Niplow: A Software Library for Predicting Social Experiments

by

Patrick Day

An essay submitted to the Department of Economics
in Partial fulfillment of the requirements for
the degree of Master of Arts

Queen's University
Kingston, Ontario, Canada

August 2009

Abstract

This paper presents a software library for predicting social experiments. The software is freely available and licensed under the GNU General Public License. Social experiments are modelled as a Discrete-choice Dynamic Programming problem. An overview of the social experiment mathematical model is given. Additionally, a detailed description of the software library is given as well as a sample application using Illinois Reemployment Bonus experiment.

Acknowledgements

Many thanks to my supervisor Christopher Ferrall, for his advice and time. Needless to say, this paper benefited greatly from his input and guidance.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

"First, structural econometric work is just very hard to do, simply in terms of the amount of labor involved. It often takes several years to write just one good paper in this genre, and this poses a daunting prospect for the young assistant professor seeking tenure, or the graduate student seeking to finish a dissertation." (Keane 2006, p.40)

Since the late 1980's, increases in computational power has allowed the estimation of more complicated structural models (Keane 2006). Available computational power is, however, only part of the problem. A recognized phenomenon is the rapid progress of hardware advances when compared to that of software (Brooks 1987). The difference is attributed to the greater complexity in developing software. In this regard software for estimating rich structural models has inherit complexity which slows the progress of development, including the concern of this paper: Discrete-choice Dynamic Programming problems (DDP hereafter).

Structural estimation's raison d'etre is to conduct policy experiments based on estimates of preferences and technology. Most real policy questions, however, are raised and answered in less time than required to formulate, program, and debug a sophisticated model, only to then compute consistent parameter estimates from data. When faced with real decisions, policy makers rarely have the patience to wait five years for an answer, which may not be definitive due to the many assumptions maintained to generate the answer. Hence policy experiments conducted in the DDP literature tend to be related to past policies or hypothetical questions and not to ones relevant to current debates. Consequentially, it is unsurprising that economists advising policy makers have failed to adopt the DDP paradigm for assessing labor market questions. Instead they use tools that provide straightforward answers in a timely fashion despite any shortcomings in the method. The development of code to streamline the development of estimated DDP models may alleviate this problem.

Although some researchers have released software so others can estimate DDP models (e.g. Rust (1987), Aguirregabiria and Mira (2002)), this area is still seen as a cottage industry. Students learn by writing their own code (or revising the advisor's code) to carry out the calculations. Despite the pedagogical value, this approach makes verification and extension of work difficult if not impossible, which undermines the main justifications for the effort reflected in Keane's quote. In addition, the issues raised will not be addressed by software that is a bare-bones program, expensive, or computationally inefficient.

This paper presents the Niplow software library for estimating social experiments, as well as, a model summary, a presentation of the user's experience, and a demonstration using the Illinois Reemployment Bonus experiment. The software is designed to automate elements of DDP that are common across models. The goal is to provide inexpensive (i.e. free) "turn-key" code that allows a user to focus on the details of their environment.

Niplow is an example of "object-oriented programming" (OOP hereafter), a software engineering technique for producing "reusable, extendible and reliable" software (Meyer 1988, p.vi preface). Although novel to the field of econometrics, OOP has not been "faddish". Instead, the technique has lead to many important developments, such as "design patterns" (Gamma, Helm, Johnson and Vlissides 1995), which describes recurrent software design problems along with their elegant solutions. Regrettably, many economists continue to use the procedural style of FORTRAN 77, which strongly discourages an OOP approach.

Currently, policy-related empirical work is expected to be based on exogenous variation in the economic environment. The techniques that satisfy this demand include randomized field trials or experiments (Burtless 1995), natural experiments (Rosenzweig and Wolpin 2000), and (valid) instrumental variables estimation (Angrist 1990). Only recently have DDP models tried to contribute to this standard either by applying techniques to experimental data (e.g. Todd and Wolpin 2006, Lise, Seitz and Smith 2003, Ferrall 2000), or comparing predictions of structural versus IV techniques (Keane 2006). The Niplow software library builds on Ferrall's (2002) framework for applying DDP to environments that include unexpected (to the subject) changes in policy. While the ideal situation is a randomized social experiment, the framework applies equally as well to natural experiments that occur to forward-looking subjects.

2 Experiments and Forward-Looking Behaviour

In modelling social experiments, if only random selection is considered exogenous, then the mean outcome in the participants is the unbiased mean outcome of the population. If, however, a model does not account for large heterogeneity effects in selection and treatment, then the mean outcome for the population cannot be determined without bias. For instance, the treatment effects of an experiment is not a good predictor of another similar but different experiment if individual characteristics including pre-treatment characteristics are not taken into account (Hotz, Imbens, Mortimer and Center 1999).

In the realm of natural experiments, the military draft lottery in the United States was thought of as a "perfect" instrument for estimating the impact of military service on future earnings for the population. Keane (2006), however, demonstrated the possibility of unobserved heterogeneity in the benefits of military service causes the lottery to be an invalid instrument. Instead of the entire population, the lottery can only be a valid instrument to determine the average effect on future earnings of men who would not serve in the military unless drafted (Angrist and Krueger 2001). The lottery is not a valid instrument for those who would volunteer for military service regardless of the draft. This phenomenon is known as the Local Average Treatment Effect (Imbens and Angrist 1994). Moreover, after reviewing twenty studies of the use of natural experiments as instruments, Rosenzweig and Wolpin (2000) conclude natural experiments reduce the number of interpretations of the estimates but do not produce conclusive interpretations due to a lack of theoretical grounding.

Ferrall's (2002) model, which is implemented by the Niplow software library, endogenously models both observed and unobserved heterogeneity in selection and treatment effects so mean outcomes for the population can be estimated without bias. The model is structural in nature and is similar to Rust's (1996) discrete decision processes which are a subclass of Markovian decision processes. Individual's choices are modelled as a DDP problem which captures the choices of forward-looking agents.

Dynamic programming models (Bellman 1966) have been used extensively in econometrics literature. The applications range from how social security effects work versus retirement choices in men (Rust and Phelan 1997) to the analysis of a randomized experiment assessing the impact school subsidies on the parental choice between fertility and child schooling (Todd and Wolpin 2006). A survey by Wolpin (1996), which focuses on the use DDP in public policy, describes applications in welfare, education, employment, agricultural development and industrial policy.

An issue with dynamic programming models is that they suffer from the “curse of dimensionality”, that is, computational time for estimation increases exponentially with the number of dimensions. In computer science nomenclature, this type of intractability is known as a “NP complete” problem for which there is no known general solution. Strategies have been suggested to help alleviate the issue.

A Bayesian approach for solving DDP models, known as the IJC algorithm, has two features that improves tractability (Ching, Imai, Ishihara and Jain 2009). First, unlike the conventional Nested Fixed Point algorithm (Rust 1987), the IJC algorithm uses information from previous iterations to reduce each iteration’s computational cost. Second, the IJC algorithm directly produces the posterior distribution of parameter vectors, and the corresponding solutions for the DDP model so searching the likelihood function for the global maximum is unnecessary.

An interpolation strategy has been suggested by Keane and Wolpin (1994a) as well as Howitt, Msangi, Reynaud and Knapp (2002) who suggest using Chebechev polynomials in particular. Unfortunately, interpolation helps but does not solve the “curse of dimensionality”. This is noted by Rust (1997) who employs a random algorithm to break the “curse” for a subclass of dynamic programming problems.

3 Predicting Impacts of Social Experiment

In this section a mathematical description of a social experiment is given followed by different aspects of the Niplow software library such as the Ox programming language and algorithms used. Finally a detailed description is given of what user input is required for Niplow to predict a social experiment.

3.1 The Social Experiment Model

This section summarizes Ferrall's (2002) model using slightly different notation. Only key equations are provided and some elements of the model are introduced and explained without repeating explicit expressions available in Ferrall's (2002) paper. Future versions will of Niplow will fully support all the model features described (i.e. GMM parameter estimation), but the current version is not full featured.

3.1.1 The Experimental Environment

The full state of a subject in the experiment is contained in a vector $\theta \in \Theta$ where Θ is the state space. The elements of θ are shown in Equation 1 along with a brief description of their roles in Table 1.

$$\theta = \begin{bmatrix} d & k & e & g & f & r & t & s_1 & \cdots & s_n \end{bmatrix} \tag{1}$$

The variables $s_1 \cdots s_n$ are defined by the user as part of their model of behaviour. The other variables have interpretations related to the overall framework. User provided definitions determine the range of variable values.

State variables d through f vary over integers from zero up to their cardinality less one. This definition corresponds to array indexing in Ox, which like C, is zero-based. Therefore, user code can use a state variable as an index into an array. The current period r has cardinality that depends on the current phase f. Each phase of treatment has a maximum length, $R(f)$, because experimental treatment is assumed to be finite-lived.

Table 1: Required State Variables

Variables	Description	Cardinality (User-Defined)	Examples
d	observed type	D	married males
k	unobserved type	K	finite mixture type
e	entry group	E	
g	treatment group	G	treatment or control
f	experiment phase	F	in training
r	current period of phase	$R(f)$	2^{nd} period of training
t	time	$t_{\max}(e) - t_{\min}(e) + 1$	periods until or since randomization

For d, k and e, Niplow simply needs to know how many different values to loop over during the solution and estimation stages. The particular values of d, k and e have no special meaning. Other state parameters, however, do have values with specific meanings.

For experimental group, the control group is defined as $g = G-1$ and all treatment groups have $g < G-1$. Phases of treatment f also have some special values. The first (i.e. $f = 0$) and the last (i.e. $f = F-1$), are *pre-experiment* and *post-experiment* phases, respectively. Thus, subjects assigned to group $G-1$ move from phase 0 to phase $F-1$ directly. Subjects in a treatment group spend time in some or all of the treatment phases before reaching $F-1$.

The time variable t has the special purpose of coordinating measurement of different entry and treatment groups across the experiment. The cardinality of t is implicitly defined for each entry group e by the definitions of $t_{\max}(e)$ and $t_{\min}(e)$.

Besides the cardinality of the required state variables, the user supplies other parameters that are not state variables but are dependent on them. These variables are summarized in Table 2. Note that $\sum_k \lambda(d,k) = 1$ for a given demographic group d.

State variables d, k, e and g are *invariant* for a subject: the probability they change is zero. This means Niplow loops over the values automatically. As explained

Table 2: Experiment and Population Parameters

Variable	Description
$t_{\min}(e)$	Initial observation period
$t_0(e)$	Randomization period
$t_{\max}(e)$	Last possible observation period
$f_0(e)$	initial phase of group e
$\lambda(d,k)$	population proportion of k given d (estimated)

below, the user will specify how their state variables s_i evolve over time for a given subject. In addition, the user specifies transition rules for f and r. These transitions must make treatment *progressive*: the phase cannot go backwards, and within a phase r counts up. For example, let $F = 4$ (i.e. $f \in \{0 \ldots 3\}$) and suppose the current phase is $f = 2$ and the phase next period is f'. Then transitions $f' = 0$ or $f' = 1$ are invalid. For period transitions, if $f' > f$ then the only valid transition for r is $r' = 0$, otherwise the phase is unchanged (i.e. $f' = f$) and the period is incremented, that is, the next period is $r' = r+1$. Together these restrictions guarantee that treatment is finite. The subject's forward-looking behavior during treatment can be solved working backwards starting from $f = F - 2$ and $r = R[F - 2]$.

Niplow allows the user to define early ends to treatment in that the transition to phase $F - 1$ can happen anytime during treatment. For example, suppose the experiment involves training (the treatment) offered to unemployed workers. Furthermore, the experiment is designed so that taking a job makes the subject ineligible for additional training. In implementing the model, the user will instruct Niplow that such a choice results in $f' = F - 1$. Subjects who do not take a job stay in training until the last (finite) period before an automatic end.

3.1.2 Subject Behaviour

Subject behaviour, both outside the experiment and during treatment, is based on an infinite horizon dynamic programming problem. The user provides the model elements, which can depend on unknown "structural" parameters to be estimated using the tools in Niplow. These elements, including a brief description are given in Table 3.

Table 3: Elements of Behaviour and Measurement

Variable	Description
$\mathbf{A}(\theta)$	feasible action sets
$U(\alpha, \theta)$	utility
$\Pr\{\theta'\|\alpha, \theta\}$	state transition
$Y(\alpha, \theta)$	vector of observations
$\delta(k)$	time discount factor for type k (estimated or fixed)
$\rho(k)$	smoothing factor for choice probabilities (estimated or fixed)

The action of a subject in a period is denoted by the vector α. Together (α, θ) define an "outcome". Not all actions may be possible in a particular state. At each state actions are chosen from the user-defined *feasible* set $\mathbf{A}(\theta)$.

The user provides $U(\alpha, \theta)$, the one period utility function of the subject given the current outcome, and $\Pr\{\theta'|\alpha, \theta\}$, the transition or law of motion of the state. This is the probability that the state next period is θ' given the current outcome. By using $\Pr\{\theta'|\alpha, \theta\}$, the user can specify endogenous transitions to subsequent treatment phases.

The value of an outcome is

$$\begin{aligned} v(\alpha, \theta) &= U(\alpha, \theta) + \delta\,\mathrm{E}[V(\theta')] \\ &= U(\alpha, \theta) + \delta \sum_{\theta'} \Pr\{\theta'|\alpha, \theta\} V(\theta'), \end{aligned}$$

where $V(\theta')$ is the indirect value of state θ' given optimal choice:

$$V(\theta) = \max_{\alpha \in \mathbf{A}(\theta)} v(\alpha, \theta) \quad \forall \theta \in \Theta.$$

Combining $v(\alpha, \theta)$ and $V(\theta)$ results in Bellman's Equation (2) for the DDP.

$$V(\theta) = \max_{\alpha \in \mathbf{A}(\theta)} \left[U(\alpha, \theta) + \delta \sum_{\theta'} \Pr\{\theta'|\alpha, \theta\} V(\theta') \right] \tag{2}$$

It is worth noting that, while this formulation is now standard, in applications researchers usually compute (2) by writing their own programs. Although there is publicly available code (e.g. Rust 1987, Aguirregabiria and Mira 2002), it is written in Gauss which is not free for academic use and not known for speed. One innovation is that using Ox, which is free for academic research, users are only required to program the primitive elements of their model. Niplow then carries out all the necessary computations. In addition, Niplow accounts for key extensions of standard frameworks: unexpected randomization, endogenous selection into eligibility for randomization, and both permanent (type-specific) and transitory unobserved states.

Typically in empirical applications of DDP, $U(\alpha,\theta)$ is made a random variable with infinite support. For example, Rust (1987) extends McFadden's static random utility model by including an additive extreme value term in $U(\alpha,\theta)$. Keane and Wolpin (1994b) include a multivariate normal error term to allow for cross-choice correlations. Either specification is designed to make choice probabilities smooth functions of underlying parameters and to rule out any (feasible) choice having zero probability. This randomness becomes structural, because current choices account for future randomness through $V(\theta)$. Niplow, however, follows Eckstein and Wolpin's (1999) approach by not adding a random term to $U(\alpha,\theta)$. Instead, the deterministic $v(\alpha,\theta)$ is computed without building error into $V(\theta)$. The choice probabilities are then smoothed using a logistic kernel with parameter $\rho(k)$:

$$\Pr\{\alpha|\theta\} = \frac{\tilde{v}(\alpha,\theta)}{\sum_{\alpha'}\tilde{v}(\alpha',\theta)} \quad \text{where}$$

$$\tilde{v}(\alpha,\theta) = \begin{cases} e^{\frac{\rho}{1-\rho}[v(\alpha,\theta)-V(\alpha,\theta)]} & \text{if } \alpha\in\mathbf{A}(\theta) \\ 0 & \text{otherwise.}\end{cases}$$

Note that $0 \le \rho < 1$. As $\rho \to 0$ then $\Pr\{\alpha|\theta\} \to \frac{1}{|\mathbf{A}(\theta)|}$, that is, each choice becomes equally likely, and value has no bearing. Also, as $\rho \to 1$ then $\Pr\{\alpha|\theta\} \to \arg\max_\alpha v(\alpha,\theta)$, so only optimal actions (based on the deterministic utility) are taken as choices probabilities are no longer smooth.

3.1.3 Measurement and Endogenous Sample Selection

Until recently most applications of DDP assumed that the econometrician could observe θ up to the value of estimated parameters, and in some cases up to the value of permanent unobserved type k. In these cases, the model choice probabilities can be used to form the likelihood for a sample of outcomes. While it is often desirable to provide for states that are not directly observable, maximum likelihood estimation becomes difficult because it requires integrating out the distribution of unobserved states at each point in time.

This framework assumes the model will include unobserved states and so uses an estimation method that is feasible under that assumption. Namely, the user specifies a function of the outcome, $Y(\alpha, \theta)$, that returns a vector of observations. The vector corresponds to data that the user wishes to use for estimation via GMM. Measurements do not include d, e, g or t, because these are required to be observed aspects of θ. The expected measurement $E[Y|\theta]$ in state θ is,

$$E[Y|\theta] = \sum_{\alpha \in \mathbf{A}(\theta)} Y(\alpha, \theta) \Pr\{\alpha|\theta\}$$

The model assumes that a population of potential subjects exists and makes choices according to the DDP. Over time their choices and current states determine transitions to states next period. The state-to-state probabilities $P_s(\theta'|\theta)$ are found by combining the primitive transitions $\Pr\{\theta'|\alpha, \theta\}$ with the smoothed choice probabilities $\Pr\{\alpha|\theta\}$:

$$P_s(\theta'|\theta) = \Pr\{\theta'|\theta\} = \sum_{\alpha \in \mathbf{A}(\theta)} \Pr\{\theta'|\alpha, \theta\} \Pr\{\alpha|\theta\}$$

The environment outside the experiment is required to be ergodic, in the sense that all states are reachable by any other state through repeated application of P_s. This means that, starting with no other information about past choices, potential subjects of the experiment are distributed across states according to the stationary (ergodic) distribution. This distribution, denoted $P_{-\infty}(\theta)$, is defined by,

$$P_{-\infty}(\theta) = \Pr\{\theta\} = \sum_{\theta'} P_s(\theta'|\theta)P_{-\infty}(\theta) \quad \text{for } \forall\theta \in \Theta$$

In many experiments, natural or otherwise, the sample is not a random sample of the entire population. Typically, eligibility for randomization is conditional on past choices that are often similar to the behaviour the experiment alters. Therefore, accounting for endogenous selection is important when making policy recommendations from the experiment, even when the conditional randomization is "clean".

An eligible subject has a panel of pre-randomization measurements that meet requirements defined by the user through the boolean function $\mathcal{H}[y; d, e, t]$. For instance, the subject is still eligible for random selection into entry group e at period $t \leq t_0(e)$ if $\mathcal{H}[y; d, e, t] = 1$ for measurement vector y.

Starting from $P_{-\infty}(\theta)$ at $t = t_{min}(e)$ updates to the distribution over states occurs by imposing the selection transition:

$$P^{\star}(\theta'|\theta) = \Pr\{\theta'|\theta\} = \sum_{\alpha\in\mathbf{A}(\theta)} \mathcal{H}[Y(\alpha,\theta); t, e, d] \Pr\{\theta'|\alpha,\theta\} \Pr\{\alpha|\theta\}$$

Selection continues sequentially until $t_0(e)$. That is, during sample selection the probability of states in the next period only includes current outcomes (α, θ) that generate a feasible $Y(\alpha, \theta)$. The software sequentially renormalizes the distribution each period to account for this attrition, until $t = t_0(e)$. The resulting distribution, denoted $\Omega_0(\theta|e, d, k)$, is the selected sample at the point of randomization, which occurs at the end of period $t_0(e)$, after choices are made but before the next state is realized. For a given entry group, $\Omega_0(\theta|e, d, k)$ is common to all treatment groups g because they are assumed to be assigned randomly just before the start of period $t_0(e) + 1$.

The distribution $\Omega_0(\theta|e, d, k)$ controls for selection on endogenous states because it depends on unobserved type k which affects choice probabilities. The population weights $\lambda[k, d]$ are updated to control for permanent heterogeneity. The endogenous proportion of unobserved types within each demographic group d is different than the underlying (unselected) population proportion. At randomization the endogenous proportion is denoted $\lambda^\star(k; t, g, e, d)$.

3.1.4 Matching Predictions to Data

The model can be "fit" to data by estimating the exogenous parameters that give the smallest discrepancy. Estimation is accomplished by using GMM to match moments from the prediction with those from the observed data (i.e. averages).

There are several reasons for choosing to match moments rather than using maximum likelihood to estimate the parameters (Ferrall 2002). First, the maximum likelihood estimates may be difficult to compute, especially if the subject makes choices using more information than is being empirically measured. In this case, costly computation must be undertaken to integrate out those choices. Second, differences between the modelled and actual experiment can give zero probability to observed measurements. Lastly, only averaged data maybe available to the researcher as data on individuals may be restricted due to privacy laws.

Let $\hat{Y}$ be a vector of observed averaged measurements (i.e. averaged data) conditional on t, g, e, and d. The empirical impact is then the difference between the mean observed outcome of the treatment groups and the control group:

$$\hat{\Delta}(t, g, e, d) = \mathrm{E}[\hat{Y}|t, g, e, d]\big|_{g<G-1} - \mathrm{E}[\hat{Y}|t, g, e, d]\big|_{g=G-1}$$

Correspondingly, the predicted mean outcome and impact of the model is,

$$\begin{aligned}\mathrm{E}[Y|t, g, e, d] &= \sum_{\theta} \lambda^\star(k; t, g, e, d)\Omega\{\theta|k, t, g, e, d\}\,\mathrm{E}[Y|\theta] \\ \Delta(t, g, e, d) &= \mathrm{E}[Y|t, g, e, d]\big|_{g<G-1} - \mathrm{E}[Y|t, g, e, d]\big|_{g=G-1}\end{aligned}$$

Denote $\bar{\Delta}$ as the difference in mean outcomes between the data and the model, that is,

$$\bar{\Delta}(t, g, e, d) = \mathrm{E}[\hat{Y}|t, g, e, d] - \mathrm{E}[Y|t, g, e, d]$$

Let θ_{exog} be exogenous model parameters that the user wishes to be estimated. These parameters can include λ, ρ, and δ, as well as, user defined model parameters (e.g. those governing utility). The discrepancy between the data and the model is given in Equation (3). Note that $A(t, g, e, d)$ is a positive definite matrix.

$$Z(\theta_{\text{exog}}) = \sum_{d=0}^{D-1} \sum_{e=0}^{E-1} \sum_{g=0}^{G-1} \sum_{t=t_0(e)}^{t_{\max}(e)} \bar{\Delta}(t, g, e, d)' A(t, g, e, d) \bar{\Delta}(t, g, e, d) \tag{3}$$

Let the impact based estimates of the model parameters be denoted as $\theta_{\text{exog}}^{\text{IE}}$ which is the solution to minimizing $Z(\theta_{\text{exog}})$, that is,

$$Z(\theta_{\text{exog}}^{\text{IE}}) = \underset{\theta_{\text{exog}}}{\arg\min}\, Z(\theta_{\text{exog}})$$

The estimate of $\theta_{\text{exog}}^{\text{IE}}$ will be consistent but inefficient. A consistent estimate of the covariance of the moments can be computed by simulation using $\theta_{\text{exog}}^{\text{IE}}$ as the model parameters. With the consistent covariance of the moments, a second stage of GMM can be estimated to produce a more efficient estimate of $\theta_{\text{exog}}^{\text{IE}}$.

3.2 Software Library

Niplow is “open source” in that it is freely available and can be found at `https://qshare.queensu.ca/Users01/ferrallc/public/Niplow/`. Niplow is licensed under the GNU General Public License (hereafter GPL) and although the software is free, the GPL does place some responsibilities on user. The accompanying file, “license.txt”, describes the licensing terms in detail. The GPL has many advantages such as the formation of developer communities and the “taming of complexity” (Raymond 1999).

3.2.1 The Ox language

Niplow was developed using the econometric programming language called Ox (Doornik and Ooms 1998). Ox, which was first released in 1996 as version 1.0, has a thirteen year history and is currently at version 5.10. It has been described as a language which is dynamically type, object-oriented, has a native matrix data-type and a syntax which is close to C, C++ or Java. There are several advantages to using Ox.

One of the main advantages of Ox is its speed. When compared with GAUSS or S-PLUS, two other popular statistical languages, Ox proves to be the fastest (Cribari-Neto 1997). For applications that require additional speed optimizations, the computational bound portion of code can be implemented in C and called seamlessly within Ox. Also, Ox has an already existing library of econometric functions, that range from ARMA(p, q) forecasting to estimation of dynamic panel data. Additionally, Ox has incorporated programming language concepts which increase expressiveness such as the objected oriented paradigm. The advice given by Kendrick and Amman (1999) is that junior economists should learn a high level econometric language and work towards learning a low level language such as Fortran or C. In this regard, the syntax similarity between Ox and C eases the effort of learning C.

3.2.2 Object Oriented Programming

Niplow was developed using object-oriented techniques. With Ox, objects are defined by classes which encapsulates both data (members hereafter) and functions (methods hereafter) for the particular object. Encapsulation hides implementation details of an object while providing a public interface to the object. Objects or instances are then created or “instantiated” when the class “constructor” method is called with the `new` operator. With languages such as Ox or C++, the constructor has the same name as the name of its class. The purpose of the constructor is to initialize the members before the object is used elsewhere.

Another feature of object oriented programming is inheritance which allows a class to be defined as a subclass of one or more base-classes. A subclass has all the non-private methods and members as the base-class plus any members and methods that are particular to it. Ox currently does not support `private` class declaration and so all members and methods are inherited by subclasses.

To clarify, in the example below A is the base-class and B the subclass. A has two methods: the public method `foo()`, and the protected method `bar()`. Public members or methods can be accessed by any function. Protected members can only be accessed by methods of the same object. Note that in Ox data members are protected and methods are public by default (i.e. in absence of `public:` or `protected:` declarations).

Returning to the example, class B inherits both `foo()` and `bar()` from A. In addition, B has a method `baz()` which is particular to it but not A.

```
class A {
public:
   foo ();
protected:
   bar ();
}

class B : A {
   baz ();
}
```

Another property of inheritance is method "overriding". This is when an inherited method is reimplemented in the subclass. A further extension is a `virtual` method which is declared and optionally implemented in the base-class. The virtual method and can be called by other base-class methods even if it is not implemented in the base-class. Method overriding is an important concept because users need to override key methods in the inherited `SocialExperiment` base-class in order to implement a social experiment. In the following example `foo()` is declared as a virtual method in class A and overridden in class B.

```
class A {
   virtual foo ();
}

class B : A {
   foo ();
}
```

3.2.3 Algorithms

Only an overview of the main algorithms used in the software library is given as several helper functions occur in the pseudo code but are not defined (e.g. Enumerate(θ)). Also, memory management, due to the large number of states, is an issue but is ignored here. The algorithms are represented using a generalized pseudo-code rather than Ox as mathematical notation can be used for conciseness.

Given a model with n endogenous state parameters, a particular vector of endogenous state parameters is represented by $s = [s_1 \dots s_n]$. The state space of s is $\mathcal{S}$, that is $s \in \mathcal{S}$. For brevity, the symbols D, K, E, G, F, $R[f]$, Φ, $\mathbf{A}(\alpha, \theta)$, $P(\theta' | \alpha, \theta)$, $U(\alpha, \theta)$, $Y(\alpha, \theta)$, $\mathcal{H}[y; t, g, e, d]$, $\rho(k)$, $\delta(k)$, $f_0(e)$, and $\lambda(d, k)$ are considered global to all functions. All other symbols (i.e. EY) are local to the procedure that they occur in. Note that comments are displayed in enclosing "{}". Also, tuples are "unpacked" with assignment statements. For example, if a function Foo() returns a two-tuple, the statement $x, y \leftarrow$ Foo() will unpack the return value, such that x and y are assigned the first and second element respectively.

The top level function is Predict-Social-Experiment which corresponds to `Predict::run()` from file predict.ox. For a given entry group, the endogenous proportion of the unobserved type k at the end of $t_0(e)$ is $\lambda^\star$ which in mathematically defined as,

$$\lambda^\star(k; t, g, e, d) = \frac{\lambda[d, k] \cdot \omega_0(k; t, e, d)}{\sum_{k=0}^{K-1} \lambda[d, k] \cdot \omega_0(k; t, e, d)}$$

Algorithm 1 Predict-Social-Experiment()

```
for $d \leftarrow D-1$ to 0 do
  for $k \leftarrow K-1$ to 0 do
    $P_{-\infty}, P_\alpha[k], P_s[k], \mathrm{Ey}[k] \leftarrow$ Solve-Behaviors$(d, k, \delta[k], \rho[k])$
    $\Omega_0[k], \omega_0[k] \leftarrow$ Solve-Entry-Group-Endog-Sample$(d, k, P_{-\infty}, P_\alpha[k], y[k])$
    $\lambda^\star[k] \leftarrow \lambda[d,k] \cdot \omega_0[k]$ {NB: $\mid \lambda^\star[k] \mid = E$}
  end for
  for $e \leftarrow E-1$ to 0 do
    $\lambda^\star_{\text{sum}} = \sum_{k \in \{0 \dots K-1\}} \lambda^\star[k][e]$
    for $k \leftarrow K-1$ to 0 do
      $\lambda^\star[k][e] \leftarrow \lambda^\star[k][e]/\lambda^\star_{\text{sum}}$ {Ensure proportions for entry groups $\in [0 \dots 1]$}
    end for
  end for
  $\mathrm{EY}[d] \leftarrow$ Compute-Expected-Outcomes$(d, \Omega_0, \omega_0, \lambda^\star, P_\alpha, P_s, \mathrm{Ey})$
end for
return EY
```

The Solve-Entry-Group-Endog-Sample($\cdot$) algorithm implements the pre-assignment period of endogenous sample selection. The corresponding Niplow implementation is `Predict::solveEntryGroupEndogSample(·)`. $\Omega(\theta'|t,e,d,k)$ is the distribution across states and $\omega(k;t,e,d)$ is the cumulative proportion of unobserved type k surviving to time t. At the start of the data generating process for entry group e (i.e. $t_{\min}(e)$) the distribution across states is the ergodic distribution. The initial values of $\Omega(\cdot)$ and $\omega(\cdot)$ are,

$$\Omega(\theta'|t,e,d,k) \mid_{t=t_{\min}(e)} = P_{-\infty}(\theta)$$

$$\omega(k;t,e,d) \mid_{t=t_{\min}(e)} = 1$$

At $t_{\min}(e)+1$ the distribution of those still eligible for entry into the experiment is no longer stationary. Both $\Omega(\cdot)$ and $\omega(\cdot)$ are calculated recursively and for $t < t_0(e)$ they are updated as follow:

$$\omega(k;t,e,d) = \omega(k;t,e,d) \mid_{t-1} \left[\sum_{\theta'} \sum_{\theta} P^\star(\theta'|\theta) \cdot \Omega(\theta'|t,e,d,k) \mid_{t-1} \right]$$

$$\Omega(\theta'|t,e,d,k) \mid_{t=t_{\min}(e)} = \frac{\omega(k;t,e,d) \mid_{t-1}}{\omega(k;t,e,d)} \left[\sum_{\theta} P^\star(\theta'|\theta) \cdot \Omega(\theta'|t,e,d,k) \mid_{t-1} \right]$$

The values returned by Solve-Entry-Group-Endog-Sample($\cdot$) are Ω_0, ω_0, the distribution of the selected sample and the proportion of unobserved type k surviving at the point of randomization, respectively.

Algorithm 2 Solve-Entry-Group-Endog-Sample($d, k, P_{-\infty}, P_\alpha, y$)

```
$g \leftarrow G - 1$ {Set $g$ to control group}
$r \leftarrow 0,\ f \leftarrow F - 1$
for $e \leftarrow E - 1$ to 0 do
    $\Omega[e] \leftarrow P_{-\infty}$
    $\omega[e] \leftarrow 1$
    for $t \leftarrow t_{\min}[e]$ to $t_0[e]$ do
        for all $s \in \mathcal{S}$ do
            $\theta \leftarrow [d, k, e, g, f, r, t] \,|s$ {Append endogenous parameters, $s$}
            $i \leftarrow$ Enumerate($\theta$)
            $P^\star[i] \leftarrow \sum_{\alpha \in \mathbf{A}(\theta)} \mathcal{H}(y[i, \alpha]; t, e, d) \cdot P_\alpha[i, \alpha] \cdot P(\theta'|\alpha, \theta)$ {$P^\star \equiv P^\star(\theta'|\theta)$}
        end for
        $\Omega_{\text{next}} \leftarrow \omega[e] \cdot \sum_\theta P^\star[\text{Enumerate}(\theta)] \cdot \Omega[e][\text{Enumerate}(\theta)]$ {sum over $\theta$}
        $\omega[e] \leftarrow \sum_{\theta'} \Omega_{\text{next}}[\text{Enumerate}(\theta')]$
        $\Omega[e] \leftarrow \Omega_{\text{next}}/\omega[e]$ {Renormalize so dist$^{\text{n}}$ sums to 1}
    end for
end for
return $\Omega, \omega$
```

The next algorithm computes the predicted treatment outcomes and is implemented as `Predict::computeExpectedOutcomes(·)`. Note that during the post-assignment period, $t_0(e) < t \leq t_{\max}(e)$, $\Omega(\cdot)$ and $\omega(\cdot)$ continue being recursively updated as defined in equations (4) and (5). The difference in definition with the pre-assignment period is that $\mathcal{H}[\cdot]$ is not used because sample selection has been completed and so $P^\star(\theta'|\theta)$ is replaced with $P_s(\theta'|\theta)$.

$$\omega(k; t, g, e, d) = \omega(k; t, g, e, d) \mid_{t-1} \left[\sum_{\theta'} \sum_{\theta} P_s(\theta'|\theta) \cdot \Omega(\theta'|t, g, e, d, k) \mid_{t-1} \right] \tag{4}$$

$$\Omega(\theta'|t, g, e, d, k) = \frac{\omega(k; t, g, e, d) \mid_{t-1}}{\omega(k; t, g, e, d)} \left[\sum_{\theta} P_s(\theta'|\theta) \cdot \Omega(\theta'|t, g, e, d, k) \mid_{t-1} \right] \tag{5}$$

Also note that both $\Omega(\cdot)$ and $\omega(\cdot)$ are now dependent on g.

Algorithm 3 Compute-Expected-Outcomes($d, \Omega, \omega, \lambda^\star, P_\alpha, P_s, \mathrm{Ey}$)

```
$r \leftarrow 0$
**for** $e \leftarrow E-1$ to $0$ **do**
    $f \leftarrow f_0[e]$
    **for** $g \leftarrow G-1$ to $0$ **do**
        tick $\leftarrow 0$
        **for** $t \leftarrow t_0[e]$ to $t_{\max}[e]$ **do**
            $\mathrm{EY}[e][g][\mathrm{tick}] = [0 \ldots 0]$
            **for** $k \leftarrow K-1$ to $0$ **do**
                **for all** $s \in \mathcal{S}$ **do**
                    $\theta \leftarrow [d, k, e, g, f, r, t]\,|s$ {Append endogenous parameters, $s$}
                    $i \leftarrow \mathrm{Enumerate}(\theta)$
                    $\mathrm{EY}[e][g][\mathrm{tick}] \leftarrow \mathrm{EY}[e][g][\mathrm{tick}] + \lambda^\star[k][e] \cdot \Omega[k][e][i] \cdot \mathrm{Ey}[k][i]$
                **end for**
                $\Omega_{\mathrm{next}} \leftarrow \omega[k][e] \cdot \sum_{\theta'} P_s[k][\mathrm{Enumerate}(\theta')] \cdot \Omega[k][e][\mathrm{Enumerate}(\theta')]$
                $\omega[k][e] \leftarrow \sum_{\theta'} \Omega_{\mathrm{next}}[\mathrm{Enumerate}(\theta')]$
                $\Omega[k][e] \leftarrow \Omega_{\mathrm{next}}/\omega[k][e]$ {Renormalize so dist$^{\mathrm{n}}$ sums to 1}
            **end for**
            tick $\leftarrow$ tick $+ 1$
        **end for**
    **end for**
**end for**
**return** EY
```

Algorithms 4 through 8 are implemented in behaviour.ox as the following methods respectively:

```
Behaviour::solve(const delta, const rho)
TreatmentBehaviour::solve(const delta, const rho, const realityV)
RealityBehaviour::solve(const delta, const rho)
RealityBehaviour::solveInfiniteHorizon(const delta)
RealityBehaviour::solveErgodicDistn(const nextStateProb)
```

Algorithm 4 Solve-Behaviors(d, k, δ, ρ)

```
$P_{-\infty}, V \leftarrow$ Solve-Reality-Behavior$(d, k, \delta, \rho)$
$P_\alpha, P_s, y \leftarrow$ Solve-Treatment-Behavior$(d, k, \delta, \rho,$ Expand-For-All-States$(V))$
**return** $P_{-\infty}, P_\alpha, P_s, y$
```

Algorithm 5 Solve-Treatment-Behavior$(d, k, \delta, \rho, V_{\text{reality}})$

```
for $e \leftarrow E - 1$ to 0 do
  for $g \leftarrow G - 1$ to 0 do
    $V_{\text{tomor}} \leftarrow V_{\text{reality}}$
    for $f \leftarrow F - 1$ to 0 do
      for $r \leftarrow R[f] - 1$ to 0 do
        for all $s \in \mathcal{S}$ do
          $\theta \leftarrow [d, k, e, g, f, r, t] \,|s$
          $i \leftarrow \text{Enumerate}(\theta)$
          $v \leftarrow U(\alpha, \theta) + \delta \sum_{\theta'} P(\theta'|\alpha, \theta) V_{\text{tomor}}[\text{Enumerate}(\theta')]$
          $V_{\text{today}}[i] \leftarrow \max_{\alpha \in \mathbf{A}(\theta)} v[\alpha]$
          $P_\alpha[i] \leftarrow \text{Compute-Choice-Prob}(\rho, v, V_{\text{today}}[i], \theta)$ {NB: $P_\alpha \equiv \Pr\{\alpha|\theta\}$}
          $P_s[i] = \sum_{\alpha \in \mathbf{A}(\theta)} P_\alpha[i, \alpha] P(\theta'|\alpha, \theta)$ {NB: $P_s \equiv P_s(\theta'|\theta)$}
          $\text{Ey}[i] \leftarrow \sum_{\alpha \in \mathbf{A}(\theta)} P_\alpha[i, \alpha] \cdot Y(\alpha, \theta)$ {NB: $\text{Ey} = E[Y|\theta]$}
        end for
      end for
    end for
  end for
end for
return $P_\alpha, P_s, \text{Ey}$
```

Algorithm 6 Solve-Reality-Behavior(d, k, δ, ρ)

```
$e \leftarrow E - 1$, $g \leftarrow G - 1$
$t \leftarrow t_{\min}[e]$, $r \leftarrow 0$, $f \leftarrow F - 1$
$v, V \leftarrow \text{Solve-Infinite-Horizon}(d, k, e, g, f, r, t, \delta)$
for all $s \in \mathcal{S}$ do
  $\theta \leftarrow [d, k, e, g, f, r, t] \,|s$ {Append endogenous parameters, $s$}
  $i \leftarrow \text{Enumerate}(\theta)$
  $P_\alpha[i] \leftarrow \text{Compute-Choice-Prob}(\rho, v, V[i], \theta)$ {NB: $P_\alpha \equiv \Pr\{\alpha|\theta\}$}
end for
for all $s \in \mathcal{S}$ do
  $\theta \leftarrow [d, k, e, g, f, r, t] \,|s$
  $i \leftarrow \text{Enumerate}(\theta)$
  $P_s[i] = \sum_{\alpha \in \mathbf{A}(\theta)} P_\alpha[i, \alpha] P(\theta'|\alpha, \theta)$ {NB: $P_s \equiv P_s(\theta'|\theta)$}
end for
$P_{-\infty} \leftarrow \text{Solve-Stationary-Distribution}(P_s)$
return $V, P_{-\infty}$
```

Algorithm 7 Solve-Infinite-Horizon($d, k, e, g, f, r, t, \delta$)

```
$V_{\text{today}} \leftarrow [0 \ldots 0]$, $V_{\text{tomor}} \leftarrow [0 \ldots 0]$
**repeat**
  **for all** $s \in \mathcal{S}$ **do**
    $\theta \leftarrow [d, k, e, g, f, r, t] \,|s$ {Append endogenous parameters, $s$}
    {Assigning $v$ a function of $\alpha$}
    $v \leftarrow U(\alpha, \theta) + \delta \sum_{\theta'} P(\theta'|\alpha, \theta) V_{\text{tomor}}[\text{Enumerate}(\theta')]$
    $V_{\text{today}}[\text{Enumerate}(\theta)] \leftarrow \max_{\alpha \in \mathbf{A}(\theta)} v[\alpha]$
  **end for**
  Swap($V_{\text{today}}, V_{\text{tomor}}$)
**until** Converged($V_{\text{today}}, V_{\text{tomor}}$)
**return** $v, V$
```

Algorithm 8 Solve-Stationary-Distribution(P_s)

```
See Judd (1998, p. 85)
```

The Compute-Choice-Prob algorithm is implemented by `Outcome::calcActionProb(·)` in behaviour.ox and returns a function of α partially evaluated on θ. Recall, a logistic kernel with ρ parameter is used to smooth state-contingent probabilities.

Algorithm 9 Compute-Choice-Prob(ρ, v, V, θ)

```
$\tilde{v} \leftarrow \text{bool}(\alpha \in \mathbf{A}(\theta)) e^{\rho[v(\alpha,\theta) - V(\theta)]}$ {Assigning $\tilde{v}$ a function of $\alpha$}
**return** $\tilde{v} / \sum_{\alpha' \in \mathbf{A}(\theta)} \tilde{v}(\alpha')$ {Returning a function of $\alpha$.}
```

3.3 Defining a social experiment

The `SocialExperiment` class is the base-class for user defined experiments. Specifically, the user starts by defining their class as a subclass of `SocialExperiment`. An example class declaration is given below. Recall, the single method shown in the example is the constructor.

```
class MyExperiment : SocialExperiment {
    MyExperiment ();
    ⋮
}
```

By inheriting the `SocialExperiment` class a member named `def`, which is an instance of the `SocialExperimentDefinition` class, is available to the user. By using `def` the user is able to define all the necessary parameters of the experiment.

To define a social experiment the researcher must define the symbols given in Table 4. Note that symbols given as arguments to the defining function calls are only to illustrate the connection between the mathematical symbol and the function call. Using a constant as a function argument is usually more natural. For instance, to define two demographic groups (i.e. $D = 2$) the function call `def.d.setSize(2)` can be made. These function calls must be made in the constructor of the subclass.

Table 4: Parameters to be Defined

Variable	Defining function call
D	`def.d.setSize(`D`);`
K	`def.k.setSize(`K`);`
E	`def.e.setSize(`E`);`
G	`def.g.setSize(`G`);`
F	`def.f.setSize(`F`);`
$R(f)$	`def.R.setValue(`f`, `$R(f)$`);`
$t_{\min}(e)$	`def.tMin.setValue(`e`, `$t_{\min}(e)$`);`
$t_0(e)$	`def.t0.setValue(`e`, `$t_0(e)$`);`
$t_{\max}(e)$	`def.tMax.setValue(`e`, `$t_{\max}(e)$`);`
$f_0(e)$	`def.startPhase.setValue(`e`, `$f_0(e)$`);`
$\rho(k)$	`def.rho.setValue(`k`, `$\rho(k)$`);`
$\delta(k)$	`def.delta.setValue(`k`, `$\delta(k)$`);`
$\lambda(d,k)$	`def.unobsWeights.setValue(`d`, `k`, `$\lambda(d,k)$`);`

The methods given in Table 4 can be tedious to use if multiple method calls are necessary to define a parameter. There are, however, alternatives which can make multiple definitions in one call. For instance, `def` members that have the `setValues(·)` method can set several values at once rather than having to make separate function calls. For example, given $K = 2$, the call `def.delta.setValues(0.9, 0.8)` can be made to define $\delta_0 = 0.9$ and $\delta_1 = 0.8$ for $k = 0$ and $k = 1$ respectively. Another alternative, for `def` members that have the `setLabels(·)` method, is to set the labels and omit setting the size as it is implicitly set to the number of labels. For example, `def.k.setLabels(‘‘unskilled’’,‘‘skilled’’)` also implicitly calls `def.k.setSize(2)`.

Of special note is `def.unobsWeights.setValues(·)` as it requires a parameter for d before the value parameters. Returning to the previous example, given $K = 2$, `def.unobsWeights.setValues(0, 0.3, 0.7)` sets $\lambda(d, k)$ for $d = 0$. Recall that the weights must conform to $\sum_k \lambda(d, k) = 1$ for a given demographic group d.

Different parameter types determine which methods are available. These are summarized in Table 5 below. As an example r has two methods available `def.r.name()` and `def.r.index()` because it's definition is done within the library using $R(f)$. Correspondingly there are two "yes" entries in the `def.r` column. Note that defining methods are those that are prefixed with "set" with the exception of `def.⋆.add(·)` which is used to define actions and endogenous state parameters.

Table 5: Available Parameter Methods

def.⋆	def.⋆ member				
method	`def.⋆`^{1}	`def.⋆`^{2}	`def.⋆`^{3}	`def.r`	`def.unobsWeights`
`.name()`	yes	yes	yes	yes	yes
`.index()`	yes			yes	
`.indices()`			yes		
`.size()`	yes	yes	yes		
`.value(·)`	yes	yes			yes
`.values()`	yes	yes			yes
`.allValues()`					yes
`.labels()`	yes	yes			
`.get(·)`			yes		
`.getAll()`			yes		
`.add(·)`			yes		
`.setValue(·)`		yes			yes
`.setValues(·)`		yes			yes
`.setLabels(·)`	yes				
`.setSize(·)`	yes	yes			

`def.⋆`^{1} includes `def.d`, `def.k`, `def.e`, `def.g` and `def.f`
`def.⋆`^{2} includes `def.R`, `def.tMin`, `def.t0`, `def.tMax`, `def.startPhase`, `def.delta` and `def.rho`
`def.⋆`^{3} includes `def.endog` and `def.action`

In addition to parameter definitions, the following six virtual methods of `SocialExperiment` must be defined in the subclass. The relationship between these methods and the social experiment mathematical description is given in Table 6.

feasibleActions (**const** actions, **const** currState);
utility (**const** feasActions, **const** currState);
measurement(**const** feasActions, **const** currState);
select (**const** measurement, **const** e, **const** time);
treatmentTransition(**const** jump, **const** feasActions, **const** currState, **const** lastPeriod);
realityTransition (**const** jump, **const** feasActions, **const** currState);

Table 6: Virtual methods to be Overridden

Mathematical entity	Method name
$\mathbf{A}(\theta)$	`feasibleActions (·)`
$P\{\theta'\|\alpha,\theta\}$	`realityTransition(·)`
	and `treatmentTransition(·)`
$U(\alpha,\theta)$	`utility(·)`
$Y(\alpha,\theta)$	`measurement(·)`

Recall, α is a vector of actions that can be undertaken by the individual if feasible. Therefore, α can be defined as $\alpha = [\alpha_1, \ldots, \alpha_m]$. The first argument of `feasibleActions(·)`, `actions`, is a matrix with $\alpha_1, \ldots, \alpha_m$ represented as rows and different combinations of possible values represented as columns. For example, say $\alpha = [\alpha_1, \alpha_2]$ where $\alpha_1 \in \{0, 1\}$ and $\alpha_2 \in \{2, 3\}$, then the `actions` argument will be:

$$\texttt{actions} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 2 & 3 & 3 \end{bmatrix}$$

Within Ox, the `selectifc` function is a convenient way to select the feasible combinations. Given θ and using `actions` as defined above, say only actions with $\alpha_2 = 3$ are feasible. The snippet of Ox code which accomplishes this is,

```
selectifc(actions,actions[1][].==3);
```

which results in,

$$\texttt{selectifc(actions,actions[1][].==3)} = \begin{bmatrix} 0 & 1 \\ 3 & 3 \end{bmatrix}$$

The second parameter of `feasibleActions(·)` is `currState` which is simply a column vector representing the values of the current state θ. The required return value of `feasibleActions(·)` is a submatrix of `actions` where the number of rows are identical but the columns are only those that represent the feasible actions, as demonstrated above.

An issue that has not yet been discussed is how to determine the index for a particular parameter. For state parameters, d, k, e, g, f and r the most obvious way to get the index is to use the `def.⋆.index()` method. Another, more direct, way to get the index for the aforementioned parameters with the exception of r is to use the return value from the defining method call. For example, `dIdx = def.d.setSize(2)` both defines $D = 2$ and as well as having the same effect as executing `dIdx = def.d.index()`.

For both `def.actions` and `def.endog` the easiest way to get the index from a newly created parameter is to use the return value from `def.⋆.add(const name, const size)` method call. For instance, `myActionIdx = def.action.add(‘‘myAction’’, 2)` defines a new action with two states and sets the row index for referencing the `actions` matrix.

The methods `utility(·)` and `measurement(·)` have the same parameter definition: `const feasActions` and `const currState`. The `feasActions` parameter is the matrix returned from the `feasibleActions(·)` method and `currState` is a vector as before. The `utility(·)` method returns the single period utility over the feasible actions, resulting in a row vector with the same number of columns as `feasActions`. The `measurement(·)` method returns a matrix of measures where rows represent particular measurements and columns represent the measurements over the feasible actions.

The two transition methods are more complicated, although neither has a `return` value. The difference between the methods is that the `realityTransition(·)` method only needs to give transitions over the endogenous parameters whereas the `treatmentTransition(·)` method must also handle phase and period transitions (i.e. f and r). To avoid code replication, `realityTransition(·)` can be called within the `treatmentTransition(·)` method to handle the endogenous parameters' transition. The parameters that the two methods have in common are `const jump`, `const feasActions` and `const currState`. Both the `feasActions` and `currState` are defined as before: the output from the `feasibleActions(·)` method and the current state vector respectively. The `const jump` parameter is an object used to define the transitions or "jump" of the relevant parameters and has two methods:

```
get(const index);
update(const index, const probs);
```

The implementation of jumps is different than Ferrall's (2002) definition, in that the default jump is defined implicitly rather than being an explicit parameter in the jump calculation. A transition definition starts by retrieving the transition matrix for a particular state parameter by using it's index. This is the same index that is used to get values from the state vector. The different state parameter values and feasible actions are represented by rows and columns respectively. The transition matrix is initialize with Ox `.NaN`s (Not a Number). An example will help clarify.

Say an endogenous state parameter is defined in the constructor using the following code snippet,

```
myEndogIdx = def.endog.add(‘‘myEndog’’, 3);
```

Note the default values for the endogenous parameter are 0,1 and 2. Within the `realityTransition(·)` method, the transition matrix is retrieved using the following line of code:

```
myEndogTrans = jump.get(myEndogIdx);
```

Assuming, for this example, that there are two feasible actions the `myEndogTrans` matrix is,

$$\texttt{myEndogTrans} = \begin{bmatrix} .\text{NaN} & .\text{NaN} \\ .\text{NaN} & .\text{NaN} \\ .\text{NaN} & .\text{NaN} \end{bmatrix}$$

Once the matrix is set using the native Ox matrix operators, the library is then notified by using the `update(·)` method as is shown below.

```
jump.update(myEndogIdx, myEndogTrans);
```

For a valid transition matrix each column vector must sum to one, although it is not necessary for each cell to be set. Rather the library detects if a column vector sums to less than one and then splits the residual equally over the remaining unset cells (i.e. cells with a `.NaN` value). Returning back to the example, say given the current state and feasible actions, the endogenous state parameter stays at its current state value with a $\frac{2}{3}$ probability or goes to any other value with an equal probability. Also, for simplicity, say the transitions are not a function of the feasible actions, that is, the jump is *autonomous* (Ferrall 2002).The desired code snippet is,

```
  myEndogVal = currState[myEndogIdx];
  myEndogTrans[myEndogVal][] = 2/3;
  jump.update(myEndogIdx, myEndogTrans);
```

Once the `update(·)` method is called, the library eliminates all remaining `.NaN`s. For illustration, say that `myEndogVal = 1` and the transition matrix is retrieved again. The now defined transition matrix would be,

$$\texttt{myEndogTrans} = \begin{bmatrix} 1/6 & 1/6 \\ 2/3 & 2/3 \\ 1/6 & 1/6 \end{bmatrix}$$

The `treatmentTransition(·)` method takes an additional `const` parameter, `lastPeriod`, which is boolean valued. If `lastPeriod` is `TRUE` then the phase is currently in its last period and the period must be set so it jumps back to zero (i.e. $r = 0$). In addition the period must also transition unless it is in its final state. If `lastPeriod` is `FALSE`, then the period must be incremented. A caveat is that anytime the phase jumps to a new value the period must be set to zero. The code snippet below will clarify the requirements,

```
MyExperiment::treatmentTransition(
              const jump,
              const feas,
              const state,
              const lastPeriod)
{
  decl
    f = state[fIdx],
    r = state[rIdx],
  ⋮
  nextPeriodProb = jump.get(rIdx);
  if (lastPeriod) {
    // If in last period don't forget to set r to zero.
    nextPeriodProb[0][] = 1.0;
```

```
   } else {
      // Have to refetch f to get proper probs (i.e. no NaN's).
      nextPhaseProb = jump.get(fIdx);
      // Increment r if phase unchanged.
      nextPeriodProb[r+1][] = nextPhaseProb[f][];
      // If phase changed set r=0.
      nextPeriodProb[0][] = 1.0 - nextPhaseProb[f][];
   }
   jump.update(rIdx,nextPeriodProb);
}
```

Building on these basics, the next section will go through a sample application in detail.

4 Application - Illinois Reemployment Bonus experiment

4.1 Background

The Illinois Reemployment Bonus experiment was a random assignment experiment of unemployment insurance (UI hereafter) claimants conducted between mid-1984 to mid-1985 by the Illinois Department of Employment Security. The goal of the experiment was to determine the effect of the UI system on the behaviour of the unemployed. The experiment allowed those in the treatment group to receive a \$500 bonus if they found employment within 11 weeks of filing their UI claim and then stayed employed for 4 months. Selection into the experiment required that the individual be a new UI claimant. A variation of this experiment was also conducted concurrently where the new employer would receive the \$500 bonus rather than the individual. In this paper only the bonus for individuals is considered.

The experimental data included characteristics such as sex, age and race (i.e. demographics) as well as history for earnings and UI benefits, whether the claimant received the \$500 bonus (i.e. endogenous characteristics) and treatment group membership (e.g. control, claimant receives bonus if eligible, employer receives bonus if eligible). The first analysis of the Illinois Reemployment Bonus experiment was conducted by Woodbury and Spiegelman (1987) and Table 2 within their report gives a breakdown of claimant counts by characteristics and treatment group.

4.2 Example model

As the Illinois Reemployment Bonus experiment is presented here as an illustration the model is simplified. For instance, rather than multiple demographic groups only two will be used. A summary of the state parameter values, including definitions of endogenous variables (i.e. `wrking` through to `npue`), are presented in Table 7. As the model is a simplification the descriptions used are purely illustrative. For instance, rather than choose men and women for the two demographic groups, two racial groups could have been chosen (e.g. black and white).

Table 7: Experiment State Parameters

$D = 2$	men and woman
$K = 2$	low and high skilled
$E = 1$	only one way to enter the experiment
$G = 2$	treatment and control
$F = 5$	`preexperiment`, `qualify`, `stayemployed`, `getbonus` and `postexperiment`
`wrking`	Working? (0=FALSE/1=TRUE)
`offer`	five wage offer indices $(0 \dots 4)$
`preearn`	five previous wage offer indices $(0 \dots 4)$
`npwrk`	number periods worked (0 to 5 months)
`npue`	number periods unemployed (0 to 5 months)

The maximum number of periods in a given phase which is measured in months is defined in Equation 6. For the qualification phase in the first period (i.e. $r = 0$) the probability of being employed is necessarily zero as only new UI claimants are considered. The remaining three periods or months in the qualification phase represents the 11 week limit, which is rounded to three months from the actual 2.5 months, to find a job in order to receive a bonus.

$$R(f) = \begin{cases} 4 & \text{if } f \in \{\text{qualify}, \text{stayemployed}\} \\ 1 & \text{otherwise} \end{cases} \tag{6}$$

Additional parameters that are dependent on state variables e, d and k are given in Table 8. There is only one entry group so $\forall e$ is equivalent to $e \in \{0\}$.

Table 8: Time, Treatment Start Phase and Weight Parameters

$t_{\min}(e)$	= -1 for $\forall e$
$t_0(e)$	$= 0$ for $\forall e$
$t_{\max}(e)$	$= 5$ for $\forall e$
$f_0(e)$	= qualify for $\forall e$
$\lambda(d,k)$	$= 0.5$ for $\forall d, k$

There are also several exogenous parameters for the model. These are given in Table 9.

Table 9: Other Exogenous Parameters

κ	$= 1.0$	Value of effort
$\mu[k]$	$= [1.1, 1.2]$	Heterogeneity of wage offers
`offprob`	$= 0.1$	Probability of job offer
`layprob`	$= 0.2$	Probability of layoff
`uiRate`	$= 0.6$	% of previous wage received as UI benefit
`REbonus`	$= 5.0$	Reemployment bonus in \$100s

Individuals have an endogenous choice to exert effort in both job search and employment. Also, the individual can choose to accept or reject a job offer. Receiving a wage offer of \$0 corresponds to no job offers. Therefore the action vector is either $\alpha = [\text{effort}, \text{accept}]$ if offered a job or $\alpha = [\text{effort}]$ if currently working. Both `effort` and `accept` are boolean valued.

The wage function is assumed to be a discretized log-normal distribution that also includes a \$0 wage possibility. The wage is given as,

$$\text{wage}[k] = \begin{cases} 0 & \text{if } \texttt{offer} = 0 \\ exp\left\{\mu[k] + \Phi(\frac{\text{offer}-1}{F})\right\} & \text{if } \texttt{offer} > 0 \end{cases}$$

The UI benefits are a portion, `uiRate`, of previous earnings. In addition, to qualify for unemployment the individual must have worked previously, that is `npwrk` > 0. Therefore the benefits received while on UI are,

$$\text{UI} = \begin{cases} 0 & \text{if } \texttt{npwrk} = 0 \\ \text{uiRate} \cdot \text{wageUI} & \text{if } \texttt{npwrk} > 0 \end{cases}$$

where,

$$\text{wageUI}[k] = \begin{cases} 0 & \text{if } \texttt{preearn} = 0 \\ exp\left\{\mu[k] + \Phi(\frac{\text{preearn}-1}{F})\right\} & \text{if } \texttt{preearn} > 0 \end{cases}$$

The single period utility function is defined as,

$$U(\alpha, \theta) = \begin{cases} \text{valueUE} + (\text{wage} - \text{valueUE}) \cdot \text{accept} + \text{UI} & \text{if unemployed} \\ \text{wage} - \text{valueUE} + \text{REbonus} \cdot \text{bRecd} & \text{otherwise} \end{cases}$$

where,

$$\text{valueUE} = \kappa(1 - \text{effort})$$
$$\text{bRecd} = \begin{cases} 1 & \text{if } f = \text{getbonus} \\ 0 & \text{otherwise} \end{cases}$$

The measurement vector, $Y(\alpha, \theta)$, are characteristics to be predicted by the Niplow software and are also used as a parameter for determining valid measurements, $\mathcal{H}[y; t, e, d]$. For this model the measurements are chosen as follows,

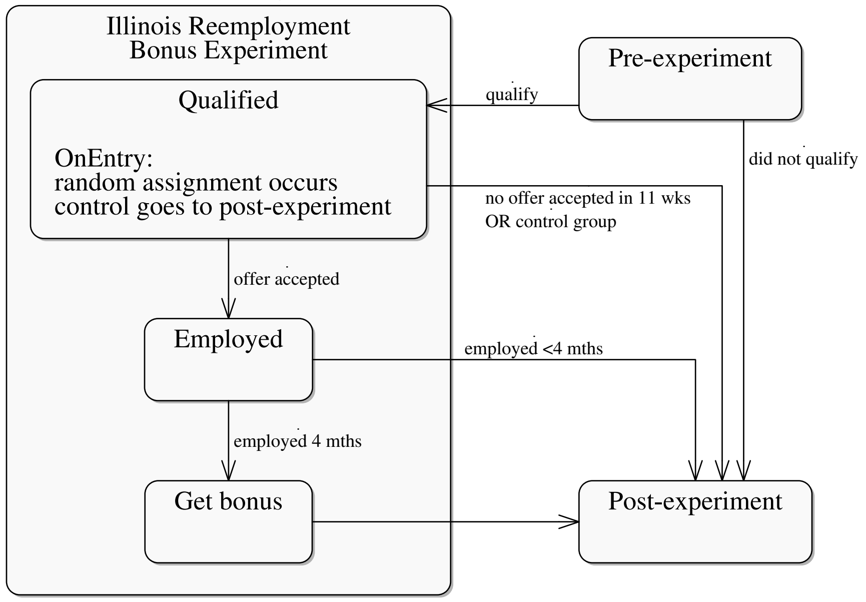
$$Y(\alpha, \theta) = \begin{bmatrix} \texttt{effort} \\ \texttt{accept} \\ \text{wage} \\ \texttt{wrking} \\ \text{UIRecd} \\ \text{UI} \\ \text{bRecd} \end{bmatrix}$$

where,

$$\text{UIRecd} = \begin{cases} \texttt{FALSE} & \text{if UI } = 0 \\ \texttt{TRUE} & \text{otherwise} \end{cases}$$

The model includes five phases, `preexperiment`, `qualify`, `stayemployed`, `getbonus` and `postexperiment`. The initial phase for the experiment is the `qualify` phase. A general overview of the phase transitions are given in Figure 1. Recall that phase changes are progressive in that if the phase changes it cannot move back to a previous phase.

Figure 1: Phase Transitions



4.2.1 Source code - reempBonus.ox

This section describes the source code in detail. Small portions of code are presented with the descriptions, but longer listings are in Section A.1 of the Appendix.

A listing of the `ReempBonus` class is in Section A.1.1. As is mandatory the class inherits from the `SocialExperiment` (line 1), and declares the parameterless constructor `ReempBonus()` (line 21) as well as declares the six methods that need to be overridden (lines 24 to 29). Notice that all the members are declared as constants using the `const decl` statement rather than using static `enum`'s. This is because some of the values will be real valued which is not support by `enum` and also may not be known until run-time. For example indices values for the state variable vector is not given by Niplow until runtime which is why members, such as dIdx, cannot be declared as an `enum`.

The constructor `ReempBonus()` is listed in Section A.1.2. The code is written for clarity but there are several points that may not be obvious. First, in Ox the constructor of the base class, `SocialExperiment()`, must be called explicitly (line 2). Second, the `def.⋆.values()` method returns an array not a vector. This allows Ox's multiple assignment syntax to be used of which there are two examples involving `def.d` and `def.f` (lines 7 and 11). Third, `def.f.setLabels(·)` implicitly sets $F = 5$ (line 10). Forth, `def.r` has a reduced set of methods compared to other state parameters. For this example, only the index of r is needed (line 13). Finally, the last treatment group defined must be the control group (line 29).

Each of the current state values are passed in separate method calls, but all the feasible actions are passed simultaneously. The underlying assumption is that the number of feasible action combinations is small and so can be passed as a group without taking excessive amounts of computer memory. Recall, the feasible action matrix stores the separate action vectors as column vectors. For this model, the two actions, `effort` $\in \{0,1\}$ and `accept` $\in \{0,1\}$, are represented as matrix rows 1 and 2 respectively. The restriction on the actions is that if an individual is working (i.e. `wrking` $= 1$) then not accepting (i.e. `accept` $= 0$) is considered infeasible. Therefore, the feasible actions matrix and the generating method `feasibleActions(·)` are as follows.

$$\texttt{feasActions} = \begin{cases} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} & \text{if } \texttt{wrking} \ = 1 \\ \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} & \text{if } \texttt{wrking} \ = 0 \end{cases}$$

```
 ReempBonus::feasibleActions (const actions, const state) {
   print(selectifc(actions,actions[accept][])); exit(1);
   // Not accepting is feasible only if not working.
   return (state[wrkingIdx]) ?
      // Remove the "not accepting" columns.
      selectifc(actions,actions[accept][]) :
      actions;
}
```

The methods `utility(·)` and `measurement(·)`, listed in Section A.1.3, only differ in terms of their `return` values (lines 2 to 11 and 20 to 29 are identical). The code does have a slight divergence from the model in that separate wages, `wage` and `wageUI`, for working and not working respectively are not calculated. Instead `wage` is contingent on `effwage` which is the wage index conditional on the `wrking` endogenous parameter (line 4). The return value for `utility(·)` is a row vector of one period utilities for each feasible action. The return value for `measurement(·)` is a matrix of measurements, as rows, for each feasible action, as columns.

Only new UI claimants are eligible to enter the experiment. The `select(·)` method ensures that all individuals are employed in the `preexperiment` phase, or equivalently when $t = -1$, and are then unemployed when the experiment starts and the individual enters the `qualify` phase at $t = 0$.

```
// Measurement vector.
enum {effort, accept, wage, employed, anyUI, UI, bonus};
const decl measurementLabels = {
   "effort", "accept", "wage", "employed", "anyUI", "UI", "bonus"
};

ReempBonus::select(const measurement, const e, const time) {
   return (time==-1) ?
      // off UI - so we include only new spells
      (1-measurement[anyUI][]) :
      // on UI - initiate new spell
      measurement[anyUI][];
}
```

The `realityTransition(·)` method, listed in Section A.1.4, only sets transitions for the endogenous state parameters. There are several lines which contain some illustrative examples that would be implemented differently or completely removed to optimize the code. The `def` member is accessible to every `ReempBonus(·)` method and can be used during estimation to fetch properties, although more efficient code would assign the properties to `const` members in the constructor. Examples are, assigning the number of offer states to `nwo` (line 9), preventing an index overrun for `npwrk` (line 71), and assigning fractional effort to `eff` (line 15), although for this model `eff` $\in \{0, 1\}$. Most jump probabilities are set in a single line, with the exception of the `preearn` jump matrix, `prevWageProb`, which is more complicated for the unemployed case (lines 58 to 63).

The `treatmentTransition(·)` method, listed in Section A.1.5, sets transitions for the phase and period and then sets the endogenous state parameters by calling `realityTransition(·)` (line 48). Setting of the period transition is dependent on whether the maximum period has been reached (i.e. $r = R(f) - 1$) which is determined by `lastPeriod`. This is “boiler-plate” code (lines 32 to 45) and will be made part of Niplow in a future version, however, its inclusion here is useful for insight into the workings of the period transition.

4.2.2 Obtaining results

The following is the `main()` of ReempBonus.ox. The model must be instantiated and run to produce a prediction (line 3). The result, `predict`, is a series of nested arrays, which are navigated using the nested `for` loops.

```
1  main () {
2     decl predict, d, e, g;
3     predict = (new ReempBonus()).run();
4     for(d=0; d<sizeof(predict); d++) {
5        for(e=0; e<sizeof(predict[d]); e++) {
6           for(g=0; g<sizeof(predict[d][e]); g++) {
7              println("d=",d,", e=",e,", g=",g);
8              print("prediction","%c",measurementLabels,
9                 "%12.5f", predict[d][e][g]');
10          }
11       }
12    }
13 }
```

The prediction output is below. Note that only values for $d = 0$ are shown because those for $d = 1$ are identical. The rows represent months in the experiment ranging from 0 to 5 (i.e $t_0 \dots t_{\max}$). Notice how control groups (i.e. $g = 1$) never receive the bonus, but some portion of those in the treatment group (i.e. $g = 0$) do in the last month.

4.2.3 Output - reempBonus.ox

```
d=0, e=0, g=0
prediction
      effort       accept         wage     employed        anyUI           UI
     0.98178      0.97096      3.76463      0.96895      0.00176      0.00264
     0.90130      0.77720      3.01307      0.77219      0.00181      0.00271
     0.83461      0.63899      2.45868      0.62111      0.01287      0.01930
     0.78331      0.53940      2.05484      0.51233      0.02076      0.03114
     0.72497      0.46609      1.75823      0.43262      0.02628      0.03943
     0.69814      0.39492      1.51671      0.35633      0.03047      0.04570
       bonus
     0.00000
     0.00000
     0.00000
     0.00000
     0.00000
     0.39398
d=0, e=0, g=1
prediction
      effort       accept         wage     employed        anyUI           UI
     0.93219      0.97110      3.76463      0.96895      0.00176      0.00264
     0.85209      0.73944      2.95999      0.73393      0.00164      0.00246
     0.78895      0.58670      2.38981      0.56610      0.01486      0.02229
     0.74529      0.48546      1.99045      0.45489      0.02358      0.03537
     0.71477      0.41689      1.70697      0.37970      0.02936      0.04404
     0.69316      0.36965      1.50340      0.32796      0.03327      0.04991
       bonus
     0.00000
     0.00000
     0.00000
     0.00000
     0.00000
     0.00000
```

5 Conclusion

A choice faced by econometricians is whether to write their software or use existing software to perform calculations. For an experimentalist approach, the decision is easier given the plethora of choices both free (e.g. R) and commercial (e.g. Stata). When dealing with structural models and, in this case, their application to social experiments the more immediate choice, given the lack of options, is to expend considerable effort programming. The Niplow software library, implementing Ferrall's (2002) comprehensive mathematical model for estimating social experiments, gives the researcher an alternative. In addition, Niplow is "open source" allowing users to verify correctness and extend the framework in a way that benefits themselves and the entire user community. Given this is the first release of the software there is plenty of opportunity for future development.

Niplow implements a subset of Ferrall's (2002) model. For example, the extension for policy innovations and functions that allow for concise jump descriptions (e.g. autonomous jumps) have not been implemented. Also unimplemented is using GMM to estimate the model using observed moments, which would be valuable addition.

There are other features which would go beyond the contemporary model. For instance, Niplow could allow for a finite horizon problem which has a non-ergodic absorbing state (e.g. death), rather than the current infinite horizon problem. Also, the user could be given the option of providing their own probability distribution over endogenous states for a given d and k.

From a design point of view, the software can be enhanced in several ways. Further code verification by inspection and additional test cases would alleviate correctness concerns. Also important is the inclusion of "asserts" at key points of code execution to ensure the program has not entered a "fault" state which otherwise may be difficult to detect. The addition of parallel execution over d and k and optimizations of serial code would allow practical estimation of more complex models. By splitting the code into a simple library for estimating DDPs and a dependent social experiment estimation library, researchers would be able to easily use the library for estimating other types of DDP models. Finally, further reducing the programming burden, such as letting the library rather than the user be responsible for “boiler-plate” code, will enhance the users experience which is a key determinant of the software’s success.

References

Aguirregabiria, V., and P. Mira (2002) 'Swapping the nested fixed point algorithm: a class of estimators for discrete Markov decision models.' *Econometrica* pp. 1519–1543

Angrist, J.D. (1990) 'Lifetime earnings and the Vietnam era draft lottery: evidence from social security administrative records.' *The American Economic Review* pp. 313–336

Angrist, J.D., and A.B. Krueger (2001) 'Instrumental variables and the search for identification: From supply and demand to natural experiments.' *Journal of Economic Perspectives* pp. 69–85

Bellman, R. (1966) 'Dynamic programming.' *Science* 153(3731), 34–37

Brooks, F.P. (1987) 'No silver bullet: Essence and accidents of software engineering.' *IEEE computer* 20(4), 10–19

Burtless, G. (1995) 'The case for randomized field trials in economic and policy research.' *The Journal of Economic Perspectives* pp. 63–84

Ching, Andrew, Susumu Imai, Masakazu Ishihara, and Neelam Jain (2009) 'A Practitioner's Guide to Bayesian Estimation of Discrete Choice Dynamic Programming Models.' *SSRN eLibrary*

Cribari-Neto, F. (1997) 'Econometric programming environments: GAUSS, Ox and S-PLUS.' *Journal of Applied Econometrics* pp. 77–89

Doornik, J.A., and M. Ooms (1998) *Introduction to Ox* (Timberlake Consultants)

Eckstein, Z., and K.I. Wolpin (1999) 'Why youths drop out of high school: The impact of preferences, opportunities, and abilities.' *Econometrica* pp. 1295–1339

Ferrall, C. (2000) 'Estimation and Inference in Social Experiments: The Self-Sufficiency Project and The Dynamics of Income Assistance.' *Social Research Demonstration Corporation*

—— (2002) 'Estimation and inference in social experiments.' *Queen's University, Institute for Economic Research*

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995) *Design patterns: elements of reusable object-oriented software*

Hotz, V.J., G.W. Imbens, J.H. Mortimer, and L. Center (1999) 'Predicting the efficacy of future training programs using past experiences.' *NBER Working Paper*

Howitt, R., S. Msangi, A. Reynaud, and K. Knapp (2002) 'Using Polynomial Approximations to Solve Stochastic Dynamic Programming Problems: or A Betty Crocker Approach to SDP.' Technical Report, Department of Agricultural and Resource Economics, University of California, Davis

Imbens, G.W., and J.D. Angrist (1994) 'Identification and estimation of local average treatment effects.' *Econometrica: Journal of the Econometric Society* pp. 467–475

Judd, K.L. (1998) *Numerical methods in economics* (The MIT Press)

Keane, M.P. (2006) 'Structural vs. atheoretic approaches to econometrics.' *Journal of Econometrics, pp*

Keane, M.P., and K.I. Wolpin (1994a) 'The solution and estimation of discrete choice dynamic programming models by simulation and interpolation: Monte Carlo evidence.' *The Review of Economics and Statistics* pp. 648–672

—— (1994b) 'The solution and estimation of discrete choice dynamic programming models by simulation and interpolation: Monte Carlo evidence.' *The Review of Economics and Statistics* pp. 648–672

Kendrick, D.A., and H.M. Amman (1999) 'Programming languages in economics.' *Computational Economics* 14(1), 151–181

Lise, Jeremy, Shannon Seitz, and Jeffrey A. Smith (2003) 'Equilibrium Policy Experiments and the Evaluation of Social Programs.' *SSRN eLibrary*

Meyer, B. (1988) *Object-oriented software construction* (Prentice-Hall New York)

Raymond, E. (1999) 'The cathedral and the bazaar.' *Knowledge, Technology, and Policy* 12(3), 23–49

Rosenzweig, M.R., and K.I. Wolpin (2000) 'Natural "natural experiments" in economics.' *Journal of Economic Literature* 38(4), 827–874

Rust, J. (1987) 'Optimal replacement of GMC bus engines: An empirical model of Harold Zurcher.' *Econometrica: Journal of the Econometric Society* pp. 999–1033

—— (1996) 'Numerical dynamic programming in economics.' *Handbook of Computational Economics* 1, 619–729

—— (1997) 'Using randomization to break the curse of dimensionality.' *Econometrica: Journal of the Econometric Society* pp. 487–516

Rust, J., and C. Phelan (1997) 'How social security and medicare affect retirement behavior in a world of incomplete markets.' *Econometrica: Journal of the Econometric Society* pp. 781–831

Todd, P.E., and K.I. Wolpin (2006) ‘Assessing the impact of a school subsidy program in Mexico: Using a social experiment to validate a dynamic behavioral model of child schooling and fertility.’ *American Economic Review* 96(5), 1384–1417

Wolpin, K.I. (1996) ‘Public-policy uses of discrete-choice dynamic programming models.’ *The American Economic Review* pp. 427–432

Woodbury, S.A., and R.G. Spiegelman (1987) ‘Bonuses to workers and employers to reduce unemployment: Randomized trials in Illinois.’ *The American Economic Review* pp. 513–530

A Appendix

A.1 Source code Listings

A.1.1 ReempBonus Class

```
class ReempBonus : SocialExperiment {
  const decl dIdx, kIdx, fIdx, rIdx, gIdx, eidx;

  // Experiment phases.
  const decl preexperiment,qualify,stayemployed,getbonus,postexperiment;

  // Indices for endogenous state parameters.
  const decl wrkingIdx, offerIdx, preearnIdx, npwrkIdx, npueIdx;

  // Agent actions.
  const decl effort, accept;

  // Utility parameters and labels.
  const decl upars, parlabels;

  // Discretized wage offer distribution.
  const decl wageDistn;

  const decl REbonus, uiRate;

  ReempBonus();

  // Methods to override.
  feasibleActions (const actions, const currState);
  utility(const feasActions, const currState);
  measurement(const feasActions, const currState);
  select(const measurement, const e, const time);
  treatmentTransition(const jump, const actions, const currState, const lastPeriod);
  realityTransition(const jump, const actions, const currState);
}
```

A.1.2 ReempBonus() Constructor

```
ReempBonus::ReempBonus() {
  SocialExperiment();

  // Define demographic groups.
  decl d1, d2;
  dIdx = def.d.setSize(2);
  [d1, d2] = def.d.values();

  // Define phases.
```

```
fIdx = def.f.setLabels("pre","qualify","stayemp","getbonus","post");
[preexperiment,qualify,stayemployed,getbonus,postexperiment] = def.f.values();

rIdx = def.r.index(); // Don't set r - it's done internally!

// Set phase limits.
def.R.setValue(qualify,4);
def.R.setValue(stayemployed,4);

// Define experiment entry types.
eidx = def.e.setSize(1);
decl e = def.e.values()[0];
def.tMin.setValue(e,-1);
def.t0.setValue(e,0);
def.tMax.setValue(e,5);
def.startPhase.setValue(e,qualify);

// Define treatment groups.
// Control MUST be the last group.
gIdx = def.g.setLabels("treatment","control");

// Define unobserved types.
kIdx = def.k.setSize(2);
def.delta.setValues(0.9, 0.9);
def.rho.setValues(2.0, 2.0);

def.unobsWeights.setValues(d1, 0.5, 0.5);
def.unobsWeights.setValues(d2, 0.5, 0.5);

upars = new matrix[def.k.size()][NNpars];
parlabels = new array[NNpars];
upars[][kappa] = 1.0;   parlabels[kappa] = "valef";
upars[][mu] =  <1.1;1.2>;  parlabels[mu] = "mnoff"; // heterogeneity
upars[][sigma] =  1.0;  parlabels[sigma] ="sdoff";
upars[][offprob] =  0.1;   parlabels[offprob] = "offprb";
upars[][layprob] =  0.2;      parlabels[layprob]="layprb";

// Endogenous parameters.
wrkingIdx  = def.endog.add("Working?", 2);  // Working? (TRUE/FALSE)
offerIdx = def.endog.add("Wage offer", 5); // Wage offer.
preearnIdx = def.endog.add("PreEarn", 5);  // Previous wage.
npwrkIdx = def.endog.add("# mths work", 6); // # periods worked.
npueIdx    = def.endog.add("# mths UE", 6); // # periods unemployed.

// Define actions.
effort  = def.action.add("effort", 2);
accept  = def.action.add("accept", 2);
```

```

   // Define wage distribution.
   // Number of offers including $0.
   decl offerCnt = def.endog.get(offerIdx).size();
   // Number of actual job offers (i.e. actual offers are >$0).
   decl jobOfferCnt = offerCnt-1;
   wageDistn = 0 ~
      exp(upars[][mu]+upars[][sigma]*quann(range(1,jobOfferCnt)/(offerCnt)));

   // Set constants.
   REbonus = 5.0; // Illinois Reemployment bonus, in hundreds of dollars
   uiRate  = 0.6; // Proportion of previous wage received as UI benefits.
}
```

A.1.3 utility(·) and measurement(·)

```
ReempBonus::utility(const feasActions, const state) {
   decl
      wrking  = state[wrkingIdx],
      effwage = state[ wrking ? preearnIdx : offerIdx ],
      k    = state[kIdx],
      wage  = wageDistn[k][effwage],
      eff     = feasActions[effort][],
      UEval = upars[k][kappa]*(1-eff),
      bRecd = state[fIdx]==getbonus,
      elig  = state[npwrkIdx]>0,
      ui    = (1-wrking)*elig*uiRate*effwage;
   decl utilVal;

   utilVal = UEval +(wage-UEval).*feasActions[accept][] + REbonus*bRecd + ui;

   return utilVal;
}

ReempBonus::measurement(const feasActions, const state) {
   decl
      wrking  = state[wrkingIdx],
      effwage = state[ wrking ? preearnIdx : offerIdx ],
      k    = state[kIdx],
      wage  = wageDistn[k][effwage],
      eff     = feasActions[effort][],
      UEval = upars[k][kappa]*(1-eff),
      bRecd = state[fIdx]==getbonus,
      elig  = state[npwrkIdx]>0,
      ui    = (1-wrking)*elig*uiRate*effwage;

   return
      (feasActions[effort][].>0) |
      feasActions[accept][] |
      wage | wrking | (ui>0) | ui | bRecd;
}
```

A.1.4 realityTransition(·)

```
ReempBonus::realityTransition(const jump, const feas, const state) {
  decl
    wrking  = state[wrkingIdx],
    offer = state[offerIdx],
    preearn = state[preearnIdx],
    npwrk = state[npwrkIdx],
    npue  = state[npueIdx],
    k  = state[kIdx],
    nwo = def.endog.get(offerIdx).size(),
    eff;

  decl wrkingProb, offerProb, prevWageProb, valueProb, npwrkProb, npueProb;

  // Get the effort value (i.e. 0, 1/2, 1) given the effort state.
  eff = feas[effort][]/(def.action.get(effort).size()-1);

  // Probability matrices come initialized with .NaN's.
  wrkingProb = jump.get(wrkingIdx);
  offerProb = jump.get(offerIdx);
  prevWageProb = jump.get(preearnIdx);
  npwrkProb = jump.get(npwrkIdx);
  npueProb = jump.get(npueIdx);

  if (wrking) { // Currently EMPLOYED.

    // TRANSITION FOR WORKING STATUS
    // Probability of NOT getting laid off.
    wrkingProb[1][] = (1-upars[k][layprob])*eff;

    // TRANSITION FOR WAGE OFFER
    // No offers while working.
    offerProb[0][] = 1.0;

    // TRANSITION FOR PREVIOUS WAGE
    // If working previous wage is unchanged.
    prevWageProb[preearn][] = 1.0;

    // TRANSITION FOR MONTHS WORKING IN PAST YEAR
    // Increment months worked in past year unless at the maximum.
    npwrkProb[min(npwrk+1,def.endog.get(npwrkIdx).size()-1)][] = 1.0;

    // TRANSITION FOR MONTHS of UI benefits remaining
```

```
      // Months on UI in past year remains unchanged.
      npueProb[npue][] = 1.0;

   } else { // Currently UNEMPLOYED.

      // TRANSITION FOR WORKING STATUS
      // Probability of accepting a job.
      wrkingProb[1][] = feas[accept][];

      // TRANSITION FOR WAGE OFFER
      // Probability of no wage offers.
      // Positive wage offers are equally likely so leave as NaNs.
      offerProb[0][] = 1-upars[k][offprob]*eff;

      //  TRANSITION FOR PREVIOUS WAGE
      // Change .NaN's to zeros for offer and preearn states.
      prevWageProb[newVector(offer,preearn)][] = 0;
      // Jump to wage offered if accepted.
      prevWageProb[offer][] += feas[accept][];
      // Otherwise keep the current wage.
      prevWageProb[preearn][] += 1.0-feas[accept][];

      // TRANSITION FOR MONTHS WORKING IN PAST YEAR
      // Months worked in past year remains unchanged.
      npwrkProb[npwrk][] = 1.0;

      // TRANSITION FOR MONTHS of UI benefits remaining
      // Increment months unemployed in past year unless at the maximum.
      npueProb[min(npue+1,def.endog.get(npueIdx).size()-1)][] = 1.0;
   }

   // Update the jump transition matrix.
   // Any remaining .NaN's are replaced with the residual probability.
   jump.update(wrkingIdx,wrkingProb);
   jump.update(offerIdx,offerProb);
   jump.update(preearnIdx,prevWageProb);
   jump.update(npwrkIdx,npwrkProb);
   jump.update(npueIdx,npueProb);
}
```

A.1.5 treatmentTransition(·)

```
ReempBonus::treatmentTransition
   (const jump, const feas, const state, const lastPeriod)
{
   decl
      wrking  = state[wrkingIdx],
      f     = state[fIdx],
      r    = state[rIdx],
      nextPhaseProb, nextPeriodProb;

   // Set phase transitions.
   nextPhaseProb = jump.get(fIdx);
   if (f==qualify) {
      // Move to stayemployed phase if accepting a job.
      nextPhaseProb[stayemployed][] = feas[accept][];
      /* Not accepting a job can cause a jump to
         postexperiment phase if in last period. */
      nextPhaseProb[lastPeriod ? postexperiment : qualify][] =
         1.0 - feas[accept][];
   } else if (f==stayemployed) {
      nextPhaseProb[wrking ?
         (lastPeriod ? getbonus : stayemployed) : postexperiment][] = 1.0;
   } else if (f==getbonus) {
      // After getting bonus go to the postexperiment phase.
      nextPhaseProb[postexperiment][] = 1.0;
   } else {
      // Other phase states are absorbing.
      nextPhaseProb[f][] = 1.0;
   }

   jump.update(fIdx,nextPhaseProb);

   // Set period transitions.
   nextPeriodProb = jump.get(rIdx);
   if (lastPeriod) {
      // If in last period don't forget to set r to zero.
      nextPeriodProb[0][] = 1.0;
   } else {
      // Have to refetch f to get proper probs (i.e. no NaN's).
      nextPhaseProb = jump.get(fIdx);
      // Increment r if phase unchanged.
      nextPeriodProb[r+1][] = nextPhaseProb[f][];
      // If phase changed set r=0.
      nextPeriodProb[0][] = 1.0 - nextPhaseProb[f][];
   }
   jump.update(rIdx,nextPeriodProb);

   // Gather the same transitions for endogenous variables as if in reality.
   realityTransition(jump, feas, state);
}
```