PyBLP

Release 1.2.0

Jeff Gortmaker and Christopher Conlon

Apr 07, 2026

TABLE OF CONTENTS

Part I

User Documentation

CHAPTER

ONE

INTRODUCTION

PyBLP is a Python 3 implementation of routines for estimating the demand for differentiated products with BLP-type random coefficients logit models. This package was created by Jeff Gortmaker in collaboration with Chris Conlon.

Development of the package has been guided by the work of many researchers and practitioners. For a full list of references, including the original work of Berry, Levinsohn, and Pakes (1995), refer to the references section of the documentation.

1.1 Citation

If you use PyBLP in your research, we ask that you also cite Conlon and Gortmaker (2020), which describes the advances implemented in the package.

```
@article{PyBLP,
    author={Conlon, Christopher and Gortmaker, Jeff},
    title={Best practices for differentiated products demand estimation with {PyBLP}},
    journal={RAND Journal of Economics},
    volume={51},
    number={4},
    pages={1108-1161},
    year={2020}
}
```

If you use micro moments with PyBLP, we ask that you also cite Conlon and Gortmaker (2025), which describes the standardized framework implemented by PyBLP for incorporating micro data into BLP-style estimation.

```
@article{MicroPyBLP,
    author={Conlon, Christopher and Gortmaker, Jeff},
    title={Incorporating micro data into differentiated products demand estimation␣
↪with {PyBLP}},
    journal={Journal of Econometrics},
    pages={105926},
    year={2025}
}
```

1.2 Installation

The PyBLP package has been tested on Python versions 3.6 through 3.9. The SciPy instructions for installing related packages is a good guide for how to install a scientific Python environment. A good choice is the Anaconda Distribution, since it comes packaged with the following PyBLP dependencies: NumPy, SciPy, SymPy, and Patsy. For

absorption of high dimension fixed effects, PyBLP also depends on its companion package PyHDFE, which will be installed when PyBLP is installed.

However, PyBLP may not work with old versions of its dependencies. You can update PyBLP's Anaconda dependencies with:

```
conda update numpy scipy sympy patsy
```

You can update PyHDFE with:

```
pip install --upgrade pyhdfe
```

You can install the current release of PyBLP with pip:

```
pip install pyblp
```

You can upgrade to a newer release with the `--upgrade` flag:

```
pip install --upgrade pyblp
```

If you lack permissions, you can install PyBLP in your user directory with the `--user` flag:

```
pip install --user pyblp
```

Alternatively, you can download a wheel or source archive from PyPI. You can find the latest development code on GitHub and the latest development documentation here.

1.3 Other Languages

Once installed, PyBLP can be incorporated into projects written in many other languages with the help of various tools that enable interoperability with Python.

For example, the reticulate package makes interacting with PyBLP in R straightforward (when supported, Python objects can be converted to their R counterparts with the `py_to_r` function, which needs to be used manually because we set `convert=FALSE` to get rid of errors about trying to automatically convert unsupported objects):

```
library(reticulate)
pyblp <- import("pyblp", convert=FALSE)
pyblp$options$flush_output <- TRUE
```

Similarly, PyCall can be used to incorporate PyBLP into a Julia workflow:

```
using PyCall
pyblp = pyimport("pyblp")
```

The py command serves a similar purpose in MATLAB:

```
py.pyblp
```

1.4 Features

- R-style formula interface
- Bertrand-Nash supply-side moments

- Multiple equation GMM
- Demographic interactions
- Product-specific demographics
- Consumer-specific product availability
- Flexible micro moments that can match statistics based on survey data
- Support for micro moments based on second choice data
- Support for optimal micro moments that match micro data scores
- Fixed effect absorption
- Nonlinear functions of product characteristics
- Concentrating out linear parameters
- Flexible random coefficient distributions
- Parameter bounds and constraints
- Random coefficients nested logit (RCNL)
- Approximation to the pure characteristics model
- Varying nesting parameters across groups
- Logit and nested logit benchmarks
- Classic BLP instruments
- Differentiation instruments
- Optimal instruments
- Covariance restrictions
- Timing assumptions using moments based on unobservable innovations
- Adjustments for simulation error
- Tests of overidentifying and model restrictions
- Parametric boostrapping post-estimation outputs
- Elasticities and diversion ratios
- Marginal costs and markups
- Passthrough calculations
- Profits and consumer surplus
- Newton and fixed point methods for computing pricing equilibria
- Merger simulation
- Custom counterfactual simulation
- Synthetic data construction
- SciPy or Artleys Knitro optimization
- Fixed point acceleration
- Monte Carlo, quasi-random sequences, quadrature, and sparse grids
- Importance sampling

- Custom optimization and iteration routines
- Robust and clustered errors
- Linear or log-linear marginal costs
- Partial ownership matrices
- Analytic gradients
- Finite difference Hessians
- Market-by-market parallelization
- Extended floating point precision
- Robust error handling

1.5 Bugs and Requests

Please use the GitHub issue tracker to submit bugs or to request features.

CHAPTER

TWO

NOTATION

The notation in PyBLP is a customized amalgamation of the notation employed by *Berry, Levinsohn, and Pakes (1995)*, *Nevo (2000a)*, *Morrow and Skerlos (2011)*, *Grigolon and Verboven (2014)*, and others.

2.1 Indices

Index	Description
j	Products
t	Markets
i	Agents/individuals
f	Firms
h	Nests
c	Clusters
m	Micro moments

2.2 Dimensions/Sets

Dimension/Set	Description
T	Markets
N	Products across all markets
F	Firms across all markets
I	Agents across all markets
J_t	Products in market t
F_t	Firms in market t
J_{ft}	Products produced by firm f in market t
I_t	Agents in market t
K_1	Demand-side linear product characteristics
K_1^{ex}	Exogenous demand-side linear product characteristics
K_1^{en}	Endogenous demand-side linear product characteristics
K_2	Demand-side nonlinear product characteristics
K_3	Supply-side product characteristics
K_3^{ex}	Exogenous supply-side product characteristics
K_3^{en}	Endogenous supply-side product characteristics
D	Demographic variables
M_D	Demand-side instruments

Continued on next page

Table 1 – continued from previous page

Dimension/Set	Description
M_S	Supply-side instruments
M_C	Covariance instruments
M_M	Micro moments
T_m	Markets over which micro moment m is averaged
T_{mn}	Markets over which micro moments m and n are both averaged
N_m	Observations underlying observed micro moment value m.
M	All moments
E_D	Absorbed dimensions of demand-side fixed effects
E_S	Absorbed dimensions of supply-side fixed effects
H	Nesting groups
J_{ht}	Products in nesting group h and market t
C	Clusters
J_{ct}	Products in cluster c and market t

2.3 Matrices, Vectors, and Scalars

Symbol	Dimensions	Description
X_1	$N \times K_1$	Demand-side linear product characteristics
X_1^{ex}	$N \times K_1^{\text{ex}}$	Exogenous demand-side linear product characteristics
X_1^{en}	$N \times K_1^{\text{en}}$	Endogenous demand-side linear product characteristics
X_2	$N \times K_2$	Demand-side Nonlinear product characteristics
X_3	$N \times K_3$	Supply-side product characteristics
X_3^{ex}	$N \times K_3^{\text{ex}}$	Exogenous supply-side product characteristics
X_3^{en}	$N \times K_3^{\text{en}}$	Endogenous supply-side product characteristics
ξ	$N \times 1$	Demand-side unobservables
ω	$N \times 1$	Supply-side unobservables
$\tilde{\xi}$	$N \times 1$	Demand-side unobservable innovations
$\tilde{\omega}$	$N \times 1$	Supply-side unobservable innovations
p	$N \times 1$	Prices
s (s_{jt})	$N \times 1$	Market shares
s (s_{ht})	$H \times 1$	Group shares in a market t
s (s_{ijt})	$N \times I_t$	Choice probabilities in a market t
c	$N \times 1$	Marginal costs
$\tilde{c}$	$N \times 1$	Linear or log-linear marginal costs, c or $\log c$
η	$N \times 1$	Markup term from the BLP-markup equation
ζ	$N \times 1$	Markup term from the ζ-markup equation
$\mathscr{H}$	$J_t \times J_t$	Ownership or product holdings matrix in market t
κ	$F_t \times F_t$	Cooperation matrix in market t
Δ	$J_t \times J_t$	Intra-firm matrix of (negative, transposed) demand derivatives in market t
Λ	$J_t \times J_t$	Diagonal matrix used to decompose η and ζ in market t
Γ	$J_t \times J_t$	Another matrix used to decompose η and ζ in market t
d	$I_t \times D$	Observed agent characteristics called demographics in market t
ν	$I_t \times K_2$	Unobserved agent characteristics called integration nodes in market t
a	$I_t \times J_t$	Agent-specific product availability in market t
w	$I_t \times 1$	Integration weights in market t
δ	$N \times 1$	Mean utility
μ	$J_t \times I_t$	Agent-specific portion of utility in market t

Continued on next page

Table 2 – continued from previous page

Symbol	Dimensions	Description
ϵ	$N \times 1$	Type I Extreme Value idiosyncratic preferences
$\bar{\epsilon}$ ($\bar{\epsilon}_{ijt}$)	$N \times 1$	Type I Extreme Value term used to decompose ϵ
$\bar{\epsilon}$ ($\bar{\epsilon}_{iht}$)	$N \times 1$	Group-specific term used to decompose ϵ
U	$J_t \times I_t$	Indirect utilities
V (V_{ijt})	$J_t \times I_t$	Indirect utilities minus ϵ
V (V_{iht})	$J_t \times I_t$	Inclusive value of a nesting group
π (π_{jt})	$N \times 1$	Population-normalized gross expected profits
π (π_{ft})	$F_t \times 1$	Population-normalized gross expected profits of a firm in market t
β	$K_1 \times 1$	Demand-side linear parameters
β^{ex}	$K_1^{\text{ex}} \times 1$	Parameters in β on exogenous product characteristics
α	$K_1^{\text{en}} \times 1$	Parameters in β on endogenous product characteristics
Σ	$K_2 \times K_2$	Cholesky root of the covariance matrix for unobserved taste heterogeneity
Π	$K_2 \times D$	Parameters that measures how agent tastes vary with demographics
ρ	$H \times 1$	Parameters that measures within nesting group correlation
ϕ	2×2	Parameters that measure unobservable autocorrelation
γ	$K_3 \times 1$	Supply-side linear parameters
γ^{ex}	$K_3^{\text{ex}} \times 1$	Parameters in γ on exogenous product characteristics
γ^{en}	$K_3^{\text{en}} \times 1$	Parameters in γ on endogenous product characteristics
θ	$P \times 1$	Parameters
Z_D	$N \times M_D$	Demand-side instruments
Z_S	$N \times M_S$	Supply-side instruments
Z_C	$N \times M_C$	Covariance instruments
W	$M \times M$	Weighting matrix
S	$M \times M$	Moment covariances
q	1×1	Objective value
g_D	$N \times M_D$	Demand-side moments
g_S	$N \times M_S$	Supply-side moments
g_C	$N \times M_C$	Covariance moments
g_M	$I \times M_M$	Micro moments
g (g_{jt})	$N \times (M_D + M_S + M_C)$	Demand-side, supply-side, and covariance moments
g (g_c)	$C \times (M_D + M_S + M_C)$	Clustered demand-side, supply-side, and covariance moments
$\bar{g}_D$	$M_D \times 1$	Averaged demand-side moments
$\bar{g}_S$	$M_S \times 1$	Averaged supply-side moments
$\bar{g}_C$	$M_C \times 1$	Averaged covariance moments
$\bar{g}_M$	$M_M \times 1$	Averaged micro moments
$\bar{g}$	$M \times 1$	Averaged moments
G	$M \times P$	Jacobian of the averaged moments with respect to θ
ε	$J_t \times J_t$	Elasticities of demand in market t
$\mathscr{D}$	$J_t \times J_t$	Diversion ratios in market t
$\mathscr{D}$	$J_t \times J_t$	Long-run diversion ratios in market t
$\mathscr{M}$	$N \times 1$	Markups
$\mathscr{E}$	1×1	Aggregate elasticity of demand of a market
CS	1×1	Population-normalized consumer surplus of a market
HHI	1×1	Herfindahl-Hirschman Index of a market

CHAPTER

THREE

BACKGROUND

The following sections provide a very brief overview of the BLP model and how it is estimated. This goal is to concisely introduce the notation and terminology used throughout the rest of the documentation. For a more in-depth overview, refer to *Conlon and Gortmaker (2020)*.

3.1 The Model

There are $t = 1, 2, \ldots, T$ markets, each with $j = 1, 2, \ldots, J_t$ products produced by $f = 1, 2, \ldots, F_t$ firms, for a total of N products across all markets. There are $i = 1, 2, \ldots, I_t$ individuals/agents who choose among the J_t products and an outside good $j = 0$. These numbers also represent sets. For example, $J_t = \{1, 2, \ldots, J_t\}$.

3.1.1 Demand

Observed demand-side product characteristics are contained in the $N \times K_1$ matrix of linear characteristics, X_1, and the $N \times K_2$ matrix of nonlinear characteristics, X_2, which is typically a subset of X_1. Unobserved demand-side product characteristics, ξ, are a $N \times 1$ vector.

In market t, observed agent characteristics are a $I_t \times D$ matrix called demographics, d. Unobserved agent characteristics are a $I_t \times K_2$ matrix, ν.

The indirect utility of agent i from purchasing product j in market t is

$$U_{ijt} = \underbrace{\delta_{jt} + \mu_{ijt}}_{V_{ijt}} + \epsilon_{ijt}, \tag{3.1}$$

in which the mean utility is, in vector-matrix form,

$$\delta = \underbrace{X_1^{\text{en}}\alpha + X_1^{\text{ex}}\beta^{\text{ex}}}_{X_1\beta} + \xi. \tag{3.2}$$

The $K_1 \times 1$ vector of demand-side linear parameters, β, is partitioned into two components: α is a $K_1^{\text{en}} \times 1$ vector of parameters on the $N \times K_1^{\text{en}}$ submatrix of endogenous characteristics, X_1^{en}, and β^{ex} is a $K_1^{\text{ex}} \times 1$ vector of parameters on the $N \times K_1^{\text{ex}}$ submatrix of exogenous characteristics, X_1^{ex}. Usually, $X_1^{\text{en}} = p$, prices, so α is simply a scalar.

The agent-specific portion of utility in a single market is, in vector-matrix form,

$$\mu = X_2(\Sigma\nu' + \Pi d'). \tag{3.3}$$

The model incorporates both observable (demographics) and unobservable taste heterogeneity though random coefficients. For the unobserved heterogeneity, we let ν denote independent draws from the standard normal distribution. These are scaled by a $K_2 \times K_2$ lower-triangular matrix Σ, which denotes the Cholesky root of the covariance matrix for unobserved taste heterogeneity. The $K_2 \times D$ matrix Π measures how agent tastes vary with demographics.

In the above expression, random coefficients are assumed to be normally distributed, but this expression supports all elliptical distributions. To incorporate one or more lognormal random coefficients, the associated columns in the parenthesized expression can be exponentiated before being pre-multiplied by X_2. For example, this allows for the coefficient on price to be lognormal so that demand slopes down for all agents. For lognormal random coefficients, a constant column is typically included in d so that its coefficients in Π parametrize the means of the logs of the random coefficients. More generally, all log-elliptical distributions are supported. A logit link function is also supported.

Random idiosyncratic preferences, ϵ_{ijt}, are assumed to be Type I Extreme Value, so that conditional on the heterogeneous coefficients, market shares follow the well-known logit form. Aggregate market shares are obtained by integrating over the distribution of individual heterogeneity. They are approximated with Monte Carlo integration or quadrature rules defined by the $I_t \times K_2$ matrix of integration nodes, ν, and an $I_t \times 1$ vector of integration weights, w:

$$s_{jt} \approx \sum_{i \in I_t} w_{it} s_{ijt}, \tag{3.4}$$

where the probability that agent i chooses product j in market t is

$$s_{ijt} = \frac{\exp V_{ijt}}{1 + \sum_{k \in J_t} \exp V_{ikt}}. \tag{3.5}$$

There is a one in the denominator because the utility of the outside good is normalized to $U_{i0t} = 0$. The scale of utility is normalized by the variance of ϵ_{ijt}.

3.1.2 Supply

Observed supply-side product characteristics are contained in the $N \times K_3$ matrix of supply-side characteristics, X_3. Prices cannot be supply-side characteristics, but non-price product characteristics often overlap with the demand-side characteristics in X_1 and X_2. Unobserved supply-side product characteristics, ω, are a $N \times 1$ vector.

Firm f chooses prices in market t to maximize the profits of its products $J_{ft} \subset J_t$:

$$\pi_{ft} = \sum_{j \in J_{ft}} (p_{jt} - c_{jt}) s_{jt}. \tag{3.6}$$

In a single market, the corresponding multi-product differentiated Bertrand first order conditions are, in vector-matrix form,

$$p - c = \underbrace{\Delta^{-1} s}_{\eta}, \tag{3.7}$$

where the multi-product Bertrand markup η depends on Δ, a $J_t \times J_t$ matrix of intra-firm (negative, transposed) demand derivatives:

$$\Delta = -\mathscr{H} \odot \frac{\partial s'}{\partial p}. \tag{3.8}$$

Here, $\mathscr{H}$ denotes the market-level ownership or product holdings matrix in the market, where $\mathscr{H}_{jk}$ is typically 1 if the same firm produces products j and k, and 0 otherwise.

To include a supply side, we must specify a functional form for marginal costs:

$$\tilde{c} = f(c) = X_3 \gamma + \omega. \tag{3.9}$$

The most common choices are $f(c) = c$ and $f(c) = \log(c)$.

3.2 Estimation

A demand side is always estimated but including a supply side is optional. With only a demand side, there are three sets of parameters to be estimated: β (which may include α), Σ and Π. With a supply side, there is also γ. The linear parameters, β and γ, are typically concentrated out of the problem. The exception is α, which cannot be concentrated out when there is a supply side because it is needed to compute demand derivatives and hence marginal costs. Linear parameters that are not concentrated out along with unknown nonlinear parameters in Σ and Π are collectively denoted θ.

The GMM problem is

$$\min_{\theta} q(\theta) = \bar{g}(\theta)' W \bar{g}(\theta), \tag{3.10}$$

in which $q(\theta)$ is the GMM objective. By default, PyBLP scales this value by N so that objectives across different problem sizes are comparable. This behavior can be disabled. In some of the BLP literature and in earlier versions of this package, the objective was scaled by N^2.

Here, W is a $M \times M$ weighting matrix and $\bar{g}$ is a $M \times 1$ vector of averaged demand- and supply-side moments:

$$\bar{g} = \begin{bmatrix} \bar{g}_D \\ \bar{g}_S \end{bmatrix} = \frac{1}{N} \begin{bmatrix} \sum_{j,t} Z_{D,jt}' \xi_{jt} \\ \sum_{j,t} Z_{S,jt}' \omega_{jt} \end{bmatrix} \tag{3.11}$$

where Z_D and Z_S are $N \times M_D$ and $N \times M_S$ matrices of demand- and supply-side instruments.

The vector $\bar{g}$ contains sample analogues of the demand- and supply-side moment conditions $E[g_{D,jt}] = E[g_{S,jt}] = 0$ where

$$\begin{bmatrix} g_{D,jt} & g_{S,jt} \end{bmatrix} = \begin{bmatrix} \xi_{jt} Z_{D,jt} & \omega_{jt} Z_{S,jt} \end{bmatrix}. \tag{3.12}$$

In each GMM stage, a nonlinear optimizer finds the $\hat{\theta}$ that minimizes the GMM objective value $q(\theta)$.

3.2.1 The Objective

Given a θ, the first step to computing the objective $q(\theta)$ is to compute $\delta(\theta)$ in each market with the following standard contraction:

$$\delta_{jt} \leftarrow \delta_{jt} + \log s_{jt} - \log s_{jt}(\delta, \theta) \tag{3.13}$$

where s are the market's observed shares and $s(\delta, \theta)$ are calculated market shares. Iteration terminates when the norm of the change in $\delta(\theta)$ is less than a small number.

With a supply side, marginal costs are then computed according to (3.7):

$$c_{jt}(\theta) = p_{jt} - \eta_{jt}(\theta). \tag{3.14}$$

Concentrated out linear parameters are recovered with linear IV-GMM:

$$\begin{bmatrix} \hat{\beta}^{\text{ex}} \\ \hat{\gamma} \end{bmatrix} = (X'ZWZ'X)^{-1} X'ZWZ'Y(\theta) \tag{3.15}$$

where

$$X = \begin{bmatrix} X_1^{\text{ex}} & 0 \\ 0 & X_3 \end{bmatrix}, \quad Z = \begin{bmatrix} Z_D & 0 \\ 0 & Z_S \end{bmatrix}, \quad Y(\theta) = \begin{bmatrix} \delta(\theta) - X_1^{\text{en}} \hat{\alpha} \\ \tilde{c}(\theta) \end{bmatrix}. \tag{3.16}$$

With only a demand side, α can be concentrated out, so $X = X_1$, $Z = Z_D$, and $Y = \delta(\theta)$ recover the full $\hat{\beta}$ in (3.15).

Finally, the product unobservables (i.e., the structural errors),

$$\begin{bmatrix} \xi(\theta) \\ \omega(\theta) \end{bmatrix} = \begin{bmatrix} \delta(\theta) - X_1 \hat{\beta} \\ \tilde{c}(\theta) - X_3 \hat{\gamma} \end{bmatrix}, \tag{3.17}$$

are interacted with the instruments to form $\bar{g}(\theta)$ in (3.11), which gives the GMM objective $q(\theta)$ in (3.10).

3.2.2 The Gradient

The gradient of the GMM objective in (3.10) is

$$\nabla q(\theta) = 2\bar{G}(\theta)' W \bar{g}(\theta) \tag{3.18}$$

where

$$\bar{G} = \begin{bmatrix} \bar{G}_D \\ \bar{G}_S \end{bmatrix} = \frac{1}{N} \begin{bmatrix} \sum_{j,t} Z'_{D,jt} \frac{\partial \xi_{jt}}{\partial \theta} \\ \sum_{j,t} Z'_{S,jt} \frac{\partial \omega_{jt}}{\partial \theta} \end{bmatrix}. \tag{3.19}$$

Writing δ as an implicit function of s in (3.4) gives the demand-side Jacobian:

$$\frac{\partial \xi}{\partial \theta} = \frac{\partial \delta}{\partial \theta} = -\left(\frac{\partial s}{\partial \delta}\right)^{-1} \frac{\partial s}{\partial \theta}. \tag{3.20}$$

The supply-side Jacobian is derived from the definition of $\tilde{c}$ in (3.9):

$$\frac{\partial \omega}{\partial \theta} = \frac{\partial \tilde{c}}{\partial \theta} = -\frac{\partial \tilde{c}}{\partial c} \frac{\partial \eta}{\partial \theta}. \tag{3.21}$$

The second term in this expression is derived from the definition of η in (3.7):

$$\frac{\partial \eta}{\partial \theta} = -\Delta^{-1} \left(\frac{\partial \Delta}{\partial \theta} \eta + \frac{\partial \Delta}{\partial \xi} \eta \frac{\partial \xi}{\partial \theta} \right). \tag{3.22}$$

One thing to note is that $\frac{\partial \xi}{\partial \theta} = \frac{\partial \delta}{\partial \theta}$ and $\frac{\partial \omega}{\partial \theta} = \frac{\partial \tilde{c}}{\partial \theta}$ need not hold during optimization if we concentrate out linear parameters because these are then functions of θ. Fortunately, one can use orthogonality conditions to show that it is fine to treat these parameters as fixed when computing the gradient.

3.2.3 Weighting Matrices

Conventionally, the 2SLS weighting matrix is used in the first stage:

$$W = \begin{bmatrix} (Z'_D Z_D / N)^{-1} & 0 \\ 0 & (Z'_S Z_S / N)^{-1} \end{bmatrix}. \tag{3.23}$$

With two-step GMM, W is updated before the second stage according to

$$W = S^{-1}. \tag{3.24}$$

For heteroscedasticity robust weighting matrices,

$$S = \frac{1}{N} \sum_{j,t} g_{jt} g'_{jt}. \tag{3.25}$$

For clustered weighting matrices with $c = 1, 2, \ldots, C$ clusters,

$$S = \frac{1}{N} \sum_{c=1}^{C} g_c g'_c, \tag{3.26}$$

where, letting the set $J_{ct} \subset J_t$ denote products in cluster c and market t,

$$g_c = \sum_{t \in T} \sum_{j \in J_{ct}} g_{jt}. \tag{3.27}$$

For unadjusted weighting matrices,

$$S = \frac{1}{N}\begin{bmatrix} \sigma_\xi^2 Z_D' Z_D & \sigma_{\xi\omega} Z_D' Z_S \\ \sigma_{\xi\omega} Z_S' Z_D & \sigma_\omega^2 Z_S' Z_S \end{bmatrix} \tag{3.28}$$

where σ_ξ^2, σ_ω^2, and $\sigma_{\xi\omega}$ are estimates of the variances and covariance between the structural errors.

Simulation error can be accounted for by resampling agents $r = 1, \ldots, R$ times, evaluating each $\bar{g}_r$, and adding the following to S:

$$\frac{1}{R-1}\sum_{r=1}^{R}(\bar{g}_r - \bar{\bar{g}})(\bar{g}_r - \bar{\bar{g}})', \quad \bar{\bar{g}} = \frac{1}{R}\sum_{r=1}^{R}\bar{g}_r. \tag{3.29}$$

3.2.4 Standard Errors

An estimate of the asymptotic covariance matrix of $\sqrt{N}(\hat{\theta} - \theta_0)$ is

$$(\bar{G}' W \bar{G})^{-1} \bar{G}' W S W \bar{G} (\bar{G}' W \bar{G})^{-1}. \tag{3.30}$$

Standard errors are the square root of the diagonal of this matrix divided by N.

If the weighting matrix was chosen such that $W = S^{-1}$, this simplifies to

$$(\bar{G}' W \bar{G})^{-1}. \tag{3.31}$$

Standard errors extracted from this simpler expression are called unadjusted.

3.3 Fixed Effects

Small numbers of fixed effects can be estimated with dummy variables in X_1, X_3, Z_D, and Z_S. However, this approach does not scale with high dimensional fixed effects because it requires constructing and inverting an infeasibly large matrix in (3.15).

Instead, fixed effects are typically absorbed into X, Z, and $Y(\theta)$ in (3.15). With $E_D = 1$ fixed effect on the demand side and/or $E_S = 1$ fixed effect on the supply side, these matrices are simply de-meaned within each level of the fixed effect. Both X and Z can be de-meaned just once, but $Y(\theta)$ must be de-meaned for each new θ.

This procedure is equivalent to replacing each column of the matrices with residuals from a regression of the column on the fixed effect. The Frish-Waugh-Lovell (FWL) theorem of *Frisch and Waugh (1933)* and *Lovell (1963)* guarantees that using these residualized matrices gives the same results as including fixed effects as dummy variables. When $E_D > 1$ or $E_S > 1$, the matrices are residualized with more involved algorithms. Once fixed effects have been absorbed, estimation is as described above.

3.4 Micro Moments

More detailed micro data on individual choices can be used to supplement the standard demand- and supply-side moments $\bar{g}_D$ and $\bar{g}_S$ in (3.11) with an additional $m = 1, 2, \ldots, M_M$ micro moments, $\bar{g}_M$, for a total of $M = M_D + M_S + M_M$ moments:

$$\bar{g} = \begin{bmatrix} \bar{g}_D \\ \bar{g}_S \\ \bar{g}_M \end{bmatrix}. \tag{3.32}$$

Conlon and Gortmaker (2025) provides a standardized framework for incorporating micro moments into BLP-style estimation. What follows is a simplified summary of this framework. Each micro moment m is the difference between an observed value $f_m(\bar{v})$ and its simulated analogue $f_m(v)$:

$$\bar{g}_{M,m} = f_m(\bar{v}) - f_m(v), \tag{3.33}$$

in which $f_m(\cdot)$ is a function that maps a vector of $p = 1, \ldots, P_M$ micro moment parts $\bar{v} = (\bar{v}_1, \ldots, \bar{v}_{P_M})'$ or $v = (v_1, \ldots, v_{P_M})'$ into a micro statistic. Each sample micro moment part p is an average over observations $n \in N_{d_m}$ in the associated micro dataset d_p:

$$\bar{v}_p = \frac{1}{N_{d_p}} \sum_{n \in N_{d_p}} v_{pi_n j_n t_n}. \tag{3.34}$$

Its simulated analogue is

$$v_p = \frac{\sum_{t \in T} \sum_{i \in I_t} \sum_{j \in J_t \cup \{0\}} w_{it} s_{ijt} w_{d_p ijt} v_{pijt}}{\sum_{t \in T} \sum_{i \in I_t} \sum_{j \in J_t \cup \{0\}} w_{it} s_{ijt} w_{d_p ijt}}, \tag{3.35}$$

In which $w_{it} s_{ijt} w_{d_p ijt}$ is the probability an observation in the micro dataset is for an agent i who chooses j in market t.

The simplest type of micro moment is just an average over the entire sample, with $f_m(v) = v_1$. For example, with v_{1ijt} equal to the income for an agent i who chooses j in market t, micro moment m would match the average income in dataset d_p. Observed values such as conditional expectations, covariances, correlations, or regression coefficients can be matched by choosing the appropriate function f_m. For example, with v_{2ijt} equal to the interaction between income and an indicator for the choice of the outside option, and with v_{3ijt} equal to an indicator for the choiced of the outside option, $f_m(v) = v_2/v_3$ would match an observed conditional mean income within those who choose the outside option.

A micro dataset d, often a survey, is defined by survey weights w_{dijt}. For example, $w_{dijt} = 1\{j \neq 0, t \in T_d\}$ defines a micro dataset that is a selected sample of inside purchasers in a few markets $T_d \subset T$, giving each market an equal sampling weight. Different micro datasets are independent.

A micro dataset will often admit multiple micro moment parts. Each micro moment part p is defined by its dataset d_p and micro values v_{pijt}. For example, a micro moment part p with $v_{pijt} = y_{it} x_{jt}$ delivers the mean $\bar{v}_p$ or expectation v_p of an interaction between some demographic y_{it} and some product characteristic x_{jt}.

A micro moment is a function of one or more micro moment parts. The simplest type is a function of only one micro moment part, and matches the simple average defined by the micro moment part. For example, $f_m(v) = v_p$ with $v_{pijt} = y_{it} x_{jt}$ matches the mean of an interaction between y_{it} and x_{jt}. Non-simple averages such as conditional means, covariances, correlations, or regression coefficients can be matched by choosing an appropriate function f_m. For example, $f_m(v) = v_1/v_2$ with $v_{1ijt} = y_{it} x_{jt} 1\{j \neq 0\}$ and $v_{2ijt} = 1\{j \neq 0\}$ matches the conditional mean of an interaction between y_{it} and x_{jt} among those who do not choose the outside option $j = 0$.

Technically, if not all micro moments m are simple averages $f_m(v) = v_m$, then the resulting estimator will no longer be a GMM estimator, but rather a more generic minimum distance estimator, since these "micro moments" are not technically sample moments. Regardless, the package uses GMM terminology for simplicity's sake, and the statistical expressions are all the same. Micro moments are computed for each θ and contribute to the GMM (or minimum distance) objective $q(\theta)$ in (3.10). Their derivatives with respect to θ are added as rows to $\bar{G}$ in (3.19), and blocks are added to both W and S in (3.23) and (3.24). The covariance between standard moments and micro moments is zero, so these matrices are block-diagonal. The delta method delivers the covariance matrix for the micro moments:

$$S_M = \frac{\partial f(v)}{\partial v'} S_P \frac{\partial f(v)'}{\partial v}. \tag{3.36}$$

The scaled covariance between micro moment parts p and q in S_P is zero if they are based on different micro datasets d_p neq d_q'; otherwise, if based on the same dataset $d_p = d_q = d$,

$$S_{P,pq} = \frac{N}{N_d} \text{Cov}(v_{pi_n j_n t_n}, v_{qi_n j_n t_n}), \tag{3.37}$$

in which

$$\text{Cov}(v_{pi_n j_n t_n}, v_{qi_n j_n t_n}) = \frac{\sum_{t \in T} \sum_{i \in I_t} \sum_{j \in J_t \cup \{0\}} w_{it} s_{ijt} w_{dijt} (v_{pijt} - v_p)(v_{qijt} - v_q)}{\sum_{t \in T} \sum_{i \in I_t} \sum_{j \in J_t \cup \{0\}} w_{it} s_{ijt} w_{dijt}}. \tag{3.38}$$

Micro moment parts based on second choice are averages over values v_{pijkt} where k indexes second choices, and are based on datasets defined by survey weights w_{dijkt}. A sample micro moment part is

$$\bar{v}_p = \frac{1}{N_{d_p}} \sum_{n \in N_{d_p}} v_{pi_n j_n k_n t_n}. \tag{3.39}$$

Its simulated analogue is

$$v_p = \frac{\sum_{t \in T} \sum_{i \in I_t} \sum_{j,k \in J_t \cup \{0\}} w_{it} s_{ijt} s_{ik(-j)t} w_{d_p ijkt} v_{pijkt}}{\sum_{t \in T} \sum_{i \in I_t} \sum_{j,k \in J_t \cup \{0\}} w_{it} s_{ijt} s_{ik(-j)t} w_{d_p ijkt}}, \tag{3.40}$$

in which $s_{ik(-j)t}$ is the probability of choosing k when j is removed from the choice set. One can also define micro moment parts based on second choices where a group of products $h(j)$ containing the first choice j is removed from the choice set. In this case, the above second choice probabilities become $s_{ik(-h(j))t}$.

Covariances are defined analogously.

3.5 Random Coefficients Nested Logit

Incorporating parameters that measure within nesting group correlation gives the random coefficients nested logit (RCNL) model of *Brenkers and Verboven (2006)* and *Grigolon and Verboven (2014)*. There are $h = 1, 2, \ldots, H$ nesting groups and each product j is assigned to a group $h(j)$. The set $J_{ht} \subset J_t$ denotes the products in group h and market t.

In the RCNL model, idiosyncratic preferences are partitioned into

$$\epsilon_{ijt} = \bar{\epsilon}_{ih(j)t} + (1 - \rho_{h(j)}) \bar{\epsilon}_{ijt} \tag{3.41}$$

where $\bar{\epsilon}_{ijt}$ is Type I Extreme Value and $\bar{\epsilon}_{ih(j)t}$ is distributed such that ϵ_{ijt} is still Type I Extreme Value.

The nesting parameters, ρ, can either be a $H \times 1$ vector or a scalar so that for all groups $\rho_h = \rho$. Letting $\rho \to 0$ gives the standard BLP model and $\rho \to 1$ gives division by zero errors. With $\rho_h \in (0, 1)$, the expression for choice probabilities in (3.5) becomes more complicated:

$$s_{ijt} = \frac{\exp[V_{ijt}/(1 - \rho_{h(j)})]}{\exp[V_{ih(j)t}/(1 - \rho_{h(j)})]} \cdot \frac{\exp V_{ih(j)t}}{1 + \sum_{h \in H} \exp V_{iht}} \tag{3.42}$$

where

$$V_{iht} = (1 - \rho_h) \log \sum_{k \in J_{ht}} \exp[V_{ikt}/(1 - \rho_h)]. \tag{3.43}$$

The contraction for $\delta(\theta)$ in (3.13) is also slightly different:

$$\delta_{jt} \leftarrow \delta_{jt} + (1 - \rho_{h(j)})[\log s_{jt} - \log s_{jt}(\delta, \theta)]. \tag{3.44}$$

Otherwise, estimation is as described above with ρ included in θ.

3.6 Logit and Nested Logit

Letting $\Sigma = 0$ gives the simpler logit (or nested logit) model where there is a closed-form solution for δ. In the logit model,

$$\delta_{jt} = \log s_{jt} - \log s_{0t}, \tag{3.45}$$

and a lack of nonlinear parameters means that nonlinear optimization is often unneeded.

In the nested logit model, ρ must be optimized over, but there is still a closed-form solution for δ:

$$\delta_{jt} = \log s_{jt} - \log s_{0t} - \rho_{h(j)}[\log s_{jt} - \log s_{h(j)t}]. \tag{3.46}$$

where

$$s_{ht} = \sum_{j \in J_{ht}} s_{jt}. \tag{3.47}$$

In both models, a supply side can still be estimated jointly with demand. Estimation is as described above with a representative agent in each market: $I_t = 1$ and $w_1 = 1$.

3.7 Equilibrium Prices

Counterfactual evaluation, synthetic data simulation, and optimal instrument generation often involve solving for prices implied by the Bertrand first order conditions in (3.7). Solving this system with Newton's method is slow and iterating over $p \leftarrow c + \eta(p)$ may not converge because it is not a contraction.

Instead, *Morrow and Skerlos (2011)* reformulate the solution to (3.7):

$$p - c = \underbrace{\Lambda^{-1}(\mathscr{H} \odot \Gamma)'(p - c) - \Lambda^{-1}s}_{\zeta} \tag{3.48}$$

where Λ is a diagonal $J_t \times J_t$ matrix approximated by

$$\Lambda_{jj} \approx \sum_{i \in I_t} w_{it} s_{ijt} \frac{\partial U_{ijt}}{\partial p_{jt}} \tag{3.49}$$

and Γ is a dense $J_t \times J_t$ matrix approximated by

$$\Gamma_{jk} \approx \sum_{i \in I_t} w_{it} s_{ijt} s_{ikt} \frac{\partial U_{ikt}}{\partial p_{kt}}. \tag{3.50}$$

Equilibrium prices are computed by iterating over the ζ-markup equation in (3.48),

$$p \leftarrow c + \zeta(p), \tag{3.51}$$

which, unlike (3.7), is a contraction. Iteration terminates when the norm of firms' first order conditions, $||\Lambda(p)(p - c - \zeta(p))||$, is less than a small number.

If marginal costs depend on quantity, then they also depend on prices and need to be updated during each iteration: $c_{jt} = c_{jt}(s_{jt}(p))$.

CHAPTER

FOUR

TUTORIAL

This section uses a series of Jupyter Notebooks to explain how PyBLP can be used to solve example problems, compute post-estimation outputs, and simulate problems.

For a more concise, targeted, and opinionated tutorial, see the Demand Estimation Mixtape Session, a three day workshop taught by Jeff Gortmaker in 2024. In order, the days cover pure logit estimation, aggregate BLP estimation, and micro BLP estimation. Each day has slides and extensive coding exercises (along with Jupyter Notebook solutions) that use PyBLP.

In addition to the notebooks here and the above workshop, other examples can be found in the *API Documentation*.

The online version of the following section may be easier to read.

4.1 Logit and Nested Logit Tutorial

```
import pyblp
import numpy as np
import pandas as pd

pyblp.options.digits = 2
pyblp.options.verbose = False
pyblp.__version__
```

```
'1.2.0'
```

In this tutorial, we'll use data from *Nevo (2000a)* to solve the paper's fake cereal problem. Locations of CSV files that contain the data are in the *data* module.

We will compare two simple models, the plain (IIA) logit model and the nested logit (GEV) model using the fake cereal dataset of *Nevo (2000a)*.

4.1.1 Theory of Plain Logit

Let's start with the plain logit model under independence of irrelevant alternatives (IIA). In this model (indirect) utility is given by

$$U_{ijt} = \alpha p_{jt} + x_{jt}\beta^{\text{ex}} + \xi_{jt} + \epsilon_{ijt}, \tag{4.1}$$

where ϵ_{ijt} is distributed IID with the Type I Extreme Value (Gumbel) distribution. It is common to normalize the mean utility of the outside good to zero so that $U_{i0t} = \epsilon_{i0t}$. This gives us aggregate market shares

$$s_{jt} = \frac{\exp(\alpha p_{jt} + x_{jt}\beta^{\text{ex}} + \xi_{jt})}{1 + \sum_k \exp(\alpha p_{kt} + x_{kt}\beta^{\text{ex}} + \xi_{kt})}. \tag{4.2}$$

If we take logs we get

$$\log s_{jt} = \alpha p_{jt} + x_{jt}\beta^{\text{ex}} + \xi_{jt} - \log \sum_k \exp(\alpha p_{kt} + x_{kt}\beta^{\text{ex}} + \xi_{kt}) \tag{4.3}$$

and

$$\log s_{0t} = -\log \sum_k \exp(\alpha p_{kt} + x_{kt}\beta^{\text{ex}} + \xi_{kt}). \tag{4.4}$$

By differencing the above we get a linear estimating equation:

$$\log s_{jt} - \log s_{0t} = \alpha p_{jt} + x_{jt}\beta^{\text{ex}} + \xi_{jt}. \tag{4.5}$$

Because the left hand side is data, we can estimate this model using linear IV GMM.

4.1.2 Application of Plain Logit

A Logit *Problem* can be created by simply excluding the formulation for the nonlinear parameters, X_2, along with any agent information. In other words, it requires only specifying the *linear component* of demand.

We'll set up and solve a simple version of the fake data cereal problem from *Nevo (2000a)*. Since we won't include any demand-side nonlinear characteristics or parameters, we don't have to worry about configuring an optimization routine.

There are some important reserved variable names:

- `market_ids` are the unique market identifiers which we subscript with t.
- `shares` specifies the market shares which need to be between zero and one, and within a market ID, $\sum_j s_{jt} \leq 1$.
- `prices` are prices p_{jt}. These have some special properties and are *always* treated as endogenous.
- `demand_instruments0`, `demand_instruments1`, and so on are numbered demand instruments. These represent only the *excluded* instruments. The exogenous regressors in X_1 will be automatically added to the set of instruments.

We begin with two steps:

1. Load the product data which at a minimum consists of `market_ids`, `shares`, `prices`, and at least a single column of demand instruments, `demand_instruments0`.
2. Define a *Formulation* for the X_1 (linear) demand model.
   - This and all other formulas are similar to R and patsy formulas.
   - It includes a constant by default. To exclude the constant, specify either a `0` or a `-1`.
   - To efficiently include fixed effects, use the `absorb` option and specify which categorical variables you would like to absorb.
   - Some model reduction may happen automatically. The constant will be excluded if you include fixed effects and some precautions are taken against collinearity. However, you will have to make sure that differently-named variables are not collinear.
3. Combine the *Formulation* and product data to construct a *Problem*.
4. Use *Problem.solve* to estimate paramters.

Loading the Data

The `product_data` argument of *Problem* should be a structured array-like object with fields that store data. Product data can be a structured NumPy array, a pandas DataFrame, or other similar objects.

```
product_data = pd.read_csv(pyblp.data.NEVO_PRODUCTS_LOCATION)
product_data.head()
```

```
  market_ids  city_ids  quarter product_ids  firm_ids  brand_ids    shares  \
0      C01Q1         1        1       F1B04         1          4  0.012417
1      C01Q1         1        1       F1B06         1          6  0.007809
2      C01Q1         1        1       F1B07         1          7  0.012995
3      C01Q1         1        1       F1B09         1          9  0.005770
4      C01Q1         1        1       F1B11         1         11  0.017934

     prices  sugar  mushy  ...  demand_instruments10  demand_instruments11  \
0  0.072088      2      1  ...              2.116358             -0.154708
1  0.114178     18      1  ...             -7.374091             -0.576412
2  0.132391      4      1  ...              2.187872             -0.207346
3  0.130344      3      0  ...              2.704576              0.040748
4  0.154823     12      0  ...              1.261242              0.034836

   demand_instruments12  demand_instruments13  demand_instruments14  \
0             -0.005796              0.014538              0.126244
1              0.012991              0.076143              0.029736
2              0.003509              0.091781              0.163773
3             -0.003724              0.094732              0.135274
4             -0.000568              0.102451              0.130640

   demand_instruments15  demand_instruments16  demand_instruments17  \
0              0.067345              0.068423              0.034800
1              0.087867              0.110501              0.087784
2              0.111881              0.108226              0.086439
3              0.088090              0.101767              0.101777
4              0.084818              0.101075              0.125169

   demand_instruments18  demand_instruments19
0              0.126346              0.035484
1              0.049872              0.072579
2              0.122347              0.101842
3              0.110741              0.104332
4              0.133464              0.121111
```

(continues on next page)

(continued from previous page)

```
[5 rows x 30 columns]
```

The product data contains `market_ids`, `product_ids`, `firm_ids`, `shares`, `prices`, a number of other IDs and product characteristics, and some pre-computed excluded `demand_instruments0`, `demand_instruments1`, and so on. The `product_ids` will be incorporated as fixed effects.

For more information about the instruments and the example data as a whole, refer to the *data* module.

Setting Up the Problem

We can combine the *Formulation* and `product_data` to construct a *Problem*. We pass the *Formulation* first and the `product_data` second. We can also display the properties of the problem by typing its name.

```
logit_formulation = pyblp.Formulation('prices', absorb='C(product_ids)')
logit_formulation
```

```
prices + Absorb[C(product_ids)]
```

```
problem = pyblp.Problem(logit_formulation, product_data)
problem
```

```
Dimensions:
================================
 T    N     F    K1    MD    ED
---  ----  ---  ----  ----  ----
94   2256   5    1     20    1
================================

Formulations:
=================================
     Column Indices:          0
-------------------------  ------
X1: Linear Characteristics  prices
=================================
```

Two sets of properties are displayed:

1. Dimensions of the data.
2. Formulations of the problem.

The dimensions describe the shapes of matrices as laid out in *Notation*. They include:

- T is the number of markets.
- N is the length of the dataset (the number of products across all markets).
- F is the number of firms, which we won't use in this example.
- K_1 is the dimension of the linear demand parameters.
- M_D is the dimension of the instrument variables (excluded instruments and exogenous regressors).
- E_D is the number of fixed effect dimensions (one-dimensional fixed effects, two-dimensional fixed effects, etc.).

There is only a single *Formulation* for this model.

- X_1 is the linear component of utility for demand and depends only on prices (after the fixed effects are removed).

Solving the Problem

The *Problem.solve* method always returns a *ProblemResults* class, which can be used to compute post-estimation outputs. See the *post estimation* tutorial for more information.

```
logit_results = problem.solve()
logit_results
```

```
Problem Results Summary:
==========================================
GMM   Objective  Clipped  Weighting Matrix
Step    Value    Shares   Condition Number
----  ---------  -------  ----------------
 2    +1.9E+02      0         +5.7E+07
==========================================

Cumulative Statistics:
========================
Computation   Objective
   Time      Evaluations
-----------  -----------
 00:00:00         2
========================

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
```

(continues on next page)

(continued from previous page)

```
----------
 -3.0E+01
(+1.0E+00)
==========
```

4.1.3 Theory of Nested Logit

We can extend the logit model to allow for correlation within a group h so that

$$U_{ijt} = \alpha p_{jt} + x_{jt}\beta^{\text{ex}} + \xi_{jt} + \bar{\epsilon}_{h(j)ti} + (1-\rho)\bar{\epsilon}_{ijt}. \tag{4.6}$$

Now, we require that $\epsilon_{jti} = \bar{\epsilon}_{h(j)ti} + (1-\rho)\bar{\epsilon}_{jti}$ is distributed IID with the Type I Extreme Value (Gumbel) distribution. As $\rho \to 1$, all consumers stay within their group. As $\rho \to 0$, this collapses to the IIA logit. Note that if we wanted, we could allow ρ to differ between groups with the notation $\rho_{h(j)}$.

This gives us aggregate market shares as the product of two logits, the within group logit and the across group logit:

$$s_{jt} = \frac{\exp[V_{jt}/(1-\rho)]}{\exp[V_{h(j)t}/(1-\rho)]} \cdot \frac{\exp V_{h(j)t}}{1 + \sum_h \exp V_{ht}}, \tag{4.7}$$

where $V_{jt} = \alpha p_{jt} + x_{jt}\beta^{\text{ex}} + \xi_{jt}$.

After some work we again obtain the linear estimating equation:

$$\log s_{jt} - \log s_{0t} = \alpha p_{jt} + x_{jt}\beta^{\text{ex}} + \rho \log s_{j|h(j)t} + \xi_{jt}, \tag{4.8}$$

where $s_{j|h(j)t} = s_{jt}/s_{h(j)t}$ and $s_{h(j)t}$ is the share of group h in market t. See *Berry (1994)* or *Cardell (1997)* for more information.

Again, the left hand side is data, though the $\ln s_{j|h(j)t}$ is clearly endogenous which means we must instrument for it. Rather than include $\ln s_{j|h(j)t}$ along with the linear components of utility, X_1, whenever `nesting_ids` are included in `product_data`, ρ is treated as a nonlinear X_2 parameter. This means that the linear component is given instead by

$$\log s_{jt} - \log s_{0t} - \rho \log s_{j|h(j)t} = \alpha p_{jt} + x_{jt}\beta^{\text{ex}} + \xi_{jt}. \tag{4.9}$$

This is done for two reasons:

1. It forces the user to treat ρ as an endogenous parameter.
2. It extends much more easily to the RCNL model of *Brenkers and Verboven (2006)*.

A common choice for an additional instrument is the number of products per nest.

4.1.4 Application of Nested Logit

By including `nesting_ids` (another reserved name) as a field in `product_data`, we tell the package to estimate a nested logit model, and we don't need to change any of the formulas. We show how to construct the category groupings in two different ways:

1. We put all products in a single nest (only the outside good in the other nest).
2. We put products into two nests (either mushy or non-mushy).

We also construct an additional instrument based on the number of products per nest. Typically this is useful as a source of exogenous variation in the within group share $\ln s_{j|h(j)t}$. However, in this example because the number of products per nest does not vary across markets, if we include product fixed effects, this instrument is irrelevant.

We'll define a function that constructs the additional instrument and solves the nested logit problem. We'll exclude product ID fixed effects, which are collinear with `mushy`, and we'll choose $\rho = 0.7$ as the initial value at which the optimization routine will start.

```
def solve_nl(df):
    groups = df.groupby(['market_ids', 'nesting_ids'])
    df['demand_instruments20'] = groups['shares'].transform(np.size)
    nl_formulation = pyblp.Formulation('0 + prices')
    problem = pyblp.Problem(nl_formulation, df)
    return problem.solve(rho=0.7)
```

First, we'll solve the problem when there's a single nest for all products, with the outside good in its own nest.

```
df1 = product_data.copy()
df1['nesting_ids'] = 1
nl_results1 = solve_nl(df1)
nl_results1
```

```
Problem Results Summary:
=========================================================================================
GMM   Objective    Projected    Reduced   Clipped  Weighting Matrix  Covariance Matrix
Step    Value    Gradient Norm  Hessian   Shares   Condition Number  Condition Number
----  ---------  -------------  --------  -------  ----------------  -----------------
 2    +2.0E+02     +1.8E-09     +1.1E+04     0         +2.0E+09          +3.0E+04
=========================================================================================

Cumulative Statistics:
================================================
Computation  Optimizer  Optimization   Objective
   Time      Converged   Iterations   Evaluations
-----------  ---------  ------------  -----------
```

(continues on next page)

(continued from previous page)

```
 00:00:01       Yes          3             8
================================================

Rho Estimates (Robust SEs in Parentheses):
==========
All Groups
----------
 +9.8E-01
(+1.4E-02)
==========

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
----------
 -1.2E+00
(+4.0E-01)
==========
```

When we inspect the *Problem*, the only changes from the plain logit model is the additional instrument that contributes to M_D and the inclusion of H, the number of nesting categories.

```
nl_results1.problem
```

```
Dimensions:
================================
 T    N     F    K1    MD    H
---  ----  ---  ----  ----  ---
94   2256   5    1     21    1
================================

Formulations:
=================================
     Column Indices:          0
-------------------------  ------
X1: Linear Characteristics  prices
=================================
```

Next, we'll solve the problem when there are two nests for mushy and non-mushy.

```
df2 = product_data.copy()
df2['nesting_ids'] = df2['mushy']
nl_results2 = solve_nl(df2)
nl_results2
```

```
Problem Results Summary:
=========================================================================================
GMM   Objective    Projected    Reduced   Clipped  Weighting Matrix  Covariance Matrix
Step    Value    Gradient Norm  Hessian   Shares   Condition Number  Condition Number
----  ---------  -------------  --------  -------  ----------------  ----------------
 2    +6.9E+02     +6.7E-09     +5.6E+03     0         +5.1E+08          +2.0E+04
=========================================================================================

Cumulative Statistics:
================================================
Computation  Optimizer  Optimization   Objective
   Time      Converged   Iterations   Evaluations
-----------  ---------  ------------  -----------
 00:00:01       Yes          3             8
================================================

Rho Estimates (Robust SEs in Parentheses):
==========
All Groups
----------
 +8.9E-01
(+1.9E-02)
==========

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
----------
 -7.8E+00
(+4.8E-01)
==========
```

For both cases we find that $\hat{\rho} > 0.8$.

Finally, we'll also look at the adjusted parameter on prices, $\alpha/(1 - \rho)$.

```
nl_results1.beta[0] / (1 - nl_results1.rho)
```

```
array([[-67.39338888]])
```

```
nl_results2.beta[0] / (1 - nl_results2.rho)
```

```
array([[-72.27074638]])
```

Treating Within Group Shares as Exogenous

The package is designed to prevent the user from treating the within group share, $\log s_{j|h(j)t}$, as an exogenous variable. For example, if we were to compute a `group_share` variable and use the algebraic functionality of *Formulation* by including the expression `log(shares / group_share)` in our formula for X_1, the package would raise an error because the package knows that `shares` should not be included in this formulation.

To demonstrate why this is a bad idea, we'll override this feature by calculating $\log s_{j|h(j)t}$ and including it as an additional variable in X_1. To do so, we'll first re-define our function for setting up and solving the nested logit problem.

```
def solve_nl2(df):
    groups = df.groupby(['market_ids', 'nesting_ids'])
    df['group_share'] = groups['shares'].transform('sum')
    df['within_share'] = df['shares'] / df['group_share']
    df['demand_instruments20'] = groups['shares'].transform(np.size)
    nl2_formulation = pyblp.Formulation('0 + prices + log(within_share)')
    problem = pyblp.Problem(nl2_formulation, df.drop(columns=['nesting_ids']))
    return problem.solve()
```

Again, we'll solve the problem when there's a single nest for all products, with the outside good in its own nest.

```
nl2_results1 = solve_nl2(df1)
nl2_results1
```

```
Problem Results Summary:
=============================================================
GMM   Objective  Clipped  Weighting Matrix  Covariance Matrix
Step    Value    Shares   Condition Number  Condition Number
----  ---------  -------  ----------------  ----------------
 2    +2.0E+02      0         +2.1E+09          +1.1E+04
=============================================================

Cumulative Statistics:
```

(continues on next page)

(continued from previous page)

```
========================
Computation   Objective
   Time      Evaluations
-----------  -----------
 00:00:00         2
========================

Beta Estimates (Robust SEs in Parentheses):
==============================
  prices    log(within_share)
----------  -----------------
 -1.0E+00       +9.9E-01
(+2.4E-01)     (+7.9E-03)
==============================
```

And again, we'll solve the problem when there are two nests for mushy and non-mushy.

```
nl2_results2 = solve_nl2(df2)
nl2_results2
```

```
Problem Results Summary:
============================================================
GMM   Objective  Clipped  Weighting Matrix  Covariance Matrix
Step    Value    Shares   Condition Number  Condition Number
----  ---------  -------  ----------------  ----------------
 2    +7.0E+02      0          +5.5E+08          +7.7E+03
============================================================

Cumulative Statistics:
========================
Computation   Objective
   Time      Evaluations
-----------  -----------
 00:00:00         2
========================

Beta Estimates (Robust SEs in Parentheses):
==============================
  prices    log(within_share)
----------  -----------------
 -6.8E+00       +9.3E-01
```

(continues on next page)

(continued from previous page)

```
(+2.9E-01)      (+1.1E-02)
==============================
```

One can observe that we obtain parameter estimates which are quite different than above.

```
nl2_results1.beta[0] / (1 - nl2_results1.beta[1])
```

```
array([-86.37368444])
```

```
nl2_results2.beta[0] / (1 - nl2_results2.beta[1])
```

```
array([-100.14496891])
```

The online version of the following section may be easier to read.

4.2 Random Coefficients Logit Tutorial with the Fake Cereal Data

```
import pyblp
import numpy as np
import pandas as pd

pyblp.options.digits = 2
pyblp.options.verbose = False
pyblp.__version__
```

```
'1.2.0'
```

In this tutorial, we'll use data from *Nevo (2000a)* to solve the paper's fake cereal problem. Locations of CSV files that contain the data are in the *data* module.

4.2.1 Theory of Random Coefficients Logit

The random coefficients model extends the plain logit model by allowing for correlated tastes for different product characteristics. In this model (indirect) utility is given by

$$u_{ijt} = \alpha_i p_{jt} + x_{jt}\beta_i^{\text{ex}} + \xi_{jt} + \epsilon_{ijt} \tag{4.10}$$

The main addition is that $\beta_i = (\alpha_i, \beta_i^{\text{ex}})$ have individual specific subscripts i.

Conditional on β_i, the individual market share follow the same logit form as before. But now we must integrate over heterogeneous individuals to get the aggregate market share:

$$s_{jt}(\alpha, \beta, \theta) = \int \frac{\exp(\alpha_i p_{jt} + x_{jt}\beta_i^{\text{ex}} + \xi_{jt})}{1 + \sum_k \exp(\alpha_i p_{jt} + x_{kt}\beta_i^{\text{ex}} + \xi_{kt})} f(\alpha_i, \beta_i \mid \theta). \tag{4.11}$$

In general, this integral needs to be calculated numerically. This also means that we can't directly linearize the model. It is common to re-parametrize the model to separate the aspects of mean utility that all individuals agree on, $\delta_{jt} = \alpha p_{jt} + x_{jt}\beta^{\text{ex}} + \xi_{jt}$, from the individual specific heterogeneity, $\mu_{ijt}(\theta)$. This gives us

$$s_{jt}(\delta_{jt}, \theta) = \int \frac{\exp(\delta_{jt} + \mu_{ijt})}{1 + \sum_k \exp(\delta_{kt} + \mu_{ikt})} f(\mu_{it}|\theta). \tag{4.12}$$

Given a guess of θ we can solve the system of nonlinear equations for the vector δ which equates observed and predicted market share $s_{jt} = s_{jt}(\delta, \theta)$. Now we can perform a linear IV GMM regression of the form

$$\delta_{jt}(\theta) = \alpha p_{jt} + x_{jt}\beta^{\text{ex}} + \xi_{jt}. \tag{4.13}$$

The moments are constructed by interacting the predicted residuals $\hat{\xi}_{jt}(\theta)$ with instruments z_{jt} to form

$$\bar{g}(\theta) = \frac{1}{N}\sum_{j,t} z'_{jt}\hat{\xi}_{jt}(\theta). \tag{4.14}$$

4.2.2 Random Coefficients

To include random coefficients we need to add a *Formulation* for the demand-side nonlinear characteristics X_2.

Just like in the logit case we have the same reserved field names in `product_data`:

- `market_ids` are the unique market identifiers which we subscript t.
- `shares` specifies the market share which need to be between zero and one, and within a market ID, $\sum_j s_{jt} < 1$.
- `prices` are prices p_{jt}. These have some special properties and are *always* treated as endogenous.
- `demand_instruments0`, `demand_instruments1`, and so on are numbered demand instruments. These represent only the *excluded* instruments. The exogenous regressors in X_1 (of which X_2 is typically a subset) will be automatically added to the set of instruments.

We proceed with the following steps:

1. Load the `product data` which at a minimum consists of `market_ids`, `shares`, `prices`, and at least a single column of demand instruments, `demand_instruments0`.
2. Define a *Formulation* for the X_1 (linear) demand model.
   - This and all other formulas are similar to R and patsy formulas.
   - It includes a constant by default. To exclude the constant, specify either a `0` or a `-1`.
   - To efficiently include fixed effects, use the `absorb` option and specify which categorical variables you would like to absorb.
   - Some model reduction may happen automatically. The constant will be excluded if you include fixed effects and some precautions are taken against collinearity. However, you will have to make sure that differently-named variables are not collinear.
3. Define a *Formulation* for the X_2 (nonlinear) demand model.
   - Include only the variables over which we want random coefficients.
   - Do not absorb or include fixed effects.

- It will include a random coefficient on the constant (to capture inside good vs. outside good preference) unless you specify not to with a `0` or a `-1`.

4. Define an *Integration* configuration to solve the market share integral from several available options:
   - Monte Carlo integration (pseudo-random draws).
   - Product rule quadrature.
   - Sparse grid quadrature.
5. Combine *Formulation* classes, `product_data`, and the *Integration* configuration to construct a *Problem*.
6. Use the *Problem.solve* method to estimate paramters.
   - It is required to specify an initial guess of the nonlinear parameters. This serves two primary purposes: speeding up estimation and indicating to the solver through initial values of zero which parameters are restricted to be always zero.

4.2.3 Specification of Random Taste Parameters

It is common to assume that $f(\beta_i \mid \theta)$ follows a multivariate normal distribution, and to break it up into three parts:

1. A mean $K_1 \times 1$ taste which all individuals agree on, β.
2. A $K_2 \times K_2$ covariance matrix, V. As is common with multivariate normal distributions, V is not estimated directly. Rather, its matrix square (Cholesky) root Σ is estimated where $\Sigma\Sigma' = V$.
3. Any $K_2 \times D$ interactions, Π, with observed $D \times 1$ demographic data, d_i.

Together this gives us that

$$\beta_i \sim N(\beta + \Pi d_i, \Sigma\Sigma'). \tag{4.15}$$

Problem.solve takes an initial guess Σ_0 of Σ. It guarantees that $\hat{\Sigma}$ (the estimated parameters) will have the same sparsity structure as Σ_0. So any zero element of Σ is restricted to be zero in the solution $\hat{\Sigma}$. For example, a popular restriction is that Σ is diagonal, this can be achieved by passing a diagonal matrix as Σ_0.

4.2.4 Loading Data

The `product_data` argument of *Problem* should be a structured array-like object with fields that store data. Product data can be a structured NumPy array, a pandas DataFrame, or other similar objects.

```
product_data = pd.read_csv(pyblp.data.NEVO_PRODUCTS_LOCATION)
product_data.head()
```

```
  market_ids  city_ids  quarter product_ids  firm_ids  brand_ids    shares  \
0       C01Q1         1        1       F1B04         1          4  0.012417
1       C01Q1         1        1       F1B06         1          6  0.007809
2       C01Q1         1        1       F1B07         1          7  0.012995
3       C01Q1         1        1       F1B09         1          9  0.005770
4       C01Q1         1        1       F1B11         1         11  0.017934

     prices  sugar  mushy  ...  demand_instruments10  demand_instruments11  \
0  0.072088      2      1  ...              2.116358             -0.154708
1  0.114178     18      1  ...             -7.374091             -0.576412
2  0.132391      4      1  ...              2.187872             -0.207346
3  0.130344      3      0  ...              2.704576              0.040748
4  0.154823     12      0  ...              1.261242              0.034836

   demand_instruments12  demand_instruments13  demand_instruments14  \
0             -0.005796              0.014538              0.126244
1              0.012991              0.076143              0.029736
2              0.003509              0.091781              0.163773
3             -0.003724              0.094732              0.135274
4             -0.000568              0.102451              0.130640

   demand_instruments15  demand_instruments16  demand_instruments17  \
0              0.067345              0.068423              0.034800
1              0.087867              0.110501              0.087784
2              0.111881              0.108226              0.086439
3              0.088090              0.101767              0.101777
4              0.084818              0.101075              0.125169

   demand_instruments18  demand_instruments19
0              0.126346              0.035484
1              0.049872              0.072579
2              0.122347              0.101842
3              0.110741              0.104332
4              0.133464              0.121111

[5 rows x 30 columns]
```

The product data contains `market_ids`, `product_ids`, `firm_ids`, `shares`, `prices`, a number of other firm IDs and product characteristics, and some pre-computed excluded `demand_instruments0`, `demand_instruments1`, and so on. The `product_ids` will be incorporated as fixed effects.

For more information about the instruments and the example data as a whole, refer to the *data* module.

4.2.5 Setting Up and Solving the Problem Without Demographics

Formulations, product data, and an integration configuration are collectively used to initialize a *Problem*. Once initialized, *Problem.solve* runs the estimation routine. The arguments to *Problem.solve* configure how estimation is performed. For example, `optimization` and `iteration` arguments configure the optimization and iteration routines that are used by the outer and inner loops of estimation.

We'll specify *Formulation* configurations for X_1, the demand-side linear characteristics, and X_2, the nonlinear characteristics.

- The formulation for X_1 consists of `prices` and fixed effects constructed from `product_ids`, which we will absorb using `absorb` argument of *Formulation*.
- If we were interested in reporting estimates for each fixed effect, we could replace the formulation for X_1 with `Formulation('prices + C(product_ids)')`.
- Because `sugar`, `mushy`, and the constant are collinear with `product_ids`, we can include them in X_2 but not in X_1.

```
X1_formulation = pyblp.Formulation('0 + prices', absorb='C(product_ids)')
X2_formulation = pyblp.Formulation('1 + prices + sugar + mushy')
product_formulations = (X1_formulation, X2_formulation)
product_formulations
```

```
(prices + Absorb[C(product_ids)], 1 + prices + sugar + mushy)
```

We also need to specify an *Integration* configuration. We consider two alternatives:

1. Monte Carlo draws: we simulate 50 individuals from a random normal distribution. This is just for simplicity. In practice quasi-Monte Carlo sequences such as Halton draws are preferable, and there should generally be many more simulated individuals than just 50.
2. Product rules: we construct nodes and weights according to a product rule that exactly integrates polynomials of degree $5 \times 2 - 1 = 9$ or less.

```
mc_integration = pyblp.Integration('monte_carlo', size=50, specification_options={'seed': 0})
mc_integration
```

```
Configured to construct nodes and weights with Monte Carlo simulation with options {seed: 0}.
```

```
pr_integration = pyblp.Integration('product', size=5)
pr_integration
```

```
Configured to construct nodes and weights according to the level-5 Gauss-Hermite product rule with options {}.
```

```
mc_problem = pyblp.Problem(product_formulations, product_data, integration=mc_integration)
mc_problem
```

```
Dimensions:
=============================================
 T    N     F    I     K1    K2    MD    ED
---  ----  ---  ----  ----  ----  ----  ----
94   2256   5   4700   1     4     20    1
=============================================

Formulations:
============================================================
       Column Indices:           0       1       2      3
----------------------------  ------  ------  -----  -----
 X1: Linear Characteristics    prices
X2: Nonlinear Characteristics    1     prices  sugar  mushy
============================================================
```

```
pr_problem = pyblp.Problem(product_formulations, product_data, integration=pr_integration)
pr_problem
```

```
Dimensions:
=============================================
 T    N     F     I     K1    K2    MD    ED
---  ----  ---  -----  ----  ----  ----  ----
94   2256   5   58750  1     4     20    1
=============================================

Formulations:
============================================================
       Column Indices:           0       1       2      3
----------------------------  ------  ------  -----  -----
 X1: Linear Characteristics    prices
X2: Nonlinear Characteristics    1     prices  sugar  mushy
============================================================
```

As an illustration of how to configure the optimization routine, we'll use a simpler, non-default *Optimization* configuration that doesn't support parameter bounds, and use a relatively loose tolerance so the problems are solved quickly. In practice along with more integration draws, it's a good idea to use a tighter termination tolerance.

```
bfgs = pyblp.Optimization('bfgs', {'gtol': 1e-4})
bfgs
```

```
Configured to optimize using the BFGS algorithm implemented in SciPy with analytic gradients and options {gtol: +1.0E-
↪04}.
```

We estimate three versions of the model:

1. An unrestricted covariance matrix for random tastes using Monte Carlo integration.
2. An unrestricted covariance matrix for random tastes using the product rule.
3. A restricted diagonal matrix for random tastes using Monte Carlo Integration.

Notice that the only thing that changes when we estimate the restricted covariance is our initial guess of Σ_0. The upper diagonal in this initial guess is ignored because we are optimizing over the lower-triangular Cholesky root of $V = \Sigma\Sigma'$.

```
results1 = mc_problem.solve(sigma=np.ones((4, 4)), optimization=bfgs)
results1
```

```
Problem Results Summary:
=========================================================================================================
GMM   Objective  Gradient      Hessian         Hessian     Clipped  Weighting Matrix  Covariance Matrix
Step    Value      Norm    Min Eigenvalue  Max Eigenvalue  Shares   Condition Number  Condition Number
----  ---------  --------  --------------  --------------  -------  ----------------  ----------------
 2    +1.5E+02   +8.7E-05     +8.5E-02        +6.5E+03        0         +5.2E+07          +8.3E+05
=========================================================================================================

Cumulative Statistics:
=============================================================================
Computation  Optimizer  Optimization   Objective   Fixed Point  Contraction
   Time      Converged   Iterations   Evaluations  Iterations   Evaluations
-----------  ---------  ------------  -----------  -----------  -----------
 00:00:55       Yes           58           75          80943       248970
=============================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=============================================================================================================================
Sigma:      1         prices      sugar       mushy     |  Sigma Squared:      1         prices      sugar       mushy
------  ----------  ----------  ----------  ----------  |  --------------  ----------  ----------  ----------  ---------
↪-
  1      +1.2E+00                                       |        1          +1.5E+00    -1.4E+01    +7.3E-02    -7.1E-01
        (+3.0E+00)                                      |                  (+7.2E+00)  (+5.2E+01)  (+2.2E-01)  (+2.
↪3E+00)
                                                        |
prices   -1.1E+01    +8.4E+00                           |      prices       -1.4E+01    +2.0E+02    -1.5E+00    +1.5E+00
        (+1.8E+01)  (+1.2E+01)                          |                  (+5.2E+01)  (+3.1E+02)  (+1.2E+00)  (+1.
↪5E+01)
                                                        |
```

(continues on next page)

(continued from previous page)

```
sugar     +6.1E-02     -9.1E-02     +3.8E-02                 |       sugar         +7.3E-02     -1.5E+00     +1.3E-02     +2.0E-02
         (+2.5E-01)   (+2.3E-01)   (+8.3E-02)                |                    (+2.2E-01)   (+1.2E+00)   (+2.8E-02)   (+2.7E-
↪01)
                                                             |
mushy     -5.9E-01     -6.2E-01     -2.3E-02     +4.8E-01    |       mushy         -7.1E-01     +1.5E+00     +2.0E-02     +9.6E-01
         (+2.1E+00)   (+1.5E+00)   (+2.5E+00)   (+1.3E+00)   |                    (+2.3E+00)   (+1.5E+01)   (+2.7E-01)   (+4.
↪0E+00)
================================================================================================================================

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
----------
 -3.1E+01
(+6.0E+00)
==========
```

```
results2 = pr_problem.solve(sigma=np.ones((4, 4)), optimization=bfgs)
results2
```

```
Problem Results Summary:
=========================================================================================================
GMM   Objective  Gradient      Hessian         Hessian     Clipped  Weighting Matrix  Covariance Matrix
Step    Value      Norm    Min Eigenvalue  Max Eigenvalue  Shares   Condition Number  Condition Number
----  ---------  --------  --------------  --------------  -------  ----------------  ----------------
 2    +1.6E+02   +3.2E-05     +1.6E-02        +5.3E+03        0         +5.3E+07          +5.3E+21
=========================================================================================================

Cumulative Statistics:
=============================================================================
Computation  Optimizer  Optimization   Objective   Fixed Point  Contraction
   Time      Converged   Iterations   Evaluations  Iterations   Evaluations
-----------  ---------  ------------  -----------  -----------  -----------
 00:01:30       Yes          64           78          68668       212105
=============================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
================================================================================================================================
Sigma:      1         prices      sugar       mushy      |  Sigma Squared:      1         prices      sugar       mushy
------  ----------  ----------  ----------  ----------  |  --------------  ----------  ----------  ----------  ---------
↪-
```

(continues on next page)

(continued from previous page)

```
  1      -7.4E-01                                          |        1          +5.5E-01    -9.4E+00    +8.3E-02    -1.1E-01
        (+2.1E+00)                                        |                  (+3.1E+00)  (+3.2E+01)  (+1.7E-01)  (+6.3E-
↪01)
                                                          |
prices   +1.3E+01    -8.7E-06                             |      prices       -9.4E+00    +1.6E+02    -1.4E+00    +1.9E+00
        (+7.7E+00)  (+2.3E+03)                            |                  (+3.2E+01)  (+2.0E+02)  (+7.8E-01)  (+8.
↪8E+00)
                                                          |
sugar    -1.1E-01    +1.1E-07    -3.0E-09                 |      sugar         +8.3E-02    -1.4E+00    +1.2E-02    -1.7E-02
        (+2.1E-01)  (+1.9E+05)  (+7.1E+02)                |                  (+1.7E-01)  (+7.8E-01)  (+2.1E-02)  (+1.5E-
↪01)
                                                          |
mushy    +1.5E-01    +4.6E-07    +1.9E-08    -1.4E-08     |      mushy         -1.1E-01    +1.9E+00    -1.7E-02    +2.2E-02
        (+6.8E-01)  (+3.5E+03)  (+6.3E+02)  (+1.8E+02)   |                  (+6.3E-01)  (+8.8E+00)  (+1.5E-01)  (+2.0E-
↪01)
==========================================================================================================================

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
----------
 -3.1E+01
(+4.4E+00)
==========
```

```
results3 = mc_problem.solve(sigma=np.eye(4), optimization=bfgs)
results3
```

```
Problem Results Summary:
=========================================================================================================
GMM   Objective  Gradient      Hessian         Hessian     Clipped  Weighting Matrix  Covariance Matrix
Step    Value      Norm    Min Eigenvalue  Max Eigenvalue  Shares   Condition Number  Condition Number
----  ---------  --------  --------------  --------------  -------  ----------------  ----------------
 2    +1.8E+02   +1.3E-06     +1.1E+00        +6.0E+03        0         +5.8E+07          +4.8E+04
=========================================================================================================

Cumulative Statistics:
=============================================================================
Computation  Optimizer  Optimization   Objective   Fixed Point  Contraction
   Time      Converged   Iterations   Evaluations  Iterations   Evaluations
```

(continues on next page)

(continued from previous page)

```
-----------  ---------  ------------  -----------  -----------  -----------
 00:00:09       Yes           16           24           19599        60673
=============================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=====================================================
Sigma:       1          prices      sugar       mushy
------  ----------  ----------  ----------  ----------
  1      +5.2E-02
        (+1.1E+00)

prices   +0.0E+00    -4.3E-01
                    (+8.0E+00)

sugar    +0.0E+00    +0.0E+00    +3.6E-02
                                (+5.8E-02)

mushy    +0.0E+00    +0.0E+00    +0.0E+00    +5.0E-01
                                             (+1.7E+00)
=====================================================

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
----------
 -3.0E+01
(+1.4E+00)
==========
```

We see that all three models give similar estimates of the price coefficient $\hat{\alpha} \approx -30$. Note a few of the estimated terms on the diagonal of Σ are negative. Since the diagonal consists of standard deviations, negative values are unrealistic. When using another optimization routine that supports bounds (like the default L-BFGS-B routine), these diagonal elements are by default bounded from below by zero.

4.2.6 Adding Demographics to the Problem

To add demographic data we need to make two changes:

1. We need to load `agent_data`, which for this cereal problem contains pre-computed Monte Carlo draws and demographics.
2. We need to add an `agent_formulation` to the model.

The `agent data` has several reserved column names.

- `market_ids` are the index that link the `agent data` to the `market_ids` in `product data`.
- `weights` are the weights w_{it} attached to each agent. In each market, these should sum to one so that $\sum_i w_{it} = 1$. It is often the case the $w_{it} = 1/I_t$ where I_t is the number of agents in market t, so that each agent gets equal weight. Other possibilities include quadrature nodes and weights.
- `nodes0`, `nodes1`, and so on are the nodes at which the unobserved agent tastes μ_{ijt} are evaluated. The nodes should be labeled from $0, \ldots, K_2 - 1$ where K_2 is the number of random coefficients.
- Other fields are the realizations of the demographics d themselves.

```
agent_data = pd.read_csv(pyblp.data.NEVO_AGENTS_LOCATION)
agent_data.head()
```

```
  market_ids  city_ids  quarter  weights    nodes0    nodes1    nodes2  \
0       C01Q1         1        1     0.05  0.434101 -1.500838 -1.151079
1       C01Q1         1        1     0.05 -0.726649  0.133182 -0.500750
2       C01Q1         1        1     0.05 -0.623061 -0.138241  0.797441
3       C01Q1         1        1     0.05 -0.041317  1.257136 -0.683054
4       C01Q1         1        1     0.05 -0.466691  0.226968  1.044424

     nodes3    income  income_squared       age     child
0  0.161017  0.495123        8.331304 -0.230109 -0.230851
1  0.129732  0.378762        6.121865 -2.532694  0.769149
2 -0.795549  0.105015        1.030803 -0.006965 -0.230851
3  0.259044 -1.485481      -25.583605 -0.827946  0.769149
4  0.092019 -0.316597       -6.517009 -0.230109 -0.230851
```

The `agent formulation` tells us which columns of demographic information to interact with X_2.

```
agent_formulation = pyblp.Formulation('0 + income + income_squared + age + child')
agent_formulation
```

```
income + income_squared + age + child
```

This tells us to include demographic realizations for `income`, `income_squared`, `age`, and the presence of children, `child`, but to ignore other possible demographics when interacting demographics with X_2. We should also generally exclude the constant from the demographic formula.

Now we configure the *Problem* to include the `agent_formulation` and `agent_data`, which follow the `product_formulations` and `product_data`.

When we display the class, it lists the demographic interactions in the table of formulations and reports $D = 4$, the dimension of the demographic draws.

```
nevo_problem = pyblp.Problem(
    product_formulations,
    product_data,
    agent_formulation,
    agent_data
)
nevo_problem
```

```
Dimensions:
================================================
 T    N     F    I     K1    K2    D    MD    ED
---  ----  ---  ----  ----  ----  ---  ----  ----
94   2256   5   1880   1     4     4    20    1
================================================

Formulations:
======================================================================
       Column Indices:           0           1            2      3
-----------------------------  ------  --------------  -----  -----
 X1: Linear Characteristics    prices
X2: Nonlinear Characteristics    1         prices       sugar  mushy
       d: Demographics         income  income_squared    age   child
======================================================================
```

The initialized problem can be solved with *Problem.solve*. We'll use the same starting values as *Nevo (2000a)*. By passing a diagonal matrix as starting values for Σ, we're choosing to ignore covariance terms. Similarly, zeros in the starting values for Π mean that those parameters will be fixed at zero.

To replicate common estimates, we'll use the non-default BFGS optimization routine (with a slightly tighter tolerance to avoid getting stuck at a spurious local minimum), and we'll configure *Problem.solve* to use 1-step GMM instead of 2-step GMM.

```
initial_sigma = np.diag([0.3302, 2.4526, 0.0163, 0.2441])
initial_pi = np.array([
  [ 5.4819,  0,      0.2037,  0     ],
  [15.8935, -1.2000, 0,       2.6342],
  [-0.2506,  0,      0.0511,  0     ],
  [ 1.2650,  0,     -0.8091,  0     ]
])
tighter_bfgs = pyblp.Optimization('bfgs', {'gtol': 1e-5})
nevo_results = nevo_problem.solve(
    initial_sigma,
    initial_pi,
    optimization=tighter_bfgs,
```

(continues on next page)

(continued from previous page)

```
    method='1s'
)
nevo_results
```

```
Problem Results Summary:
=========================================================================================================
GMM   Objective  Gradient      Hessian         Hessian     Clipped  Weighting Matrix  Covariance Matrix
Step    Value      Norm    Min Eigenvalue  Max Eigenvalue  Shares   Condition Number  Condition Number
----  ---------  --------  --------------  --------------  -------  ----------------  ----------------
 1    +4.6E+00   +6.9E-06     +5.1E-05        +1.6E+04        0         +6.9E+07          +8.4E+08
=========================================================================================================

Cumulative Statistics:
=============================================================================
Computation  Optimizer  Optimization   Objective   Fixed Point  Contraction
   Time      Converged   Iterations   Evaluations  Iterations   Evaluations
-----------  ---------  ------------  -----------  -----------  -----------
 00:00:42       Yes          51           57          46379       143975
=============================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=============================================================================================================================
Sigma:      1         prices      sugar       mushy      |   Pi:      income    income_squared     age         child
------  ----------  ----------  ----------  ----------  |  ------  ----------  --------------  ----------  ----------
  1      +5.6E-01                                       |    1      +2.3E+00      +0.0E+00      +1.3E+00    +0.0E+00
        (+1.6E-01)                                      |          (+1.2E+00)                   (+6.3E-01)
                                                        |
prices   +0.0E+00    +3.3E+00                           |  prices   +5.9E+02      -3.0E+01      +0.0E+00    +1.1E+01
                    (+1.3E+00)                          |          (+2.7E+02)    (+1.4E+01)                (+4.1E+00)
                                                        |
sugar    +0.0E+00    +0.0E+00    -5.8E-03               |  sugar    -3.8E-01      +0.0E+00      +5.2E-02    +0.0E+00
                                (+1.4E-02)              |          (+1.2E-01)                   (+2.6E-02)
                                                        |
mushy    +0.0E+00    +0.0E+00    +0.0E+00    +9.3E-02   |  mushy    +7.5E-01      +0.0E+00      -1.4E+00    +0.0E+00
                                            (+1.9E-01)  |          (+8.0E-01)                   (+6.7E-01)
=============================================================================================================================

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
----------
```

(continues on next page)

(continued from previous page)

```
 -6.3E+01
(+1.5E+01)
==========
```

Results are similar to those in the original paper with a (scaled) objective value of $q(\hat{\theta}) = 4.65$ and a price coefficient of $\hat{\alpha} = -62.7$.

4.2.7 Restricting Parameters

Because the interactions between `price`, `income`, and `income_squared` are potentially collinear, we might worry that $\hat{\Pi}_{21} = 588$ and $\hat{\Pi}_{22} = -30.2$ are pushing the price coefficient in opposite directions. Both are large in magnitude but statistically insignficant. One way of dealing with this is to restrict $\Pi_{22} = 0$.

There are two ways we can do this:

1. Change the initial Π_0 values to make this term zero.
2. Change the agent formula to drop `income_squared`.

First, we'll change the initial Π_0 values.

```
restricted_pi = initial_pi.copy()
restricted_pi[1, 1] = 0
nevo_problem.solve(
    initial_sigma,
    restricted_pi,
    optimization=tighter_bfgs,
    method='1s'
)
```

```
Problem Results Summary:
=========================================================================================================
GMM   Objective  Gradient      Hessian         Hessian     Clipped  Weighting Matrix  Covariance Matrix
Step    Value      Norm    Min Eigenvalue  Max Eigenvalue  Shares   Condition Number  Condition Number
----  ---------  --------  --------------  --------------  -------  ----------------  ----------------
 1    +1.5E+01   +1.4E-05     +4.7E-02        +1.7E+04        0         +6.9E+07          +5.7E+05
=========================================================================================================

Cumulative Statistics:
==========================================================================
Computation  Optimizer  Optimization   Objective   Fixed Point  Contraction
   Time      Converged   Iterations   Evaluations  Iterations   Evaluations
```

(continues on next page)

(continued from previous page)

```
-----------  ---------  ------------  -----------  -----------  -----------
 00:00:41       No            33           63          51353       159105
============================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=============================================================================================================================
Sigma:      1          prices      sugar       mushy      |   Pi:      income    income_squared     age         child
------  ----------  ----------  ----------  ----------  |  ------  ----------  --------------  ----------  ----------
  1      +3.7E-01                                       |    1      +3.1E+00       +0.0E+00      +1.2E+00    +0.0E+00
        (+1.2E-01)                                      |           (+1.1E+00)                   (+1.0E+00)
                                                        |
prices   +0.0E+00    +1.8E+00                           |  prices   +4.2E+00       +0.0E+00      +0.0E+00    +1.2E+01
                    (+9.2E-01)                          |           (+4.6E+00)                               (+5.2E+00)
                                                        |
sugar    +0.0E+00    +0.0E+00    -4.4E-03               |  sugar    -1.9E-01       +0.0E+00      +2.8E-02    +0.0E+00
                                (+1.2E-02)              |           (+3.5E-02)                   (+3.2E-02)
                                                        |
mushy    +0.0E+00    +0.0E+00    +0.0E+00    +8.6E-02   |  mushy    +1.5E+00       +0.0E+00      -1.5E+00    +0.0E+00
                                            (+1.9E-01)  |           (+6.5E-01)                   (+1.1E+00)
=============================================================================================================================

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
----------
 -3.2E+01
(+2.3E+00)
==========
```

Now we'll drop both `income_squared` and the corresponding column in Π_0.

```
restricted_formulation = pyblp.Formulation('0 + income  + age + child')
deleted_pi = np.delete(initial_pi, 1, axis=1)
restricted_problem = pyblp.Problem(
    product_formulations,
    product_data,
    restricted_formulation,
    agent_data
)
restricted_problem.solve(
```

(continues on next page)

(continued from previous page)

```
    initial_sigma,
    deleted_pi,
    optimization=tighter_bfgs,
    method='1s'
)
```

```
Problem Results Summary:
=========================================================================================================
GMM   Objective  Gradient      Hessian         Hessian     Clipped  Weighting Matrix  Covariance Matrix
Step    Value      Norm    Min Eigenvalue  Max Eigenvalue  Shares   Condition Number  Condition Number
----  ---------  --------  --------------  --------------  -------  ----------------  ----------------
 1    +1.5E+01   +1.4E-05     +4.7E-02        +1.7E+04        0         +6.9E+07          +5.7E+05
=========================================================================================================

Cumulative Statistics:
=========================================================================
Computation  Optimizer  Optimization   Objective   Fixed Point  Contraction
   Time      Converged   Iterations   Evaluations  Iterations   Evaluations
-----------  ---------  ------------  -----------  -----------  -----------
 00:00:45       No           33           63          51353       159105
=========================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=============================================================================================================
Sigma:      1         prices      sugar       mushy      |   Pi:      income       age         child
------  ----------  ----------  ----------  ----------  |  ------  ----------  ----------  ----------
  1      +3.7E-01                                       |    1      +3.1E+00    +1.2E+00    +0.0E+00
        (+1.2E-01)                                      |           (+1.1E+00)  (+1.0E+00)
                                                        |
prices   +0.0E+00    +1.8E+00                           |  prices   +4.2E+00    +0.0E+00    +1.2E+01
                    (+9.2E-01)                          |           (+4.6E+00)              (+5.2E+00)
                                                        |
sugar    +0.0E+00    +0.0E+00    -4.4E-03               |  sugar    -1.9E-01    +2.8E-02    +0.0E+00
                                (+1.2E-02)              |           (+3.5E-02)  (+3.2E-02)
                                                        |
mushy    +0.0E+00    +0.0E+00    +0.0E+00    +8.6E-02   |  mushy    +1.5E+00    -1.5E+00    +0.0E+00
                                            (+1.9E-01)  |           (+6.5E-01)  (+1.1E+00)
=============================================================================================================

Beta Estimates (Robust SEs in Parentheses):
==========
```

(continues on next page)

(continued from previous page)

```
  prices
----------
 -3.2E+01
(+2.3E+00)
==========
```

The parameter estimates and standard errors are identical for both approaches. Based on the number of fixed point iterations, there is some evidence that the solver took a slightly different path for each problem, but both restricted problems arrived at identical answers. The $\hat{\Pi}_{12}$ interaction term is still insignificant.

The online version of the following section may be easier to read.

4.3 Supply Side Tutorial with Automobile Data

```
import pyblp
import numpy as np
import pandas as pd

pyblp.options.digits = 2
pyblp.options.verbose = False
pyblp.__version__
```

```
'1.2.0'
```

In this tutorial, we'll use data from *Berry, Levinsohn, and Pakes (1995)* to solve the paper's automobile problem. This tutorial is similar to the *fake cereal tutorial*, but exhibits some other features of pyblp:

- Incorporating a supply side into demand estimation.
- Allowing for simple price-income demographic effects.
- Calculating clustered standard errors.

4.3.1 Loading Data

We'll use pandas to load two sets of data:

1. `product_data`, which contains prices, shares, and other product characteristics.
2. `agent_data`, which contains draws from the distribution of heterogeneity.

```
product_data = pd.read_csv(pyblp.data.BLP_PRODUCTS_LOCATION)
product_data.head()
```

```
   market_ids clustering_ids  car_ids  firm_ids region    shares    prices  \
0        1971       AMGREM71      129        15     US  0.001051  4.935802
1        1971       AMHORN71      130        15     US  0.000670  5.516049
2        1971       AMJAVL71      132        15     US  0.000341  7.108642
```

(continues on next page)

(continued from previous page)

```
3        1971       AMMATA71      134        15     US  0.000522  6.839506
4        1971       AMAMBS71      136        15     US  0.000442  8.928395

       hpwt  air       mpd  ...  supply_instruments2  supply_instruments3  \
0  0.528997    0  1.888146  ...                  0.0             1.705933
1  0.494324    0  1.935989  ...                  0.0             1.680910
2  0.467613    0  1.716799  ...                  0.0             1.801067
3  0.426540    0  1.687871  ...                  0.0             1.818061
4  0.452489    0  1.504286  ...                  0.0             1.933210

   supply_instruments4  supply_instruments5  supply_instruments6  \
0             1.595656                 87.0           -61.959985
1             1.490295                 87.0           -61.959985
2             1.357703                 87.0           -61.959985
3             1.261347                 87.0           -61.959985
4             1.237365                 87.0           -61.959985

   supply_instruments7  supply_instruments8  supply_instruments9  \
0                  0.0            46.060389            29.786989
1                  0.0            46.060389            29.786989
2                  0.0            46.060389            29.786989
3                  0.0            46.060389            29.786989
4                  0.0            46.060389            29.786989

   supply_instruments10  supply_instruments11
0                   0.0              1.888146
1                   0.0              1.935989
2                   0.0              1.716799
3                   0.0              1.687871
4                   0.0              1.504286

[5 rows x 33 columns]
```

The `product_data` contains market IDs, product IDs, firm IDs, shares, prices, a number of product characteristics, and instruments. The product IDs are called `clustering_ids` because they will be used to compute clustered standard errors. For more information about the instruments and the example data as a whole, refer to the *data* module.

The `agent_data` contains market IDs, integration weights w_{it}, integration nodes ν_{it}, and demographics d_{it}. Here we use the $I_t = 200$ importance sampling weights and nodes from the original paper.

In non-example problems, it is usually a better idea to use many more draws, or a more sophisticated *Integration* configuration such as sparse grid quadrature.

```
agent_data = pd.read_csv(pyblp.data.BLP_AGENTS_LOCATION)
agent_data.head()
```

```
   market_ids   weights    nodes0    nodes1    nodes2    nodes3    nodes4  \
0        1971  0.000543  1.192188  0.478777  0.980830 -0.824410  2.473301
1        1971  0.000723  1.497074 -2.026204 -1.741316  1.412568 -0.747468
2        1971  0.000544  1.438081  0.813280 -1.749974 -1.203509  0.049558
3        1971  0.000701  1.768655 -0.177453  0.286602  0.391517  0.683669
4        1971  0.000549  0.849970 -0.135337  0.735920  1.036247 -1.143436

       income
0  109.560369
1   45.457314
2  127.146548
3   22.604045
4  170.226032
```

4.3.2 Setting up the Problem

Unlike the fake cereal problem, we won't absorb any fixed effects in the automobile problem, so the linear part of demand X_1 has more components. We also need to specify a formula for the random coefficients X_2, including a random coefficient on the constant, which captures correlation among all inside goods.

The primary new addition to the model relative to the fake cereal problem is that we add a supply side formula for product characteristics that contribute to marginal costs, X_3. The patsy-style formulas support functions of regressors such as the `log` function used below.

We stack the three product formulations in order: X_1, X_2, and X_3.

```
product_formulations = (
    pyblp.Formulation('1 + hpwt + air + mpd + space'),
    pyblp.Formulation('1 + prices + hpwt + air + mpd + space'),
    pyblp.Formulation('1 + log(hpwt) + air + log(mpg) + log(space) + trend')
)
product_formulations
```

```
(1 + hpwt + air + mpd + space,
 1 + prices + hpwt + air + mpd + space,
 1 + log(hpwt) + air + log(mpg) + log(space) + trend)
```

The original specification for the automobile problem includes the term $\log(y_i - p_j)$, in which y is income and p are prices. Instead of including this term, which gives rise to a host of numerical problems, we'll follow *Berry, Levinsohn, and Pakes (1999)* and use its first-order linear approximation, p_j/y_i.

The agent formulation for demographics, d, includes a column of $1/y_i$ values, which we'll interact with p_j. To do this, we will treat draws of y_i as demographic variables.

```
agent_formulation = pyblp.Formulation('0 + I(1 / income)')
agent_formulation
```

```
I(1 / income)
```

As in the cereal example, the *Problem* can be constructed by combining the `product_formulations`, `product_data`, `agent_formulation`, and `agent_data`. We'll also choose the functional form of marginal costs c_{jt}. A linear marginal cost specification is the default setting, so we'll need to use the `costs_type` argument of *Problem* to employ the log-linear specification used by *Berry, Levinsohn, and Pakes (1995)*.

When initializing the problem, we get a warning about integration weights not summing to one. This is because the above product data were created by the original paper with importance sampling. To disable this warning, we could increase `pyblp.options.weights_tol`.

```
problem = pyblp.Problem(product_formulations, product_data, agent_formulation, agent_data, costs_type='log')
problem
```

```
Integration weights in the following markets sum to a value that differs from 1 by more than options.weights_tol: all␣
↪markets. Sometimes this is fine, for example when weights were built with importance sampling. Otherwise, it is a␣
↪sign that there is a data problem.
```

```
Dimensions:
=============================================================
 T     N      F     I      K1     K2     K3     D     MD     MS
---   ----   ---   ----   ----   ----   ----   ---   ----   ----
20    2217   26    4000   5      6      6      1     13     18
=============================================================

Formulations:
================================================================================================
       Column Indices:             0          1        2        3           4          5
-----------------------------  --------  ---------  ----  --------  ----------  -----
 X1: Linear Characteristics        1         hpwt      air      mpd        space
X2: Nonlinear Characteristics      1        prices     hpwt     air         mpd       space
X3: Log Cost Characteristics       1       log(hpwt)  air    log(mpg)   log(space)  trend
       d: Demographics          1/income
================================================================================================
```

The problem outputs a table of dimensions:

- T denotes the number of markets.
- N is the length of the dataset (the number of products across all markets).

- F denotes the number of firms.
- $I = \sum_t I_t$ is the total number of agents across all markets (200 draws per market times 20 markets).
- K_1 is the number of linear demand characteristics.
- K_2 is the number of nonlinear demand characteristics.
- K_3 is the number of linear supply characteristics.
- D is the number of demographic variables.
- M_D is the number of demand instruments, including exogenous regressors.
- M_S is the number of supply instruments, including exogenous regressors.

The formulations table describes all four formulas for demand-side linear characteristics, demand-side nonlinear characteristics, supply-side characteristics, and demographics.

4.3.3 Solving the Problem

The only remaining decisions are:

- Choosing Σ and Π starting values, Σ_0 and Π_0.
- Potentially choosing bounds for Σ and Π.

The decisions we will use are:

- Use published estimates as our starting values in Σ_0.
- Interact the inverse of income, $1/y_i$, only with prices, and use the published estimate on $\log(y_i - p_j)$ as our starting value for α in Π_0.
- Bound Σ_0 to be positive since it is a diagonal matrix where the diagonal consists of standard deviations.

When using a routine that supports bounds, it's usually a good idea to set your own more bounds so that the routine doesn't try out large parameter values that create numerical issues.

```
initial_sigma = np.diag([3.612, 0, 4.628, 1.818, 1.050, 2.056])
initial_pi = np.c_[[0, -43.501, 0, 0, 0, 0]]
```

Note that there are only 5 nonzeros on the diagonal of Σ, which means that we only need 5 columns of integration nodes to integrate over these 5 dimensions of unobserved heterogeneity. Indeed, `agent_data` contains exactly 5 columns of nodes. If we were to ignore the $\log(y_i - p_j)$ term (by not configuring Π) and include a term on prices in Σ instead, we would have needed 6 columns of integration nodes in our `agent_data`.

A downside of the log-linear marginal costs specification is that nonpositive estimated marginal costs can create problems for the optimization routine when computing $\log c(\hat{\theta})$. We'll use the `costs_bounds` argument to bound marginal costs from below by a small number.

Finally, as in the original paper, we'll use `W_type` and `se_type` to cluster by product IDs, which were specified as `clustering_ids` in `product_data`, and set `initial_update=True` to update the initial GMM weighting matrix and the mean utility at the starting parameter values.

```
results = problem.solve(
    initial_sigma,
    initial_pi,
    costs_bounds=(0.001, None),
    W_type='clustered',
    se_type='clustered',
    initial_update=True,
)
results
```

```
Problem Results Summary:
=========================================================================================================================
GMM   Objective    Projected    Reduced Hessian  Reduced Hessian  Clipped  Clipped  Weighting Matrix  Covariance Matrix
Step    Value    Gradient Norm  Min Eigenvalue   Max Eigenvalue   Shares    Costs   Condition Number  Condition Number
----  ---------  -------------  ---------------  ---------------  -------  -------  ----------------  ----------------
 2    +5.0E+02     +3.9E-07        +4.8E-01         +5.1E+02         0        0         +4.2E+09          +3.8E+08
=========================================================================================================================

Cumulative Statistics:
=============================================================================
Computation  Optimizer  Optimization   Objective   Fixed Point  Contraction
   Time      Converged   Iterations   Evaluations  Iterations   Evaluations
-----------  ---------  ------------  -----------  -----------  -----------
 00:02:52       No           58           182         53501       164123
=============================================================================

Nonlinear Coefficient Estimates (Robust SEs Adjusted for 999 Clusters in Parentheses):
=============================================================================================================
Sigma:      1         prices      hpwt        air         mpd         space      |   Pi:     1/income
------  ----------  --------  ----------  ----------  ----------  ----------  |  ------  ----------
  1      +2.0E+00                                                             |    1      +0.0E+00
        (+6.1E+00)                                                            |
                                                                              |
prices   +0.0E+00   +0.0E+00                                                  |  prices   -4.5E+01
                                                                              |          (+9.2E+00)
                                                                              |
```

(continues on next page)

(continued from previous page)

```
 hpwt     +0.0E+00    +0.0E+00    +6.1E+00                                          |   hpwt     +0.0E+00
                                 (+2.2E+00)                                        |
                                                                                   |
 air      +0.0E+00    +0.0E+00    +0.0E+00     +4.0E+00                             |   air      +0.0E+00
                                               (+2.1E+00)                          |
                                                                                   |
 mpd      +0.0E+00    +0.0E+00    +0.0E+00     +0.0E+00     +2.5E-01               |   mpd      +0.0E+00
                                                           (+5.5E-01)              |
                                                                                   |
space     +0.0E+00    +0.0E+00    +0.0E+00     +0.0E+00     +0.0E+00     +1.9E+00  |  space     +0.0E+00
                                                                       (+1.1E+00)  |
=============================================================================================================

Beta Estimates (Robust SEs Adjusted for 999 Clusters in Parentheses):
============================================================
    1           hpwt         air          mpd         space
----------  ----------  ----------  ----------  ----------
 -7.3E+00    +3.5E+00    -1.0E+00    +4.2E-01    +4.2E+00
(+2.8E+00)  (+1.4E+00)  (+2.1E+00)  (+2.5E-01)  (+6.6E-01)
============================================================

Gamma Estimates (Robust SEs Adjusted for 999 Clusters in Parentheses):
=====================================================================
    1       log(hpwt)       air        log(mpg)   log(space)    trend
----------  ----------  ----------  ----------  ----------  ----------
 +2.8E+00    +9.0E-01    +4.2E-01    -5.2E-01    -2.6E-01    +2.7E-02
(+1.2E-01)  (+7.2E-02)  (+8.7E-02)  (+7.3E-02)  (+2.1E-01)  (+3.1E-03)
=====================================================================
```

There are some discrepancies between our results and the original paper, but results are similar.

The online version of the following section may be easier to read.

4.4 Micro Moments Tutorial with Automobile Data

```
from IPython.display import display, HTML
display(HTML("<style>pre { white-space: pre !important; }</style>"))

import pyblp
import numpy as np
import pandas as pd

pyblp.options.digits = 2
pyblp.options.verbose = False
pyblp.__version__
```

```
<IPython.core.display.HTML object>
```

```
'1.2.0'
```

In this tutorial, we'll use data from *Petrin (2002)* to solve the paper's automobile problem. This tutorial is similar to the *first automobile tutorial*, but exhibits how to incorporate micro moments into estimation.

4.4.1 Loading Data

We'll use pandas to load two sets of data:

1. `product_data`, which contains prices, shares, and other product characteristics.
2. `agent_data`, which contains draws from the distribution of heterogeneity.

```
product_data = pd.read_csv(pyblp.data.PETRIN_PRODUCTS_LOCATION)
product_data.head()
```

```
   market_ids  clustering_ids  firm_ids region  jp  eu       q  households  \
0        1981               1         7     EU   0   1  18.647       83527
1        1981               2         7     EU   0   1  17.611       83527
2        1981               2         7     EU   0   1   6.139       83527
3        1981               3         7     EU   0   1   2.553       83527
```

(continues on next page)

(continued from previous page)

```
4        1981               4        15     US   0   0  43.198       83527

     shares     prices  ...  supply_instruments6  supply_instruments7  \
0  0.000223  10.379538  ...                  9.0                  0.0
1  0.000211  13.140814  ...                  9.0                  0.0
2  0.000073  19.746975  ...                  9.0                  0.0
3  0.000031  13.085809  ...                  9.0                  0.0
4  0.000517   6.660066  ...                  0.0                  0.0

   supply_instruments8  supply_instruments9  supply_instruments10  \
0                  0.0                144.0           -151.682461
1                  0.0                144.0           -151.682461
2                  0.0                144.0           -151.682461
3                  0.0                144.0           -151.682461
4                  0.0                149.0           -157.647246

   supply_instruments11  supply_instruments12  supply_instruments13  \
0            108.724278                  30.0                  32.0
1            108.724278                  30.0                  32.0
2            108.724278                  30.0                  32.0
3            108.724278                  30.0                  32.0
4            114.055507                  30.0                  35.0

   supply_instruments14  supply_instruments15
0                  32.0            460.419731
1                  32.0            460.419731
2                  32.0            460.419731
3                  32.0            460.419731
4                  42.0            467.806186

[5 rows x 62 columns]
```

The `product_data` contains market IDs, product IDs, firm IDs, shares, prices, a number of product characteristics, and instruments. The product IDs are called `clustering_ids` because they will be used to compute clustered standard errors. For more information about the instruments and the example data as a whole, refer to the *data* module.

The `agent_data` contains market IDs, integration weights w_{it}, integration nodes ν_{it}, and demographics d_{it}. Here we use $I_t = 1000$ scrambled Halton draws in each market, along with demographics resampled from the Consumer Expenditure Survey (CEX) used by the original paper. These draws are slightly different from those used in the original paper (pseudo Monte Carlo draws and importance sampling). Note that following the original paper, the integration nodes are actually draws from a truncated $\chi^2(3)$ distribution, rather than the more typical $N(0, 1)$ draws that we have seen in prior tutorials.

```
agent_data = pd.read_csv(pyblp.data.PETRIN_AGENTS_LOCATION)
agent_data.head()
```

```
   market_ids  weights    nodes0    nodes1    nodes2    nodes3    nodes4  \
0        1981    0.001  2.533314  7.496742  2.649343  3.892549  0.833761
1        1981    0.001  4.422582  0.858539  1.646447  2.973352  0.033288
2        1981    0.001  1.341509  5.041918  4.118932  2.166338  1.314582
3        1981    0.001  3.324113  2.354892  0.802351  0.261043  3.911970
4        1981    0.001  1.895857  1.807990  1.827797  4.080565  1.709768

     nodes5        fv     income  low  mid  high  fs  age
0  1.928344  0.749785  10.346577    1    0     0   4    1
1  1.683242  5.232336  13.944210    0    1     0   2    1
2  0.360087  1.860212   5.898788    1    0     0   4    0
3  1.027856  6.980909   8.125445    1    0     0   2    0
4  0.707514  2.450663  34.397295    0    0     1   2    1
```

4.4.2 Setting up the Problem

The problem configuration is based on that of the first automobile problem. It is very similar, with both demand and supply sides, although with a few more product characteristics.

Again, we stack the three product formulations in order: X_1, X_2, and X_3.

```
product_formulations = (
    pyblp.Formulation('1 + hpwt + space + air + mpd + fwd + mi + sw + su + pv + pgnp + trend + trend2'),
    pyblp.Formulation('1 + I(-prices) + hpwt + space + air + mpd + fwd + mi + sw + su + pv'),
    pyblp.Formulation('1 + log(hpwt) + log(wt) + log(mpg) + air + fwd + trend * (jp + eu) + log(q)'),
)
product_formulations
```

```
(1 + hpwt + space + air + mpd + fwd + mi + sw + su + pv + pgnp + trend + trend2,
 1 + I(-prices) + hpwt + space + air + mpd + fwd + mi + sw + su + pv,
 1 + log(hpwt) + log(wt) + log(mpg) + air + fwd + trend + jp + eu + trend:jp + trend:eu + log(q))
```

Again, we'll use a first-order linear approximation to $\log(y_i - p_j)$, in which y is income and p are prices. Unlike the previous automobile problem, however, we'll allow its coefficient to vary for low- mid- and high-income consumers.

As in the original paper, we'll also include $log(fs_i) \times fv_i$ where fs_i is family size and fv_i is another truncated $\chi^2(3)$ draw. Finally, to help with constructing micro moments below, we'll also include various additional demographics in the agent formulation.

```
agent_formulation = pyblp.Formulation('1 + I(low / income) + I(mid / income) + I(high / income) + I(log(fs) * fv) + age
↪+ fs + mid + high')
agent_formulation
```

```
1 + I(low / income) + I(mid / income) + I(high / income) + I(log(fs) * fv) + age + fs + mid + high
```

The *Problem* can again be constructed by combining the `product_formulations`, `product_data`, `agent_formulation`, and `agent_data`. We'll again choose a log-linear specification for marginal costs c_{jt}.

```
problem = pyblp.Problem(product_formulations, product_data, agent_formulation, agent_data, costs_type='log')
problem
```

```
Dimensions:
=============================================================
 T     N      F      I      K1     K2     K3     D     MD     MS
---   ----   ---   -----   ----   ----   ----   ---   ----   ----
13    2407   27    13000   13     11     12     9     35     28
=============================================================

Formulations:
=============================================================================================================================================
       Column Indices:           0        1            2             3             4          5      6      7     8         9          
↪ 10        11       12
-----------------------------  ---  ----------  -----------  ------------  -----------  ---  -------  ---  -----  ---------  --
↪------  -------  -------
 X1: Linear Characteristics     1       hpwt        space          air          mpd       fwd    mi     sw     su       pv        
↪pgnp     trend    trend2
X2: Nonlinear Characteristics   1     -prices        hpwt         space         air       mpd    fwd    mi     sw       su        
↪ pv
X3: Log Cost Characteristics    1    log(hpwt)     log(wt)      log(mpg)        air       fwd   trend   jp     eu    jp*trend   
↪eu*trend  log(q)
       d: Demographics          1    low/income  mid/income  high/income  fv*log(fs)   age    fs     mid   high                
=============================================================================================================================================
```

The problem outputs a table of dimensions:

- T denotes the number of markets.
- N is the length of the dataset (the number of products across all markets).
- F denotes the number of firms.
- $I = \sum_t I_t$ is the total number of agents across all markets (1000 draws per market times 13 markets).

- K_1 is the number of linear demand characteristics.
- K_2 is the number of nonlinear demand characteristics.
- K_3 is the number of linear supply characteristics.
- D is the number of demographic variables.
- M_D is the number of demand instruments, including exogenous regressors.
- M_S is the number of supply instruments, including exogenous regressors.

The formulations table describes all four formulas for demand-side linear characteristics, demand-side nonlinear characteristics, supply-side characteristics, and demographics.

4.4.3 Setting up Micro Moments

Next, we will configure the micro moments that we will be adding to the problem. For background and notation involving micro moments, see *Micro Moments*.

Specifically, we will be adding a few more moments that match key statistics computed from the CEX survey of potential automobile consumers. For a tutorial on how to compute optimal micro moments that use all the information in a full micro dataset linking individual choices to demographics, see the *post estimation tutorial*.

To start, we will have to define a *MicroDataset* configuration that contains metadata about the micro dataset/survey. These metadata include a unique name for the dataset indexed by d, the number of observations N_d, a function that defines survey weights w_{dijt}, and if relevant, a subset of markets from which the micro data was sampled.

```
micro_dataset = pyblp.MicroDataset(
    name="CEX",
    observations=29125,
    compute_weights=lambda t, p, a: np.ones((a.size, 1 + p.size)),
)
micro_dataset
```

```
CEX: 29125 Observations in All Markets
```

We called the dataset "CEX", defined the number of observations in it, and also defined a lambda function for computing survey weights in a market. The `compute_weights` function has three arguments: the current market's ID t, the J_t *Products* inside the market, and the I_t *Agents* inside the market. In this case, we are assuming that each product and agent/consumer type are sampled with equal probability, so we simply return a matrix of ones of shape $I_t \times (1 + J_t)$. This sets each $w_{dijt} = 1$.

By using $1 + J_t$ instead of J_t, we are specifying that the micro dataset contains observations of the outside option $j = 0$. If we instead specified a matrix of shape $I_t \times J_t$, this would be the same as setting the first column equal to all zeros, so that outside choices are not sampled from.

We will be matching a few different statistics that were computed from this survey. For convenience, they are packaged in a data file with pyblp.

```
micro_statistics = pd.read_csv(pyblp.data.PETRIN_VALUES_LOCATION, index_col=0)
micro_statistics
```

```
                value
E[age | mi]    0.7830
E[fs | mi]     3.8600
E[age | sw]    0.7300
E[fs | sw]     3.1700
E[age | su]    0.7400
E[fs | su]     2.9700
E[age | pv]    0.6520
E[fs | pv]     3.4700
E[new | mid]   0.0794
E[new | high]  0.1581
```

We will match the average age and family size ("fs") conditional on purchasing a minivan ("mi"), station wagon ("sw"), sport-utility ("su"), and full-size passenger van ("pv"). We will also match the probability that a consumer actually purchases a new vehicle, conditional on them being mid- and high-income.

Each of these statistics is a conditional expectation, which we can rewrite as a ration of unconditional expectations over all consumers. Each of these unconditional expectations is called a *MicroPart* (used to form full micro moments), which we will now configure.

Each micro part is an average/expectation in the sample/population over micro values v_{pijt}. To match the above micro values, we will need averages/expectations over interactions between agent/family size and dummies for purchasing the different automobile types. These will form the numerators in our conditional expectations.

```
age_mi_part = pyblp.MicroPart(
    name="E[age_i * mi_j]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 5], np.r_[0, p.X2[:, 7]]),
)
age_sw_part = pyblp.MicroPart(
    name="E[age_i * sw_j]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 5], np.r_[0, p.X2[:, 8]]),
)
age_su_part = pyblp.MicroPart(
    name="E[age_i * su_j]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 5], np.r_[0, p.X2[:, 9]]),
)
```

(continues on next page)

(continued from previous page)

```
age_pv_part = pyblp.MicroPart(
    name="E[age_i * pv_j]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 5], np.r_[0, p.X2[:, 10]]),
)
fs_mi_part = pyblp.MicroPart(
    name="E[fs_i * mi_j]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 6], np.r_[0, p.X2[:, 7]]),
)
fs_sw_part = pyblp.MicroPart(
    name="E[fs_i * sw_j]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 6], np.r_[0, p.X2[:, 8]]),
)
fs_su_part = pyblp.MicroPart(
    name="E[fs_i * su_j]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 6], np.r_[0, p.X2[:, 9]]),
)
fs_pv_part = pyblp.MicroPart(
    name="E[fs_i * pv_j]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 6], np.r_[0, p.X2[:, 10]]),
)
```

We will also need the denominators, which are simple averages/expectations of purchasing the different types of automobiles.

```
mi_part = pyblp.MicroPart(
    name="E[mi_j]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 0], np.r_[0, p.X2[:, 7]]),
)
sw_part = pyblp.MicroPart(
    name="E[sw_j]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 0], np.r_[0, p.X2[:, 8]]),
)
su_part = pyblp.MicroPart(
    name="E[su_j]",
```

(continues on next page)

(continued from previous page)

```
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 0], np.r_[0, p.X2[:, 9]]),
)
pv_part = pyblp.MicroPart(
    name="E[pv_j]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 0], np.r_[0, p.X2[:, 10]]),
)
```

To form our probability that a consumer actually purchases a new vehicle, conditional on them being mid- and high-income, we will also need the following micro parts.

```
inside_mid_part = pyblp.MicroPart(
    name="E[1{j > 0} * mid_i]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 7], np.r_[0, p.X2[:, 0]]),
)
inside_high_part = pyblp.MicroPart(
    name="E[1{j > 0} * high_i]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 8], np.r_[0, p.X2[:, 0]]),
)
mid_part = pyblp.MicroPart(
    name="E[mid_i]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 7], np.r_[1, p.X2[:, 0]]),
)
high_part = pyblp.MicroPart(
    name="E[high_i]",
    dataset=micro_dataset,
    compute_values=lambda t, p, a: np.outer(a.demographics[:, 8], np.r_[1, p.X2[:, 0]]),
)
```

Finally, we'll put these micro parts together into *MicroMoment*s. Each micro moment is configured to have a name, a value (one of the statistics above), and micro parts that go into it.

If our micro moments were simple unconditional expectations, we could just pass a single micro part to each micro moment and be done. However, since our micro moments are functions of multiple micro parts, we have to specify this function. We also have to specify its derivative for computing standard errors and analytic objective gradients.

```
compute_ratio = lambda v: v[0] / v[1]
compute_ratio_gradient = lambda v: [1 / v[1], -v[0] / v[1]**2]
```

Given our functions that define a conditional expectation and its derivatives, we can form our micro moments.

```
micro_moments = [
    pyblp.MicroMoment(
        name="E[age_i | mi_j]",
        value=0.783,
        parts=[age_mi_part, mi_part],
        compute_value=compute_ratio,
        compute_gradient=compute_ratio_gradient,
    ),
    pyblp.MicroMoment(
        name="E[age_i | sw_j]",
        value=0.730,
        parts=[age_sw_part, sw_part],
        compute_value=compute_ratio,
        compute_gradient=compute_ratio_gradient,
    ),
    pyblp.MicroMoment(
        name="E[age_i | su_j]",
        value=0.740,
        parts=[age_su_part, su_part],
        compute_value=compute_ratio,
        compute_gradient=compute_ratio_gradient,
    ),
    pyblp.MicroMoment(
        name="E[age_i | pv_j]",
        value=0.652,
        parts=[age_pv_part, pv_part],
        compute_value=compute_ratio,
        compute_gradient=compute_ratio_gradient,
    ),
    pyblp.MicroMoment(
        name="E[fs_i | mi_j]",
        value=3.86,
        parts=[fs_mi_part, mi_part],
        compute_value=compute_ratio,
        compute_gradient=compute_ratio_gradient,
    ),
```

(continues on next page)

(continued from previous page)

```
    pyblp.MicroMoment(
        name="E[fs_i | sw_j]",
        value=3.17,
        parts=[fs_sw_part, sw_part],
        compute_value=compute_ratio,
        compute_gradient=compute_ratio_gradient,
    ),
    pyblp.MicroMoment(
        name="E[fs_i | su_j]",
        value=2.97,
        parts=[fs_su_part, su_part],
        compute_value=compute_ratio,
        compute_gradient=compute_ratio_gradient,
    ),
    pyblp.MicroMoment(
        name="E[fs_i | pv_j]",
        value=3.47,
        parts=[fs_pv_part, pv_part],
        compute_value=compute_ratio,
        compute_gradient=compute_ratio_gradient,
    ),
    pyblp.MicroMoment(
        name="E[1{j > 0} | mid_i]",
        value=0.0794,
        parts=[inside_mid_part, mid_part],
        compute_value=compute_ratio,
        compute_gradient=compute_ratio_gradient,
    ),
    pyblp.MicroMoment(
        name="E[1{j > 0} | high_i]",
        value=0.1581,
        parts=[inside_high_part, high_part],
        compute_value=compute_ratio,
        compute_gradient=compute_ratio_gradient,
    ),
]
```

4.4.4 Solving the Problem

Like for the first automobile problem, here will will just use the publisehd estimates for Σ and Π starting values.

```
initial_sigma = np.diag([3.23, 0, 4.43, 0.46, 0.01, 2.58, 4.42, 0, 0, 0, 0])
initial_pi = np.array([
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 7.52, 31.13, 34.49, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0.57, 0, 0, 0, 0],
    [0, 0, 0, 0, 0.28, 0, 0, 0, 0],
    [0, 0, 0, 0, 0.31, 0, 0, 0, 0],
    [0, 0, 0, 0, 0.42, 0, 0, 0, 0],
])
```

Finally, as in the original paper, we'll use `W_type` and `se_type` to cluster by product IDs, which were specified as `clustering_ids` in `product_data`. We will use a simple BFGS optimization routine and slightly loosen the default tolerance of our inner SQUAREM iteration algorithm from `1e-14` to `1e-13` because the tighter tolerance tended to lead to convergence failures for this problem. We also pass our configured `micro_moments` when solving the problem.

```
results = problem.solve(
    sigma=initial_sigma,
    pi=initial_pi,
    optimization=pyblp.Optimization('bfgs', {'gtol': 1e-4}),
    iteration=pyblp.Iteration('squarem', {'atol': 1e-13}),
    se_type='clustered',
    W_type='clustered',
    micro_moments=micro_moments,
)
results
```

```
Problem Results Summary:
=========================================================================================================
GMM   Objective  Gradient      Hessian         Hessian     Clipped  Weighting Matrix  Covariance Matrix
Step    Value      Norm    Min Eigenvalue  Max Eigenvalue  Shares   Condition Number  Condition Number
----  ---------  --------  --------------  --------------  -------  ----------------  ----------------
 2    +1.8E+02   +4.3E-05     +2.4E-01        +1.3E+03        0         +2.8E+11          +8.1E+07
=========================================================================================================
```

(continues on next page)

(continued from previous page)

```
Cumulative Statistics:
=============================================================================
Computation  Optimizer  Optimization   Objective   Fixed Point  Contraction
   Time      Converged   Iterations   Evaluations  Iterations   Evaluations
-----------  ---------  ------------  -----------  -----------  -----------
 00:17:25       Yes          72           87          10903        33588
=============================================================================

Nonlinear Coefficient Estimates (Robust SEs Adjusted for 898 Clusters in Parentheses):
=============================================================================================================================================
Sigma:       1       -prices      hpwt       space         air          mpd          fwd          mi          sw          su
    pv     |    Pi:       1      low/income  mid/income  high/income  fv*log(fs)     age          fs          mid         high
-------  ----------  --------  ----------  ----------  ----------  ----------  ----------  --------  --------  --------  --------
 --------  |  -------  --------  ----------  ----------  ----------  ----------  ----------  --------  --------  --------  ------
--
   1      +3.0E-02
            |     1     +0.0E+00    +0.0E+00    +0.0E+00    +0.0E+00     +0.0E+00    +0.0E+00  +0.0E+00  +0.0E+00  +0.
0E+00
         (+5.3E-01)
            |
            |
-prices    +0.0E+00   +0.0E+00
            |  -prices  +0.0E+00    +3.9E+00    +1.2E+01    +2.4E+01     +0.0E+00    +0.0E+00  +0.0E+00  +0.0E+00  +0.
0E+00
                                  (+3.6E-01)  (+1.0E+00)  (+2.4E+00)
            |
 hpwt      +0.0E+00   +0.0E+00   +1.2E-01
            |   hpwt    +0.0E+00    +0.0E+00    +0.0E+00    +0.0E+00     +0.0E+00    +0.0E+00  +0.0E+00  +0.0E+00  +0.
0E+00
                                (+8.1E-01)
            |
            |
 space     +0.0E+00   +0.0E+00   +0.0E+00    -9.2E-02
            |   space   +0.0E+00    +0.0E+00    +0.0E+00    +0.0E+00     +0.0E+00    +0.0E+00  +0.0E+00  +0.0E+00  +0.
0E+00
                                           (+6.1E-01)
            |
```

(continues on next page)

(continued from previous page)

```
↪            |
  air      +0.0E+00    +0.0E+00    +0.0E+00     +0.0E+00     -1.3E+00
↪            |     air     +0.0E+00    +0.0E+00     +0.0E+00     +0.0E+00      +0.0E+00    +0.0E+00  +0.0E+00  +0.0E+00  +0.
↪0E+00
                                                  (+1.1E+00)
↪            |

↪            |
  mpd      +0.0E+00    +0.0E+00    +0.0E+00     +0.0E+00     +0.0E+00     -1.6E-01
↪            |     mpd     +0.0E+00    +0.0E+00     +0.0E+00     +0.0E+00      +0.0E+00    +0.0E+00  +0.0E+00  +0.0E+00  +0.
↪0E+00
                                                               (+2.2E-01)
↪            |

↪            |
  fwd      +0.0E+00    +0.0E+00    +0.0E+00     +0.0E+00     +0.0E+00     +0.0E+00     +1.6E+00
↪            |     fwd     +0.0E+00    +0.0E+00     +0.0E+00     +0.0E+00      +0.0E+00    +0.0E+00  +0.0E+00  +0.0E+00  +0.
↪0E+00
                                                                        (+3.7E-01)
↪            |

↪            |
  mi       +0.0E+00    +0.0E+00    +0.0E+00     +0.0E+00     +0.0E+00     +0.0E+00     +0.0E+00    +0.0E+00
↪            |     mi      +0.0E+00    +0.0E+00     +0.0E+00     +0.0E+00      +4.2E-01    +0.0E+00  +0.0E+00  +0.0E+00  +0.
↪0E+00

↪            |                                                         (+5.2E-02)

↪            |
  sw       +0.0E+00    +0.0E+00    +0.0E+00     +0.0E+00     +0.0E+00     +0.0E+00     +0.0E+00    +0.0E+00  +0.0E+00
↪            |     sw      +0.0E+00    +0.0E+00     +0.0E+00     +0.0E+00      +1.7E-01    +0.0E+00  +0.0E+00  +0.0E+00  +0.
↪0E+00

↪            |                                                         (+4.2E-02)

↪            |
  su       +0.0E+00    +0.0E+00    +0.0E+00     +0.0E+00     +0.0E+00     +0.0E+00     +0.0E+00    +0.0E+00  +0.0E+00  +0.0E+00
↪            |     su      +0.0E+00    +0.0E+00     +0.0E+00     +0.0E+00      +1.0E-01    +0.0E+00  +0.0E+00  +0.0E+00  +0.
↪0E+00
```

(continues on next page)

(continued from previous page)

```
↪           |                                                      (+5.2E-02)

↪           |
  pv      +0.0E+00   +0.0E+00   +0.0E+00    +0.0E+00    +0.0E+00    +0.0E+00    +0.0E+00   +0.0E+00  +0.0E+00  +0.0E+00
↪ +0.0E+00  |    pv      +0.0E+00   +0.0E+00    +0.0E+00    +0.0E+00     +2.5E-01   +0.0E+00  +0.0E+00  +0.0E+00  +0.
↪0E+00

↪           |                                                      (+8.1E-02)
==========================================================================================================================================

Beta Estimates (Robust SEs Adjusted for 898 Clusters in Parentheses):
==========================================================================================================================================
    1          hpwt       space       air         mpd         fwd         mi          sw          su          pv
↪   pgnp       trend       trend2
----------  ----------  ----------  ----------  ----------  ----------  ----------  ----------  ----------  ----------
↪----------  ----------  ----------
 -8.9E+00    +8.3E+00    +4.9E+00    +3.8E+00    -1.4E-01    -6.5E+00    -2.1E+00    -1.3E+00    -1.1E+00    -3.3E+00
↪ +3.4E-02    +2.2E-01    -1.5E-02
(+1.4E+00)  (+2.4E+00)  (+1.6E+00)  (+1.2E+00)  (+3.2E-01)  (+1.8E+00)  (+4.8E-01)  (+2.0E-01)  (+2.8E-01)  (+5.2E-01)
↪(+1.2E-02)  (+9.2E-02)  (+6.4E-03)
==========================================================================================================================================

Gamma Estimates (Robust SEs Adjusted for 898 Clusters in Parentheses):
==========================================================================================================================================
    1       log(hpwt)    log(wt)     log(mpg)      air         fwd        trend         jp          eu       jp*trend
↪ eu*trend     log(q)
----------  ----------  ----------  ----------  ----------  ----------  ----------  ----------  ----------  ----------
↪----------  ----------
 +1.4E+00    +8.8E-01    +1.4E+00    +1.2E-01    +2.7E-01    +6.9E-02    -1.2E-02    +1.0E-01    +4.6E-01    +1.6E-03
↪ -1.1E-02    -6.9E-02
(+1.4E-01)  (+4.9E-02)  (+8.0E-02)  (+6.0E-02)  (+2.4E-02)  (+1.8E-02)  (+2.6E-03)  (+2.5E-02)  (+4.3E-02)  (+2.9E-03)
↪(+4.2E-03)  (+6.7E-03)
==========================================================================================================================================

Estimated Micro Moments:
==============================================================================================================
Observed  Estimated  Difference         Moment                  Part           Dataset  Observations  Markets
--------  ---------  ----------  --------------------  --------------------  -------  ------------  -------
+7.8E-01  +7.5E-01    +2.9E-02     E[age_i | mi_j]       E[age_i * mi_j]       CEX       29125        All
```

(continues on next page)

(continued from previous page)

```
                                                                  E[mi_j]           CEX       29125        All
+7.3E-01  +6.8E-01    +4.7E-02     E[age_i | sw_j]       E[age_i * sw_j]       CEX       29125        All
                                                                  E[sw_j]           CEX       29125        All
+7.4E-01  +6.8E-01    +5.9E-02     E[age_i | su_j]       E[age_i * su_j]       CEX       29125        All
                                                                  E[su_j]           CEX       29125        All
+6.5E-01  +7.3E-01    -7.7E-02     E[age_i | pv_j]       E[age_i * pv_j]       CEX       29125        All
                                                                  E[pv_j]           CEX       29125        All
+3.9E+00  +3.9E+00    -1.2E-02      E[fs_i | mi_j]        E[fs_i * mi_j]       CEX       29125        All
                                                                  E[mi_j]           CEX       29125        All
+3.2E+00  +3.2E+00    -7.6E-03      E[fs_i | sw_j]        E[fs_i * sw_j]       CEX       29125        All
                                                                  E[sw_j]           CEX       29125        All
+3.0E+00  +3.0E+00    -8.5E-03      E[fs_i | su_j]        E[fs_i * su_j]       CEX       29125        All
                                                                  E[su_j]           CEX       29125        All
+3.5E+00  +3.5E+00    -1.7E-02      E[fs_i | pv_j]        E[fs_i * pv_j]       CEX       29125        All
                                                                  E[pv_j]           CEX       29125        All
+7.9E-02  +8.0E-02    -4.5E-04   E[1{j > 0} | mid_i]   E[1{j > 0} * mid_i]     CEX       29125        All
                                                                  E[mid_i]          CEX       29125        All
+1.6E-01  +1.6E-01    -2.1E-03   E[1{j > 0} | high_i]  E[1{j > 0} * high_i]    CEX       29125        All
                                                                 E[high_i]          CEX       29125        All
==========================================================================================================
```

There are some discrepances between these results and those in the original paper, but broadly estimates are similar. Although the estimates of β looks substantially off, this is primarily because the $\chi^2(3)$ distributions are not mean-zero, so differences in estimates of Σ results in shifted estimates of β too.

4.4.5 Running the Main Counterfactual

One result that is very similar is the paper's headline number: a $367.29 million compensating variation from a counterfactual that removes the minivan in 1984. Using our estimates, we get a very similar number.

This subsection previews some of the routines used in the *next tutorial* on functions available after estimation. First, we will compute implied marginal costs in 1984.

```
year = 1984
costs_1984 = results.compute_costs(market_id=year)
```

Next, we will set up a counterfactual simulation in which the minivan is removed.

```
product_data_1984 = product_data[product_data['market_ids'] == year]
xi_1984 = results.xi[product_data['market_ids'] == year]
```

(continues on next page)

(continued from previous page)

```
agent_data_1984 = agent_data[agent_data['market_ids'] == year]
simulation = pyblp.Simulation(
    product_formulations=product_formulations[:2],
    product_data=product_data_1984[product_data_1984['mi'] == 0],
    xi=xi_1984[product_data_1984['mi'] == 0],
    agent_formulation=problem.agent_formulation,
    agent_data=agent_data_1984,
    beta=results.beta,
    sigma=results.sigma,
    pi=results.pi,
)
```

We will then solve for equilibrium prices and shares under this counterfactual, using the above-computed marginal costs.

```
simulation_results = simulation.replace_endogenous(costs=costs_1984[product_data_1984['mi'] == 0])
```

Finally, we will compute the change in consumer surplus.

```
households = product_data_1984['households'].values[0]
cs = households * results.compute_consumer_surpluses(market_id=year)
counterfactual_cs = households * simulation_results.compute_consumer_surpluses()
cs - counterfactual_cs
```

```
array([[425.90825081]])
```

We get an estimate that is in the same ballpark as \$367.29 million. When bootstrapping this procedure (see the *next tutorial* for more on this), we get a standard error around \$250 million.

The online version of the following section may be easier to read.

4.5 Post-Estimation Tutorial

```
%matplotlib inline

import pyblp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

pyblp.options.digits = 2
pyblp.options.verbose = False
pyblp.__version__
```

```
'1.2.0'
```

This tutorial covers several features of `pyblp` which are available after estimation including:

1. Calculating elasticities and diversion ratios.
2. Calculating marginal costs and markups.
3. Computing the effects of mergers: prices, shares, and HHI.
4. Using a parametric bootstrap to estimate standard errors.
5. Estimating optimal instruments.
6. Constructing optimal micro moments.

4.5.1 Problem Results

As in the *fake cereal tutorial*, we'll first solve the fake cereal problem from *Nevo (2000a)*. We load the fake data and estimate the model as in the previous tutorial. We output the setup of the model to confirm we have correctly configured the *Problem*

```
product_data = pd.read_csv(pyblp.data.NEVO_PRODUCTS_LOCATION)
agent_data = pd.read_csv(pyblp.data.NEVO_AGENTS_LOCATION)
```

(continues on next page)

(continued from previous page)

```
product_formulations = (
   pyblp.Formulation('0 + prices', absorb='C(product_ids)'),
   pyblp.Formulation('1 + prices + sugar + mushy')
)
agent_formulation = pyblp.Formulation('0 + income + income_squared + age + child')
problem = pyblp.Problem(product_formulations, product_data, agent_formulation, agent_data)
problem
```

```
Dimensions:
================================================
 T    N     F    I     K1    K2    D    MD    ED
---  ----  ---  ----  ----  ----  ---  ----  ----
94   2256   5   1880   1     4     4    20    1
================================================

Formulations:
================================================================
       Column Indices:           0           1           2      3
-----------------------------  ------  --------------  -----  -----
 X1: Linear Characteristics    prices
X2: Nonlinear Characteristics    1         prices      sugar  mushy
       d: Demographics         income  income_squared   age   child
================================================================
```

We'll solve the problem in the same way as before. The *Problem.solve* method returns a *ProblemResults* class, which displays basic estimation results. The results that are displayed are simply formatted information extracted from various class attributes such as *ProblemResults.sigma* and *ProblemResults.sigma_se*.

```
initial_sigma = np.diag([0.3302, 2.4526, 0.0163, 0.2441])
initial_pi = [
  [ 5.4819,  0,      0.2037,  0     ],
  [15.8935, -1.2000, 0,       2.6342],
  [-0.2506,  0,      0.0511,  0     ],
  [ 1.2650,  0,     -0.8091,  0     ]
]
results = problem.solve(
    initial_sigma,
    initial_pi,
    optimization=pyblp.Optimization('bfgs', {'gtol': 1e-5}),
    method='1s'
)
results
```

```
Problem Results Summary:
=========================================================================================================
GMM   Objective  Gradient      Hessian         Hessian     Clipped  Weighting Matrix  Covariance Matrix
Step    Value      Norm    Min Eigenvalue  Max Eigenvalue  Shares   Condition Number  Condition Number
----  ---------  --------  --------------  --------------  -------  ----------------  -----------------
 1    +4.6E+00   +6.9E-06     +5.1E-05        +1.6E+04        0         +6.9E+07          +8.4E+08
=========================================================================================================

Cumulative Statistics:
=========================================================================
Computation  Optimizer  Optimization   Objective   Fixed Point  Contraction
   Time      Converged   Iterations   Evaluations  Iterations   Evaluations
-----------  ---------  ------------  -----------  -----------  -----------
 00:00:39       Yes           51           57          46379       143975
=========================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=============================================================================================================================
Sigma:      1         prices      sugar       mushy      |   Pi:      income    income_squared     age         child
------  ----------  ----------  ----------  ----------  |  ------  ----------  --------------  ----------  ----------
  1      +5.6E-01                                       |    1      +2.3E+00      +0.0E+00      +1.3E+00    +0.0E+00
        (+1.6E-01)                                      |          (+1.2E+00)                   (+6.3E-01)
                                                        |
prices   +0.0E+00    +3.3E+00                           |  prices   +5.9E+02      -3.0E+01      +0.0E+00    +1.1E+01
                    (+1.3E+00)                          |          (+2.7E+02)    (+1.4E+01)                (+4.1E+00)
                                                        |
sugar    +0.0E+00    +0.0E+00    -5.8E-03               |  sugar    -3.8E-01      +0.0E+00      +5.2E-02    +0.0E+00
                                (+1.4E-02)              |          (+1.2E-01)                   (+2.6E-02)
                                                        |
mushy    +0.0E+00    +0.0E+00    +0.0E+00    +9.3E-02   |  mushy    +7.5E-01      +0.0E+00      -1.4E+00    +0.0E+00
                                           (+1.9E-01)   |          (+8.0E-01)                   (+6.7E-01)
=============================================================================================================================

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
----------
 -6.3E+01
(+1.5E+01)
==========
```

Additional post-estimation outputs can be computed with *ProblemResults* methods.

4.5.2 Elasticities and Diversion Ratios

We can estimate elasticities, ε, and diversion ratios, $\mathscr{D}$, with *ProblemResults.compute_elasticities* and *ProblemResults.compute_diversion_ratios*.

As a reminder, elasticities in each market are

$$\varepsilon_{jk} = \frac{x_k}{s_j} \frac{\partial s_j}{\partial x_k}. \tag{4.16}$$

Diversion ratios are

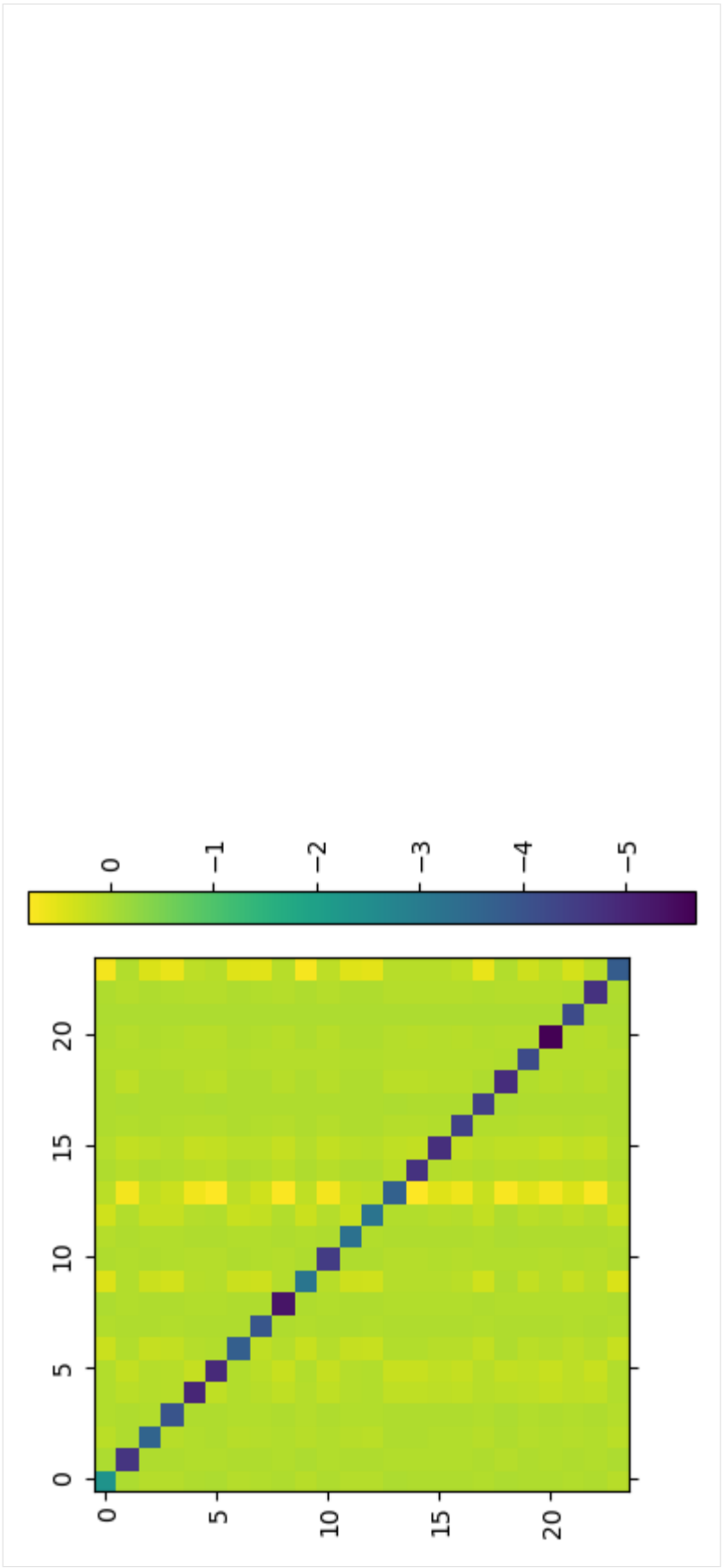
$$\mathscr{D}_{jk} = -\frac{\partial s_k}{\partial x_j} \Big/ \frac{\partial s_j}{\partial x_j}. \tag{4.17}$$

Following *Conlon and Mortimer (2021)*, we report the diversion to the outside good D_{j0} on the diagonal instead of $D_{jj} = -1$.

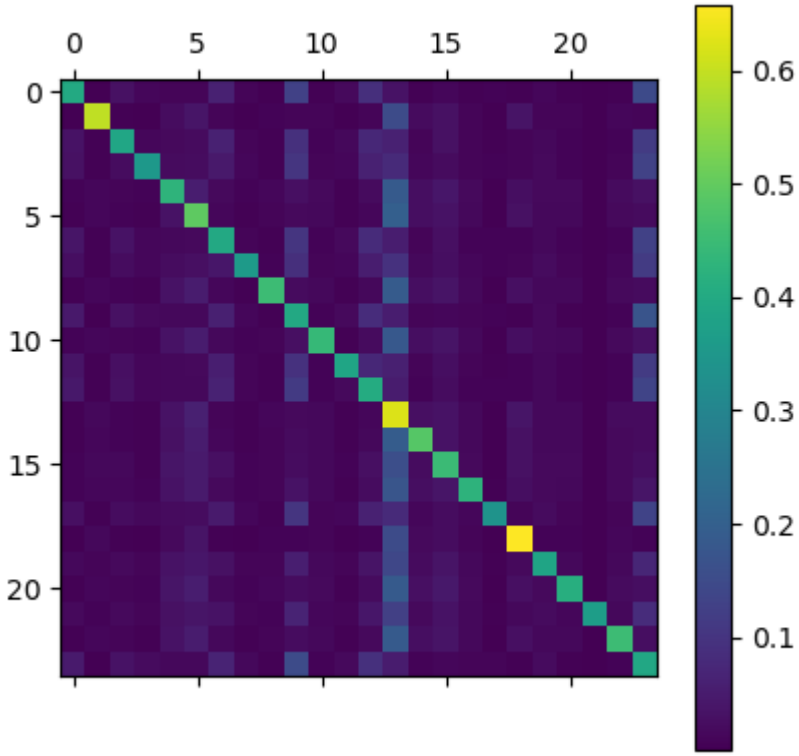
```
elasticities = results.compute_elasticities()
diversions = results.compute_diversion_ratios()
```

Post-estimation outputs are computed for each market and stacked. We'll use matplotlib functions to display the matrices associated with a single market.

```
single_market = product_data['market_ids'] == 'C01Q1'
plt.colorbar(plt.matshow(elasticities[single_market]));
```




```
plt.colorbar(plt.matshow(diversions[single_market]));
```



The diagonal of the first image consists of own elasticities and the diagonal of the second image consists of diversion ratios to the outside good. As one might expect, own price elasticities are large and negative while cross-price elasticities are positive but much smaller.

Elasticities and diversion ratios can be computed with respect to variables other than `prices` with the `name` argument of *ProblemResults.compute_elasticities* and *ProblemResults.compute_diversion_ratios*. Additionally, *ProblemResults.compute_long_run_diversion_ratios* can be used to used to understand substitution when products are eliminated from the choice set.

The convenience methods *ProblemResults.extract_diagonals* and *ProblemResults.extract_diagonal_means* can be used to extract information about own elasticities of demand from elasticity matrices.

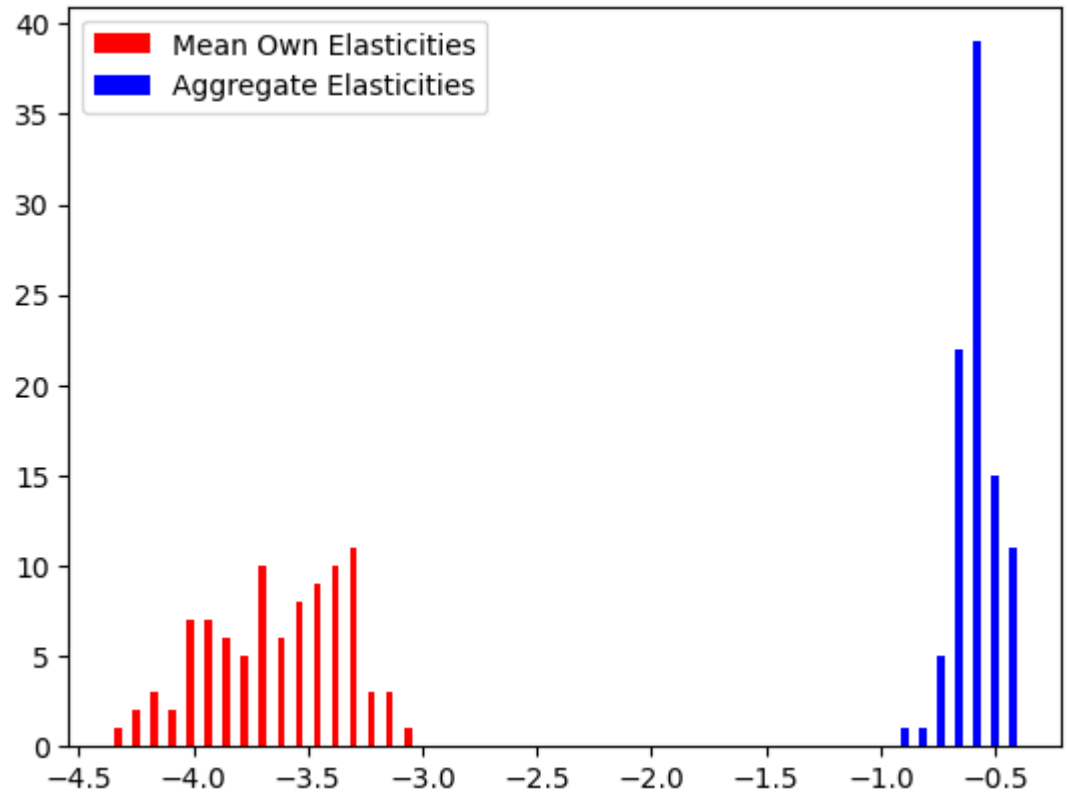
```
means = results.extract_diagonal_means(elasticities)
```

An alternative to summarizing full elasticity matrices is to use *ProblemResults.compute_aggregate_elasticities* to estimate aggregate elasticities of demand, E, in each market, which reflect the change in total sales under a proportional sales tax of some factor.

```
aggregates = results.compute_aggregate_elasticities(factor=0.1)
```

Since demand for an entire product category is generally less elastic than the average elasticity of individual products, mean own elasticities are generally larger in magnitude than aggregate elasticities.

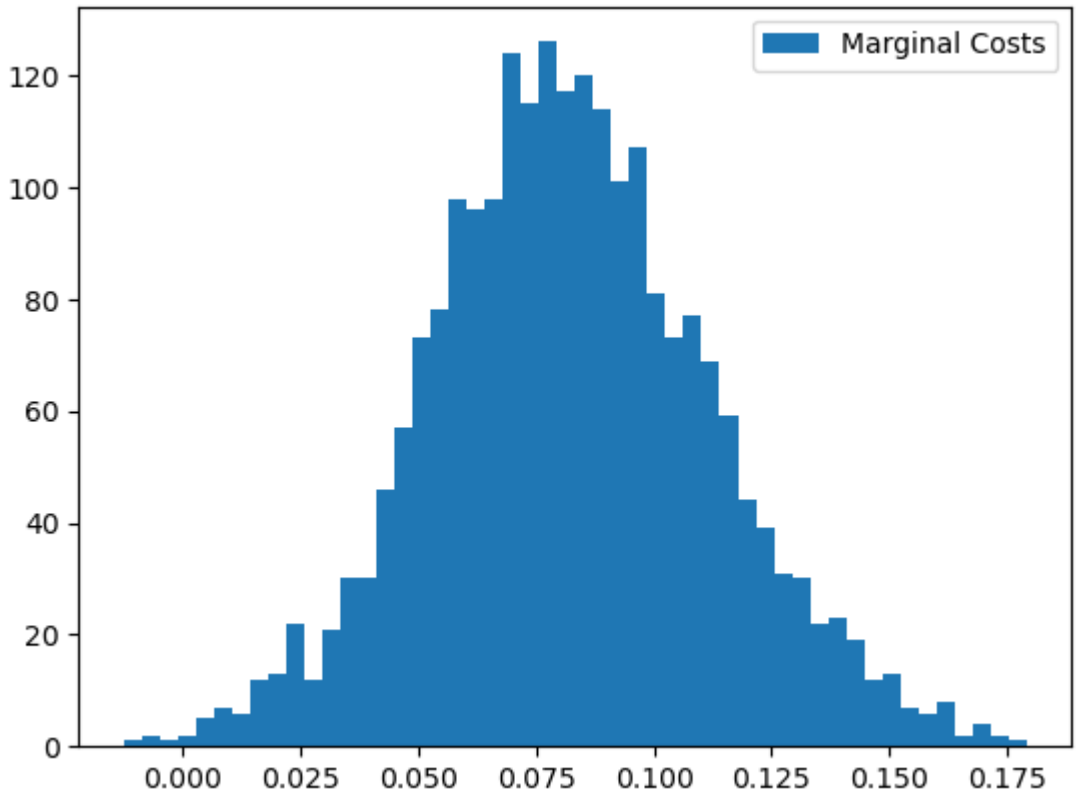
```
plt.hist(
    [means.flatten(), aggregates.flatten()],
    color=['red', 'blue'],
    bins=50
);
plt.legend(['Mean Own Elasticities', 'Aggregate Elasticities']);
```



4.5.3 Marginal Costs and Markups

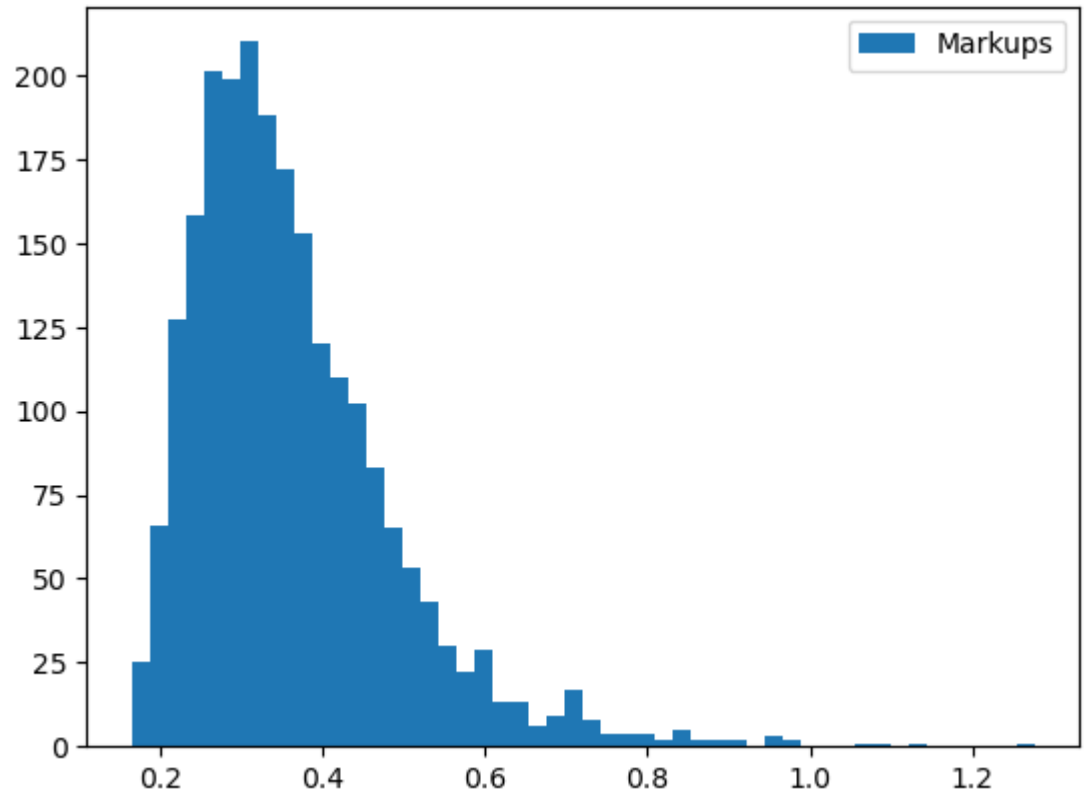
To compute marginal costs, c, the `product_data` passed to *Problem* must have had a `firm_ids` field. Since we included firm IDs when configuring the problem, we can use *ProblemResults.compute_costs*.

```
costs = results.compute_costs()
plt.hist(costs, bins=50);
plt.legend(["Marginal Costs"]);
```



Other methods that compute supply-side outputs often compute marginal costs themselves. For example, *ProblemResults.compute_markups* will compute marginal costs when estimating markups, $\mathscr{M}$, but computation can be sped up if we just use our pre-computed values.

```
markups = results.compute_markups(costs=costs)
plt.hist(markups, bins=50);
plt.legend(["Markups"]);
```



4.5.4 Mergers

Before computing post-merger outputs, we'll supplement our pre-merger markups with some other outputs. We'll compute Herfindahl-Hirschman Indices, HHI, with *ProblemResults.compute_hhi*; population-normalized gross expected profits, π, with *ProblemResults.compute_profits*; and population-normalized consumer surpluses, CS, with *ProblemResults.compute_consumer_surpluses*.

```
hhi = results.compute_hhi()
profits = results.compute_profits(costs=costs)
cs = results.compute_consumer_surpluses()
```

To compute post-merger outputs, we'll create a new set of firm IDs that represent a merger of firms `2` and `1`.

```
product_data['merger_ids'] = product_data['firm_ids'].replace(2, 1)
```

We can use *ProblemResults.compute_approximate_prices* or *ProblemResults.compute_prices* to estimate post-merger prices. The first method, which is in the spirit of early approaches to merger evaluation such as *Hausman, Leonard, and Zona (1994)* and *Werden (1997)*, is only a partial merger simulation in that it assumes shares and their price derivatives are unaffected by the merger.

The second method, which is used by *Nevo (2000b)*, is a full merger simulation in that it does not make these assumptions, and is the preferred approach to merger simulation. By default, we iterate over the ζ-markup equation from *Morrow and Skerlos (2011)* to solve the full system of J_t equations and J_t unknowns in each market t. We'll use the latter, since it is fast enough for this example problem.

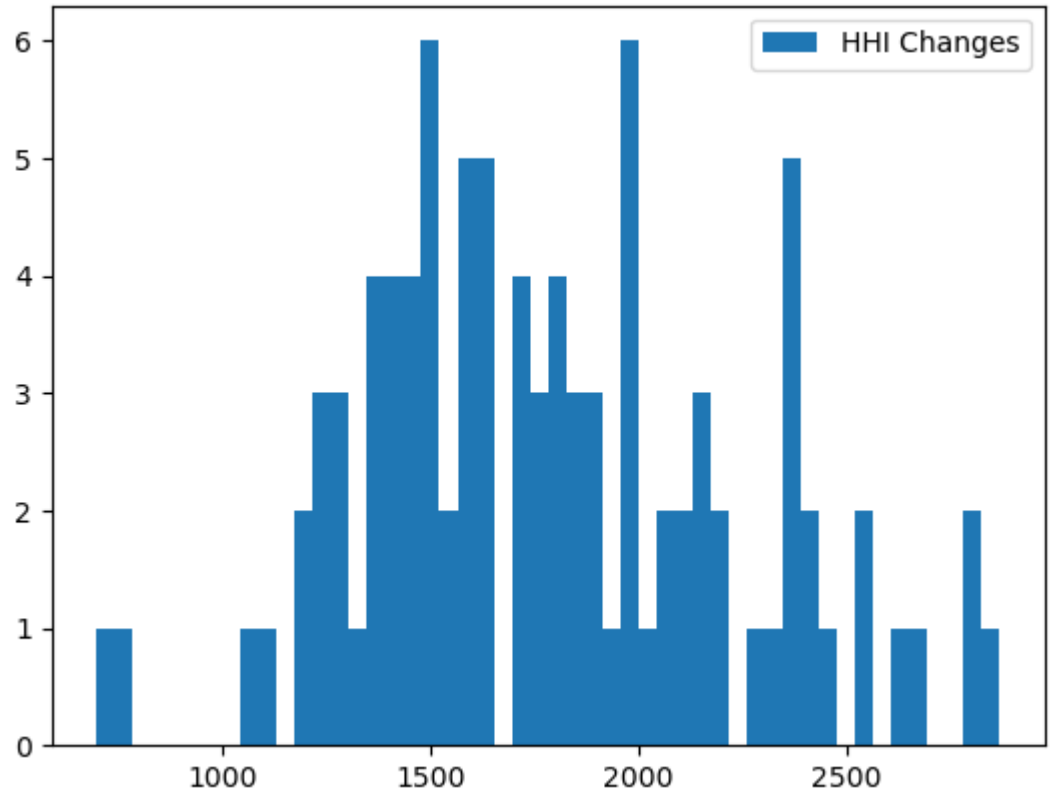
```
changed_prices = results.compute_prices(
    firm_ids=product_data['merger_ids'],
    costs=costs
)
```

We'll compute post-merger shares with *ProblemResults.compute_shares*.

```
changed_shares = results.compute_shares(changed_prices)
```

Post-merger prices and shares are used to compute other post-merger outputs. For example, HHI increases.

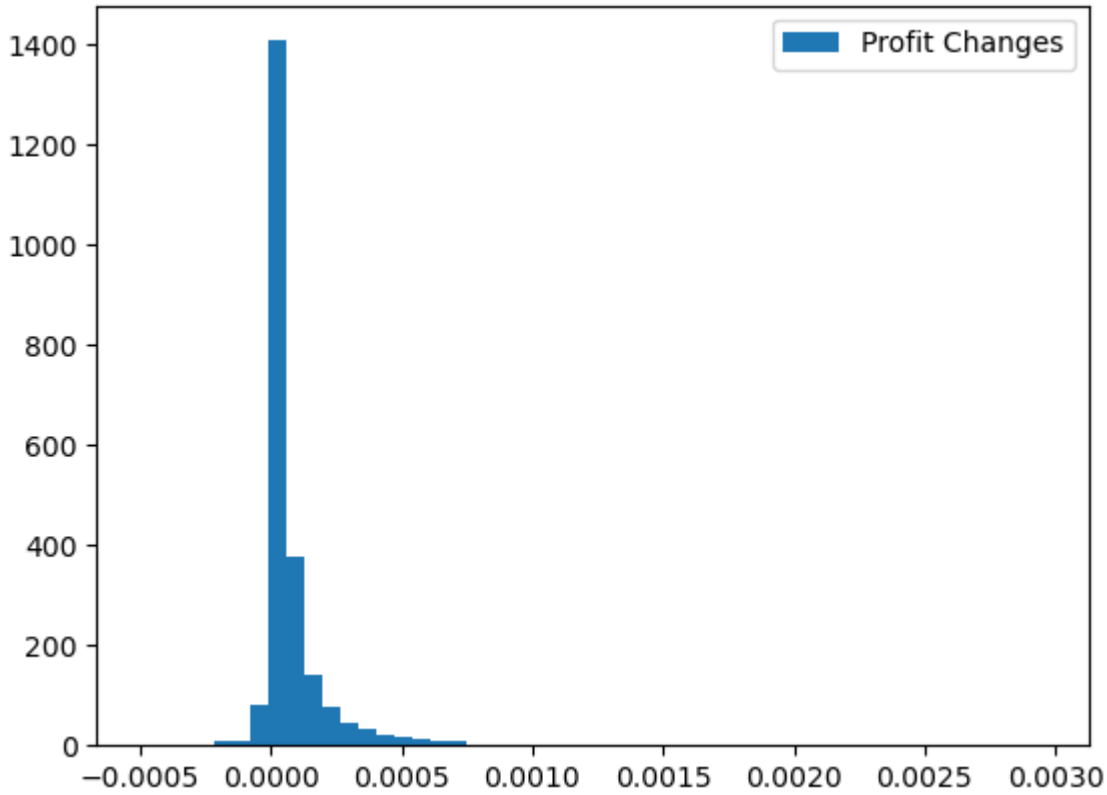
```
changed_hhi = results.compute_hhi(
    firm_ids=product_data['merger_ids'],
    shares=changed_shares
)
plt.hist(changed_hhi - hhi, bins=50);
plt.legend(["HHI Changes"]);
```



Markups, $\mathscr{M}$, and profits, π, generally increase as well.

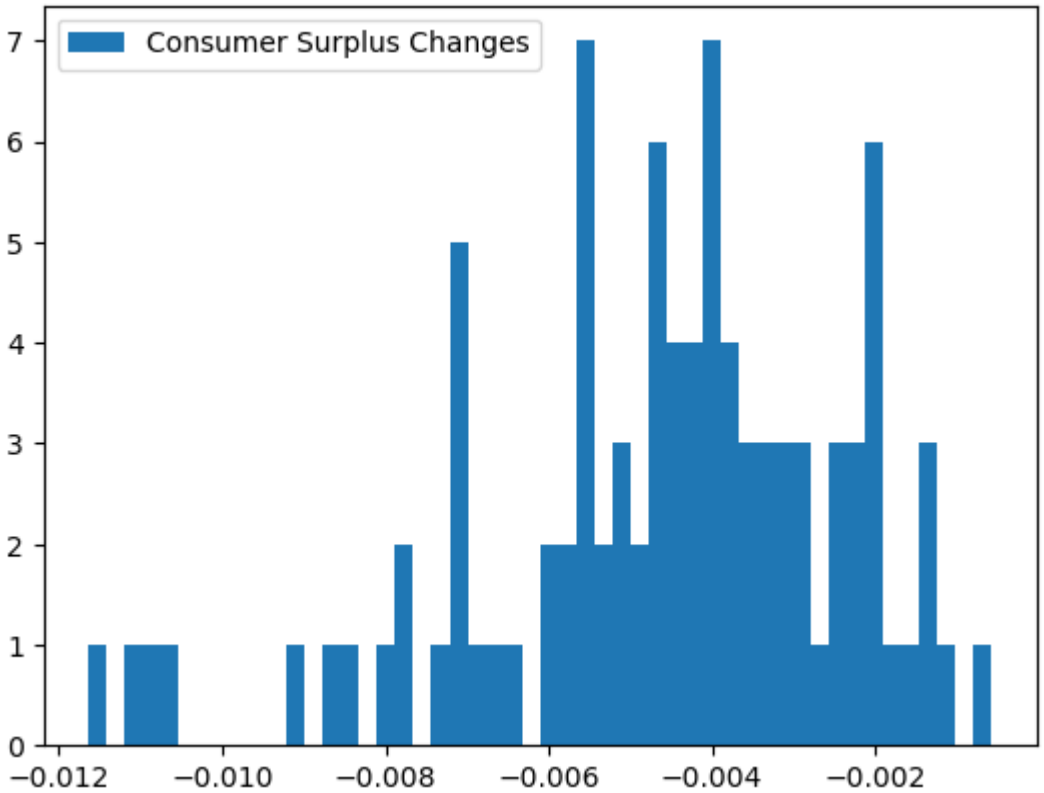
```
changed_markups = results.compute_markups(changed_prices, costs)
plt.hist(changed_markups - markups, bins=50);
plt.legend(["Markup Changes"]);
```

```
changed_profits = results.compute_profits(changed_prices, changed_shares, costs)
plt.hist(changed_profits - profits, bins=50);
plt.legend(["Profit Changes"]);
```



On the other hand, consumer surpluses, CS, generally decrease.

```
changed_cs = results.compute_consumer_surpluses(changed_prices)
plt.hist(changed_cs - cs, bins=50);
plt.legend(["Consumer Surplus Changes"]);
```



4.5.5 Bootstrapping Results

Post-estimation outputs can be informative, but they don't mean much without a sense sample-to-sample variability. One way to estimate confidence intervals for post-estimation outputs is with a standard bootstrap procedure:

1. Construct a large number of bootstrap samples by sampling with replacement from the original product data.
2. Initialize and solve a *Problem* for each bootstrap sample.
3. Compute the desired post-estimation output for each bootstrapped *ProblemResults* and from the resulting empirical distribution, construct boostrap confidence intervals.

Although appealing because of its simplicity, the computational resources required for this procedure are often prohibitively expensive. Furthermore, human oversight of the optimization routine is often required to determine whether the routine ran into any problems and if it successfully converged. Human oversight of estimation for each bootstrapped problem is usually not feasible.

A more reasonable alternative is a parametric bootstrap procedure:

1. Construct a large number of draws from the estimated joint distribution of parameters.
2. Compute the implied mean utility, δ, and shares, s, for each draw. If a supply side was estimated, also computed the implied marginal costs, c, and prices, p.
3. Compute the desired post-estimation output under each of these parametric bootstrap samples. Again, from the resulting empirical distribution, construct boostrap confidence intervals.

Compared to the standard bootstrap procedure, the parametric bootstrap requires far fewer computational resources, and is simple enough to not require human oversight of each bootstrap iteration. The primary complication to this procedure is that when supply is estimated, equilibrium prices and shares need to be computed for each parametric bootstrap sample by iterating over the ζ-markup equation from *Morrow and Skerlos (2011)*. Although nontrivial, this fixed point iteration problem is much less demanding than the full optimization routine required to solve the BLP problem from the start.

An empirical distribution of results computed according to this parametric bootstrap procedure can be created with the *ProblemResults.bootstrap* method, which returns a *BootstrappedResults* class that can be used just like *ProblemResults* to compute various post-estimation outputs. The difference is that *BootstrappedResults* methods return arrays with an extra first dimension, along which bootstrapped results are stacked.

We'll construct 90% parametric bootstrap confidence intervals for estimated mean own elasticities in each market of the fake cereal problem. Usually, bootstrapped confidence intervals should be based on thousands of draws, but we'll only use a few for the sake of speed in this example.

```
bootstrapped_results = results.bootstrap(draws=100, seed=0)
bootstrapped_results
```

```
Bootstrapped Results Summary:
=====================
Computation  Bootstrap
   Time        Draws
-----------  ---------
 00:00:03       100
=====================
```

```
bounds = np.percentile(
    bootstrapped_results.extract_diagonal_means(
        bootstrapped_results.compute_elasticities()
    ),
    q=[10, 90],
    axis=0
)
```

(continues on next page)

(continued from previous page)

```
table = pd.DataFrame(index=problem.unique_market_ids, data={
    'Lower Bound': bounds[0].flatten(),
    'Mean Own Elasticity': means.flatten(),
    'Upper Bound': bounds[1].flatten()
})
table.round(2).head()
```

```
       Lower Bound  Mean Own Elasticity  Upper Bound
C01Q1        -4.31                -4.21        -3.88
C01Q2        -4.07                -3.96        -3.68
C03Q1        -3.71                -3.40        -3.20
C03Q2        -3.65                -3.34        -3.16
C04Q1        -3.31                -3.15        -2.97
```

4.5.6 Optimal Instruments

Given a consistent estimate of θ, we may want to compute the optimal instruments of *Chamberlain (1987)* and use them to re-solve the problem. Optimal instruments have been shown, for example, by *Reynaert and Verboven (2014)*, to reduce bias, improve efficiency, and enhance stability of BLP estimates.

The *ProblemResults.compute_optimal_instruments* method computes the expected Jacobians that comprise the optimal instruments by integrating over the density of ξ (and ω if a supply side was estimated). By default, the method approximates this integral by averaging over the Jacobian realizations computed under draws from the asymptotic normal distribution of the error terms. Since this process is computationally expensive and often doesn't make much of a difference, we'll use `method='approximate'` in this example to simply evaluate the Jacobians at the expected value of ξ, zero.

```
instrument_results = results.compute_optimal_instruments(method='approximate')
instrument_results
```

```
Optimal Instrument Results Summary:
=======================
Computation  Error Term
   Time        Draws
-----------  ----------
 00:00:00        1
=======================
```

We can use the *OptimalInstrumentResults.to_problem* method to re-create the fake cereal problem with the estimated optimal excluded instruments.

```
updated_problem = instrument_results.to_problem()
updated_problem
```

```
Dimensions:
================================================
 T    N     F    I     K1    K2    D    MD    ED
---  ----  ---  ----  ----  ----  ---  ----  ----
94   2256   5   1880   1     4     4    14    1
================================================

Formulations:
======================================================================
       Column Indices:           0           1           2      3
-----------------------------  ------  --------------  -----  -----
 X1: Linear Characteristics    prices
X2: Nonlinear Characteristics    1         prices      sugar  mushy
       d: Demographics         income  income_squared   age   child
======================================================================
```

We can solve this updated problem just like the original one. We'll start at our consistent estimate of θ.

```
updated_results = updated_problem.solve(
    results.sigma,
    results.pi,
    optimization=pyblp.Optimization('bfgs', {'gtol': 1e-5}),
    method='1s'
)
updated_results
```

```
Problem Results Summary:
=========================================================================================================================
GMM   Objective  Gradient      Hessian         Hessian     Clipped  Weighting Matrix  Covariance Matrix
Step    Value      Norm    Min Eigenvalue  Max Eigenvalue  Shares   Condition Number  Condition Number
----  ---------  --------  --------------  --------------  -------  ----------------  ----------------
 1    +8.0E-14   +3.0E-06     +1.6E-04        +2.9E+04        0         +7.8E+07          +1.8E+08
=========================================================================================================================

Cumulative Statistics:
==========================================================================
Computation  Optimizer  Optimization   Objective   Fixed Point  Contraction
   Time      Converged   Iterations   Evaluations  Iterations   Evaluations
-----------  ---------  ------------  -----------  -----------  -----------
 00:00:38       Yes          42           50          45893       142156
==========================================================================
```

(continues on next page)

(continued from previous page)

```
Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=============================================================================================================
Sigma:      1         prices      sugar       mushy      |   Pi:      income    income_squared     age         child
------  ----------  ----------  ----------  ----------   |  ------  ----------  --------------  ----------  ----------
  1      +2.1E-01                                        |    1      +6.0E+00       +0.0E+00      +1.6E-01    +0.0E+00
        (+7.8E-02)                                       |           (+5.2E-01)                   (+2.0E-01)
                                                         |
prices   +0.0E+00    +3.0E+00                            |  prices   +9.8E+01       -5.6E+00      +0.0E+00    +4.1E+00
                    (+6.5E-01)                           |           (+8.6E+01)    (+4.5E+00)                 (+2.2E+00)
                                                         |
sugar    +0.0E+00    +0.0E+00    +2.7E-02                |  sugar    -3.1E-01       +0.0E+00      +4.9E-02    +0.0E+00
                                (+7.2E-03)               |           (+3.5E-02)                   (+1.3E-02)
                                                         |
mushy    +0.0E+00    +0.0E+00    +0.0E+00    +3.0E-01    |  mushy    +9.7E-01       +0.0E+00      -5.4E-01    +0.0E+00
                                            (+1.0E-01)   |           (+2.9E-01)                   (+1.8E-01)
=============================================================================================================

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
----------
 -3.1E+01
(+4.5E+00)
==========
```

4.5.7 Optimal Micro Moments

Similarly, if we have micro data that links individual choices to demographics, we can use all the information in this data by constructing optimal micro moments that match the score of the micro data, evaluated at our consistent estimate of θ. See the *micro moments tutorial* for an introduction to constructing non-optimal micro moments.

We don't have actual micro data for this empirical example, but we can simulate some just to demonstrate how to construct optimal micro moments. We'll use the *ProblemResults.simulate_micro_data* method to simulate 1,000 observations from a micro dataset at the estimated θ. Like most micro datasets, we'll define `compute_weights` such that we only have observations from consumer who purchase an inside good $j \neq 0$. For simplicity, we'll assume we only have micro data from a single market, `'C61Q1'`. Again, see the *micro moments tutorial* for a more in-depth discussion of *MicroDataset* and micro moments.

```
micro_dataset = pyblp.MicroDataset(
```

(continues on next page)

(continued from previous page)

```
    name="Simulated micro data",
    observations=1_000,
    compute_weights=lambda t, p, a: np.ones((a.size, 1 + p.size)),
    market_ids=['C61Q1'],
)
micro_data = results.simulate_micro_data(
    dataset=micro_dataset,
    seed=0,
)
```

The simulated micro data are a record array, which can be difficult to visualize. We'll convert it to a pandas dataframe, which is what we would usually load from an actual micro dataset file.

```
micro_data = pd.DataFrame(pyblp.data_to_dict(micro_data))
micro_data
```

```
    micro_ids market_ids  agent_indices  choice_indices
0           0      C61Q1             10              24
1           1      C61Q1             14               0
2           2      C61Q1             12               0
3           3      C61Q1             10              21
4           4      C61Q1              8              10
..        ...        ...            ...             ...
995       995      C61Q1              1              23
996       996      C61Q1             10               4
997       997      C61Q1             18               0
998       998      C61Q1              4               0
999       999      C61Q1             13               2

[1000 rows x 4 columns]
```

The simulated micro data contain four columns:

- `micro_ids`: This is simply an index from 0 to 999, indexing each micro observation.
- `market_ids`: This is the market of each observation. We configured our *MicroDataset*to only sample from one market.
- `agent_indices`: This is the within-market index (from 0 to $I_t - 1$ in market t) of the agent data row that was sampled with probability w_{it}, configured by the `weights` column in agent data.
- `choice_indices`: This is the within-market index (from 0 to $J_t - 1$ in market t) of the produt data row that was sampled with probability s_{ijt} evaluated at the consistent estimate of θ.

The simulated data contain a bit more information than we would usually have in actual micro data. The `agent_indices` contain information not only about observed demographics of the micro observation, but also about unobserved preferences. We will merge in only observed demographics for each agent index to get a more realistic simulated micro dataset.

```
agent_data['agent_indices'] = agent_data.groupby('market_ids').cumcount()
micro_data = micro_data.merge(
    agent_data[['market_ids', 'agent_indices', 'income', 'income_squared', 'age', 'child']],
    on=['market_ids', 'agent_indices']
)
del micro_data['agent_indices']
micro_data
```

```
    micro_ids market_ids  choice_indices    income  income_squared       age  \
0           0      C61Q1              24  1.212283       22.546328  0.624306
1           1      C61Q1               0  0.020036       -0.519111 -0.335470
2           2      C61Q1               0 -1.106004      -19.693212 -1.839547
3           3      C61Q1              21  1.212283       22.546328  0.624306
4           4      C61Q1              10  0.563580        9.643785  0.357678
..        ...        ...             ...       ...             ...       ...
995       995      C61Q1              23 -0.417249       -8.266351  1.273968
996       996      C61Q1               4  1.212283       22.546328  0.624306
997       997      C61Q1               0 -1.106004      -19.693212 -0.006965
998       998      C61Q1               0 -1.185986      -20.958686  0.851696
999       999      C61Q1               2 -0.239006       -5.154642 -1.616403

        child
0   -0.230851
1   -0.230851
2    0.769149
3   -0.230851
4   -0.230851
..        ...
995 -0.230851
996 -0.230851
997 -0.230851
998 -0.230851
999  0.769149

[1000 rows x 7 columns]
```

This is more like real micro data that we would actually load from a file. For each observation, we know the market, choice, and demographics of the consumer.

To compute optimal micro moments at our consistent estimate of θ, we need to compute two types of scores:

1. The score for each observation in our micro data via *ProblemResults.compute_micro_scores*.
2. The score for each possible agent-choice in the model via *ProblemResults.compute_agent_scores*.

For each, we will integrate over unobserved heterogeneity with quadrature, but one could use Monte Carlo methods too. We will do so by just passing an *Integration* configuration to these two methods, but we could also do so by duplicating each row of micro data by as many integration nodes/weights we wanted for each, adding `weights` and `nodes` columns as with agent data.

```
score_integration = pyblp.Integration('product', 5)
micro_scores = results.compute_micro_scores(micro_dataset, micro_data, integration=score_integration)
agent_scores = results.compute_agent_scores(micro_dataset, integration=score_integration)
```

Both `micro_scores` and `agent_scores` are lists, with one element for each nonlinear parameter in θ. The ordering is

```
results.theta_labels
```

```
[np.str_('1 x 1'),
 np.str_('prices x prices'),
 np.str_('sugar x sugar'),
 np.str_('mushy x mushy'),
 np.str_('1 x income'),
 np.str_('1 x age'),
 np.str_('prices x income'),
 np.str_('prices x income_squared'),
 np.str_('prices x child'),
 np.str_('sugar x income'),
 np.str_('sugar x age'),
 np.str_('mushy x income'),
 np.str_('mushy x age')]
```

The first element of `micro_scores` corresponds to the σ on the constant term (i.e., the `'1 x 1'` above). It has the estimated score for each observation in `micro_data`:

```
micro_scores[0].shape
```

```
(1000,)
```

The first element of `agent_scores` also correspondds to the σ on the constant term. It is a mapping from market IDs to arrays of scores for each of the $I_t \times J_t$ possible agent-choices in that market.

```
agent_scores[0]['C61Q1'].shape
```

```
(20, 25)
```

We will construct one optimal *MicroMoment* for each parameter, matching the average score (via a *MicroPart*) for that parameter from the micro data with its model counterpart.

```
optimal_micro_moments = []
for m, (micro_scores_m, agent_scores_m) in enumerate(zip(micro_scores, agent_scores)):
    optimal_micro_moments.append(pyblp.MicroMoment(
        name=f"Score for parameter #{m}",
        value=micro_scores_m.mean(),
        parts=pyblp.MicroPart(
            name=f"Score for parameter #{m}",
            dataset=micro_dataset,
            compute_values=lambda t, p, a, v=agent_scores_m: v[t],
        ),
    ))
```

For example, some information about the optimal micro moment for the first parameter is as follows.

```
optimal_micro_moments[0]
```

```
Score for parameter #0: -2.8E-02 (Score for parameter #0 on Simulated micro data: 1000 Observations in Market 'C61Q1')
```

Now, we can use our problem with updated optimal IVs, including our optimal micro moments, to obtain an efficient estimator.

```
updated_results = updated_problem.solve(
    results.sigma,
    results.pi,
    optimization=pyblp.Optimization('bfgs', {'gtol': 1e-5}),
    method='1s',
    micro_moments=optimal_micro_moments,
)
updated_results
```

```
Problem Results Summary:
=========================================================================================================
GMM   Objective  Gradient      Hessian         Hessian     Clipped  Weighting Matrix  Covariance Matrix
Step    Value      Norm    Min Eigenvalue  Max Eigenvalue  Shares   Condition Number  Condition Number
----  ---------  --------  --------------  --------------  -------  ----------------  ----------------
 1    +1.1E+02   +3.8E-06     +2.1E-03        +8.3E+04        0         +2.2E+08          +4.1E+07
=========================================================================================================

Cumulative Statistics:
```

(continues on next page)

(continued from previous page)

```
==========================================================================
Computation  Optimizer  Optimization   Objective   Fixed Point  Contraction
   Time      Converged   Iterations   Evaluations  Iterations   Evaluations
-----------  ---------  ------------  -----------  -----------  -----------
 00:00:39       Yes           34           42          33341       103798
==========================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
==========================================================================================================================
Sigma:      1          prices      sugar       mushy      |   Pi:      income    income_squared     age         child
------  ----------  ----------  ----------  ----------  |  ------  ----------  --------------  ----------  ----------
  1      +4.6E-01                                       |    1      +4.8E+00      +0.0E+00      +6.5E-01    +0.0E+00
        (+6.7E-02)                                      |           (+3.5E-01)                  (+1.5E-01)
                                                        |
prices   +0.0E+00    +3.1E+00                           |  prices   +4.7E+02      -2.5E+01      +0.0E+00    +3.2E+00
                    (+5.3E-01)                          |           (+3.1E+01)    (+1.6E+00)                (+1.7E+00)
                                                        |
sugar    +0.0E+00    +0.0E+00    +2.0E-02               |  sugar    -4.0E-01      +0.0E+00      +4.6E-02    +0.0E+00
                                (+6.1E-03)              |           (+2.4E-02)                  (+9.2E-03)
                                                        |
mushy    +0.0E+00    +0.0E+00    +0.0E+00    -6.6E-02   |  mushy    +1.1E+00      +0.0E+00      -1.1E+00    +0.0E+00
                                            (+8.3E-02)  |           (+1.9E-01)                  (+1.1E-01)
==========================================================================================================================

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
----------
 -5.2E+01
(+2.2E+00)
==========

Estimated Micro Moments:
=============================================================================================================================
Observed  Estimated  Difference          Moment                    Part                  Dataset          Observations
↪Markets
--------  ---------  ----------  ----------------------  ----------------------  --------------------  ------------  -
↪------
-2.8E-02  -2.3E-02    -5.5E-03   Score for parameter #0   Score for parameter #0   Simulated micro data      1000
↪  1
```

(continues on next page)

(continued from previous page)

```
+6.4E-04  +9.8E-04    -3.4E-04   Score for parameter #1   Score for parameter #1   Simulated micro data      1000
↪  1
+1.1E-01  +1.5E-01    -3.4E-02   Score for parameter #2   Score for parameter #2   Simulated micro data      1000
↪  1
-2.9E-04  -3.6E-04    +7.0E-05   Score for parameter #3   Score for parameter #3   Simulated micro data      1000
↪  1
-2.0E-03  +2.1E-02    -2.3E-02   Score for parameter #4   Score for parameter #4   Simulated micro data      1000
↪  1
-3.6E-02  -3.0E-02    -6.3E-03   Score for parameter #5   Score for parameter #5   Simulated micro data      1000
↪  1
-2.5E-04  +2.1E-03    -2.4E-03   Score for parameter #6   Score for parameter #6   Simulated micro data      1000
↪  1
-6.8E-03  +4.0E-02    -4.7E-02   Score for parameter #7   Score for parameter #7   Simulated micro data      1000
↪  1
+1.7E-03  +1.0E-03    +6.8E-04   Score for parameter #8   Score for parameter #8   Simulated micro data      1000
↪  1
+5.5E-02  +1.8E-01    -1.3E-01   Score for parameter #9   Score for parameter #9   Simulated micro data      1000
↪  1
-4.8E-01  -4.7E-01    -1.0E-02   Score for parameter #10  Score for parameter #10  Simulated micro data      1000
↪  1
-2.9E-03  +8.1E-03    -1.1E-02   Score for parameter #11  Score for parameter #11  Simulated micro data      1000
↪  1
-2.6E-02  -5.5E-03    -2.0E-02   Score for parameter #12  Score for parameter #12  Simulated micro data      1000
↪  1
==========================================================================================================================
```

Results are fairly similar to before because we simulated the micro data from our first-stage estimate of θ, which was somewhat close to our second-stage estimate. Scores are not matched perfectly because the model is now over-identified, with two times as many moments as there are parameters (one optimal IV and one optimal micro moment for each nonlinear parameter).

The online version of the following section may be easier to read.

4.6 Problem Simulation Tutorial

```
import pyblp
import numpy as np
import pandas as pd

pyblp.options.digits = 2
pyblp.options.verbose = False
pyblp.__version__
```

```
'1.2.0'
```

Before configuring and solving a problem with real data, it may be a good idea to perform Monte Carlo analysis on simulated data to verify that it is possible to accurately estimate model parameters. For example, before configuring and solving the example problems in the prior tutorials, it may have been a good idea to simulate data according to the assumed models of supply and demand. During such Monte Carlo anaysis, the data would only be used to determine sample sizes and perhaps to choose reasonable true parameters.

Simulations are configured with the *Simulation* class, which requires many of the same inputs as *Problem*. The two main differences are:

1. Variables in formulations that cannot be loaded from `product_data` or `agent_data` will be drawn from independent uniform distributions.
2. True parameters and the distribution of product unobservables are specified.

First, we'll use *build_id_data* to build market and firm IDs for a model in which there are $T = 50$ markets, and in each market t, a total of $J_t = 20$ products produced by $F = 10$ firms.

```
id_data = pyblp.build_id_data(T=50, J=20, F=10)
```

Next, we'll create an *Integration* configuration to build agent data according to a Gauss-Hermite product rule that exactly integrates polynomials of degree $2 \times 9 - 1 = 17$ or less.

```
integration = pyblp.Integration('product', 9)
integration
```

```
Configured to construct nodes and weights according to the level-9 Gauss-Hermite product rule with options {}.
```

We'll then pass these data to *Simulation*. We'll use *Formulation* configurations to create an X_1 that consists of a constant, prices, and an exogenous characteristic; an X_2 that consists only of the same exogenous characteristic; and an X_3 that consists of the common exogenous characteristic and a cost-shifter.

```
simulation = pyblp.Simulation(
   product_formulations=(
       pyblp.Formulation('1 + prices + x'),
       pyblp.Formulation('0 + x'),
       pyblp.Formulation('0 + x + z')
   ),
   beta=[1, -2, 2],
   sigma=1,
   gamma=[1, 4],
   product_data=id_data,
   integration=integration,
   seed=0
)
simulation
```

```
Dimensions:
====================================
 T    N     F    I    K1    K2    K3
---  ----  ---  ---  ----  ----  ----
50   1000  10   450   3     1     2
====================================

Formulations:
================================================
        Column Indices:           0     1      2
------------------------------  ---  ------  ---
  X1: Linear Characteristics      1   prices   x
 X2: Nonlinear Characteristics    x
X3: Linear Cost Characteristics   x     z
================================================

Nonlinear Coefficient True Values:
================
Sigma:     x
------  --------
  x     +1.0E+00
================

Beta True Values:
============================
   1       prices      x
```

(continues on next page)

(continued from previous page)

```
--------  --------  --------
+1.0E+00  -2.0E+00  +2.0E+00
============================

Gamma True Values:
==================
   x         z
--------  --------
+1.0E+00  +4.0E+00
==================
```

When *Simulation* is initialized, it constructs *Simulation.agent_data* and simulates *Simulation.product_data*.

The *Simulation* can be further configured with other arguments that determine how product unobservables are simulated and how marginal costs are specified.

At this stage, simulated variables are not consistent with true parameters, so we still need to solve the simulation with *Simulation.replace_endogenous*. This method replaced simulated prices and market shares with values that are consistent with the true parameters. Just like *ProblemResults.compute_prices*, to do so it iterates over the ζ-markup equation from *Morrow and Skerlos (2011)*.

```
simulation_results = simulation.replace_endogenous()
simulation_results
```

```
Simulation Results Summary:
=============================================================================================================
Computation  Fixed Point  Fixed Point  Contraction  Profit Gradients  Profit Hessians  Profit Hessians
   Time       Failures    Iterations   Evaluations      Max Norm      Min Eigenvalue   Max Eigenvalue
-----------  -----------  -----------  -----------  ----------------  ---------------  ---------------
 00:00:00         0           721          721          +1.3E-13         -8.4E-01         -9.6E-06
=============================================================================================================
```

Now, we can try to recover the true parameters by creating and solving a *Problem*.

The convenience method *SimulationResults.to_problem* constructs some basic "sums of characteristics" BLP instruments that are functions of all exogenous numerical variables in the problem. In this example, excluded demand-side instruments are the cost-shifter `z` and traditional BLP instruments constructed from `x`. Excluded supply-side instruments are traditional BLP instruments constructed from `x` and `z`.

```
problem = simulation_results.to_problem()
problem
```

```
Dimensions:
================================================
 T    N     F    I    K1    K2    K3    MD    MS
```

(continues on next page)

(continued from previous page)

```
---  ----  ---  ---  ----  ----  ----  ----  ----
50   1000  10   450   3     1     2     5     6
=================================================

Formulations:
=================================================
        Column Indices:           0     1      2
------------------------------  ---  ------  ---
  X1: Linear Characteristics     1   prices   x
 X2: Nonlinear Characteristics   x
X3: Linear Cost Characteristics  x     z
=================================================
```

We'll choose starting values that are half the true parameters so that the optimization routine has to do some work. Note that since we're jointly estimating the supply side, we need to provide an initial value for the linear coefficient on prices because this parameter cannot be concentrated out of the problem (unlike linear coefficients on exogenous characteristics).

```
results = problem.solve(
    sigma=0.5 * simulation.sigma,
    pi=0.5 * simulation.pi,
    beta=[None, 0.5 * simulation.beta[1, 0], None],
    optimization=pyblp.Optimization('l-bfgs-b', {'gtol': 1e-5})
)
results
```

```
Problem Results Summary:
=========================================================================================================================
GMM   Objective    Projected    Reduced Hessian  Reduced Hessian  Clipped  Weighting Matrix  Covariance Matrix
Step    Value    Gradient Norm  Min Eigenvalue   Max Eigenvalue   Shares   Condition Number  Condition Number
----  ---------  -------------  ---------------  ---------------  -------  ----------------  ----------------
 2    +6.4E+00     +6.9E-08        +7.2E+00         +3.8E+03         0         +3.7E+04          +1.5E+04
=========================================================================================================================

Cumulative Statistics:
=============================================================================
Computation  Optimizer  Optimization   Objective   Fixed Point  Contraction
   Time      Converged   Iterations   Evaluations  Iterations   Evaluations
-----------  ---------  ------------  -----------  -----------  -----------
 00:00:05       Yes          23           30          8291         26303
=============================================================================
```

(continues on next page)

(continued from previous page)

```
Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
==================
Sigma:      x
------  ----------
  x      +7.8E-01
        (+5.2E-01)
==================

Beta Estimates (Robust SEs in Parentheses):
===================================
    1         prices        x
----------  ----------  ----------
 +9.6E-01    -2.0E+00    +2.1E+00
(+9.3E-02)  (+2.4E-02)  (+1.4E-01)
===================================

Gamma Estimates (Robust SEs in Parentheses):
======================
    x           z
----------  ----------
 +9.8E-01    +4.0E+00
(+8.7E-02)  (+8.5E-02)
======================
```

The parameters seem to have been estimated reasonably well.

```
np.c_[simulation.beta, results.beta]
```

```
array([[ 1.        ,  0.96223514],
       [-2.        , -2.00792431],
       [ 2.        ,  2.10032015]])
```

```
np.c_[simulation.gamma, results.gamma]
```

```
array([[1.        , 0.97820624],
       [4.        , 4.03121577]])
```

```
np.c_[simulation.sigma, results.sigma]
```

```
array([[1.        , 0.78358853]])
```

In addition to checking that the configuration for a model based on actual data makes sense, the *Simulation* class can also be a helpful tool for better understanding under what general conditions BLP models can be accurately estimated. Simulations are also used extensively in pyblp's test suite.

CHAPTER

FIVE

API DOCUMENTATION

The majority of the package consists of classes, which compartmentalize different aspects of the BLP model. There are some convenience functions as well.

5.1 Configuration Classes

Various components of the package require configurations for how to approximate integrals, solve fixed point problems, and solve optimimzation problems. Such configurations are specified with the following classes.

Formulation(formula[, absorb, ...])	Configuration for designing matrices and absorbing fixed effects.
Integration(specification, size[, ...])	Configuration for building integration nodes and weights.
Iteration(method[, method_options, ...])	Configuration for solving fixed point problems.
Optimization(method[, method_options, ...])	Configuration for solving optimization problems.

5.1.1 pyblp.Formulation

class pyblp.**Formulation**(*formula*, *absorb=None*, *absorb_method=None*, *absorb_options=None*)

Configuration for designing matrices and absorbing fixed effects.

Internally, the patsy package is used to convert data and R-style formulas into matrices. All of the standard binary operators can be used to design complex matrices of factor interactions:

- `+` - Set union of terms.
- `-` - Set difference of terms.
- `*` - Short-hand. The formula `a * b` is the same as `a + b + a:b`.
- `/` - Short-hand. The formula `a / b` is the same as `a + a:b`.
- `:` - Interactions between two sets of terms.
- `**` - Interactions up to an integer degree.

However, since factors need to be differentiated (for example, when computing elasticities), only the most essential functions are supported:

- `C` - Mark a variable as categorical. See `patsy.builtins.C()`. Arguments are not supported.
- `I` - Encapsulate mathematical operations. See `patsy.builtins.I()`.
- `log` - Natural logarithm function.

- `exp` - Natural exponential function.

Data associated with variables should generally already be transformed. However, when encapsulated by `I()`, these operators function like normal mathematical operators on numeric variables: `+` adds, `-` subtracts, `*` multiplies, `/` divides, and `**` exponentiates.

Internally, mathematical operations are parsed and evaluated by the SymPy package, which is also used to symbolically differentiate terms when derivatives are needed.

Parameters

- **formula** (*str*) – R-style formula used to design a matrix. Variable names will be validated when this formulation and data are passed to a function that uses them. By default, an intercept is included, which can be removed with `0` or `-1`. If `absorb` is specified, intercepts are ignored.
- **absorb** (*str, optional*) – R-style formula used to design a matrix of categorical variables representing fixed effects, which will be absorbed into the matrix designed by `formula` by the PyHDFE package. Fixed effect absorption is only supported for some matrices. Unlike `formula`, intercepts are ignored. Only categorical variables are supported.
- **absorb_method** (*str, optional*) – Method by which fixed effects will be absorbed. For a full list of supported methods, refer to the `residualize_method` argument of `pyhdfe.create()`.

  By default, the simplest methods are used: simple de-meaning for a single fixed effect and simple iterative de-meaning by way of the method of alternating projections (MAP) for multiple dimensions of fixed effects. For multiple dimensions, non-accelerated MAP is unlikely to be the fastest algorithm. If fixed effect absorption seems to be taking a long time, consider using a different method such as `'lsmr'`, using `absorb_options` to specify a MAP acceleration method, or configuring other options such as termination tolerances.
- **absorb_options** (*dict, optional*) – Configuration options for the chosen `method`, which will be passed to the `options` argument of `pyhdfe.create()`.

Examples

The online version of the following section may be easier to read.

Formulation Example

```
import pyblp

pyblp.__version__
```

```
'1.2.0'
```

In this example, we'll design a matrix without an intercept, but with both prices and another numeric size variable.

```
formulation = pyblp.Formulation('0 + prices + size')
formulation
```

```
prices + size
```

Next, we'll design a second matrix with an intercept, with first- and second-degree size terms, with categorical product IDs and years, and with the interaction of the last two. The first formulation will include the fixed effects as indicator variables, and the second will absorb them.

```
formulation1 = pyblp.Formulation('size + I(size ** 2) + C(product) * C(year)')
formulation1
```

```
1 + size + I(size ** 2) + C(product) + C(year) + C(product):C(year)
```

```
formulation2 = pyblp.Formulation('size + I(size ** 2)', absorb='C(product) * C(year)')
formulation2
```

```
size + I(size ** 2) + Absorb[C(product)] + Absorb[C(year)] + Absorb[C(product):C(year)]
```

Finally, we'll design a third matrix with an intercept and with a yearly trend interacted with the natural logarithm of income and categorical education. Absorption of continuous variables is not supported, so we need to use dummy variables.

```
formulation = pyblp.Formulation('year:(log(income) + C(education))')
formulation
```

```
1 + year:log(income) + year:C(education)
```

5.1.2 pyblp.Integration

class pyblp.**Integration**(*specification*, *size*, *specification_options=None*)

Configuration for building integration nodes and weights.

Parameters

- **specification** (*str*) – How to build nodes and weights. One of the following:
  - `'monte_carlo'` - Draw from a pseudo-random standard multivariate normal distribution. Integration weights are `1 / size`. The `seed` field of `options` can be used to seed the random number generator.
  - `'halton'` - Generate nodes according to the Halton. A different prime (starting with 2, 3, 5, etc.) is used for each dimension of integration. To eliminate correlation between dimensions, the first `1000` values are by default discarded in each dimension. To further improve performance (particularly in settings with many dimensions), sequences are also by default scrambled with the algorithm of *Owen (2017)*. The `discard`, `scramble`, and `seed` fields of `options` can be used to configure these default settings.
  - `'lhs'` - Generate nodes according to Latin Hypercube Sampling (LHS). Integration weights are `1 / size`. The `seed` field of `options` can be used to seed the random number generator.
  - `'mlhs'` - Generate nodes according to Modified Latin Hypercube Sampling (MLHS) described by *Hess, Train, and Polak (2004)*. Integration weights are `1 / size`. The `seed` field of `options` can be used to seed the random number generator.
  - `'product'` - Generate nodes and weights according to the level-`size` Gauss-Hermite product rule.
  - `'nested_product'` - Generate nodes and weights according to the level-`size` nested Gauss-Hermite product rule. Weights can be negative.
  - `'grid'` - Generate a sparse grid of nodes and weights according to the level-`size` Gauss-Hermite quadrature rule. Weights can be negative.
  - `'nested_grid'` - Generate a sparse grid of nodes and weights according to the level `size` nested Gauss-Hermite quadrature rule. Weights can be negative.

  Best practice for low dimensions is probably to use `'product'` to a relatively high degree of polynomial accuracy. In higher dimensions, `'grid'` or `'halton'` appears to scale the best. For more information, see *Judd and Skrainka (2011)* and *Conlon and Gortmaker (2020)*.

  Sparse grids are constructed in analogously to the Matlab function nwspgr created by Florian Heiss and Viktor Winschel. For more information, see *Heiss and Winschel (2008)*.
- **size** (*int*) – The number of draws if `specification` is `'monte_carlo'`, `'halton'`, `'lhs'`, or `'mlhs'`, and the level of the quadrature rule otherwise.
- **specification_options** (*dict, optional*) – Options for the integration specification. The `'monte_carlo'`, `'halton'`, `'lhs'`, and `'mlhs'` specifications support the following option:
  - **seed** : (*int*) - Passed to `numpy.random.RandomState` to seed the random number generator before building integration nodes. By default, a seed is not passed to the random number generator. For `'halton'` draws, this is only relevant if `scramble` is `True` (which is the default).

  The `'halton'` specification supports the following options:

- **discard** : (*int*) - How many values at the beginning of each dimension's Halton sequence to discard. Discarding values at the start of each dimension's sequence is the simplest way to eliminate correlation between dimensions. By default, the first `1000` values in each dimension are discarded.
- **scramble** : (*bool*) - Whether to scramble the sequences with the algorithm of *Owen (2017)*. By default, sequences are scrambled.

Examples

The online version of the following section may be easier to read.

Integration Example

```
import pyblp

pyblp.__version__
```

```
'1.2.0'
```

In this example, we'll build a Monte Carlo configuration with 1,000 draws for each market and a fixed seed.

```
integration = pyblp.Integration('monte_carlo', size=1000, specification_options={'seed': 0})
integration
```

```
Configured to construct nodes and weights with Monte Carlo simulation with options {seed: 0}.
```

Depending on the dimension of the integration problem, a level six sparse grid configuration may have a similar number of nodes. However, even if there are fewer nodes, it is likely to perform better in the BLP problem. Sparse grid construction is deterministic, so a seed is not needed to fix the grid every time we use this configuration.

```
integration = pyblp.Integration('grid', size=7)
integration
```

```
Configured to construct nodes and weights in a sparse grid according to the level-7 Gauss-Hermite rule with options {}.
```

5.1.3 pyblp.Iteration

class pyblp.**Iteration**(*method*, *method_options=None*, *compute_jacobian=False*, *universal_display=False*)

Configuration for solving fixed point problems.

Parameters

- **method** (*str or callable*) – The fixed point iteration routine that will be used. The following routines do not use analytic Jacobians:
  - `'simple'` - Non-accelerated iteration.
  - `'squarem'` - SQUAREM acceleration method of *Varadhan and Roland (2008)* and considered in the context of the BLP problem in *Reynaerts, Varadhan, and Nash (2012)*. This implementation uses a first-order squared non-monotone extrapolation scheme.
  - `'broyden1'` - Use the `scipy.optimize.root()` Broyden's first Jacobian approximation method, known as Broyden's good method.
  - `'broyden2'` - Use the `scipy.optimize.root()` Broyden's second Jacobian approximation method, known as Broyden's bad method.
  - `'anderson'` - Use the `scipy.optimize.root()` Anderson method.
  - `'krylov'` - Use the `scipy.optimize.root()` Krylov approximation for inverse Jacobian method.
  - `'diagbroyden'` - Use the `scipy.optimize.root()` diagonal Broyden Jacobian approximation method.
  - `'df-sane'` - Use the `scipy.optimize.root()` derivative-free spectral method.

  The following routines can use analytic Jacobians:

  - `'hybr'` - Use the `scipy.optimize.root()` modification of the Powell hybrid method implemented in MINIPACK.
  - `'lm'` - Uses the `scipy.optimize.root()` modification of the Levenberg-Marquardt algorithm implemented in MINIPACK.

  The following trivial routine can be used to simply return the initial values:

  - `'return'` - Assume that the initial values are the optimal ones.

  Also accepted is a custom callable method with the following form:

```
method(initial, contraction, callback, **options) -> (final,␣
↪converged)
```

  where `initial` is an array of initial values, `contraction` is a callable contraction mapping of the form specified below, `callback` is a function that should be called without any arguments after each major iteration (it is used to record the number of major iterations), `options` are specified below, `final` is an array of final values, and `converged` is a flag for whether the routine converged.

  The `contraction` function has the following form:

  contraction(x0) -> (x1, weights, jacobian)

  where `weights` are either `None` or a vector of weights that should multiply `x1 - x` before computing the norm of the differences, and `jacobian` is `None` if `compute_jacobian` is `False`.

Regardless of the chosen routine, if there are any computational issues that create infinities or null values, `final` will be the second to last iteration's values.

- **`method_options`** (*dict, optional*) – Options for the fixed point iteration routine.

  For routines other and `'simple'`, `'squarem'`, and `'return'`, these options will be passed to `options` in `scipy.optimize.root()`. Refer to the SciPy documentation for information about which options are available. By default, the `tol_norm` option is configured to use the infinity norm for SciPy methods other than `'hybr'` and `'lm'`, for which a norm cannot be specified.

  The `'simple'` and `'squarem'` methods support the following options:

  - **max_evaluations** : (*int*) - Maximum number of contraction mapping evaluations. The default value is `5000`.
  - **atol** : (*float*) - Absolute tolerance for convergence of the configured norm. The default value is `1e-14`. To use only a relative tolerance, set this to zero.
  - **rtol** (*float*) - Relative tolerance for convergence of the configured norm. The default value is zero; that is, only absolute tolerance is used by default.
  - **norm** : (*callable*) - The norm to be used. By default, the ℓ^∞-norm is used. If specified, this should be a function that accepts an array of differences and that returns a scalar norm.

  The `'squarem'` routine accepts additional options that mirror those in the SQUAREM package, written in R by Ravi Varadhan, which identifies the step length with $-\alpha$ from *Varadhan and Roland (2008)*:

  - **scheme** : (*int*) - The default value is `3`, which corresponds to S3 in *Varadhan and Roland (2008)*. Other acceptable schemes are `1` and `2`, which correspond to S1 and S2.
  - **step_min** : (*float*) - The initial value for the minimum step length. The default value is `1.0`.
  - **step_max** : (*float*) - The initial value for the maximum step length. The default value is `1.0`.
  - **step_factor** : (*float*) - When the step length exceeds `step_max`, it is set equal to `step_max`, but `step_max` is scaled by this factor. Similarly, if `step_min` is negative and the step length is below `step_min`, it is set equal to `step_min` and `step_min` is scaled by this factor. The default value is `4.0`.

- **`compute_jacobian`** (*bool, optional*) – Whether to compute an analytic Jacobian during iteration. By default, analytic Jacobians are not computed, and if a `method` is selected that supports analytic Jacobians, they will by default be numerically approximated.

- **`universal_display`** (*bool, optional*) – Whether to format iteration progress such that the display looks the same for all routines. By default, the universal display is not used and no iteration progress is displayed. Setting this to `True` can be helpful for debugging iteration issues. For example, iteration may get stuck above the configured termination tolerance.

Examples

The online version of the following section may be easier to read.

Iteration Example

```
import pyblp
import numpy as np

pyblp.__version__
```

```
'1.2.0'
```

In this example, we'll build a SQUAREM configuration with a ℓ^2-norm and use scheme S1 from *Varadhan and Roland (2008)*.

```
iteration = pyblp.Iteration('squarem', {'norm': np.linalg.norm, 'scheme': 1})
iteration
```

```
Configured to iterate using the SQUAREM acceleration method without analytic Jacobians with options {atol: +1.000000E-
↪14, rtol: 0, max_evaluations: 5000, norm: numpy.linalg.norm, scheme: 1, step_min: +1.000000E+00, step_max: +1.
↪000000E+00, step_factor: +4.000000E+00}.
```

Next, instead of using a built-in routine, we'll create a custom method that implements a version of simple iteration, which, for the sake of having a nontrivial example, arbitrarily identifies a major iteration with three objective evaluations.

```
def custom_method(initial, contraction, callback, max_evaluations, tol, norm):
    x = initial
    evaluations = 0
    while evaluations < max_evaluations:
        x0, (x, weights, _) = x, contraction(x)
        evaluations += 1
        if evaluations % 3 == 0:
            callback()
        if weights is None:
            difference = norm(x - x0)
        else:
            difference = norm(weights * (x - x0))
        if difference < tol:
```

(continues on next page)

(continued from previous page)

```
            break
    return x, evaluations < max_evaluations
```

We can then use this custom method to build a custom iteration configuration.

```
iteration = pyblp.Iteration(custom_method)
iteration
```

```
Configured to iterate using a custom method without analytic Jacobians with options {}.
```

5.1.4 pyblp.Optimization

class pyblp.**Optimization**(*method*, *method_options=None*, *compute_gradient=True*, *universal_display=True*)

Configuration for solving optimization problems.

Parameters

- **method** (*str or callable*) – The optimization routine that will be used. The following routines support parameter bounds and use analytic gradients:
  - `'knitro'` - Uses an installed version of Artleys Knitro. Python 3 is supported by Knitro version 10.3 and newer. A number of environment variables most likely need to be configured properly, such as `KNITRODIR`, `ARTELYS_LICENSE`, `LD_LIBRARY_PATH` (on Linux), and `DYLD_LIBRARY_PATH` (on Mac OS X). For more information, refer to the Knitro installation guide.
  - `'slsqp'` - Uses the `scipy.optimize.minimize()` SLSQP routine.
  - `'trust-constr'` - Uses the `scipy.optimize.minimize()` trust-region routine.
  - `'l-bfgs-b'` - Uses the `scipy.optimize.minimize()` L-BFGS-B routine.
  - `'tnc'` - Uses the `scipy.optimize.minimize()` TNC routine.

  The following routines also use analytic gradients but will ignore parameter bounds (not bounding the problem may create issues if the optimizer tries out large parameter values that create overflow errors):
  - `'cg'` - Uses the `scipy.optimize.minimize()` CG routine.
  - `'bfgs'` - Uses the `scipy.optimize.minimize()` BFGS routine.
  - `'newton-cg'` - Uses the `scipy.optimize.minimize()` Newton-CG routine.

  The following routines do not use analytic gradients and will also ignore parameter bounds (without analytic gradients, optimization will likely be much slower):
  - `'nelder-mead'` - Uses the `scipy.optimize.minimize()` Nelder-Mead routine.
  - `'powell'` - Uses the `scipy.optimize.minimize()` Powell routine.

  The following trivial routine can be used to evaluate an objective at specific parameter values:
  - `'return'` - Assume that the initial parameter values are the optimal ones.

  Also accepted is a custom callable method with the following form:

```
method(initial, bounds, objective_function, iteration_callback,␣
↪**options) -> (final, converged)
```

  where `initial` is an array of initial parameter values, `bounds` is a list of `(min, max)` pairs for each element in `initial`, `objective_function` is a callable objective function of the form specified below, `iteration_callback` is a function that should be called without any arguments after each major iteration (it is used to record the number of major iterations), `options` are specified below, `final` is an array of optimized parameter values, and `converged` is a flag for whether the routine converged.

  The `objective_function` has the following form:

  objective_function(theta) -> (objective, gradient, progress)

where `gradient` is `None` if `compute_gradient is ``False` and `progress` is an *`OptimizationProgress`* object that contains additional information about optimization progress so far, which may be helpful for debugging or to inform non-standard optimization routines.

- **`method_options`** (*dict, optional*) – Options for the optimization routine.

  For any non-custom `method` other than `'knitro'` and `'return'`, these options will be passed to `options` in `scipy.optimize.minimize()`, with the exception of `'keep_feasible'`, which is by default `True` and is passed to any `scipy.optimize.Bounds`. Refer to the SciPy documentation for information about which options are available for each optimization routine.

  If `method` is `'knitro'`, these options should be Knitro user options. The non-standard `knitro_dir` option can also be specified. The following options have non-standard default values:

  - **knitro_dir** : (*str*) - By default, the KNITRODIR environment variable is used. Otherwise, this option should point to the installation directory of Knitro, which contains direct subdirectories such as `'examples'` and `'lib'`. For example, on Windows this option could be `'/Program Files/Artleys3/Knitro 10.3.0'`.
  - **algorithm** : (*int*) - The optimization algorithm to be used. The default value is `1`, which corresponds to the Interior/Direct algorithm.
  - **gradopt** : (*int*) - How the objective's gradient is computed. The default value is `1` if `compute_gradient` is `True` and is `2` otherwise, which corresponds to estimating the gradient with finite differences.
  - **hessopt** : (*int*) - How the objective's Hessian is computed. The default value is `2`, which corresponds to computing a quasi-Newton BFGS Hessian.
  - **honorbnds** : (*int*) - Whether to enforce satisfaction of simple variable bounds. The default value is `1`, which corresponds to enforcing that the initial point and all subsequent solution estimates satisfy the bounds.

- **`compute_gradient`** (*bool, optional*) – Whether to compute an analytic objective gradient during optimization, which must be `False` if `method` does not use analytic gradients, and must be `True` if `method` is `'newton-cg'`, which requires an analytic gradient.

  By default, analytic gradients are computed. Not using an analytic gradient will likely slow down estimation a good deal. If `False`, an analytic gradient may still be computed once at the end of optimization to compute optimization results. To always use finite differences, `finite_differences` in *`Problem.solve()`* can be set to `True`.

- **`universal_display`** (*bool, optional*) – Whether to format optimization progress such that the display looks the same for all routines. By default, the universal display is used and some `method_options` are used to prevent default displays from showing up.

Examples

The online version of the following section may be easier to read.

Optimization Example

```
import pyblp
import numpy as np

pyblp.__version__
```

```
'1.2.0'
```

In this example, we'll build a L-BFGS-B configuration with a non-default tolerance.

```
optimization = pyblp.Optimization('l-bfgs-b', {'gtol': 1e-3})
optimization
```

```
Configured to optimize using the L-BFGS-B algorithm implemented in SciPy with analytic gradients and options {gtol: +1.
↪000000E-03}.
```

Next, instead of using a non-custom routine, we'll create a custom method that implements a grid search over parameter values between specified bounds.

```
from itertools import product
def custom_method(initial, bounds, objective_function, iteration_callback):
    best_values = initial
    best_objective = np.inf
    for values in product(*(np.linspace(l, u, 10) for l, u in bounds)):
        objective, _, _ = objective_function(values)
        if objective < best_objective:
            best_values = values
            best_objective = objective
        iteration_callback()
    return best_values, True
```

We can then use this custom method to build an optimization configuration.

```
optimization = pyblp.Optimization(custom_method, compute_gradient=False)
optimization
```

```
Configured to optimize using a custom method without analytic gradients and options {}.
```

Custom optimization configurations can be used to help debug optimization, to define non-standard optimization routines, or to add ad-hoc moments to configured problems. They can use various information about optimization progress so far.

OptimizationProgress	Information about the current progress of optimization.

5.1.5 pyblp.OptimizationProgress

class pyblp.**OptimizationProgress**

Information about the current progress of optimization.

They key attributes of this class needed to define a custom optimization routine are `objective` and `gradient`. Many other attributes of *ProblemResults* are also included, and can be used to help define an alternative optimization routine (e.g., Gauss-Newton), to debug issues, or to add custom ad-hoc moments to the configured problem.

problem

Problem that created this progress.

Type *Problem*

fp_converged

Flags for convergence of the iteration routine used to compute $\delta(\theta)$ in each market. Values are in the same order as *Problem.unique_market_ids*.

Type *ndarray*

fp_iterations

Number of major iterations completed by the iteration routine used to compute $\delta(\theta)$ in each market. Values are in the same order as *Problem.unique_market_ids*.

Type *ndarray*

contraction_evaluations

Number of times the contraction used to compute $\delta(\theta)$ was evaluated in each market. Values are in the same order as *Problem.unique_market_ids*.

Type *ndarray*

theta

Unfixed parameters, θ, in the following order: Σ, Π, ρ, non-concentrated out elements from β, and non-concentrated out elements from γ.

Type *ndarray*

sigma

Cholesky root of the covariance matrix for unobserved taste heterogeneity, Σ.

Type *ndarray*

sigma_squared

Covariance matrix for unobserved taste heterogeneity, $\Sigma\Sigma'$.

Type *ndarray*

pi

Parameters that measures how agent tastes vary with demographics, Π.

Type *ndarray*

rho
Parameters that measure within nesting group correlations, ρ.

Type *ndarray*

phi
Parameters that measure unobservable autocorrelation, ϕ.

Type *ndarray*

beta
Demand-side linear parameters, β.

Type *ndarray*

gamma
Supply-side linear parameters, γ.

Type *ndarray*

sigma_bounds
Bounds for Σ that were used during optimization, which are of the form `(lb, ub)`.

Type *tuple*

pi_bounds
Bounds for Π that were used during optimization, which are of the form `(lb, ub)`.

Type *tuple*

rho_bounds
Bounds for ρ that were used during optimization, which are of the form `(lb, ub)`.

Type *tuple*

phi_bounds
Bounds for ϕ that were used during optimization, which are of the form `(lb, ub)`.

Type *tuple*

beta_bounds
Bounds for β that were used during optimization, which are of the form `(lb, ub)`.

Type *tuple*

gamma_bounds
Bounds for γ that were used during optimization, which are of the form `(lb, ub)`.

Type *tuple*

sigma_labels
Variable labels for rows and columns of Σ, which are derived from the formulation for X_2.

Type *list of str*

pi_labels
Variable labels for columns of Π, which are derived from the formulation for demographics.

Type *list of str*

rho_labels
Variable labels for ρ. If ρ is not a scalar, this is *Problem.unique_nesting_ids*.

Type *list of str*

phi_labels
Variable labels for ϕ.

Type *list of str*

beta_labels
Variable labels for β, which are derived from the formulation for X_1.

Type *list of str*

gamma_labels
Variable labels for γ, which are derived from the formulation for X_3.

Type *list of str*

theta_labels
Variable labels for θ, which are derived from the above labels.

Type *list of str*

delta
Mean utility, $\delta(\theta)$.

Type *ndarray*

clipped_shares
Vector of booleans indicating whether the associated simulated shares were clipped during the last fixed point iteration to compute $\delta(\theta)$. All elements will be `False` if `shares_bounds` in *Problem.solve()* is disabled (by default shares are bounded from below by a small number to alleviate issues with underflow and negative shares).

Type *ndarray*

tilde_costs
Estimated transformed marginal costs, $\tilde{c}(\theta)$ from (3.9). If `costs_bounds` were specified in *Problem.solve()*, c may have been clipped.

Type *ndarray*

clipped_costs
Vector of booleans indicating whether the associated marginal costs were clipped. All elements will be `False` if `costs_bounds` in *Problem.solve()* was not specified.

Type *ndarray*

xi
Unobserved demand-side product characteristics, $\xi(\theta)$, or equivalently, the demand-side structural error term. Any absorbed fixed effects are not included.

Type *ndarray*

omega
Unobserved supply-side product characteristics, $\omega(\theta)$, or equivalently, the supply-side structural error term. Any absorbed fixed effects are not included.

Type *ndarray*

micro
Micro moments, $\bar{g}_M$, in (3.33).

Type *ndarray*

micro_values
Micro moment values, $f_m(v)$. Rows are in the same order as *ProblemResults.micro*.

Type *ndarray*

objective
GMM objective value, $q(\theta)$, defined in (3.10). If `scale_objective` was `True` in *Problem.solve()* (which is the default), this value was scaled by N so that objective values are more comparable across different problem sizes. Note that in some of the BLP literature (and earlier versions of this package), this expression was previously scaled by N^2.

Type *float*

xi_by_theta_jacobian
$\frac{\partial \xi}{\partial \theta} = \frac{\partial \delta}{\partial \theta}$.

Type *ndarray*

omega_by_theta_jacobian
$\frac{\partial \omega}{\partial \theta} = \frac{\partial \tilde{c}}{\partial \theta}$.

Type *ndarray*

micro_by_theta_jacobian
$\frac{\partial \bar{g}_M}{\partial \theta}$.

Type *ndarray*

gradient
Gradient of the GMM objective, $\nabla q(\theta)$, defined in (3.18).

Type *ndarray*

projected_gradient
Projected gradient of the GMM objective. When there are no parameter bounds, this will always be equal to *ProblemResults.gradient*. Otherwise, if an element in $\hat{\theta}$ is equal to its lower (upper) bound, the corresponding projected gradient value will be truncated at a maximum (minimum) of zero.

Type *ndarray*

projected_gradient_norm
Infinity norm of *ProblemResults.projected_gradient*.

Type *ndarray*

W
Weighting matrix, W, used to compute these results.

Type *ndarray*

Methods

—

5.2 Data Manipulation Functions

There are also a number of convenience functions that can be used to construct common components of product and agent data, or manipulate other PyBLP objects.

build_matrix(formulation, data)	Construct a matrix according to a formulation.
build_blp_instruments(formulation, product_data)	Construct "sums of characteristics" excluded BLP instruments.

Continued on next page

Table 4 – continued from previous page

build_differentiation_instruments(... [, ...])	Construct excluded differentiation instruments.
build_id_data(T, J, F)	Build a balanced panel of market and firm IDs.
build_ownership(product_data[, ...])	Build ownership matrices, O.
build_integration(integration, dimensions)	Build nodes and weights for integration over agent choice probabilities.
data_to_dict(data[, ignore_empty])	Convert a NumPy record array into a dictionary.
save_pickle(x, path)	Save an object as a pickle file.
read_pickle(path)	Load a pickled object into memory.

5.2.1 pyblp.build_matrix

pyblp.**build_matrix**(*formulation*, *data*)

Construct a matrix according to a formulation.

Parameters

- **formulation** (*Formulation*) – *Formulation* configuration for the matrix. Variable names should correspond to fields in `data`. The `absorb` argument of *Formulation* can be used to absorb fixed effects after the matrix has been constructed.
- **data** (*structured array-like*) – Fields can be used as variables in `formulation`.

Returns The built matrix.

Return type *ndarray*

Examples

The online version of the following section may be easier to read.

Building a Matrix Example

```
import pyblp
import pandas as pd

pyblp.__version__
```

```
'1.2.0'
```

In this example, we'll load the fake cereal data from *Nevo (2000a)* and create a simple matrix involving a constant, prices, and shares.

```
formulation = pyblp.Formulation('1 + prices + shares')
formulation
```

```
1 + prices + shares
```

```
product_data = pd.read_csv(pyblp.data.NEVO_PRODUCTS_LOCATION)
product_data.head()
```

```
  market_ids  city_ids  quarter product_ids  firm_ids  brand_ids    shares  \
0      C01Q1         1        1       F1B04         1          4  0.012417
1      C01Q1         1        1       F1B06         1          6  0.007809
2      C01Q1         1        1       F1B07         1          7  0.012995
3      C01Q1         1        1       F1B09         1          9  0.005770
4      C01Q1         1        1       F1B11         1         11  0.017934

     prices  sugar  mushy  ...  demand_instruments10  demand_instruments11  \
0  0.072088      2      1  ...              2.116358             -0.154708
1  0.114178     18      1  ...             -7.374091             -0.576412
2  0.132391      4      1  ...              2.187872             -0.207346
3  0.130344      3      0  ...              2.704576              0.040748
4  0.154823     12      0  ...              1.261242              0.034836

   demand_instruments12  demand_instruments13  demand_instruments14  \
0             -0.005796              0.014538              0.126244
1              0.012991              0.076143              0.029736
```

(continues on next page)

(continued from previous page)

```
2               0.003509              0.091781               0.163773
3              -0.003724              0.094732               0.135274
4              -0.000568              0.102451               0.130640

   demand_instruments15  demand_instruments16  demand_instruments17  \
0              0.067345              0.068423              0.034800
1              0.087867              0.110501              0.087784
2              0.111881              0.108226              0.086439
3              0.088090              0.101767              0.101777
4              0.084818              0.101075              0.125169

   demand_instruments18  demand_instruments19
0              0.126346              0.035484
1              0.049872              0.072579
2              0.122347              0.101842
3              0.110741              0.104332
4              0.133464              0.121111

[5 rows x 30 columns]
```

```
matrix = pyblp.build_matrix(formulation, product_data)
matrix
```

```
array([[1.        , 0.07208794, 0.01241721],
       [1.        , 0.11417849, 0.00780939],
       [1.        , 0.13239066, 0.01299451],
       ...,
       [1.        , 0.13701741, 0.00222918],
       [1.        , 0.10017433, 0.01146267],
       [1.        , 0.12755747, 0.02620832]], shape=(2256, 3))
```

For various reasons, we may want to absorb fixed effects into the matrix. This can be done with the `absorb` argument of *Formulation*. We'll now re-create the matrix, absorbing product-specific fixed effects. Note that the constant column is now ignored.

```
absorb_formulation = pyblp.Formulation('prices + shares', absorb='product_ids')
absorb_formulation
```

```
prices + shares + Absorb[product_ids]
```

```
demeaned_matrix = pyblp.build_matrix(absorb_formulation, product_data)
demeaned_matrix
```

```
array([[-0.01124832, -0.00052161],
       [-0.00713476, -0.03144549],
       [ 0.02367765, -0.01664996],
       ...,
       [ 0.03371995, -0.00779841],
       [-0.00417404, -0.0117508 ],
       [-0.01195648,  0.00666695]], shape=(2256, 2))
```

5.2.2 pyblp.build_blp_instruments

pyblp.**build_blp_instruments**(*formulation*, *product_data*)

Construct "sums of characteristics" excluded BLP instruments.

Traditional "sums of characteristics" BLP instruments are

$$Z^{\text{BLP}}(X) = [Z^{\text{BLP,Other}}(X), Z^{\text{BLP,Rival}}(X)], \tag{5.1}$$

in which X is a matrix of product characteristics, $Z^{\text{BLP,Other}}(X)$ is a second matrix that consists of sums over characteristics of non-rival goods, and $Z^{\text{BLP,Rival}}(X)$ is a third matrix that consists of sums over rival goods. All three matrices have the same dimensions.

Note: To construct simpler, firm-agnostic instruments that are sums over characteristics of other goods, specify a constant column of firm IDs and keep only the first half of the instrument columns.

Let x_{jt} be the vector of characteristics in X for product j in market t, which is produced by firm f. That is, $j \in J_{ft}$. Then,

$$\begin{aligned} Z^{\text{BLP,Other}}_{jt}(X) &= \sum_{k \in J_{ft} \setminus \{j\}} x_{kt}, \\ Z^{\text{BLP,Rival}}_{jt}(X) &= \sum_{k \notin J_{ft}} x_{kt}. \end{aligned} \tag{5.2}$$

Note: Usually, any supply or demand shifters are added to these excluded instruments, depending on whether they are meant to be used for demand- or supply-side estimation.

Parameters

- **formulation** (*Formulation*) – `Formulation` configuration for X, the matrix of product characteristics used to build excluded instruments. Variable names should correspond to fields in `product_data`.
- **product_data** (*structured array-like*) – Each row corresponds to a product. Markets can have differing numbers of products. The following fields are required:
  - **market_ids** : (*object*) - IDs that associate products with markets.
  - **firm_ids** : (*object*) - IDs that associate products with firms.

  Along with `market_ids` and `firm_ids`, the names of any additional fields can be used as variables in `formulation`.

Returns Traditional "sums of characteristics" BLP instruments, $Z^{\text{BLP}}(X)$.

Return type *ndarray*

Examples

The online version of the following section may be easier to read.

Building "Sums of Characteristics" BLP Instruments Example

```
import pyblp
import numpy as np
import pandas as pd

np.set_printoptions(precision=3)
pyblp.__version__
```

```
'1.2.0'
```

In this example, we'll load the automobile product data from *Berry, Levinsohn, and Pakes (1995)* and show how to construct the included instruments from scratch.

```
product_data = pd.read_csv(pyblp.data.BLP_PRODUCTS_LOCATION)
product_data.head()
```

```
   market_ids clustering_ids  car_ids  firm_ids region    shares    prices  \
0        1971       AMGREM71      129        15     US  0.001051  4.935802
1        1971       AMHORN71      130        15     US  0.000670  5.516049
2        1971       AMJAVL71      132        15     US  0.000341  7.108642
3        1971       AMMATA71      134        15     US  0.000522  6.839506
4        1971       AMAMBS71      136        15     US  0.000442  8.928395

       hpwt  air       mpd  ...  supply_instruments2  supply_instruments3  \
0  0.528997    0  1.888146  ...                  0.0             1.705933
1  0.494324    0  1.935989  ...                  0.0             1.680910
2  0.467613    0  1.716799  ...                  0.0             1.801067
3  0.426540    0  1.687871  ...                  0.0             1.818061
4  0.452489    0  1.504286  ...                  0.0             1.933210

   supply_instruments4  supply_instruments5  supply_instruments6  \
0             1.595656                 87.0           -61.959985
1             1.490295                 87.0           -61.959985
2             1.357703                 87.0           -61.959985
3             1.261347                 87.0           -61.959985
4             1.237365                 87.0           -61.959985
```

(continues on next page)

(continued from previous page)

```
   supply_instruments7  supply_instruments8  supply_instruments9  \
0                  0.0            46.060389            29.786989
1                  0.0            46.060389            29.786989
2                  0.0            46.060389            29.786989
3                  0.0            46.060389            29.786989
4                  0.0            46.060389            29.786989

   supply_instruments10  supply_instruments11
0                   0.0              1.888146
1                   0.0              1.935989
2                   0.0              1.716799
3                   0.0              1.687871
4                   0.0              1.504286

[5 rows x 33 columns]
```

```
product_data[[f'demand_instruments{i}' for i in range(8)]]
```

```
      demand_instruments0  demand_instruments1  demand_instruments2  \
0                     4.0             1.840967                  0.0
1                     4.0             1.875639                  0.0
2                     4.0             1.902350                  0.0
3                     4.0             1.943423                  0.0
4                     4.0             1.917475                  0.0
...                   ...                  ...                  ...
2212                  2.0             0.826512                  2.0
2213                  2.0             0.776462                  2.0
2214                  0.0             0.000000                  0.0
2215                  1.0             0.693796                  1.0
2216                  1.0             0.814913                  1.0

      demand_instruments3  demand_instruments4  demand_instruments5  \
0                6.844945                 87.0            44.555539
1                6.797102                 87.0            44.555539
2                7.016291                 87.0            44.555539
3                7.045220                 87.0            44.555539
4                7.228805                 87.0            44.555539
...                   ...                  ...                  ...
2212             4.775577                128.0            57.660253
```

(continues on next page)

(continued from previous page)

```
2213              5.278269                 128.0             57.660253
2214              0.000000                 130.0             58.514393
2215              3.267500                 129.0             57.363973
2216              3.016154                 129.0             57.363973

      demand_instruments6  demand_instruments7
0                     0.0           167.325082
1                     0.0           167.325082
2                     0.0           167.325082
3                     0.0           167.325082
4                     0.0           167.325082
...                   ...                  ...
2212                 57.0           351.758942
2213                 57.0           351.758942
2214                 60.0           355.654808
2215                 58.0           352.890000
2216                 58.0           352.890000

[2217 rows x 8 columns]
```

```
product_data[[f'supply_instruments{i}' for i in range(12)]]
```

```
      supply_instruments0  supply_instruments1  supply_instruments2  \
0                     4.0            -3.109718                  0.0
1                     4.0            -3.041927                  0.0
2                     4.0            -2.986377                  0.0
3                     4.0            -2.894442                  0.0
4                     4.0            -2.953498                  0.0
...                   ...                  ...                  ...
2212                  2.0            -1.770401                  2.0
2213                  2.0            -1.892345                  2.0
2214                  0.0             0.000000                  0.0
2215                  1.0            -0.365578                  1.0
2216                  1.0            -0.204674                  1.0

      supply_instruments3  supply_instruments4  supply_instruments5  \
0                1.705933             1.595656                 87.0
1                1.680910             1.490295                 87.0
2                1.801067             1.357703                 87.0
3                1.818061             1.261347                 87.0
```

(continues on next page)

(continued from previous page)

```
4                1.933210             1.237365                 87.0
...                   ...                  ...                  ...
2212             1.272566             0.511989                128.0
2213             1.483875             0.511989                128.0
2214             0.000000             0.000000                130.0
2215             0.955511             0.142876                129.0
2216             0.875469             0.089795                129.0

      supply_instruments6  supply_instruments7  supply_instruments8  \
0              -61.959985                  0.0            46.060389
1              -61.959985                  0.0            46.060389
2              -61.959985                  0.0            46.060389
3              -61.959985                  0.0            46.060389
4              -61.959985                  0.0            46.060389
...                   ...                  ...                  ...
2212          -104.631050                 57.0            97.039220
2213          -104.631050                 57.0            97.039220
2214          -106.327167                 60.0            98.024103
2215          -106.783331                 58.0            97.222743
2216          -106.783331                 58.0            97.222743

      supply_instruments9  supply_instruments10  supply_instruments11
0               29.786989                   0.0              1.888146
1               29.786989                   0.0              1.935989
2               29.786989                   0.0              1.716799
3               29.786989                   0.0              1.687871
4               29.786989                   0.0              1.504286
...                   ...                  ...                  ...
2212            27.861181                  38.0              2.639135
2213            27.861181                  38.0              2.136442
2214            28.809765                   0.0              3.518846
2215            28.407171                  19.0              3.016154
2216            28.407171                  19.0              3.267500

[2217 rows x 12 columns]
```

The demand-side "sums of characterstics" BLP instruments included in `product_data` can be built from scratch with the *build_blp_instruments* function.

```
demand_instruments = pyblp.build_blp_instruments(pyblp.Formulation('1 + hpwt + air + mpd'), product_data)
```

(continues on next page)

(continued from previous page)

```
demand_instruments
```

```
array([[  4.   ,   1.841,   0.   , ...,  44.556,   0.   , 167.325],
       [  4.   ,   1.876,   0.   , ...,  44.556,   0.   , 167.325],
       [  4.   ,   1.902,   0.   , ...,  44.556,   0.   , 167.325],
       ...,
       [  0.   ,   0.   ,   0.   , ...,  58.514,  60.   , 355.655],
       [  1.   ,   0.694,   1.   , ...,  57.364,  58.   , 352.89 ],
       [  1.   ,   0.815,   1.   , ...,  57.364,  58.   , 352.89 ]],
      shape=(2217, 8))
```

The supply-side instruments from the original paper are "sums of characteristics" BLP instruments as well, but also include a standalone `mpd` shifter. Because of collinearity issues, the "rival" instrument constructed from the `trend` variable is excluded, and only the "own" instrument is retained.

```
supply_instruments = np.c_[
    pyblp.build_blp_instruments(pyblp.Formulation('1 + log(hpwt) + air + log(mpg) + log(space)'), product_data),
    pyblp.build_blp_instruments(pyblp.Formulation('0 + trend'), product_data)[:, 0],
    product_data['mpd'],
]
supply_instruments
```

```
array([[ 4.   , -3.11 ,  0.   , ..., 29.787,  0.   ,  1.888],
       [ 4.   , -3.042,  0.   , ..., 29.787,  0.   ,  1.936],
       [ 4.   , -2.986,  0.   , ..., 29.787,  0.   ,  1.717],
       ...,
       [ 0.   ,  0.   ,  0.   , ..., 28.81 ,  0.   ,  3.519],
       [ 1.   , -0.366,  1.   , ..., 28.407, 19.   ,  3.016],
       [ 1.   , -0.205,  1.   , ..., 28.407, 19.   ,  3.268]],
      shape=(2217, 12))
```

5.2.3 pyblp.build_differentiation_instruments

pyblp.**build_differentiation_instruments**(*formulation*, *product_data*, *version='local'*, *interact=False*)

Construct excluded differentiation instruments.

Differentiation instruments in the spirit of *Gandhi and Houde (2025)* are

$$Z^{\text{Diff}}(X) = [Z^{\text{Diff,Other}}(X), Z^{\text{Diff,Rival}}(X)], \tag{5.3}$$

in which X is a matrix of product characteristics, $Z^{\text{Diff,Other}}(X)$ is a second matrix that consists of sums over functions of differences between non-rival goods, and $Z^{\text{Diff,Rival}}(X)$ is a third matrix that consists of sums over rival goods. Without optional interaction terms, all three matrices have the same dimensions.

Note: To construct simpler, firm-agnostic instruments that are sums over functions of differences between all different goods, specify a constant column of firm IDs and keep only the first half of the instrument columns.

Let $x_{jt\ell}$ be characteristic ℓ in X for product j in market t, which is produced by firm f. That is, $j \in J_{ft}$. Then in the "local" version of $Z^{\text{Diff}}(X)$,

$$\begin{aligned} Z^{\text{Local,Other}}_{jt\ell}(X) &= \sum_{k \in J_{ft} \setminus \{j\}} 1(|d_{jkt\ell}| < \text{SD}_\ell), \\ Z^{\text{Local,Rival}}_{jt\ell}(X) &= \sum_{k \notin J_{ft}} 1(|d_{jkt\ell}| < \text{SD}_\ell), \end{aligned} \tag{5.4}$$

where $d_{jkt\ell} = x_{kt\ell} - x_{jt\ell}$ is the difference between products j and k in terms of characteristic ℓ, SD_ℓ is the standard deviation of these pairwise differences computed across all markets, and $1(|d_{jkt\ell}| < \text{SD}_\ell)$ indicates that products j and k are close to each other in terms of characteristic ℓ.

The intuition behind this "local" version is that demand for products is often most influenced by a small number of other goods that are very similar. For the "quadratic" version of $Z^{\text{Diff}}(X)$, which uses a more continuous measure of the distance between goods,

$$\begin{aligned} Z^{\text{Quad,Other}}_{jtk}(X) &= \sum_{k \in J_{ft} \setminus \{j\}} d^2_{jkt\ell}, \\ Z^{\text{Quad,Rival}}_{jtk}(X) &= \sum_{k \notin J_{ft}} d^2_{jkt\ell}. \end{aligned} \tag{5.5}$$

With interaction terms, which reflect covariances between different characteristics, the summands for the "local" versions are $1(|d_{jkt\ell}| < \text{SD}_\ell) \times d_{jkt\ell'}$ for all characteristics ℓ', and the summands for the "quadratic" versions are $d_{jkt\ell} \times d_{jkt\ell'}$ for all $\ell' \geq \ell$.

Note: Usually, any supply or demand shifters are added to these excluded instruments, depending on whether they are meant to be used for demand- or supply-side estimation.

Parameters

- **formulation** (*Formulation*) – `Formulation` configuration for X, the matrix of product characteristics used to build excluded instruments. Variable names should correspond to fields in `product_data`.
- **product_data** (*structured array-like*) – Each row corresponds to a product. Markets can have differing numbers of products. The following fields are required:

- **market_ids** : (*object*) - IDs that associate products with markets.
- **firm_ids** : (*object*) - IDs that associate products with firms.

Along with `market_ids` and `firm_ids`, the names of any additional fields can be used as variables in `formulation`.

- **version** (*str, optional*) – The version of differentiation instruments to construct:
  - `'local'` (default) - Construct the instruments in (5.4) that consider only the characteristics of "close" products in each market.
  - `'quadratic'` - Construct the more continuous instruments in (5.5) that consider all products in each market.
- **interact** (*bool, optional*) – Whether to include interaction terms between different product characteristics, which can help capture covariances between product characteristics.

Returns Excluded differentiation instruments, $Z^{\text{Diff}}(X)$.

Return type *ndarray*

Examples

The online version of the following section may be easier to read.

Building Differentiation Instruments Example

```
import pyblp
import numpy as np
import pandas as pd

np.set_printoptions(precision=3)
pyblp.__version__
```

```
'1.2.0'
```

In this example, we'll load the automobile product data from *Berry, Levinsohn, and Pakes (1995)*, build some very simple excluded demand-side instruments for the problem in the spirit of *Gandhi and Houde (2025)*, and demonstrate how to update the problem data to use these instrument instead of the default ones.

```
product_data = pd.read_csv(pyblp.data.BLP_PRODUCTS_LOCATION)
product_data.head()
```

```
   market_ids clustering_ids  car_ids  firm_ids region    shares    prices  \
0        1971       AMGREM71      129        15     US  0.001051  4.935802
1        1971       AMHORN71      130        15     US  0.000670  5.516049
2        1971       AMJAVL71      132        15     US  0.000341  7.108642
3        1971       AMMATA71      134        15     US  0.000522  6.839506
4        1971       AMAMBS71      136        15     US  0.000442  8.928395

       hpwt  air       mpd  ...  supply_instruments2  supply_instruments3  \
0  0.528997    0  1.888146  ...                  0.0             1.705933
1  0.494324    0  1.935989  ...                  0.0             1.680910
2  0.467613    0  1.716799  ...                  0.0             1.801067
3  0.426540    0  1.687871  ...                  0.0             1.818061
4  0.452489    0  1.504286  ...                  0.0             1.933210

   supply_instruments4  supply_instruments5  supply_instruments6  \
0             1.595656                 87.0           -61.959985
1             1.490295                 87.0           -61.959985
2             1.357703                 87.0           -61.959985
3             1.261347                 87.0           -61.959985
```

(continues on next page)

(continued from previous page)

```
4             1.237365                  87.0           -61.959985

   supply_instruments7  supply_instruments8  supply_instruments9  \
0                  0.0            46.060389            29.786989
1                  0.0            46.060389            29.786989
2                  0.0            46.060389            29.786989
3                  0.0            46.060389            29.786989
4                  0.0            46.060389            29.786989

   supply_instruments10  supply_instruments11
0                   0.0              1.888146
1                   0.0              1.935989
2                   0.0              1.716799
3                   0.0              1.687871
4                   0.0              1.504286

[5 rows x 33 columns]
```

We'll first build "local" differentiation instruments, which are constructed by default, and which consist of counts of "close" rival and non-rival products in each market. Note that we're excluding the constant column because it yields collinear constant columns of differentiation instruments.

```
formulation = pyblp.Formulation('0 + hpwt + air + mpd')
local_instruments = pyblp.build_differentiation_instruments(
    formulation,
    product_data
)
local_instruments
```

```
array([[ 4.,  4.,  4., 42., 87., 83.],
       [ 4.,  4.,  4., 53., 87., 84.],
       [ 4.,  4.,  4., 51., 87., 78.],
       ...,
       [ 0.,  0.,  0., 86., 70., 62.],
       [ 1.,  1.,  1.,  3., 58., 91.],
       [ 1.,  1.,  1., 13., 58., 72.]], shape=(2217, 6))
```

Next, we'll build a more continuous "quadratic" version of the instruments, which consist of sums over squared differences between rival and non-rival products in each market.

```
quadratic_instruments = pyblp.build_differentiation_instruments(
    formulation,
    product_data,
    version='quadratic'
)
quadratic_instruments
```

```
array([[2.132e-02, 0.000e+00, 2.191e-01, 2.011e+00, 0.000e+00, 1.208e+01],
       [8.261e-03, 0.000e+00, 2.983e-01, 2.014e+00, 0.000e+00, 1.198e+01],
       [6.397e-03, 0.000e+00, 1.234e-01, 2.159e+00, 0.000e+00, 1.568e+01],
       ...,
       [0.000e+00, 0.000e+00, 0.000e+00, 2.239e+00, 6.000e+01, 1.312e+02],
       [1.467e-02, 0.000e+00, 6.317e-02, 1.864e+01, 7.100e+01, 6.185e+01],
       [1.467e-02, 0.000e+00, 6.317e-02, 8.961e+00, 7.100e+01, 8.819e+01]],
      shape=(2217, 6))
```

We could also use `interact=True` to include interaction terms in either version of instruments, which would help capture covariances between different product characteristics.

To use these instruments when setting up a *Problem*, the existing product data has to be updated or new product data has to be constructed. Since the existing product data is a Pandas `DataFrame`, it does not support matrices, so each column of instruments has to be added individually after deleting the existing instruments.

```
for i in range(8):
    del product_data[f'demand_instruments{i}']

for i, column in enumerate(local_instruments.T):
    product_data[f'demand_instruments{i}'] = column

product_data
```

```
      market_ids clustering_ids  car_ids  firm_ids region    shares  \
0           1971       AMGREM71      129        15     US  0.001051
1           1971       AMHORN71      130        15     US  0.000670
2           1971       AMJAVL71      132        15     US  0.000341
3           1971       AMMATA71      134        15     US  0.000522
4           1971       AMAMBS71      136        15     US  0.000442
...          ...            ...      ...       ...    ...       ...
2212        1990        VV74085     5584         6     EU  0.000488
2213        1990       VV760G87     5585         6     EU  0.000091
2214        1990       YGGVPL90     5589        23     EU  0.000067
2215        1990       PS911C90     5590        12     EU  0.000039
```

(continues on next page)

(continued from previous page)

```
2216        1990        PS94490     5592        12     EU  0.000025

         prices      hpwt  air       mpd  ...  supply_instruments8  \
0      4.935802  0.528997    0  1.888146  ...            46.060389
1      5.516049  0.494324    0  1.935989  ...            46.060389
2      7.108642  0.467613    0  1.716799  ...            46.060389
3      6.839506  0.426540    0  1.687871  ...            46.060389
4      8.928395  0.452489    0  1.504286  ...            46.060389
...         ...       ...  ...       ...  ...                  ...
2212  16.140015  0.385917    1  2.639135  ...            97.039220
2213  25.986993  0.435967    1  2.136442  ...            97.039220
2214   3.393267  0.358289    0  3.518846  ...            98.024103
2215  44.758990  0.814913    1  3.016154  ...            97.222743
2216  32.058148  0.693796    1  3.267500  ...            97.222743

      supply_instruments9  supply_instruments10  supply_instruments11  \
0               29.786989                   0.0              1.888146
1               29.786989                   0.0              1.935989
2               29.786989                   0.0              1.716799
3               29.786989                   0.0              1.687871
4               29.786989                   0.0              1.504286
...                   ...                   ...                   ...
2212            27.861181                  38.0              2.639135
2213            27.861181                  38.0              2.136442
2214            28.809765                   0.0              3.518846
2215            28.407171                  19.0              3.016154
2216            28.407171                  19.0              3.267500

      demand_instruments0  demand_instruments1  demand_instruments2  \
0                     4.0                  4.0                  4.0
1                     4.0                  4.0                  4.0
2                     4.0                  4.0                  4.0
3                     4.0                  4.0                  4.0
4                     4.0                  4.0                  4.0
...                   ...                  ...                  ...
2212                  2.0                  2.0                  2.0
2213                  2.0                  2.0                  2.0
2214                  0.0                  0.0                  0.0
2215                  1.0                  1.0                  1.0
2216                  1.0                  1.0                  1.0
```

(continues on next page)

(continued from previous page)

```
      demand_instruments3  demand_instruments4  demand_instruments5
0                    42.0                 87.0                 83.0
1                    53.0                 87.0                 84.0
2                    51.0                 87.0                 78.0
3                    52.0                 87.0                 77.0
4                    52.0                 87.0                 69.0
...                   ...                  ...                  ...
2212                102.0                 57.0                109.0
2213                112.0                 57.0                 86.0
2214                 86.0                 70.0                 62.0
2215                  3.0                 58.0                 91.0
2216                 13.0                 58.0                 72.0

[2217 rows x 31 columns]
```

Any data type that has fields can be used as product data. An alternative way to specify `problem_data` for *Problem* initialization is to simply use a `dict`, where fields can be matrices. For example, we could use the following `dict`, which includes both the new demand instruments as well as a few other variables that might be used when setting up the problem.

```
product_data_dict = {k: product_data[k] for k in ['market_ids', 'firm_ids', 'shares', 'prices', 'hpwt', 'air', 'mpd']}
product_data_dict['demand_instruments'] = local_instruments
```

5.2.4 pyblp.build_id_data

pyblp.**build_id_data**(*T*, *J*, *F*)

Build a balanced panel of market and firm IDs.

This function can be used to build `id_data` for *Simulation* initialization.

Parameters

- **T** (*int*) – Number of markets.
- **J** (*int*) – Number of products in each market.
- **F** (*int*) – Number of firms. If `J` is divisible by `F`, firms produce `J / F` products in each market. Otherwise, firms with smaller IDs will produce excess products.

Returns

IDs that associate products with markets and firms. Each of the `T * J` rows corresponds to a product. Fields:

- **market_ids** : (*object*) - Market IDs that take on values from `0` to `T - 1`.
- **firm_ids** : (*object*) - Firm IDs that take on values from `0` to `F - 1`.

Return type *recarray*

Examples

The online version of the following section may be easier to read.

Building ID Data Example

```
import pyblp
import numpy as np

np.set_printoptions(linewidth=1)
pyblp.__version__
```

```
'1.2.0'
```

In this example, we'll build a small panel of market and firm IDs.

```
id_data = pyblp.build_id_data(T=2, J=5, F=4)
id_data
```

```
rec.array([([0], [0]),
           ([0], [0]),
           ([0], [1]),
           ([0], [2]),
           ([0], [3]),
           ([1], [0]),
           ([1], [0]),
           ([1], [1]),
           ([1], [2]),
           ([1], [3])],
          dtype=[('market_ids', 'O', (1,)), ('firm_ids', 'O', (1,))])
```

5.2.5 pyblp.build_ownership

pyblp.**build_ownership**(*product_data*, *kappa_specification=None*)

Build ownership matrices, O.

Ownership or product holding matrices are defined by their cooperation matrix counterparts, κ. For each market t, $\mathscr{H}_{jk} = \kappa_{fg}$ where $j \in J_{ft}$, the set of products produced by firm f in the market, and similarly, $g \in J_{gt}$.

Parameters

- **product_data** (*structured array-like*) – Each row corresponds to a product. Markets can have differing numbers of products. The following fields are required (except for `firm_ids` when `kappa_specification` is one of the special cases):
  - **market_ids** : (*object*) - IDs that associate products with markets.
  - **firm_ids** : (*object*) - IDs that associate products with firms. This field is ignored if `kappa_specification` is one of the special cases and not a function.
- **kappa_specification** (*str or callable, optional*) – Specification for each market's cooperation matrix, κ, which can either be a general function or a string that implements a special case. The general function is is of the following form:

```
kappa(f, g) -> value
```

  where `value` is $\mathscr{H}_{jk}$ and both `f` and `g` are firm IDs from the `firm_ids` field of `product_data`.

  The default specification, `lambda: f, g: int(f == g)`, constructs traditional ownership matrices. That is, $\kappa = I$, the identify matrix, implies that $\mathscr{H}_{jk}$ is 1 if the same firm produces products j and k, and is 0 otherwise.

  If `firm_ids` happen to be indices for an actual κ matrix, `lambda f, g: kappa[f, g]` will build ownership matrices according to the matrix `kappa`.

  When one of the special cases is specified, `firm_ids` in `product_data` are not required and if specified will be ignored:
  - `'monopoly'` - Monopoly ownership matrices are all ones: $\mathscr{H}_{jk} = 1$ for all j and k.
  - `'single'` - Single product firm ownership matrices are identity matrices: $\mathscr{H}_{jk} = 1$ if $j = k$ and 0 otherwise.

Returns Stacked $J_t \times J_t$ ownership matrices, $\mathscr{H}$, for each market t. If a market has fewer products than others, extra columns will contain `numpy.nan`.

Return type *ndarray*

Examples

The online version of the following section may be easier to read.

Building Ownership Matrices Example

```
import pyblp
import numpy as np

np.set_printoptions(threshold=100)
pyblp.__version__
```

```
'1.2.0'
```

In this example, we'll use the IDs created in the *building ID data example* to build a stack of standard ownership matrices. We'll delete the first data row to demonstrate what ownership matrices should look like when markets have varying numbers of products.

```
id_data = pyblp.build_id_data(T=2, J=5, F=4)
id_data = id_data[1:]
standard_ownership = pyblp.build_ownership(id_data)
standard_ownership
```

```
array([[ 1.,  0.,  0.,  0., nan],
       [ 0.,  1.,  0.,  0., nan],
       [ 0.,  0.,  1.,  0., nan],
       [ 0.,  0.,  0.,  1., nan],
       [ 1.,  1.,  0.,  0.,  0.],
       [ 1.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

We'll now modify the default κ specification so that the elements associated with firm IDs `0` and `1` are equal to `0.5`.

```
def kappa_specification(f, g):
    if f == g:
        return 1
    return 0.5 if f < 2 and g < 2 else 0
```

We can use this specification to build a stack of alternative ownership matrices.

```
alternative_ownership = pyblp.build_ownership(id_data, kappa_specification)
alternative_ownership
```

```
array([[1. , 0.5, 0. , 0. , nan],
       [0.5, 1. , 0. , 0. , nan],
       [0. , 0. , 1. , 0. , nan],
       [0. , 0. , 0. , 1. , nan],
       [1. , 1. , 0.5, 0. , 0. ],
       [1. , 1. , 0.5, 0. , 0. ],
       [0.5, 0.5, 1. , 0. , 0. ],
       [0. , 0. , 0. , 1. , 0. ],
       [0. , 0. , 0. , 0. , 1. ]])
```

In addition to specifying a custom function, there are also a couple of special case strings that efficiently construct monopoly and single-product firm ownership matrices.

```
monopoly_ownership = pyblp.build_ownership(id_data, 'monopoly')
monopoly_ownership
```

```
array([[ 1.,  1.,  1.,  1., nan],
       [ 1.,  1.,  1.,  1., nan],
       [ 1.,  1.,  1.,  1., nan],
       [ 1.,  1.,  1.,  1., nan],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

```
single_ownership = pyblp.build_ownership(id_data, 'single')
single_ownership
```

```
array([[ 1.,  0.,  0.,  0., nan],
       [ 0.,  1.,  0.,  0., nan],
       [ 0.,  0.,  1.,  0., nan],
       [ 0.,  0.,  0.,  1., nan],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

5.2.6 pyblp.build_integration

pyblp.**build_integration**(*integration*, *dimensions*)

Build nodes and weights for integration over agent choice probabilities.

This function can be used to build custom `agent_data` for *Problem* initialization. Specifically, this function affords more flexibility than passing an *Integration* configuration directly to *Problem*. For example, if agents have unobserved tastes over only a subset of demand-side nonlinear product characteristics (i.e., if `sigma` in *Problem.solve()* has columns of zeros), this function can be used to build agent data with fewer columns of integration nodes than the number of nonlinear product characteristics, K_2. This function can also be used to construct nodes that can be transformed into demographic variables.

To build nodes and weights for multiple markets, this function can be called multiple times, once for each market.

Parameters

- **integration** (*Integration*) – *Integration* configuration for how to build nodes and weights for integration.
- **dimensions** (*int*) – Number of dimensions over which to integrate, or equivalently, the number of columns of integration nodes. When an *Integration* configuration is passed directly to *Problem*, this is the number of demand-side nonlinear product characteristics, K_2.

Returns

Nodes and weights for integration over agent utilities. Fields:

- **weights** : (*numeric*) - Integration weights, w.
- **nodes** : (*numeric*) - Unobserved agent characteristics called integration nodes, ν.

Return type *recarray*

Examples

The online version of the following section may be easier to read.

Building Nodes and Weights for Integration Example

```
import pyblp

pyblp.__version__
```

```
'1.2.0'
```

In this example, we'll build nodes and weights for integration over agent choice probabilities according to a *Integration* configuration. We'll construct a sparse grid of nodes and weights according to a level-5 Gauss-Hermite quadrature rule.

```
integration = pyblp.Integration('grid', 5)
integration
```

```
Configured to construct nodes and weights in a sparse grid according to the level-5 Gauss-Hermite rule with options {}.
```

Usually, this configuration should be passed directly to *Problem*, which will create a sparse grid of dimension K_2, the number of demand-side nonlinear product characteristics. Alternatively, we can build the sparse grid ourselves and pass the constructed agent data to *Problem*, possibly after modifying the nodes and weights. If we want to allow agents to have heterogeneous tastes over 2 product characteristics, we'll need a grid of dimension 2.

```
agent_data = pyblp.build_integration(integration, 2)
agent_data.nodes.shape
```

```
(53, 2)
```

```
agent_data.weights.shape
```

```
(53, 1)
```

If we wanted to construct nodes and weights for each market, we could call *build_integration* once for each market, add a column of market IDs, and stack the arrays.

5.2.7 pyblp.data_to_dict

pyblp.**data_to_dict**(*data*, *ignore_empty=True*)

Convert a NumPy record array into a dictionary.

Most data in PyBLP are structured as NumPy record arrays (e.g., *Problem.products* and *SimulationResults.product_data*) which can be cumbersome to work with when working with data types that can't represent matrices, such as the pandas.DataFrame.

This function converts record arrays created by PyBLP into dictionaries that map field names to one-dimensional arrays. Matrices in the original record array (e.g., demand_instruments) are split into as many fields as there are columns (e.g., demand_instruments0, demand_instruments1, and so on).

Parameters

- **data** (*recarray*) – Record array created by PyBLP.
- **ignore_empty** (*bool, optional*) – Whether to ignore matrices with zero size. By default, these are ignored.

Returns The data re-structured as a dictionary.

Return type *dict*

Examples

The online version of the following section may be easier to read.

Converting Data into a Dictionary Example

```
import pyblp
import numpy as np
import pandas as pd

np.set_printoptions(precision=1)
pyblp.options.digits = 2
pyblp.options.verbose = False
pyblp.__version__
```

```
'1.2.0'
```

In this example, we'll convert a dataset constructed by PyBLP into a dictionary that can more easily ingested by other Python packages. Note that you can also pickle most PyBLP objects, which may be more convenient.

First we'll initialize a *Problem* with the fake cereal data from *Nevo (2000a)*.

```
product_data = pd.read_csv(pyblp.data.NEVO_PRODUCTS_LOCATION)
formulation = pyblp.Formulation('0 + prices', absorb='C(product_ids)')
problem = pyblp.Problem(formulation, product_data)
problem
```

```
Dimensions:
================================
 T    N     F    K1    MD    ED
---  ----  ---  ----  ----  ----
94   2256   5    1     20    1
================================

Formulations:
==================================
     Column Indices:          0
-------------------------  ------
X1: Linear Characteristics  prices
==================================
```

The *Problem.products* attribute is a typical example of the type of NumPy record array that PyBLP uses to structure data throughout the package.

```
problem.products
```

```
rec.array([(['C01Q1'], [1], ['F1B04'], [], [], ['F1B04'], [], [], [], [0.], [-2.5e-01,  4.1e-02, -1.6e+00, -2.7e-01, -1.
↪0e-02,  6.9e-03, -9.2e-01,  5.1e-03,  1.3e-01,  2.8e-01,  2.0e-01,  2.5e-01, -4.1e-03, -3.6e-02,  7.1e-02,  1.2e-02, ␣
↪1.7e-02, -1.5e-02,  8.1e-02, -1.6e-02], [], [], [-0.], [], [], [0.1]),
           (['C01Q1'], [1], ['F1B06'], [], [], ['F1B06'], [], [], [], [0.], [-2.1e-01,  5.7e-02, -1.0e+01,  1.5e-01,  4.
↪0e-02,  6.1e-03,  1.1e+00,  8.6e-02,  1.1e-01, -2.7e-02, -1.2e+00, -1.3e-01,  2.6e-03, -6.8e-03, -4.5e-02,  6.7e-05, ␣
↪3.1e-02,  5.8e-03, -3.2e-02, -1.1e-02], [], [], [-0.], [], [], [0.1]),
           (['C01Q1'], [1], ['F1B07'], [], [], ['F1B07'], [], [], [], [0.], [-2.1e-01,  4.6e-02, -2.3e+00, -3.0e-02,  2.
↪4e-03, -1.3e-02,  3.3e-01, -1.7e-01, -2.3e-01,  3.1e-01,  1.0e+00,  2.0e-01,  9.9e-04,  1.8e-02,  8.2e-02,  3.5e-02, ␣
↪2.8e-02,  1.3e-02,  4.7e-02,  2.7e-02], [], [], [ 0.], [], [], [0.1]),
           ...,
           (['C65Q2'], [4], ['F4B10'], [], [], ['F4B10'], [], [], [], [0.], [-1.2e-01, -3.2e-04, -1.1e+00,  1.8e-01,  3.
↪6e-02, -1.9e-02,  2.4e-01,  5.4e-02, -3.2e-01,  8.7e-02,  2.7e+00,  1.6e-01,  8.8e-04,  3.8e-02,  1.9e-02, -5.2e-02, -
↪1.8e-02,  3.7e-02, -5.8e-02,  3.6e-02], [], [], [ 0.], [], [], [0.1]),
           (['C65Q2'], [4], ['F4B12'], [], [], ['F4B12'], [], [], [], [0.], [-2.0e-01,  3.3e-04, -5.1e-01, -4.5e-03,  3.
↪2e-02,  6.1e-03,  5.7e-01,  2.3e-02,  1.1e-01,  1.9e-01,  2.1e+00,  1.3e-01, -8.1e-03, -1.2e-02, -3.6e-02, -4.3e-03, -
↪1.7e-02, -6.6e-03,  7.2e-03, -1.5e-02], [], [], [-0.], [], [], [0.1]),
           (['C65Q2'], [6], ['F6B18'], [], [], ['F6B18'], [], [], [], [0.], [-1.4e-01,  3.5e-03, -2.9e-01,  2.9e-01,  3.
↪9e-02,  2.0e-02, -1.9e+00, -4.0e-02,  3.8e-01,  1.1e-01,  3.4e+00,  1.1e-01, -6.1e-03, -1.2e-03, -4.7e-02, -2.4e-02, -
↪2.1e-02, -2.9e-02, -2.6e-02, -2.5e-02], [], [], [-0.], [], [], [0.1])],
          dtype=[('market_ids', 'O', (1,)), ('firm_ids', 'O', (1,)), ('demand_ids', 'O', (1,)), ('supply_ids', 'O', (0,
↪)), ('nesting_ids', 'O', (0,)), ('product_ids', 'O', (1,)), ('clustering_ids', 'O', (0,)), ('lag_indices', '<i8', (0,
↪)), ('ownership', '<f8', (0,)), ('shares', '<f8', (1,)), ('ZD', '<f8', (20,)), ('ZS', '<f8', (0,)), ('ZC', '<f8', (0,
↪)), (((prices,), 'X1'), '<f8', (1,)), ((((), 'X2'), '<f8', (0,)), ((((), 'X3'), '<f8', (0,)), ('prices', '<f8', (1,))])
```

This is hard to read, and if we try to convert it into a `pandas.DataFrame`, we'll get an error. This is because `pandas.DataFrame` doesn't support matrices.

Instead, we'll use the *data_to_dict* function to first convert the record array into a dictionary, which can be easily ingested by Pandas. Matrices are converted into multiple fields, one for each column.

```
x = pyblp.data_to_dict(problem.products)
print({k: v.size for k, v in x.items()})

df = pd.DataFrame(pyblp.data_to_dict(problem.products))
df
```

```
{'market_ids': 2256, 'firm_ids': 2256, 'demand_ids': 2256, 'product_ids': 2256, 'shares': 2256, 'ZD0': 2256, 'ZD1': ␣
↪2256, 'ZD2': 2256, 'ZD3': 2256, 'ZD4': 2256, 'ZD5': 2256, 'ZD6': 2256, 'ZD7': 2256, 'ZD8': 2256, 'ZD9': 2256, 'ZD10': ␣
↪2256, 'ZD11': 2256, 'ZD12': 2256, 'ZD13': 2256, 'ZD14': 2256, 'ZD15': 2256, 'ZD16': 2256, 'ZD17': 2256, 'ZD18': 2256, ␣
↪'ZD19': 2256, 'X1': 2256, 'prices': 2256}
```

(continues on next page)

(continued from previous page)

```
     market_ids firm_ids demand_ids product_ids    shares       ZD0       ZD1  \
0          C01Q1        1      F1B04       F1B04  0.012417 -0.249518  0.040943
1          C01Q1        1      F1B06       F1B06  0.007809 -0.205951  0.057100
2          C01Q1        1      F1B07       F1B07  0.012995 -0.212031  0.046246
3          C01Q1        1      F1B09       F1B09  0.005770 -0.170725  0.049143
4          C01Q1        1      F1B11       F1B11  0.017934 -0.164983  0.047168
...          ...      ...        ...         ...       ...       ...       ...
2251       C65Q2        3      F3B14       F3B14  0.024702 -0.126940  0.002240
2252       C65Q2        4      F4B02       F4B02  0.007914 -0.109756  0.011192
2253       C65Q2        4      F4B10       F4B10  0.002229 -0.119689 -0.000324
2254       C65Q2        4      F4B12       F4B12  0.011463 -0.201890  0.000334
2255       C65Q2        6      F6B18       F6B18  0.026208 -0.139453  0.003468

             ZD2       ZD3       ZD4  ...      ZD12      ZD13      ZD14  \
0     -1.577566 -0.269073 -0.010004  ... -0.004142 -0.035593  0.070587
1    -10.383954  0.150476  0.039816  ...  0.002585 -0.006776 -0.045453
2     -2.278160 -0.029976  0.002390  ...  0.000992  0.018425  0.081555
3     -1.159784 -0.244789  0.002848  ... -0.004274  0.026440  0.064169
4     -4.737563 -0.070873  0.012273  ... -0.004694 -0.029179 -0.000454
...         ...       ...       ...  ...       ...       ...       ...
2251  -1.067171  0.150626  0.037091  ... -0.004787 -0.012775 -0.059399
2252   0.458133  0.066193  0.006838  ...  0.009385  0.037487  0.086225
2253  -1.109521  0.175027  0.036227  ...  0.000884  0.037634  0.019278
2254  -0.507311 -0.004538  0.031569  ... -0.008093 -0.011750 -0.036333
2255  -0.285143  0.291132  0.039259  ... -0.006138 -0.001181 -0.046888

          ZD15      ZD16      ZD17      ZD18      ZD19        X1    prices
0     0.011768  0.017287 -0.015031  0.081201 -0.015833 -0.011248  0.072088
1     0.000067  0.031229  0.005841 -0.032121 -0.010614 -0.007135  0.114178
2     0.034975  0.027932  0.013156  0.047484  0.026800  0.023678  0.132391
3     0.021496  0.032372  0.033063  0.045501  0.036154  0.029725  0.130344
4    -0.045272 -0.025446 -0.006794 -0.007560 -0.011364 -0.015585  0.154823
...        ...       ...       ...       ...       ...       ...       ...
2251  0.043775  0.059339 -0.021934  0.034592 -0.021052 -0.017337  0.126086
2252  0.060856  0.028264  0.051264  0.032965  0.033324  0.044542  0.199167
2253 -0.052403 -0.018107  0.036733 -0.057647  0.035662  0.033720  0.137017
2254 -0.004333 -0.017427 -0.006647  0.007228 -0.015403 -0.004174  0.100174
2255 -0.023637 -0.021410 -0.029402 -0.025971 -0.025435 -0.011956  0.127557
```

(continues on next page)

(continued from previous page)

```
[2256 rows x 27 columns]
```

5.2.8 pyblp.save_pickle

pyblp.**save_pickle**(*x*, *path*)
Save an object as a pickle file.

This is a simple wrapper around *pickle.dump*.

Parameters

- **x** (*object*) – Object to be pickled.
- **path** (*str or Path*) – File path to which the object will be saved.

5.2.9 pyblp.read_pickle

pyblp.**read_pickle**(*path*)
Load a pickled object into memory.

This is a simple wrapper around *pickle.load*.

Parameters **path** (*str or Path*) – File path of a pickled object.

Returns The unpickled object.

Return type *object*

5.3 Problem Class

Given data and appropriate configurations, a BLP-type problem can be structured by initializing the following class.

Problem(product_formulations, product_data)	A BLP-type problem.

5.3.1 pyblp.Problem

class pyblp.**Problem**(*product_formulations*, *product_data*, *agent_formulation=None*, *agent_data=None*, *integration=None*, *rc_types=None*, *epsilon_scale=1.0*, *costs_type='linear'*, *add_exogenous=True*)
A BLP-type problem.

This class is initialized with relevant data and solved with *Problem.solve()*.

Parameters

- **product_formulations** (*Formulation or sequence of Formulation*) – *Formulation* configuration or a sequence of up to three *Formulation* configurations for the matrix of demand-side linear product characteristics, X_1, for the matrix of demand-side nonlinear product characteristics, X_2, and for the matrix of supply-side characteristics, X_3, respectively. If the formulation for X_3 is not specified or is `None`, a supply side will not be estimated. Similarly, if the formulation for X_2 is not specified or is `None`, the logit (or nested logit) model will be estimated.

  Variable names should correspond to fields in `product_data`. The `shares` variable should not be included in the formulations for X_1 or X_2. The formulation for X_3 can include shares to allow marginal costs to depend on quantity.

The `prices` variable should not be included in the formulation for X_3, but it should be included in the formulation for X_1 or X_2 (or both). The `absorb` argument of *`Formulation`* can be used to absorb fixed effects into X_1 and X_3, but not X_2. Characteristics in X_2 should generally be included in X_1. The typical exception is characteristics that are collinear with fixed effects that have been absorbed into X_1.

By default, characteristics in X_1 that do not involve `prices`, X_1^{ex}, will be combined with excluded demand-side instruments (specified below) to create the full set of demand-side instruments, Z_D. Any fixed effects absorbed into X_1 will also be absorbed into Z_D. Similarly, characteristics in X_3 that do not involve `shares`, X_3^{ex}, will be combined with the excluded supply-side instruments to create Z_S, and any fixed effects absorbed into X_3 will also be absorbed into Z_S. The `add_exogenous` flag can be used to disable this behavior.

> **Warning:** Characteristics that involve prices, p, or shares, s, should always be formulated with the `prices` and `shares` variables, respectively. If another name is used, *`Problem`* will not understand that the characteristic is endogenous, so it will be erroneously included in Z_D or Z_S, and derivatives computed with respect to prices or shares will likely be wrong. For example, to include a p^2 characteristic, include `I(prices**2)` in a formula instead of manually constructing and including a `prices_squared` variable.

- **`product_data`** (*structured array-like*) – Each row corresponds to a product. Markets can have differing numbers of products. The following fields are required:
  - **market_ids** : (*object*) - IDs that associate products with markets.
  - **shares** : (*numeric*) - Market shares, s, which should be between zero and one, exclusive. Outside shares should also be between zero and one. Shares in each market should sum to less than one.
  - **prices** : (*numeric*) - Product prices, p.

  If a formulation for X_3 is specified in `product_formulations`, firm IDs are also required, since they will be used to estimate the supply side of the problem:

  - **firm_ids** : (*object, optional*) - IDs that associate products with firms.

  Excluded instruments are typically specified with the following fields:

  - **demand_instruments** : (*numeric*) - Excluded demand-side instruments, which, together with the formulated exogenous demand-side linear product characteristics, X_1^{ex}, constitute the full set of demand-side instruments, Z_D. To instead specify the full matrix Z_D, set `add_exogenous` to `False`.
  - **supply_instruments** : (*numeric, optional*) - Excluded supply-side instruments, which, together with the formulated exogenous supply-side characteristics, X_3^{ex}, constitute the full set of supply-side instruments, Z_S. To instead specify the full matrix Z_S, set `add_exogenous` to `False`.
  - **covariance_instruments** : (*numeric, optional*) - Covariance instruments Z_C. If specified, additional moments $E[g_{C,jt}] = E[\xi_{jt}\omega_{jt}Z_{C,jt}] = 0$ will be added, as in *MacKay and Miller (2025)*. The default 2SLS weighting matrix will have an additional $(Z_C'Z_C/N)^{-1}$ block after the first two.

    Note: Using covariance restrictions to identify a parameter on price can sometimes yield two solutions, where the "upper" solution may be positive (i.e., implying upward-sloping demand). See *MacKay and Miller (2025)* for more discussion of this point. In these cases

when the "lower" root is the correct solution, consider imposing a one-sided bound (e.g., zero) on the parameter on price to ensure the appropriate sign using `beta_bounds` (if the parameter is in `beta`) or replacing it with a lognormal coefficient on price via the `rc_type` argument to *`Problem`*.

Note: In the current implementation, these covariance restrictions only affect the nonlinear parameters. The linear parameters are estimated using other moments. In the case of overidentification, the estimator may not be fully efficient because of this implementation decision.

If `firm_ids` are specified, custom ownership matrices can be specified as well:

– **ownership** : (*numeric, optional*) - Custom stacked $J_t \times J_t$ ownership or product holding matrices, $\mathscr{H}$, for each market t, which can be built with *`build_ownership()`*. By default, standard ownership matrices are built only when they are needed to reduce memory usage. If specified, there should be as many columns as there are products in the market with the most products. Rightmost columns in markets with fewer products will be ignored.

Note: Fields that can have multiple columns (`demand_instruments`, `supply_instruments`, and `ownership`) can either be matrices or can be broken up into multiple one-dimensional fields with column index suffixes that start at zero. For example, if there are three columns of excluded demand-side instruments, a `demand_instruments` field with three columns can be replaced by three one-dimensional fields: `demand_instruments0`, `demand_instruments1`, and `demand_instruments2`.

To estimate a nested logit or random coefficients nested logit (RCNL) model, nesting groups must be specified:

– **nesting_ids** (*object, optional*) - IDs that associate products with nesting groups. When these IDs are specified, `rho` must be specified in *`Problem.solve()`* as well.

It may be convenient to define IDs for different products:

– **product_ids** (*object, optional*) - IDs that identify products within markets. There can be multiple columns.

To estimate unobservable autocorrelation with `phi` in *`Problem.solve()`*, indices that define lags of the data must be specified:

– **lag_indices** : (*int, optional*) - Indices that take on values from 0 to $N - 1$, which define the lag operator L on the data. For example, if markets t are simply time periods and the identity of products j are persistent across periods, then $Lx_{jt} = x_{j,t-1}$.

The value of the current row index indicates that this is the initial period for a product. Otherwise, the value should correspond to the row that is the lagged version of the current row.

Finally, clustering groups can be specified to account for within-group correlation while updating the weighting matrix and estimating standard errors:

– **clustering_ids** (*object, optional*) - Cluster group IDs, which will be used if `W_type` or `se_type` in *`Problem.solve()`* is `'clustered'`.

Along with `market_ids`, `firm_ids`, `nesting_ids`, `product_ids`, `clustering_ids`, and `prices`, the names of any additional fields can typically be used as variables in `product_formulations`. However, there are a few variable names such as `'X1'`, which are reserved for use by *`Products`*.

- **`agent_formulation`** (*Formulation, optional*) – *`Formulation`* configuration for the matrix of observed agent characteristics called demographics, d, which will only be included in the model if this formulation is specified. Since demographics are only used if there are demand-side nonlinear product characteristics, this formulation should only be specified if X_2 is formulated in `product_formulations`. Variable names should correspond to fields in `agent_data`. See the information under `agent_data` for how to give fields for product-specific demographics d_{ijt}.
- **`agent_data`** (*structured array-like, optional*) – Each row corresponds to an agent. Markets can have differing numbers of agents. Since simulated agents are only used if there are demand-side nonlinear product characteristics, agent data should only be specified if X_2 is formulated in `product_formulations`. If agent data are specified, market IDs are required:

  – **market_ids** : (*object*) - IDs that associate agents with markets. The set of distinct IDs should be the same as the set in `product_data`. If `integration` is specified, there must be at least as many rows in each market as the number of nodes and weights that are built for the market.

  If `integration` is not specified, the following fields are required:

  – **weights** : (*numeric, optional*) - Integration weights, w, for integration over agent choice probabilities.

  – **nodes** : (*numeric, optional*) - Unobserved agent characteristics called integration nodes, ν. If there are more than K_2 columns (the number of demand-side nonlinear product characteristics), only the first K_2 will be retained. If any columns of `sigma` in *`Problem.solve()`* are fixed at zero, only the first few columns of these nodes will be used.

  The convenience function *`build_integration()`* can be useful when constructing custom nodes and weights.

  Note: If `nodes` has multiple columns, it can be specified as a matrix or broken up into multiple one-dimensional fields with column index suffixes that start at zero. For example, if there are three columns of nodes, a `nodes` field with three columns can be replaced by three one-dimensional fields: `nodes0`, `nodes1`, and `nodes2`.

  It may be convenient to define IDs for different agents:

  – **agent_ids** (*object, optional*) - IDs that identify agents within markets. There can be multiple of the same ID within a market.

  Along with `market_ids` and `agent_ids`, the names of any additional fields can be typically be used as variables in `agent_formulation`. Exceptions are the names `'demographics'` and `'availability'`, which are reserved for use by *`Agents`*.

  In addition to standard demographic variables d_{it}, it is also possible to specify product-specific demographics d_{ijt}. A typical example is geographic distance of agent i from product j. If `agent_formulation` has, for example, `'distance'`, instead of including a single `'distance'` field in `agent_data`, one should instead include `'distance0'`, `'distance1'`, `'distance2'` and so on, where the index corresponds to the order in which products appear within market in `product_data`. For example, `'distance5'` should measure the distance of agents to the fifth product within the market, as ordered in

`product_data`. The last index should be the number of products in the largest market, minus one. For markets with fewer products than this maximum number, latter columns will be ignored.

Finally, by default each agent i in market t is faced with the same choice set of product j, but it is possible to specify agent-specific availability a_{ijt} much in the same way that product-specific demographics are specified. To do so, the following field can be specified:

– **availability** : (*numeric, optional*) - Agent-specific product availability, a. Choice probabilities in (3.5) are modified according to

$$s_{ijt} = \frac{a_{ijt} \exp V_{ijt}}{1 + \sum_{k \in J_t} a_{ijt} \exp V_{ikt}}, \tag{5.6}$$

and similarly for the nested logit model and consumer surplus calculations. By default, all $a_{ijt} = 1$. To have a product j be unavailable to agent i, set $a_{ijt} = 0$.

Agent-specific availability is specified in the same way that product-specific demographics are specified. In `agent_data`, one can include `'availability0'`, `'availability1'`, `'availability2'`, and so on, where the index corresponds to the order in which products appear within market in `product_data`. The last index should be the number of products in the largest market, minus one. For markets with fewer products than this maximum number, latter columns will be ignored.

- **`integration`** (*Integration, optional*) – *`Integration`* configuration for how to build nodes and weights for integration over agent choice probabilities, which will replace any `nodes` and `weights` fields in `agent_data`. This configuration is required if `nodes` and `weights` in `agent_data` are not specified. It should not be specified if X_2 is not formulated in `product_formulations`.

  If this configuration is specified, K_2 columns of nodes (the number of demand-side nonlinear product characteristics) will be built. However, if `sigma` in *`Problem.solve()`* is left unspecified or specified with columns fixed at zero, fewer columns will be used.

- **`rc_types`** (*sequence of str, optional*) – Random coefficient types:

  – `'linear'` (default) - The random coefficient is as defined in (3.3). All elliptical distributions are supported, including the normal distribution.

  – `'log'` - The random coefficient's column in (3.3) is exponentiated before being pre-multiplied by X_2. It will take on values bounded from below by zero. All log-elliptical distributions are supported, including the lognormal distribution.

  – `'logit'` - The random coefficient's column in (3.3) is passed through the inverse logit function before being pre-multiplied by X_2. It will take on values bounded from below by zero and above by one.

  The list should have as many strings as there are columns in X_2. Each string determines the type of the random coefficient on the corresponding product characteristic in X_2.

  A typical example of when to use `'log'` is to have a lognormal coefficient on prices. Implementing this typically involves having an `I(-prices)` in the formulation for X_2, and instead of including `prices` in X_1, including a `1` in the `agent_formulation`. Then the corresponding coefficient in Π will serve as the mean parameter for the lognormal random coefficient on negative prices, $-p_{jt}$.

- **`epsilon_scale`** (*float, optional*) – Factor by which the Type I Extreme Value idiosyncratic preference term, ϵ_{ijt}, is scaled. By default, ϵ_{ijt} is not scaled. The typical use of this parameter is to approximate the pure characteristics model of *Berry and Pakes (2007)* by choosing a value smaller than `1.0`. As this scaling factor approaches zero, the model approaches the pure characteristics model in which there is no idiosyncratic preference term.

In practice, this is implemented by dividing $V_{ijt} = \delta_{jt} + \mu_{ijt}$ by the scaling factor when solving for the mean utility δ_{jt}. For small scaling factors, this leads to large values of V_{ijt}, which when exponentiated in the logit expression can lead to overflow issues discussed in *Berry and Pakes (2007)*. The safe versions of the contraction mapping discussed in the documentation for `fp_type` in *`Problem.solve()`* (which is used by default) eliminate overflow issues at the cost of introducing fewer (but still common for a small scaling factor) underflow issues. Throughout the contraction mapping, some values of the simulated shares $s_{jt}(\delta, \theta)$ can underflow to zero, causing the contraction to fail when taking logs. By default, `shares_bounds` in *`Problem.solve()`* bounds these simulated shares from below by `1e-300`, which eliminates these underflow issues at the cost of making it more difficult for iteration routines to converge.

With this in mind, scaling epsilon is not supported for nonlinear contractions, and is also not supported when there are nesting groups, since these further complicate the problem. In practice, if the goal is to approximate the pure characteristics model, it is a good idea to slowly decrease the scale of epsilon (e.g., starting with `0.5`, trying `0.1`, etc.) until the contraction begins to fail. To further decrease the scale, there are a few things that can help. One is passing a different *`Iteration`* configuration to `iteration` in *`Problem.solve()`*, such as `'lm'`, which can be robust in this situation. Another is to set `pyblp.options.dtype = np.longdouble` when on a system that supports extended precision (see *`options`* for more information about this) and choose a smaller lower bound by configuring `shares_bounds` in *`Problem.solve()`*. Ultimately the model will stop being solvable at a certain point, and this point will vary by problem, so approximating the pure characteristics model requires some degree of experimentation.

- **`costs_type`** (*str, optional*) – Functional form of the marginal cost function $\tilde{c} = f(c)$ in (3.9). The following specifications are supported:
  - `'linear'` (default) - Linear specification: $\tilde{c} = c$.
  - `'log'` - Log-linear specification: $\tilde{c} = \log c$.

  This specification is only relevant if X_3 is formulated.

- **`add_exogenous`** (*bool, optional*) – Whether to add characteristics in X_1 that do not involve prices, X_1^{ex}, to the `demand_instruments` field in `product_data` (including absorbed fixed effects), and similarly, whether to add characteristics in X_3 that do not involve shares, X_3^{ex}, to the `supply_instruments` field. This is by default `True` so that only excluded instruments need to be specified.

  If this is set to `False`, `demand_instruments` and `supply_instruments` should specify the full sets of demand- and supply-side instruments, Z_D and Z_S, and fixed effects should be manually absorbed (for example, with the *`build_matrix()`* function). This behavior can be useful, for example, when price is not the only endogenous product characteristic over which consumers have preferences. This model could be correctly estimated by manually adding the truly exogenous characteristics in X_1 to Z_D.

  Warning: If this flag is set to `False` because there are multiple endogenous product characteristics, care should be taken when including a supply side or computing optimal instruments. These routines assume that price is the only endogenous variable over which consumers have preferences.

`product_formulations`

`Formulation` configurations for X_1, X_2, and X_3, respectively.

Type *Formulation or sequence of Formulation*

agent_formulation
Formulation configuration for d.

Type *Formulation*

products
Product data structured as *Products*, which consists of data taken from `product_data` along with matrices built according to *Problem.product_formulations*. The *data_to_dict()* function can be used to convert this into a more usable data type.

Type *Products*

agents
Agent data structured as *Agents*, which consists of data taken from `agent_data` or built by `integration` along with any demographics built according to *Problem.agent_formulation*. The *data_to_dict()* function can be used to convert this into a more usable data type.

Type *Agents*

unique_market_ids
Unique market IDs in product and agent data.

Type *ndarray*

unique_firm_ids
Unique firm IDs in product data.

Type *ndarray*

unique_nesting_ids
Unique nesting group IDs in product data.

Type *ndarray*

unique_product_ids
Unique product IDs in product data.

Type *ndarray*

unique_agent_ids
Unique agent IDs in agent data.

Type *ndarray*

rc_types
Random coefficient types.

Type *list of str*

epsilon_scale
Factor by which the Type I Extreme Value idiosyncratic preference term, ϵ_{ijt}, is scaled.

Type *float*

costs_type
Functional form of the marginal cost function $\tilde{c} = f(c)$.

Type *str*

T
Number of markets, T.

Type *int*

N
Number of products across all markets, N.

Type *int*

F

Number of firms across all markets, F.

Type *int*

I

Number of agents across all markets, I.

Type *int*

K1

Number of demand-side linear product characteristics, K_1.

Type *int*

K2

Number of demand-side nonlinear product characteristics, K_2.

Type *int*

K3

Number of supply-side product characteristics, K_3.

Type *int*

D

Number of demographic variables, D.

Type *int*

MD

Number of demand-side instruments, M_D, which is typically the number of excluded demand-side instruments plus the number of exogenous demand-side linear product characteristics, K_1^{ex}.

Type *int*

MS

Number of supply-side instruments, M_S, which is typically the number of excluded supply-side instruments plus the number of exogenous supply-side linear product characteristics, K_3^{ex}.

Type *int*

MC

Number of covariance instruments, M_C.

Type *int*

ED

Number of absorbed dimensions of demand-side fixed effects, E_D.

Type *int*

ES

Number of absorbed dimensions of supply-side fixed effects, E_S.

Type *int*

H

Number of nesting groups, H.

Type *int*

Examples

- *Tutorial*

Methods

solve([sigma, pi, rho, phi, beta, gamma, ...])	Solve the problem.

Once initialized, the following method solves the problem.

Problem.solve([sigma, pi, rho, phi, beta, ...])	Solve the problem.

5.3.2 pyblp.Problem.solve

Problem.**solve**(*sigma=None*, *pi=None*, *rho=None*, *phi=None*, *beta=None*, *gamma=None*, *sigma_bounds=None*, *pi_bounds=None*, *rho_bounds=None*, *phi_bounds=None*, *beta_bounds=None*, *gamma_bounds=None*, *delta=None*, *method='2s'*, *initial_update=None*, *optimization=None*, *scale_objective=True*, *check_optimality='both'*, *finite_differences=False*, *error_behavior='revert'*, *error_punishment=1*, *delta_behavior='first'*, *iteration=None*, *fp_type='safe_linear'*, *shares_bounds=(1e-300, None)*, *costs_bounds=None*, *W=None*, *center_moments=True*, *W_type='robust'*, *se_type='robust'*, *demand_moment_types='levels'*, *supply_moment_types='levels'*, *covariance_moments_mean=0*, *micro_moments=()*, *micro_sample_covariances=None*, *resample_agent_data=None*)

Solve the problem.

The problem is solved in one or more GMM steps. During each step, any parameters in θ are optimized to minimize the GMM objective value, giving the estimated $\hat{\theta}$. If there are no parameters in θ (for example, in the logit model there are no nonlinear parameters and all linear parameters can be concentrated out), the objective is evaluated once during the step.

If there are nonlinear parameters, the mean utility, $\delta(\theta)$ is computed market-by-market with fixed point iteration. Otherwise, it is computed analytically according to the solution of the logit model. If a supply side is to be estimated, marginal costs, $c(\theta)$, are also computed market-by-market. Linear parameters are then estimated, which are used to recover structural error terms, which in turn are used to form the objective value. By default, the objective gradient is computed as well.

Note: This method supports *parallel()* processing. If multiprocessing is used, market-by-market computation of $\delta(\theta)$ (and $\tilde{c}(\theta)$ if a supply side is estimated), along with associated Jacobians, will be distributed among the processes.

Parameters

- **sigma** (*array-like, optional*) – Configuration for which elements in the lower-triangular Cholesky root of the covariance matrix for unobserved taste heterogeneity, Σ, are fixed at zero and starting values for the other elements, which, if not fixed by `sigma_bounds`, are in the vector of unknown elements, θ.

  Rows and columns correspond to columns in X_2, which is formulated according `product_formulations` in *Problem*. If X_2 was not formulated, this should not be specified, since the logit model will be estimated.

Values above the diagonal are ignored. Zeros are assumed to be zero throughout estimation and nonzeros are, if not fixed by `sigma_bounds`, starting values for unknown elements in θ. If any columns are fixed at zero, only the first few columns of integration nodes (specified in *Problem*) will be used.

To have nonzero covariances for only a subset of the random coefficients, the characteristics for those random coefficients with zero covariances should come first in X_2. This can be seen by looking at the expression for $\Sigma\Sigma'$, the actual covariance matrix of the random coefficients.

- **pi** (*array-like, optional*) – Configuration for which elements in the matrix of parameters that measures how agent tastes vary with demographics, Π, are fixed at zero and starting values for the other elements, which, if not fixed by `pi_bounds`, are in the vector of unknown elements, θ.

  Rows correspond to the same product characteristics as in `sigma`. Columns correspond to columns in d, which is formulated according to `agent_formulation` in *Problem*. If d was not formulated, this should not be specified.

  Zeros are assumed to be zero throughout estimation and nonzeros are, if not fixed by `pi_bounds`, starting values for unknown elements in θ.

- **rho** (*array-like, optional*) – Configuration for which elements in the vector of parameters that measure within nesting group correlation, ρ, are fixed at zero and starting values for the other elements, which, if not fixed by `rho_bounds`, are in the vector of unknown elements, θ.

  If this is a scalar, it corresponds to all groups defined by the `nesting_ids` field of `product_data` in *Problem*. If this is a vector, it must have H elements, one for each nesting group. Elements correspond to group IDs in the sorted order of *Problem.unique_nesting_ids*. If nesting IDs were not specified, this should not be specified either.

  Zeros are assumed to be zero throughout estimation and nonzeros are, if not fixed by `rho_bounds`, starting values for unknown elements in θ.

- **phi** (*array-like, optional*) – Configuration for which elements in the vector of parameters that measure unobservable autocorrelation, ϕ, are fixed at zero and starting values for the other elements, which, if not fixed by `phi_bounds`, are in the vector of unknown elements, θ.

  If `lag_indices` in `product_data` were specified, this is either a scalar $\phi = \phi_\xi$ or, if a supply side was specified, a 2×2 matrix

$$\phi = \begin{bmatrix} \phi_\xi & \phi_{\xi\omega} \\ \phi_{\omega\xi} & \phi_\omega \end{bmatrix} \tag{5.7}$$

  corresponding to demand- and supply-side unobservable autocorrelations:

$$\begin{aligned} \xi_{jt} &= \phi_\xi \cdot L\xi_{jt} + \phi_{\xi\omega} \cdot L\omega_{jt} + \tilde{\xi}_{jt}, \\ \omega_{jt} &= \phi_\omega \cdot L\omega_{jt} + \phi_{\omega\xi} \cdot L\xi_{jt} + \tilde{\omega}_{jt}, \end{aligned} \tag{5.8}$$

  where the `lag_indices` field in `product_data` defines the lag operator L.

- **beta** (*array-like, optional*) – Configuration for which elements in the vector of demand-side linear parameters, β, are concentrated out of the problem. Usually, this is left unspecified, unless there is a supply side, in which case parameters on endogenous product characteristics cannot be concentrated out of the problem. Values specify which elements are fixed at zero and starting values for the other elements, which, if not fixed by `beta_bounds`, are in the vector of unknown elements, θ.

Elements correspond to columns in X_1, which is formulated according to `product_formulations` in *Problem*.

Both `None` and `numpy.nan` indicate that the parameter should be concentrated out of the problem. That is, it will be estimated, but does not have to be included in θ. Zeros are assumed to be zero throughout estimation and nonzeros are, if not fixed by `beta_bounds`, starting values for unknown elements in θ.

- **gamma** (*array-like, optional*) – Configuration for which elements in the vector of supply-side linear parameters, γ, are concentrated out of the problem. Usually, this is left unspecified. Values specify which elements are fixed at zero and starting values for the other elements, which, if not fixed by `gamma_bounds`, are in the vector of unknown elements, θ.

  Elements correspond to columns in X_3, which is formulated according to `product_formulations` in *Problem*. If X_3 was not formulated, this should not be specified.

  Both `None` and `numpy.nan` indicate that the parameter should be concentrated out of the problem. That is, it will be estimated, but does not have to be included in θ. Zeros are assumed to be zero throughout estimation and nonzeros are, if not fixed by `gamma_bounds`, starting values for unknown elements in θ.

- **sigma_bounds** (*tuple, optional*) – Configuration for Σ bounds of the form `(lb, ub)`, in which both `lb` and `ub` are of the same size as `sigma`. Each element in `lb` and `ub` determines the lower and upper bound for its counterpart in `sigma`. If `optimization` does not support bounds, these will be ignored. If bounds are supported, the diagonal of `sigma` is by default bounded from below by zero.

  Values above the diagonal are ignored. Lower and upper bounds corresponding to zeros in `sigma` are set to zero. Setting a lower bound equal to an upper bound fixes the corresponding element, removing it from θ. Both `None` and `numpy.nan` are converted to `-numpy.inf` in `lb` and to `numpy.inf` in `ub`.

- **pi_bounds** (*tuple, optional*) – Configuration for Π bounds of the form `(lb, ub)`, in which both `lb` and `ub` are of the same size as `pi`. Each element in `lb` and `ub` determines the lower and upper bound for its counterpart in `pi`. If `optimization` does not support bounds, these will be ignored. By default, `pi` is unbounded.

  Lower and upper bounds corresponding to zeros in `pi` are set to zero. Setting a lower bound equal to an upper bound fixes the corresponding element, removing it from θ. Both `None` and `numpy.nan` are converted to `-numpy.inf` in `lb` and to `numpy.inf` in `ub`.

- **rho_bounds** (*tuple, optional*) – Configuration for ρ bounds of the form `(lb, ub)`, in which both `lb` and `ub` are of the same size as `rho`. Each element in `lb` and `ub` determines the lower and upper bound for its counterpart in `rho`. If `optimization` does not support bounds, these will be ignored.

  If bounds are supported, `rho` is by default bounded from below by `0`, which corresponds to the simple logit model, and bounded from above by `0.99` because values greater than `1` are inconsistent with utility maximization.

  Lower and upper bounds corresponding to zeros in `rho` are set to zero. Setting a lower bound equal to an upper bound fixes the corresponding element, removing it from θ. Both `None` and `numpy.nan` are converted to `-numpy.inf` in `lb` and to `numpy.inf` in `ub`.

- **phi_bounds** (*tuple, optional*) – Configuration for ϕ bounds of the form `(lb, ub)`, in which both `lb` and `ub` are of the same size as `phi`. If `optimization` does not support bounds, these will be ignored.

Setting a lower bound equal to an upper bound fixes this parameter, removing it from θ. Both `None` and `numpy.nan` are converted to `-numpy.inf` in `lb` and to `numpy.inf` in `ub`.

- **`beta_bounds`** (*tuple, optional*) – Configuration for β bounds of the form `(lb, ub)`, in which both `lb` and `ub` are of the same size as `beta`. Each element in `lb` and `ub` determines the lower and upper bound for its counterpart in `beta`. If `optimization` does not support bounds, these will be ignored.

  Usually, this is left unspecified unless there is a supply side, in which case parameters on endogenous product characteristics cannot be concentrated out of the problem. It is generally a good idea to constrain such parameters to be nonzero so that the intra-firm Jacobian of shares with respect to prices does not become singular.

  By default, all non-concentrated out parameters are unbounded. Bounds should only be specified for parameters that are included in θ; that is, those with initial values specified in `beta`.

  Lower and upper bounds corresponding to zeros in `beta` are set to zero. Setting a lower bound equal to an upper bound fixes the corresponding element, removing it from θ. Both `None` and `numpy.nan` are converted to `-numpy.inf` in `lb` and to `numpy.inf` in `ub`.

- **`gamma_bounds`** (*tuple, optional*) – Configuration for γ bounds of the form `(lb, ub)`, in which both `lb` and `ub` are of the same size as `gamma`. Each element in `lb` and `ub` determines the lower and upper bound for its counterpart in `gamma`. If `optimization` does not support bounds, these will be ignored.

  By default, all non-concentrated out parameters are unbounded. Bounds should only be specified for parameters that are included in θ; that is, those with initial values specified in `gamma`.

  Lower and upper bounds corresponding to zeros in `gamma` are set to zero. Setting a lower bound equal to an upper bound fixes the corresponding element, removing it from θ. Both `None` and `numpy.nan` are converted to `-numpy.inf` in `lb` and to `numpy.inf` in `ub`.

- **`delta`** (*array-like, optional*) – Initial values for the mean utility, δ. If there are any nonlinear parameters, these are the values at which the fixed point iteration routine will start during the first objective evaluation. By default, the solution to the logit model in (3.45) is used. If ρ is specified, the solution to the nested logit model in (3.46) under the initial `rho` is used instead.

- **`method`** (*str, optional*) – The estimation routine that will be used. The following methods are supported:

  - `'1s'` - One-step GMM.
  - `'2s'` (default) - Two-step GMM.

  Iterated GMM can be manually implemented by executing single GMM steps in a loop, in which after the first iteration, nonlinear parameters and weighting matrices from the last *`ProblemResults`* are passed as arguments.

- **`initial_update`** (*bool, optional*) – Whether to update starting values for the mean utility δ and the weighting matrix W at the initial parameter values before the first GMM step. This initial update will be called a zeroth step.

  By default, an initial update will not be used unless `micro_moments` are specified without an initial weighting matrix `W`.

Note: When trying multiple parameter starting values to verify that the optimization routine

converges to the same optimum, using `initial_update` is not recommended because different weighting matrices will be used for these different runs. A better option is to use `optimization=Optimization('return')` at the best guess for parameter values and pass *`ProblemResults.updated_W`* to `W` for each set of different parameter starting values.

- **`optimization`** (*Optimization, optional*) – *`Optimization`* configuration for how to solve the optimization problem in each GMM step, which is only used if there are unfixed nonlinear parameters over which to optimize. By default, `Optimization('l-bfgs-b', {'ftol': 0, 'gtol': 1e-8})` is used. If available, `Optimization('knitro')` may be preferable. Generally, it is recommended to consider a number of different optimization routines and starting values, verifying that $\hat{\theta}$ satisfies both the first and second order conditions. Choosing a routine that supports bounds (and configuring bounds) is typically a good idea. Choosing a routine that does not use analytic gradients will often down estimation.

- **`scale_objective`** (*bool, optional*) – Whether to scale the objective in (3.10) by N, the number of observations, in which case the objective after two GMM steps is equal to the J statistic from *Hansen (1982)*. By default, the objective is scaled by N.

  In theory the scale of the objective should not matter, but in practice having similar objective values for different problem sizes is helpful because similar optimization tolerances can be used.

- **`check_optimality`** (*str, optional*) – How to check for optimality (first and second order conditions) after the optimization routine finishes. The following configurations are supported:

  - `'gradient'` - Analytically compute the gradient after optimization finishes, but do not compute the Hessian. Since Jacobians needed to compute standard errors will already be computed, gradient computation will not take a long time. This option may be useful if Hessian computation takes a long time when, for example, there are a large number of parameters.

  - `'both'` (default) - Also compute the Hessian with central finite differences after optimization finishes.

- **`finite_differences`** (*bool, optional*) – Whether to use finite differences to compute Jacobians and the gradient instead of analytic expressions. Since finite differences comes with numerical approximation error and is typically slower, analytic expressions are used by default.

  One situation in which finite differences may be preferable is when there are a sufficiently large number of products and integration nodes in individual markets to make computing analytic Jacobians infeasible because of memory requirements. Note that an analytic expression for the Hessian has not been implemented, so when computed it is always approximated with finite differences.

- **`error_behavior`** (*str, optional*) – How to handle any errors. For example, there can sometimes be overflow or underflow when computing $\delta(\theta)$ at a large $\hat{\theta}$. The following behaviors are supported:

  - `'revert'` (default) - Revert problematic values to their last computed values. If there are problematic values during the first objective evaluation, revert values in $\delta(\theta)$ to their starting values; in $\tilde{c}(\hat{\theta})$, to prices; in the objective, to `1e10`; and in other matrices such as Jacobians, to zeros.

  - `'punish'` - Set the objective to `1` and its gradient to all zeros. This option along with

a large `error_punishment` can be helpful for routines that do not use analytic gradients.

- `'raise'` - Raise an exception.

- **`error_punishment`** (*float, optional*) – How to scale the GMM objective value after an error. By default, the objective value is not scaled.

- **`delta_behavior`** (*str, optional*) – Configuration for the values at which the fixed point computation of $\delta(\theta)$ in each market will start. This configuration is only relevant if there are unfixed nonlinear parameters over which to optimize. The following behaviors are supported:

  - `'first'` (default) - Start at the values configured by `delta` during the first GMM step, and at the values computed by the last GMM step for each subsequent step.

  - `'logit'` - Start at the solution to the logit model in (3.45), or if ρ is specified, the solution to the nested logit model in (3.46). If the initial `delta` is left unspecified and there is no nesting parameter being optimized over, this will generally be equivalent to `'first'`.

  - `'last'` - Start at the values of $\delta(\theta)$ computed during the last objective evaluation, or, if this is the first evaluation, at the values configured by `delta`. This behavior tends to speed up computation but may introduce some instability into estimation.

- **`iteration`** (*Iteration, optional*) – *`Iteration`* configuration for how to solve the fixed point problem used to compute $\delta(\theta)$ in each market. This configuration is only relevant if there are nonlinear parameters, since δ can be estimated analytically in the logit model. By default, `Iteration('squarem', {'atol': 1e-14})` is used. Newton-based routines such as `Iteration('lm'`) that compute the Jacobian can often be faster (especially when there are nesting parameters), but the Jacobian-free SQUAREM routine is used by default because it speed is often comparable and in practice it can be slightly more stable.

- **`fp_type`** (*str, optional*) – Configuration for the type of contraction mapping used to compute $\delta(\theta)$. The following types are supported:

  - `'safe_linear'` (default) - The standard linear contraction mapping in (3.13) (or (3.44) when there is nesting) with safeguards against numerical overflow. Specifically, $\max_j V_{ijt}$ (or $\max_j V_{ijt}/(1-\rho_{h(j)})$ when there is nesting) is subtracted from V_{ijt} and the logit expression for choice probabilities in (3.5) (or (3.42)) is re-scaled accordingly. Such re-scaling is known as the log-sum-exp trick.

  - `'linear'` - The standard linear contraction mapping without safeguards against numerical overflow. This option may be preferable to `'safe_linear'` if utilities are reasonably small and unlikely to create overflow problems.

  - `'nonlinear'` - Iteration over $\exp\delta_{jt}$ instead of δ_{jt}. This can be faster than `'linear'` because it involves fewer logarithms. Also, following *Brunner, Heiss, Romahn, and Weiser (2017)*, the $\exp\delta_{jt}$ term can be cancelled out of the expression because it also appears in the numerator of (3.5) in the definition of $s_{jt}(\delta,\theta)$. This second trick only works when there are no nesting parameters.

  - `'safe_nonlinear'` - Exponentiated version with minimal safeguards against numerical overflow. Specifically, $\max_j \mu_{ijt}$ is subtracted from μ_{ijt}. This helps with stability but is less helpful than subtracting from the full V_{ijt}, so this version is less stable than `'safe_linear'`.

  This option is only relevant if `sigma` or `pi` are specified because δ can be estimated analytically in the logit model with (3.45) and in the nested logit model with (3.46).

- **shares_bounds** (*tuple, optional*) – Configuration for $s_{jt}(\delta, \theta)$ bounds in the contraction in (3.13) of the form `(lb, ub)`, in which both `lb` and `ub` are floats or `None`. By default, simulated shares are bounded from below by `1e-300`. This is only relevant if `fp_type` is `'safe_linear'` or `'linear'`. Bounding shares in the contraction does nothing with a nonlinear fixed point.

  It can be particularly helpful to bound shares in the contraction from below by a small number to prevent the contraction from failing when there are issues with zero or negative simulated shares. Zero shares can occur when there are underflow issues and negative shares can occur when there are issues with the numerical integration routine having negative integration weights (e.g., for sparse grid integration).

  The idea is that a small lower bound will allow the contraction to converge even when it encounters some issues with small or negative shares. However, if these issues are unlikely, disabling this behavior can speed up the iteration routine because fewer checks will be done.

  Both `None` and `numpy.nan` are converted to `-numpy.inf` in `lb` and to `numpy.inf` in `ub`.

- **costs_bounds** (*tuple, optional*) – Configuration for $c_{jt}(\theta)$ bounds of the form `(lb, ub)`, in which both `lb` and `ub` are floats or `None`. This is only relevant if X_3 was formulated by `product_formulations` in *Problem*. By default, marginal costs are unbounded.

  When `costs_type` in *Problem* is `'log'`, nonpositive $c(\theta)$ values can create problems when computing $\tilde{c}(\theta) = \log c(\theta)$. One solution is to set `lb` to a small number. Rows in Jacobians associated with clipped marginal costs will be zero.

  Both `None` and `numpy.nan` are converted to `-numpy.inf` in `lb` and to `numpy.inf` in `ub`.

- **W** (*array-like, optional*) – Starting values for the weighting matrix, W. By default, the 2SLS weighting matrix in (3.23) is used, unless there are any `micro_moments`, in which case an `initial_update` will be used to update starting values W and the mean utility δ at the initial parameter values before the first GMM step.

- **center_moments** (*bool, optional*) – Whether to center each column of the demand- and supply-side moments g before updating the weighting matrix W according to (3.24). By default, the moments are centered. This has no effect if `W_type` is `'unadjusted'`.

- **W_type** (*str, optional*) – How to update the weighting matrix. This has no effect if `method` is `'1s'`. Usually, `se_type` should be the same. The following types are supported:

  - `'robust'` (default) - Heteroscedasticity robust weighting matrix defined in (3.24) and (3.25).
  - `'clustered'` - Clustered weighting matrix defined in (3.24) and (3.26). Clusters must be defined by the `clustering_ids` field of `product_data` in *Problem*.
  - `'unadjusted'` - Homoskedastic weighting matrix defined in (3.24) and (3.28).

  This only affects the standard demand- and supply-side block of the updated weighting matrix. If there are micro moments, this matrix will be block-diagonal with a micro moment block equal to the inverse of the scaled covariance matrix defined in (3.36).

- **se_type** (*str, optional*) – How to compute parameter covariances and standard errors. Usually, `W_type` should be the same. The following types are supported:

  - `'robust'` (default) - Heteroscedasticity robust covariances defined in (3.30) and (3.25).
  - `'clustered'` - Clustered covariances defined in (3.30) and (3.26). Clusters must be defined by the `clustering_ids` field of `product_data` in *Problem*.

  - `'unadjusted'` - Homoskedastic covariances defined in (3.31), which are computed under the assumption that the weighting matrix is optimal.

  This only affects the standard demand- and supply-side block of the matrix of averaged moment covariances. If there are micro moments, the S matrix defined in the expressions referenced above will be block-diagonal with a micro moment block equal to the scaled covariance matrix defined in (3.36).

- **`demand_moment_types`** (*str or sequence of (str, int) tuples, optional*) – This can be used to replace demand-side moments $\bar{g}_D$. The following types are supported:
  - `'levels'` (default) - Standard moments in (3.12): $g_{D,jt} = \xi_{jt} \cdot Z_{D,jt}$.
  - `'innovations'` - Replace ξ_{jt} with its innovation in (5.8): $g_{D,jt} = \tilde{\xi}_{jt} \cdot Z_{D,jt}$.
  - `'differenced_innovations'` - Further difference the innovation in (5.8) to eliminate, for example, any product-market fixed effects: $g_{D,jt} = (\tilde{\xi}_{jt} - L\tilde{\xi}_{jt}) \cdot Z_{D,jt}$.

  To specify multiple stacked types, a list can be used instead. Each element in the list is a `(type, instruments)` tuple where `type` is one of the above strings and `instruments` is the number of columns in Z_D to use. For example, `demand_moment_types=[('levels', 1), ('innovations', 3), ('differenced_innovations', 3)]` implements *Arellano and Bond (1991)*'s "system GMM" estimator with an additional moment restriction on levels:

$$g_{D,jt} = \begin{bmatrix} \xi_{jt} \cdot Z_{D,jt,1} \\ \tilde{\xi}_{jt} \cdot Z_{D,jt,2:4} \\ (\tilde{\xi}_{jt} - L\tilde{\xi}_{jt}) \cdot Z_{D,jt,5:7} \end{bmatrix}. \tag{5.9}$$

  Warning: Alternative moment types are still an experimental feature. The way in which they are implemented and used may change somewhat in future releases.

- **`supply_moment_types`** (*str or sequence of (str, int) tuples, optional*) – This can be used to replace supply-side moments $\bar{g}_S$ analogously to how `demand_moment_types` replaces demand-side moments, but with ξ replaced by ω and Z_D replaced by Z_S.

- **`covariance_moments_mean`** (*float, optional*) – If `covariance_instruments` were specified in `product_data`, this can be used to choose a $m \neq 0$ in covariance moments $E[g_{C,jt}] = E[(\xi_{jt}\omega_{jt} - m)Z_{C,jt}] = 0$ where m is by default zero. This can be used for sensitivity testing to see how different covariances may affect estimates.

- **`micro_moments`** (*sequence of MicroMoment, optional*) – Configurations for the M_M *`MicroMoment`* instances that will be added to the standard set of moments. By default, no micro moments are used, so $M_M = 0$.

  When micro moments are specified, unless an initial weighting matrix `W` is specified as well (with a lower right micro moment block that reflects micro moment covariances), an `initial_update` will be used to update starting values W and the mean utility δ at the initial parameter values before the first GMM step.

  Note: When trying multiple parameter starting values to verify that the optimization routine converges to the same optimum, using `initial_update` is not recommended because different weighting matrices will be used for these different runs. A better option is to use `optimization=Optimization('return')` at the best guess for parameter values and pass *`ProblemResults.updated_W`* to `W` for each set of different parameter starting values.

- **micro_sample_covariances** (*array-like, optional*) – Sample covariance matrix for the M_M micro moments. By default, their asymptotic covariance matrix is computed according to (3.36). This override could be used, for example, if instead of estimating covariances at some estimated $\hat{\theta}$, one wanted to use a boostrap procedure to compute their covariances directly from the micro data.
- **resample_agent_data** (*callable, optional*) – If specified, simulation error in moment covariances will be accounted for by resampling $r = 1, \ldots, R$ sets of agents by iteratively calling this function, which should be of the following form:

```
resample_agent_data(index) --> agent_data or None
```

  where `index` increments from `0` to `1` and so on and `agent_data` is the corresponding resampled agent data, which should be a resampled version of the `agent_data` passed to *`Problem`*. Each `index` should correspond to a different set of randomly drawn agent data, with different integration nodes and demographics. If `index` is larger than $R - 1$, this function should return `None`, at which point agents will stop being resampled.

Returns *`ProblemResults`* of the solved problem.

Return type *ProblemResults*

Examples

- *Tutorial*

5.4 Micro Moment Classes

Micro dataset configurations are passed to micro part configurations, which are passed to micro moment configurations, which in turn can be passed to *`Problem.solve()`*.

`MicroDataset`(name, observations, compute_weights)	Configuration for a micro dataset d on which micro moments are computed.
`MicroPart`(name, dataset, compute_values)	Configuration for a micro moment part p.
`MicroMoment`(name, value, parts[, ...])	Configuration for a micro moment m.

5.4.1 pyblp.MicroDataset

class pyblp.**MicroDataset**(*name*, *observations*, *compute_weights*, *eliminated_product_ids_index=None*, *market_ids=None*)

Configuration for a micro dataset d on which micro moments are computed.

A micro dataset d, often a survey, is defined by survey weights w_{dijt}, which are used in (3.33). For example, $w_{dijt} = 1\{j \neq 0, t \in T_d\}$ defines a micro dataset that is a selected sample of inside purchasers in a few markets $T_d \subset T$, giving each market an equal sampling weight. Different micro datasets are independent.

See *Conlon and Gortmaker (2025)* for a more in-depth discussion of the standardized framework used by PyBLP for incorporating micro data into BLP-style estimation.

Parameters

- **name** (*str*) – The unique name of the dataset, which will be used for outputting information about micro moments.

- **observations** (*int*) – The number of observations N_d in the micro dataset.
- **compute_weights** (*callable*) – Function for computing survey weights w_{dijt} in a market of the following form:

```
compute_weights(t, products, agents) --> weights
```

where `t` is the market in which to compute weights, `products` is the market's *Products* (with J_t rows), and `agents` is the market's *Agents* (with I_t rows), unless `pyblp.options.micro_computation_chunks` is larger than its default of `1`, in which case `agents` is a chunk of the market's *Agents*. Denoting the number of rows in `agents` by I, the returned `weights` should be an array of one of the following shapes:

- $I \times J_t$: Conditions on inside purchases by assuming $w_{di0t} = 0$. Rows correspond to agents $i \in I$ in the same order as `agent_data` in *Problem* or *Simulation* and columns correspond to inside products $j \in J_t$ in the same order as `product_data` in *Problem* or *Simulation*.
- $I \times (1 + J_t)$: The first column indexes the outside option, which can have nonzero survey weights w_{di0t}.

Warning: If using different lambda functions to define different `compute_weights` functions in a loop, any variables that are changing within the loop should be passed as extra arguments to the function to preserve their scope. For example, `lambda t, p, a: weights[t]` where `weights` is some dictionary that is changing in the outer loop should instead be `lambda t, p, a, weights=weights: weights[t]`; otherwise, the `weights` in the current loop's iteration will be lost.

Warning: If using product-specific demographics, `agents.demographics` will be a $I_t \times D \times J_t$ array, instead of a $I_t \times D$ array like usual. Non-product specific demographics will be repeated J_t times.

Note: Particularly when using product-specific demographics or second choices, it may be convenient to use `numpy.einsum`, which handles many multiplying multi-dimensional arrays with common dimensions in an elegant way.

If the micro dataset contains second choice data, `weights` can have a third axis corresponding to second choices k in w_{dijkt}:

- $I \times J_t \times J_t$: Conditions on inside purchases by assuming $w_{di0kt} = w_{dij0t} = 0$.
- $I \times (1 + J_t) \times J_t$: The first column indexes the outside option, but the second choice is assumed to be an inside option, $w_{dij0t} = 0$.
- $I \times J_t \times (1 + J_t)$: The first index in the third axis indexes the outside option, but the first choice is assumed to be an inside option, $w_{di0k} = 0$.
- $I \times (1 + J_t) \times (1 + J_t)$: The first column and the first index in the third axis index the outside option as the first and second choice.

Warning: Second choice moments can use a lot of memory, especially when J_t is large. If this becomes an issue, consider setting `pyblp.options.micro_computation_chunks` to a value higher than its default of `1`, such as the highest J_t. This will cut down on memory usage without much affecting speed.

- **`eliminated_product_ids_index`** (*int, optional*) – This option determines whether the dataset's second choices are after only the first choice product j is eliminated from the choice set, in which case this should be `None`, the default, or if a group of products including the first choice product is eliminated, in which case this should be a number between `0` and the number of columns in the `product_ids` field of `product_data` minus one, inclusive. The column of `product_ids` determines the groups.
- **`market_ids`** (*array-like, optional*) – Distinct market IDs with nonzero survey weights w_{dijt}. For other markets, $w_{dijt} = 0$, and `compute_weights` will not be called.

Examples

- *Tutorial*

Methods

—

5.4.2 pyblp.MicroPart

`class` `pyblp.`**`MicroPart`**(*name*, *dataset*, *compute_values*)

Configuration for a micro moment part p.

Each micro moment part p is defined by its dataset d_p and micro values v_{pijt}, which are used in (3.34) and (3.35). For example, a micro moment part p with $v_{pijt} = y_{it}x_{jt}$ yields the mean $\bar{v}_p$ or expectation v_p of an interaction between some demographic y_{it} and product characteristic x_{jt}.

See *Conlon and Gortmaker (2025)* for a more in-depth discussion of the standardized framework used by PyBLP for incorporating micro data into BLP-style estimation.

Parameters

- **`name`** (*str*) – The unique name of the micro moment part, which will be used for outputting information about micro moments.
- **`dataset`** (*MicroDataset*) – The *`MicroDataset`* d_p on which the micro part is computed.
- **`compute_values`** (*callable*) – Function for computing micro values v_{pijt} (or v_{pijkt} if the dataset d_p contains second choice data) in a market of the following form:

```
compute_values(t, products, agents) --> values
```

where `t` is the market in which to compute values, `products` is the market's *`Products`* (with J_t rows), and `agents` is the market's *`Agents`* (with I_t rows), unless `pyblp.options.micro_computation_chunks` is larger than its default of `1`, in which case `agents` is a chunk of the market's *`Agents`*. The returned `values` should be an array of the same shape as the `weights` returned by `compute_weights` of `dataset`.

Warning: If using different lambda functions to define different `compute_values` functions in a loop, any variables that are changing within the loop should be passed as extra arguments to the function to preserve their scope. For example, `lambda t, p, a: np.outer(a.demographics[:, d], p.X2[:, c])` where `d` and `c` are indices that are changing in the outer loop should instead be `lambda t, p, a, d=d, c=c: np.outer(a.demographics[:, d], p.X2[:, c])`; otherwise, the values of `d` and `c` in the current loop's iteration will be lost.

Warning: If using product-specific demographics, `agents.demographics` will be a $I_t \times D \times J_t$ array, instead of a $I_t \times D$ array like usual. Non-product specific demographics will be repeated J_t times.

Note: Particularly when using product-specific demographics or second choices, it may be convenient to use `numpy.einsum`, which handles many multiplying multi-dimensional arrays with common dimensions in an elegant way.

Examples

- *Tutorial*

Methods

—

5.4.3 pyblp.MicroMoment

class pyblp.**MicroMoment**(*name*, *value*, *parts*, *compute_value=None*, *compute_gradient=None*)

Configuration for a micro moment m.

Each micro moment m matches a function $f_m(v)$ of one or more micro moment parts v in (3.33). For example, $f_m(v) = v_p$ with $v_{pijt} = y_{it}x_{jt}$ matches the mean of an interaction between some demographic y_{it} and some product characteristic x_{jt}.

Non-simple averages such as conditional means, covariances, correlations, or regression coefficients can be matched by choosing an appropriate function f_m. For example, $f_m(v) = v_1/v_2$ with $v_{1ijt} = y_{it}x_{jt}1\{j \neq 0\}$ and $v_{2ijt} = 1\{j \neq 0\}$ matches the conditional mean of an interaction between y_{it} and x_{jt} among those who do not choose the outside option $j = 0$.

See *Conlon and Gortmaker (2025)* for a more in-depth discussion of the standardized framework used by PyBLP for incorporating micro data into BLP-style estimation.

Parameters

- **name** (*str*) – The unique name of the micro moment, which will be used for outputting information about micro moments.
- **value** (*float*) – The observed value $f_m(\bar{v})$.

- **parts** (*MicroPart or sequence of MicroPart*) – The `MicroPart` configurations on which $f_m(\cdot)$ depends. If this is just a single part p and not a sequence, it is assumed that $f_m = v_p$ so that the micro moment matches v_p. If this is a sequence, both `compute_value` and `compute_gradient` need to be specified.
- **compute_value** (*callable, optional*) – Function for computing the simulated micro value $f_m(v)$ (only if `parts` is a sequence) of the following form:

```
compute_value(part_values) --> value
```

  where `part_values` is the array v with as many values as there are `parts` and the returned `value` is the scalar $f_m(v)$.
- **compute_gradient** (*callable, optional*) – Function for computing the gradient of the simulated micro value with respect to its parts (only required if `parts` is a sequence) of the following form:

```
compute_gradient(part_values) --> gradient
```

  where `part_values` is the array v with as many value as there are `parts` and the returned `gradient` is $\frac{\partial f_m(v)}{\partial v}$, an array of the same shape. This is used to compute both analytic gradients and moment covariances.

Examples

- *Tutorial*

Methods

5.5 Problem Results Class

Solved problems return the following results class.

ProblemResults	Results of a solved BLP problem.

5.5.1 pyblp.ProblemResults

class pyblp.**ProblemResults**

Results of a solved BLP problem.

Many results are class attributes. Other post-estimation outputs be computed by calling class methods.

Note: Methods in this class that compute one or more post-estimation output per market support *parallel()* processing. If multiprocessing is used, market-by-market computation of each post-estimation output will be distributed among the processes.

problem

Problem that created these results.

Type *Problem*

last_results
ProblemResults from the last GMM step.

Type *ProblemResults*

step
GMM step that created these results.

Type *int*

optimization_time
Number of seconds it took the optimization routine to finish.

Type *float*

cumulative_optimization_time
Sum of *ProblemResults.optimization_time* for this step and all prior steps.

Type *float*

total_time
Sum of *ProblemResults.optimization_time* and the number of seconds it took to set up the GMM step and compute results after optimization had finished.

Type *float*

cumulative_total_time
Sum of *ProblemResults.total_time* for this step and all prior steps.

Type *float*

converged
Whether the optimization routine converged.

Type *bool*

cumulative_converged
Whether the optimization routine converged for this step and all prior steps.

Type *bool*

optimization_iterations
Number of major iterations completed by the optimization routine.

Type *int*

cumulative_optimization_iterations
Sum of *ProblemResults.optimization_iterations* for this step and all prior steps.

Type *int*

objective_evaluations
Number of GMM objective evaluations.

Type *int*

cumulative_objective_evaluations
Sum of *ProblemResults.objective_evaluations* for this step and all prior steps.

Type *int*

fp_converged
Flags for convergence of the iteration routine used to compute $\delta(\theta)$ in each market during each objective evaluation. Rows are in the same order as *Problem.unique_market_ids* and column indices correspond to objective evaluations.

Type *ndarray*

cumulative_fp_converged
Concatenation of *ProblemResults.fp_converged* for this step and all prior steps.

Type *ndarray*

fp_iterations
Number of major iterations completed by the iteration routine used to compute $\delta(\theta)$ in each market during each objective evaluation. Rows are in the same order as *Problem.unique_market_ids* and column indices correspond to objective evaluations.

Type *ndarray*

cumulative_fp_iterations
Concatenation of *ProblemResults.fp_iterations* for this step and all prior steps.

Type *ndarray*

contraction_evaluations
Number of times the contraction used to compute $\delta(\theta)$ was evaluated in each market during each objective evaluation. Rows are in the same order as *Problem.unique_market_ids* and column indices correspond to objective evaluations.

Type *ndarray*

cumulative_contraction_evaluations
Concatenation of *ProblemResults.contraction_evaluations* for this step and all prior steps.

Type *ndarray*

parameters
Stacked parameters in the following order: $\hat{\theta}$, concentrated out elements of $\hat{\beta}$, and concentrated out elements of $\hat{\gamma}$.

Type *ndarray*

parameter_covariances
Estimated asymptotic covariance matrix for $\sqrt{N}(\hat{\theta} - \theta_0)$, in which θ are the stacked parameters. Standard errors are the square root of the diagonal of this matrix divided by N. Parameter covariances are not estimated during the first step of two-step GMM.

Type *ndarray*

parameter_sensitivity
Estimated local measure of the sensitivity of parameter estimates to moments $-(\bar{G}'W\bar{G})^{-1}\bar{G}'W$ from *Andrews, Gentzkow, and Shapiro (2017)*.

Type *ndarray*

theta
Estimated unfixed parameters, $\hat{\theta}$, in the following order: $\hat{\Sigma}$, $\hat{\Pi}$, $\hat{\rho}$, $\hat{\phi}$, non-concentrated out elements from $\hat{\beta}$, and non-concentrated out elements from $\hat{\gamma}$.

Type *ndarray*

sigma
Estimated Cholesky root of the covariance matrix for unobserved taste heterogeneity, $\hat{\Sigma}$.

Type *ndarray*

sigma_squared
Estimated covariance matrix for unobserved taste heterogeneity, $\hat{\Sigma}\hat{\Sigma}'$.

Type *ndarray*

pi
Estimated parameters that measures how agent tastes vary with demographics, $\hat{\Pi}$.

Type *ndarray*

rho
Estimated parameters that measure within nesting group correlations, $\hat{\rho}$.

Type *ndarray*

phi
Estimated demand-side unobservable autocorrelation, $\hat{\phi}$.

Type *ndarray*

beta
Estimated demand-side linear parameters, $\hat{\beta}$.

Type *ndarray*

gamma
Estimated supply-side linear parameters, $\hat{\gamma}$.

Type *ndarray*

sigma_se
Estimated standard errors for $\hat{\Sigma}$, which are not estimated in the first step of two-step GMM.

Type *ndarray*

sigma_squared_se
Estimated standard errors for $\hat{\Sigma}\hat{\Sigma}'$, which are computed with the delta method, and are not estimated in the first step of two-step GMM.

Type *ndarray*

pi_se
Estimated standard errors for $\hat{\Pi}$, which are not estimated in the first step of two-step GMM.

Type *ndarray*

rho_se
Estimated standard errors for $\hat{\rho}$, which are not estimated in the first step of two-step GMM.

Type *ndarray*

phi_se
Estimated standard error for $\hat{\phi}$, which is not estimated in the first step of two-step GMM.

Type *ndarray*

beta_se
Estimated standard errors for $\hat{\beta}$, which are not estimated in the first step of two-step GMM.

Type *ndarray*

gamma_se
Estimated standard errors for $\hat{\gamma}$, which are not estimated in the first step of two-step GMM.

Type *ndarray*

sigma_bounds
Bounds for Σ that were used during optimization, which are of the form `(lb, ub)`.

Type *tuple*

pi_bounds
Bounds for Π that were used during optimization, which are of the form `(lb, ub)`.

Type *tuple*

rho_bounds
Bounds for ρ that were used during optimization, which are of the form `(lb, ub)`.

Type *tuple*

phi_bounds
Bounds for ϕ that were used during optimization, which are of the form `(lb, ub)`.

Type *tuple*

beta_bounds
Bounds for β that were used during optimization, which are of the form `(lb, ub)`.

Type *tuple*

gamma_bounds
Bounds for γ that were used during optimization, which are of the form `(lb, ub)`.

Type *tuple*

sigma_labels
Variable labels for rows and columns of Σ, which are derived from the formulation for X_2.

Type *list of str*

pi_labels
Variable labels for columns of Π, which are derived from the formulation for demographics.

Type *list of str*

rho_labels
Variable labels for ρ. If ρ is not a scalar, this is *Problem.unique_nesting_ids*.

Type *list of str*

phi_labels
Variable labels for ϕ.

Type *list of str*

beta_labels
Variable labels for β, which are derived from the formulation for X_1.

Type *list of str*

gamma_labels
Variable labels for γ, which are derived from the formulation for X_3.

Type *list of str*

theta_labels
Variable labels for θ, which are derived from the above labels.

Type *list of str*

delta
Estimated mean utility, $\delta(\hat{\theta})$.

Type *ndarray*

clipped_shares
Vector of booleans indicating whether the associated simulated shares were clipped during the last fixed point iteration to compute $\delta(\hat{\theta})$. All elements will be `False` if `shares_bounds` in *Problem.solve()* is disabled (by default shares are bounded from below by a small number to alleviate issues with underflow and negative shares).

Type *ndarray*

tilde_costs
Estimated transformed marginal costs, $\tilde{c}(\hat{\theta})$ from (3.9). If `costs_bounds` were specified in *Problem.solve()*, c may have been clipped.

Type *ndarray*

clipped_costs
Vector of booleans indicating whether the associated marginal costs were clipped. All elements will be `False` if `costs_bounds` in *Problem.solve()* was not specified.

Type *ndarray*

xi
Estimated unobserved demand-side product characteristics, $\xi(\hat{\theta})$, or equivalently, the demand-side structural error term. Any absorbed fixed effects are not included.

Type *ndarray*

omega
Estimated unobserved supply-side product characteristics, $\omega(\hat{\theta})$, or equivalently, the supply-side structural error term. Any absorbed fixed effects are not included.

Type *ndarray*

xi_fe
Estimated demand-side fixed effects $\delta(\hat{\theta}) - X_1\hat{\beta} - \xi(\hat{\theta})$, which are only computed when there are absorbed demand-side fixed effects.

Type *ndarray*

omega_fe
Estimated supply-side fixed effects $\tilde{c}(\hat{\theta}) - X_3\hat{\gamma} - \omega(\hat{\theta})$, which are only computed when there are absorbed demand-side fixed effects.

Type *ndarray*

micro
Micro moments, $\bar{g}_M$, in (3.33).

Type *ndarray*

micro_values
Estimated micro moment values, $f_m(v)$. Rows are in the same order as *ProblemResults.micro*.

Type *ndarray*

micro_covariances
Estimated micro moment asymptotic covariance matrix S_M in (3.36) divided by N. Equal to `micro_sample_covariances` if overridden in *Problem.solve()*.

Type *ndarray*

moments
Moments, $\bar{g}$, in (3.11).

Type *ndarray*

moments_jacobian
Jacobian $\bar{G}$ of moments with respect to θ, in (3.19).

Type *ndarray*

moments_covariances
Estimated asymptotic covariance matrix of moments S in (3.25), (3.26), or (3.28), depending on `se_type` in *Problem.solve()*.

Type *ndarray*

simulation_covariances
Adjustment in (3.29) to moment covariances to account for simulation error. This will be all zeros unless `resample_agent_data` was specified in *Problem.solve()*.

Type *ndarray*

objective
GMM objective value, $q(\hat{\theta})$, defined in (3.10). If `scale_objective` was `True` in *Problem.solve()* (which is the default), this value was scaled by N so that objective values are more comparable across different problem sizes. Note that in some of the BLP literature (and earlier versions of this package), this expression was previously scaled by N^2.

Type *float*

xi_by_theta_jacobian
Estimated $\frac{\partial \xi}{\partial \theta} = \frac{\partial \delta}{\partial \theta}$, which is used to compute the gradient and standard errors.

Type *ndarray*

omega_by_theta_jacobian
Estimated $\frac{\partial \omega}{\partial \theta} = \frac{\partial \tilde{c}}{\partial \theta}$, which is used to compute the gradient and standard errors.

Type *ndarray*

micro_by_theta_jacobian
Estimated $\frac{\partial \bar{g}_M}{\partial \theta}$, which is used to compute the gradient and standard errors.

Type *ndarray*

gradient
Gradient of the GMM objective, $\nabla q(\hat{\theta})$, defined in (3.18). This is computed after the optimization routine finishes even if the routine was configured to not use analytic gradients.

Type *ndarray*

projected_gradient
Projected gradient of the GMM objective. When there are no parameter bounds, this will always be equal to *ProblemResults.gradient*. Otherwise, if an element in $\hat{\theta}$ is equal to its lower (upper) bound, the corresponding projected gradient value will be truncated at a maximum (minimum) of zero.

Type *ndarray*

projected_gradient_norm
Infinity norm of *ProblemResults.projected_gradient*.

Type *ndarray*

hessian
Estimated Hessian of the GMM objective. By default, this is computed with finite central differences after the optimization routine finishes.

Type *ndarray*

reduced_hessian
Reduced Hessian of the GMM objective. When there are no parameter bounds, this will always be equal to *ProblemResults.hessian*. Otherwise, if an element in $\hat{\theta}$ is equal to either its lower or upper bound, the corresponding row and column in the reduced Hessian will be all zeros.

Type *ndarray*

reduced_hessian_eigenvalues
Eigenvalues of *ProblemResults.reduced_hessian*.

Type *ndarray*

W
Weighting matrix, W, used to compute these results.

Type *ndarray*

updated_W
Weighting matrix updated according to (3.24).

Type *ndarray*

Examples

- *Tutorial*

Methods

bootstrap([draws, seed, iteration, ...])	Use a parametric bootstrap to create an empirical distribution of results.
compute_agent_scores(dataset[, micro_data, ...])	Compute scores for all agent-choices, treated as observations $n \in N_d$ from a micro dataset d.
compute_aggregate_elasticities([factor, ...])	Estimate aggregate elasticities of demand, $\mathscr{E}$, with respect to a variable, x.
compute_approximate_prices([firm_ids, ...])	Approximate equilibrium prices after firm or cost changes, p^*, under the assumption that shares and their price derivatives are unaffected by such changes.
compute_consumer_surpluses([prices, ...])	Estimate population-normalized consumer surpluses, CS.
compute_costs([firm_ids, ownership, market_id])	Estimate marginal costs, c.
compute_delta([agent_data, integration, ...])	Estimate mean utilities, δ.
compute_demand_hessians([name, market_id])	Estimate arrays of second derivatives of demand with respect to a variable, x.
compute_demand_jacobians([name, market_id])	Estimate matrices of derivatives of demand with respect to a variable, x.
compute_diversion_ratios([name, market_id])	Estimate matrices of diversion ratios, $\mathscr{D}$, with respect to a variable, x.
compute_elasticities([name, market_id])	Estimate matrices of elasticities of demand, ε, with respect to a variable, x.
compute_hhi([firm_ids, shares, market_id])	Estimate Herfindahl-Hirschman Indices, HHI.
compute_long_run_diversion_ratios([market_id])	Estimate matrices of long-run diversion ratios, $\bar{\mathscr{D}}$.
compute_markups([prices, costs, market_id])	Estimate markups, $\mathscr{M}$.

Continued on next page

Table 13 – continued from previous page

compute_micro_scores(dataset, micro_data[, ...])	Compute scores for observations $n \in N_d$ from a micro dataset d.
compute_micro_values(micro_moments)	Estimate micro moment values, $f_m(v)$.
compute_optimal_instruments([method, draws, ...])	Estimate feasible optimal or efficient instruments, Z_D^{opt} and Z_S^{opt}.
compute_passthrough([firm_ids, ownership, ...])	Estimate matrices of passthrough of marginal costs to equilibrium prices, Υ.
compute_prices([firm_ids, ownership, costs, ...])	Estimate equilibrium prices after firm or cost changes, p^*.
compute_probabilities([prices, delta, ...])	Estimate matrices of choice probabilities.
compute_profit_hessians([prices, costs, ...])	Estimate arrays of second derivatives of profits with respect to a prices.
compute_profits([prices, shares, costs, ...])	Estimate population-normalized gross expected profits, π.
compute_shares([prices, delta, agent_data, ...])	Estimate shares.
extract_diagonal_means(matrices[, market_id])	Extract means of diagonals from stacked $J_t \times J_t$ matrices for each market t.
extract_diagonals(matrices[, market_id])	Extract diagonals from stacked $J_t \times J_t$ matrices for each market t.
importance_sampling(draws[, ar_constant, ...])	Use importance sampling to construct nodes and weights for integration.
run_distance_test(unrestricted)	Test the validity of model restrictions with the distance test.
run_hansen_test()	Test the validity of overidentifying restrictions with the Hansen J test.
run_lm_test()	Test the validity of model restrictions with the Lagrange multiplier test.
run_wald_test(restrictions, ...)	Test the validity of model restrictions with the Wald test.
simulate_micro_data(dataset[, seed])	Simulate observations $n \in N_d$ from a micro dataset d.
to_dict([attributes])	Convert these results into a dictionary that maps attribute names to values.
to_pickle(path)	Save these results as a pickle file.

The results can be pickled or converted into a dictionary.

ProblemResults.to_pickle(path)	Save these results as a pickle file.
ProblemResults.to_dict([attributes])	Convert these results into a dictionary that maps attribute names to values.

5.5.2 pyblp.ProblemResults.to_pickle

ProblemResults.**to_pickle**(*path*)

Save these results as a pickle file.

Parameters **path** (*str or Path*) – File path to which these results will be saved.

5.5.3 pyblp.ProblemResults.to_dict

ProblemResults.**to_dict**(*attributes=('step', 'optimization_time', 'cumulative_optimization_time', 'total_time', 'cumulative_total_time', 'converged', 'cumulative_converged', 'optimization_iterations', 'cumulative_optimization_iterations', 'objective_evaluations', 'cumulative_objective_evaluations', 'fp_converged', 'cumulative_fp_converged', 'fp_iterations', 'cumulative_fp_iterations', 'contraction_evaluations', 'cumulative_contraction_evaluations', 'parameters', 'parameter_covariances', 'parameter_sensitivity', 'theta', 'sigma', 'sigma_squared', 'pi', 'rho', 'phi', 'beta', 'gamma', 'sigma_se', 'sigma_squared_se', 'pi_se', 'rho_se', 'phi_se', 'beta_se', 'gamma_se', 'sigma_bounds', 'pi_bounds', 'rho_bounds', 'phi_bounds', 'beta_bounds', 'gamma_bounds', 'sigma_labels', 'pi_labels', 'rho_labels', 'phi_labels', 'beta_labels', 'gamma_labels', 'theta_labels', 'delta', 'tilde_costs', 'clipped_shares', 'clipped_costs', 'xi', 'omega', 'xi_fe', 'omega_fe', 'micro', 'micro_values', 'micro_covariances', 'moments', 'moments_jacobian', 'moments_covariances', 'simulation_covariances', 'objective', 'xi_by_theta_jacobian', 'omega_by_theta_jacobian', 'micro_by_theta_jacobian', 'gradient', 'projected_gradient', 'projected_gradient_norm', 'hessian', 'reduced_hessian', 'reduced_hessian_eigenvalues', 'W', 'updated_W')*)

Convert these results into a dictionary that maps attribute names to values.

Parameters **attributes** (*sequence of str, optional*) – Name of attributes that will be added to the dictionary. By default, all *ProblemResults* attributes are added except for *ProblemResults.problem* and *ProblemResults.last_results*.

Returns Mapping from attribute names to values.

Return type *dict*

Examples

- *Tutorial*

The following methods test the validity of overidentifying and model restrictions.

ProblemResults.run_hansen_test()	Test the validity of overidentifying restrictions with the Hansen J test.
ProblemResults.run_distance_test(unrestricted)	Test the validity of model restrictions with the distance test.
ProblemResults.run_lm_test()	Test the validity of model restrictions with the Lagrange multiplier test.
ProblemResults.run_wald_test(restrictions, ...)	Test the validity of model restrictions with the Wald test.

5.5.4 pyblp.ProblemResults.run_hansen_test

ProblemResults.**run_hansen_test**()

Test the validity of overidentifying restrictions with the Hansen J test.

Following *Hansen (1982)*, the J statistic is

$$J = N\bar{g}(\hat{\theta})'W\bar{g}(\hat{\theta}) \tag{5.10}$$

where $\bar{g}(\hat{\theta})$ is defined in (3.11) and W is the optimal weighting matrix in (3.24).

Note: The statistic can equivalently be written as $J = Nq(\hat{\theta})$ where the GMM objective value is defined in (3.10), or the same but without the N if the GMM objective value was scaled by N, which is the default behavior.

When the overidentifying restrictions in this model are valid, the J statistic is asymptotically χ^2 with degrees of freedom equal to the number of overidentifying restrictions. This requires that there are more moments than parameters.

Warning: This test requires `ProblemResults.W` to be an optimal weighting matrix, so it should typically be run only after two-step GMM or after one-step GMM with a pre-specified optimal weighting matrix.

Returns The J statistic.

Return type *float*

Examples

- *Tutorial*

5.5.5 pyblp.ProblemResults.run_distance_test

`ProblemResults.`**`run_distance_test`**(*unrestricted*)

Test the validity of model restrictions with the distance test.

Following *Newey and West (1987)*, the distance or likelihood ratio-like statistic is

$$\text{LR} = J(\hat{\theta}^r) - J(\hat{\theta}^u) \tag{5.11}$$

where $J(\hat{\theta}^r)$ is the J statistic defined in (5.10) for this restricted model and $J(\hat{\theta}^u)$ is the J statistic for the unrestricted model.

Note: The statistic can equivalently be written as $\text{LR} = N[q(\hat{\theta}^r) - q(\hat{\theta}^u)]$ where the GMM objective value is defined in (3.10), or the same but without the N if the GMM objective value was scaled by N, which is the default behavior.

If the restrictions in this model are valid, the distance statistic is asymptotically χ^2 with degrees of freedom equal to the number of restrictions.

Warning: This test requires each model's `ProblemResults.W` to be the optimal weighting matrix, so it should typically be run only after two-step GMM or after one-step GMM with pre-specified optimal weighting matrices.

Parameters **`unrestricted`** (*ProblemResults*) – `ProblemResults` for the unrestricted model.

Returns The distance statistic.

Return type *float*

Examples

- *Tutorial*

5.5.6 pyblp.ProblemResults.run_lm_test

`ProblemResults.`**`run_lm_test`**`()`

Test the validity of model restrictions with the Lagrange multiplier test.

Following *Newey and West (1987)*, the Lagrange multiplier or score statistic is

$$\text{LM} = N\bar{g}(\hat{\theta})'W\bar{G}(\hat{\theta})V\bar{G}(\hat{\theta})'W\bar{g}(\hat{\theta}) \tag{5.12}$$

where $\bar{g}(\hat{\theta})$ is defined in (3.11), $\bar{G}(\hat{\theta})$ is defined in (3.19), W is the optimal weighting matrix in (3.24), and V is the covariance matrix of $\sqrt{N}(\hat{\theta} - \theta)$ in (3.30).

If the restrictions in this model are valid, the Lagrange multiplier statistic is asymptotically χ^2 with degrees of freedom equal to the number of restrictions.

Warning: This test requires `ProblemResults.W` to be an optimal weighting matrix, so it should typically be run only after two-step GMM or after one-step GMM with a pre-specified optimal weighting matrix.

Returns The Lagrange multiplier statistic.

Return type *float*

Examples

- *Tutorial*

5.5.7 pyblp.ProblemResults.run_wald_test

`ProblemResults.`**`run_wald_test`**`(`*restrictions*, *restrictions_jacobian*`)`

Test the validity of model restrictions with the Wald test.

Following *Newey and West (1987)*, the Wald statistic is

$$\text{Wald} = Nr(\hat{\theta})'[R(\hat{\theta})VR(\hat{\theta})']^{-1}r(\hat{\theta}) \tag{5.13}$$

where the restrictions are $r(\theta) = 0$ under the test's null hypothesis, their Jacobian is $R(\theta) = \frac{\partial r(\theta)}{\partial \theta}$, and V is the covariance matrix of $\sqrt{N}(\hat{\theta} - \theta)$ in (3.30).

If the restrictions are valid, the Wald statistic is asymptotically χ^2 with degrees of freedom equal to the number of restrictions.

Parameters

- **`restrictions`** (*array-like*) – Column vector of the model restrictions evaluated at the estimated parameters, $r(\hat{\theta})$.
- **`restrictions_jacobian`** (*array-like*) – Estimated Jacobian of the restrictions with respect to all parameters, $R(\hat{\theta})$. This matrix should have as many rows as `restrictions` and as many columns as `ProblemResults.parameter_covariances`.

Returns The Wald statistic.

Return type *float*

Examples

- *Tutorial*

In addition to class attributes, other post-estimation outputs can be estimated market-by-market with the following methods, each of which return an array.

`ProblemResults.compute_aggregate_elasticities`([agg...])	Estimate aggregate elasticities of demand, $\mathscr{E}$, with respect to a variable, x.
`ProblemResults.compute_elasticities`([name, ...])	Estimate matrices of elasticities of demand, ε, with respect to a variable, x.
`ProblemResults.compute_demand_jacobians`([...])	Estimate matrices of derivatives of demand with respect to a variable, x.
`ProblemResults.compute_demand_hessians`([...])	Estimate arrays of second derivatives of demand with respect to a variable, x.
`ProblemResults.compute_profit_hessians`([...])	Estimate arrays of second derivatives of profits with respect to a prices.
`ProblemResults.compute_diversion_ratios`([...])	Estimate matrices of diversion ratios, $\mathscr{D}$, with respect to a variable, x.
`ProblemResults.compute_long_run_diversion_ratios`([...])	Estimate matrices of long-run diversion ratios, $\bar{\mathscr{D}}$.
`ProblemResults.compute_probabilities`([...])	Estimate matrices of choice probabilities.
`ProblemResults.extract_diagonals`(matrices[, ...])	Extract diagonals from stacked $J_t \times J_t$ matrices for each market t.
`ProblemResults.extract_diagonal_means`(matrices)	Extract means of diagonals from stacked $J_t \times J_t$ matrices for each market t.
`ProblemResults.compute_delta`([agent_data, ...])	Estimate mean utilities, δ.
`ProblemResults.compute_costs`([firm_ids, ...])	Estimate marginal costs, c.
`ProblemResults.compute_passthrough`([...])	Estimate matrices of passthrough of marginal costs to equilibrium prices, Υ.
`ProblemResults.compute_approximate_prices`([...])	Approximate equilibrium prices after firm or cost changes, p^*, under the assumption that shares and their price derivatives are unaffected by such changes.
`ProblemResults.compute_prices`([firm_ids, ...])	Estimate equilibrium prices after firm or cost changes, p^*.
`ProblemResults.compute_shares`([prices, ...])	Estimate shares.
`ProblemResults.compute_hhi`([firm_ids, ...])	Estimate Herfindahl-Hirschman Indices, HHI.
`ProblemResults.compute_markups`([prices, ...])	Estimate markups, $\mathscr{M}$.
`ProblemResults.compute_profits`([prices, ...])	Estimate population-normalized gross expected profits, π.
`ProblemResults.compute_consumer_surpluses`([...])	Estimate population-normalized consumer surpluses, CS.

5.5.8 pyblp.ProblemResults.compute_aggregate_elasticities

`ProblemResults.`**`compute_aggregate_elasticities`**(*factor=0.1*, *name='prices'*, *market_id=None*)

Estimate aggregate elasticities of demand, $\mathscr{E}$, with respect to a variable, x.

In market t, the aggregate elasticity of demand is

$$\mathscr{E} = \sum_{j \in J_t} \frac{s_{jt}(x + \Delta x) - s_{jt}}{\Delta}, \tag{5.14}$$

in which Δ is a scalar factor and $s_{jt}(x + \Delta x)$ is the share of product j in market t, evaluated at the scaled values of the variable.

Parameters

- **`factor`** (*float, optional*) – The scalar factor, Δ.
- **`name`** (*str, optional*) – Name of the variable, x. By default, $x = p$, prices. If this is `None`, the variable will be $x = \delta$, the mean utility.
- **`market_id`** (*object, optional*) – ID of the market in which to compute aggregate elasticities. By default, aggregate elasticities are computed in all markets and stacked.

Returns Estimates of aggregate elasticities of demand, $\mathscr{E}$. If `market_id` was not specified, rows are in the same order as *`Problem.unique_market_ids`*.

Return type *ndarray*

Examples

- *Tutorial*

5.5.9 pyblp.ProblemResults.compute_elasticities

`ProblemResults.`**`compute_elasticities`**(*name='prices'*, *market_id=None*)

Estimate matrices of elasticities of demand, ε, with respect to a variable, x.

In market t, the value in row j and column k of ε is

$$\varepsilon_{jk} = \frac{x_{kt}}{s_{jt}} \frac{\partial s_{jt}}{\partial x_{kt}}. \tag{5.15}$$

Parameters

- **`name`** (*str, optional*) – Name of the variable, x. By default, $x = p$, prices. If this is `None`, the variable will be $x = \delta$, the mean utility.
- **`market_id`** (*object, optional*) – ID of the market in which to compute elasticities. By default, elasticities are computed in all markets and stacked.

Returns Estimated $J_t \times J_t$ matrices of elasticities of demand, ε. If `market_id` was not specified, matrices are estimated in each market t and stacked. Columns for a market are in the same order as products for the market. If a market has fewer products than others, extra columns will contain `numpy.nan`.

Return type *ndarray*

Examples

- *Tutorial*

5.5.10 pyblp.ProblemResults.compute_demand_jacobians

`ProblemResults.`**`compute_demand_jacobians`**(*name='prices'*, *market_id=None*)

Estimate matrices of derivatives of demand with respect to a variable, x.

In market t, the value in row j and column k is

$$\frac{\partial s_{jt}}{\partial x_{kt}}. \tag{5.16}$$

Parameters

- **`name`** (*str, optional*) – Name of the variable, x. By default, $x = p$, prices.
- **`market_id`** (*object, optional*) – ID of the market in which to compute Jacobians. By default, Jacobians are computed in all markets and stacked.

Returns Estimated $J_t \times J_t$ matrices of derivatives of demand. If `market_id` was not specified, matrices are estimated in each market t and stacked. Columns for a market are in the same order as products for the market. If a market has fewer products than others, extra columns will contain `numpy.nan`.

Return type *ndarray*

Examples

- *Tutorial*

5.5.11 pyblp.ProblemResults.compute_demand_hessians

`ProblemResults.`**`compute_demand_hessians`**(*name='prices'*, *market_id=None*)

Estimate arrays of second derivatives of demand with respect to a variable, x.

In market t, the value indexed by (j, k, ℓ) is

$$\frac{\partial^2 s_{jt}}{\partial x_{kt} \partial x_{\ell t}}. \tag{5.17}$$

Parameters

- **`name`** (*str, optional*) – Name of the variable, x. By default, $x = p$, prices.
- **`market_id`** (*object, optional*) – ID of the market in which to compute Hessians. By default, Hessians are computed in all markets and stacked.

Returns Estimated $J_t \times J_t \times J_t$ arrays of second derivatives of demand. If `market_id` was not specified, arrays are estimated in each market t and stacked. Indices for a market are in the same order as products for the market. If a market has fewer products than others, extra indices will contain `numpy.nan`.

Return type *ndarray*

Examples

- *Tutorial*

5.5.12 pyblp.ProblemResults.compute_profit_hessians

`ProblemResults.`**`compute_profit_hessians`**(*prices=None*, *costs=None*, *market_id=None*)

Estimate arrays of second derivatives of profits with respect to a prices.

In market t, the value indexed by (j, k, ℓ) is

$$\frac{\partial^2 \pi_{jt}}{\partial p_{kt} \partial p_{\ell t}}. \tag{5.18}$$

Profit Hessians can be used to check second order conditions for firms' pricing problem. See *SimulationResults.profit_hessians* and *SimulationResults.profit_hessian_eigenvalues* for more information.

Parameters

- **`prices`** (*array-like, optional*) – Prices, p, such as equilibrium prices, p^*, computed by *ProblemResults.compute_prices()*. By default, unchanged prices are used.
- **`costs`** (*array-like*) – Marginal costs, c. By default, marginal costs are computed with *ProblemResults.compute_costs()*. Costs under a changed ownership structure can be computed by specifying the `firm_ids` or `ownership` arguments of *ProblemResults.compute_costs()*.
- **`market_id`** (*object, optional*) – ID of the market in which to compute Hessians. By default, Hessians are computed in all markets and stacked.

Returns Estimated $J_t \times J_t \times J_t$ arrays of second derivatives of profits. If `market_id` was not specified, arrays are estimated in each market t and stacked. Indices for a market are in the same order as products for the market. If a market has fewer products than others, extra indices will contain `numpy.nan`.

Return type *ndarray*

Examples

- *Tutorial*

5.5.13 pyblp.ProblemResults.compute_diversion_ratios

`ProblemResults.`**`compute_diversion_ratios`**(*name='prices'*, *market_id=None*)

Estimate matrices of diversion ratios, $\mathscr{D}$, with respect to a variable, x.

In market t, the value in row j and column $k \neq j$ is

$$\mathscr{D}_{jk} = -\frac{\partial s_{kt}}{\partial x_{jt}} \Big/ \frac{\partial s_{jt}}{\partial x_{jt}}. \tag{5.19}$$

Diversion ratios for the outside good are reported on diagonals:

$$\mathscr{D}_{jj} = -\frac{\partial s_{0t}}{\partial x_{jt}} \Big/ \frac{\partial s_{jt}}{\partial x_{jt}}. \tag{5.20}$$

Unlike *ProblemResults.compute_long_run_diversion_ratios()*, this gives the marginal treatment effect (MTE) version of the diversion ratio. For more information, see *Conlon and Mortimer (2021)*.

Parameters

- **name** (*str, optional*) – Name of the variable, x. By default, $x = p$, prices. If this is `None`, the variable will be $x = \delta$, the mean utility.
- **market_id** (*object, optional*) – ID of the market in which to compute diversion ratios. By default, diversion ratios are computed in all markets and stacked.

Returns Estimated $J_t \times J_t$ matrices of diversion ratios, $\mathscr{D}$. If `market_id` was not specified, matrices are estimated in each market t and stacked. Columns for a market are in the same order as products for the market. If a market has fewer products than others, extra columns will contain `numpy.nan`.

Return type *ndarray*

Examples

- *Tutorial*

5.5.14 pyblp.ProblemResults.compute_long_run_diversion_ratios

`ProblemResults.`**`compute_long_run_diversion_ratios`**(*market_id=None*)

Estimate matrices of long-run diversion ratios, $\bar{\mathscr{D}}$.

In market t, the value in row j and column $k \neq j$ is

$$\bar{\mathscr{D}}_{jk} = \frac{s_{k(-j)t} - s_{kt}}{s_{jt}}, \tag{5.21}$$

in which $s_{k(-j)t}$ is the share of product k computed with j removed from the choice set. Long-run diversion ratios for the outside good are reported on diagonals:

$$\bar{\mathscr{D}}_{jj} = \frac{s_{0(-j)t} - s_0}{s_{jt}}. \tag{5.22}$$

Unlike *`ProblemResults.compute_diversion_ratios()`*, this gives the average treatment effect (ATE) version of the diversion ratio. For more information, see *Conlon and Mortimer (2021)*.

Parameters **market_id** (*object, optional*) – ID of the market in which to compute long-run diversion ratios. By default, long-run diversion ratios are computed in all markets and stacked.

Returns Estimated $J_t \times J_t$ matrices of long-run diversion ratios, $\bar{\mathscr{D}}$. If `market_id` was not specified, matrices are estimated in each market t and stacked. Columns for a market are in the same order as products for the market. If a market has fewer products than others, extra columns will contain `numpy.nan`.

Return type *ndarray*

Examples

- *Tutorial*

5.5.15 pyblp.ProblemResults.compute_probabilities

`ProblemResults.compute_probabilities`(*prices=None*, *delta=None*, *agent_data=None*, *integration=None*, *market_id=None*)

Estimate matrices of choice probabilities.

For each market, the value in row j and column i is given by (3.5) when there are random coefficients, and by (3.42) when there is additionally a nested structure. For the logit and nested logit models, choice probabilities are market shares.

It may be desirable to compute the probabilities associated with equilibrium prices that have been computed, for example, by *ProblemResults.compute_prices()*.

Note: To compute equilibrium shares (and prices) associated with a more complicated counterfactual, a *Simulation* for the counterfactual can be initialized with the estimated parameters, structural errors, and marginal costs from these results, and then solved with *Simulation.replace_endogenous()*.

Alternatively, this method can also be used to evaluate the performance of different numerical integration configurations. One way to do so is to use *ProblemResults.compute_delta()* to compute mean utilities with a very precise integration rule (one that is infeasible to use during estimation), use these same mean utilities and integration rule to precisely compute probabilities, and then compare error between these precisely-computed probabilities and probabilities computed with less precise (but feasible to use during estimation) integration rules, still using the precisely-computed mean utilities.

Parameters

- **prices** (*array-like, optional*) – Prices at which to evaluate probabilities, such as equilibrium prices, p^*, computed by *ProblemResults.compute_prices()*. By default, unchanged prices are used.
- **delta** (*array-like, optional*) – Mean utilities that will be used to evaluate probabilities, such as those computed more precisely by *ProblemResults.compute_delta()*. By default, the estimated *ProblemResults.delta* is used, and updated with any specified `prices`.
- **agent_data** (*structured array-like, optional*) – Agent data that will be used to compute probabilities. By default, `agent_data` in *Problem* is used. For more information, refer to *Problem*.
- **integration** (*Integration, optional*) – *Integration* configuration that will be used to compute probabilities, which will replace any `nodes` field in `agent_data`. This configuration is required if `agent_data` is specified without a nodes field. By default, `agent_data` in *Problem* is used. For more information, refer to *Problem*.
- **market_id** (*object, optional*) – ID of the market in which to compute choice probabilities. By default, choice probabilities are computed in all markets and stacked.

Returns Estimated $J_t \times I_t$ matrices of choice probabilities. If `market_id` was not specified, matrices are estimated in each market t and stacked. Columns for a market are in the same order as agents for the market. If a market has fewer agents than others, extra columns will contain `numpy.nan`.

Return type *ndarray*

Examples

- *Tutorial*

5.5.16 pyblp.ProblemResults.extract_diagonals

ProblemResults.**extract_diagonals**(*matrices*, *market_id=None*)

Extract diagonals from stacked $J_t \times J_t$ matrices for each market t.

Parameters

- **matrices** (*array-like*) – Stacked matrices, such as estimates of ε, computed by *ProblemResults.compute_elasticities()*; $\mathscr{D}$, computed by *ProblemResults.compute_diversion_ratios()*; $\bar{\mathscr{D}}$, computed by *ProblemResults.compute_long_run_diversion_ratios()*; or s_{ijt} computed by *ProblemResults.compute_probabilities()*.
- **market_id** (*object, optional*) – ID of the market in which to extract diagonals. By default, diagonals are extracted in all markets and stacked.

Returns Stacked matrix diagonals. If `market_id` was not specified, diagonals are extracted in each market t and stacked. If the matrices are estimates of ε, a diagonal is a market's own elasticities of demand; if they are estimates of $\mathscr{D}$ or $\bar{\mathscr{D}}$, a diagonal is a market's diversion ratios to the outside good.

Return type *ndarray*

Examples

- *Tutorial*

5.5.17 pyblp.ProblemResults.extract_diagonal_means

ProblemResults.**extract_diagonal_means**(*matrices*, *market_id=None*)

Extract means of diagonals from stacked $J_t \times J_t$ matrices for each market t.

Parameters

- **matrices** (*array-like*) – Stacked matrices, such as estimates of ε, computed by *ProblemResults.compute_elasticities()*; $\mathscr{D}$, computed by *ProblemResults.compute_diversion_ratios()*; $\bar{\mathscr{D}}$, computed by *ProblemResults.compute_long_run_diversion_ratios()*; or s_{ijt} computed by *ProblemResults.compute_probabilities()*.
- **market_id** (*object, optional*) – ID of the market in which to extract diagonal means. By default, diagonal means are extracted in all markets and stacked.

Returns Stacked diagonal means. If `market_id` was not specified, diagonal means are extracted in each market t and stacked. If the matrices are estimates of ε, the mean of a diagonal is a market's mean own elasticity of demand; if they are estimates of $\mathscr{D}$ or $\bar{\mathscr{D}}$, the mean of a diagonal is a market's mean diversion ratio to the outside good. Rows are in the same order as *Problem.unique_market_ids*.

Return type *ndarray*

Examples

- *Tutorial*

5.5.18 pyblp.ProblemResults.compute_delta

`ProblemResults.compute_delta`(*agent_data=None*, *integration=None*, *iteration=None*, *fp_type='safe_linear'*, *shares_bounds=(1e-300, None)*, *market_id=None*)

Estimate mean utilities, δ.

This method can be used to compute mean utilities at the estimated parameters with a different integration configuration or with different fixed point iteration settings than those used during estimation. The estimated `ProblemResults.delta` will be used as starting values for the fixed point routine.

A more precisely estimated mean utility can be used, for example, by `ProblemResults.importance_sampling()`. It can also be used to `ProblemResults.compute_shares()` to compare the performance of different integration routines.

Parameters

- **agent_data** (*structured array-like, optional*) – Agent data that will be used to compute δ. By default, `agent_data` in `Problem` is used. For more information, refer to `Problem`.
- **integration** (*Integration, optional*) – `Integration` configuration that will be used to compute δ, which will replace any `nodes` field in `agent_data`. This configuration is required if `agent_data` is specified without a nodes field. By default, `agent_data` in `Problem` is used. For more information, refer to `Problem`.
- **iteration** (*Iteration, optional*) – `Iteration` configuration for how to solve the fixed point problem used to compute δ in each market. By default, `Iteration('squarem', {'atol': 1e-14})` is used. For more information, refer to `Problem.solve()`.
- **fp_type** (*str, optional*) – Configuration for the type of contraction mapping used to compute δ in each market. By default, `'safe_linear'` is used. For more information, refer to `Problem.solve()`.
- **shares_bounds** (*tuple, optional*) – Configuration for $s_{jt}(\delta, \theta)$ bounds of the form `(lb, ub)`, in which both `lb` and `ub` are floats or `None`. By default, simulated shares are bounded from below by `1e-300`. This is only relevant if `fp_type` is `'safe_linear'` or `'linear'`. Bounding shares in the contraction does nothing with a nonlinear fixed point. For more information, refer to `Problem.solve()`.
- **market_id** (*object, optional*) – ID of the market in which to compute mean utilities. By default, mean utilities is computed in all markets and stacked.

Returns Mean utilities, δ.

Return type *ndarray*

Examples

- *Tutorial*

5.5.19 pyblp.ProblemResults.compute_costs

`ProblemResults.compute_costs`(*firm_ids=None*, *ownership=None*, *market_id=None*)

Estimate marginal costs, c.

Marginal costs are computed with the η-markup equation in (3.7):

$$c = p - \eta. \tag{5.23}$$

Parameters

- **firm_ids** (*array-like, optional*) – Firm IDs. By default, the `firm_ids` field of `product_data` in *Problem* will be used.
- **ownership** (*array-like, optional*) – Ownership matrices. By default, standard ownership matrices based on `firm_ids` will be used unless the `ownership` field of `product_data` in *Problem* was specified.
- **market_id** (*object, optional*) – ID of the market in which to compute marginal costs. By default, marginal costs are computed in all markets and stacked.

Returns Marginal costs, c.

Return type *ndarray*

Examples

- *Tutorial*

5.5.20 pyblp.ProblemResults.compute_passthrough

`ProblemResults.`**compute_passthrough**(*firm_ids=None*, *ownership=None*, *market_id=None*)

Estimate matrices of passthrough of marginal costs to equilibrium prices, Υ.

In market t, the value in row j and column k of Υ is

$$\Upsilon_{jk} = \frac{\partial p_j}{\partial c_k}. \tag{5.24}$$

Parameters

- **firm_ids** (*array-like, optional*) – Firm IDs. By default, the `firm_ids` field of `product_data` in *Problem* will be used.
- **ownership** (*array-like, optional*) – Ownership matrices. By default, standard ownership matrices based on `firm_ids` will be used unless the `ownership` field of `product_data` in *Problem* was specified.
- **market_id** (*object, optional*) – ID of the market in which to compute passthrough. By default, passthrough matrices are computed in all markets and stacked.

Returns Estimated $J_t \times J_t$ passthrough matrices, Υ. If `market_id` was not specified, matrices are estimated in each market t and stacked. Columns for a market are in the same order as products for the market. If a market has fewer products than others, extra columns will contain `numpy.nan`.

Return type *ndarray*

Examples

- *Tutorial*

5.5.21 pyblp.ProblemResults.compute_approximate_prices

ProblemResults.**compute_approximate_prices**(*firm_ids=None*, *ownership=None*, *costs=None*, *market_id=None*)

Approximate equilibrium prices after firm or cost changes, p^*, under the assumption that shares and their price derivatives are unaffected by such changes.

This approximation is in the spirit of *Hausman, Leonard, and Zona (1994)* and *Werden (1997)*. Prices in each market are computed according to the η-markup equation in (3.7):

$$p^* = c^* + \eta^*, \tag{5.25}$$

in which the markup term is approximated with

$$\eta^* \approx -\left(\mathscr{H}^* \odot \frac{\partial s}{\partial p}\right)^{-1} s \tag{5.26}$$

where $\mathscr{H}^*$ is the ownership or product holding matrix associated with firm changes.

Parameters

- **firm_ids** (*array-like, optional*) – Potentially changed firm IDs. By default, the unchanged `firm_ids` field of `product_data` in *Problem* will be used.
- **ownership** (*array-like, optional*) – Potentially changed ownership matrices. By default, standard ownership matrices based on `firm_ids` will be used unless the `ownership` field of `product_data` in *Problem* was specified.
- **costs** (*array-like, optional*) – Potentially changed marginal costs, c^*. By default, unchanged marginal costs are computed with *ProblemResults.compute_costs()*. Costs under a changed ownership structure can be computed by specifying the `firm_ids` or `ownership` arguments of *ProblemResults.compute_costs()*.
- **market_id** (*object, optional*) – ID of the market in which to compute approximate equilibrium prices. By default, approximate equilibrium prices are computed in all markets and stacked.

Returns Approximation of equilibrium prices after any firm or cost changes, p^*.

Return type *ndarray*

Examples

- *Tutorial*

5.5.22 pyblp.ProblemResults.compute_prices

ProblemResults.**compute_prices**(*firm_ids=None*, *ownership=None*, *costs=None*, *prices=None*, *iteration=None*, *constant_costs=True*, *market_id=None*)

Estimate equilibrium prices after firm or cost changes, p^*.

Note: To compute equilibrium prices (and shares) associated with a more complicated counterfactual, a *Simulation* for the counterfactual can be initialized with the estimated parameters, structural errors, and marginal costs from these results, and then solved with *Simulation.replace_endogenous()*. The returned *SimulationResults* gives more information about the contraction than this method, such as the number of contraction evaluations. It also automatically reports first and second order conditions.

Prices are computed in each market by iterating over the ζ-markup contraction in (3.51):

$$p^* \leftarrow c^* + \zeta^*(p^*), \tag{5.27}$$

in which the markup term from (3.48) is

$$\zeta^*(p^*) = \Lambda^{-1}(p^*)[\mathscr{H}^* \odot \Gamma(p^*)]'(p^* - c^*) - \Lambda^{-1}(p^*)s(p^*) \tag{5.28}$$

where $\mathscr{H}^*$ is the ownership matrix associated with firm changes.

Parameters

- **firm_ids** (*array-like, optional*) – Potentially changed firm IDs. By default, the unchanged `firm_ids` field of `product_data` in *Problem* will be used.
- **ownership** (*array-like, optional*) – Potentially changed ownership matrices. By default, standard ownership matrices based on `firm_ids` will be used unless the `ownership` field of `product_data` in *Problem* was specified.
- **costs** (*array-like*) – Potentially changed marginal costs, c^*. By default, unchanged marginal costs are computed with *ProblemResults.compute_costs()*. Costs under a changed ownership structure can be computed by specifying the `firm_ids` or `ownership` arguments of *ProblemResults.compute_costs()*. If marginal costs depend on prices through market shares, they will be updated to reflect different prices during each iteration of the routine. Updated marginal costs can be obtained by instead using *Simulation.replace_endogenous()*.
- **prices** (*array-like, optional*) – Prices at which the fixed point iteration routine will start. By default, unchanged prices, p, are used as starting values. Other reasonable starting prices include the approximate equilibrium prices computed by *ProblemResults.compute_approximate_prices()*.
- **iteration** (*Iteration, optional*) – *Iteration* configuration for how to solve the fixed point problem in each market. By default, `Iteration('simple', {'atol': 1e-12})` is used.
- **constant_costs** (*bool, optional*) – Whether to assume that marginal costs, c, remain constant as equilibrium prices and shares change. By default this is `True`, which means that firms treat marginal costs as constant (equal to `costs`) when setting prices. This assumption is implicit in how *ProblemResults.compute_costs()* computes marginal costs. If set to `False`, marginal costs will be allowed to adjust if `shares` was included in the formulation for X_3 in *Problem*.
- **market_id** (*object, optional*) – ID of the market in which to compute equilibrium prices. By default, equilibrium prices are computed in all markets and stacked.

Returns Estimates of equilibrium prices after any firm or cost changes, p^*.

Return type *ndarray*

Examples

- *Tutorial*

5.5.23 pyblp.ProblemResults.compute_shares

`ProblemResults.`**compute_shares**(*prices=None*, *delta=None*, *agent_data=None*, *integration=None*, *market_id=None*)

Estimate shares.

It may be desirable to compute the shares associated with equilibrium prices that have been computed, for example, by *ProblemResults.compute_prices()*.

Note: To compute equilibrium shares (and prices) associated with a more complicated counterfactual, a *Simulation* for the counterfactual can be initialized with the estimated parameters, structural errors, and marginal costs from these results, and then solved with *Simulation.replace_endogenous()*.

Alternatively, this method can also be used to evaluate the performance of different numerical integration configurations. One way to do so is to use *ProblemResults.compute_delta()* to compute mean utilities with a very precise integration rule (one that is infeasible to use during estimation), use these same mean utilities and integration rule to precisely compute shares, and then compare error between these precisely-computed shares and shares computed with less precise (but feasible to use during estimation) integration rules, still using the precisely-computed mean utilities.

Parameters

- **prices** (*array-like, optional*) – Prices at which to evaluate shares, such as equilibrium prices, p^*, computed by *ProblemResults.compute_prices()*. By default, unchanged prices are used.
- **delta** (*array-like, optional*) – Mean utilities that will be used to evaluate shares, such as those computed more precisely by *ProblemResults.compute_delta()*. By default, the estimated *ProblemResults.delta* is used, and updated with any specified `prices`.
- **agent_data** (*structured array-like, optional*) – Agent data that will be used to compute shares. By default, `agent_data` in *Problem* is used. For more information, refer to *Problem*.
- **integration** (*Integration, optional*) – *Integration* configuration that will be used to compute shares, which will replace any `nodes` field in `agent_data`. This configuration is required if `agent_data` is specified without a nodes field. By default, `agent_data` in *Problem* is used. For more information, refer to *Problem*.
- **market_id** (*object, optional*) – ID of the market in which to compute shares. By default, shares are computed in all markets and stacked.

Returns Estimates of shares.

Return type *ndarray*

Examples

- *Tutorial*

5.5.24 pyblp.ProblemResults.compute_hhi

`ProblemResults.`**compute_hhi**(*firm_ids=None*, *shares=None*, *market_id=None*)

Estimate Herfindahl-Hirschman Indices, HHI.

The index in market t is

$$\text{HHI} = 10{,}000 \times \sum_{f \in F_t} \left(\sum_{j \in J_{ft}} s_{jt} \right)^2. \tag{5.29}$$

Parameters

- **firm_ids** (*array-like, optional*) – Firm IDs. By default, the unchanged `firm_ids` field of `product_data` in *Problem* will be used.
- **shares** (*array-like, optional*) – Shares, s, such as those computed by *ProblemResults.compute_shares()*. By default, unchanged shares are used.
- **market_id** (*object, optional*) – ID of the market in which to compute the index. By default, indices are computed in all markets and stacked.

Returns Estimated Herfindahl-Hirschman Indices, HHI. If `market_ids` was not specified, rows are in the same order as *Problem.unique_market_ids*.

Return type *ndarray*

Examples

- *Tutorial*

5.5.25 pyblp.ProblemResults.compute_markups

`ProblemResults.`**compute_markups**(*prices=None*, *costs=None*, *market_id=None*)

Estimate markups, $\mathscr{M}$.

The markup of product j in market t is

$$\mathscr{M}_{jt} = \frac{p_{jt} - c_{jt}}{p_{jt}}. \tag{5.30}$$

Parameters

- **prices** (*array-like, optional*) – Prices, p, such as equilibrium prices, p^*, computed by *ProblemResults.compute_prices()*. By default, unchanged prices are used.
- **costs** (*array-like*) – Marginal costs, c. By default, marginal costs are computed with *ProblemResults.compute_costs()*. Costs under a changed ownership structure can be computed by specifying the `firm_ids` or `ownership` arguments of *ProblemResults.compute_costs()*.
- **market_id** (*object, optional*) – ID of the market in which to compute markups. By default, markups are computed in all markets and stacked.

Returns Estimated markups, $\mathscr{M}$.

Return type *ndarray*

Examples

- *Tutorial*

5.5.26 pyblp.ProblemResults.compute_profits

`ProblemResults.`**compute_profits**(*prices=None*, *shares=None*, *costs=None*, *market_id=None*)

Estimate population-normalized gross expected profits, π.

With constant costs, the profit from product j in market t is

$$\pi_{jt} = (p_{jt} - c_{jt})s_{jt}. \tag{5.31}$$

Parameters

- **prices** (*array-like, optional*) – Prices, p, such as equilibrium prices, p^*, computed by `ProblemResults.compute_prices()`. By default, unchanged prices are used.
- **shares** (*array-like, optional*) – Shares, s, such as those computed by `ProblemResults.compute_shares()`. By default, unchanged shares are used.
- **costs** (*array-like*) – Marginal costs, c. By default, marginal costs are computed with `ProblemResults.compute_costs()`. Costs under a changed ownership structure can be computed by specifying the `firm_ids` or `ownership` arguments of `ProblemResults.compute_costs()`.
- **market_id** (*object, optional*) – ID of the market in which to compute profits. By default, profits are computed in all markets and stacked.

Returns Estimated population-normalized gross expected profits, π.

Return type *ndarray*

Examples

- *Tutorial*

5.5.27 pyblp.ProblemResults.compute_consumer_surpluses

`ProblemResults.`**compute_consumer_surpluses**(*prices=None*, *keep_all=False*, *eliminate_product_ids=None*, *product_ids_index=0*, *market_id=None*)

Estimate population-normalized consumer surpluses, CS.

Assuming away nonlinear income effects, the surplus in market t is

$$\text{CS} = \sum_{i \in I_t} w_{it} \text{CS}_{it}, \tag{5.32}$$

in which the consumer surplus for individual i is

$$\text{CS}_{it} = \log\left(1 + \sum_{j \in J_t} \exp V_{ijt}\right) \Big/ \left(-\frac{\partial V_{i1t}}{\partial p_{1t}}\right), \tag{5.33}$$

or with nesting parameters,

$$\text{CS}_{it} = \log\left(1 + \sum_{h \in H} \exp V_{iht}\right) \Big/ \left(-\frac{\partial V_{i1t}}{\partial p_{1t}}\right) \tag{5.34}$$

where V_{ijt} is defined in (3.1) and V_{iht} is defined in (3.43).

Warning: $\frac{\partial V_{1ti}}{\partial p_{1t}}$ is the derivative of utility for the first product with respect to its price. The first product is chosen arbitrarily because this method assumes that there are no nonlinear income effects, which implies that this derivative is the same for all products. Computed consumer surpluses will likely be incorrect if prices are formulated in a nonlinear fashion like `log(prices)`.

Comparing consumer surpluses with the same values computed after eliminating one or more products from the agents' choice sets (i.e. setting $\exp V_{ijt} = 0$ for eliminated products j) gives a measure of willingness to pay. This can be done with the `eliminate_product_ids` argument.

Parameters

- **prices** (*array-like, optional*) – Prices at which utilities and price derivatives will be evaluated, such as equilibrium prices, p^*, computed by `ProblemResults.compute_prices()`. By default, unchanged prices are used.
- **keep_all** (*bool, optional*) – Whether to keep all individuals' surpluses CS_{it} or just market-level surpluses. By default only market-level surpluses are returned, but returning all surpluses will be important for analysis by agent type or demographic category.
- **eliminate_product_ids** (*sequence of object, optional*) – IDs of the products to eliminate from the choice set. These IDs should show up in the `product_ids` field of `product_data` in `Problem`. Eliminating one or more products and comparing consumer surpluses gives a measure of willingness to pay for these products.
- **product_ids_index** (*int, optional*) – Index between `0` and the number of columns in the `product_ids` field of `product_data` minus one, inclusive, which determines which column of product IDs `eliminate_product_ids` refers to. By default, it refers to the first column, which is index `0`.
- **market_id** (*object, optional*) – ID of the market in which to compute consumer surplus. By default, consumer surpluses are computed in all markets and stacked.

Returns Estimated population-normalized consumer surpluses, CS (or individuals' surpluses if `keep_all` is `True`). If `market_ids` was not specified, rows are in the same order as `Problem.unique_market_ids`. If `keep_all` is `True`, columns for a market are in the same order as agents for the market. If a market has fewer agents than others, extra columns will contain `numpy.nan`.

Return type *ndarray*

Examples

- *Tutorial*

A parametric bootstrap can be used, for example, to compute standard errors forpost-estimation outputs. The following method returns a results class with the same methods in the list directly above, which returns a distribution of post-estimation outputs corresponding to different bootstrapped samples.

`ProblemResults.bootstrap`([draws, seed, ...])	Use a parametric bootstrap to create an empirical distribution of results.

5.5.28 pyblp.ProblemResults.bootstrap

`ProblemResults.`**bootstrap**(*draws=1000*, *seed=None*, *iteration=None*, *constant_costs=True*)

Use a parametric bootstrap to create an empirical distribution of results.

The constructed `BootstrappedResults` can be used just like `ProblemResults` to compute various post-estimation outputs for different markets. The only difference is that `BootstrappedResults` methods return arrays with an extra first dimension, along which bootstrapped results are stacked. These stacked results can be used to construct, for example, confidence intervals for post-estimation outputs.

For each bootstrap draw, parameters are drawn from the estimated multivariate normal distribution of all parameters defined by `ProblemResults.parameters` and `ProblemResults.parameter_covariances` (where the second covariance matrix is divided by N). Any bounds configured in `Problem.solve()` will also bound parameter draws. Each parameter draw is used to compute the implied mean utility, δ, and shares,

s. If a supply side was estimated, the implied marginal costs, c, and prices, p, are computed as well by iterating over the ζ-markup contraction in (3.51). If marginal costs depend on prices through market shares, they will be updated to reflect different prices during each iteration of the routine.

Note: By default, parametric bootstrapping may use a lot of memory. This is because all bootstrapped results (for all `draws`) are stored in memory at the same time. Memory usage can be reduced by calling this method in a loop with `draws = 1`. In each iteration of the loop, compute the desired post-estimation output with the proper method of the returned *`BootstrappedResults`* class and store these outputs.

Parameters

- **`draws`** (*int, optional*) – The number of draws that will be taken from the joint distribution of the parameters. The default value is `1000`.
- **`seed`** (*int, optional*) – Passed to `numpy.random.RandomState` to seed the random number generator before any draws are taken. By default, a seed is not passed to the random number generator.
- **`iteration`** (*Iteration, optional*) – *`Iteration`* configuration used to compute bootstrapped prices by iterating over the ζ-markup equation in (3.51). By default, if a supply side was estimated, this is `Iteration('simple', {'atol': 1e-12})`. Analytic Jacobians are not supported for solving this system. This configuration is not used if a supply side was not estimated.
- **`constant_costs`** (*bool, optional*) – Whether to assume that marginal costs, c, remain constant as equilibrium prices and shares change. By default this is `True`, which means that firms treat marginal costs as constant when setting prices. If set to `False`, marginal costs will be allowed to adjust if `shares` was included in the formulation for X_3 in *`Problem`*. This is not relevant if a supply side was not estimated.

Returns Computed *`BootstrappedResults`*.

Return type *BootstrappedResults*

Examples

- *Tutorial*

Optimal instruments, which also return a results class instead of an array, can be estimated with the following method.

`ProblemResults.compute_optimal_instrume`	Estimate feasible optimal or efficient instruments, Z_D^{opt} and Z_S^{opt}.

5.5.29 pyblp.ProblemResults.compute_optimal_instruments

`ProblemResults.`**`compute_optimal_instruments`**(*method='approximate', draws=1, seed=None, expected_prices=None, iteration=None, constant_costs=True*)

Estimate feasible optimal or efficient instruments, Z_D^{opt} and Z_S^{opt}.

Optimal instruments have been shown, for example, by *Reynaert and Verboven (2014)* and *Conlon and Gortmaker (2020)*, to reduce bias, improve efficiency, and enhance stability of BLP estimates.

Optimal instruments in the spirit of *Amemiya (1977)* or *Chamberlain (1987)* are defined by

$$\begin{bmatrix} Z_{D,jt}^{\text{opt}} \\ Z_{S,jt}^{\text{opt}} \end{bmatrix} = \Sigma_{\xi\omega}^{-1} E \left[\begin{array}{c} \frac{\partial \xi_{jt}}{\partial \theta} \\ \frac{\partial \omega_{jt}}{\partial \theta} \end{array} \middle| Z \right], \tag{5.35}$$

in which Z are all exogenous variables.

Feasible optimal instruments are estimated by evaluating this expression at an estimated $\hat{\theta}$. The expectation is taken by approximating an integral over the joint density of ξ and ω. For each error term realization, if not already estimated, equilibrium prices and shares are computed by iterating over the ζ-markup contraction in (3.51). If marginal costs depend on prices through market shares, they will be updated to reflect different prices during each iteration of the routine.

The expected Jacobians are estimated with the average over all computed Jacobian realizations. The 2×2 normalizing matrix $\Sigma_{\xi\omega}$ is estimated with the sample covariance matrix of the error terms.

Optimal instruments for linear parameters not included in θ are simple product characteristics, so they are not computed here but are rather included in the final set of instruments by `OptimalInstrumentResults.to_problem()`.

Note: When both a supply and demand side are estimated, there are usually collinear rows in (5.35) because of overlapping product characteristics in X_1 and X_3. The expression can be corrected by multiplying it with a conformable matrix of ones and zeros that remove the collinearity problem. The question of which rows to exclude is addressed in `OptimalInstrumentResults.to_problem()`.

Warning: Currently, only optimal instruments for the standard demand- and supply-side moments are supported. If `covariance_instruments` were specified in `product_data`, the computed optimal instruments will only be optimal with respect to the demand- and supply-side moments, not with respect to the addition of any covariance moments as well.

Parameters

- **`method`** (*str, optional*) – The method by which the integral over the joint density of ξ and ω is approximated. The following methods are supported:
  - `'approximate'` (default) - Evaluate the Jacobians at the expected value of the error terms: zero (`draws` will be ignored).
  - `'normal'` - Draw from the normal approximation to the joint distribution of the error terms and take the average over the computed Jacobians (`draws` determines the number of draws).
  - `'empirical'` - Draw with replacement from the empirical joint distribution of the error terms and take the average over the computed Jacobians (`draws` determines the number of draws).
- **`draws`** (*int, optional*) – The number of draws that will be taken from the joint distribution of the error terms. This is ignored if `method` is `'approximate'`. Because the default `method` is `'approximate'`, the default number of draws is `1`, even though it will be ignored. For `'normal'` or empirical, larger numbers such as `100` or `1000` are recommended.
- **`seed`** (*int, optional*) – Passed to `numpy.random.RandomState` to seed the random number generator before any draws are taken. By default, a seed is not passed to the random number generator.

- **expected_prices** (*array-like, optional*) – Vector of expected prices conditional on all exogenous variables, $E[p \mid Z]$. By default, if a supply side was estimated and `shares` did not enter into the formulation for X_3 in *Problem*, `iteration` is used. Otherwise, this is by default estimated with the fitted values from a reduced form regression of endogenous prices onto Z_D.
- **iteration** (*Iteration, optional*) – *Iteration* configuration used to estimate expected prices by iterating over the ζ-markup contraction in (3.51). By default, if a supply side was estimated, this is `Iteration('simple', {'atol': 1e-12})`. Analytic Jacobians are not supported for solving this system. This configuration is not used if `expected_prices` is specified.
- **constant_costs** (*bool, optional*) – Whether to assume that marginal costs, c, remain constant as equilibrium prices and shares change. By default this is `True`, which means that firms treat marginal costs as constant when setting prices. If set to `False`, marginal costs will be allowed to adjust if `shares` was included in the formulation for X_3 in *Problem*. This is not relevant if a supply side was not estimated.

Returns Computed *OptimalInstrumentResults*.

Return type *OptimalInstrumentResults*

Examples

- *Tutorial*

Importance sampling can be used to create new integration nodes and weights. Its method also returns a results class.

ProblemResults.importance_sampling(draws[, ...])	Use importance sampling to construct nodes and weights for integration.

5.5.30 pyblp.ProblemResults.importance_sampling

`ProblemResults.`**`importance_sampling`**(*draws*, *ar_constant=1.0*, *seed=None*, *agent_data=None*, *integration=None*, *delta=None*)

Use importance sampling to construct nodes and weights for integration.

Importance sampling is done with the accept/reject procedure of *Berry, Levinsohn, and Pakes (1995)*. First, `agent_data` and/or `integration` are used to provide a large number of candidate sampling nodes ν and any demographics d.

Out of these candidate agent data, each candidate agent i in market t is accepted with probability $\frac{1-s_{i0t}}{M}$ where $M \geq 1$ is some accept-reject constant. The probability of choosing an inside good $1 - s_{i0t}$, is evaluated at the estimated $\hat{\theta}$ and $\delta(\hat{\theta})$.

Optionally, *ProblemResults.compute_delta()* can be used to provide a more precise $\delta(\hat{\theta})$ than the estimated *ProblemResults.delta*. The idea is that more precise agent data (i.e., more integration nodes) would be infeasible to use during estimation, but is feasible here because $\delta(\hat{\theta})$ only needs to be computed once given a $\hat{\theta}$.

Out of the remaining accepted agents, I_t equal to `draws` are randomly selected within each market t and assigned integration weights $w_{it} = \frac{1}{I_t} \cdot \frac{1-s_{0t}}{1-s_{i0t}}$.

If this procedure accepts fewer than `draws` agents in a market, an exception will be raised. A good rule of thumb is to provide more candidate draws in each market than $\frac{M \times I_t}{1-s_{0t}}$.

Parameters

- **draws** (*int, optional*) – Number of draws to take from `sampling_agent_data` in each market.
- **ar_constant** (*float, optional*) – Accept/reject constant $M \geq 1$, which is by default, `1.0`.
- **seed** (*int, optional*) – Passed to `numpy.random.RandomState` to seed the random number generator before importance sampling is done. By default, a seed is not passed to the random number generator.
- **agent_data** (*structured array-like, optional*) – Agent data from which draws will be sampled, which should have the same structure as `agent_data` in *Problem*. The `weights` field does not need to be specified, and if it is specified it will be ignored. By default, the same agent data used to solve the problem will be used.
- **integration** (*Integration, optional*) – *Integration* configuration for how to build nodes from which draws will be sampled, which will replace any `nodes` field in `sampling_agent_data`. This configuration is required if `sampling_agent_data` is specified without a `nodes` field.
- **delta** (*array-like, optional*) – More precise $\delta(\hat{\theta})$ than the estimated *ProblemResults.delta*, which can be computed by passing a more precise integration rule to *ProblemResults.compute_delta()*. By default, *ProblemResults.delta* is used.

Returns Computed *ImportanceSamplingResults*.

Return type *ImportanceSamplingResults*

Examples

- *Tutorial*

The following methods can compute micro moment values, compute scores from micro data, or simulate such data.

ProblemResults.compute_micro_values(...)	Estimate micro moment values, $f_m(v)$.
ProblemResults.compute_micro_scores(dataset, ...)	Compute scores for observations $n \in N_d$ from a micro dataset d.
ProblemResults.compute_agent_scores(dataset)	Compute scores for all agent-choices, treated as observations $n \in N_d$ from a micro dataset d.
ProblemResults.simulate_micro_data(dataset)	Simulate observations $n \in N_d$ from a micro dataset d.

5.5.31 pyblp.ProblemResults.compute_micro_values

`ProblemResults.`**compute_micro_values**(*micro_moments*)

Estimate micro moment values, $f_m(v)$.

Parameters **micro_moments** (*sequence of MicroMoment*) – *MicroMoment* instances. The `value` argument is ignored.

Returns Micro moment values $f_m(v)$.

Return type *ndarray*

Examples

- *Tutorial*

5.5.32 pyblp.ProblemResults.compute_micro_scores

`ProblemResults.compute_micro_scores`(*dataset*, *micro_data*, *integration=None*)

Compute scores for observations $n \in N_d$ from a micro dataset d.

The score for observation $n \in N_d$ is

$$\mathscr{S}_n = \frac{\partial \log \mathscr{P}_n}{\partial \theta'}, \tag{5.36}$$

in which the conditional probability of observation n is

$$\mathscr{P}_n = \frac{\sum_{i \in I_n} w_{it_n} s_{ij_n t_n} w_{dij_n t_n}}{\sum_{t \in T} \sum_{i \in I_t} \sum_{j \in J_t \cup \{0\}} w_{it} s_{ijt} w_{dijt}} \tag{5.37}$$

where $i \in I_n$ integrates over unobserved heterogeneity for observation n.

Parameters

- **dataset** (*MicroDataset*) – The `MicroDataset` for which scores will be computed. The `compute_weights` function is called separately for each observation n.
- **micro_data** (*structured array-like*) – Each row corresponds either to an observation n or if there are multiple rows per observation, to an $i \in I_n$ that integrates over unobserved heterogeneity. In addition to the names of any demographics used in the `agent_formulation` and any specification of agent-specific product `'availability'`, the following fields are required:
  - **market_ids** : (*object*) - Market IDs t_n for each observation n.
  - **choice_indices** : (*int*) - Within-market indices of choices j_n. If `compute_weights` passed to the `dataset` returns an array with J_t elements in its second axis, then choice indices take on values from 0 to $J_t - 1$ where 0 corresponds to the first inside good. If it returns an array with $1 + J_t$ elements in its second axis, then choice indices take on values from 0 to J_t where 0 corresponds to the outside good.

  If the `dataset` is configured to support second choice data, second choices are also required:

  - **second_choice_indices** : (*int, optional*) - Within-market indices of second choices k_n. If `compute_weights` passed to the `dataset` returns an array with J_t elements in its third axis, then second choice indices take on values from 0 to $J_t - 1$ where 0 corresponds to the first inside good. If it returns an array with $1 + J_t$ elements in its third axis, then second choice indices take on values from 0 to J_t where 0 corresponds to the outside good.

  The following fields are required if `integration` is not specified:

  - **micro_ids** : (*object, optional*) - IDs corresponding to observations n, which should be pre-sorted, from smallest to largest.
  - **weights** : (*numeric, optional*) - Integration weights, w_{it_n}, for integration over unobserved heterogeneity $i \in I_n$.
  - **nodes** : (*numeric, optional*) - Unobserved agent characteristics called integration nodes, ν. If there are more than K_2 columns (the number of demand-side nonlinear product characteristics), only the first K_2 will be retained. If any columns of `sigma` are fixed at zero, only the first few columns of these nodes will be used.

  If these fields are specified, each row corresponds to an $i \in I_n$, and there should generally be multiple rows per observation n.

The convenience function *build_integration()* can be useful when constructing custom nodes and weights.

Note: If `nodes` has multiple columns, it can be specified as a matrix or broken up into multiple one-dimensional fields with column index suffixes that start at zero. For example, if there are three columns of nodes, a `nodes` field with three columns can be replaced by three one-dimensional fields: `nodes0`, `nodes1`, and `nodes2`.

- **integration** (*Integration, optional*) – *Integration* configuration for how to build `nodes` and `weights` fields in `micro_data` for each observation n. If this configuration is specified, any `micro_ids`, `weights`, and `nodes` in `micro_data` will be ignored.

  If specified, each row of `micro_data` is treated as corresponding to a unique observation n, and will be duplicated by as many rows of nodes as are created by the *Integration* configuration. Specifically, up to K_2 columns of nodes (the number of demand-side nonlinear product characteristics) will be built for each observation n. If there are zeros on the diagonal of Σ, nodes will not be built for those characteristics, to cut down on memory usage.

Returns

Scores $\mathscr{S}_n$. The list is in the same order as *ProblemResults.theta* (also see *ProblemResults.theta_labels*). Each element of the list is an array of scores for the corresponding parameter. The array is in the same order as observations appear in the `micro_data`. Note that it is possible for parameters in *ProblemResults.theta* to mechanically have zero scores, for example if they are on a constant demographic.

Taking the mean of a parameter's scores delivers the observed `value` for an optimal *MicroMoment* that matches the score for that parameter.

If any scores are `numpy.nan`, this means that the probability of that observation is $\mathscr{P}_n = 0$, suggesting that the observation was not generated by the sampling process defined by the `dataset`.

Return type *list*

5.5.33 pyblp.ProblemResults.compute_agent_scores

`ProblemResults.`**compute_agent_scores**(*dataset*, *micro_data=None*, *integration=None*)

Compute scores for all agent-choices, treated as observations $n \in N_d$ from a micro dataset d.

This method is the same as *ProblemResults.compute_micro_scores()*, except it computes scores for all possible choices of all *Problem.agents*. Each agent-choice is treated as a separate observation n. Instead of returning an array, this method returns a mapping from market IDs to scores, to facilitate use by `compute_values` of an optimal *MicroMoment*.

Parameters

- **dataset** (*MicroDataset*) – The *MicroDataset* for which scores will be computed. The `compute_weights` function is called separately for each observation n.
- **micro_data** (*structured array-like, optional*) – By default, each row in *Problem.agents* and each possible choice is treated as an observation n. In this case, `integration` should generally be specified to define integration $i \in I_n$ over unobserved heterogeneity.

If `micro_data` is specified, it should be of the form required by *ProblemResults.compute_micro_scores()*, except without `choice_indices` or `second_choice_indices`, since scores will be computed for all choices.

- **integration** (*Integration, optional*) – *Integration* configuration of the form required by *ProblemResults.compute_micro_scores()*.

Returns

Scores $\mathscr{S}_n$. The list is in the same order as *ProblemResults.theta* (also see *ProblemResults.theta_labels*). Each element of the list is a mapping from market IDs supported by the `dataset` to an array of scores for the corresponding parameter and market. The array's dimensions correspond to the dimensions of the weights returned by `compute_weights` passed to `dataset`. Note that it is possible for parameters in *ProblemResults.theta* to mechanically have zero scores, for example if they are on a constant demographic.

To build an optimal *MicroMoment* that matches the score for a parameter, `compute_values` in its single *MicroPart* should select the array corresponding to that parameter and the requested market `t`. Any `numpy.nan` values in this array correspond to agent-choices that are assigned a probability of $\mathscr{P}_n = 0$ by the sampling process defined by `dataset`, so should be replaced by some arbitrary number (e.g., by passing the array of scores through `numpy.nan_to_num`).

Return type *list*

5.5.34 pyblp.ProblemResults.simulate_micro_data

`ProblemResults.`**`simulate_micro_data`**(*dataset*, *seed=None*)

Simulate observations $n \in N_d$ from a micro dataset d.

Each micro observation n underlying the dataset d is simulated according to agent weights w_{it}, choice probabilities s_{ijt}, and survey weights w_{dijt}.

Parameters

- **dataset** (*MicroDataset*) – The *MicroDataset* for which micro data will be simulated.
- **seed** (*int, optional*) – Passed to `numpy.random.RandomState` to seed the random number generator before data are simulated. By default, a seed is not passed to the random number generator.

Returns

Micro data with as many rows as `observations` passed to the `dataset`. Fields:

- **micro_ids** : (*object*) - IDs corresponding to observations n.
- **market_ids** : (*object*) - Market IDs t_n for each observation n.
- **agent_indices** : (*int*) - Within-market indices of agents i_n that take on values from 0 to $I_t - 1$.
- **choice_indices** : (*int*) - Within-market indices of simulated choices j_n. If `compute_weights` passed to the `dataset` returns an array with J_t elements in its second axis, then choice indices take on values from 0 to $J_t - 1$ where 0 corresponds to the first inside good. If it returns an array with $1 + J_t$ elements in its second axis, then choice indices take on values from 0 to J_t where 0 corresponds to the outside good.

If the `dataset` is configured to support second choice data, second choices will also be simulated:

- **second_choice_indices** : (*int*) - Within-market indices of simulated second choices k_n. If `compute_weights` passed to the `dataset` returns an array with J_t elements in its third axis, then second choice indices take on values from 0 to $J_t - 1$ where 0 corresponds to the first inside good. If it returns an array with $1 + J_t$ elements in its third axis, then second choice indices take on values from 0 to J_t where 0 corresponds to the outside good.

Integration nodes and demographics can be merged in on the `market_ids` and `agent_indices` fields. Product characteristics can be merged in on the `market_ids` and `choice_indices` fields. Product characteristics of any second choices can be merged in on the `market_ids` and `second_choice_indices` fields.

Return type *recarray*

Examples

- *Tutorial*

5.6 Bootstrapped Problem Results Class

Parametric bootstrap computation returns the following class.

BootstrappedResults	Bootstrapped results of a solved problem.

5.6.1 pyblp.BootstrappedResults

class pyblp.**BootstrappedResults**

Bootstrapped results of a solved problem.

This class has slightly modified versions of the following *ProblemResults* methods:

- *ProblemResults.compute_aggregate_elasticities()*
- *ProblemResults.compute_elasticities()*
- *ProblemResults.compute_demand_jacobians()*
- *ProblemResults.compute_demand_hessians()*
- *ProblemResults.compute_profit_hessians()*
- *ProblemResults.compute_diversion_ratios()*
- *ProblemResults.compute_long_run_diversion_ratios()*
- *ProblemResults.compute_probabilities()*
- *ProblemResults.extract_diagonals()*
- *ProblemResults.extract_diagonal_means()*
- *ProblemResults.compute_delta()*
- *ProblemResults.compute_costs()*
- *ProblemResults.compute_passthrough()*
- *ProblemResults.compute_approximate_prices()*
- *ProblemResults.compute_prices()*

- *ProblemResults.compute_shares()*
- *ProblemResults.compute_hhi()*
- *ProblemResults.compute_markups()*
- *ProblemResults.compute_profits()*
- *ProblemResults.compute_consumer_surpluses()*

The difference is that each method returns an array with an extra first dimension along which bootstrapped results are stacked These stacked results can be used to construct, for example, confidence intervals for post-estimation outputs. Similarly, arrays of data (except for firm IDs and ownership matrices) passed as arguments to methods should have an extra first dimension of size *BootstrappedResults.draws*.

problem_results
ProblemResults that was used to compute these bootstrapped results.

Type *ProblemResults*

bootstrapped_sigma
Bootstrapped Cholesky decomposition of the covariance matrix for unobserved taste heterogeneity, Σ.

Type *ndarray*

bootstrapped_pi
Bootstrapped parameters that measures how agent tastes vary with demographics, Π.

Type *ndarray*

bootstrapped_rho
Bootstrapped parameters that measure within nesting group correlations, ρ.

Type *ndarray*

bootstrapped_phi
Bootstrapped parameter that measures demand-side unobservable autocorrelation, ϕ.

Type *ndarray*

bootstrapped_beta
Bootstrapped demand-side linear parameters, β.

Type *ndarray*

bootstrapped_gamma
Bootstrapped supply-side linear parameters, γ.

Type *ndarray*

bootstrapped_prices
Bootstrapped prices, p. If a supply side was not estimated, these are unchanged prices. Otherwise, they are equilibrium prices implied by each draw.

Type *ndarray*

bootstrapped_shares
Bootstrapped market shares, s, implied by each draw.

Type *ndarray*

bootstrapped_delta
Bootstrapped mean utility, δ, implied by each draw.

Type *ndarray*

computation_time
Number of seconds it took to compute the bootstrapped results.

Type *float*

draws
Number of bootstrap draws.

Type *int*

fp_converged
Flags for convergence of the iteration routine used to compute equilibrium prices in each market. Rows are in the same order as `Problem.unique_market_ids` and column indices correspond to draws.

Type *ndarray*

fp_iterations
Number of major iterations completed by the iteration routine used to compute equilibrium prices in each market for each draw. Rows are in the same order as `Problem.unique_market_ids` and column indices correspond to draws.

Type *ndarray*

contraction_evaluations
Number of times the contraction used to compute equilibrium prices was evaluated in each market for each draw. Rows are in the same order as `Problem.unique_market_ids` and column indices correspond to draws.

Type *ndarray*

Examples

- *Tutorial*

This class has many of the same methods as `ProblemResults()`. It can also be pickled or converted into a dictionary.

`BootstrappedResults.to_pickle`(path)	Save these results as a pickle file.
`BootstrappedResults.to_dict`([attributes])	Convert these results into a dictionary that maps attribute names to values.

5.6.2 pyblp.BootstrappedResults.to_pickle

BootstrappedResults.**to_pickle**(*path*)
Save these results as a pickle file.

Parameters **path** (*str or Path*) – File path to which these results will be saved.

5.6.3 pyblp.BootstrappedResults.to_dict

BootstrappedResults.**to_dict**(*attributes=('bootstrapped_sigma', 'bootstrapped_pi', 'bootstrapped_rho', 'bootstrapped_phi', 'bootstrapped_beta', 'bootstrapped_gamma', 'bootstrapped_prices', 'bootstrapped_shares', 'bootstrapped_delta', 'computation_time', 'draws', 'fp_converged', 'fp_iterations', 'contraction_evaluations')*)
Convert these results into a dictionary that maps attribute names to values.

Parameters **attributes** (*sequence of str, optional*) – Name of attributes that will be added to the dictionary. By default, all *BootstrappedResults* attributes are added except for *BootstrappedResults.problem_results*.

Returns Mapping from attribute names to values.

Return type *dict*

5.7 Optimal Instrument Results Class

Optimal instrument computation returns the following results class.

OptimalInstrumentResults	Results of optimal instrument computation.

5.7.1 pyblp.OptimalInstrumentResults

class pyblp.**OptimalInstrumentResults**

Results of optimal instrument computation.

The *OptimalInstrumentResults.to_problem()* method can be used to update the original *Problem* with the computed optimal instruments.

problem_results

ProblemResults that was used to compute these optimal instrument results.

Type *ProblemResults*

demand_instruments

Estimated optimal demand-side instruments for θ, denoted Z_D^{opt}.

Type *ndarray*

supply_instruments

Estimated optimal supply-side instruments for θ, denoted Z_S^{opt}.

Type *ndarray*

supply_shifter_formulation

Formulation configuration for supply shifters that will by default be included in the full set of optimal demand-side instruments. This is only constructed if a supply side was estimated, and it can be changed in *OptimalInstrumentResults.to_problem()*. By default, this is the formulation for X_3^{ex} from *Problem* excluding any variables in the formulation for X_1^{ex}.

Type *Formulation or None*

demand_shifter_formulation

Formulation configuration for demand shifters that will by default be included in the full set of optimal supply-side instruments. This is only constructed if a supply side was estimated, and it can be changed in *OptimalInstrumentResults.to_problem()*. By default, this is the formulation for X_1^{ex} from *Problem* excluding any variables in the formulation for X_3^{ex}.

Type *Formulation or None*

inverse_covariance_matrix

Inverse of the sample covariance matrix of the estimated ξ and ω, which is used to normalize the expected Jacobians. If a supply side was not estimated, this is simply the sample estimate of $1/\sigma_\xi^2$.

Type *ndarray*

expected_xi_by_theta_jacobian
Estimated $E[\frac{\partial \xi}{\partial \theta} \mid Z]$.

Type *ndarray*

expected_omega_by_theta_jacobian
Estimated $E[\frac{\partial \omega}{\partial \theta} \mid Z]$.

Type *ndarray*

expected_prices
Vector of expected prices conditional on all exogenous variables, $E[p \mid Z]$, which may have been specified in `ProblemResults.compute_optimal_instruments()`.

Type *ndarray*

expected_shares
Vector of expected market shares conditional on all exogenous variables, $E[s \mid Z]$.

Type *ndarray*

computation_time
Number of seconds it took to compute optimal excluded instruments.

Type *float*

draws
Number of draws used to approximate the integral over the error term density.

Type *int*

fp_converged
Flags for convergence of the iteration routine used to compute equilibrium prices in each market. Rows are in the same order as `Problem.unique_market_ids` and column indices correspond to draws.

Type *ndarray*

fp_iterations
Number of major iterations completed by the iteration routine used to compute equilibrium prices in each market for each error term draw. Rows are in the same order as `Problem.unique_market_ids` and column indices correspond to draws.

Type *ndarray*

contraction_evaluations
Number of times the contraction used to compute equilibrium prices was evaluated in each market for each error term draw. Rows are in the same order as `Problem.unique_market_ids` and column indices correspond to draws.

Type *ndarray*

Examples

- *Tutorial*

Methods

`to_dict`([attributes])	Convert these results into a dictionary that maps attribute names to values.

Continued on next page

Table 24 – continued from previous page

to_pickle(path)	Save these results as a pickle file.
to_problem([supply_shifter_formulation, ...])	Re-create the problem with estimated feasible optimal instruments.

The results can be pickled or converted into a dictionary.

OptimalInstrumentResults.to_pickle(path)	Save these results as a pickle file.
OptimalInstrumentResults.to_dict([attributes])	Convert these results into a dictionary that maps attribute names to values.

5.7.2 pyblp.OptimalInstrumentResults.to_pickle

OptimalInstrumentResults.**to_pickle**(*path*)

Save these results as a pickle file.

Parameters **path** (*str or Path*) – File path to which these results will be saved.

5.7.3 pyblp.OptimalInstrumentResults.to_dict

OptimalInstrumentResults.**to_dict**(*attributes=('demand_instruments', 'supply_instruments', 'inverse_covariance_matrix', 'expected_xi_by_theta_jacobian', 'expected_omega_by_theta_jacobian', 'expected_prices', 'expected_shares', 'computation_time', 'draws', 'fp_converged', 'fp_iterations', 'contraction_evaluations')*)

Convert these results into a dictionary that maps attribute names to values.

Parameters **attributes** (*sequence of str, optional*) – Name of attributes that will be added to the dictionary. By default, all *OptimalInstrumentResults* attributes are added except for *OptimalInstrumentResults.problem_results*, *OptimalInstrumentResults.supply_shifter_formulation*, and *OptimalInstrumentResults.demand_shifter_formulation*.

Returns Mapping from attribute names to values.

Return type *dict*

Examples

- *Tutorial*

They can also be converted into a *Problem* with the following method.

OptimalInstrumentResults.to_problem([...])	Re-create the problem with estimated feasible optimal instruments.

5.7.4 pyblp.OptimalInstrumentResults.to_problem

OptimalInstrumentResults.**to_problem**(*supply_shifter_formulation=None*, *demand_shifter_formulation=None*, *product_data=None*, *drop_indices=None*)

Re-create the problem with estimated feasible optimal instruments.

The re-created problem will be exactly the same, except that instruments will be replaced with estimated feasible optimal instruments.

Note: Most of the explanation here is only important if a supply side was estimated.

The optimal excluded demand-side instruments consist of the following:

1. Estimated optimal demand-side instruments for θ, Z_D^{opt}, excluding columns of instruments for any parameters on exogenous linear characteristics that were not concentrated out, but rather included in θ by *Problem.solve()*.
2. Optimal instruments for any linear demand-side parameters on endogenous product characteristics, α, which were concentrated out and hence not included in θ. These optimal instruments are simply an integral of the endogenous product characteristics, X_1^{en}, over the joint density of ξ and ω. It is only possible to concentrate out α when there isn't a supply side, so the approximation of these optimal instruments is simply X_1^{en} evaluated at the constant vector of expected prices, $E[p \mid Z]$, specified in *ProblemResults.compute_optimal_instruments()*.
3. If a supply side was estimated, any supply shifters, which are by default formulated by *OptimalInstrumentResults.supply_shifter_formulation*: all characteristics in X_3^{ex} not in X_1^{ex}.

Similarly, if a supply side was estimated, the optimal excluded supply-side instruments consist of the following:

1. Estimated optimal supply-side instruments for θ, Z_S^{opt}, excluding columns of instruments for any parameters on exogenous linear characteristics that were not concentrated out, but rather included in θ by *Problem.solve()*.
2. Optimal instruments for any linear supply-side parameters on endogenous product characteristics, γ^{en}, which were concentrated out an hence not included in θ. This is only relevant if `shares` were included in the formulation for X_3 in *Problem*. The corresponding optimal instruments are simply an integral of the endogenous product characteristics, X_3^{en}, over the joint density of ξ and ω. The approximation of these optimal instruments is simply X_3^{en} evaluated at the market shares that arise under the constant vector of expected prices, $E[p \mid Z]$, specified in *ProblemResults.compute_optimal_instruments()*.

2. If a supply side was estimated, any demand shifters, which are by default formulated by *OptimalInstrumentResults.demand_shifter_formulation*: all characteristics in X_1^{ex} not in X_3^{ex}.

As usual, the excluded demand-side instruments will be supplemented with X_1^{ex} and the excluded supply-side instruments will be supplemented with X_3^{ex}. The same fixed effects configured in *Problem* will be absorbed.

Warning: If a supply side was estimated, the addition of supply- and demand-shifters may create collinearity issues. Make sure to check that shifters and other product characteristics are not collinear.

Parameters

- **supply_shifter_formulation** (*Formulation, optional*) – *Formulation* configuration for supply shifters to be included in the set of optimal demand-side instruments. This is only used if a supply side was estimated. Intercepts will be ignored. By default, *OptimalInstrumentResults.supply_shifter_formulation* is used.
- **demand_shifter_formulation** (*Formulation, optional*) – *Formulation* configuration for demand shifters to be included in the set of optimal supply-side instruments. This is only used if a supply side was estimated. Intercepts will be ignored. By default, *OptimalInstrumentResults.demand_shifter_formulation* is used.
- **product_data** (*structured array-like, optional*) – Product data used instead of what was saved from `product_data` when initializing the original *Problem*. This may need to be specified if either the supply or demand shifter formulation contains some term that was not stored into memory, such as a categorical variable or a mathematical expression.
- **drop_indices** (*sequence of int, optional*) – Which column indices to drop from *OptimalInstrumentResults.demand_instruments* and *OptimalInstrumentResults.supply_instruments*. By default, the only columns dropped are those that correspond to parameters in θ on exogenous linear characteristics.

Returns *OptimalInstrumentProblem*, which is a *Problem* updated to use the estimated optimal instruments.

Return type *OptimalInstrumentProblem*

Examples

- *Tutorial*

This method returns the following class, which behaves exactly like a *Problem*.

OptimalInstrumentProblem	A BLP problem updated with optimal excluded instruments.

5.7.5 pyblp.OptimalInstrumentProblem

class pyblp.**OptimalInstrumentProblem**

A BLP problem updated with optimal excluded instruments.

This class can be used exactly like *Problem*.

5.8 Importance Sampling Results Class

Importance sampling returns the following results class:

ImportanceSamplingResults	Results of importance sampling.

5.8.1 pyblp.ImportanceSamplingResults

class pyblp.**ImportanceSamplingResults**

Results of importance sampling.

Along with the sampled agents, these results also contain a number of useful importance sampling diagnostics from *Owen (2013)*.

The `ImportanceSamplingResults.to_problem()` method can be used to update the original `Problem` with the importance sampling agent data.

problem_results
`ProblemResults` that was used to compute these importance sampling results.

Type *ProblemResults*

sampled_agents
Importance sampling agent data structured as `Agents`. The `data_to_dict()` function can be used to convert this into a more usable data type.

Type *Agents*

computation_time
Number of seconds it took to do importance sampling.

Type *float*

draws
Number of importance sampling draws in each market.

Type *int*

diagnostic_market_ids
Market IDs the correspond to the ordering of the following arrays of weight diagnostics.

Type *ndarray*

weight_sums
Sum of weights in each market: $\sum_i w_{it}$. If importance sampling was successful, weights should not sum to numbers too far from one.

Type *ndarray*

effective_draws
Effective sample sizes in each market: $\frac{(\sum_i w_{it})^2}{\sum_i w_{it}^2}$.

Type *ndarray*

effective_draws_for_variance
Effective sample sizes for variance estimates in each market: $\frac{(\sum_i w_{it}^2)^2}{\sum_i w_{it}^4}$.

Type *ndarray*

effective_draws_for_skewness
Effective sample sizes for gauging skewness in each market: $\frac{(\sum_i w_{it}^2)^3}{(\sum_i w_{it}^3)^2}$.

Type *ndarray*

Examples

- *Tutorial*

Methods

to_dict([attributes])	Convert these results into a dictionary that maps attribute names to values.
to_pickle(path)	Save these results as a pickle file.
to_problem()	Re-create the problem with the agent data constructed from importance sampling.

The results can be pickled or converted into a dictionary.

ImportanceSamplingResults.to_pickle(path)	Save these results as a pickle file.
ImportanceSamplingResults.to_dict([attributes])	Convert these results into a dictionary that maps attribute names to values.

5.8.2 pyblp.ImportanceSamplingResults.to_pickle

ImportanceSamplingResults.**to_pickle**(*path*)

Save these results as a pickle file.

Parameters **path** (*str or Path*) – File path to which these results will be saved.

5.8.3 pyblp.ImportanceSamplingResults.to_dict

ImportanceSamplingResults.**to_dict**(*attributes=('sampled_agents', 'computation_time', 'draws', 'diagnostic_market_ids', 'weight_sums', 'effective_draws', 'effective_draws_for_variance', 'effective_draws_for_skewness')*)

Convert these results into a dictionary that maps attribute names to values.

Parameters **attributes** (*sequence of str, optional*) – Names of attributes that will be added to the dictionary. By default, all *ImportanceSamplingResults* attributes are added except for *ImportanceSamplingResults.problem_results*.

Returns Mapping from attribute names to values.

Return type *dict*

Examples

- *Tutorial*

They can also be converted into a *Problem* with the following method.

ImportanceSamplingResults.to_problem()	Re-create the problem with the agent data constructed from importance sampling.

5.8.4 pyblp.ImportanceSamplingResults.to_problem

ImportanceSamplingResults.**to_problem**()

Re-create the problem with the agent data constructed from importance sampling.

The re-created problem will be exactly the same, except `Problem.agents` will be replaced with `ImportanceSamplingResults.sampled_agents`.

Returns `ImportanceSamplingProblem`, which is a `Problem` updated to use agent data constructed from importance sampling.

Return type *ImportanceSamplingProblem*

Examples

- *Tutorial*

This method returns the following class, which behaves exactly like a `Problem`.

`ImportanceSamplingProblem`	A BLP problem updated after importance sampling.

5.8.5 pyblp.ImportanceSamplingProblem

class pyblp.**ImportanceSamplingProblem**

A BLP problem updated after importance sampling.

This class can be used exactly like `Problem`.

5.9 Simulation Class

The following class allows for evaluation of more complicated counterfactuals than is possible with `ProblemResults` methods, or for simulation of synthetic data from scratch.

`Simulation`(product_formulations, . . . [, . . .])	Simulation of data in BLP-type models.

5.9.1 pyblp.Simulation

class pyblp.**Simulation**(*product_formulations*, *product_data*, *beta*, *sigma=None*, *pi=None*, *gamma=None*, *rho=None*, *phi=None*, *agent_formulation=None*, *agent_data=None*, *integration=None*, *xi=None*, *omega=None*, *xi_variance=1*, *omega_variance=1*, *correlation=0.9*, *rc_types=None*, *epsilon_scale=1.0*, *costs_type='linear'*, *seed=None*)

Simulation of data in BLP-type models.

Any data left unspecified are simulated during initialization. Simulated prices and shares can be replaced by `Simulation.replace_endogenous()` with equilibrium values that are consistent with true parameters. Less commonly, simulated exogenous variables can be replaced instead by `Simulation.replace_exogenous()`. To choose your own prices, refer to the first note in `Simulation.replace_endogenous()`. Simulations are typically used for two purposes:

1. Solving for equilibrium prices and shares under more complicated counterfactuals than is possible with `ProblemResults.compute_prices()` and `ProblemResults.compute_shares()`. For example, this class can be initialized with estimated parameters, structural errors, and marginal costs from a `ProblemResults()`, but with changed data (fewer products, new products, different characteristics, etc.) and `Simulation.replace_endogenous()` can be used to compute the corresponding prices and shares.

2. Simulation of BLP-type models from scratch. For example, a model with fixed true parameters can be simulated many times, converted into problems with *SimulationResults.to_problem()*, and solved with *Problem.solve()* to evaluate in a Monte Carlo study how well the true parameters can be recovered.

If data for variables (used to formulate product characteristics in X_1, X_2, and X_3, as well as agent demographics, d, and endogenous prices and market shares p and s) are not provided, the values for each unspecified variable are drawn independently from the standard uniform distribution. In each market t, market shares are divided by the number of products in the market J_t. Typically, *Simulation.replace_endogenous()* is used to replace prices and shares with equilibrium values that are consistent with true parameters.

If data for unobserved demand-and supply-side product characteristics, ξ and ω, are not provided, they are by default drawn from a mean-zero bivariate normal distribution.

After variables are loaded or simulated, any unspecified integration nodes and weights, ν and w, are constructed according to a specified *Integration* configuration.

Parameters

- **product_formulations** (*Formulation or sequence of Formulation*) – *Formulation* configuration or a sequence of up to three *Formulation* configurations for the matrix of demand-side linear product characteristics, X_1, for the matrix of demand-side nonlinear product characteristics, X_2, and for the matrix of supply-side characteristics, X_3, respectively. If the formulation for X_2 is not specified or is `None`, the logit (or nested logit) model will be simulated.

  The `shares` variable should not be included in the formulations for X_1 or X_2. If `shares` is included in the formulation for X_3 and `product_data` does not include `shares`, one will likely want to set `constant_costs=False` in *Simulation.replace_endogenous()*.

  The `prices` variable should not be included in the formulation for X_3, but it should be included in the formulation for X_1 or X_2 (or both). Variables that cannot be loaded from `product_data` will be drawn from independent standard uniform distributions. Unlike in *Problem*, fixed effect absorption is not supported during simulation.

  Warning: Characteristics that involve prices, p, or shares, s, should always be formulated with the `prices` and `shares` variables, respectively. If another name is used, *Simulation* will not understand that the characteristic is endogenous. For example, to include a p^2 characteristic, include `I(prices**2)` in a formula instead of manually constructing and including a `prices_squared` variable.

- **product_data** (*structured array-like*) – Each row corresponds to a product. Markets can have differing numbers of products. The convenience function *build_id_data()* can be used to construct the following required ID data:

  – **market_ids** : (*object*) - IDs that associate products with markets.

  – **firm_ids** : (*object*) - IDs that associate products with firms.

  Custom ownership matrices can be specified as well:

  – **ownership** : (*numeric, optional*) - Custom stacked $J_t \times J_t$ ownership or product holding matrices, $\mathscr{H}$, for each market t, which can be built with *build_ownership()*. By default, standard ownership matrices are built only when they are needed to reduce memory usage. If specified, there should be as many columns as there are products in the market with the most products. Rightmost columns in markets with fewer products will be ignored.

Note: The `ownership` field can either be a matrix or can be broken up into multiple one-dimensional fields with column index suffixes that start at zero. For example, if there are three products in each market, a `ownership` field with three columns can be replaced by three one-dimensional fields: `ownership0`, `ownership1`, and `ownership2`.

To simulate a nested logit or random coefficients nested logit (RCNL) model, nesting groups must be specified:

– **nesting_ids** (*object, optional*) - IDs that associate products with nesting groups. When these IDs are specified, `rho` must be specified as well.

It may be convenient to define IDs for different products:

– **product_ids** (*object, optional*) - IDs that identify products within markets. There can be multiple columns.

To specify unobservable autocorrelation with `phi`, indices that define lags of the data must be specified:

– **lag_indices** : (*int, optional*) - Indices that take on values from 0 to $N - 1$, which define the lag operator L on the data. For example, if markets t are simply time periods and the identity of products j are persistent across periods, then $Lx_{jt} = x_{j,t-1}$.

The value of the current row index indicates that this is the initial period for a product. Otherwise, the value should correspond to the row that is the lagged version of the current row.

Along with `market_ids`, `firm_ids`, `product_ids`, and `nesting_ids`, the names of any additional fields can typically be used as variables in `product_formulations`. However, there are a few variable names such as `'X1'`, which are reserved for use by *Products*.

- **beta** (*array-like*) – Vector of demand-side linear parameters, β. Elements correspond to columns in X_1, which is formulated by `product_formulations`.
- **sigma** (*array-like, optional*) – Lower-triangular Cholesky root of the covariance matrix for unobserved taste heterogeneity, Σ. Rows and columns correspond to columns in X_2, which is formulated by `product_formulations`. If X_2 is not formulated, this should not be specified, since the logit model will be simulated.
- **pi** (*array-like, optional*) – Parameters that measure how agent tastes vary with demographics, Π. Rows correspond to the same product characteristics as in `sigma`. Columns correspond to columns in d, which is formulated by `agent_formulation`. If d is not formulated, this should not be specified.
- **gamma** (*array-like, optional*) – Vector of supply-side linear parameters, γ. Elements correspond to columns in X_3, which is formulated by `product_formulations`. If X_3 is not formulated, this should not be specified.
- **rho** (*array-like, optional*) – Parameters that measure within nesting group correlation, ρ. If this is a scalar, it corresponds to all groups defined by the `nesting_ids` field of `product_data`. If this is a vector, it must have H elements, one for each nesting group. Elements correspond to group IDs in the sorted order of *Simulation.unique_nesting_ids*. If nesting IDs are not specified, this should not be specified either.
- **phi** (*float, optional*) – Parameters measuring unobservable autocorrelation,

$$\phi = \begin{bmatrix} \phi_\xi & \phi_{\xi\omega} \\ \phi_{\omega\xi} & \phi_\omega \end{bmatrix}, \tag{5.38}$$

which must be specified if `lag_indices` in `product_data` are specified. This is ignored during simulation if `xi` and `omega` are specified. Otherwise, if specified, unobservables are drawn according to AR(1) processes:

$$\begin{aligned} \xi_{jt} &= \phi_\xi \cdot L\xi_{jt} + \phi_{\xi\omega} \cdot L\omega_{jt} + \tilde{\xi}_{jt}, \\ \omega_{jt} &= \phi_\omega \cdot L\omega_{jt} + \phi_{\omega\xi} \cdot L\xi_{jt} + \tilde{\omega}_{jt}, \end{aligned} \tag{5.39}$$

where the `lag_indices` field in `product_data` defines the lag operator L.

- **`agent_formulation`** (*Formulation, optional*) – *`Formulation`* configuration for the matrix of observed agent characteristics called demographics, d, which will only be included in the model if this formulation is specified. Any variables that cannot be loaded from `agent_data` will be drawn from independent standard uniform distributions.

- **`agent_data`** (*structured array-like, optional*) – Each row corresponds to an agent. Markets can have differing numbers of agents. Since simulated agents are only used if there are demand-side nonlinear product characteristics, agent data should only be specified if X_2 is formulated in `product_formulations`. If agent data are specified, market IDs are required:

  – **market_ids** : (*object, optional*) - IDs that associate agents with markets. The set of distinct IDs should be the same as the set in `product_data`. If `integration` is specified, there must be at least as many rows in each market as the number of nodes and weights that are built for the market.

  If `integration` is not specified, the following fields are required:

  – **weights** : (*numeric, optional*) - Integration weights, w, for integration over agent choice probabilities.

  – **nodes** : (*numeric, optional*) - Unobserved agent characteristics called integration nodes, ν. If there are more than K_2 columns (the number of demand-side nonlinear product characteristics), only the first K_2 will be used. If any columns of `sigma` are fixed at zero, only the first few columns of these nodes will be used.

  The convenience function *`build_integration()`* can be useful when constructing custom nodes and weights.

  Note: If `nodes` has multiple columns, it can be specified as a matrix or broken up into multiple one-dimensional fields with column index suffixes that start at zero. For example, if there are three columns of nodes, a `nodes` field with three columns can be replaced by three one-dimensional fields: `nodes0`, `nodes1`, and `nodes2`.

  It may be convenient to define IDs for different agents:

  – **agent_ids** (*object, optional*) - IDs that identify agents within markets. There can be multiple of the same ID within a market.

  Along with `market_ids` and `agent_ids`, the names of any additional fields can typically be used as variables in `agent_formulation`. The exception is the name `'demographics'`, which is reserved for use by *`Agents`*.

  In addition to standard demographic variables d_{it}, it is also possible to specify product-specific demographics d_{ijt}. A typical example is geographic distance of agent i from product j. If `agent_formulation` has, for example, `'distance'`, instead of including a single `'distance'` field in `agent_data`, one should instead include `'distance0'`,

'distance1', 'distance2' and so on, where the index corresponds to the order in which products appear within market in product_data. For example, 'distance5' should measure the distance of agents to the fifth product within the market, as ordered in product_data. The last index should be the number of products in the largest market, minus one. For markets with fewer products than this maximum number, latter columns will be ignored.

Finally, by default each agent i in market t is faced with the same choice set of product j, but it is possible to specify agent-specific availability a_{ijt} much in the same way that product-specific demographics are specified. To do so, the following field can be specified:

– **availability** : (*numeric, optional*) - Agent-specific product availability, a. Choice probabilities in (3.5) are modified according to

$$s_{ijt} = \frac{a_{ijt} \exp V_{ijt}}{1 + \sum_{k \in J_t} a_{ijt} \exp V_{ikt}}, \tag{5.40}$$

and similarly for the nested logit model and consumer surplus calculations. By default, all $a_{ijt} = 1$. To have a product j be unavailable to agent i, set $a_{ijt} = 0$.

Agent-specific availability is specified in the same way that product-specific demographics are specified. In agent_data, one can include 'availability0', 'availability1', 'availability2', and so on, where the index corresponds to the order in which products appear within market in product_data. The last index should be the number of products in the largest market, minus one. For markets with fewer products than this maximum number, latter columns will be ignored.

- **integration** (*Integration, optional*) – Integration configuration for how to build nodes and weights for integration over agent choice probabilities, which will replace any nodes and weights fields in agent_data. This configuration is required if nodes and weights in agent_data are not specified. It should not be specified if X_2 is not formulated in product_formulations.

  If this configuration is specified, K_2 columns of nodes (the number of demand-side nonlinear product characteristics) will be built. However, if sigma is left unspecified or is specified with columns fixed at zero, fewer columns will be used.

- **xi** (*array-like, optional*) – Demand-side unobservable, ξ. This must be specified if X_3 is not formulated or if omega is specified.

  By default, if X_3 is formulated, this and ω_{jt} are drawn from a mean-zero bivariate normal distribution. If phi is specified, then innovations $\tilde{\xi}_{jt}$ and $\tilde{\omega}_{jt}$ in (5.39) are drawn instead, and initial values are draws from their stationary distribution.

- **omega** (*array-like, optional*) – Supply-side unobservable, ω. This must be specified if X_3 is formulated and xi is specified. It is ignored if X_3 is not formulated.

  By default, if X_3 is formulated, this and ξ_{jt} are drawn from a mean-zero bivariate normal distribution. If phi is specified, then innovations are drawn instead, as described for xi.

- **xi_variance** (*float, optional*) – Variance of ξ_{jt} (or its innovation if phi is specified). The default value is 1.0. This is ignored if xi or omega is specified.

- **omega_variance** (*float, optional*) – Variance of ω_{jt} (or its innovation if phi is specified). The default value is 1.0. This is ignored if xi or omega is specified.

- **correlation** (*float, optional*) – Correlation between ξ_{jt} and ω_{jt} (or their innovations if phi is specified). The default value is 0.9. This is ignored if xi or omega is specified.

- **rc_types** (*sequence of str, optional*) – Random coefficient types:

  – 'linear' (default) - The random coefficient is as defined in (3.3).

- `'log'` - The random coefficient's column in (3.3) is exponentiated before being pre-multiplied by X_2. It will take on values bounded from below by zero.
- `'logit'` - The random coefficient's column in (3.3) is passed through the inverse logit function before being pre-multiplied by X_2. It will take on values bounded from below by zero and above by one.

The list should have as many strings as there are columns in X_2. Each string determines the type of the random coefficient on the corresponding product characteristic in X_2.

A typical example of when to use `'log'` is to have a lognormal coefficient on prices. Implementing this typically involves having an `I(-prices)` in the formulation for X_2, and instead of including `prices` in X_1, including a `1` in the `agent_formulation`. Then the corresponding coefficient in Π will serve as the mean parameter for the lognormal random coefficient on negative prices, $-p_{jt}$.

- **epsilon_scale** (*float, optional*) – Factor by which the Type I Extreme Value idiosyncratic preference term, ϵ_{ijt}, is scaled. By default, ϵ_{ijt} is not scaled. The typical use of this parameter is to approximate the pure characteristics model of *Berry and Pakes (2007)* by choosing a value smaller than `1.0`. As this scaling factor approaches zero, the model approaches the pure characteristics model in which there is no idiosyncratic preference term.

  For more information about choosing this parameter and estimating models where it is smaller than `1.0`, refer to the same argument in *Problem.solve()*. In some situations, it may be easier to solve simulations with small epsilon scaling factors by using *Simulation.replace_exogenous()* rather than *Simulation.replace_endogenous()*.

- **costs_type** (*str, optional*) – Specification of the marginal cost function $\tilde{c} = f(c)$ in (3.9). The following specifications are supported:
  - `'linear'` (default) - Linear specification: $\tilde{c} = c$.
  - `'log'` - Log-linear specification: $\tilde{c} = \log c$.
- **seed** (*int, optional*) – Passed to `numpy.random.RandomState` to seed the random number generator before data are simulated. By default, a seed is not passed to the random number generator.

product_formulations
Formulation configurations for X_1, X_2, and X_3, respectively.

Type *tuple*

agent_formulation
Formulation configuration for d.

Type *tuple*

product_data
Synthetic product data that were loaded or simulated during initialization. Typically, *Simulation.replace_endogenous()* is used replace prices and shares with equilibrium values that are consistent with true parameters. The *data_to_dict()* function can be used to convert this into a more usable data type.

Type *recarray*

agent_data
Synthetic agent data that were loaded or simulated during initialization. The *data_to_dict()* function can be used to convert this into a more usable data type.

Type *recarray*

integration
Integration configuration for how any nodes and weights were built during initialization.

Type *Integration*

products
Product data structured as *Products*, which consists of data taken from *Simulation.product_data* along with matrices build according to *Simulation.product_formulations*. The *data_to_dict()* function can be used to convert this into a more usable data type.

Type *Products*

agents
Agent data structured as *Agents*, which consists of data taken from *Simulation.agent_data* or built by *Simulation.integration* along with any demographics formulated by *Simulation.agent_formulation*. The *data_to_dict()* function can be used to convert this into a more usable data type.

Type *Agents*

unique_market_ids
Unique market IDs in product and agent data.

Type *ndarray*

unique_firm_ids
Unique firm IDs in product data.

Type *ndarray*

unique_nesting_ids
Unique nesting IDs in product data.

Type *ndarray*

unique_product_ids
Unique product IDs in product data.

Type *ndarray*

unique_agent_ids
Unique agent IDs in agent data.

Type *ndarray*

beta
Demand-side linear parameters, β.

Type *ndarray*

sigma
Cholesky root of the covariance matrix for unobserved taste heterogeneity, Σ.

Type *ndarray*

gamma
Supply-side linear parameters, γ.

Type *ndarray*

pi
Parameters that measures how agent tastes vary with demographics, Π.

Type *ndarray*

rho
Parameters that measure within nesting group correlation, ρ.

Type *ndarray*

phi
Parameters that measure unobservable autocorrelation, ϕ.

Type *ndarray*

xi
Unobserved demand-side product characteristics, ξ.

Type *ndarray*

omega
Unobserved supply-side product characteristics, ω.

Type *ndarray*

rc_types
Random coefficient types.

Type *list of str*

epsilon_scale
Factor by which the Type I Extreme Value idiosyncratic preference term, ϵ_{ijt}, is scaled.

Type *float*

costs_type
Functional form of the marginal cost function $\tilde{c} = f(c)$.

Type *str*

T
Number of markets, T.

Type *int*

N
Number of products across all markets, N.

Type *int*

F
Number of firms across all markets, F.

Type *int*

I
Number of agents across all markets, I.

Type *int*

K1
Number of demand-side linear product characteristics, K_1.

Type *int*

K2
Number of demand-side nonlinear product characteristics, K_2.

Type *int*

K3
Number of supply-side characteristics, K_3.

Type *int*

D

Number of demographic variables, D.

Type *int*

MD

Number of demand-side instruments, M_D, which is always zero because instruments are added or constructed in `SimulationResults.to_problem()`.

Type *int*

MS

Number of supply-side instruments, M_S, which is similarly aways zero.

Type *int*

MC

Number of covariance instruments, M_C.

Type *int*

ED

Number of absorbed dimensions of demand-side fixed effects, E_D, which is always zero because simulations do not support fixed effect absorption.

Type *int*

ES

Number of absorbed dimensions of supply-side fixed effects, E_S, which is always zero because simulations do not support fixed effect absorption.

Type *int*

H

Number of nesting groups, H.

Type *int*

Examples

- *Tutorial*

Methods

`replace_endogenous`([costs, prices, ...])	Replace simulated prices and market shares with equilibrium values that are consistent with true parameters.
`replace_exogenous`(X1_name[, X3_name, delta, ...])	Replace exogenous product characteristics with values that are consistent with true parameters.

Once initialized, the following method replaces prices and shares with equilibrium values that are consistent with true parameters.

`Simulation.replace_endogenous`([costs, ...])	Replace simulated prices and market shares with equilibrium values that are consistent with true parameters.

5.9.2 pyblp.Simulation.replace_endogenous

Simulation.**replace_endogenous**(*costs=None*, *prices=None*, *iteration=None*, *constant_costs=True*, *compute_gradients=True*, *compute_hessians=True*, *error_behavior='raise'*)

Replace simulated prices and market shares with equilibrium values that are consistent with true parameters.

This method is the standard way of solving the simulation. Prices and market shares are computed in each market by iterating over the ζ-markup contraction in (3.51):

$$p \leftarrow c + \zeta(p). \tag{5.41}$$

Note: To not replace prices, pass the desired prices to `prices` and use an *Iteration* configuration with `method='return'`. This just uses the iteration "routine" that simply returns the the starting values, which are `prices`.

Using this same fake iteration routine and not setting prices will result in a simulation under perfect (instead of Bertrand) competition because the default starting values for the iteration routine are marginal costs.

Note: This method supports *parallel()* processing. If multiprocessing is used, market-by-market computation of prices and shares will be distributed among the processes.

Parameters

- **costs** (*array-like, optional*) – Marginal costs, c. By default, $c = X_3\gamma + \omega$ if `costs_type` was `'linear'` in *Simulation* (the default), and the exponential of this if it was `'log'`. Marginal costs must be specified if X_3 was not formulated in *Simulation*. If marginal costs depend on prices through market shares, they will be updated to reflect different prices during each iteration of the routine.
- **prices** (*array-like, optional*) – Prices at which the fixed point iteration routine will start. By default, `costs`, are used as starting values.
- **iteration** (*Iteration, optional*) – *Iteration* configuration for how to solve the fixed point problem. By default, `Iteration('simple', {'atol': 1e-12})` is used.
- **constant_costs** (*bool, optional*) – Whether to assume that marginal costs, c, remain constant as equilibrium prices and shares change. By default this is `True`, which means that firms treat marginal costs as constant (equal to `costs`) when setting prices. If set to `False`, marginal costs will be allowed to adjust if `shares` was included in the formulation for X_3. When simulating fake data, it likely makes more sense to set this to `False` since otherwise arbitrary `shares` simulated by *Simulation* will be used in marginal costs.
- **compute_gradients** (*bool, optional*) – Whether to compute profit gradients to verify first order conditions. This is by default `True`. Setting it to `False` will slightly speed up computation, but first order conditions will not be reported.
- **compute_hessians** (*bool, optional*) – Whether to compute profit Hessians to verify second order conditions. This is by default `True`. Setting it to `False` will slightly speed up computation, but second order conditions will not be reported.
- **error_behavior** (*str, optional*) – How to handle errors when computing prices and shares. For example, the fixed point routine may not converge if the effects of nonlinear parameters on price overwhelm the linear parameter on price, which should be sufficiently negative. The following behaviors are supported:

- `'raise'` (default) - Raise an exception.
- `'warn'` - Use the last computed prices and shares. If the fixed point routine fails to converge, these are the last prices and shares computed by the routine. If there are other issues, these are the starting prices and their associated shares.

Returns *SimulationResults* of the solved simulation.

Return type *SimulationResults*

Examples

- *Tutorial*

A less common way to solve the simulation is to assume simulated prices and shares represent and equilibrium and to replace exogenous variables instead.

Simulation.replace_exogenous(X1_name[, ...])	Replace exogenous product characteristics with values that are consistent with true parameters.

5.9.3 pyblp.Simulation.replace_exogenous

Simulation.**replace_exogenous**(*X1_name*, *X3_name=None*, *delta=None*, *iteration=None*, *fp_type='safe_linear'*, *shares_bounds=(1e-300, None)*, *error_behavior='raise'*)

Replace exogenous product characteristics with values that are consistent with true parameters.

This method implements a less common way of solving the simulation. It may be preferable to *Simulation.replace_endogenous()* when for some reason it is desirable to retain the prices and market shares from *Simulation*, which are assumed to be in equilibrium. For example, it can be helpful when approximating the pure characteristics model of *Berry and Pakes (2007)* by setting a small `epsilon_scale` value in *Simulation*.

For this method of solving the simulation to be used, there must be an exogenous product characteristic v that shows up only in X_1^{ex}, and if there is a supply side, another product characteristic w that shows up only in X_3^{ex}. These characteristics will be replaced with values that are consistent with true parameters.

First, the mean utility δ is computed in each market by iterating over the contraction in (3.13) and $(\delta - \xi - X_1\beta)\beta_v^{-1}$ is added to the v from *Simulation*. Here, β_v is the linear parameter in β on v.

With a supply side, the marginal cost function $\tilde{c}$ is computed according to (3.7) and (3.9) and $(\tilde{c} - \omega - X_3\gamma)\gamma_w^{-1}$ is added to the w from *Simulation*. Here, γ_w is the linear parameter in γ on w.

Note: This method supports *parallel()* processing. If multiprocessing is used, market-by-market computation of prices and shares will be distributed among the processes.

Parameters

- **X1_name** (*str*) – The name of the variable v in X_1^{ex} that will be replaced. It should show up only once in the formulation for X_1 from *Simulation* and it should not be transformed in any way.
- **X3_name** (*str, optional*) – The name of the variable w in X_3^{ex} that will be replaced. It should show up only once in the formulation for X_3 from *Simulation* and it should not be transformed in any way. This will only be used if there is a supply side.

- **delta** (*array-like, optional*) – Initial values for the mean utility, δ, which the fixed point iteration routine will start at. By default, the solution to the logit model in (3.45) is used. If there is a nesting structure, solution to the nested logit model in (3.46) under the initial `rho` is used instead.
- **iteration** (*Iteration, optional*) – *Iteration* configuration for how to solve the fixed point problem used to compute δ in each market. This configuration is only relevant if there are nonlinear parameters, since δ can be estimated analytically in the logit model. By default, `Iteration('squarem', {'atol': 1e-14})` is used. For more information, refer to the same argument in *Problem.solve()*.
- **fp_type** (*str, optional*) – Configuration for the type of contraction mapping used to compute δ. For information about the different types, refer to the same argument in *Problem.solve()*.
- **shares_bounds** (*tuple, optional*) – Configuration for $s_{jt}(\delta, \theta)$ bounds of the form `(lb, ub)`, in which both `lb` and `ub` are floats or `None`. By default, simulated shares are bounded from below by `1e-300`. This is only relevant if `fp_type` is `'safe_linear'` or `'linear'`. Bounding shares in the contraction does nothing with a nonlinear fixed point. For more information, refer to *Problem.solve()*.
- **error_behavior** (*str, optional*) – How to handle errors when computing δ and $\tilde{c}$. The following behaviors are supported:
  - `'raise'` (default) - Raise an exception.
  - `'warn'` - Use the last computed δ and $\tilde{c}$. If the fixed point routine fails to converge, these are the last δ and the associated $\tilde{c}$ by the routine. If there are other issues, these are the starting δ values and their associated $\tilde{c}$.

Returns *SimulationResults* of the solved simulation.

Return type *SimulationResults*

Examples

- *Tutorial*

5.10 Simulation Results Class

Solved simulations return the following results class.

SimulationResults	Results of a solved simulation of synthetic BLP data.

5.10.1 pyblp.SimulationResults

class pyblp.**SimulationResults**

Results of a solved simulation of synthetic BLP data.

The *SimulationResults.to_problem()* method can be used to convert the full set of simulated data (along with some basic default instruments) and configured information into a *Problem*. Additionally, this class has duplicates of the following *ProblemResults* methods:

- *ProblemResults.compute_aggregate_elasticities()*

- *ProblemResults.compute_elasticities()*
- *ProblemResults.compute_demand_jacobians()*
- *ProblemResults.compute_demand_hessians()*
- *ProblemResults.compute_profit_hessians()*
- *ProblemResults.compute_diversion_ratios()*
- *ProblemResults.compute_long_run_diversion_ratios()*
- *ProblemResults.compute_probabilities()*
- *ProblemResults.extract_diagonals()*
- *ProblemResults.extract_diagonal_means()*
- *ProblemResults.compute_delta()*
- *ProblemResults.compute_costs()*
- *ProblemResults.compute_passthrough()*
- *ProblemResults.compute_approximate_prices()*
- *ProblemResults.compute_prices()*
- *ProblemResults.compute_shares()*
- *ProblemResults.compute_hhi()*
- *ProblemResults.compute_markups()*
- *ProblemResults.compute_profits()*
- *ProblemResults.compute_consumer_surpluses()*
- *ProblemResults.compute_micro_values()*
- *ProblemResults.compute_micro_scores()*
- *ProblemResults.compute_agent_scores()*
- *ProblemResults.simulate_micro_data()*

simulation
Simulation that created these results.

Type *Simulation*

product_data
Simulated *Simulation.product_data* with product characteristics replaced so as to be consistent with the true parameters. If *Simulation.replace_endogenous()* was used to create these results, prices and market shares were replaced. If *Simulation.replace_exogenous()* was used, exogenous characteristics were replaced instead. The *data_to_dict()* function can be used to convert this into a more usable data type.

Type *recarray*

delta
Simulated mean utility, δ.

Type *ndarray*

costs
Simulated marginal costs, c.

Type *ndarray*

computation_time
Number of seconds it took to compute prices and market shares.

Type *float*

fp_converged
Flags for convergence of the iteration routine used to compute prices or δ (depending on the method used to create these results) in each market. Flags are in the same order as `Simulation.unique_market_ids`.

Type *ndarray*

fp_iterations
Number of major iterations completed by the iteration routine used to compute prices or δ in each market. Counts are in the same order as `Simulation.unique_market_ids`.

Type *ndarray*

contraction_evaluations
Number of times the contraction used to compute prices or δ was evaluated in each market. Counts are in the same order as `Simulation.unique_market_ids`.

Type *ndarray*

profit_gradients
Mapping from market IDs t to mappings from firm IDs f to profit gradients. This is only computed if these results were created by `Simulation.replace_endogenous()`. The profit gradient for firm f in market t is a J_{ft} vector with element $k \in J_{ft}$

$$\frac{\partial \pi_{ft}}{\partial p_{kt}} = \sum_{j \in J_{ft}} \frac{\partial \pi_{jt}}{\partial p_{kt}} \tag{5.42}$$

where population-normalized profits are

$$\pi_{jt} = (p_{jt} - c_{jt}) s_{jt}. \tag{5.43}$$

When there is a nontrivial ownership structure, the sum is over all products $j \in J_t$ and the terms are weighted by the firm's (possibly partial) ownership of product j, given by $\mathscr{H}_{jk}$.

Type *dict*

profit_gradient_norms
Mapping from market IDs t to mappings from firm IDs f to the infinity norm of profit gradients. This is only computed if these results were created by `Simulation.replace_endogenous()`. If a norm is near to zero, the firm's choice of profits is near to a local optimum.

Type *dict*

profit_hessians
Mapping from market IDs t to mappings from firm IDs f to profit Hessians. This is only computed if these results were created by `Simulation.replace_endogenous()`. The profit Hessian for firm f in market t is a $J_{ft} \times J_{ft}$ matrix with element $(k, \ell) \in J_{ft}^2$

$$\frac{\partial^2 \pi_{ft}}{\partial p_{kt} \partial p_{\ell t}} = \sum_{j \in J_{ft}} \frac{\partial^2 \pi_{jt}}{\partial p_{kt} \partial p_{\ell t}} \tag{5.44}$$

where population-normalized profits are

$$\pi_{jt} = (p_{jt} - c_{jt}) s_{jt}. \tag{5.45}$$

When there is a nontrivial ownership structure, the sum is over all products $j \in J_t$ and the terms are weighted by the firm's (possibly partial) ownership of product j, given by $\mathscr{H}_{jk}$.

Type *dict*

profit_hessian_eigenvalues
Mapping from market IDs t to mappings from firm IDs f to the eigenvalues of profit Hessians. This is only computed if these results were created by `Simulation.replace_endogenous()`. If the fixed point converged and all eigenvalues are negative, the firm's choice of profits is a local maximum.

Type *dict*

Examples

- *Tutorial*

This class has many of the same methods as `ProblemResults`. It can also be pickled or converted into a dictionary.

`SimulationResults.to_pickle`(path)	Save these results as a pickle file.
`SimulationResults.to_dict`([attributes])	Convert these results into a dictionary that maps attribute names to values.

5.10.2 pyblp.SimulationResults.to_pickle

SimulationResults.**to_pickle**(*path*)
Save these results as a pickle file.

Parameters **path** (*str or Path*) – File path to which these results will be saved.

5.10.3 pyblp.SimulationResults.to_dict

SimulationResults.**to_dict**(*attributes=('product_data', 'delta', 'costs', 'computation_time', 'fp_converged', 'fp_iterations', 'contraction_evaluations', 'profit_gradients', 'profit_gradient_norms', 'profit_hessians', 'profit_hessian_eigenvalues')*)
Convert these results into a dictionary that maps attribute names to values.

Parameters **attributes** (*sequence of str, optional*) – Name of attributes that will be added to the dictionary. By default, all `SimulationResults` attributes are added except for `SimulationResults.simulation`.

Returns Mapping from attribute names to values.

Return type *dict*

Examples

- *Tutorial*

It can also be converted into a `Problem` with the following method.

`SimulationResults.to_problem`([...])	Convert the solved simulation into a problem.

5.10.4 pyblp.SimulationResults.to_problem

SimulationResults.**to_problem**(*product_formulations=None*, *product_data=None*, *agent_formulation=None*, *agent_data=None*, *integration=None*, *rc_types=None*, *epsilon_scale=None*, *costs_type=None*, *add_exogenous=True*)

Convert the solved simulation into a problem.

Arguments are the same as those of *Problem*. By default, the structure of the problem will be the same as that of the solved simulation.

By default, some simple "sums of characteristics" BLP instruments are constructed. Demand-side instruments are constructed by *build_blp_instruments()* from variables in X_1^{extex}, along with any supply shifters (variables in X_3^{extex} but not X_1^{extex}). Supply side instruments are constructed from variables in X_3^{extex}, along with any demand shifters (variables in X_1^{extex} but not X_3^{extex}). Instruments will also be constructed from columns of ones if there is variation in J_t, the number of products per market. Any constant columns will be dropped. For example, if each firm owns exactly one product in each market, the "rival" columns of instruments will be zero and hence dropped.

Note: These excluded instruments are constructed only for convenience. Especially for more complicated problems, they should be replaced with better instruments.

Parameters

- **product_formulations** (*Formulation or sequence of Formulation, optional*) – By default, *Simulation.product_formulations*.
- **product_data** (*structured array-like, optional*) – By default, *SimulationResults.product_data* with excluded instruments.
- **agent_formulation** (*Formulation, optional*) – By default, *Simulation.agent_formulation*.
- **agent_data** (*structured array-like, optional*) – By default, *Simulation.agent_data*.
- **integration** (*Integration, optional*) – By default, this is unspecified.
- **rc_types** (*sequence of str, optional*) – By default, *Simulation.rc_types*.
- **epsilon_scale** (*float, optional*) – By default, *Simulation.epsilon_scale*.
- **costs_type** (*str, optional*) – By default, *Simulation.costs_type*.
- **add_exogenous** (*bool, optional*) – By default, `True`.

Returns A BLP problem.

Return type *Problem*

Examples

- *Tutorial*

5.11 Structured Data Classes

Product and agent data that are passed or constructed by *Problem* and *Simulation* are structured internally into classes with field names that more closely resemble BLP notation. Although these structured data classes are not directly constructable, they can be accessed with *Problem* and *Simulation* class attributes. It can be helpful to compare these structured data classes with the data or configurations used to create them.

Products	Product data structured as a record array.
Agents	Agent data structured as a record array.

5.11.1 pyblp.Products

class pyblp.**Products**

Product data structured as a record array.

Attributes in addition to the ones below are the variables underlying X_1, X_2, and X_3.

market_ids

IDs that associate products with markets.

Type *ndarray*

firm_ids

IDs that associate products with firms.

Type *ndarray*

demand_ids

IDs used to create demand-side fixed effects.

Type *ndarray*

supply_ids

IDs used to create supply-side fixed effects.

Type *ndarray*

nesting_ids

IDs that associate products with nesting groups.

Type *ndarray*

product_ids

IDs that identify products within markets.

Type *ndarray*

clustering_ids

IDs used to compute clustered standard errors.

Type *ndarray*

lag_indices

Indices of products that correspond to their lags or the current row index to indicate an initial period.

Type *ndarray*

ownership

Stacked $J_t \times J_t$ ownership or product holding matrices, $\mathscr{H}$, for each market t.

Type *ndarray*

shares
Market shares, s.

Type *ndarray*

prices
Product prices, p.

Type *ndarray*

ZD
Full set of demand-side instruments, Z_D, which typically consists of excluded demand-side instruments and X_1^{ex}. If there are any demand-side fixed effects, these instruments will be residualized with respect to these fixed effects.

Type *ndarray*

ZS
Full set of supply-side instruments, Z_S, which typically consists of excluded supply-side instruments and X_3^{ex}. If there are any supply-side fixed effects, these instruments will be residualized with respect to these fixed effects.

Type *ndarray*

ZC
Covariance instruments, Z_C, as in *MacKay and Miller (2025)*.

Type *ndarray*

X1
Demand-side linear product characteristics, X_1. If there are any demand-side fixed effects, these characteristics will be residualized with respect to these fixed effects.

Type *ndarray*

X2
Demand-side nonlinear product characteristics, X_2.

Type *ndarray*

X3
Supply-side product characteristics, X_3. If there are any supply-side fixed effects, these characteristics will be residualized with respect to these fixed effects.

Type *ndarray*

5.11.2 pyblp.Agents

class pyblp.**Agents**
Agent data structured as a record array.

market_ids
IDs that associate agents with markets.

Type *ndarray*

agent_ids
IDs that identify agents within markets.

Type *ndarray*

weights
Integration weights, w.

Type *ndarray*

nodes
Unobserved agent characteristics called integration nodes, ν.

Type *ndarray*

demographics
Observed agent characteristics, d.

Type *ndarray*

availability
Agent-specific product availability, a.

Type *ndarray*

5.12 Multiprocessing

A context manager can be used to enable parallel processing for methods that perform market-by-market computation.

parallel(processes[, use_pathos])	Context manager used for parallel processing in a `with` statement context.

5.12.1 pyblp.parallel

pyblp.**parallel**(*processes*, *use_pathos=False*)
Context manager used for parallel processing in a `with` statement context.

This manager creates a context in which a pool of Python processes will be used by any method that requires market-by-market computation. These methods will distribute their work among the processes. After the context created by the `with` statement ends, all worker processes in the pool will be terminated. Outside this context, such methods will not use multiprocessing.

Importantly, multiprocessing will only improve speed if gains from parallelization outweigh overhead from serializing and passing data between processes. For example, if computation for a single market is very fast and there is a lot of data in each market that must be serialized and passed between processes, using multiprocessing may reduce overall speed.

Parameters

- **processes** (*int*) – Number of Python processes that will be created and used by any method that supports parallel processing.
- **use_pathos** (*bool, optional*) – Whether to use pathos (which will need to be installed) instead of the default, built-in `multiprocessing` module. Since pathos uses dill to pickle and pass objects between processes, it can support more objects than the default multiprocessing module, which uses the default pickle module. However, dill can be much slower, so using pathos can further increase overhead of passing data between processes.

Examples

The online version of the following section may be easier to read.

Parallel Processing Example

```
import pyblp
import pandas as pd

pyblp.options.digits = 2
pyblp.options.verbose = False
pyblp.__version__
```

```
'1.2.0'
```

In this example, we'll use parallel processing to compute elasticities market-by-market for a simple Logit problem configured with some of the fake cereal data from *Nevo (2000a)*.

```
product_data = pd.read_csv(pyblp.data.NEVO_PRODUCTS_LOCATION)
formulation = pyblp.Formulation('0 + prices', absorb='C(product_ids)')
problem = pyblp.Problem(formulation, product_data)
results = problem.solve()
results
```

```
Problem Results Summary:
========================================
GMM   Objective  Clipped  Weighting Matrix
Step    Value    Shares   Condition Number
----  ---------  -------  ----------------
 2    +1.9E+02      0         +5.7E+07
========================================

Cumulative Statistics:
========================
Computation   Objective
   Time      Evaluations
-----------  -----------
 00:00:00         2
========================

Beta Estimates (Robust SEs in Parentheses):
```

(continues on next page)

(continued from previous page)

```
==========
  prices
----------
 -3.0E+01
(+1.0E+00)
==========
```

```
pyblp.options.verbose = True
with pyblp.parallel(2):
    elasticities = results.compute_elasticities()
```

```
Starting a pool of 2 processes ...
Started the process pool after 00:00:00.
Computing elasticities with respect to prices ...
Finished after 00:00:00.

Terminating the pool of 2 processes ...
Terminated the process pool after 00:00:00.
```

Solving a Logit problem does not require market-by-market computation, so parallelization does not change its estimation procedure. Although elasticity computation does happen market-by-market, this problem is very small, so in this small example there are no gains from parallelization.

If the problem were much larger, running *Problem.solve* and *ProblemResults.compute_elasticities* under the `with` statement could substantially speed up estimation and elasticity computation.

5.13 Options and Example Data

In addition to classes and functions, there are also two modules that can be used to configure global package options and locate example data that comes with the package.

options	Global options.
data	Locations of example data that are included in the package for convenience.

5.13.1 pyblp.options

Global options.

pyblp.options.**digits**
: Number of digits displayed by status updates. The default number of digits is `7`. The number of digits can be changed to, for example, `2`, with `pyblp.options.digits = 2`.

  Type *int*

pyblp.options.**verbose**
: Whether to output status updates. By default, verbosity is turned on. Verbosity can be turned off with `pyblp.options.verbose = False`.

  Type *bool*

pyblp.options.**verbose_tracebacks**
: Whether to include full tracebacks in error messages. By default, full tracebacks are turned off. These can be useful when attempting to find the source of an error message. Tracebacks can be turned on with `pyblp.options.verbose_tracebacks = True`.

  Type *bool*

pyblp.options.**verbose_output**
: Function used to output status updates. The default function is simply `print`. The function can be changed, for example, to include an indicator that statuses are from this package, with `pyblp.verbose_output = lambda x: print(f"pyblp: {x}")`.

  Type *callable*

pyblp.options.**flush_output**
: Whether to call `sys.stdout.flush()` after outputting a status update. By default, output is not flushed to standard output. To force standard output flushes after every status update, set `pyblp.options.flush_output = True`. This may be particularly desirable for R users who are calling PyBLP from reticulate, since standard output is typically not automatically flushed to the screen in this environment. If PyBLP is imported as `pyblp`, this setting can be enabled in R with `pyblp$options$flush_output <- TRUE`.

  Type *bool*

pyblp.options.**dtype**
: The data type used for internal calculations, which is by default `numpy.float64`. The other recommended option is `numpy.longdouble`, which is the only extended precision floating point type currently supported by NumPy. Although this data type will be used internally, `numpy.float64` will be used when passing arrays to optimization and fixed point routines, which may not support extended precision. The library underlying `scipy.linalg`, which is used for matrix inversion, may also use `numpy.float64`.

  One instance in which extended precision can be helpful in the BLP problem is when there are a large number of near zero choice probabilities with small integration weights, which, under standard precision are called zeros when in aggregate they are nonzero.

The precision of `numpy.longdouble` depends on the platform on which NumPy is installed. If the platform in use does not support extended precision, using `numpy.longdouble` may lead to unreliable results. For example, on Windows, NumPy is usually compiled such that `numpy.longdouble` often behaves like `numpy.float64`. Precisions can be compared with `numpy.finfo` by running `numpy.finfo(numpy.float64)` and `numpy.finfo(numpy.longdouble)`. For more information, refer to this discussion.

If extended precisions is supported, the data type can be switched with `pyblp.options.dtype = numpy.longdouble`. On Windows, it is often easier to install Linux in a virtual machine than it is to build NumPy from source with a non-standard compiler.

Type *dtype*

pyblp.options.**finite_differences_epsilon**

Perturbation ϵ used to numerically approximate derivatives with central finite differences:

$$f'(x) = \frac{f(x + \epsilon/2) - f(x - \epsilon/2)}{\epsilon}. \tag{5.46}$$

By default, this is the square root of the machine epsilon: `numpy.sqrt(numpy.finfo(options.dtype).eps)`. The typical example where this is used is when computing the Hessian, but it may also be used to compute Jacobians required for standard errors when analytic gradients are disabled.

Type *float*

pyblp.options.**pseudo_inverses**

Whether to compute Moore-Penrose pseudo-inverses of matrices with `scipy.linalg.pinv()` instead of their classic inverses with `scipy.linalg.inv()`. This is by default `True`, so pseudo-inverses will be used. Up to small numerical differences, the pseudo-inverse is identical to the classic inverse for invertible matrices. Using the pseudo-inverse by default can help alleviate problems from, for example, near-singular weighting matrices.

To always attempt to compute classic inverses first, set `pyblp.options.pseudo_inverses = False`. If a classic inverse cannot be computed, an error will be displayed, and a pseudo-inverse may be computed instead.

Type *bool*

pyblp.options.**weights_tol**

Tolerance for detecting integration weights that do not sum to one, which is by default `1e-10`. In most setups weights should essentially sum to one, but for example with importance sampling they may be slightly different. Warnings can be disabled by setting this to `numpy.inf`.

Type *float*

pyblp.options.**singular_tol**

Tolerance for detecting singular matrices, which is by default `1 / numpy.finfo(options.dtype).eps`. If a matrix has a condition number larger than this tolerance, a warning will be displayed. To disable singularity checks, set `pyblp.options.singular_tol = numpy.inf`.

Type *float*

pyblp.options.**collinear_atol**

Absolute tolerance for detecting collinear columns in each matrix of product characteristics and instruments: X_1, X_2, X_3, Z_D, and Z_S. Micro moments can also be checked; see `detect_micro_collinearity`.

Each matrix is decomposed into a QR decomposition and an error is raised for any column whose diagonal element in R has a magnitude less than `collinear_atol + collinear_rtol * sd` where `sd` is the column's standard deviation.

The default absolute tolerance is `1e-10`. To disable collinearity checks, set `pyblp.options.collinear_atol = pyblp.options.collinear_rtol = 0`.

Type *float*

pyblp.options.**collinear_rtol**
Relative tolerance for detecting collinear columns, which is by default also `1e-10`.

Type *float*

pyblp.options.**psd_atol**
Absolute tolerance for detecting non-positive semidefinite matrices. For example, this check is applied to any custom weighting matrix, W.

Singular value decomposition factorizes the matrix into $U\Sigma V$ and an error is raised if any element in the original matrix differs in absolute value from $V'\Sigma V$ by more than `psd_atol + psd_rtol * abs` where `abs` is the element's absolute value.

The default tolerance is `1e-8`. To disable positive semidefinite checks, set `pyblp.options.psd_atol = pyblp.options.psd_rtol = numpy.inf`.

Type *float*

pyblp.options.**psd_rtol**
Relative tolerance for detecting non-positive definite matrices, which is by default also `1e-8`.

Type *float*

pyblp.options.**stable_atol**
Absolute tolerance for detecting an unstable autoregressive coefficient matrix ϕ.

Eigenvalue decomposition computes the roots of ϕ, and an error is raised if the spectral radius exceeds one by more than `stable_atol + stable_rtol * abs` where `abs` is the absolute value of the spectral radius.

The default tolerance is `1e-8`. To disable stability checks, set `pyblp.options.stable_atol = pyblp.options.stable_rtol = numpy.inf`.

Type *float*

pyblp.options.**stable_rtol**
Relative tolerance for detecting an unstable autoregressive coefficient matrix, which is by default also `1e-8`.

Type *float*

pyblp.options.**detect_micro_collinearity**
Whether to check if micro values v_{pijt} (or v_{pijkt} with second choices) are collinear with one another by computing these values once, stacking them, and using `pyblp.options.collinear_atol` and `pyblp.options.collinear_rtol`.

By default, micro values are not checked for collinearity because this procedure can require a large amount of memory. To enable this check, set `pyblp.options.detect_micro_collinearity = True`. If this uses a large amount of memory, one option is to temporarily reduce the number of markets, observations, or agents to cut down on memory while debugging one's code to see which micro moments are collinear with one another.

Type *bool*

pyblp.options.**micro_computation_chunks**
How finely to break up micro moment computation within market. Computation is broken up by groups of agents within market. This can help reduce the amount of memory being used by micro moments when there are a large number of agents and products, and especially when second choice micro moments are being used.

By default, micro moment computation is done in one chunk for each market. To reduce memory usage without changing any estimation results, for example by splitting up computation into 10 chunks, use `pyblp.options.micro_computation_chunks = 10`.

If a dictionary, this should map market IDs to the number of chunks to use. For example, to only chunk computation in market ID `'big market'`, use `pyblp.options.micro_computation_chunks = {'big_market': 10}`.

Type *int or dict*

`pyblp.options.`**`drop_product_fields`**
Whether to conserve memory by dropping product data fields that are not needed for market-level computation when initializing a market. By default, these fields are not dropped. Setting `pyblp.options.drop_product_fields = True` may reduce memory usage, especially if there are many instruments, at the cost of extra time needed to drop these fields.

Type *bool*

5.13.2 pyblp.data

Locations of example data that are included in the package for convenience.

`pyblp.data.`**`NEVO_PRODUCTS_LOCATION`**
Location of a CSV file containing the fake cereal product data from *Nevo (2000a)*. The file includes the same pre-computed excluded instruments used in the original paper. The data are from Aviv Nevo's Matlab code, which was archived on Eric Rasmusen's website.

Type *str*

`pyblp.data.`**`NEVO_AGENTS_LOCATION`**
Location of a CSV file containing the agent data from *Nevo (2000a)*. Included in the file are Monte Carlo weights and draws along with demographics from the original paper. The data are from Aviv Nevo's Matlab code, which was archived on Eric Rasmusen's website.

Type *str*

`pyblp.data.`**`BLP_PRODUCTS_LOCATION`**
Location of a CSV file containing the automobile product data extracted by *Andrews, Gentzkow, and Shapiro (2017)* from the original GAUSS code for *Berry, Levinsohn, and Pakes (1999)*, which is commonly assumed to be the same data used in *Berry, Levinsohn, and Pakes (1995)*.

The file also includes a set of excluded instruments. First, "sums of characteristics" BLP instruments from the original paper were computed with *`build_blp_instruments()`*. The examples section in the documentation for this function shows how to construct these instruments from scratch. As in the original paper, the "rival" instrument constructed from the `trend` variable was excluded due to collinearity issues, and the `mpd` variable was added to the set of excluded instruments for supply.

Type *str*

`pyblp.data.`**`BLP_AGENTS_LOCATION`**
Location of a CSV file containing the agent data from *Berry, Levinsohn, and Pakes (1999)*. Included in the file are the importance sampling weights and draws along with the income demographic from the original paper. These data are also from the replication code of *Andrews, Gentzkow, and Shapiro (2017)*.

Type *str*

`pyblp.data.`**`PETRIN_PRODUCTS_LOCATION`**
Location of a CSV file containing the automobile product data from *Petrin (2002)*. The file includes the same pre-computed excluded instruments used in the original paper. The data are from Amil Petrin's GAUSS code, available on his website.

Type *str*

pyblp.data.**PETRIN_AGENTS_LOCATION**
Location of a CSV file containing agent data similar to that used by *Petrin (2002)*. The file includes 1,000 scrambled Halton draws in each market, along with demographics resampled from the Consumer Expenditure Survey (CEX) used by the original paper. The original paper used pseudo Monte Carlo draws and importance sampling. The demographics that were resampled are from Amil Petrin's GAUSS code, available on his website.

Type *str*

pyblp.data.**PETRIN_VALUES_LOCATION**
Location of a CSV file containing micro moment values matched by *Petrin (2002)*. These are the rounded values reported in Table 6a of the working paper version of the original paper.

Type *str*

pyblp.data.**PETRIN_COVARIANCES_LOCATION**
Location of a CSV file containing micro moment sample covariances used by *Petrin (2002)*. The data are from Amil Petrin's GAUSS code, available on his website.

Type *str*

Examples

The online version of the following section may be easier to read.

Loading Data Example

```
import pyblp

pyblp.__version__
```

```
'1.2.0'
```

Any number of functions can be used to load the example data into memory. In this example, we'll first use NumPy.

```
import numpy as np
blp_product_data = np.genfromtxt(pyblp.data.BLP_PRODUCTS_LOCATION, delimiter=',', names=True, encoding='utf-8')
blp_agent_data = np.genfromtxt(pyblp.data.BLP_AGENTS_LOCATION, delimiter=',', names=True, encoding='utf-8')
```

Record arrays can be cumbersome to manipulate. A more flexible alternative is the pandas DataFrame. Unlike NumPy, pyblp does not directly depend on pandas, but it can be useful when manipulating data.

```
import pandas as pd
blp_product_data = pd.read_csv(pyblp.data.BLP_PRODUCTS_LOCATION)
blp_agent_data = pd.read_csv(pyblp.data.BLP_AGENTS_LOCATION)
```

Another benefit of DataFrame objects is that they display nicely in Jupyter notebooks.

```
blp_product_data.head()
```

```
   market_ids clustering_ids  car_ids  firm_ids region    shares    prices  \
0        1971       AMGREM71      129        15     US  0.001051  4.935802
1        1971       AMHORN71      130        15     US  0.000670  5.516049
2        1971       AMJAVL71      132        15     US  0.000341  7.108642
3        1971       AMMATA71      134        15     US  0.000522  6.839506
4        1971       AMAMBS71      136        15     US  0.000442  8.928395

       hpwt  air       mpd  ...  supply_instruments2  supply_instruments3  \
0  0.528997    0  1.888146  ...                  0.0             1.705933
1  0.494324    0  1.935989  ...                  0.0             1.680910
2  0.467613    0  1.716799  ...                  0.0             1.801067
3  0.426540    0  1.687871  ...                  0.0             1.818061
```

(continues on next page)

(continued from previous page)

```
4  0.452489    0  1.504286  ...                  0.0             1.933210

   supply_instruments4  supply_instruments5  supply_instruments6  \
0             1.595656                 87.0           -61.959985
1             1.490295                 87.0           -61.959985
2             1.357703                 87.0           -61.959985
3             1.261347                 87.0           -61.959985
4             1.237365                 87.0           -61.959985

   supply_instruments7  supply_instruments8  supply_instruments9  \
0                  0.0            46.060389            29.786989
1                  0.0            46.060389            29.786989
2                  0.0            46.060389            29.786989
3                  0.0            46.060389            29.786989
4                  0.0            46.060389            29.786989

   supply_instruments10  supply_instruments11
0                   0.0              1.888146
1                   0.0              1.935989
2                   0.0              1.716799
3                   0.0              1.687871
4                   0.0              1.504286

[5 rows x 33 columns]
```

```
blp_agent_data.head()
```

```
   market_ids   weights    nodes0    nodes1    nodes2    nodes3    nodes4  \
0        1971  0.000543  1.192188  0.478777  0.980830 -0.824410  2.473301
1        1971  0.000723  1.497074 -2.026204 -1.741316  1.412568 -0.747468
2        1971  0.000544  1.438081  0.813280 -1.749974 -1.203509  0.049558
3        1971  0.000701  1.768655 -0.177453  0.286602  0.391517  0.683669
4        1971  0.000549  0.849970 -0.135337  0.735920  1.036247 -1.143436

       income
0  109.560369
1   45.457314
2  127.146548
3   22.604045
4  170.226032
```

This tutorial demonstrates how the instruments included in this dataset can be constructed from scratch.

5.14 Exceptions

When errors occur, they will either be displayed as warnings or raised as exceptions.

exceptions.MultipleErrors	Multiple errors that occurred around the same time.
exceptions.NonpositiveCostsError	Encountered nonpositive marginal costs in a log-linear specification.
exceptions.NonpositiveSyntheticCostsError	Encountered nonpositive synthetic marginal costs in a log-linear specification.
exceptions.InvalidParameterCovariancesError	Failed to compute standard errors because of invalid estimated covariances of GMM parameters.
exceptions.InvalidMomentCovariancesError	Failed to compute a weighting matrix because of invalid estimated covariances of GMM moments.
exceptions.GenericNumericalError	Encountered a numerical error.
exceptions.DeltaNumericalError	Encountered a numerical error when computing δ.
exceptions.CostsNumericalError	Encountered a numerical error when computing marginal costs.
exceptions.MicroMomentsNumericalError	Encountered a numerical error when computing micro moments.
exceptions.XiByThetaJacobianNumericalError	Encountered a numerical error when computing the Jacobian (holding β fixed) of ξ (equivalently, of δ) with respect to θ.
exceptions.OmegaByThetaJacobianNumericalError	Encountered a numerical error when computing the Jacobian (holding γ fixed) of ω (equivalently, of transformed marginal costs) with respect to θ.
exceptions.MicroMomentsByThetaJacobianNumericalError	Encountered a numerical error when computing the Jacobian of micro moments with respect to θ.
exceptions.MicroMomentCovariancesNumericalError	Encountered a numerical error when computing micro moment covariances.
exceptions.SyntheticPricesNumericalError	Encountered a numerical error when computing synthetic prices.
exceptions.SyntheticSharesNumericalError	Encountered a numerical error when computing synthetic shares.
exceptions.SyntheticDeltaNumericalError	Encountered a numerical error when computing the synthetic δ.
exceptions.SyntheticCostsNumericalError	Encountered a numerical error when computing synthetic marginal costs.
exceptions.SyntheticMicroDataNumericalError	Encountered a numerical error when computing synthetic micro data.
exceptions.SyntheticMicroMomentsNumericalError	Encountered a numerical error when computing synthetic micro moments.
exceptions.MicroScoresNumericalError	Encountered a numerical error when computing micro scores.
exceptions.EquilibriumRealizationNumericalError	Encountered a numerical error when solving for a realization of equilibrium prices and shares.
exceptions.JacobianRealizationNumericalError	Encountered a numerical error when computing a realization of the Jacobian (holding β fixed) of ξ (equivalently, of δ) or ω (equivalently, of transformed marginal costs) with respect to θ.
exceptions.PostEstimationNumericalError	Encountered a numerical error when computing a post-estimation output.

Continued on next page

Table 43 – continued from previous page

exceptions.AbsorptionError	A fixed effect absorption procedure failed to properly absorb fixed effects.
exceptions.ClippedSharesError	Shares were clipped during the final iteration of the fixed point routine for computing δ.
exceptions.ThetaConvergenceError	The optimization routine failed to converge.
exceptions.DeltaConvergenceError	The fixed point computation of δ failed to converge.
exceptions.SyntheticPricesConvergenceError	The fixed point computation of synthetic prices failed to converge.
exceptions.SyntheticDeltaConvergenceError	The fixed point computation of the synthetic δ failed to converge.
exceptions.EquilibriumPricesConvergenceError	The fixed point computation of equilibrium prices failed to converge.
exceptions.ObjectiveReversionError	Reverted a problematic GMM objective value.
exceptions.GradientReversionError	Reverted problematic elements in the GMM objective gradient.
exceptions.DeltaReversionError	Reverted problematic elements in δ.
exceptions.CostsReversionError	Reverted problematic marginal costs.
exceptions.MicroMomentsReversionError	Reverted problematic micro moments.
exceptions.XiByThetaJacobianReversionError	Reverted problematic elements in the Jacobian (holding β fixed) of ξ (equivalently, of δ) with respect to θ.
exceptions.OmegaByThetaJacobianReversionError	Reverted problematic elements in the Jacobian (holding γ fixed) of ω (equivalently, of transformed marginal costs) with respect to θ.
exceptions.MicroMomentsByThetaJacobianReversionError	Reverted problematic elements in the Jacobian of micro moments with respect to θ.
exceptions.HessianEigenvaluesError	Failed to compute eigenvalues for the GMM objective's (reduced) Hessian matrix.
exceptions.ProfitHessianEigenvaluesError	Failed to compute eigenvalues for a firm's profit Hessian.
exceptions.FittedValuesInversionError	Failed to invert an estimated covariance when computing fitted values.
exceptions.SharesByXiJacobianInversionError	Failed to invert a Jacobian of shares with respect to ξ when computing the Jacobian (holding β fixed) of ξ (equivalently, of δ) with respect to θ.
exceptions.IntraFirmJacobianInversionError	Failed to invert an intra-firm Jacobian of shares with respect to prices.
exceptions.PassthroughInversionError	Failed to invert the matrix to recover the passthrough matrix.
exceptions.LinearParameterCovariancesInversionError	Failed to invert an estimated covariance matrix of linear parameters.
exceptions.GMMParameterCovariancesInversionError	Failed to invert an estimated covariance matrix of GMM parameters.
exceptions.GMMMomentCovariancesInversionError	Failed to invert an estimated covariance matrix of GMM moments.

5.14.1 pyblp.exceptions.MultipleErrors

class pyblp.exceptions.**MultipleErrors**

Multiple errors that occurred around the same time.

5.14.2 pyblp.exceptions.NonpositiveCostsError

class pyblp.exceptions.**NonpositiveCostsError**
Encountered nonpositive marginal costs in a log-linear specification.

This problem can sometimes be mitigated by bounding costs from below, choosing more reasonable initial parameter values, setting more conservative parameter bounds, or using a linear costs specification.

5.14.3 pyblp.exceptions.NonpositiveSyntheticCostsError

class pyblp.exceptions.**NonpositiveSyntheticCostsError**
Encountered nonpositive synthetic marginal costs in a log-linear specification.

This problem can sometimes be mitigated by more reasonable initial parameter values or using a linear costs specification.

5.14.4 pyblp.exceptions.InvalidParameterCovariancesError

class pyblp.exceptions.**InvalidParameterCovariancesError**
Failed to compute standard errors because of invalid estimated covariances of GMM parameters.

5.14.5 pyblp.exceptions.InvalidMomentCovariancesError

class pyblp.exceptions.**InvalidMomentCovariancesError**
Failed to compute a weighting matrix because of invalid estimated covariances of GMM moments.

5.14.6 pyblp.exceptions.GenericNumericalError

class pyblp.exceptions.**GenericNumericalError**
Encountered a numerical error.

5.14.7 pyblp.exceptions.DeltaNumericalError

class pyblp.exceptions.**DeltaNumericalError**
Encountered a numerical error when computing δ.

This problem is often due to prior problems, overflow, or nonpositive shares, and can sometimes be mitigated by choosing smaller initial parameter values, setting more conservative bounds on parameters or shares, rescaling data, removing outliers, changing the floating point precision, or using different optimization, iteration, or integration configurations.

5.14.8 pyblp.exceptions.CostsNumericalError

class pyblp.exceptions.**CostsNumericalError**
Encountered a numerical error when computing marginal costs.

This problem is often due to prior problems or overflow and can sometimes be mitigated by choosing smaller initial parameter values, setting more conservative bounds, rescaling data, removing outliers, changing the floating point precision, or using different optimization or cost configurations.

5.14.9 pyblp.exceptions.MicroMomentsNumericalError

class pyblp.exceptions.**MicroMomentsNumericalError**
Encountered a numerical error when computing micro moments.

This problem is often due to prior problems, overflow, or nonpositive shares, and can sometimes be mitigated by choosing smaller initial parameter values, setting more conservative bounds, rescaling data, removing outliers, changing the floating point precision, or using different optimization, iteration, or integration configurations.

5.14.10 pyblp.exceptions.XiByThetaJacobianNumericalError

class pyblp.exceptions.**XiByThetaJacobianNumericalError**
Encountered a numerical error when computing the Jacobian (holding β fixed) of ξ (equivalently, of δ) with respect to θ.

This problem is often due to prior problems, overflow, or nonpositive shares, and can sometimes be mitigated by choosing smaller initial parameter values, setting more conservative bounds, rescaling data, removing outliers, changing the floating point precision, or using different optimization, iteration, or integration configurations.

5.14.11 pyblp.exceptions.OmegaByThetaJacobianNumericalError

class pyblp.exceptions.**OmegaByThetaJacobianNumericalError**
Encountered a numerical error when computing the Jacobian (holding γ fixed) of ω (equivalently, of transformed marginal costs) with respect to θ.

This problem is often due to prior problems or overflow, and can sometimes be mitigated by choosing smaller initial parameter values, setting more conservative bounds, rescaling data, removing outliers, changing the floating point precision, or using different optimization or cost configurations.

5.14.12 pyblp.exceptions.MicroMomentsByThetaJacobianNumericalError

class pyblp.exceptions.**MicroMomentsByThetaJacobianNumericalError**
Encountered a numerical error when computing the Jacobian of micro moments with respect to θ.

5.14.13 pyblp.exceptions.MicroMomentCovariancesNumericalError

class pyblp.exceptions.**MicroMomentCovariancesNumericalError**
Encountered a numerical error when computing micro moment covariances.

5.14.14 pyblp.exceptions.SyntheticPricesNumericalError

class pyblp.exceptions.**SyntheticPricesNumericalError**
Encountered a numerical error when computing synthetic prices.

This problem is often due to prior problems or overflow and can sometimes be mitigated by making sure that the specified parameters are reasonable. For example, the parameters on prices should generally imply a downward sloping demand curve.

5.14.15 pyblp.exceptions.SyntheticSharesNumericalError

class pyblp.exceptions.**SyntheticSharesNumericalError**
Encountered a numerical error when computing synthetic shares.

This problem is often due to prior problems or overflow and can sometimes be mitigated by making sure that the specified parameters are reasonable. For example, the parameters on prices should generally imply a downward sloping demand curve.

5.14.16 pyblp.exceptions.SyntheticDeltaNumericalError

class pyblp.exceptions.**SyntheticDeltaNumericalError**
Encountered a numerical error when computing the synthetic δ.

This problem is often due to prior problems, overflow, or nonpositive shares, and can sometimes be mitigated by making sure that the specified parameters are reasonable.

5.14.17 pyblp.exceptions.SyntheticCostsNumericalError

class pyblp.exceptions.**SyntheticCostsNumericalError**
Encountered a numerical error when computing synthetic marginal costs.

This problem is often due to prior problems or overflow and can sometimes be mitigated by making sure that the specified parameters are reasonable.

5.14.18 pyblp.exceptions.SyntheticMicroDataNumericalError

class pyblp.exceptions.**SyntheticMicroDataNumericalError**
Encountered a numerical error when computing synthetic micro data.

5.14.19 pyblp.exceptions.SyntheticMicroMomentsNumericalError

class pyblp.exceptions.**SyntheticMicroMomentsNumericalError**
Encountered a numerical error when computing synthetic micro moments.

5.14.20 pyblp.exceptions.MicroScoresNumericalError

class pyblp.exceptions.**MicroScoresNumericalError**
Encountered a numerical error when computing micro scores.

5.14.21 pyblp.exceptions.EquilibriumRealizationNumericalError

class pyblp.exceptions.**EquilibriumRealizationNumericalError**
Encountered a numerical error when solving for a realization of equilibrium prices and shares.

5.14.22 pyblp.exceptions.JacobianRealizationNumericalError

class pyblp.exceptions.**JacobianRealizationNumericalError**
Encountered a numerical error when computing a realization of the Jacobian (holding β fixed) of ξ (equivalently, of δ) or ω (equivalently, of transformed marginal costs) with respect to θ.

5.14.23 pyblp.exceptions.PostEstimationNumericalError

class pyblp.exceptions.**PostEstimationNumericalError**
Encountered a numerical error when computing a post-estimation output.

5.14.24 pyblp.exceptions.AbsorptionError

class pyblp.exceptions.**AbsorptionError**
A fixed effect absorption procedure failed to properly absorb fixed effects.

Consider configuring absorption options or choosing a different absorption method. For information about absorption options and defaults, refer to the PyHDFE package's documentation.

5.14.25 pyblp.exceptions.ClippedSharesError

class pyblp.exceptions.**ClippedSharesError**
Shares were clipped during the final iteration of the fixed point routine for computing δ.

5.14.26 pyblp.exceptions.ThetaConvergenceError

class pyblp.exceptions.**ThetaConvergenceError**
The optimization routine failed to converge.

This problem can sometimes be mitigated by choosing more reasonable initial parameter values, setting more conservative bounds, or configuring other optimization settings.

5.14.27 pyblp.exceptions.DeltaConvergenceError

class pyblp.exceptions.**DeltaConvergenceError**
The fixed point computation of δ failed to converge.

This problem can sometimes be mitigated by increasing the maximum number of fixed point iterations, increasing the fixed point tolerance, choosing more reasonable initial parameter values, setting more conservative parameter or share bounds, or using different iteration or optimization configurations.

5.14.28 pyblp.exceptions.SyntheticPricesConvergenceError

class pyblp.exceptions.**SyntheticPricesConvergenceError**
The fixed point computation of synthetic prices failed to converge.

This problem can sometimes be mitigated by increasing the maximum number of fixed point iterations, increasing the fixed point tolerance, configuring other iteration settings, or making sure the specified parameters are reasonable. For example, the parameters on prices should generally imply a downward sloping demand curve.

5.14.29 pyblp.exceptions.SyntheticDeltaConvergenceError

class pyblp.exceptions.**SyntheticDeltaConvergenceError**
The fixed point computation of the synthetic δ failed to converge.

This problem can sometimes be mitigated by increasing the maximum number of fixed point iterations, increasing the fixed point tolerance, choosing more reasonable parameter values, or using a different iteration configuration.

5.14.30 pyblp.exceptions.EquilibriumPricesConvergenceError

class pyblp.exceptions.**EquilibriumPricesConvergenceError**
The fixed point computation of equilibrium prices failed to converge.

This problem can sometimes be mitigated by increasing the maximum number of fixed point iterations, increasing the fixed point tolerance, or configuring other iteration settings.

5.14.31 pyblp.exceptions.ObjectiveReversionError

class pyblp.exceptions.**ObjectiveReversionError**
Reverted a problematic GMM objective value.

5.14.32 pyblp.exceptions.GradientReversionError

class pyblp.exceptions.**GradientReversionError**
Reverted problematic elements in the GMM objective gradient.

5.14.33 pyblp.exceptions.DeltaReversionError

class pyblp.exceptions.**DeltaReversionError**
Reverted problematic elements in δ.

5.14.34 pyblp.exceptions.CostsReversionError

class pyblp.exceptions.**CostsReversionError**
Reverted problematic marginal costs.

5.14.35 pyblp.exceptions.MicroMomentsReversionError

class pyblp.exceptions.**MicroMomentsReversionError**
Reverted problematic micro moments.

5.14.36 pyblp.exceptions.XiByThetaJacobianReversionError

class pyblp.exceptions.**XiByThetaJacobianReversionError**
Reverted problematic elements in the Jacobian (holding β fixed) of ξ (equivalently, of δ) with respect to θ.

5.14.37 pyblp.exceptions.OmegaByThetaJacobianReversionError

class pyblp.exceptions.**OmegaByThetaJacobianReversionError**
Reverted problematic elements in the Jacobian (holding γ fixed) of ω (equivalently, of transformed marginal costs) with respect to θ.

5.14.38 pyblp.exceptions.MicroMomentsByThetaJacobianReversionError

class pyblp.exceptions.**MicroMomentsByThetaJacobianReversionError**
Reverted problematic elements in the Jacobian of micro moments with respect to θ.

5.14.39 pyblp.exceptions.HessianEigenvaluesError

class pyblp.exceptions.**HessianEigenvaluesError**
Failed to compute eigenvalues for the GMM objective's (reduced) Hessian matrix.

5.14.40 pyblp.exceptions.ProfitHessianEigenvaluesError

class pyblp.exceptions.**ProfitHessianEigenvaluesError**
Failed to compute eigenvalues for a firm's profit Hessian.

5.14.41 pyblp.exceptions.FittedValuesInversionError

class pyblp.exceptions.**FittedValuesInversionError**
Failed to invert an estimated covariance when computing fitted values.

There are probably collinearity issues.

5.14.42 pyblp.exceptions.SharesByXiJacobianInversionError

class pyblp.exceptions.**SharesByXiJacobianInversionError**
Failed to invert a Jacobian of shares with respect to ξ when computing the Jacobian (holding β fixed) of ξ (equivalently, of δ) with respect to θ.

5.14.43 pyblp.exceptions.IntraFirmJacobianInversionError

class pyblp.exceptions.**IntraFirmJacobianInversionError**
Failed to invert an intra-firm Jacobian of shares with respect to prices.

5.14.44 pyblp.exceptions.PassthroughInversionError

class pyblp.exceptions.**PassthroughInversionError**
Failed to invert the matrix to recover the passthrough matrix.

5.14.45 pyblp.exceptions.LinearParameterCovariancesInversionError

class pyblp.exceptions.**LinearParameterCovariancesInversionError**
Failed to invert an estimated covariance matrix of linear parameters.

One or more data matrices may be highly collinear.

5.14.46 pyblp.exceptions.GMMParameterCovariancesInversionError

class pyblp.exceptions.**GMMParameterCovariancesInversionError**
Failed to invert an estimated covariance matrix of GMM parameters.

One or more data matrices may be highly collinear.

5.14.47 pyblp.exceptions.GMMMomentCovariancesInversionError

class pyblp.exceptions.**GMMMomentCovariancesInversionError**
Failed to invert an estimated covariance matrix of GMM moments.

One or more data matrices may be highly collinear.

CHAPTER

SIX

REFERENCES

This page contains a full list of references cited in the documentation, including the original work of *Berry, Levinsohn, and Pakes (1995)*. If you use PyBLP in your research, we ask that you also cite the below *Conlon and Gortmaker (2020)*, which describes the advances implemented in the package. If you use micro moments with PyBLP, we ask that you also cite *Conlon and Gortmaker (2025)*, which describes the standardized framework implemented by PyBLP for incorporating micro data into BLP-style estimation.

6.1 Conlon and Gortmaker (2020)

Conlon, Christopher, and Jeff Gortmaker (2020). Best practices for differentiated products demand estimation with PyBLP. *RAND Journal of Economics, 51* (4), 1108-1161.

6.2 Conlon and Gortmaker (2025)

Conlon, Christopher, and Jeff Gortmaker (2025). Incorporating micro data into differentiated products demand estimation with PyBLP. *Journal of Econometrics*, 105926.

6.3 Other References

6.3.1 Amemiya (1977)

Amemiya, Takeshi (1977). A note on a heteroscedastic model. *Journal of Econometrics, 6* (3), 365-370.

6.3.2 Andrews, Gentzkow, and Shapiro (2017)

Andrews, Isaiah, Matthew Gentzkow, and Jesse M. Shapiro (2017). Measuring the sensitivity of parameter estimates to estimation moments. *Quarterly Journal of Economics, 132* (4), 1553-1592.

6.3.3 Arellano and Bond (1991)

Arellano, Manuel and Stephen Bond (1991). Some tests of specification for panel data: Monte Carlo evidence and an application to employment equations. *Review of Economic Studies, 58* (2), 277–297.

6.3.4 Armstrong (2016)

Armstrong, Timothy B. (2016). Large market asymptotics for differentiated product demand estimators with economic models of supply. *Econometrica, 84* (5), 1961-1980.

6.3.5 Berry (1994)

Berry, Steven (1994). Estimating discrete-choice models of product differentiation. *RAND Journal of Economics, 25* (2), 242-262.

6.3.6 Berry, Levinsohn, and Pakes (1995)

Berry, Steven, James Levinsohn, and Ariel Pakes (1995). Automobile prices in market equilibrium. *Econometrica, 63* (4), 841-890.

6.3.7 Berry, Levinsohn, and Pakes (1999)

Berry, Steven, James Levinsohn, and Ariel Pakes (1999). Voluntary export restraints on automobiles: Evaluating a trade policy. *American Economic Review, 83* (9), 400-430.

6.3.8 Berry, Levinsohn, and Pakes (2004)

Berry, Steven, James Levinsohn, and Ariel Pakes (2004). Differentiated products demand systems from a combination of micro and macro data: The new car market. *Journal of Political Economy, 112* (1), 68-105.

6.3.9 Berry and Pakes (2007)

Berry, Steven, and Ariel Pakes (2007). The pure characteristics demand model. *International Economic Review, 48* (4), 1193-1225.

6.3.10 Brenkers and Verboven (2006)

Brenkers, Randy, and Frank Verboven (2006). Liberalizing a distribution system: The European car market. *Journal of the European Economic Association, 4* (1), 216-251.

6.3.11 Brunner, Heiss, Romahn, and Weiser (2017)

Brunner, Daniel, Florian Heiss, André Romahn, and Constantin Weiser (2017) Reliable estimation of random coefficient logit demand models. DICE Discussion Paper 267.

6.3.12 Cardell (1997)

Cardell, N. Scott (1997). Variance components structures for the extreme-value and logistic distributions with application to models of heterogeneity. *Econometric Theory, 13* (2), 185-213.

6.3.13 Chamberlain (1987)

Chamberlain, Gary (1987) Asymptotic efficiency in estimation with conditional moment restrictions. *Journal of Econometrics, 34* (3), 305-334.

6.3.14 Conlon and Mortimer (2021)

Conlon, Christopher, and Julie H. Mortimer (2021). Empirical properties of diversion ratios. *RAND Journal of Economics, 52* (4), 693-726.

6.3.15 Frisch and Waugh (1933)

Frisch, Ragnar, and Frederick V. Waugh (1933). Partial time regressions as compared with individual trends. *Econometrica, 1* (4), 387-401.

6.3.16 Gandhi and Houde (2025)

Gandhi, Amit, and Jean-Francois Houde (2025). Measuring substitution patterns in differentiated products industries.

6.3.17 Grigolon and Verboven (2014)

Grigolon, Laura, and Frank Verboven (2014). Nested logit or random coefficients logit? A comparison of alternative discrete choice models of product differentiation. *Review of Economics and Statistics, 96* (5), 916-935.

6.3.18 Hausman, Leonard, and Zona (1994)

Hausman, Jerry, Gregory Leonard, and J. Douglas Zona (1994). Competitive analysis with differentiated products. *Annals of Economics and Statistics, 34*, 143-157.

6.3.19 Hansen (1982)

Hansen, Lars Peter (1982). Large sample properties of generalized method of moments estimators. *Econometrica, 50* (4), 1029-1054.

6.3.20 Heiss and Winschel (2008)

Heiss, Florian, and Viktor Winschel (2008). Likelihood approximation by numerical integration on sparse grids. *Journal of Econometrics, 144* (1), 62-80.

6.3.21 Hess, Train, and Polak (2004)

Hess, Stephane, Kenneth E. Train, and John W. Polak (2004). On the use of a Modified Latin Hypercube Sampling (MLHS) method in the estimation of a mixed logit model for vehicle choice. *Transportation Research Part B* (40), 147-167.

6.3.22 Imbens and Lancaster (1994)

Imbens, Guido W., and Tony Lancaster (1994). Combining micro and macro data in microeconometric models. *Review of Economic Studies, 61* (4), 655-680.

6.3.23 Judd and Skrainka (2011)

Judd, Kenneth L., and Ben Skrainka (2011). High performance quadrature rules: How numerical integration affects a popular model of product differentiation. CeMMAP working paper CWP03/11.

6.3.24 Knittel and Metaxoglou (2014)

Knittel, Christopher R., and Konstantinos Metaxoglou (2014). Estimation of random-coefficient demand models: Two empiricists' perspective. *Review of Economics and Statistics, 96* (1), 34-59.

6.3.25 Lovell (1963)

Lovell, Michael C. (1963). Seasonal adjustment of economic time series and multiple regression analysis. *Journal of the American Statistical Association, 58* (304), 993-1010.

6.3.26 MacKay and Miller (2025)

MacKay, Alexander and Nathan Miller (2025). Estimating models of supply and demand: Instruments and covariance restrictions. *American Economic Journal: Microeconomics, 17* (1), 238-281.

6.3.27 Morrow and Skerlos (2011)

Morrow, W. Ross, and Steven J. Skerlos (2011). Fixed-point approaches to computing Bertrand-Nash equilibrium prices under mixed-logit demand. *Operations Research, 59* (2), 328-345.

6.3.28 Nevo (2000a)

Nevo, Aviv (2000). A practitioner's guide to estimation of random-coefficients logit models of demand. *Journal of Economics & Management Strategy, 9* (4), 513-548.

6.3.29 Nevo (2000b)

Nevo, Aviv (2000). Mergers with differentiated products: The case of the ready-to-eat cereal industry. *RAND Journal of Economics, 31* (3), 395-421.

6.3.30 Newey and West (1987)

Newey, Whitney K., and Kenneth D. West (1987). Hypothesis testing with efficient method of moments estimation. *International Economic Review, 28* (3), 777-787.

6.3.31 Owen (2013)

Owen, Art B. (2013). Monte Carlo theory, methods and examples.

6.3.32 Owen (2017)

Owen, Art B. (2017). A randomized Halton algorithm in R.

6.3.33 Petrin (2002)

Petrin, Amil (2002). Quantifying the benefits of new products: The case of the minivan. *Journal of Political Economy, 110* (4), 705-729.

6.3.34 Reynaert and Verboven (2014)

Reynaert, Mathias, and Frank Verboven (2014). Improving the performance of random coefficients demand models: The role of optimal instruments. *Journal of Econometrics, 179* (1), 83-98.

6.3.35 Reynaerts, Varadhan, and Nash (2012)

Reynaerts, Jo, Ravi Varadhan, and John C. Nash (2012). Enhancing the convergence properties of the BLP (1995) contraction mapping. VIVES discussion paper 35.

6.3.36 Varadhan and Roland (2008)

Varadhan, Ravi, and Christophe Roland (2008). Simple and globally convergent methods for accelerating the convergence of any EM algorithm. *Scandinavian Journal of Statistics, 35* (2), 335-353.

6.3.37 Werden (1997)

Werden, Gregory J. (1997). Simulating the effects of differentiated products mergers: A practitioners' guide. Economic Analysis Group, Proceedings of NE-165 Conference, Washington, D.C., June 20–21, 1996, 1997.

CHAPTER

SEVEN

LEGAL

Part II

Developer Documentation

CHAPTER

EIGHT

CONTRIBUTING

Please use the GitHub issue tracker to report bugs or to request features. Contributions are welcome. Examples include:

- Code optimizations.
- Documentation improvements.
- Alternate formulations that have been implemented in the literature but not in PyBLP.

CHAPTER

NINE

TESTING

Testing is done with the tox automation tool, which runs a pytest-backed test suite in the `tests/` directory.

9.1 Testing Requirements

In addition to the installation requirements for the package itself, running tests and building documentation requires additional packages specified by the `tests` and `docs` extras in `setup.py`, along with any other explicitly specified `deps` in `tox.ini`.

The full suite of tests also requires installation of the following software:

- Artleys Knitro version 10.3 or newer: testing optimization routines.
- MATLAB: comparing sparse grids with those created by the function nwspgr created by Florian Heiss and Viktor Winschel, which must be included in a directory on the MATLAB path.
- R: simulating nested logit errors created by the package evd created by Alec Stephenson, which must be installed.

If software is not installed, its associated tests will be skipped. Additionally, some tests that require support for extended precision will be skipped if on the platform running the tests, `numpy.longdouble` has the same precision as `numpy.float64`. This tends to be the case on Windows.

9.2 Running Tests

Defined in `tox.ini` are environments that test the package under different python versions, check types, enforce style guidelines, verify the integrity of the documentation, and release the package. First, tox should be installed on top of an Anaconda installation. The following command can be run in the top-level `pyblp` directory to run all testing environments:

```
tox
```

You can choose to run only one environment, such as the one that builds the documentation, with the `-e` flag:

```
tox -e docs
```

9.3 Test Organization

Fixtures, which are defined in `tests.conftest`, configure the testing environment and simulate problems according to a range of specifications.

Most BLP-specific tests in `tests.test_blp` verify properties about results obtained by solving the simulated problems under various parameterizations. Examples include:

- Reasonable formulations of problems should give rise to estimated parameters that are close to their true values.
- Cosmetic changes such as the number of processes should not change estimates.
- Post-estimation outputs should satisfy certain properties.
- Optimization routines should behave as expected.
- Derivatives computed with finite differences should approach analytic derivatives.

Tests of generic utilities in `tests.test_formulation`, `tests.test_integration`, `tests.test_iteration`, and `tests.test_optimization` verify that matrix formulation, integral approximation, fixed point iteration, and nonlinear optimization all work as expected. Example include:

- Nonlinear formulas give rise to expected matrices and derivatives.
- Gauss-Hermite integrals are better approximated with quadrature based on Gauss-Hermite rules than with Monte Carlo integration.
- To solve a fixed point iteration problem for which it was developed, SQUAREM requires fewer fixed point evaluations than does simple iteration.
- All optimization routines manage to solve a well-known optimization problem under different parameterizations.

CHAPTER

TEN

VERSION NOTES

These notes will only include major changes.

10.1 1.2

- Timing assumptions

10.2 1.1

- Covariance restrictions
- Demographic-specific product availability

10.3 1.0

- Support matching smooth functions of micro means
- Optimal micro moments
- Support elimination of groups of products for second choices
- Micro data simulation
- Micro moment tutorials

10.4 0.13

- Overhauled micro moment API
- Product-specific demographics
- Passthrough calculations
- Added problem results methods to simulation results
- Profit Hessian computation
- Checks of pricing second order conditions
- Newton-based methods for computing equilibrium prices

- Large speedups for supply-side and micro moment derivatives
- Universal display for fixed point iteration progress
- Support adjusting for simulation error in moment covariances

10.5 0.12

- Refactored micro moment API
- Custom micro moments
- Properly scale micro moment covariances
- Pickling support

10.6 0.11

- Elasticities and diversion ratios with respect to mean utility
- Willingness to pay calculations

10.7 0.10

- Simplify micro moment API
- Second choice or diversion micro moments
- Add share clipping to make fixed point more robust
- Report covariance matrix estimates in addition to Cholesky root
- Approximation to the pure characteristics model
- Add option to always use finite differences

10.8 0.9

- More control over matrices of instruments
- Split off fixed effect absorption into companion package PyHDFE
- Scrambled Halton and Modified Latin Hypercube Sampling (MLHS) integration
- Importance sampling
- Quantity dependent marginal costs
- Speed up various matrix construction routines
- Option to do initial GMM update at starting values
- Update BLP example data to better replicate original paper
- Lognormal random coefficients
- Removed outdated default parameter bounds

- Change default objective scaling for more comparable objective values across problem sizes
- Add post-estimation routines to simplify integration error comparison

10.9 0.8

- Micro moments that match product and agent characteristic covariances
- Extended use of pseudo-inverses
- Added more information to error messages
- More flexible simulation interface
- Alternative way to simulate data with specified prices and shares
- Tests of overidentifying and model restrictions
- Report projected gradients and reduced Hessians
- Change objective gradient scaling
- Switch to a lower-triangular covariance matrix to fix a bug with off-diagonal parameters

10.10 0.7

- Support more fixed point and optimization solvers
- Hessian computation with finite differences
- Simplified interface for firm changes
- Construction of differentiation instruments
- Add collinearity checks
- Update notation and explanations

10.11 0.6

- Optimal instrument estimation
- Structured all results as classes
- Additional information in progress reports
- Parametric bootstrapping of post-estimation outputs
- Replaced all examples in the documentation with Jupyter notebooks
- Updated the instruments for the BLP example problem
- Improved support for multiple equation GMM
- Made concentrating out linear parameters optional
- Better support for larger nesting parameters
- Improved robustness to overflow

10.12 0.5

- Estimation of nesting parameters
- Performance improvements for matrix algebra and matrix construction
- Support for Python 3.7
- Computation of reasonable default bounds on nonlinear parameters
- Additional information in progress updates
- Improved error handling and documentation
- Simplified multiprocessing interface
- Cancelled out delta in the nonlinear contraction to improve performance
- Additional example data and improvements to the example problems
- Cleaned up covariance estimation
- Added type annotations and overhauled the testing suite

10.13 0.4

- Estimation of a Logit benchmark model
- Support for fixing of all nonlinear parameters
- More efficient two-way fixed effect absorption
- Clustered standard errors

10.14 0.3

- Patsy- and SymPy-backed R-style formula API
- More informative errors and displays of information
- Absorption of arbitrary fixed effects
- Reduction of memory footprint

10.15 0.2

- Improved support for longdouble precision
- Custom ownership matrices
- New benchmarking statistics
- Supply-side gradient computation
- Improved configuration for the automobile example problem

10.16 0.1

- Initial release

Part III

Indices

PYTHON MODULE INDEX

p

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z