pyblp

Release 0.7.0

Jeff Gortmaker

Apr 27, 2019

TABLE OF CONTENTS

Part I

User Documentation

CHAPTER

ONE

INTRODUCTION

Note: This package is in beta. The API may change in future versions. Please use the GitHub issue tracker to report bugs or to request features.

The pyblp package 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 (2019), which describes the advances implemented in the package.

1.2 Installation

The pyblp package has been tested on Python versions 3.6 and 3.7. 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, along with many other packages that are useful for scientific computing, it comes packaged with pyblp's only required dependencies: NumPy, SciPy, SymPy, and Patsy.

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

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

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 Features

- R-style formula interface
- Bertrand-Nash supply-side moments
- Multiple equation GMM
- Demographic interactions
- Fixed effect absorption
- Nonlinear functions of product characteristics
- Concentrating out of linear parameters
- Parameter bounds and constraints
- Random coefficients nested logit (RCNL)
- Varying nesting parameters across groups
- Logit and nested logit benchmarks
- Classic BLP instruments
- Optimal instruments
- Elasticities and diversion ratios
- Marginal costs and markups
- Profits and consumer surplus
- Merger simulation
- Parametric boostrapping of post-estimation outputs
- Synthetic data construction
- SciPy or Artleys Knitro optimization
- Fixed point acceleration
- Monte Carlo, product rule, or sparse grid integration
- 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

1.4 Features Slated for Future Versions

- Micro moments
- Fast, "Robust," and Approximately Correct (FRAC) estimation
- Analytic Hessians
- Mathematical Program with Equilibrium Constraints (MPEC)
- Generalized Empirical Likelihood (GEL)
- Discrete types
- Pure characteristics model
- Newton methods for computing equilibrium prices

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 (2000)*, *Morrow and Skerlos (2011)*, *Grigolon and Verboven (2014)*, and others.

2.1 Dimensions and Sets

Symbol	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
I_t	Agents in market t
K_1	Linear product characteristics
K_1^x	Exogenous linear product characteristics
K_1^p	Endogenous linear product characteristics
K_2	Nonlinear product characteristics
K_3	Cost product characteristics
D	Demographic variables
M_D	Demand-side instruments, which is the number of exluded demand-side instruments plus K_1^x
M_S	Supply-side instruments, which is the number of exluded supply-side instruments plus K_3
E_D	Absorbed dimensions of demand-side fixed effects
E_S	Absorbed dimensions of supply-side fixed effects
H	Nesting groups
C	Clusters
P	Parameters
$\mathscr{J}_{ft}$	Set of products produced by firm f in market t
$\mathscr{J}_{ht}$	Set of products in nesting group h and market t
$\mathscr{J}_c$	Set of products in cluster c

2.2 Matrices, Vectors, and Scalars

Dimensions that can differ across markets are reported for a single market t. Some notation differs depending on how it is indexed.

Symbol	Dimensions	Description
X_1	$N \times K_1$	Linear product characteristics
X_1^x	$N \times K_1^x$	Exogenous linear product characteristics
X_1^p	$N \times K_1^p$	Endogenous linear product characteristics
X_2	$N \times K_2$	Nonlinear product characteristics
X_3	$N \times K_3$	Cost product characteristics
ξ	$N \times 1$	Unobserved demand-side product characteristics
ω	$N \times 1$	Unobserved supply-side product characteristics
p	$N \times 1$	Prices
s (s_{jt})	$N \times 1$	Marketshares
s (s_{ht})	$H \times 1$	Group shares in a market t
s (s_{jti})	$N \times I_t$	Choice probabiltiies 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
O	$J_t \times J_t$	Ownership 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) 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
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
ϵ	$N \times 1$	Type I Extreme Value idiosyncratic preferences
$\bar{\epsilon}$ ($\bar{\epsilon}_{jti}$)	$N \times 1$	Type I Extreme Value term used to decompose ϵ
$\bar{\epsilon}$ ($\bar{\epsilon}_{hti}$)	$N \times 1$	Group-specific term used to decompose ϵ
U	$J_t \times I_t$	Indirect utilities
V (V_{jti})	$J_t \times I_t$	Indirect utilities minus ϵ
V (V_{hti})	$J_t \times I_t$	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^x	$K_1^x \times 1$	Parameters in β on exogenous product characteristics
α	$K_1^p \times 1$	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
γ	$K_3 \times 1$	Supply-side linear parameters
θ	$P \times 1$	Parameters
Z_D	$N \times M_D$	Excluded demand-side instruments and X_1, except for X_1^p
Z_S	$N \times M_S$	Excluded supply-side instruments and X_3
W	$(M_D + M_S) \times (M_D + M_S)$	Weighting matrix
S	$(M_D + M_S) \times (M_D + M_S)$	Sample moment covariances or inverse of the weighting matrix
q	1×1	Objective value
g (g_{jt})	$N \times (M_D + M_S)$	Sample moments
g (g_c)	$C \times (M_D + M_S)$	Clustered sample moments
$\bar{g}$	$(M_D + M_S) \times 1$	Sample moment conditions
G	$(M_D + M_S) \times P$	Jacobian of the sample moment conditions with respect to θ

Continued on next page

Table 1 – continued from previous page

Symbol	Dimensions	Description
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 (2019)*.

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 $i = 1, 2, \ldots, I_t$ agents who choose among the J_t products and an outside good $j = 0$.

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

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

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

$$\delta = \underbrace{X_1^p \alpha + X_1^x \beta^x}_{X_1 \beta} + \xi. \tag{3.2}$$

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

The agent-specific portion of utility in a single market is

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

The model incorporates both observable (demographic) 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 $K_2 \times K_2$ upper 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.

Random idiosyncratic preferences, ϵ_{jti}, are assumed to be Type I Extreme Value, so that conditional on the heterogeneous coefficients, marketshares follow the well known logit form. Aggregate marketshares 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 a $I_t \times 1$ vector of integration weights, w:

$$s_{jt} \approx \sum_{i=1}^{I_t} w_i s_{jti}, \tag{3.4}$$

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

$$s_{jti} = \frac{\exp V_{jti}}{1 + \sum_{k=1}^{J_t} \exp V_{kti}}. \tag{3.5}$$

There is a one in the denominator because the utility of the outside good is normalized to $U_{0ti} = 0$.

3.1.2 Supply

Observed supply-side product characteristics are contained in the $N \times K_3$ matrix of cost characteristics, X_3. Prices cannot be cost 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 $\mathscr{J}_{ft} \subset \{1, 2, \ldots, J_t\}$:

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

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) demand derivatives:

$$\Delta = -O \odot \frac{\partial s}{\partial p}. \tag{3.8}$$

Here, O denotes the market-level ownership matrix, where O_{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, a $P \times 1$ vector.

The GMM problem is

$$\min_{\theta} q(\theta) = N^2 \bar{g}(\theta)' W \bar{g}(\theta), \tag{3.10}$$

in which W is a $(M_D + M_S) \times (M_D + M_S)$ weighting matrix and $\bar{g}$ is a $(M_D + M_S) \times 1$ vector of sample moment conditions:

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

in which Z_D and Z_S are $N \times M_D$ and $N \times M_S$ matrices of demand- and supply-side instruments containing excluded instruments along with X_1^x and X_3, respectively.

The vector $\bar{g}$ is the sample analogue of the GMM moment conditions $E[g_{jt}] = 0$ where

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

defines the $N \times (M_D + M_S)$ matrix of GMM moments g.

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

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

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

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

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

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

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

where

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

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

Finally, the unobserved product characteristics (structural errors),

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

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

3.2.2 The Gradient

The gradient of the GMM objective in (3.10) is

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

where

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

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):

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

where the second term is derived from the definition of η in (3.7):

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

3.2.3 Weighting Matrices

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

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

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

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

For heteroscedasticity robust weighting matrices,

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

For clustered weighting matrices, which account for arbitrary correlation within $c = 1, 2, \ldots, C$ clusters,

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

where, letting the set $\mathscr{J}_c \subset \{1, 2, \ldots, N\}$ denote products in cluster c,

$$q_c = \sum_{j \in \mathscr{J}_c} g_j. \tag{3.27}$$

For unadjusted weighting matrices,

$$S = \frac{1}{N} \begin{bmatrix} \text{Var}(\xi) Z_D' Z_D & \text{Cov}(\xi, \omega) Z_D' Z_S \\ \text{Cov}(\xi, \omega) Z_S' Z_D & \text{Var}(\omega) Z_S' Z_S \end{bmatrix}. \tag{3.28}$$

3.2.4 Standard Errors

The covariance matrix of the estimated parameters is

$$\text{Var}(\hat{\theta}) = (\bar{G}'W\bar{G})^{-1}\bar{G}'WSW\bar{G}(\bar{G}'W\bar{G})^{-1}. \tag{3.29}$$

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

$$\text{Var}(\hat{\theta}) = (\bar{G}'W\bar{G})^{-1}. \tag{3.30}$$

Standard errors extracted from this simpler expression are called unadjusted.

3.3 Fixed Effects

The unobserved product characteristics can be partitioned into

$$\begin{bmatrix}\xi_{jt} \\ \omega_{jt}\end{bmatrix} = \begin{bmatrix}\xi_{k_1} + \xi_{k_2} + \cdots + \xi_{k_{E_D}} + \Delta\xi_{jt} \\ \omega_{\ell_1} + \omega_{\ell_2} + \cdots + \omega_{\ell_{E_S}} + \Delta\omega_{jt}\end{bmatrix} \tag{3.31}$$

where $k_1, k_2, \ldots, k_{E_D}$ and $\ell_1, \ell_2, \ldots, \ell_{E_S}$ index unobserved characteristics that are fixed across E_D and E_S dimensions. For example, with $E_D = 1$ dimension of product fixed effects, $\xi_{jt} = \xi_j + \Delta\xi_{jt}$.

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 $Y(\hat{\theta})$ in (3.15). With one fixed effect, these matrices are simply de-meaned within each level of the fixed effect. Both X and Z can be de-meaned just once, but $Y(\hat{\theta})$ must be de-meaned for each new $\hat{\theta}$.

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 with the structural errors $\Delta\xi$ and $\Delta\omega$.

3.4 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 $\mathscr{J}_{ht} \subset \{1, 2, \ldots, J_t\}$ denotes the products in group h and market t.

In the RCNL model, idiosyncratic preferences are partitioned into

$$\epsilon_{jti} = \bar{\epsilon}_{h(j)ti} + (1 - \rho_{h(j)})\bar{\epsilon}_{jti} \tag{3.32}$$

where $\bar{\epsilon}_{jti}$ is Type I Extreme Value and $\bar{\epsilon}_{h(j)ti}$ is distributed such that ϵ_{jti} 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:

$$s_{jti} = \frac{\exp[V_{jti}/(1 - \rho_{h(j)})]}{\exp[V_{h(j)ti}/(1 - \rho_{h(j)})]} \cdot \frac{\exp V_{h(j)ti}}{1 + \sum_{h=1}^{H} \exp V_{hti}} \tag{3.33}$$

where

$$V_{hti} = (1 - \rho_h) \log \sum_{k \in \mathscr{J}_{ht}} \exp[V_{kti}/(1 - \rho_h)]. \tag{3.34}$$

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

$$\delta_{jt} \leftarrow \delta_{jt} + (1 - \rho_{h(j)})[\log s_{jt} - \log s_{jt}(\delta, \hat{\theta})]. \tag{3.35}$$

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

3.5 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,

$$\delta_{jt} = \log s_{jt} - \log s_{0t}, \tag{3.36}$$

and a lack of nonlinear parameters means that nonlinear optimization is not needed.

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

$$\delta_{jt} = \log s_{jt} - \log s_{0t} - \rho_{h(j)}[\log s_{jt} - \log s_{h(j)t}]. \tag{3.37}$$

where

$$s_{ht} = \sum_{j \in \mathscr{J}_{ht}} s_{jt}. \tag{3.38}$$

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 $w_1 = 1$.

3.6 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):

$$p - c = \underbrace{\Lambda^{-1}(O \odot \Gamma)'(p - c) - \Lambda^{-1}}_{\zeta} \tag{3.39}$$

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

$$\Lambda_{jj} \approx \sum_{i=1}^{I_t} w_i s_{jti} \frac{\partial U_{jti}}{\partial p_{jt}} \tag{3.40}$$

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

$$\Gamma_{jk} \approx \sum_{i=1}^{I_t} w_i s_{jti} s_{kti} \frac{\partial U_{jti}}{\partial p_{jt}}. \tag{3.41}$$

Equilibrium prices are computed by iterating over the ζ-markup equation in (3.39),

$$p \leftarrow c + \zeta(p), \tag{3.42}$$

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.

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. 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__
```

```
'0.7.0'
```

In this tutorial, we'll use data from *Nevo (2000)* 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 (2000)*.

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

$$U_{jti} = \alpha p_{jt} + x_{jt}\beta^x + \xi_{jt} + \epsilon_{jti}, \tag{4.1}$$

where ϵ_{jti} 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 $U_{0ti} = \epsilon_{0ti}$. This gives us aggregate marketshares

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

If we take logs we get

$$\log s_{jt} = \alpha p_{jt} + x_{jt}\beta^x + \xi_{jt} - 0 - \log \sum_k \exp(\alpha p_{jt} + x_{kt}\beta^x + \xi_{kt}) \tag{4.3}$$

and

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

By differencing the above we get a linear estimating equation:

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

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 (2000)*. Since we won't include any 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 marketshares 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  \
0  0.072088      2      1          ...                       2.116358
1  0.114178     18      1          ...                      -7.374091
2  0.132391      4      1          ...                       2.187872
3  0.130344      3      0          ...                       2.704576
4  0.154823     12      0          ...                       1.261242

   demand_instruments11  demand_instruments12  demand_instruments13  \
0             -0.154708             -0.005796              0.014538
1             -0.576412              0.012991              0.076143
2             -0.207346              0.003509              0.091781
3              0.040748             -0.003724              0.094732
4              0.034836             -0.000568              0.102451

   demand_instruments14  demand_instruments15  demand_instruments16  \
0              0.126244              0.067345              0.068423
1              0.029736              0.087867              0.110501
2              0.163773              0.111881              0.108226
3              0.135274              0.088090              0.101767
4              0.130640              0.084818              0.101075

   demand_instruments17  demand_instruments18  demand_instruments19
0              0.034800              0.126346              0.035484
1              0.087784              0.049872              0.072579
2              0.086439              0.122347              0.101842
3              0.101777              0.110741              0.104332
4              0.125169              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:
=========================================
Computation  GMM    Objective   Objective
   Time      Step  Evaluations    Value
-----------  ----  -----------  ---------
 00:00:01     2          2       +4.2E+05
=========================================

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

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

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 marketshares 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 $V_{jt} = \alpha p_{jt} + x_{jt}\beta^x + \xi_{jt}$.

After some work we again obtain the linear estimating equation:

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

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

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

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:
=========================================================================================
Computation  GMM    Optimization   Objective   Objective    Gradient      Hessian
   Time      Step   Iterations   Evaluations    Value    Infinity Norm  Eigenvalue
-----------  ----  ------------  -----------  ---------  -------------  ----------
 00:00:11     2          3             8       +4.6E+05     +3.1E-06      +2.4E+07
=========================================================================================

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

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

(continues on next page)

(continued from previous page)

```
 -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:
================================================================================
Computation  GMM   Optimization   Objective   Objective    Gradient      Hessian
   Time      Step   Iterations   Evaluations    Value    Infinity Norm  Eigenvalue
-----------  ----  ------------  -----------  ---------  -------------  ----------
 00:00:13     2          3             8       +1.6E+06     +9.4E-06      +1.3E+07
================================================================================

Rho Estimates (Robust SEs in Parentheses):
==========
All Groups
```

(continues on next page)

(continued from previous page)

```
----------
 +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(np.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)')
```

(continues on next page)

(continued from previous page)

```
    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:
=========================================
Computation  GMM    Objective   Objective
   Time      Step  Evaluations    Value
-----------  ----  -----------  ---------
 00:00:01     2          2       +4.6E+05
=========================================

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:
=========================================
Computation  GMM    Objective   Objective
   Time      Step  Evaluations    Value
-----------  ----  -----------  ---------
 00:00:01     2          2       +1.6E+06
=========================================

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

(continues on next page)

(continued from previous page)

```
 -6.8E+00        +9.3E-01
(+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([-86.37368446])
```

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

```
array([-100.14496892])
```

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__
```

```
'0.7.0'
```

In this tutorial, we'll use data from *Nevo (2000)* 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

$$u_{jti} = \alpha_i p_{jt} + x_{jt}\beta_i^x + \xi_{jt} + \epsilon_{jti} \tag{4.10}$$

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

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

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

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, $\delta_{jt} = \alpha p_{jt} + x_{jt}\beta^x + \xi_{jt}$, from the individual specific heterogeneity, $\mu_{jti}(\theta)$. This gives us

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

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

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

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 Application of Random Coefficients Logit with the Fake Cereal Data

To include random coefficients we need to add a *Formulation* for the 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 marketshares 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.

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, Σ .
3. Any $K_2 \times D$ interactions, Π, with observed $D \times 1$ demographic data, d_i.

Together this gives us that

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

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.

As is common with multivariate normal distributions, Σ is not estimated directly. Rather, its matrix square (Cholesky) root L is estimated where $LL' = \Sigma$.

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  \
0  0.072088      2      1          ...                       2.116358
1  0.114178     18      1          ...                      -7.374091
2  0.132391      4      1          ...                       2.187872
3  0.130344      3      0          ...                       2.704576
4  0.154823     12      0          ...                       1.261242

   demand_instruments11  demand_instruments12  demand_instruments13  \
0             -0.154708             -0.005796              0.014538
1             -0.576412              0.012991              0.076143
2             -0.207346              0.003509              0.091781
3              0.040748             -0.003724              0.094732
4              0.034836             -0.000568              0.102451

   demand_instruments14  demand_instruments15  demand_instruments16  \
0              0.126244              0.067345              0.068423
1              0.029736              0.087867              0.110501
2              0.163773              0.111881              0.108226
3              0.135274              0.088090              0.101767
4              0.130640              0.084818              0.101075

   demand_instruments17  demand_instruments18  demand_instruments19
0              0.034800              0.126346              0.035484
1              0.087784              0.049872              0.072579
2              0.086439              0.122347              0.101842
3              0.101777              0.110741              0.104332
4              0.125169              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.

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 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.
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, seed=0)
mc_integration
```

```
Configured to construct nodes and weights with Monte Carlo simulation.
```

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

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

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

```
Dimensions:
============================================
```

(continues on next page)

(continued from previous page)

```
 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.

```
bfgs = pyblp.Optimization('bfgs')
bfgs
```

```
Configured to optimize using the BFGS algorithm implemented in SciPy with analytic gradients and options {}.
```

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 lower diagonal in this initial guess is ignored because we are optimizing over the upper triangular (Cholesky) root of Σ.

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

```
Problem Results Summary:
=========================================================================================================================
                                                                                                  Smallest    Largest
Computation  GMM   Optimization   Objective   Fixed Point  Contraction  Objective    Gradient      Hessian     Hessian
   Time      Step   Iterations   Evaluations  Iterations   Evaluations    Value    Infinity Norm  Eigenvalue  Eigenvalue
-----------  ----  ------------  -----------  -----------  -----------  ---------  -------------  ----------  ----------
 00:05:19     2         62           171        175814       541988     +3.0E+05     +8.0E-03      +1.5E+02    +1.2E+07
=========================================================================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=====================================================
Sigma:       1          prices       sugar        mushy
------  ----------  ----------  ----------  ----------
  1      +9.1E-02    +1.3E-01    +1.1E-01    -5.7E-02
        (+1.8E+00)  (+8.2E+00)  (+9.5E+00)  (+4.3E+00)

prices                -1.4E+00    +3.5E+00    +1.3E+01
                     (+6.6E+01)  (+7.4E+01)  (+1.9E+01)

sugar                             -1.1E-02    -1.4E-01
                                 (+1.7E-01)  (+1.8E-01)

mushy                                         -2.6E-02
                                             (+7.5E-01)
=====================================================

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
----------
 -3.4E+01
(+1.5E+01)
==========
```

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

```
Problem Results Summary:
=========================================================================================================================
                                                                                                  Smallest    Largest
Computation  GMM   Optimization   Objective   Fixed Point  Contraction  Objective    Gradient      Hessian     Hessian
   Time      Step   Iterations   Evaluations  Iterations   Evaluations    Value    Infinity Norm  Eigenvalue  Eigenvalue
-----------  ----  ------------  -----------  -----------  -----------  ---------  -------------  ----------  ----------
 00:10:49     2         69           153        100768       312910     +3.6E+05     +1.3E-02      +3.3E+00    +1.2E+07
=========================================================================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=====================================================
Sigma:      1          prices      sugar       mushy
------  ----------  ----------  ----------  ----------
  1      -9.8E-10    -1.0E-09    +3.2E-01    +6.7E-01
        (+2.2E+09)  (+5.1E+09)  (+9.9E+01)  (+5.2E+01)

prices                +1.4E-08    -6.2E+00    -1.1E+01
                     (+2.8E+10)  (+1.6E+03)  (+9.3E+02)

sugar                            +5.7E-02    +9.7E-02
                                (+1.4E+01)  (+7.9E+00)

mushy                                        -1.6E-01
                                            (+1.5E+01)
=====================================================

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

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

```
Problem Results Summary:
```

(continues on next page)

(continued from previous page)

```
=============================================================================================================================
                                                                                                    Smallest    Largest
Computation  GMM   Optimization   Objective   Fixed Point  Contraction  Objective    Gradient      Hessian     Hessian
   Time      Step   Iterations   Evaluations  Iterations   Evaluations    Value    Infinity Norm  Eigenvalue  Eigenvalue
-----------  ----  ------------  -----------  -----------  -----------  ---------  -------------  ----------  ----------
 00:01:19     2         16           77          43049       134868     +4.0E+05     +3.9E-04      +2.4E+03    +1.4E+07
=============================================================================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=====================================================
Sigma:      1          prices       sugar       mushy
------  ----------  ----------  ----------  ----------
  1      +5.2E-02    +0.0E+00    +0.0E+00    +0.0E+00
        (+1.1E+00)

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

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

mushy                                         +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.

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_{ti} attached to each agent. In each market, these should sum to one so that $\sum_i w_{ti} = 1$. It is often the case the $w_{ti} = 1/I_t$ 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_{jti} 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 (2000)*. 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, 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     ]
])
nevo_results = nevo_problem.solve(
    initial_sigma,
    initial_pi,
    optimization=bfgs,
    method='1s'
```

(continues on next page)

(continued from previous page)

```
)
nevo_results
```

```
Problem Results Summary:
=========================================================================================================================
                                                                                                  Smallest    Largest
Computation  GMM   Optimization   Objective   Fixed Point  Contraction  Objective    Gradient      Hessian     Hessian
   Time      Step   Iterations   Evaluations  Iterations   Evaluations    Value    Infinity Norm  Eigenvalue  Eigenvalue
-----------  ----  ------------  -----------  -----------  -----------  ---------  -------------  ----------  ----------
 00:01:18     1          51           58          46236       143503     +4.6E+00     +6.9E-06      +2.8E-05    +1.6E+04
=========================================================================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=============================================================================================================
Sigma:      1          prices      sugar       mushy      |   Pi:      income    income_squared     age         child
------  ----------  ----------  ----------  ----------  |  ------  ----------  --------------  ----------  ----------
  1      +5.6E-01    +0.0E+00    +0.0E+00    +0.0E+00   |    1      +2.3E+00      +0.0E+00      +1.3E+00    +0.0E+00
        (+1.6E-01)                                      |          (+1.2E+00)                   (+6.3E-01)
                                                        |
prices                +3.3E+00    +0.0E+00    +0.0E+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                            -5.8E-03    +0.0E+00   |  sugar    -3.8E-01      +0.0E+00      +5.2E-02    +0.0E+00
                                (+1.4E-02)              |          (+1.2E-01)                   (+2.6E-02)
                                                        |
mushy                                       +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)
==========
```

Results are similar to those in the original paper with a GMM objective value of $q(\hat{\theta}) = 4.56$ and a price coefficient of $\hat{\alpha} = -62.7$.

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=bfgs,
    method='1s'
)
```

```
Problem Results Summary:
=========================================================================================================================
                                                                                                  Smallest    Largest
Computation  GMM   Optimization   Objective   Fixed Point  Contraction  Objective    Gradient      Hessian     Hessian
   Time      Step   Iterations   Evaluations  Iterations   Evaluations    Value    Infinity Norm  Eigenvalue  Eigenvalue
-----------  ----  ------------  -----------  -----------  -----------  ---------  -------------  ----------  ----------
 00:00:57     1          34           41          34457       106706     +1.5E+01     +5.2E-06       +4.7E-02    +1.7E+04
=========================================================================================================================

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

(continues on next page)

(continued from previous page)

```
                                                  |
mushy                                      +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(
    initial_sigma,
    deleted_pi,
    optimization=bfgs,
    method='1s'
)
```

```
Problem Results Summary:
=============================================================================================================================
                                                                                                  Smallest    Largest
Computation  GMM   Optimization   Objective   Fixed Point  Contraction  Objective    Gradient      Hessian     Hessian
   Time      Step   Iterations   Evaluations  Iterations   Evaluations    Value    Infinity Norm  Eigenvalue  Eigenvalue
-----------  ----  ------------  -----------  -----------  -----------  ---------  -------------  ----------  ----------
 00:00:54     1         34           41          34472       106726     +1.5E+01     +5.2E-06      +4.7E-02    +1.7E+04
=============================================================================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
============================================================================================
```

(continues on next page)

(continued from previous page)

```
Sigma:      1          prices      sugar       mushy      |   Pi:      income       age        child
------  ----------  ----------  ----------  ----------  |  ------  ----------  ----------  ----------
  1      +3.7E-01    +0.0E+00    +0.0E+00    +0.0E+00   |    1      +3.1E+00    +1.2E+00    +0.0E+00
        (+1.2E-01)                                      |           (+1.1E+00)  (+1.0E+00)
                                                        |
prices                +1.8E+00    +0.0E+00    +0.0E+00   |  prices   +4.2E+00    +0.0E+00    +1.2E+01
                     (+9.2E-01)                         |           (+4.6E+00)              (+5.2E+00)
                                                        |
sugar                             -4.4E-03    +0.0E+00   |  sugar    -1.9E-01    +2.8E-02    +0.0E+00
                                 (+1.2E-02)             |           (+3.5E-02)  (+3.2E-02)
                                                        |
mushy                                        +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):
==========
  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 Random Coefficients Logit Tutorial with the Automobile Data

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

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

```
'0.7.0'
```

In this tutorial, we'll use data from *Berry, Levinsohn, and Pakes (1995)* to solve the paper's automobile problem.

4.3.1 Application of Random Coefficients Logit with the Automobile Data

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.

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

       hpwt  air       mpd         ...           demand_instruments2  \
0  0.528997    0  1.888146         ...                      0.566217
1  0.494324    0  1.935989         ...                      0.566217
2  0.467613    0  1.716799         ...                      0.566217
3  0.426540    0  1.687871         ...                      0.566217
4  0.452489    0  1.504286         ...                      0.566217

   demand_instruments3  demand_instruments4  demand_instruments5  \
0             0.365328             0.659480             0.141017
1             0.290959             0.173552             0.128205
2             0.599771            -0.546387             0.002634
3             0.620544            -1.122968             0.089023
4             0.877198            -1.258575            -0.153840

   supply_instruments0  supply_instruments1  supply_instruments2  \
0            -0.011161             1.478879            -0.546875
1            -0.079317             1.088327            -0.546875
2             0.021034             0.609213            -0.546875
3            -0.090014             0.207461            -0.546875
4             0.038013             0.385211            -0.546875

   supply_instruments3  supply_instruments4  supply_instruments5
0            -0.163302            -0.833091             0.301411
1            -0.095609            -0.390314             0.289947
2            -0.449818             0.400461             0.434632
3            -0.454159             0.934641             0.331099
4            -0.728959             1.146654             0.520555

[5 rows x 25 columns]
```

The `product_data` contains market IDs, product IDs, firm IDs, shares, prices, a number of product characteristics, and some pre-computed excluded 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_{ti}, integration nodes ν_{ti}, and demographics d_{ti}. Here we use $I_t = 200$ equally weighted Monte

Carlo draws per market.

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    0.005  0.548814  0.457760  0.564690  0.395537  0.392173
1        1971    0.005  0.715189  0.376918  0.839746  0.844017  0.041157
2        1971    0.005  0.602763  0.702335  0.376884  0.150442  0.923301
3        1971    0.005  0.544883  0.207324  0.499676  0.306309  0.406235
4        1971    0.005  0.423655  0.074280  0.081302  0.094570  0.944282

      income
0   9.728478
1   7.908957
2  11.079404
3  17.641671
4  12.423995
```

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`.

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

```
Dimensions:
=============================================================
 T     N      F     I      K1     K2     K3     D     MD     MS
---   ----   ---   ----   ----   ----   ----   ---   ----   ----
20    2217   26    4000   5      6      6      1     11     12
=============================================================

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: 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 linear characteristics, nonlinear characteristics, cost characteristics, and demographics.

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.
- Choosing a form for marginal costs, c_{jt}: either a linear or log-linear functional form.

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.
- Constrain the p_j/y_i coefficient to be negative. Specifically, we'll use a bound that's slightly smaller than zero because when the parameter is exactly zero, there are matrix inversion problems with estimating marginal costs.

When using a routine that supports bounds, *Problem* chooses some default bounds. These bounds are often very wide, so it's usually a good idea to set your own more restrictive bounds.

```
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]]
sigma_bounds = (
   np.zeros_like(initial_sigma),
   np.diag([100, 0, 100, 100, 100, 100])
)
pi_bounds = (
   np.c_[[0, -100, 0, 0, 0, 0]],
   np.c_[[0, -0.1, 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 linear marginal cost specification is the default setting, so we'll need to use the `costs_type` argument of *Problem.solve* to employ the log-linear specification used by *Berry, Levinsohn, and Pakes (1995)*. A downside of this 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`.

```
results = problem.solve(
    initial_sigma,
    initial_pi,
    sigma_bounds=sigma_bounds,
    pi_bounds=pi_bounds,
    costs_type='log',
    costs_bounds=(0.001, None),
    W_type='clustered',
    se_type='clustered'
)
results
```

```
Problem Results Summary:
=======================================================================================================================================
                                                                                                                 Smallest     Largest ␣
↪   Clipped
Computation  GMM   Optimization   Objective   Fixed Point  Contraction  Objective    Gradient      Hessian      Hessian ␣
↪  Marginal
   Time      Step   Iterations   Evaluations  Iterations   Evaluations    Value    Infinity Norm  Eigenvalue ␣
↪Eigenvalue   Costs
-----------  ----  ------------  -----------  -----------  -----------  ---------  -------------  ----------  ----------
↪   --------
 00:31:03     2         39           49          72322       217903     +2.1E+05     +2.8E+03     +1.8E+02    +1.8E+04␣
↪      0
=======================================================================================================================================

Nonlinear Coefficient Estimates (Robust SEs Adjusted for 999 Clusters in Parentheses):
=====================================================================================================
Sigma:      1          prices        hpwt          air          mpd         space      |  Pi:     1/income
------  ----------  ----------  ----------  ----------  ----------  ----------  |  ------  ----------
  1      +1.2E+00    +0.0E+00    +0.0E+00    +0.0E+00    +0.0E+00    +0.0E+00   |   1      +0.0E+00
        (+2.1E+00)                                                               |
```

(continues on next page)

(continued from previous page)

```
                                                                                                  |
prices                +0.0E+00    +0.0E+00    +0.0E+00    +0.0E+00    +0.0E+00                   |  prices   -1.2E+01
                                                                                                  |           (+4.8E+00)
                                                                                                  |
 hpwt                             +0.0E+00    +0.0E+00    +0.0E+00    +0.0E+00                  |   hpwt     +0.0E+00
                                 (+1.0E+01)                                                       |
                                                                                                  |
 air                                         +0.0E+00    +0.0E+00    +0.0E+00                  |   air      +0.0E+00
                                            (+2.6E+00)                                            |
                                                                                                  |
 mpd                                                     +2.6E+00    +0.0E+00                  |   mpd      +0.0E+00
                                                        (+1.4E+00)                                |
                                                                                                  |
space                                                                +1.1E+00                  |  space     +0.0E+00
                                                                    (+1.9E+00)                    |
==================================================================================================================

Beta Estimates (Robust SEs Adjusted for 999 Clusters in Parentheses):
============================================================
    1          hpwt        air         mpd        space
----------  ----------  ----------  ----------  ----------
 -7.6E+00    +4.8E+00    +2.6E+00    -2.2E+00    +1.8E+00
(+1.2E+00)  (+2.7E+00)  (+1.4E+00)  (+1.5E+00)  (+1.3E+00)
============================================================

Gamma Estimates (Robust SEs Adjusted for 999 Clusters in Parentheses):
======================================================================
    1       log(hpwt)      air       log(mpg)   log(space)    trend
----------  ----------  ----------  ----------  ----------  ----------
 +2.3E+00    +6.4E-01    +8.8E-01    -3.9E-01    +2.8E-01    +1.4E-02
(+1.4E-01)  (+1.2E-01)  (+7.1E-02)  (+1.2E-01)  (+1.2E-01)  (+3.5E-03)
======================================================================
```

There are some discrepancies between our results and the original paper. The instruments we constructed to are meant to mimic the original instruments, which we were unable to re-construct perfectly. We also use different agent data and the first-order linear approximation to the $\ln(y_i - p_j)$ term.

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

4.4 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__
```

```
'0.7.0'
```

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.

4.4.1 Problem Results

As in the *fake cereal tutorial*, we'll first solve the fake cereal problem from *Nevo (2000)*. 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)
product_formulations = (
    pyblp.Formulation('0 + prices', absorb='C(product_ids)'),
```

(continues on next page)

(continued from previous page)

```
   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     ]
]
bfgs = pyblp.Optimization('bfgs')
results = problem.solve(
    initial_sigma,
    initial_pi,
    optimization=bfgs,
    method='1s'
)
results
```

```
Problem Results Summary:
=========================================================================================================================
                                                                                                  Smallest    Largest
Computation  GMM   Optimization   Objective   Fixed Point  Contraction  Objective    Gradient      Hessian     Hessian
   Time      Step   Iterations   Evaluations  Iterations   Evaluations    Value    Infinity Norm  Eigenvalue  Eigenvalue
-----------  ----  ------------  -----------  ----------  -----------  ---------  -------------  ----------  ----------
 00:02:06     1         51           58          46236       143503     +4.6E+00     +6.9E-06      +2.8E-05    +1.6E+04
=========================================================================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=============================================================================================================
Sigma:      1          prices      sugar       mushy      |   Pi:      income    income_squared     age         child
------  ----------  ----------  ----------  ----------  |  ------  ----------  --------------  ----------  ----------
  1      +5.6E-01    +0.0E+00    +0.0E+00    +0.0E+00   |    1      +2.3E+00      +0.0E+00      +1.3E+00    +0.0E+00
        (+1.6E-01)                                      |          (+1.2E+00)                   (+6.3E-01)
                                                        |
prices                +3.3E+00    +0.0E+00    +0.0E+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                             -5.8E-03    +0.0E+00   |  sugar    -3.8E-01      +0.0E+00      +5.2E-02    +0.0E+00
                                 (+1.4E-02)             |          (+1.2E-01)                   (+2.6E-02)
                                                        |
mushy                                         +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.

4.4.2 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

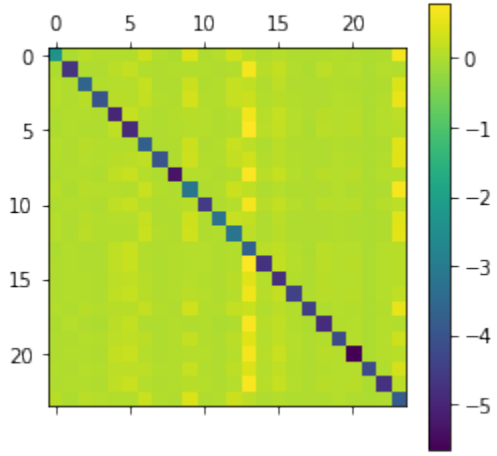
$$\mathscr{D}_{jk} = -\frac{\partial s_k}{\partial x_j} \Big/ \frac{\partial s_j}{\partial x_j}. \tag{4.17}$$

Following *Conlon and Mortimer (2018)*, we report the diversion to the outside good D_{j0} on the diagonal instead of $D_{jj} = -1$.

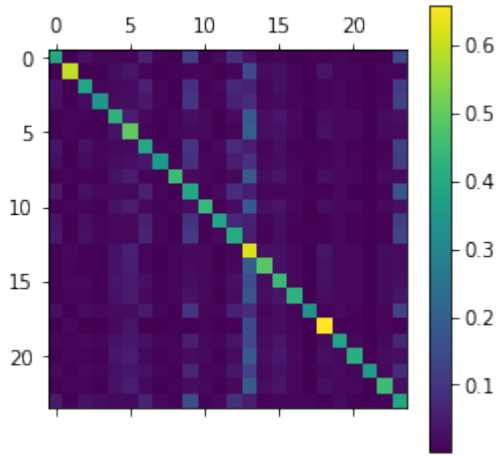
```
elasticities = results.compute_elasticities()
diversions = results.compute_diversion_ratios()
```

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)
```

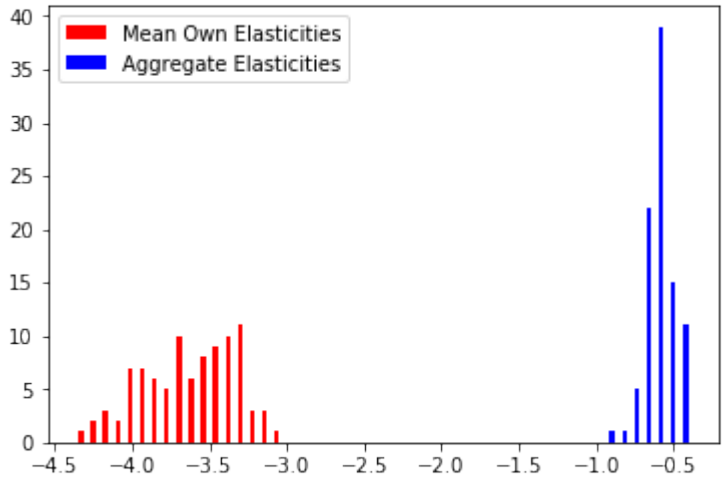
Since demand for an entire product category is generally less elastic than the average elasticity of individual products, mean own elasticities are generally larger in magnitude than aggregate elasticities.

```
plt.hist(
    [means.flatten(), aggregates.flatten()],
    color=['red', 'blue'],
```

(continues on next page)

(continued from previous page)

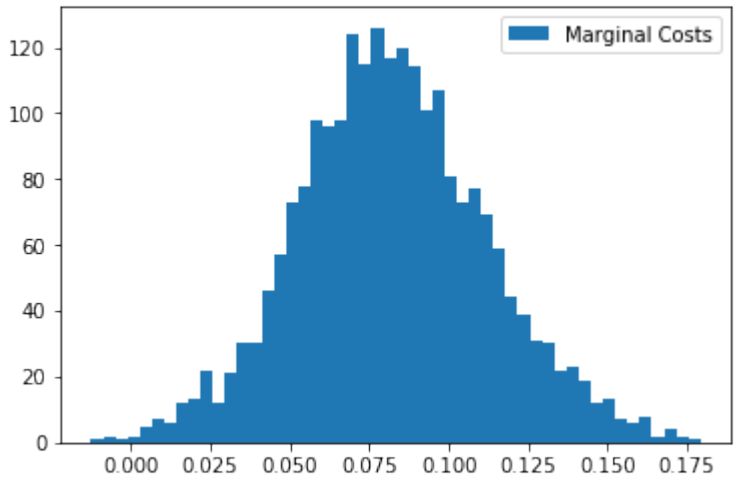
```
    bins=50
);
plt.legend(['Mean Own Elasticities', 'Aggregate Elasticities']);
```



4.4.3 Marginal Costs and Markups

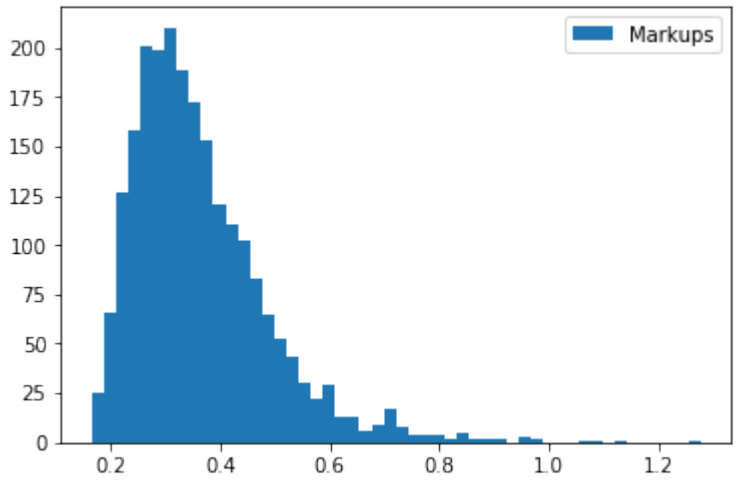
To compute marginal costs, c, the `product_data` passed to *Problem* must have had a `firm_ids` field. Since we included firm IDs when configuring the problem, we can use *ProblemResults.compute_costs*.

```
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"]);
```



4.4.4 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 discussed, for example, in *Nevo (1997)*, assumes that shares and their price derivatives are unaffected by the merger. The second method does not make these assumptions and iterates 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'],
```

(continues on next page)

(continued from previous page)

```
    costs=costs
)
```

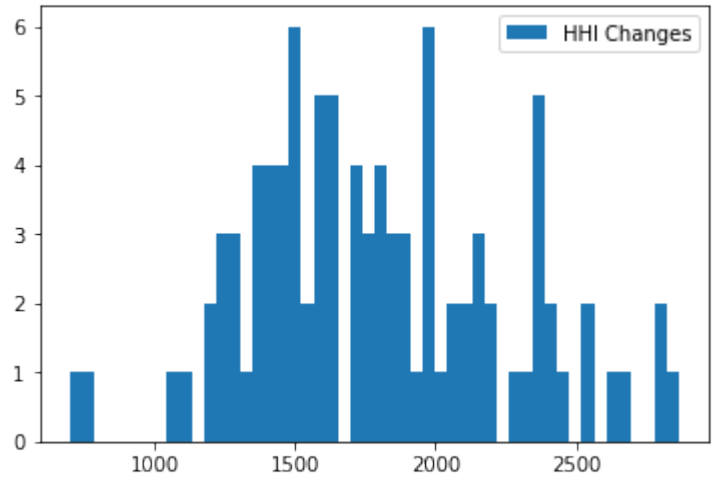
If the problem was configured with more than two columns of firm IDs, we could estimate post-merger prices for the other mergers with the `firms_index` argument, which is by default `1`.

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"]);
```



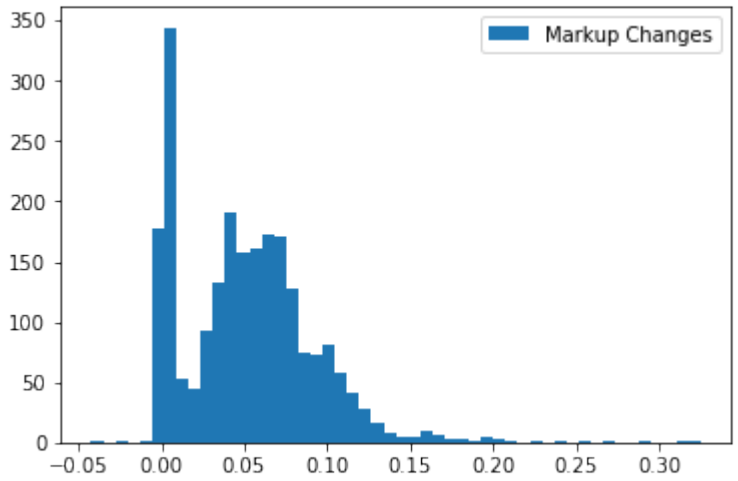
Markups, $\mathscr{M}$, and profits, π, generally increase as well.

```
changed_markups = results.compute_markups(changed_prices, costs)
plt.hist(changed_markups - markups, bins=50);
```

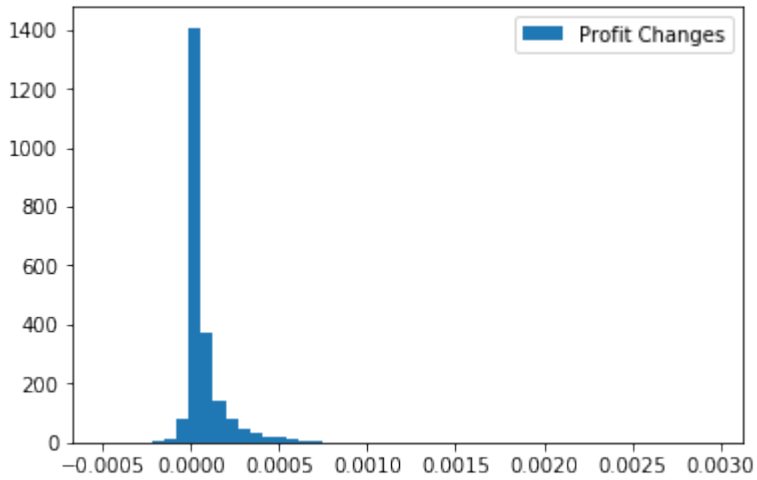
(continues on next page)

(continued from previous page)

```
plt.legend(["Markup Changes"]);
```

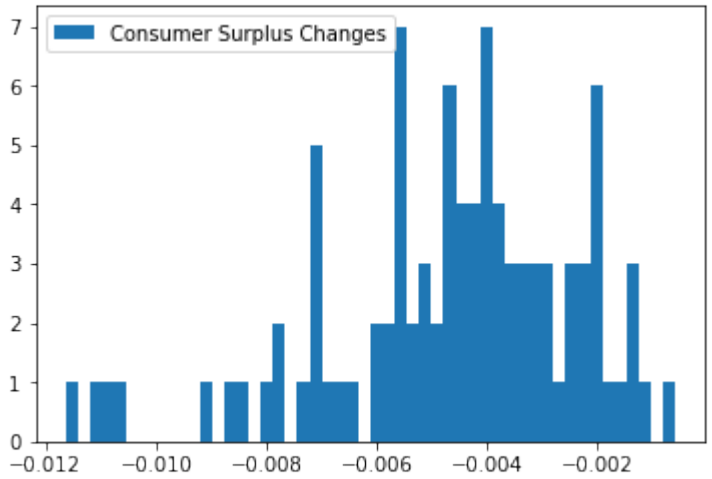



```
changed_profits = results.compute_profits(changed_prices, changed_shares, costs)
plt.hist(changed_profits - profits, bins=50);
plt.legend(["Profit Changes"]);
```



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"]);
```



4.4.5 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 prohibatively 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 Problem Results Summary:
=====================
Computation  Bootstrap
   Time        Draws
-----------  ---------
```

(continues on next page)

(continued from previous page)

```
 00:00:24       100
======================
```

```
bounds = np.percentile(
    bootstrapped_results.extract_diagonal_means(
        bootstrapped_results.compute_elasticities()
    ),
    q=[10, 90],
    axis=0
)
table = pd.DataFrame(index=problem.unique_market_ids, data={
    'Lower Bound': bounds[0].flatten(),
    'Mean Own Elasticity': aggregates.flatten(),
    'Upper Bound': bounds[1].flatten()
})
table.round(2).head()
```

```
       Lower Bound  Mean Own Elasticity  Upper Bound
C01Q1       -14.06                -0.81        10.66
C01Q2        -8.97                -0.69         7.54
C03Q1       -11.64                -0.55         7.98
C03Q2       -14.46                -0.60         6.47
C04Q1       -12.00                -0.68         5.61
```

4.4.6 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:
=======================
```

(continues on next page)

(continued from previous page)

```
Computation  Error Term
   Time        Draws
-----------  ----------
 00:00:01        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'),
    method='1s'
)
updated_results
```

```
Problem Results Summary:
=============================================================================================================================
                                                                                                          Smallest    Largest
Computation  GMM   Optimization   Objective   Fixed Point  Contraction  Objective    Gradient           Hessian     Hessian
```

(continues on next page)

(continued from previous page)

```
   Time      Step   Iterations   Evaluations  Iterations   Evaluations    Value    Infinity Norm  Eigenvalue  Eigenvalue
----------  ----  ------------  -----------  -----------  -----------  ---------  -------------  ----------  ----------
 00:01:56     1         45           53          47406       146847     +5.7E+00     +2.1E-06      +3.0E-12    +1.5E+04
=============================================================================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=============================================================================================================================
Sigma:      1         prices      sugar       mushy      |   Pi:      income    income_squared     age         child
------  ----------  ----------  ----------  ----------  |  ------  ----------  --------------  ----------  ----------
  1      +2.2E-01    +0.0E+00    +0.0E+00    +0.0E+00   |    1      +6.2E+00      +0.0E+00      +1.0E-01    +0.0E+00
        (+8.1E-02)                                      |          (+5.4E-01)                   (+2.2E-01)
                                                        |
prices                +3.1E+00    +0.0E+00    +0.0E+00   |  prices   +4.5E+01      -2.8E+00      +0.0E+00    +3.3E+00
                     (+7.0E-01)                         |          (+9.3E+01)    (+4.9E+00)                 (+2.4E+00)
                                                        |
sugar                             -5.8E-03    +0.0E+00   |  sugar    -2.8E-01      +0.0E+00      +3.7E-02    +0.0E+00
                                    (NA)                |          (+3.6E-02)                   (+1.6E-02)
                                                        |
mushy                                         +9.5E-01   |  mushy    +5.1E-01      +0.0E+00      -1.8E-01    +0.0E+00
                                             (+3.0E-01)  |          (+2.7E-01)                   (+2.3E-01)
=============================================================================================================================

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
----------
 -2.9E+01
(+4.5E+00)
==========
```

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

4.5 Problem Simulation Tutorial

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

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

```
'0.7.0'
```

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 unobserved product characteristics 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.
```

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    MD    MS
---  ----  ---  ---  ----  ----  ----  ----  ----
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: 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 all *Simulation.product_data* except for prices and shares, which are initialized as zeros and need to be solved for.

The excluded instruments in *Simulation.product_data* include basic instruments computed with *build_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`.

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

Since at this stage prices and shares are all zeros, we still need to solve the simulation with *Simulation.solve*. This method computes synthetic prices and shares. Just like *ProblemResults.compute_prices*, to do so it iterates over the ζ-markup equation from *Morrow and Skerlos (2011)*.

```
simulation_results = simulation.solve()
simulation_results
```

```
Simulation Results Summary:
====================================
Computation  Fixed Point  Contraction
   Time      Iterations   Evaluations
-----------  -----------  -----------
 00:00:01        697          697
====================================
```

Now, we can try to recover the true parameters by creating and solving a *Problem*. By default, the convenience method *SimulationResults.to_problem* uses the same formulations and unobserved agent data as the simulation, so estimation is relatively easy. However, 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).

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

```
Dimensions:
================================================
 T    N     F    I    K1    K2    K3    MD    MS
---  ----  ---  ---  ----  ----  ----  ----  ----
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: Cost Characteristics       x      z
================================================
```

```
results = problem.solve(
   sigma=0.5 * simulation.sigma,
   pi=0.5 * simulation.pi,
   beta=[None, 0.5 * simulation.beta[1], None]
)
results
```

```
Problem Results Summary:
=========================================================================================================================
                                                                                                  Smallest    Largest
Computation  GMM   Optimization   Objective   Fixed Point  Contraction  Objective    Gradient      Hessian     Hessian
   Time      Step   Iterations   Evaluations  Iterations   Evaluations    Value    Infinity Norm  Eigenvalue  Eigenvalue
-----------  ----  ------------  -----------  -----------  -----------  ---------  -------------  ----------  ----------
 00:00:42     2         27           33          8318         26498     +9.3E+03     +2.2E-02      +9.5E+03    +4.9E+06
=========================================================================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
==================
Sigma:      x
------  ----------
  x      +1.5E+00
        (+4.5E-01)
```

(continues on next page)

(continued from previous page)

```
==================

Beta Estimates (Robust SEs in Parentheses):
==================================
    1         prices        x
----------  ----------  ----------
 +1.1E+00    -2.0E+00    +1.8E+00
(+1.2E-01)  (+2.3E-02)  (+1.7E-01)
==================================

Gamma Estimates (Robust SEs in Parentheses):
======================
    x           z
----------  ----------
 +9.3E-01    +4.2E+00
(+8.5E-02)  (+7.4E-02)
======================
```

The parameters seem to have been estimated reasonably well.

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

```
array([[ 1.        ,  1.05244587],
       [-2.        , -1.96850864],
       [ 2.        ,  1.84086991]])
```

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

```
array([[1.        , 0.93276371],
       [4.        , 4.17298671]])
```

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

```
array([[1.        , 1.48688669]])
```

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, absorb_method])	Configuration for designing matrices and absorbing fixed effects.
Integration(specification, size[, seed])	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*)

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`. Fixed effect absorption is only supported for some matrices. Unlike `formula`, intercepts are ignored. Only categorical variables are supported.
- **`absorb_method`** (*str or Iteration, optional*) – The method with which fixed effects will be absorbed. One of the following:
  - `'simple'` (default for one fixed effect) - Use simple de-meaning. This method is very unlikely to fully absorb more than one fixed effect.
  - `'memory'` (default for two fixed effects) - Use the *Somaini and Wolak (2016)* algorithm, which only works for two-way fixed effects, and which requires inversion of a dense matrix with dimensions equal to the smaller number of fixed effect groups.
  - `'speed'` - Use the same *Somaini and Wolak (2016)* algorithm but pre-compute the A matrix, which is a dense matrix with dimensions equal to the larger number of fixed effect groups. Again, this method only works for two-way fixed effects.
  - `Iteration` (default for more than two fixed effects) - Use the method of alternating projections described, for example, in *Guimarães and Portugal (2010)*. By default, `Iteration('simple', {'atol': 1e-12})` is used to iteratively de-mean the matrix within each fixed effect level until convergence. This method is equivalent to `'simple'` for one fixed effect, and it will also work for two fixed effects, although either variant of the *Somaini and Wolak (2016)* algorithm is often more performant.

Examples

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

Formulation Example

```
import pyblp

pyblp.__version__
```

```
'0.7.0'
```

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 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*, *seed=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`.
  - `'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'` appears to scale the best. For more information, see *Judd and Skrainka (2011)* and *Conlon and Gortmaker (2019)*.

  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'`, and the level of the quadrature rule otherwise.
- **seed** (*int, optional*) – Passed to `numpy.random.RandomState` when `specification` is `'monte_carlo'` to seed the random number generator before building nodes. By default, a seed is not passed to the random number generator.

Examples

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

Integration Example

```
import pyblp

pyblp.__version__
```

```
'0.7.0'
```

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, seed=0)
integration
```

```
Configured to construct nodes and weights with Monte Carlo simulation.
```

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.
```

5.1.3 pyblp.Iteration

class pyblp.**Iteration**(*method*, *method_options=None*, *compute_jacobian=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. If there are any errors during the acceleration step, it uses the last values for the next iteration of the algorithm.
  - `'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, which must be `False` if `method` does not use analytic Jacobians. By default, analytic Jacobians are not computed, and if a `method` is selected that supports analytic Jacobians, they will by default be numerically approximated.

Examples

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

Iteration Example

```
import pyblp
import numpy as np

pyblp.__version__
```

```
'0.7.0'
```

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.
↪000000000E-14, rtol: 0, max_evaluations: 5000, norm: numpy.linalg.norm, scheme: 1, step_min: +1.000000000E+00, step_
↪max: +1.000000000E+00, step_factor: +4.000000000E+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)

where `gradient` is `None` if `compute_gradient is ``False`.

- **`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()`. 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.

- **`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__
```

```
'0.7.0'
```

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.
↪000000000E-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 {}.
```

5.2 Data Construction Functions

There are also a number of convenience functions that can be used to construct common components of product and agent data.

build_matrix(formulation, data)	Construct a matrix according to a formulation.
build_blp_instruments(formulation, product_data)	Construct traditional excluded BLP instruments.
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.

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`.
- **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__
```

```
'0.7.0'
```

In this example, we'll load the fake cereal data from *Nevo (2000)* and create a simple matrix involving a constant, prices, and shares.

```
formulation = pyblp.Formulation(f'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  \
0  0.072088      2      1          ...                       2.116358
1  0.114178     18      1          ...                      -7.374091
2  0.132391      4      1          ...                       2.187872
3  0.130344      3      0          ...                       2.704576
4  0.154823     12      0          ...                       1.261242

   demand_instruments11  demand_instruments12  demand_instruments13  \
0             -0.154708             -0.005796              0.014538
1             -0.576412              0.012991              0.076143
```

(continues on next page)

(continued from previous page)

```
2              -0.207346               0.003509               0.091781
3               0.040748              -0.003724               0.094732
4               0.034836              -0.000568               0.102451

   demand_instruments14  demand_instruments15  demand_instruments16  \
0              0.126244              0.067345              0.068423
1              0.029736              0.087867              0.110501
2              0.163773              0.111881              0.108226
3              0.135274              0.088090              0.101767
4              0.130640              0.084818              0.101075

   demand_instruments17  demand_instruments18  demand_instruments19
0              0.034800              0.126346              0.035484
1              0.087784              0.049872              0.072579
2              0.086439              0.122347              0.101842
3              0.101777              0.110741              0.104332
4              0.125169              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]])
```

5.2.2 pyblp.build_blp_instruments

pyblp.**build_blp_instruments**(*formulation*, *product_data*)

Construct traditional excluded BLP instruments.

Traditional excluded BLP instruments are

$$Z^{BLP}(X) = [Z^{BLP,Other}(X), Z^{BLP,Rival}(X)], \tag{5.1}$$

in which X is a matrix of product characteristics, $Z^{BLP,Other}(X)$ is a second matrix that consists of sums over characteristics of non-rival goods, and $Z^{BLP,Rival}(X)$ 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, $j \in \mathscr{J}_{ft}$. Then,

$$\begin{aligned} Z_{jt}^{BLP,Other}(X) &= \sum_{k \in \mathscr{J}_{ft} \setminus \{j\}} x_{kt}, \\ Z_{jt}^{BLP,Rival}(X) &= \sum_{k \notin \mathscr{J}_{ft}} x_{kt}. \end{aligned} \tag{5.2}$$

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 excluded BLP instruments, $Z^{BLP}(X)$.

Return type *ndarray*

Examples

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

Building BLP Instruments Example

```
import pyblp
import pandas as pd

pyblp.__version__
```

```
'0.7.0'
```

In this example, we'll load the automobile product data from *Berry, Levinsohn, and Pakes (1995)* and build some very simple excluded demand-side instruments for the problem. These instruments are different from the pre-built ones included in the automobile product data file.

```
formulation = pyblp.Formulation('1 + hpwt + air + mpd + space')
formulation
```

```
1 + hpwt + air + mpd + space
```

```
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         ...           demand_instruments2  \
0  0.528997    0  1.888146         ...                      0.566217
1  0.494324    0  1.935989         ...                      0.566217
2  0.467613    0  1.716799         ...                      0.566217
3  0.426540    0  1.687871         ...                      0.566217
4  0.452489    0  1.504286         ...                      0.566217

   demand_instruments3  demand_instruments4  demand_instruments5  \
0             0.365328             0.659480             0.141017
```

(continues on next page)

(continued from previous page)

```
1              0.290959             0.173552             0.128205
2              0.599771            -0.546387             0.002634
3              0.620544            -1.122968             0.089023
4              0.877198            -1.258575            -0.153840

   supply_instruments0  supply_instruments1  supply_instruments2  \
0            -0.011161             1.478879            -0.546875
1            -0.079317             1.088327            -0.546875
2             0.021034             0.609213            -0.546875
3            -0.090014             0.207461            -0.546875
4             0.038013             0.385211            -0.546875

   supply_instruments3  supply_instruments4  supply_instruments5
0            -0.163302            -0.833091             0.301411
1            -0.095609            -0.390314             0.289947
2            -0.449818             0.400461             0.434632
3            -0.454159             0.934641             0.331099
4            -0.728959             1.146654             0.520555

[5 rows x 25 columns]
```

```
instruments = pyblp.build_blp_instruments(formulation, product_data)
instruments.shape
```

```
(2217, 10)
```

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 (2017)* are

$$Z^{Diff}(X) = [Z^{Diff,Other}(X), Z^{Diff,Rival}(X)], \tag{5.3}$$

in which X is a matrix of product characteristics, $Z^{Diff,Other}(X)$ is a second matrix that consists of sums over functions of differences between non-rival goods, and $Z^{Diff,Rival}(X)$ 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, $j \in \mathscr{J}_{ft}$. Then in the "local" version of $Z^{Diff}(X)$,

$$\begin{aligned} Z^{Diff,Other,Local}_{jt\ell}(X) &= \sum_{k \in \mathscr{J}_{ft} \backslash \{j\}} 1(|d_{jkt\ell}| < \text{SD}_\ell), \\ Z^{Diff,Rival,Local}_{jt\ell}(X) &= \sum_{k \notin \mathscr{J}_{ft}} 1(|d_{jkt\ell}| < \text{SD}_\ell), \end{aligned} \tag{5.4}$$

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 $Z^{Diff}(X)$, which uses a more continuous measure of the distance between goods,

$$\begin{aligned} Z^{Diff,Other,Quad}_{jtk}(X) &= \sum_{k \in \mathscr{J}_{ft} \backslash \{j\}} d^2_{jkt\ell}, \\ Z^{Diff,Rival,Quad}_{jtk}(X) &= \sum_{k \notin \mathscr{J}_{ft}} d^2_{jkt\ell}. \end{aligned} \tag{5.5}$$

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, $Z^{Diff}(X)$.

Return type *ndarray*

Examples

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

Building Differentiation Instruments Example

```
import pyblp
import pandas as pd

pyblp.__version__
```

```
'0.7.0'
```

In this example, we'll load the automobile product data from *Berry, Levinsohn, and Pakes (1995)* and build some very simple excluded demand-side instruments for the problem in the spirit of *Gandhi and Houde (2017)*.

```
formulation = pyblp.Formulation('0 + hpwt + air + mpd + space')
formulation
```

```
hpwt + air + mpd + space
```

```
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         ...           demand_instruments2  \
0  0.528997    0  1.888146         ...                      0.566217
1  0.494324    0  1.935989         ...                      0.566217
2  0.467613    0  1.716799         ...                      0.566217
3  0.426540    0  1.687871         ...                      0.566217
4  0.452489    0  1.504286         ...                      0.566217

   demand_instruments3  demand_instruments4  demand_instruments5  \
0             0.365328             0.659480             0.141017
1             0.290959             0.173552             0.128205
```

(continues on next page)

(continued from previous page)

```
2              0.599771            -0.546387              0.002634
3              0.620544            -1.122968              0.089023
4              0.877198            -1.258575             -0.153840

   supply_instruments0  supply_instruments1  supply_instruments2  \
0            -0.011161             1.478879            -0.546875
1            -0.079317             1.088327            -0.546875
2             0.021034             0.609213            -0.546875
3            -0.090014             0.207461            -0.546875
4             0.038013             0.385211            -0.546875

   supply_instruments3  supply_instruments4  supply_instruments5
0            -0.163302            -0.833091             0.301411
1            -0.095609            -0.390314             0.289947
2            -0.449818             0.400461             0.434632
3            -0.454159             0.934641             0.331099
4            -0.728959             1.146654             0.520555

[5 rows x 25 columns]
```

Note that we're excluding the constant column because it yields collinear constant columns of differentiation instruments.

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.

```
local_instruments = pyblp.build_differentiation_instruments(
    formulation,
    product_data
)
local_instruments.shape
```

```
(2217, 8)
```

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'
```

(continues on next page)

(continued from previous page)

```
)
quadratic_instruments.shape
```

```
(2217, 8)
```

We could also use `interact=True` to include interaction terms in either version of instruments, which would help capture covariances between different product characteristics.

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__
```

```
'0.7.0'
```

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 matrices are defined by their cooperation matrix counterparts, κ. For each market t, $O_{jk} = \kappa_{fg}$ where $j \in \mathscr{J}_{ft}$, the set of products produced by firm f in the market, and similarly, $g \in \mathscr{J}_{gt}$.

Parameters

- **product_data** (*structured array-like*) – Each row corresponds to a product. Markets can have differing numbers of products. The following fields are required:
  - **market_ids** : (*object*) - IDs that associate products with markets.
  - **firm_ids** : (*object*) - IDs that associate products with firms.
- **kappa_specification** (*callable, optional*) – A function that specifies each market's cooperation matrix, κ. The function is of the following form:

```
kappa(f, g) -> value
```

  where `value` is O_{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 O_{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`.

Returns Stacked $J_t \times J_t$ ownership matrices, O, 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__
```

```
'0.7.0'
```

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
```

Finally, we'll 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. ]])
```

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 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 unobserved 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 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__
```

```
'0.7.0'
```

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.
```

Usually, this configuration should be passed directly to *Problem*, which will create a sparse grid of dimension K_2, the number of 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.

5.3 Simulation Class

In addition to reading from data files, data can be simulated by initializing the following class.

Simulation(product_formulations, beta, ...)	Simulation of synthetic data from BLP-type models.

5.3.1 pyblp.Simulation

class pyblp.**Simulation**(*product_formulations*, *beta*, *sigma*, *gamma*, *product_data*, *agent_formulation=None*, *pi=None*, *agent_data=None*, *integration=None*, *rho=None*, *xi=None*, *omega=None*, *xi_variance=1*, *omega_variance=1*, *correlation=0.9*, *costs_type='linear'*, *seed=None*)

Simulation of synthetic data from BLP-type models.

All data are either loaded or simulated during initialization, except for synthetic prices and shares, which are computed by *Simulation.solve()*.

Unspecified exogenous variables that are used to formulate product characteristics in X_1, X_2, and X_3, as well as agent demographics, d, are all drawn independently from the standard uniform distribution.

Unobserved demand- and supply-side product characteristics, ξ and ω, are 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.

Next, traditional excluded BLP instruments are constructed. Demand-side instruments are BLP instruments constructed by *build_blp_instruments()* from X_1^x, along with any supply shifters (variables in X_3 but not X_1. Supply side instruments are BLP instruments constructed from X_3, along with any demand shifters (variables in X_1 but not X_3). BLP instruments for constant characteristics are constructed only if there is variation in J_t, the number of products per market.

Note: These excluded instruments are constructed only for convenience. Especially for more complicated formulations, they should be replaced with better instruments. For example, instruments constructed with *build_differentiation_instruments()* may be preferable.

Parameters

- **product_formulations** (*tuple*) – Tuple of three *Formulation* configurations for the matrix of linear product characteristics, X_1, for the matrix of nonlinear product characteristics, X_2, and for the matrix of cost characteristics, X_3, respectively. If the formulation for X_2 is `None`, the logit (or nested logit) model will be simulated.

  The `shares` variable should not be included in any of the formulations and `prices` should be included in the formulation for X_1 or X_2 (or both). All exogenous characteristics in X_2 should also be included in X_1. Any additional 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.

- **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*) – Cholesky root of the covariance matrix for unobserved taste heterogeneity, Σ, which is an upper triangular matrix. Rows and columns correspond to columns

in X_2, which is formulated by `product_formulations`. If the formulation for X_2 is `None`, this should be `None` as well.

- **`gamma`** (*array-like*) – Vector of supply-side linear parameters, γ. Elements correspond to columns in X_3, which is formulated by `product_formulations`.

- **`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 :math:'J_t times J_t* ownership matrices, O, 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: If `ownership` 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 ownership information, 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.

  Along with `market_ids`, `firm_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`*.

- **`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.

- **`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`.

- **`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 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 nonlinear product characteristics), only the first K_2 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`.

Along with `market_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*.

- **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 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.

- **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 were not specified, this should not be specified either.

- **xi** (*array-like, optional*) – Unobserved demand-side product characteristics, ξ. By default, each pair of unobserved characteristics in this and ω is drawn from a mean-zero bivariate normal distribution. This must be specified if `omega` is specified.

- **omega** (*array-like, optional*) – Unobserved supply-side product characteristics, ω. By default, each pair of unobserved characteristics in this and ξ is drawn from a mean-zero bivariate normal distribution. This must be specified if `xi` is specified.

- **xi_variance** (*float, optional*) – Variance of ξ. The default value is `1.0`. This is ignored if `xi` and `omega` are specified.

- **omega_variance** (*float, optional*) – Variance of ω. The default value is `1.0`. This is ignored if `xi` and `omega` are specified.

- **correlation** (*float, optional*) – Correlation between ξ and ω. The default value is `0.9`. This is ignored if `xi` and `omega` are specified.

- **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, except for synthetic prices and shares, which are computed by *Simulation.solve()*.

Type *recarray*

agent_data
Synthetic agent data that were loaded or simulated during initialization.

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*.

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*.

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*

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*

xi
Unobserved demand-side product characteristics, ξ.

Type *ndarray*

omega
Unobserved supply-side product characteristics, ω.

Type *ndarray*

costs
Marginal costs, c, which was constructed during initialization.

Type *ndarray*

costs_type
The specification according to which `Simulation.costs` was constructed during initialization.

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 linear product characteristics, K_1.

Type *int*

K2
Number of nonlinear product characteristics, K_2.

Type *int*

K3
Number of cost product characteristics, K_3.

Type *int*

D
Number of demographic variables, D.

Type *int*

MD
Number of demand-side instruments, M_D, which is the number of excluded demand-side instruments plus the number of exogenous linear product characteristics, K_1^x.

Type *int*

MS
Number of supply-side instruments, M_S, which is the number of excluded supply-side instruments plus the number of cost product characteristics, K_3.

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

solve([firm_ids, ownership, prices, ...])	Compute synthetic prices and shares.

Once initialized, the following method computes equilibrium prices and shares.

Simulation.solve([firm_ids, ownership, ...])	Compute synthetic prices and shares.

5.3.2 pyblp.Simulation.solve

Simulation.**solve**(*firm_ids=None*, *ownership=None*, *prices=None*, *iteration=None*, *error_behavior='raise'*)
Compute synthetic prices and shares.

Prices and shares are computed in each market by iterating over the ζ-markup contraction in (3.42):

$$p \leftarrow c + \zeta(p). \tag{5.6}$$

Note: To create a simulation under perfect (instead of Bertrand) competition, use an *Iteration* configuration with `method='return'`. This will set prices equal to the default starting values for the iteration routine, which 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

- **firm_ids** (*array-like, optional*) – Potentially changed firms IDs. By default, the `firm_ids` field of `product_data` in *Simulation* will be used.
- **ownership** (*array-like, optional*) – Custom ownership matrices. By default, standard ownership matrices based on `firm_ids` will be used unless the `ownership` field of `product_data` in *Simulation* was specified.
- **prices** (*array-like, optional*) – Prices at which the fixed point iteration routine will start. By default, marginal costs, c, 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. Analytic Jacobians are not supported for solving this system.
- **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*

5.4 Simulation Results Class

Solved simulations return the following results class.

SimulationResults	Results of a solved simulation of synthetic BLP data.

5.4.1 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 and configured information into a *Problem*.

simulation

Simulation that created these results.

Type *Simulation*

product_data

Simulated *Simulation.product_data* that are updated with synthetic prices and shares.

Type *recarray*

delta

Simulated mean utility, δ.

Type *ndarray*

computation_time

Number of seconds it took to compute synthetic prices and shares.

Type *float*

fp_converged

Flags for convergence of the iteration routine used to compute synthetic prices 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 synthetic prices 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 synthetic prices was evaluated in each market. Counts are in the same order as *Simulation.unique_market_ids*.

Type *ndarray*

Examples

- *Tutorial*

Methods

to_problem([product_formulations, . . .])	Convert the solved simulation into a problem.

The simulation results can be converted into a *Problem* with the following method.

SimulationResults.to_problem([...])	Convert the solved simulation into a problem.

5.4.2 pyblp.SimulationResults.to_problem

SimulationResults.**to_problem**(*product_formulations=None*, *product_data=None*, *agent_formulation=None*, *agent_data=None*, *integration=None*)

Convert the solved simulation into a problem.

Parameters are the same as those of *Problem*. By default, the structure of the problem will be the same as that of the solved simulation.

Parameters

- **product_formulations** (*Formulation or tuple of Formulation, optional*) – By default, *Simulation.product_formulations*.
- **product_data** (*structured array-like, optional*) – By default, *SimulationResults.product_data*.
- **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.

Returns A BLP problem.

Return type *Problem*

Examples

- *Tutorial*

5.5 Problem Class

Given real or simulated data and appropriate configurations, the BLP problem can be structured by initializing the following class.

Problem(product_formulations, product_data)	A BLP-type problem.

5.5.1 pyblp.Problem

class pyblp.**Problem**(*product_formulations*, *product_data*, *agent_formulation=None*, *agent_data=None*, *integration=None*)

A BLP-type problem.

This class is initialized with relevant data and solved with *Problem.solve()*.

Parameters

- **product_formulations** (*Formulation or tuple of Formulation*) – *Formulation* configuration or tuple of up to three *Formulation* configurations for the matrix of linear product characteristics, X_1, for the matrix of nonlinear product characteristics, X_2, and for

the matrix of cost 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 any of the formulations and `prices` 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.

Characteristics in X_1 that do not involve `prices`, X_1^x, 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 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.

> **Warning:** Characteristics that involve prices, p, should always be formulated with the `prices` variable. If another name is used, *`Problem`* will not understand that the characteristic is endogenous, so it will be erroneously included in Z_D, and derivatives computed with respect to prices will likely be wrong. For example, to include a p^2 characteristic, include `I(prices**2)` in a formula instead of manually including a `prices_squared` variable in `product_data` and a formula.

- **`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*) - Marketshares, 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 should generally be specified with the following fields:

  - **demand_instruments** : (*numeric*) - Excluded demand-side instruments, which, together with the formulated exogenous linear product characteristics, X_1^x, constitute the full set of demand-side instruments, Z_D.
  - **supply_instruments** : (*numeric, optional*) - Excluded supply-side instruments, which, together with the formulated cost characteristics, X_3, constitute the full set of supply-side instruments, Z_S.

  The recommendation in *Conlon and Gortmaker (2019)* is to start with differentiation instruments of *Gandhi and Houde (2017)*, which can be built with *`build_differentiation_instruments()`*, and then compute feasible optimal instruments with *`ProblemResults.compute_optimal_instruments()`* in the second stage.

  If `firm_ids` are specified, custom ownership matrices can be specified as well:

– **ownership** : (*numeric, optional*) - Custom stacked $J_t \times J_t$ ownership matrices, O, 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.

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`, `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 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`.
- **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 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 nonlinear product characteristics), only the first K_2 will be retained.

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`.

Along with `market_ids`, the names of any additional fields can be typically be used as variables in `agent_formulation`. The exception is the name `'demographics'`, which is reserved for use by *Agents*.

- **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 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.

product_formulations
Formulation configurations for X_1, X_2, and X_3, respectively.

Type *Formulation or tuple 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*.

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*.

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*

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 linear product characteristics, K_1.

Type *int*

K2

Number of nonlinear product characteristics, K_2.

Type *int*

K3

Number of cost product characteristics, K_3.

Type *int*

D

Number of demographic variables, D.

Type *int*

MD

Number of demand-side instruments, M_D, which is the number of excluded demand-side instruments plus the number of exogenous linear product characteristics, K_1^x.

Type *int*

MS

Number of supply-side instruments, M_S, which is the number of excluded supply-side instruments plus the number of cost product characteristics, K_3.

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, beta, gamma, ...])	Solve the problem.

Once initialized, the following method solves the problem.

Problem.solve([sigma, pi, rho, beta, gamma, ...])	Solve the problem.

5.5.2 pyblp.Problem.solve

Problem.**solve**(*sigma=None*, *pi=None*, *rho=None*, *beta=None*, *gamma=None*, *sigma_bounds=None*, *pi_bounds=None*, *rho_bounds=None*, *beta_bounds=None*, *gamma_bounds=None*, *delta=None*, *W=None*, *method='2s'*, *optimization=None*, *check_optimality='both'*, *error_behavior='revert'*, *error_punishment=1*, *delta_behavior='first'*, *iteration=None*, *fp_type='safe_linear'*, *costs_type='linear'*, *costs_bounds=None*, *center_moments=True*, *W_type='robust'*, *se_type='robust'*)

Solve the problem.

The problem is solved in one or more GMM steps. During each step, any parameters in $\hat{\theta}$ are optimized to minimize the GMM objective value. If there are no parameters in $\hat{\theta}$ (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, $\delta(\hat{\theta})$ 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, $c(\hat{\theta})$, 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 $\delta(\hat{\theta})$ (and $\tilde{c}(\hat{\theta})$ 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 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 below 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.

- **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.

- **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.

By default, if bounds are supported, the diagonal of `sigma` is bounded from below by zero. Conditional on X_2, μ, and an initial estimate of μ, default bounds for off-diagonal parameters are chosen to reduce the need for overflow safety precautions.

Values below 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, if bounds are supported, conditional on X_2, d, and an initial estimate of μ, default bounds are chosen to reduce the need for overflow safety precautions.

  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.

  By default, if bounds are supported, all elements are bounded from below by `0`, which corresponds to the simple logit model. Conditional on an initial estimate of μ, upper bounds are chosen to reduce the need for overflow safety precautions, and are less than `1` because larger values 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`.

- **`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 (3.36) is used. If ρ is specified, the solution to the nested logit model in (3.37) under the initial rho is used instead.
- **W** (*array-like, optional*) – Starting values for the weighting matrix, W. By default, the 2SLS weighting matrix in (3.23) is used.
- **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.
- **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') 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. Routines that do not support bounds will ignore sigma_bounds and pi_bounds. Choosing a routine that does not use analytic gradients will often down estimation.
- **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. Specifically, analytically compute the gradient $2P$ times, perturbing each of the P parameters by $\pm\epsilon/2$ where ϵ is the square root of the machine precision.
- **error_behavior** (*str, optional*) – How to handle any errors. For example, there can sometimes be overflow or underflow when computing $\delta(\hat{\theta})$ at a large $\hat{\theta}$. The following behaviors are supported:
  - 'revert' (default) - Revert problematic $\delta(\hat{\theta})$ elements to their last computed values and use reverted values to compute $\frac{\partial \xi}{\partial \theta}$, and, if there is a supply side, to compute both $\tilde{c}(\hat{\theta})$ and $\frac{\partial \omega}{\partial \theta}$ as well. If there are problematic elements in $\frac{\partial \xi}{\partial \theta}$, $\tilde{c}(\hat{\theta})$, or $\frac{\partial \omega}{\partial \theta}$, revert these to their last computed values as well. If there are problematic elements after the first objective evaluation, revert values in $\delta(\hat{\theta})$ to their starting values; in $\tilde{c}(\hat{\theta})$, to prices; and in

Jacobians, to zeros. In the unlikely event that the gradient or its objective have problematic elements, revert them as well, and if this happens during the first objective evaluation, revert the objective to `1e10` and its gradient 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 $\delta(\hat{\theta})$ 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.
  - `'last'` - Start at the values of $\delta(\hat{\theta})$ 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 $\delta(\hat{\theta})$ 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 non-Jacobian 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 $\delta(\hat{\theta})$. The following types are supported:
  - `'safe_linear'` (default) - The standard linear contraction mapping in (3.13) (or (3.35) when there is nesting) with safeguards against numerical overflow. Specifically, $\max_j V_{jti}$ (or $\max_j V_{jti}/(1-\rho_{h(j)})when there is nesting) is subtracted from :math: and the logit expression for choice probabilities in (3.5) (or (3.33)) 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 $s_{jt}(\delta, \hat{\theta})$. This second trick only works when there are no nesting parameters.
  - `'safe_nonlinear'` - Exponentiated version with minimal safeguards against numerical overflow. Specifically, $\max_j \mu_{jti}$ is subtracted from μ_{jti}. This helps with stability but is less helpful than subtracting from the full V_{jti}, 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 (3.36) and in the nested logit model with (3.37).

- **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: $\tilde{c} = \log c$.

  This specification is only relevant if X_3 was formulated by `product_formulations` in *Problem*.
- **costs_bounds** (*tuple, optional*) – Configuration for c bounds of the form `(lb, ub)`, in which both `lb` and `ub` are floats. This is only relevant if X_3 was formulated by `product_formulations` in *Problem*. By default, marginal costs are unbounded.

  When `costs_type` is `'log'`, nonpositive $c(\hat{\theta})$ values can create problems when computing $\tilde{c}(\hat{\theta}) = \log c(\hat{\theta})$. 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`.
- **center_moments** (*bool, optional*) – Whether to center each column of the sample moments g before updating the weighting matrix W. By default, sample 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'`. Often, `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).
- **se_type** (*str, optional*) – How to compute standard errors. Typically, `W_type` should be the same. The following types are supported:
  - `'robust'` (default) - Heteroscedasticity robust standard errors defined in (3.29) and (3.25).
  - `'clustered'` - Clustered standard errors defined in (3.29) and (3.26). Clusters must be defined by the `clustering_ids` field of `product_data` in *Problem*.
    - `'unadjusted'` - Homoskedastic standard errors defined in (3.30), which are computed under the assumption that the weighting matrix is optimal.

Returns *ProblemResults* of the solved problem.

Return type *ProblemResults*

Examples

- *Tutorial*

5.6 Problem Results Class

Solved problems return the following results class.

ProblemResults	Results of a solved BLP problem.

5.6.1 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: All methods in this class 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*

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 $\delta(\hat{\theta})$ 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 $\delta(\hat{\theta})$ 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 $\delta(\hat{\theta})$ 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*

converged
Whether the optimization routine converged.

Type *bool*

cumulative_converged
Whether the optimization routine converged for this step and all prior steps.

Type *bool*

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 covariance matrix of the stacked parameters, from which standard errors are extracted.

Type *ndarray*

theta
Estimated unfixed parameters, $\hat{\theta}$, in the following order: $\hat{\Sigma}$, $\hat{\Pi}$, $\hat{\rho}$, 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*

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*

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 $\hat{\Sigma}$.

Type *ndarray*

pi_se
Estimated standard errors for $\hat{\Pi}$.

Type *ndarray*

rho_se
Estimated standard errors for $\hat{\rho}$.

Type *ndarray*

beta_se
Estimated standard errors for $\hat{\beta}$.

Type *ndarray*

gamma_se
Estimated standard errors for $\hat{\gamma}$.

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*

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*

delta
Estimated mean utility, $\delta(\hat{\theta})$.

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. When there are demand-side fixed effects, this is $\Delta\xi(\hat{\theta})$ in (3.31). That is, 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. When there are supply-side fixed effects, this is $\Delta\omega(\hat{\theta})$ in (3.31). That is, fixed effects are not included.

Type *ndarray*

objective
GMM objective value, $q(\hat{\theta})$, defined in (3.10).

Type *float*

xi_by_theta_jacobian
Estimated $\frac{\partial \xi}{\partial \theta} = \frac{\partial \delta}{\partial \theta}$.

Type *ndarray*

omega_by_theta_jacobian
Estimated $\frac{\partial \omega}{\partial \theta} = \frac{\partial \tilde{c}}{\partial \theta}$.

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*

sigma_gradient
Estimated gradient of the GMM objective with respect to Σ elements in θ.

Type *ndarray*

pi_gradient
Estimated gradient of the GMM objective with respect to Π elements in θ.

Type *ndarray*

rho_gradient
Estimated gradient of the GMM objective with respect to ρ elements in θ.

Type *ndarray*

beta_gradient
Estimated gradient of the GMM objective with respect to β elements in θ.

Type *ndarray*

gamma_gradient
Estimated gradient of the GMM objective with respect to γ elements in θ.

Type *ndarray*

gradient_norm
Infinity norm of *ProblemResults.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*

hessian_eigenvalues
Eigenvalues of *ProblemResults.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_aggregate_elasticities([factor, name])	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()	Estimate marginal costs, c.
compute_diversion_ratios([name])	Estimate matrices of diversion ratios, $\mathscr{D}$, with respect to a variable, x.
compute_elasticities([name])	Estimate matrices of elasticities of demand, ε, with respect to a variable, x.
compute_hhi([firm_ids, shares])	Estimate Herfindahl-Hirschman Indices, HHI.
compute_long_run_diversion_ratios()	Estimate matrices of long-run diversion ratios, $\mathscr{D}$.
compute_markups([prices, costs])	Estimate markups, $\mathscr{M}$.
compute_optimal_instruments([method, draws, ...])	Estimate feasible optimal or efficient instruments, Z_D^{Opt} and Z_S^{Opt}.
compute_prices([firm_ids, ownership, costs, ...])	Estimate equilibrium prices after firm or cost changes, p^*.
compute_profits([prices, shares, costs])	Estimate population-normalized gross expected profits, π.
compute_shares([prices])	Estimate shares evaluated at specified prices.
extract_diagonal_means(matrices)	Extract means of diagonals from stacked $J_t \times J_t$ matrices for each market t.
extract_diagonals(matrices)	Extract diagonals from stacked $J_t \times J_t$ matrices for each market t.

In addition to class attributes, other post-estimation outputs can be estimated with the following methods, which each return an array.

ProblemResults.compute_aggregate_elasticities([factor, name])	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_diversion_ratios([name])	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, $\mathscr{D}$.
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_costs()	Estimate marginal costs, c.
ProblemResults.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.
ProblemResults.compute_prices([firm_ids, ...])	Estimate equilibrium prices after firm or cost changes, p^*.
ProblemResults.compute_shares([prices])	Estimate shares evaluated at specified prices.

Continued on next page

Table 14 – continued from previous page

ProblemResults.compute_hhi([firm_ids, shares])	Estimate Herfindahl-Hirschman Indices, HHI.
ProblemResults.compute_markups([prices, costs])	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.

5.6.2 pyblp.ProblemResults.compute_aggregate_elasticities

ProblemResults.**compute_aggregate_elasticities**(*factor=0.1*, *name='prices'*)

Estimate aggregate elasticities of demand, $\mathscr{E}$, with respect to a variable, x.

In market t, the aggregate elasticity of demand is

$$\mathscr{E} = \sum_{j=1}^{J_t} \frac{s_{jt}(x + \Delta x) - s_{jt}}{\Delta}, \tag{5.7}$$

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.

Returns Estimates of aggregate elasticities of demand, $\mathscr{E}$, for all markets. Rows are in the same order as *Problem.unique_market_ids*.

Return type *ndarray*

Examples

- *Tutorial*

5.6.3 pyblp.ProblemResults.compute_elasticities

ProblemResults.**compute_elasticities**(*name='prices'*)

Estimate matrices of elasticities of demand, ε, with respect to a variable, x.

For each market, the value in row j and column k of ε is

$$\varepsilon_{jk} = \frac{x_k}{s_j} \frac{\partial s_j}{\partial x_k}. \tag{5.8}$$

Parameters **name** (*str, optional*) – Name of the variable, x. By default, $x = p$, prices.

Returns Stacked $J_t \times J_t$ estimated matrices of elasticities of demand, ε, for each market t. 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*

5.6.4 pyblp.ProblemResults.compute_diversion_ratios

`ProblemResults.`**`compute_diversion_ratios`**(*name='prices'*)

Estimate matrices of diversion ratios, $\mathscr{D}$, with respect to a variable, x.

Diversion ratios to the outside good are reported on diagonals. For each market, the value in row j and column k is

$$\mathscr{D}_{jk} = -\frac{\partial s_{k(j)}}{\partial x_j} \Big/ \frac{\partial s_j}{\partial x_j}, \tag{5.9}$$

in which $s_{k(j)}$ is $s_0 = 1 - \sum_j s_j$ if $j = k$, and is s_k otherwise.

Parameters **`name`** (*str, optional*) – Name of the variable, x. By default, $x = p$, prices.

Returns Stacked $J_t \times J_t$ estimated matrices of diversion ratios, $\mathscr{D}$, for all markets. 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*

5.6.5 pyblp.ProblemResults.compute_long_run_diversion_ratios

`ProblemResults.`**`compute_long_run_diversion_ratios`**()

Estimate matrices of long-run diversion ratios, $\bar{\mathscr{D}}$.

Long-run diversion ratios to the outside good are reported on diagonals. For each market, the value in row j and column k is

$$\bar{\mathscr{D}}_{jk} = \frac{s_{k(-j)} - s_k}{s_j}, \tag{5.10}$$

in which $s_{k(-j)}$ is the share of product k computed with the outside option removed from the choice set if $j = k$, and with product j removed otherwise.

Returns Stacked $J_t \times J_t$ estimated matrices of long-run diversion ratios, $\bar{\mathscr{D}}$, for all markets. 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*

5.6.6 pyblp.ProblemResults.extract_diagonals

ProblemResults.**extract_diagonals**(*matrices*)

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()*; or $\bar{\mathscr{D}}$, computed by *ProblemResults.compute_long_run_diversion_ratios()*.

Returns Stacked diagonals for all markets. 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*

5.6.7 pyblp.ProblemResults.extract_diagonal_means

ProblemResults.**extract_diagonal_means**(*matrices*)

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()*; or $\bar{\mathscr{D}}$, computed by *ProblemResults.compute_long_run_diversion_ratios()*.

Returns Stacked means of diagonals for all markets. 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*

5.6.8 pyblp.ProblemResults.compute_costs

ProblemResults.**compute_costs**()

Estimate marginal costs, c.

Marginal costs are computed with the η-markup equation in (3.7):

$$c = p - \eta. \tag{5.11}$$

Returns Marginal costs, c.

Return type *ndarray*

Examples

- *Tutorial*

5.6.9 pyblp.ProblemResults.compute_approximate_prices

`ProblemResults.`**`compute_approximate_prices`**(*firm_ids=None*, *ownership=None*, *costs=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 discussed in, for example, *Nevo (1997)*. Prices in each market are computed according to the η-markup equation in (3.7):

$$p^* = c^* + \eta^*, \tag{5.12}$$

in which the markup term is approximated with

$$\eta^* \approx -\left(O^* \odot \frac{\partial s}{\partial p}\right)^{-1} s \tag{5.13}$$

where O^* 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, optional*) – Potentially changed marginal costs, c^*. By default, unchanged marginal costs are computed with *`ProblemResults.compute_costs()`*.

Returns Approximation of equilibrium prices after any firm or cost changes, p^*.

Return type *ndarray*

Examples

- *Tutorial*

5.6.10 pyblp.ProblemResults.compute_prices

`ProblemResults.`**`compute_prices`**(*firm_ids=None*, *ownership=None*, *costs=None*, *prices=None*, *iteration=None*)

Estimate equilibrium prices after firm or cost changes, p^*.

Prices are computed in each market by iterating over the ζ-markup contraction in (3.42):

$$p^* \leftarrow c^* + \zeta^*(p^*), \tag{5.14}$$

in which the markup term from (3.39) is

$$\zeta^*(p^*) = \Lambda^{-1}(p^*)[O^* \odot \Gamma(p^*)]'(p^* - c^*) - \Lambda^{-1}(p^*) \tag{5.15}$$

where O^* 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()*.
- **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. Analytic Jacobians are not supported for solving this system.

Returns Estimates of equilibrium prices after any firm or cost changes, p^*.

Return type *ndarray*

Examples

- *Tutorial*

5.6.11 pyblp.ProblemResults.compute_shares

`ProblemResults.`**compute_shares**(*prices=None*)

Estimate shares evaluated at specified prices.

Parameters **prices** (*array-like*) – Prices at which to evaluate shares, such as equilibrium prices, p^*, computed by *ProblemResults.compute_prices()*. By default, unchanged prices are used.

Returns Estimates of shares evaluated at the specified prices.

Return type *ndarray*

Examples

- *Tutorial*

5.6.12 pyblp.ProblemResults.compute_hhi

`ProblemResults.`**compute_hhi**(*firm_ids=None*, *shares=None*)

Estimate Herfindahl-Hirschman Indices, HHI.

The index in market t is

$$\text{HHI} = 10,000 \times \sum_{f=1}^{F_t} \left(\sum_{j \in \mathscr{J}_{ft}} s_j \right)^2. \tag{5.16}$$

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.

Returns Estimated Herfindahl-Hirschman Indices, HHI, for all markets. Rows are in the same order as *Problem.unique_market_ids*.

Return type *ndarray*

Examples

- *Tutorial*

5.6.13 pyblp.ProblemResults.compute_markups

`ProblemResults.`**compute_markups**(*prices=None*, *costs=None*)

Estimate markups, $\mathscr{M}$.

The markup of product j in market t is

$$\mathscr{M}_{jt} = \frac{p_{jt} - c_{jt}}{p_{jt}}. \tag{5.17}$$

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()*.

Returns Estimated markups, $\mathscr{M}$.

Return type *ndarray*

Examples

- *Tutorial*

5.6.14 pyblp.ProblemResults.compute_profits

`ProblemResults.`**compute_profits**(*prices=None*, *shares=None*, *costs=None*)

Estimate population-normalized gross expected profits, π.

The profit from product j in market t is

$$\pi_{jt} = (p_{jt} - c_{jt})s_{jt}. \tag{5.18}$$

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()`.

Returns Estimated population-normalized gross expected profits, π.

Return type *ndarray*

Examples

- *Tutorial*

5.6.15 pyblp.ProblemResults.compute_consumer_surpluses

`ProblemResults.`**`compute_consumer_surpluses`**(*prices=None*)

Estimate population-normalized consumer surpluses, CS.

Assuming away nonlinear income effects, the surplus in market t is

$$\text{CS} = \sum_{i=1}^{I_t} w_i \text{CS}_i, \tag{5.19}$$

in which the consumer surplus for individual i is

$$\text{CS}_i = \log\left(1 + \sum_{j=1}^{J_t} \exp V_{jti}\right) \Big/ \frac{\partial V_{1ti}}{\partial p_{1t}}, \tag{5.20}$$

or with nesting parameters,

$$\text{CS}_i = \log\left(1 + \sum_{h=1}^{H} \exp V_{hti}\right) \Big/ \frac{\partial V_{1ti}}{\partial p_{1t}} \tag{5.21}$$

where V_{jti} is defined in (3.1) and V_{hti} is defined in (3.34).

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)`.

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.

Returns Estimated population-normalized consumer surpluses, CS, for all markets. Rows are in the same order as `Problem.unique_market_ids`.

Return type *ndarray*

Examples

- *Tutorial*

A parametric bootstrap can be used, for example, to compute standard errors for the above post-estimation outputs. The following method returns a results class with all of the above methods, 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.

5.6.16 pyblp.ProblemResults.bootstrap

`ProblemResults.`**`bootstrap`**(*draws=1000*, *seed=None*, *iteration=None*)

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. 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`. 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 (3.42).

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 (3.42). 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.

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_instruments([...])`	Estimate feasible optimal or efficient instruments, Z_D^{Opt} and Z_S^{Opt}.

5.6.17 pyblp.ProblemResults.compute_optimal_instruments

`ProblemResults.`**`compute_optimal_instruments`**(*method='approximate'*, *draws=1*, *seed=None*, *expected_prices=None*, *iteration=None*)

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 (2019)*, 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

$$\begin{bmatrix} Z_{D,jt}^{Opt} \\ Z_{S,jt}^{Opt} \end{bmatrix} = \text{Var}(\xi, \omega)^{-1} E\left[\begin{bmatrix} \frac{\partial \xi_{jt}}{\partial \theta} \\ \frac{\partial \omega_{jt}}{\partial \theta} \end{bmatrix} \middle| Z\right], \tag{5.22}$$

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 (3.42).

The expected Jacobians are estimated with the average over all computed Jacobian realizations. The 2×2 normalizing matrix $\text{Var}(\xi, \omega)$ 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 (5.22) 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()`.

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, `iteration` is used. If only a demand side was estimated, 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 (3.42). 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.

Returns Computed *OptimalInstrumentResults*.

Return type *OptimalInstrumentResults*

Examples

- *Tutorial*

5.7 Boostrapped Problem Results Class

Parametric bootstrap computation returns the following class.

BootstrappedResults	Bootstrapped results of a solved problem.

5.7.1 pyblp.BootstrappedResults

class pyblp.**BootstrappedResults**

Bootstrapped results of a solved problem.

This class has all of the same methods as *ProblemResults* except for *ProblemResults.bootstrap()* and *ProblemResults.compute_optimal_instruments()*. The only other difference is that 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). 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_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 marketshares, s, implied by each draw.

Type *ndarray*

bootstrapped_delta
Bootstrapped mean utility, δ, implied by each draw.

Type *ndarray*

bootstrapped_costs
Bootstrapped marginal costs, c, 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 all of the same methods as *ProblemResults*, except for *ProblemResults.bootstrap()* and *ProblemResults.compute_optimal_instruments()*.

5.8 Optimal Instrument Results Class

Optimal instrument computation returns the following results class.

OptimalInstrumentResults	Results of optimal instrument computation.

5.8.1 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, Z_D^{Opt}.

Type *ndarray*

supply_instruments
Estimated optimal supply-side instruments for θ, 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 from *Problem* excluding any variables in the formulation for X_1.

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^x from *Problem* excluding any variables in the formulation for X_3.

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 $1/\text{Var}(\xi)$.

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*

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_problem`([supply_shifter_formulation, ...])	Re-create the problem with estimated feasible optimal instruments.

The optimal instrument results can be converted into a `Problem` with the following method.

`OptimalInstrumentResults.to_problem`([...])	Re-create the problem with estimated feasible optimal instruments.

5.8.2 pyblp.OptimalInstrumentResults.to_problem

OptimalInstrumentResults.**to_problem**(*supply_shifter_formulation=None*, *demand_shifter_formulation=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 exogenous linear parameters 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^p, 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^p 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 not in X_1.

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 exogenous linear parameters that were not concentrated out, but rather included in θ by `Problem.solve()`.
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^x not in X_3.

As usual, the excluded demand-side instruments will be supplemented with X_1^x and the excluded supply-side instruments will be supplemented with X_3. 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.

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.

5.8.3 pyblp.OptimalInstrumentProblem

class pyblp.**OptimalInstrumentProblem**

A BLP problem updated with optimal excluded instruments.

This class can be used exactly like *Problem*.

5.9 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.

5.9.1 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*

clustering_ids
IDs used to compute clustered standard errors.

Type *ndarray*

ownership
Stacked $J_t \times J_t$ ownership matrices, O, for each market t.

Type *ndarray*

shares
Marketshares, s.

Type *ndarray*

prices
Product prices, p.

Type *ndarray*

ZD
Full set of demand-side instruments, Z_D: excluded demand-side instruments and X_1^x.

Type *ndarray*

ZS
Full set of supply-side instruments, Z_S: excluded supply-side instruments and X_3.

Type *ndarray*

X1
Linear product characteristics, X_1.

Type *ndarray*

X2
Nonlinear product characteristics, X_2.

Type *ndarray*

X3
Cost product characteristics, X_3.

Type *ndarray*

5.9.2 pyblp.Agents

class pyblp.**Agents**
Agent data structured as a record array.

market_ids
IDs that associate agents with 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*

5.10 Multiprocessing

A context manager can be used to enable parallel processing for methods that perform market-by-market computation.

parallel(processes)	Context manager used for parallel processing in a `with` statement context.

5.10.1 pyblp.parallel

pyblp.**parallel**(*processes*)
Context manager used for parallel processing in a `with` statement context.

This manager creates a a context in which a pool of Python processes will be used by any of the following methods, which all support parallel processing:

- *Simulation.solve()*
- *Problem.solve()*
- Any method in *ProblemResults*.

These methods, which perform market-by-market computation, 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 of 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.

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__
```

```
'0.7.0'
```

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 (2000)*.

```
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:
=========================================
Computation  GMM    Objective   Objective
   Time      Step  Evaluations    Value
-----------  ----  -----------  ---------
 00:00:00     2          2       +4.2E+05
=========================================

Beta Estimates (Robust SEs in Parentheses):
==========
  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 00:00:03.

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.

5.11 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.

5.11.1 pyblp.options

Global options.

pyblp.options.**digits**
: Number of digits displayed by status updates. The default number of digits is `10`. The number of digits can be changed to, for example, `20`, with `pyblp.options.digits = 20`.

  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_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.**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. For example, *Skrainka (2012)* finds that using long doubles is sufficient to solve many utility floating point problems.

  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 unreliably 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 `np.finfo(np.float64)` and `np.finfo(np.longdouble)`. For more information, refer to this discussion.

  If extended precisions is supported, the data type can be switched with `pyblp.options.dtype = np.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.**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.

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-14`. 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-14`.

Type *float*

5.11.2 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 (2000)*. The file includes the same pre-computed excluded instruments used in the original paper.

Type *str*

pyblp.data.**NEVO_AGENTS_LOCATION**
Location of a CSV file containing the fake cereal agent data. Included in the file are Monte Carlo weights and draws along with demographics, which are used by *Nevo (2000)* to solve the fake cereal problem.

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 optimal excluded instruments computed in the spirit of *Chamberlain (1987)* for the automobile problem from *Berry, Levinsohn, and Pakes (1995)*, which are used to solve the problem in the tutorial. These instruments were computed according to the following procedure:

1. Traditional excluded BLP instruments from the original paper were computed with *build_blp_instruments()*. As in the original paper, the `mpd` variable was added to the set of excluded supply-side instruments.
2. Each set of excluded instruments was interacted up to the third degree, standardized, replaced with the minimum set of principal components that explained at least 99% of the variance, and standardized again.
3. These two sets of principal components were used as excluded demand- and supply-side instruments when solving the first GMM stage of a *Problem* configured as in the tutorial, but with non-optimal instruments.
4. The *compute_optimal_instruments()* method was used to estimate the optimal excluded instruments for the problem, which were standardized.

Type *str*

pyblp.data.**BLP_AGENTS_LOCATION**
Location of a CSV file containing automobile agent data. Included in the file are 200 Monte Carlo weights and draws for each market, which, unlike in the fake cereal data, are not the same draws used in the original paper.

Also included is an income demographic, which consists of draws from lognormal distributions with common standard deviation `1.72` and the following market-varying means:

Year	Mean
1971	2.01156
1972	2.06526
1973	2.07843
1974	2.05775
1975	2.02915
1976	2.05346
1977	2.06745
1978	2.09805
1979	2.10404
1980	2.07208
1981	2.06019
1982	2.06561
1983	2.07672
1984	2.10437
1985	2.12608
1986	2.16426
1987	2.18071
1988	2.18856
1989	2.21250
1990	2.18377

These numbers were extracted also extracted from the original GAUSS code for *Berry, Levinsohn, and Pakes (1999)*.

Type *str*

Examples

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

Loading Data Example

```
import pyblp

pyblp.__version__
```

```
'0.7.0'
```

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.recfromcsv(pyblp.data.BLP_PRODUCTS_LOCATION, encoding='utf-8')
blp_agent_data = np.recfromcsv(pyblp.data.BLP_AGENTS_LOCATION, 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         ...           demand_instruments2  \
0  0.528997    0  1.888146         ...                      0.566217
1  0.494324    0  1.935989         ...                      0.566217
2  0.467613    0  1.716799         ...                      0.566217
3  0.426540    0  1.687871         ...                      0.566217
```

(continues on next page)

(continued from previous page)

```
4  0.452489    0  1.504286         ...                      0.566217

   demand_instruments3  demand_instruments4  demand_instruments5  \
0             0.365328             0.659480             0.141017
1             0.290959             0.173552             0.128205
2             0.599771            -0.546387             0.002634
3             0.620544            -1.122968             0.089023
4             0.877198            -1.258575            -0.153840

   supply_instruments0  supply_instruments1  supply_instruments2  \
0            -0.011161             1.478879            -0.546875
1            -0.079317             1.088327            -0.546875
2             0.021034             0.609213            -0.546875
3            -0.090014             0.207461            -0.546875
4             0.038013             0.385211            -0.546875

   supply_instruments3  supply_instruments4  supply_instruments5
0            -0.163302            -0.833091             0.301411
1            -0.095609            -0.390314             0.289947
2            -0.449818             0.400461             0.434632
3            -0.454159             0.934641             0.331099
4            -0.728959             1.146654             0.520555

[5 rows x 25 columns]
```

```
blp_agent_data.head()
```

```
   market_ids  weights    nodes0    nodes1    nodes2    nodes3    nodes4  \
0        1971    0.005  0.548814  0.457760  0.564690  0.395537  0.392173
1        1971    0.005  0.715189  0.376918  0.839746  0.844017  0.041157
2        1971    0.005  0.602763  0.702335  0.376884  0.150442  0.923301
3        1971    0.005  0.544883  0.207324  0.499676  0.306309  0.406235
4        1971    0.005  0.423655  0.074280  0.081302  0.094570  0.944282

      income
0   9.728478
1   7.908957
2  11.079404
3  17.641671
4  12.423995
```

5.12 Exceptions

When errors occur, they will either be displayed as warnings or raised as exceptions.

MultipleErrors	Multiple errors that occurred around the same time.
NonpositiveCostsError	Encountered nonpositive marginal costs in a log-linear specification.
InvalidParameterCovariancesError	Failed to compute standard errors because of invalid estimated covariances of GMM parameters.
InvalidMomentCovariancesError	Failed to compute a weighting matrix because of invalid estimated covariances of GMM moments.
DeltaFloatingPointError	Encountered floating point issues when computing δ.
XiByThetaJacobianFloatingPointError	Encountered floating point issues when computing the Jacobian of ξ (equivalently, of δ) with respect to θ.
CostsFloatingPointError	Encountered floating point issues when computing marginal costs.
OmegaByThetaJacobianFloatingPointError	Encountered floating point issues when computing the Jacobian of ω (equivalently, of transformed marginal costs) with respect to θ.
SyntheticPricesFloatingPointError	Encountered floating point issues when computing synthetic prices.
SyntheticSharesFloatingPointError	Encountered floating point issues when computing synthetic shares.
EquilibriumPricesFloatingPointError	Encountered floating point issues when computing equilibrium prices.
EquilibriumSharesFloatingPointError	Encountered floating point issues when computing equilibrium shares.
AbsorptionConvergenceError	An iterative de-meaning procedure failed to converge when absorbing fixed effects.
ThetaConvergenceError	The optimization routine failed to converge.
DeltaConvergenceError	The fixed point computation of δ failed to converge.
SyntheticPricesConvergenceError	The fixed point computation of synthetic prices failed to converge.
EquilibriumPricesConvergenceError	The fixed point computation of equilibrium prices failed to converge.
ObjectiveReversionError	Reverted a problematic GMM objective value.
GradientReversionError	Reverted problematic elements in the GMM objective gradient.
DeltaReversionError	Reverted problematic elements in δ.
CostsReversionError	Reverted problematic marginal costs.
XiByThetaJacobianReversionError	Reverted problematic elements in the Jacobian of ξ (equivalently, of δ) with respect to θ.
OmegaByThetaJacobianReversionError	Reverted problematic elements in the Jacobian of ω (equivalently, of transformed marginal costs) with respect to θ.
AbsorptionInversionError	Failed to invert the A matrix from *Somaini and Wolak (2016)* when absorbing two-way fixed effects.
HessianEigenvaluesError	Failed to compute eigenvalues for the GMM objective's Hessian matrix.
FittedValuesInversionError	Failed to invert an estimated covariance when computing fitted values.

Continued on next page

Table 25 – continued from previous page

SharesByXiJacobianInversionError	Failed to invert a Jacobian of shares with respect to ξ when computing the Jacobian of ξ (equivalently, of δ) with respect to θ.
IntraFirmJacobianInversionError	Failed to invert an intra-firm Jacobian of shares with respect to prices when computing η.
LinearParameterCovariancesInversionError	Failed to invert an estimated covariance matrix of linear parameters.
GMMParameterCovariancesInversionError	Failed to invert an estimated covariance matrix of GMM parameters.
GMMMomentCovariancesInversionError	Failed to invert an estimated covariance matrix of GMM moments.

5.12.1 pyblp.MultipleErrors

class pyblp.**MultipleErrors**
Multiple errors that occurred around the same time.

5.12.2 pyblp.NonpositiveCostsError

class pyblp.**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.

5.12.3 pyblp.InvalidParameterCovariancesError

class pyblp.**InvalidParameterCovariancesError**
Failed to compute standard errors because of invalid estimated covariances of GMM parameters.

5.12.4 pyblp.InvalidMomentCovariancesError

class pyblp.**InvalidMomentCovariancesError**
Failed to compute a weighting matrix because of invalid estimated covariances of GMM moments.

5.12.5 pyblp.DeltaFloatingPointError

class pyblp.**DeltaFloatingPointError**
Encountered floating point issues 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, rescaling data, removing outliers, changing the floating point precision, or using different optimization, iteration, or integration configurations.

5.12.6 pyblp.XiByThetaJacobianFloatingPointError

class pyblp.**XiByThetaJacobianFloatingPointError**
Encountered floating point issues when computing the Jacobian 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.

5.12.7 pyblp.CostsFloatingPointError

class pyblp.**CostsFloatingPointError**

Encountered floating point issues 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.

5.12.8 pyblp.OmegaByThetaJacobianFloatingPointError

class pyblp.**OmegaByThetaJacobianFloatingPointError**

Encountered floating point issues when computing the Jacobian 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.

5.12.9 pyblp.SyntheticPricesFloatingPointError

class pyblp.**SyntheticPricesFloatingPointError**

Encountered floating point issues 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.

5.12.10 pyblp.SyntheticSharesFloatingPointError

class pyblp.**SyntheticSharesFloatingPointError**

Encountered floating point issues 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.

5.12.11 pyblp.EquilibriumPricesFloatingPointError

class pyblp.**EquilibriumPricesFloatingPointError**

Encountered floating point issues when computing equilibrium prices.

This problem is often due to prior problems or overflow and can sometimes be mitigated by rescaling data, removing outliers, or changing the floating point precision.

5.12.12 pyblp.EquilibriumSharesFloatingPointError

class pyblp.**EquilibriumSharesFloatingPointError**
Encountered floating point issues when computing equilibrium shares.

This problem is often due to prior problems or overflow and can sometimes be mitigated by rescaling data, removing outliers, or changing the floating point precision.

5.12.13 pyblp.AbsorptionConvergenceError

class pyblp.**AbsorptionConvergenceError**
An iterative de-meaning procedure failed to converge when absorbing fixed effects.

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 choosing less complicated sets of fixed effects.

5.12.14 pyblp.ThetaConvergenceError

class pyblp.**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.

5.12.15 pyblp.DeltaConvergenceError

class pyblp.**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 bounds, or using different iteration or optimization configurations.

5.12.16 pyblp.SyntheticPricesConvergenceError

class pyblp.**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.

5.12.17 pyblp.EquilibriumPricesConvergenceError

class pyblp.**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.

5.12.18 pyblp.ObjectiveReversionError

class pyblp.**ObjectiveReversionError**
Reverted a problematic GMM objective value.

5.12.19 pyblp.GradientReversionError

class pyblp.**GradientReversionError**
Reverted problematic elements in the GMM objective gradient.

5.12.20 pyblp.DeltaReversionError

class pyblp.**DeltaReversionError**
Reverted problematic elements in δ.

5.12.21 pyblp.CostsReversionError

class pyblp.**CostsReversionError**
Reverted problematic marginal costs.

5.12.22 pyblp.XiByThetaJacobianReversionError

class pyblp.**XiByThetaJacobianReversionError**
Reverted problematic elements in the Jacobian of ξ (equivalently, of δ) with respect to θ.

5.12.23 pyblp.OmegaByThetaJacobianReversionError

class pyblp.**OmegaByThetaJacobianReversionError**
Reverted problematic elements in the Jacobian of ω (equivalently, of transformed marginal costs) with respect to θ.

5.12.24 pyblp.AbsorptionInversionError

class pyblp.**AbsorptionInversionError**
Failed to invert the A matrix from *Somaini and Wolak (2016)* when absorbing two-way fixed effects.

The formulated fixed effects may be highly collinear.

5.12.25 pyblp.HessianEigenvaluesError

class pyblp.**HessianEigenvaluesError**
Failed to compute eigenvalues for the GMM objective's Hessian matrix.

5.12.26 pyblp.FittedValuesInversionError

class pyblp.**FittedValuesInversionError**
Failed to invert an estimated covariance when computing fitted values.

There are probably collinearity issues.

5.12.27 pyblp.SharesByXiJacobianInversionError

class pyblp.**SharesByXiJacobianInversionError**
Failed to invert a Jacobian of shares with respect to ξ when computing the Jacobian of ξ (equivalently, of δ) with respect to θ.

5.12.28 pyblp.IntraFirmJacobianInversionError

class pyblp.**IntraFirmJacobianInversionError**
Failed to invert an intra-firm Jacobian of shares with respect to prices when computing η.

5.12.29 pyblp.LinearParameterCovariancesInversionError

class pyblp.**LinearParameterCovariancesInversionError**
Failed to invert an estimated covariance matrix of linear parameters.

One or more data matrices may be highly collinear.

5.12.30 pyblp.GMMParameterCovariancesInversionError

class pyblp.**GMMParameterCovariancesInversionError**
Failed to invert an estimated covariance matrix of GMM parameters.

One or more data matrices may be highly collinear.

5.12.31 pyblp.GMMMomentCovariancesInversionError

class pyblp.**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 (2019)*, which describes the advances implemented in the package.

6.1 Conlon and Gortmaker (2019)

Conlon, Christopher T., and Jeff Gortmaker (2019). Best practices for differentiated products demand estimation with pyblp. Working paper.

6.2 Other References

6.2.1 Amemiya (1977)

Amemiya, Takeshi (1977). A note on a heteroscedastic model. *Journal of Econometrics, 6* (3), 365-370.

6.2.2 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.

6.2.3 Armstrong (2016)

Armstrong, Timothy B. (2016). Large market asymptotics for differentiated product demand estimators with economic models of supply. *Econometrica, 84* (5), 1961-1980.

6.2.4 Berry (1994)

Berry, Steven (1994). Estimating discrete-choice models of product differentiation. *RAND Journal of Economics, 25* (2), 242-262.

6.2.5 Berry, Levinsohn, and Pakes (1995)

Berry, Steven, James Levinsohn, and Ariel Pakes (1995). Automobile prices in market equilibrium. *Econometrica, 63* (4), 841-890.

6.2.6 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.

6.2.7 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.

6.2.8 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.

6.2.9 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.

6.2.10 Chamberlain (1987)

Chamberlain, Gary (1987) Asymptotic efficiency in estimation with conditional moment restrictions. *Journal of Econometrics, 34* (3), 305-334.

6.2.11 Conlon and Mortimer (2018)

Conlon, Christopher T., and Julie H. Mortimer (2018). Empirical properties of diversion ratios. NBER working paper 24816.

6.2.12 Frisch and Waugh (1933)

Frisch, Ragnar, and Frederick V. Waugh (1933). Partial time regressions as compared with individual trends. *Econometrica, 1* (4), 387-401.

6.2.13 Gandhi and Houde (2017)

Gandhi, Amit, and Jean-François Houde (2017). Measuring substitution patterns in differentiated products industries. Working paper.

6.2.14 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.

6.2.15 Guimarães and Portugal (2010)

Guimarães, Paulo, and Pedro Portugal (2010). A simple feasible procedure to fit models with high-dimensional fixed effects. *Stata Journal, 10* (4), 628-649.

6.2.16 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.

6.2.17 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.

6.2.18 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.

6.2.19 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.

6.2.20 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.

6.2.21 Nevo (1997)

Nevo, Aviv (1997). Mergers with differentiated products: The case of the ready-to-eat cereal industry. Competition Policy Center, Working Paper Series qt1d53t6ts, Competition Policy Center, Institute for Business and Economic Research, UC Berkeley.

6.2.22 Nevo (2000)

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.

6.2.23 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.

6.2.24 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.

6.2.25 Rios-Avila (2015)

Rios-Avila, Fernando (2015). Feasible fitting of linear models with N fixed effects. *Stata Journal, 15* (3), 881-898.

6.2.26 Skrainka (2012)

Skrainka, Benjamin S. (2012). A large scale study of the small sample performance of random coefficient models of demand.

6.2.27 Somaini and Wolak (2016)

Somaini, Paulo, and Frank A. Wolak (2016). An algorithm to estimate the two-way fixed effects model. *Journal of Econometric Methods, 5* (1), 143-152.

6.2.28 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.

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. This FAQ contains some useful information about how to use tox on Windows.

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.

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. 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 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

10.2 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

10.3 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

10.4 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

10.5 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

10.6 0.2

- Improved support for longdouble precision
- Custom ownership matrices
- New benchmarking statistics
- Supply-side gradient computation
- Improved configuration for the automobile example problem

10.7 0.1

- Initial release

Part III

Indices

PYTHON MODULE INDEX

p

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z