respy Documentation

Release 1.0.0

Philipp Eisenhauer

Aug 21, 2018

Contents

PyPI | GitHub | Issues

`respy` is an open-source Python package for the simulation and estimation of a prototypical finite-horizon discrete choice dynamic programming model. We build on the baseline model presented in:

> Keane, M. P. and Wolpin, K. I. (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): 648-672.

Contents:

CHAPTER 1

Background

`respy` is a research tool. It provides the computational support for several research projects that analyze the economics driving agents' educational and occupational choices over their life cycle within the framework of a finite-horizon discrete choice dynamic programming model.

Here is some of the recent work:

- Eisenhauer, P. (2016). The Approximate Solution of Finite-Horizon Discrete Choice Dynamic Programming Models: Revisiting Keane & Wolpin (1994). *Unpublished Manuscript*.

  > The estimation of finite-horizon discrete choice dynamic programming models is computationally expensive. This limits their realism and impedes verification and validation efforts. Keane & Wolpin (1994) propose an interpolation method that ameliorates the computational burden but introduces approximation error. I describe their approach in detail, successfully recompute their original quality diagnostics, and provide some additional insights that underscore the trade-off between computation time and the accuracy of estimation results.

- Eisenhauer, P. (2016). Risk and Ambiguity in Dynamic Models of Educational Choice. *Unpublished Manuscript*.

  > I instill a fear of model misspecification into the agents of a finite-horizon discrete choice dynamic programming model. Agents are ambiguity averse and seek robust decisions for a variety of alternative models. I study the implications for agents' decisions and the design and impact of alternative policies.

We provide the package and its documentation to ensure the recomputability, transparency, and extensibility of this research. We also hope to showcase how software engineering practices can help in achieving these goals.

The rest of this documentation is structured as follows. First, we provide the installation instructions. Then we present the underlying economic model and discuss its solution and estimation. Next, we illustrate the basic capabilities of the package in a tutorial. We continue by providing more details regarding the numerical components of the package and showcase the package's reliability and scalability. Finally, we outline the software engineering practices adopted for the ongoing development of the package.

CHAPTER 2

Installation

The `respy` package can be conveniently installed from the Python Package Index (PyPI) or directly from its source files. We currently support Python 2.7 and Python 3.3+. We develop the package on Linux systems, but it can also be installed on MacOS and Windows.

2.1 Python Package Index

You can install the stable version of the package the usual way.

```
$ pip install respy
```

We provide a pure Python implementation as our baseline. However, to address performance constraints, we also maintain scalar and parallel Fortran implementations. If additional requirements are met, both are installed automatically.

2.1.1 ... adding Fortran

Please make sure that the `gfortran` compiler is available on your path and it knows where to find the Linear Algebra PACKage (LAPACK) library.

On Ubuntu systems, both can be achieved by the following commands:

```
$ sudo apt-get install gfortran
$ sudo apt-get install libblas-dev liblapack-dev
```

If so, just call a slightly modified version of the installation command.

```
$ pip install --no-binary respy respy
```

The *–no-binary* flag is required for now to avoid the use of Python Wheels and ensure a compilation of the Fortran source code during the build.

2.1.2 ... adding Parallelism

We use the Message Passing Interface (MPI) library. This requires a recent version of its MPICH implementation available on your compiler's search path which was build with shared/dynamic libraries.

2.2 Source Files

You can download the sources directly from our GitHub repository.

```
$ git clone https://github.com/restudToolbox/package.git
```

Once you obtained a copy of the source files, installing the package in editable model is straightforward.

```
$ pip install -e .
```

2.3 Test Suite

Please make sure that the package is working properly by running our test suite using `pytest`.

```
$ python -c "import respy; respy.test()"
```

CHAPTER 3

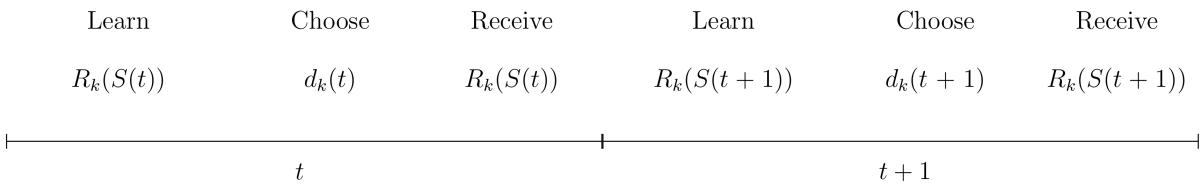
Setup

We now start with the economics motivating the model and then turn to the solution and estimation approach. We conclude with a discussion of a simulated example.

3.1 Economics

Keane and Wolpin (1994) develop a model in which an agent decides among K possible alternatives in each of T (finite) discrete periods of time. Alternatives are defined to be mutually exclusive and $d_k(t) = 1$ indicates that alternative k is chosen at time t and $d_k(t) = 0$ indicates otherwise. Associated with each choice is an immediate reward $R_k(S(t))$ that is known to the agent at time t but partly unknown from the perspective of periods prior to t. All the information known to the agent at time t that affects immediate and future rewards is contained in the state space $S(t)$.

We depict the timing of events below. At the beginning of period t the agent fully learns about all immediate rewards, chooses one of the alternatives and receives the corresponding benefits. The state space is then updated according to the agent's state experience and the process is repeated in $t + 1$.



Agents are forward looking. Thus, they do not simply choose the alternative with the highest immediate rewards each period. Instead, their objective at any time τ is to maximize the expected rewards over the remaining time horizon:

$$\max_{\{d_k(t)\}_{k \in K}} E \left[\sum_{\tau=t}^{T} \delta^{\tau - t} \sum_{k \in K} R_k(\tau) d_k(\tau) \middle| S(t) \right]$$

The discount factor $0 > \delta > 1$ captures the agent's preference for immediate over future rewards. Agents maximize the equation above by choosing the optimal sequence of alternatives $\{d_k(t)\}_{k \in K}$ for $t = \tau, .., T$.

Within this more general framework, Keane and Wolpin (1994) consider the case where agents are risk neutral and each period choose to work in either of two occupations ($k = 1, 2$), to attend school ($k = 3$), or to remain at home ($k = 4$). The immediate reward functions are given by:

$$
\begin{aligned}
R_1(t) &= w_{1t} = \exp\{\alpha_{10} + \alpha_{11} s_t + \alpha_{12} x_{1t} - \alpha_{13} x_{1t}^2 + \alpha_{14} x_{2t} - \alpha_{15} x_{2t}^2 + \epsilon_{1t}\} \\
R_2(t) &= w_{2t} = \exp\{\alpha_{20} + \alpha_{21} s_t + \alpha_{22} x_{1t} - \alpha_{23} x_{1t}^2 + \alpha_{24} x_{2t} - \alpha_{25} x_{2t}^2 + \epsilon_{2t}\} \\
R_3(t) &= \beta_0 - \beta_1 I(s_t \geq 12) - \beta_2 (1 - d_3(t-1)) + \epsilon_{3t} \\
R_4(t) &= \gamma_0 + \epsilon_{4t},
\end{aligned}
$$

where s_t is the number of periods of schooling obtained by the beginning of period t, x_{1t} is the number of periods that the agent worked in occupation one by the beginning of period t, x_{2t} is the analogously defined level of experience in occupation two, α_1 and α_2 are parameter vectors associated with the wage functions, β_0 is the consumption value of schooling, β_1 is the post-secondary tuition cost of schooling, with I an indicator function equal to one if the agent has completed high school and zero otherwise, β_2 is an adjustment cost associated with returning to school, γ_0 is the (mean) value of the non-market alternative. The ϵ_{kt}'s are alternative-specific shocks to occupational productivity, to the consumption value of schooling, and to the value of non-market time. The productivity and taste shocks follow a four-dimensional multivariate normal distribution with mean zero and covariance matrix $\Sigma = [\sigma_{ij}]$. The realizations are independent across time. We collect the parametrization of the reward functions in $\theta = \{\alpha_1, \alpha_2, \beta, \gamma, \Sigma\}$.

Given the structure of the reward functions and the agents objective, the state space at time t is:

$$
S(t) = \{s_t, x_{1t}, x_{2t}, d_3(t-1), \epsilon_{1t}, \epsilon_{2t}, \epsilon_{3t}, \epsilon_{4t}\}.
$$

It is convenient to denote its observable elements as $\bar{S}(t)$. The elements of $S(t)$ evolve according to:

$$
\begin{aligned}
x_{1,t+1} &= x_{1t} + d_1(t) \\
x_{2,t+1} &= x_{2t} + d_2(t) \\
s_{t+1} &= s_t \ \ + d_3(t) \\
f(\epsilon_{t+1} \mid S(t), d_k(t)) &= f(\epsilon_{t+1} \mid \bar{S}(t), d_k(t)),
\end{aligned}
$$

where the last equation reflects the fact that the ϵ_{kt}'s are serially independent. We set $x_{1t} = x_{2t} = 0$ as the initial conditions.

3.2 Solution

From a mathematical perspective, this type of model boils down to a finite-horizon DP problem under uncertainty that can be solved by backward induction. For the discussion, it is useful to define the value function $V(S(t), t)$ as a shorthand for the agents objective function. $V(S(t), t)$ depends on the state space at t and on t itself due to the finiteness of the time horizon and can be written as:

$$
V(S(t), t) = \max_{k \in K} \{V_k(S(t), t)\}, \tag{3.1}
$$

with $V_k(S(t), t)$ as the alternative-specific value function. $V_k(S(t), t)$ obeys the Bellman equation (Bellman, 1957) and is thus amenable to a backward recursion.

$$
V_k(S(t), t) = \begin{cases} R_k(S(t)) + \delta E\left[V(S(t+1), t+1) \mid S(t), d_k(t) = 1\right] & \text{if } t < T \\ R_k(S(t)) & \text{if } t = T. \end{cases} \tag{3.2}
$$

Assuming continued optimal behavior, the expected future value of state $S(t+1)$ for all K alternatives given today's state $S(t)$ and choice $d_k(t) = 1$, $E \max(S(t+1))$ for short, can be calculated:

$$
E \max(S(t+1)) = E\left[V(S(t+1), t+1) \mid S(t), d_k(t) = 1\right].
$$

This requires the evaluation of a K - dimensional integral as future rewards are partly uncertain due to the unknown realization of the shocks:

$$E \max(S(t)) = \int_{\epsilon_1(t)} ... \int_{\epsilon_K(t)} \max\{R_1(t), ..., R_K(t)\} f_\epsilon(\epsilon_1(t), ..., \epsilon_K(t)) d\epsilon_1(t) ... d\epsilon_K(t),$$

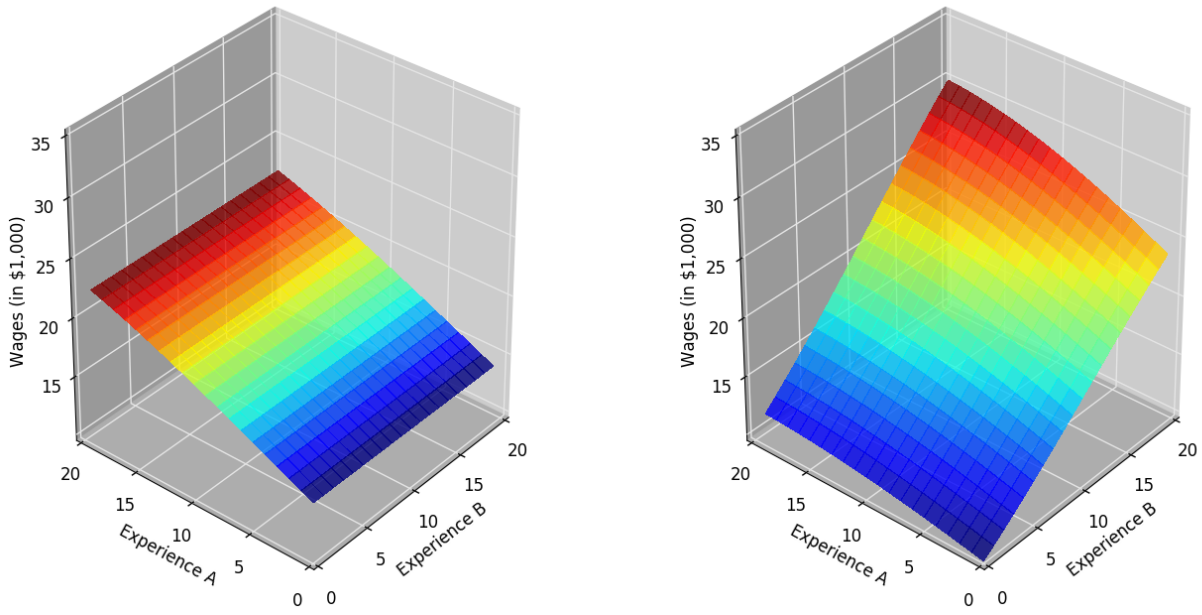
where f_ϵ is the joint density of the uncertain component of the rewards in t not known at $t-1$. With all ingredients at hand, the solution of the model by backward induction is straightforward.

3.3 Estimation

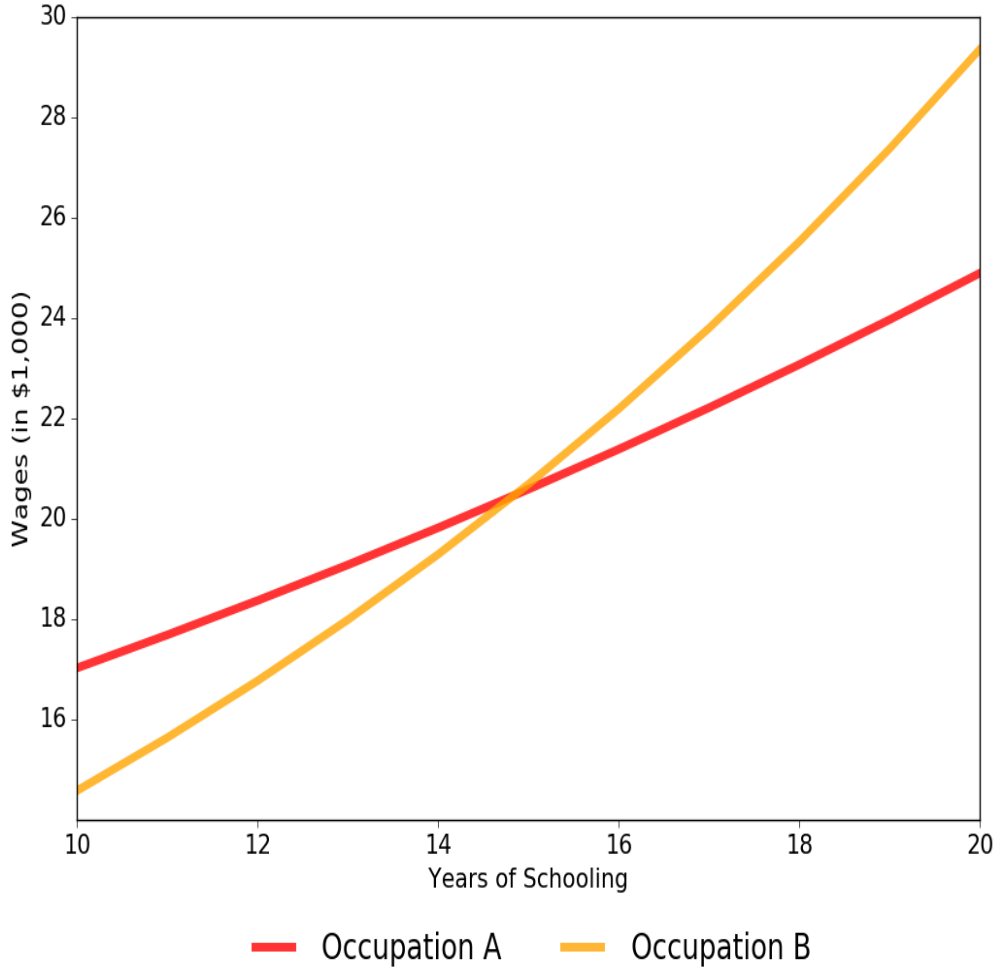
We estimate the parameters of the reward functions θ based on a sample of agents whose behavior and state experiences are described by the model. Although all shocks to the rewards are eventually known to the agent, they remain unobserved by the econometrician. So each parameterization induces a different probability distribution over the sequence of observed agent choices and their state experience. We implement maximum likelihood estimation and appraise each candidate parameterization of the model using the likelihood function of the observed sample (Fisher, 1922). Given the serial independence of the shocks, We can compute the likelihood contribution by agent and period. The sample likelihood is then just the product of the likelihood contributions over all agents and time periods. As we need to simulate the agent's choice probabilities, we end up with a simulated maximum likelihood estimator (Manski and Lerman, 1977) and minimize the simulated negative log-likelihood of the observed sample.

3.4 Simulated Example

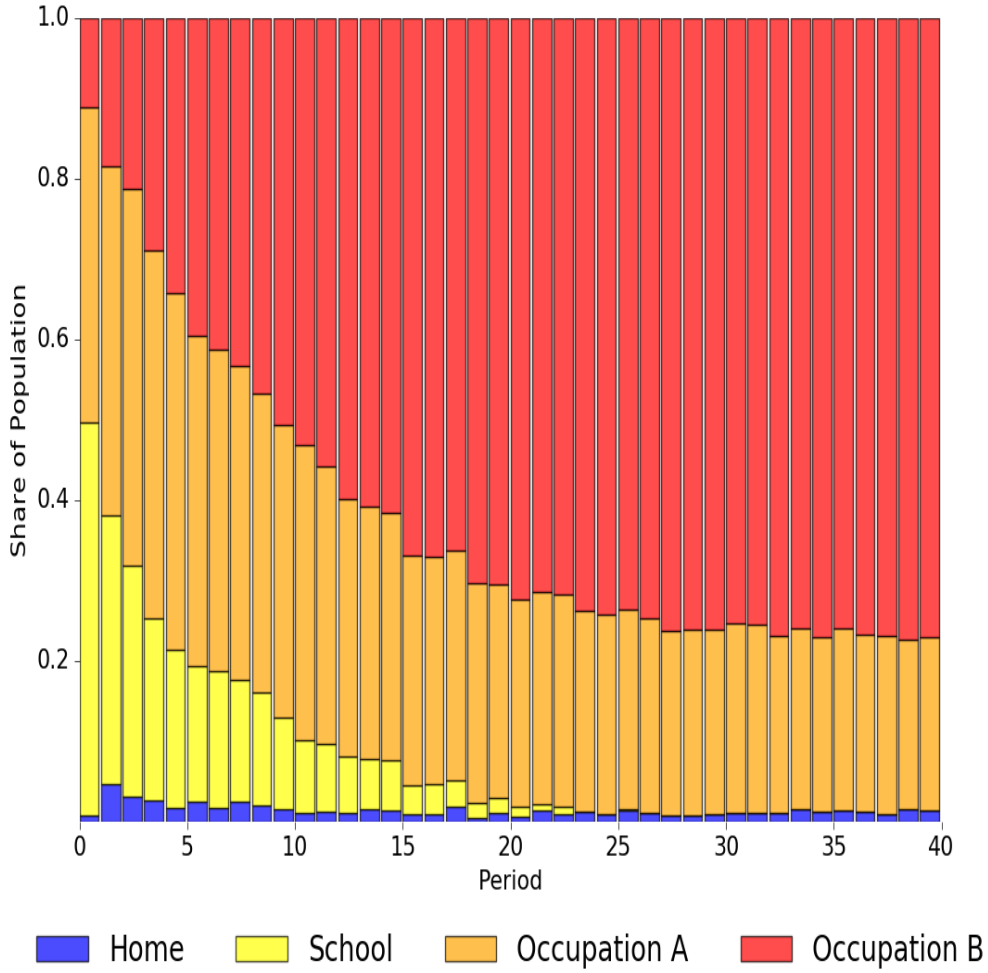
Keane and Wolpin (1994) generate three different Monte Carlo samples. We study their first parameterization in more detail now. We label the two occupations as Occupation A and Occupation B. We first plot the returns to experience. Occupation B is more skill intensive in the sense that own experience has higher return than is the case for Occupation A. There is some general skill learned in Occupation A which is transferable to Occupation B. However, work experience in is occupation-specific in Occupation B.



The next figure shows that the returns to schooling are larger in Occupation B. While its initial wage is lower, it does decrease faster with schooling compared to Occupation A.



Simulating a sample of 1,000 agents from the model allows us to study how these features interact in determining agent decisions over their life cycle. Note that all agents start out identically, different choices are simply the cumulative effects of different shocks. Initially, 50% of agents increase their level of schooling but the share of agents enrolled in school declines sharply over time. The share working in Occupation A hovers around 40% at first, but then declines to 21%. Occupation B continuously gains in popularity, initially only 11% work in Occupation B but its share increases to about 77%. Around 1.5% stay at home each period. We visualize this choice pattern in detail below.



We start out with the large majority of agents working in Occupation A. Eventually, however, most agents ends up working in Occupation B. As the returns to education are higher for Occupation B and previous work experience is transferable, Occupation B gets more and more attractive as agents increase their level of schooling and gain experience in the labor market.

CHAPTER 4

Tutorial

We now illustrate the basic capabilities of the `respy` package. We start with the model specification and then turn to some example use cases.

4.1 Model Specification

The model is specified in an initialization file. For an example, check out the first parameterization analyzed in Keane and Wolpin (1994) here. Let us discuss each of its elements in more detail.

BASICS

Key	Value	Interpretation
periods	int	number of periods
delta	float	discount factor

Warning: There are two small differences compared to Keane and Wolpin (1994). First, all coefficients enter the return function with a positive sign, while the squared terms enter with a minus in the original paper. Second, the order of covariates is fixed across the two occupations. In the original paper, own experience always comes before other experience.

OCCUPATION A

Key	Value	Interpretation
coeff	float	intercept
coeff	float	return to schooling
coeff	float	experience Occupation A, linear
coeff	float	experience Occupation A, squared
coeff	float	experience Occupation B, linear
coeff	float	experience Occupation B, squared

OCCUPATION B

Key	Value	Interpretation
coeff	float	intercept
coeff	float	return to schooling
coeff	float	experience Occupation A, linear
coeff	float	experience Occupation A, squared
coeff	float	experience Occupation B, linear
coeff	float	experience Occupation B, squared

EDUCATION

Key	Value	Interpretation
coeff	float	consumption value
coeff	float	tuition cost
coeff	float	adjustment cost
max	int	maximum level of schooling
start	int	initial level of schooling

Warning: Again, there is a small difference between this setup and Keane and Wolpin (1994). There is no automatic change in sign for the tuition and adjustment costs. Thus, a $1,000 tuition cost must be specified as -1000.

HOME

Key	Value	Interpretation
coeff	float	mean value of non-market alternative

SHOCKS

Key	Value	Interpretation
coeff	float	σ_1
coeff	float	σ_{12}
coeff	float	σ_{13}
coeff	float	σ_{14}
coeff	float	σ_2
coeff	float	σ_{23}
coeff	float	σ_{24}
coeff	float	σ_3
coeff	float	σ_{34}
coeff	float	σ_4

SOLUTION

Key	Value	Interpretation
draws	int	number of draws for $E\max$
store	bool	persistent storage of results
seed	int	random seed for $E\max$

SIMULATION

Key	Value	Interpretation
file	str	file to print simulated sample
agents	int	number of simulated agents
seed	int	random seed for agent experience

ESTIMATION

Key	Value	Interpretation
file	str	file to read observed sample
tau	float	scale parameter for function smoothing
agents	int	number of agents to read from sample
draws	int	number of draws for choice probabilities
maxfun	int	maximum number of function evaluations
seed	int	random seed for choice probability
optimizer	str	optimizer to use

PROGRAM

Key	Value	Interpretation
debug	bool	debug mode
version	str	program version

PARALLELISM

Key	Value	Interpretation
flag	bool	parallel executable
procs	int	number of processors

INTERPOLATION

Key	Value	Interpretation
points	int	number of interpolation points
flag	bool	flag to use interpolation

DERIVATIVES

Key	Value	Interpretation
version	str	approximation scheme
eps	float	step size

SCALING

Key	Value	Interpretation
flag	bool	apply scaling to parameters
minimum	float	minimum value for gradient approximation

The implemented optimization algorithms vary with the program's version. If you request the Python version of the program, you can choose from the `scipy` implementations of the BFGS (Norcedal and Wright, 2006) and POWELL

(Powell, 1964) algorithm. Their implementation details are available here. For Fortran, we implemented the BFGS and NEWUOA (Powell, 2004) algorithms.

SCIPY-BFGS

Key	Value	Interpretation
gtol	float	gradient norm must be less than gtol before successful termination
maxiter	int	maximum number of iterations

SCIPY-POWELL

Key	Value	Interpretation
maxfun	int	maximum number of function evaluations to make
ftol	float	relative error in func(xopt) acceptable for convergence
xtol	float	line-search error tolerance

SCIPY-LBFGSB

Key	Value	Interpretation
eps	float	Step size used when approx_grad is True, for numerically calculating the gradient
factr	float	Multiple of the default machine precision used to determine the relative error in func(xopt) acceptable for convergence
m	int	Maximum number of variable metric corrections used to define the limited memory matrix.
maxiter	int	maximum number of iterations
maxls	int	Maximum number of line search steps (per iteration). Default is 20.
pgtol	float	gradient norm must be less than gtol before successful termination

FORT-BFGS

Key	Value	Interpretation
gtol	float	gradient norm must be less than gtol before successful termination
maxiter	int	maximum number of iterations

FORT-NEWUOA

Key	Value	Interpretation
maxfun	float	maximum number of function evaluations
npt	int	number of points for approximation model
rhobeg	float	starting value for size of trust region
rhoend	float	minimum value of size for trust region

FORT-BOBYQA

Key	Value	Interpretation
maxfun	float	maximum number of function evaluations
npt	int	number of points for approximation model
rhobeg	float	starting value for size of trust region
rhoend	float	minimum value of size for trust region

4.2 Constraints for the Optimizer

If you want to keep any parameter fixed at the value you specified (i.e. not estimate this parameter) you can simply add an exclamation mark after the value. If you want to provide bounds for a constrained optimizer you can specify a lower and upper bound in round brackets. A section of such an .ini file would look as follows:

```
coeff             -0.049538516229344
coeff              0.020000000000000     !
coeff             -0.037283956168153       (-0.5807488086366478,None)
coeff              0.036340835226155     ! (None,0.661243603948984)
```

In this example, the first coefficient is free. The second one is fixed at 0.2. The third one will be estimated but has a lower bound. In the fourth case, the parameter is fixed and the bounds will be ignored.

If you specify bounds for any free parameter, you have to choose a constraint optimizer such as SCIPY-LBFGSB or FORT-BOBYQA.

4.3 Dataset

To use respy, you need a dataset with the following columns:

- Identifier: identifies the different individuals in the sample
- Period: identifies the different rounds of observation for each individual
- **Choice: an integer variable that indicates the labor market choice**
  - 1 = Occupation A
  - 2 = Occupation B
  - 3 = Education
  - 4 = Home
- Earnings: a float variable that indicates how much people are earning. This variable is missing (indicated by a dot) if individuals don't work.
- Experience_A: labor market experience in sector A
- Experience_B: labor market experience in sector B
- Years_Schooling: years of schooling
- Lagged_Choice: choice in the period before the model starts. Codes are the same as in Choice.

Datasets for respy are stored in simple text files, where columns are separated by spaces. The easiest way to write such a text file in Python is to create a pandas DataFrame with all relevant columns and then storing it in the following way:

```
with open('my_data.respy.dat', 'w') as file:
    df.to_string(file, index=False, header=True, na_rep='.')
```

4.4 Examples

Let us explore the basic capabilities of the `respy` package with a couple of examples. All the material is available online.

Simulation and Estimation

We always first initialize an instance of the `RespyCls` by passing in the path to the initialization file.

```
from respy import RespyCls

respy_obj = RespyCls('example.ini')
```

Now we can simulate a sample from the specified model.

```
respy_obj.simulate()
```

During the simulation, several files will appear in the current working directory. `sol.respy.log` allows to monitor the progress of the solution algorithm, while `sim.respy.log` records the progress of the simulation. The simulated dataset with the agents' choices and state experiences is stored in `data.respy.dat`, `data.respy.info` provides some basic descriptives about the simulated dataset. See our section on *Additional Details* for more information regarding the output files.

Now that we simulated some data, we can start an estimation. Here we are using the simulated data for the estimation. However, you can of course also use other data sources. Just make sure they follow the layout of the simulated sample. The coefficient values in the initialization file serve as the starting values.

```
x, crit_val = respy_obj.fit()
```

This directly returns the value of the coefficients at the final step of the optimizer as well as the value of the criterion function. However, some additional files appear in the meantime. Monitoring the estimation is best done using `est.respy.info` and more details about each evaluation of the criterion function are available in `est.respy.log`.

We can now simulate a sample using the estimated parameters by updating the instance of the `RespyCls`.

```
respy_obj.update_model_paras(x)

respy_obj.simulate()
```

Recomputing Keane and Wolpin (1994)

Just using the capabilities outlined so far, it is straightforward to recompute some of the key results in the original paper with a simple script.

```
#!/usr/bin/env python
""" This module recomputes some of the key results of Keane and Wolpin (1994).
"""

from respy import RespyCls

# We can simply iterate over the different model specifications outlined in
# Table 1 of their paper.
for spec in ['kw_data_one.ini', 'kw_data_two.ini', 'kw_data_three.ini']:

    # Process relevant model initialization file
    respy_obj = RespyCls(spec)

    # Let us simulate the datasets discussed on the page 658.
    respy_obj.simulate()

    # To start estimations for the Monte Carlo exercises. For now, we just
    # evaluate the model at the starting values, i.e. maxfun set to zero in
    # the initialization file.
    respy_obj.unlock()
    respy_obj.set_attr('maxfun', 0)
```

(continues on next page)

(continued from previous page)

```
    respy_obj.lock()

    respy_obj.fit()
```

In an earlier working paper, Keane and Wolpin (1994b) provide a full account of the choice distributions for all three specifications. The results from the recomputation line up well with their reports.

CHAPTER 5

Numerical Methods

The `respy` package contains several numerical components. We discuss each in turn.

5.1 Differentiation

Derivatives are approximated by forward finite differences and used by derivative-based optimization algorithms and the scaling procedure. The step-size can be controlled in the *DERIVATIVES* section of the initialization file.

5.2 Integration

Integrals are approximated by Monte Carlo integration and occur in two different places:

- The solution of the model requires the evaluation of $E\max$. This integral is approximated using the number of random draws specified in the *SOLUTION* section of the initialization file. The same random draws are used for all integrals within the same period.
- The estimation of the model requires the simulation of the choice probabilities to evaluate the sample likelihood. This integral is approximated using the number of random draws specified in the *ESTIMATION* section of the initialization file. The same random draws are used for all integrals within the same period.

5.3 Optimization

The estimation of the model involves the minimization of the simulated negative log-likelihood of the sample. The available optimizers depend on the version of the program. If you use the Python implementation, then the Powell (Powell, 1964) and BFGS (Norcedal and Wright, 2006) algorithms are available through their `scipy` implementations. For the Fortran implementation, we provide the BFGS and NEWUOA (Powell, 2004) algorithms. The algorithm to be used is specified in the *ESTIMATION* section of the initialization file.

- **Preconditioning**

We implemented a diagonal scale-based preconditioner based on the gradient. To stabilize the routine, the user needs to specify a minimum value for the derivative approximation. The details are governed by the *SCALING* section of the initialization file.

5.4 Function Approximation

We follow Keane and Wolpin (1994) and allow to alleviate the computational burden by calculating the $E\max$ only at a subset of states each period and interpolating its value for the rest. We implement their proposed interpolation function:

$$E\max - \max E = \pi_0 + \sum_{j=1}^{4} \pi_{1j}(\max E - \bar{V}_j) + \sum_{j=1}^{4} \pi_{2j} \left(\max E - \bar{V}_j\right)^{\frac{1}{2}} . \tag{5.1}$$

$\bar{V}_j$ is shorthand for the expected value of the alternative-specific value function and $\max E = \max_k\{\bar{V}_j\}$ is its maximum among the choices available to the agent. The π's are time-varying as they are estimated by ordinary least squares each period. The subset of interpolation points for the interpolating function is chosen at random for each period. The number of interpolation points remains constant across all periods. The number of interpolation points is selected in the *INTERPOLATION* section of the initialization file.

5.5 Function Smoothing

We simulate the agents' choice probabilities to evaluate the negative log-likelihood of the sample. With only a finite number of draws, there is always the risk of simulating a zero probability for an agent's observed decision. So we implement the logit-smoothed accept-reject simulator as suggested by McFadden (1989). The scale parameter λ is set in the *ESTIMATION* section of the initialization file.

5.6 Miscellaneous

We use the LAPACK library for all numerical algebra. The generation of pseudorandom numbers differs between the Python and Fortran implementations. While they are generated by the Mersenne Twister (Matsumoto and Nishimura, 1998) in Python, we rely on the George Marsaglia's KISS generator (Marsaglia, 1968) in Fortran.

CHAPTER 6

Reliability

We document the results of two straightforward Monte Carlo exercises to illustrate the reliability of the `respy` package. We use the first parameterization from Keane and Wolpin (1994) and simulate a sample of 1,000 agents. Then we run two estimations with alternative starting values. We use the root-mean squared error (RMSE) of the simulated choice probabilities to assess the estimator's overall reliability. We use the NEWUOA algorithm with its default tuning parameters and allow for a maximum of 3,000 evaluations of the criterion function.

6.1 ... starting at true values

Initially we start at the true parameter values. While taking a total of 1,491 steps, the actual effect on the parameter values and the criterion function is negligible. The RMSE remains literally unchanged at zero.

Start	Stop	Steps	Evaluations
0.00	0.00	1,491	3,000

6.2 ... starting with myopic agents

Again we start from the true parameters of the reward functions, but now estimate a static ($\delta = 0$) model first. We then use the estimation results as starting values for the subsequent estimation of a correctly specified dynamic model ($\delta = 0.95$). For the static estimation, we start with a RMSE of about 0.44 which, after 950 steps, is cut to 0.25. Most of this discrepancy is driven by the relatively low school enrollments as there is no investment motive for the myopic agents.

Start	Stop	Steps	Evaluations
0.44	0.25	950	3,000

We then set up the estimation of the dynamic model. Initially, the RMSE is about 0.23 but is quickly reduced to only 0.01 after 1,453 steps of the optimizer.

Start	Stop	Steps	Evaluations
0.23	0.01	1,453	3,000

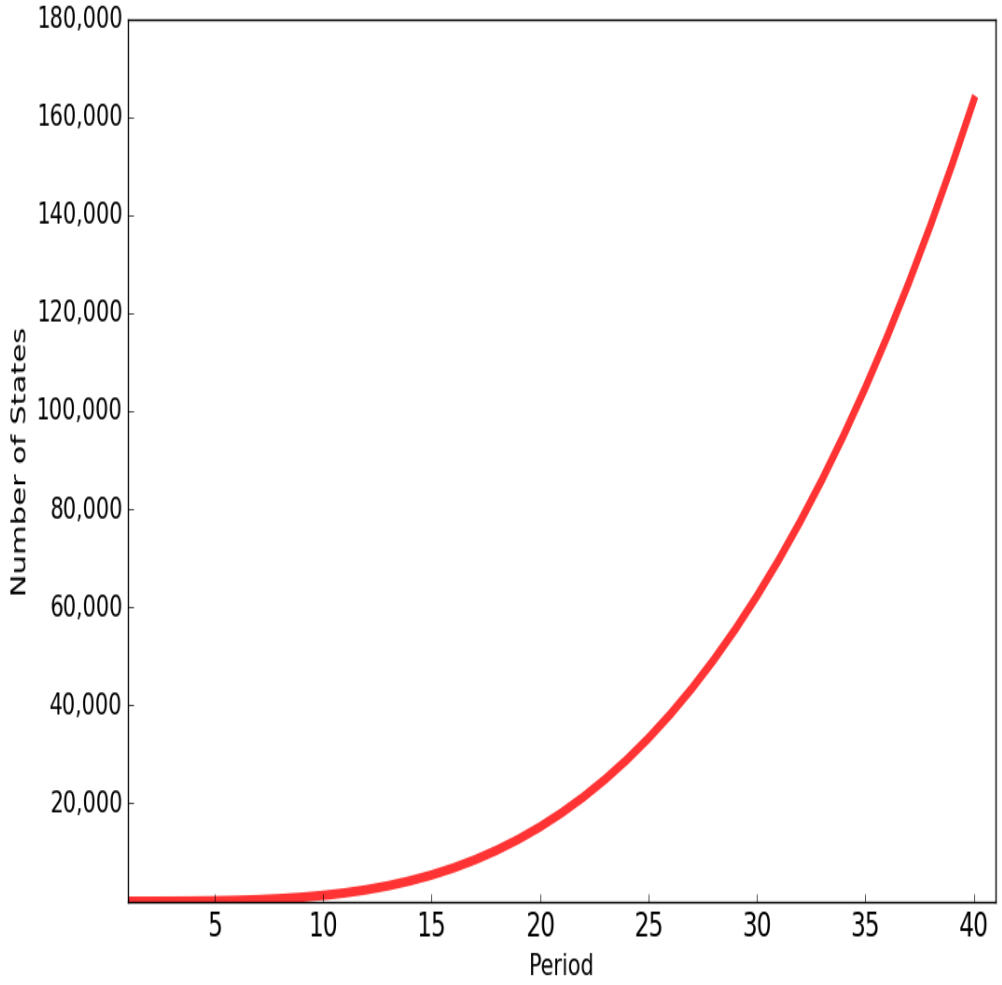
Overall the results are encouraging. However, doubts about the correctness of our implementation always remain. So, if you are struggling with a particularly poor performance in your application, please do not hesitate to let us know so we can help with the investigation.

For more details, see the script online. The results for all the parameterizations analyzed in Keane and Wolpin (1994) are available here.

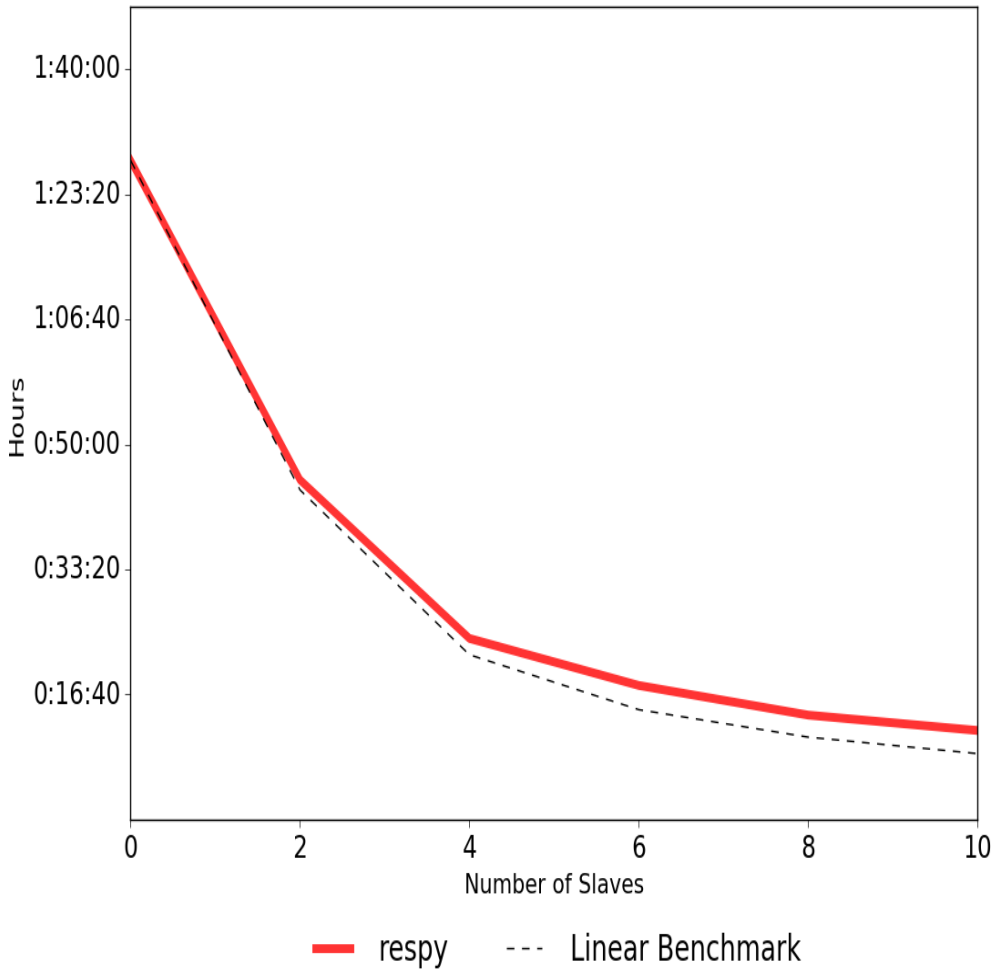
CHAPTER 7

Scalability

The solution and estimation of finite-horizon discrete choice dynamic programming model appears straightforward. However, it entails a considerable computational burden due to the well known curse of dimensionality (Bellman and Dreyfus, 1962). The figure below illustrates how the total number of states increases exponentially with each period.



During an estimation, thousands of different candidate parameterizations of the model are appraised with respect to the sample likelihood. Each time we need to evaluate the four-dimensional integral of $E\max$ at a total of 163,410 states. Thus, in addition to Python, we also maintain a scalar and parallel Fortran implementation. We parallelize the workload using the master-slave paradigm. We assign each slave a subset of states to evaluate the $E\max$ and a subset of agents to simulate their choice probabilities. Below, we show the total computation time required for 1,000 evaluations of the criterion function as we increase the number of slave processors to ten. Judging against the linear benchmark, the code scales well over this range.



Adding even more processors, however, does not lead to any further improvements, it even increases the computational time. The main reason is the time spend on the synchronization of $E\max$ across all processes each period. Even though each slave is only working on a subset of states each period, they need access all previous $E\max$ results during the backward induction procedure.

For more details, see the script online and the logfile.

CHAPTER 8

Software Engineering

We now briefly discuss our software engineering practices that help us to ensure the transparency, reliability, scalability, and extensibility of the `respy` package.

8.1 Development Infrastructure

We maintain a dedicated development and testing server on the Amazon Elastic Compute Cloud. We treat our infrastructure as code thus making it versionable, testable, and repeatable. We create our machine images using Packer and Chef and manage our compute resources with Terraform. Our definition files are available here.

8.2 Program Design

We build on the design of the original authors (codes). We maintain a pure Python implementation with a focus on readability and a scalar and parallel Fortran implementation to address any performance constraints. We keep the structure of the Python and Fortran implementation aligned as much as possible. For example, we standardize the naming and interface design of the routines across versions.

8.3 Test Battery

We use pytest as our test runner. We broadly group our tests in four categories:

- **property-based testing**

  We create random model parameterizations and estimation requests and test for a valid return of the program. For example, we estimate the same model specification using the parallel and scalar implementations as both results need to be identical. Also, we maintain a an `f2py` interface to ensure that core functions of our Python and Fortran implementation return the same results. Finally, we also upgraded the codes by Keane and Wolpin (1994) and can compare the results of the `respy`

package with their implementation for a restricted set of estimation requests that are valid for both programs.

- **regression testing**

  We retain a set of 10,000 fixed model parameterizations and store their estimation results. This allows to ensure that a simple refactoring of the code or the addition of new features does not have any unintended consequences on the existing capabilities of the package.

- **scalability testing**

  We maintain a scalar and parallel Fortran implementation of the package, we regularly test the scalability of our code against the linear benchmark.

- **reliability testing**

  We conduct numerous Monte Carlo exercises to ensure that we can recover the true underlying parameterization with an estimation. Also by varying the tuning parameters of the estimation (e.g. random draws for integration) and the optimizers, we learn about their effect on estimation performance.

- **release testing**

  We thoroughly test new release candidates against previous releases. For minor and micro releases, user requests should yield identical results. For major releases, our goal is to ensure that the same is true for at least a subset of requests. If required, we will build supporting code infrastructure.

- **robustness testing**

  Numerical instabilities often only become apparent on real world data that is less well behaved than simulated data. To test the stability of our package we start thousands of estimation tasks on the NLSY dataset used by Keane and Wolpin. We use random start values for the parameter vector that can be far from the true values and make sure that the code can handle those cases.

Our tests and the testing infrastructure are available online. As new features are added and the code matures, we constantly expand our testing harness. We run a test battery nightly on our development server, see here for an example output.

8.4 Documentation

The documentation is created using Sphinx and hosted on Read the Docs.

8.5 Code Review

We use several automatic code review tools to help us improve the readability and maintainability of our code base. For example, we work with Codacy and Landscape

8.6 Continuous Integration Workflow

We set up a continuous integration workflow around our GitHub Organization. We use the continuous integration services provided by Travis CI. tox helps us to ensure the proper workings of the package for alternative Python implementations. Our build process is managed by Waf. We rely on Git as our version control system and follow the Gitflow Workflow. We use GitLab for our issue tracking. The package is distributed through PyPI which automatically updated from our development server.

CHAPTER 9

Contributing

Great, you are interesting in contributing to the package. Please announce your interest on our mailing list so we can find you something to work on.

To get acquainted with the code base, you can check out our issue tracker for some immediate and clearly defined tasks. For more involved contributions, please see our roadmap below. Feel free to set up your development infrastructure using our Amazon Machine Image or Chef cookbook.

9.1 Roadmap

We aim for improvements to `respy` in three domains: Economics, Software Engineering, and Numerical Methods.

9.1.1 Economics

- support the full model of Keane and Wolpin (1997)

9.1.2 Software Engineering

- explore *Autotools* as a new build system
- research the *hypothesis* package to replace the hand-crafted property-based testing routines

9.1.3 Numerical Methods

- link the package to optimization toolkits such as *TAO* or *HOPSPACK*
- implement additional integration strategies following Skrainka and Judd (2011)

CHAPTER 10

Additional Details

10.1 Output Files

Depending on the user's request, the `respy` package creates several output files.

Warning: There is a slight difference between the estimation parameters in the files below and the model specification. The difference is in the parameters for the covariance matrix. During the estimation we iterate on a flattened version of the upper-triangular Cholesky decomposition. This ensures that the requirements for a valid covariance matrix, e.g. positive semidefiniteness and strictly positive variances, are always met as the optimizer appraises alternative model parameterizations.

10.1.1 Simulation

- **data.respy.dat**

This file contains the agent choices and state experiences. The simulated dataset has the following structure.

Column	Information
1	agent identifier
2	time period
3	choice (1 = Occupation A, 2 = Occupation B, 3 = education, 4 = home)
4	wages (missing value if not working)
5	work experience in Occupation A
6	work experience in Occupation B
7	years of schooling
8	lagged schooling

- **data.respy.info**

This file provides descriptive statistics such as the choice probabilities and the wage distributions. It also prints out the underlying parameterization of the model.

- **sim.respy.log**

This file allows to monitor the progress of the simulation. It provides information about the seed used to sample the random components of the agents' state experience and the total number of simulated agents.

- **sol.respy.log**

This file records the progress of the backward induction procedure. If the interpolation method is used during the backward induction procedure, the coefficient estimates and goodness of fit statistics are provided.

- **solution.respy.pkl**

This file is an instance of the `RespyCls` and contains detailed information about the solution of model such as the $E\max$ of each state for example. For details, please consult the source code directly. It is created if persistent storage of results is requested in the *SOLUTION* section of the initialization file.

10.1.2 Estimation

- **est.respy.info**

This file allows to monitor the estimation as it progresses. It provides information about starting values, step values, and current values as well as the corresponding value of the criterion function.

- **est.respy.log**

This file documents details about each of the evaluations of the criterion function. Most importantly, once an estimation is completed, it provides the return message from the optimizer.

10.2 API Reference

The API reference provides detailed descriptions of `respy` classes and functions. It should be helpful if you plan to extend `respy` with custom components.

class respy.**RespyCls**(*fname*)

Class to process and manage the user's initialization file.

Parameters **fname** (*str*) – Path to initialization file

Returns Instance of RespyCls

classmethod **update_model_paras**(*x*)

Function to update model parameterization.

Parameters **x** (*numpy.ndarray*) – Model parameterization

CHAPTER 11

Developer Documentation

11.1 The pre_processing directory

This directory contains code to process the model specifications and estimation dataset. The code here is Python only. Speed is irrelevant since most of the functions here are only called once (when a new model is defined) and not each time the model is solved.

11.1.1 The model_processing module

CHAPTER 12

Contact and Credits

If you have any questions or comments, please do not hesitate to contact us directly.

12.1 Development Lead

Philipp Eisenhauer

12.2 Contributors

Janos Gabler

12.3 Acknowledgments

We are grateful to the Social Science Computing Services at the University of Chicago who let us use the Acropolis cluster for scalability and performance testing. We appreciate the financial support of the AXA Research Fund and the University of Bonn. We are indebted to the open source community as we build on top of numerous open source tools such as the SciPy Stack, statsmodels, and waf.

12.4 Suggested Citation

We appreciate citations for `respy` because it helps us to find out how people have been using the package and it motivates further work. Please use our Digital Object Identifier (DOI) and see here for other citation styles.

CHAPTER 13

Changes

This is a record of all past `respy` releases and what went into them in reverse chronological order. We follow semantic versioning and all releases are available on PyPI.

13.1 1.0.0 - 2016-09-01

This is the initial release of the `respy` package.

CHAPTER 14

Bibliography

Bellman, R. (1957). Dynamic Programming. *Princeton University Press*, Princeton, NJ.

Bellman, R. and Dreyfus, S. E. (1962). Applied Dynamic Programming. *Princeton University Press*, Princeton, NJ.

Eisenhauer, P. and Wild, S. M. (2016). Numerical Upgrade to Finite-Horizon Discrete Choice Programming Models. *Unpublished Manuscript*.

Eisenhauer, P. (2016). The Approximate Solution of Finite-Horizon Discrete Choice Dynamic Programming Models: Revisiting Keane & Wolpin (1994). *Unpublished Manuscript*.

Eisenhauer, P. (2016b). Risk and Ambiguity in Dynamic Models of Educational Choice. *Unpublished Manuscript*.

Fisher, R. A. (1922). On the Mathematical Foundations of Theoretical Statistics. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 222(594-604): 309-368.

Skrainka, B. S. and Judd, K. L. (2011). High Performance Quadrature Rules: How Numerical Integration Affects a Popular Model of Product Differentiation. *SSRN Working Paper*.

Keane, M. P. and Wolpin, K. I. (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): 648-672.

Keane, M. P. and Wolpin, K. I. (1994b). The Solution and Estimation of Discrete Choice Dynamic Programming Models by Simulation and Interpolation: Monte Carlo Evidence. *Federal Reserve Bank of Minneapolis*, No. 181.

Keane, M. P. and Wolpin, K. I. (1996). The Career Decisions of Young Men. *Journal of Political Economy*, 105(3): 473-522.

Manski, C. and Lerman S. (1977). The Estimation of Choice Probabilities from Choice Based Samples. *Econometrica*, 45(8): 1977-1988.

Marsaglia, G. (1968). Random Numbers fall Mainly in the Planes. *Proceedings of the National Academy of Sciences*, 61(1): 25-28.

Matsumoto, M. and Nishimura, T. (1998). Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1): 3-30.

McFadden, D. (1989). A Method of Simulated Moments for Estimation of Discrete Response Models Without Numerical Integration. *Econometrica*, 57(5):995–1026.

Nocedal, J. and Wright, S. J. (2006). Numerical Optimization. *Springer*, New York, NY.

Powell, M. J. D. (1964). An Efficient Method for Finding the Minimum of a Function of Several Variables without Calculating Derivatives. *The Computer Journal*, 7(2): 155-162.

Powell, M. J. D. (2006). The NEWUOA Software for Unconstrained Optimization Without Derivatives. In Pillo, Gianni, R. M., editor, Large-Scale Nonlinear Optimization, number 83 in Nonconvex Optimization and Its Applications, pages 255–297. Springer, New York, NY.

Index

R

U