Execution," *Computational Economics 25*, 343-379.

[9] Hammermash, Daniel 2007. "Viewpoint: Replication in economics," *Canadian Journal of Economics 40*, 3, 715-733.

[10] Hotz, V. Joseph and Robert A. Miller 1993. "Conditional Choice Probabilities and the Estimation of Dynamic Models," *The Review of Economic Studies 60*, 3 (July), 497-529.

[11] Imai, Susumu, Neelan Jain, and Andrew Ching 2009. "Bayesian Estimation of Dynamic Discrete Choice Models," *Econometrica 77*, 6, 1865-1899.

[12] Kasahara, Hiroyuki and Katsumi Shimotsu 2012. "Sequential Estimation of Structural Models With a Fixed Point Constraint" *Econometrica 80*, 5, 2303-2319.

[13] Keane, Michael P. and Kenneth I. Wolpin 1994. "The Solution and Estimation of Discrete Choice Dynamic Programming Models by Simulation and Interpolation: Monte Carlo Evidence," *The Review of Economics and Statistics 76*, 4 (November), 648-672.

[14] Keane, Michael P. and Kenneth I. Wolpin 1997. "The Career Decisions of Young Men," *Journal of Political Economy 105*, 3 (June), 473-522.

[15] Keane, Michael P., Petra E. Todd, and Kenneth I. Wolpin 2010. "The Structural Estimation of Behavioral Models: Discrete Choice Dynamic Programming Methods and Applications," in *Handbook of Labor Economics, Volume 4a*, Orley Ashenfelter and David Card (eds), 331-461.

[16] Kirkby, R. A 2017. "Toolkit for Value Function Iteration," *Computational Economics 49*,1â€“15.

[17] Rosenzweig, Wolpin 1993. "Credit Market Constraints, Consumption Smoothing, and the Accumulation of Durable Production Assets in Low-Income Countries: Investment in Bullocks in India," *Journal of Political Economy 1993*, 101, 2, 223-44.

[18] Rust, John 1987. "Optimal Replacement of GMC Bus Engines: An Empirical Model of Harold Zurcher", *Econometrica 55*, 5 (September),, 999-1033.

[19] ----- 1997. "Using Randomization to Break the Curse of Dimensionality," *Econometrica 65*,3 (May), 487-516.

[20] Wolpin, Kenneth I. 1982. "An Estimable Dynamic Model of Fertility and Child Mortality," working paper, Yale Univ., October.

[21] ----- 1984. "An Estimable Dynamic Stochastic Model of Fertility and Child Mortality," *Journal of Political Economy 92*, 5 (October), 852-874.

[22] ----- 1987. "Estimating a Structural Search Model: The Transition from School to Work," *Econometrica 55* 4, 801-817.