Econ 450-1: Industrial Organization
Problem Set #4

Johannes Hirvonen, Russell Miles

December 3, 2024

1 Distributions

This section provides our answers to the first part of the problem set, asking about the distributions of prices, profits, and consumer surplus.

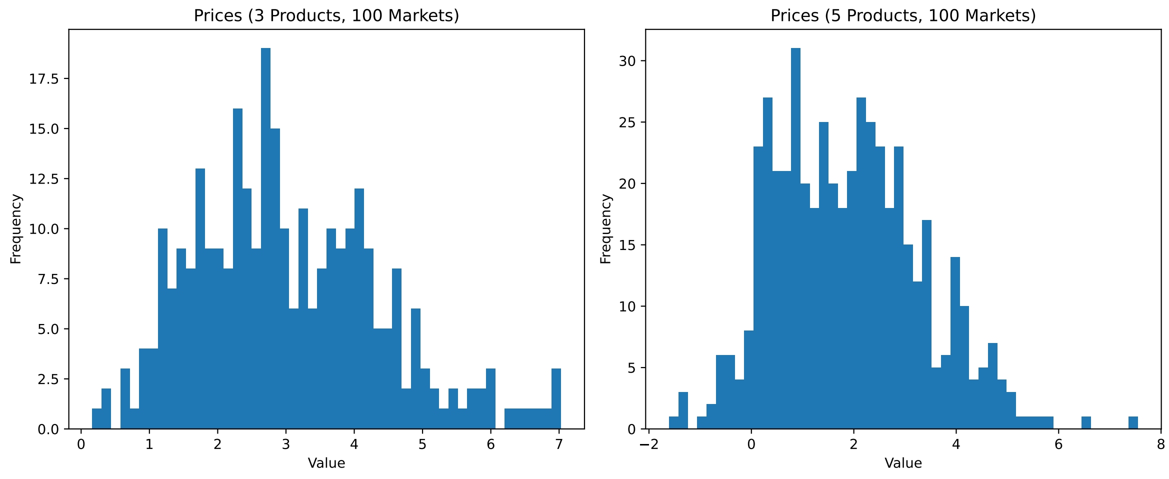


Figure 1: Price Distributions

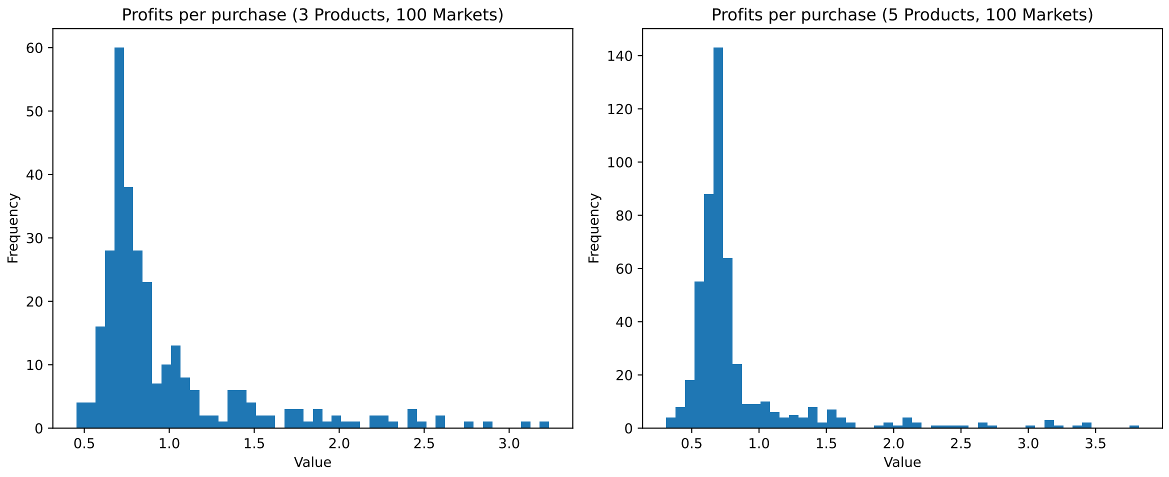


Figure 2: Profit Distributions

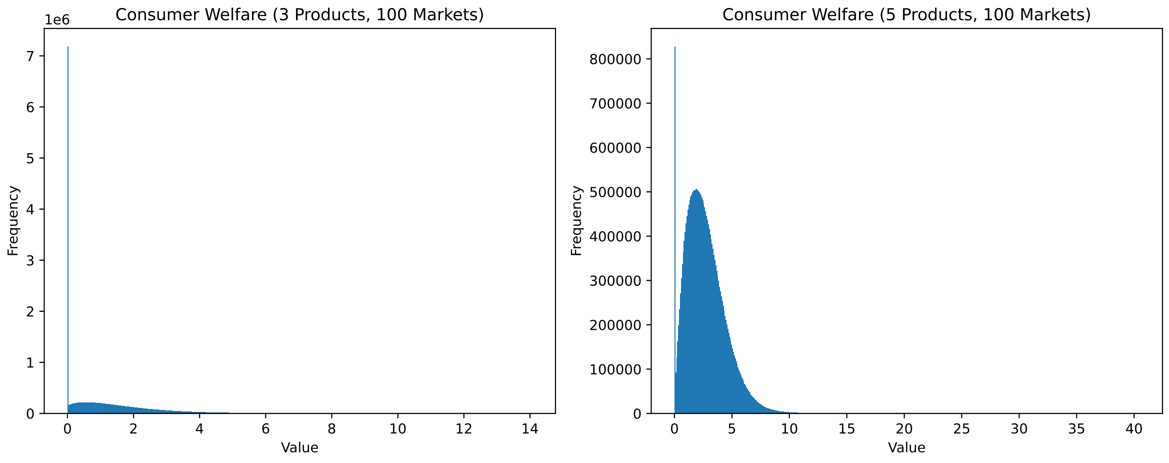


Figure 3: Consumer Welfare Distributions

2 BLP and Hausman Instruments

Throughout the problem set we use the notation $X = \begin{bmatrix} 1 & x_1 & x_2 \end{bmatrix}$ as opposed to the notation in the assignment handout, which labels the regressors as x_1, x_2, and x_3. The computed values for the moment conditions are:

$$\begin{aligned} E[\xi_{jm} X_{jm}] &= \begin{bmatrix} 0.04346 & 0.02074 & 0.03553 \end{bmatrix}, \\ E[\xi_{jm} p_{jm}] &= 0.2950, \\ E[\xi_{jm} \bar{p}_{jm}] &= 0.1437. \end{aligned}$$

2.1 Which of these moment conditions is valid? Which are relevant? Why?

Consider $E[\xi|X] = 0$ first. For the moment condition to be valid for all X (including the firm's own product characteristics), we would need the observable product characteristics X_{jm} to be independent of the unobserved product characteristics (or things like market-specific brand value), ξ_{jm}. In practice, this would mean that firms with certain types of observable product characteristics do not have systematically different unobserved product characteristics. This seems unlikely to hold in practice, since firms are able to both see the unobserved (to the econometrician) product characteristics before choosing the observed (again, to the econometrician) product characteristics. For example, we would expect that a "premium" car (something that is in ξ) typically has different product characteristics that a non-premium car. Still, to know whether this moment condition would be valid in this context, we would need more information on what type of products we are studying.

The moment condition is relevant though, as long as X is included in, or correlated with, the cost shifter z_{jm}^{cost}, i.e. the product characteristics are costly. The relevance is then clear from the fact that prices are a function of marginal costs, which are a function of X.

As for $E[\xi|p] = 0$, the validity depends on which prices we consider. With own prices, it's clear that validity is not satisfied, since in general, prices are higher for products with greater ξ, as firms choose prices with knowledge of their market-specific ξ_{jm}. But the mean independence might be satisfied for own prices in other markets (Hausman instruments), if we include firm fixed effects to control for the firm-level component of ξ that is invariant accross markets and construct the moment condition with the market-specific part of ξ. Note that relevance is trivially satisfied in both these cases, since (1) prices are obviously correlated with themselves, and (2) prices in other markets are correlated with prices in a specific market because there is a common cost shifter for all markets, W_j.

2.2 Can you use both BLP and Hausman instruments in this setting? Why? Why not?

If the way firms choose product characteristics X is such that the unobserved product characteristics are not observed before choosing observable characteristics, the BLP instruments can be used. Similarly, in the previous part we have argued that the Hausman instruments are relevant, and are also valid if there are e.g. no demand shocks that are common across all markets or marketing campaigns of the firms that include multiple markets.

3 BLP Estimation: Results

Our methods are described in more detail in the Appendix. We use BLP instruments (no Hausman instruments). Below, we present the results.

3.1 Demand-side estimation, 100 markets

3.1.1 Parameter Estimates

This estimation was robust to various starting points. Table 1 shows the estimated parameter values. The estimates are unbiased (up to the standard error) and statistically different from zero. Note that for all estimations, we used an identity weighting matrix; it would be possible to further increase precision by using the optimal weighting matrix in a two-step estimation procedure.

Table 1: Demand-Side Estimation, 100 Markets

Parameter	Estimate	Std. Error	True Value	Bias
σ_α	0.9888	0.0163	1	-0.0112
β_0	4.9282	0.2712	5	-0.0718
β_1	0.9552	0.2135	1	-0.0448
β_2	1.0287	0.0606	1	0.0287
α	0.9676	0.0913	1	-0.0324

For the following parts, we average across markets and/or individuals to report elasticities, profits, and consumer surplus. We manually checked the estimated and true values for each market, and they are consistently close (generally within 2 decimal points).

3.1.2 Price Elasticity of Demand

Table 2: Comparison of True and Estimated Elasticities: ϵ_{ij}

	Estimated			**True**		
	p_1	p_2	p_3	p_1	p_2	p_3
s_1	-4.9767	1.3583	0.9733	-4.9792	1.3620	0.9764
s_2	0.5777	-2.5661	0.9614	0.5798	-2.5689	0.9648
s_3	0.5867	1.4055	-3.3316	0.5889	1.4105	-3.3357

We average elasticities across markets. Our elasticity estimates are accurate to two decimal points.

3.1.3 Profits

To make profits comparable across products, **we report the average profit per unit sold, i.e. price minus marginal cost (the markup)**. Our estimates are fairly accurate.

Table 3: Average Profits per Purchase, Demand-Side

Product	**Estimate**	**True**	**Difference**
Product 1	1.0293	1.0088	0.0205
Product 2	0.9857	0.9664	0.0194
Product 3	0.9706	0.9509	0.0197

3.1.4 Consumer Surplus

We report the average consumer surplus across individuals. Our estimate is close.

Table 4: Consumer surplus comparison, demand-side estimation

	Estimated	**True**
Consumer Surplus	1.3469	1.3139

3.2 Demand-side estimation, 10 markets

Using only 10 markets, the estimator becomes unstable, and the estimate becomes dependent on the initial guess. We include results for two initial guesses: one with $\delta_{jm} = 3$ and the other with $\delta_{jm} = 1$ for all products and markets. Both use $\sigma_\alpha = \alpha = 1$ for initial guesses.

Table 5: Demand-Side Estimation, 10 Markets, $\delta = 3$

Parameter	Estimate	Std. Error	True Value	Bias
σ_α	1.2211	0.0814	1	0.2211
β_0	3.1485	2.2535	5	-1.8515
β_1	0.6091	1.2361	1	-0.3909
β_2	1.5000	0.4542	1	0.5000
α	-0.1478	0.8271	1	-1.1478

Table 6: Demand-Side Estimation, 10 Markets, $\delta = 1$

Parameter	Estimate	Std. Error	True Value	Bias
σ_α	0.5041	0.0673	1	-0.4959
β_0	2.6193	2.4979	5	-2.3807
β_1	0.4210	1.0830	1	-0.5790
β_2	1.3671	0.4674	1	0.3671
α	0.6554	0.9338	1	-0.3446

As we can see in Tables 5 and 6, the standard errors become considerably larger when using only 10 markets. The estimates are biased.

3.3 Demand-side estimation assuming exogeneity of price, 100 markets

In this exercise, we falsely assume $E[\xi|p] = 0$. Under this assumption, we do not need BLP instruments, so our only regressors are $1, x_1, x_2$, and p.

Table 7: Demand-Side assuming exog. price, 100 Markets, $\delta = 3$

Parameter	Estimate	Std. Error	True Value	Bias
σ_α	1.1463	0.0174	1	0.1463
β_0	5.2185	0.1690	5	0.2185
β_1	1.0087	0.2201	1	0.0087
β_2	1.0850	0.0653	1	0.0850
α	0.8920	0.0455	1	-0.1080

Table 8: Demand-Side assuming exog. price, 100 Markets, $\delta = 1$

Parameter	Estimate	Std. Error	True Value	Bias
σ_α	0.6143	0.0154	1	-0.3857
β_0	3.7970	0.1584	5	-1.2030
β_1	0.7472	0.1926	1	-0.2528
β_2	0.8889	0.0560	1	-0.1111
α	0.9885	0.0410	1	-0.0115

The estimate is unstable and dependent on the initial guess. This is not surprising, since we are using an endogenous variable as an instrument. The results in Table 7 using the initial guess of $\delta = 3$ look better than the results in Table 8 using $\delta = 1$; however, the underlying method is invalid, so we do not want to read too much into these results.

4 Including the supply side

4.1 Demand-side estimation using cost shifter w^{cost}, 100 markets

This estimation procedure was robust to various initial guesses. The results are shown in Table 9.

Table 9: Demand-Side using w^{cost} as an instrument, 100 Markets

Parameter	Estimate	Std. Error	True Value	Bias
σ_α	0.9900	0.0163	1	-0.0100
β_0	4.9296	0.2714	5	-0.0704
β_1	0.9591	0.2142	1	-0.0409
β_2	1.0293	0.0607	1	0.0293
α	0.9674	0.0913	1	-0.0326

4.1.1 Price Elasticity of Demand

Table 10: Comparison of True and Estimated Elasticities: ϵ_{ij}

	Estimated			**True**		
	p_1	p_2	p_3	p_1	p_2	p_3
s_1	-4.9771	1.3587	0.9736	-4.9793	1.3620	0.9764
s_2	0.5780	-2.5665	0.9618	0.5799	-2.5690	0.9648
s_3	0.5869	1.4061	-3.3321	0.5889	1.4105	-3.3358

4.1.2 Profits

Table 11: Average Profits per Purchase, Demand-Side with w^{cost}

Product	**Estimate**	**True**	**Difference**
Product 1	1.0291	1.0088	0.0203
Product 2	0.9856	0.9664	0.0193
Product 3	0.9705	0.9509	0.0196

4.1.3 Consumer Surplus

Table 12: Consumer Surplus, Demand-Side with w^{cost}

	Estimated	**True**
Consumer Surplus	1.3412	1.3156

As with the first specification using 100 markets, our estimates are close to the true values.

4.2 Demand-side estimation using cost shifter w^{cost}, 10 markets

As in the previous case, this estimator is unstable with only 10 markets, and sensitive to the initial guess. Table 13 shows one example of the output, choosing $\sigma_\alpha = \alpha = 1$ and $\delta = 3$.

Table 13: Demand-Side using w^{cost} as an instrument, 10 Markets, $\delta = 1$

Parameter	Estimate	Std. Error	True Value	Bias
σ_α	1.4919	0.0914	1	0.4919
β_0	3.3318	2.4061	5	-1.6682
β_1	0.7327	1.3836	1	-0.2673
β_2	1.5587	0.4956	1	0.5587
α	-0.4552	0.8720	1	-1.4552

4.3 Comparison with the estimates using BLP instruments alone

See Tables 1, 2, 3 and 4, and compare to Tables 9, 10, 11 and 12. The elasticities, profits, and consumer surplus are very close to the ones estimated without using w^{cost} as an instrument.

5 Joint estimation of demand and supply side

5.1 Oligopoly

Oligopoly is the true market structure. Thus, as expected, the estimates are mostly unbiased (the true value is generally within 2 $\times$ the standard error). γ_2 appears to be biased downwards slightly. Note that we use a first-order Delta method approximation of the variance-covariance matrix, which may underestimate the true standard errors for a nonlinear function such a the BLP estimator.

Table 14: Joint Estimation, Oligopoly, 100 Markets

Parameter	Estimate	Std. Error	True Value	Bias
σ_α	0.9900	0.0163	1	-0.0100
β_0	4.9296	0.2714	5	-0.0704
β_1	0.9591	0.2142	1	-0.0409
β_2	1.0293	0.0607	1	0.0293
α	0.9674	0.0913	1	-0.0326
γ_0	1.9326	0.0445	2	-0.0674
γ_1	0.9788	0.0210	1	-0.0212
γ_2	0.9166	0.0051	1	-0.0834

5.2 Collusion

When we (falsely) assume the data is generated from collusion (see Table 15), the estimates of γ have a large downward bias. This is because the estimator needs to rationalize the prices in the data under collusion conduct, which would imply low marginal costs.

Table 15: Joint Estimation, Collusion, 100 Markets

Parameter	Estimate	Std. Error	True Value	Bias
σ_α	0.9900	0.0163	1	-0.0100
β_0	4.9296	0.2714	5	-0.0704
β_1	0.9591	0.2142	1	-0.0409
β_2	1.0293	0.0607	1	0.0293
α	0.9674	0.0913	1	-0.0326
γ_0	-1.1232	0.1113	2	-3.1232
γ_1	0.6047	0.0123	1	-0.3953
γ_2	0.8008	0.0263	1	-0.1992

5.3 Perfect Competition

Table 16: Joint Estimation, Perfect Competition, 100 Markets

Parameter	Estimate	Std. Error	True Value	Bias
σ_α	0.9900	0.0163	1	-0.0100
β_0	4.9296	0.2714	5	-0.0704
β_1	0.9591	0.2142	1	-0.0409
β_2	1.0293	0.0607	1	0.0293
α	0.9674	0.0913	1	-0.0326
γ_0	2.9509	0.0557	2	0.9509
γ_1	0.7755	0.0677	1	-0.2245
γ_2	0.7897	0.0554	1	-0.2103

When we (falsely) assume the data is generated from perfect competition (see table 16), γ_0 has a large upward bias. Now the estimator sets marginal cost equal to the observed price, leading to a biased estimate for γ. Table 17 shows the estimated marginal costs for each mode of conduct.

5.4 Comparison of Marginal Costs

Table 17: Comparison of True and Estimated Mean Marginal Costs

		Mean Marginal Costs		
		Estimated	**True**	**Difference**
Perfect Competition	Product 1	2.991	1.982	1.009
	Product 2	2.958	1.992	0.966
	Product 3	3.131	2.180	0.951
Oligopoly	Product 1	1.962	1.982	-0.020
	Product 2	1.973	1.992	-0.019
	Product 3	2.160	2.180	-0.020
Collusion	Product 1	-2.392	1.982	-4.375
	Product 2	-0.790	1.992	-2.782
	Product 3	-0.034	2.180	-2.213

See Table 17 for our estimates of mean marginal costs by product. Our estimates using oligopoly (the true conduct) have a very small bias. Assuming perfect competition, the estimates are biased upwards, as marginal cost is assumed to equal the observed price. Assuming collusion, marginal costs are biased downwards (and become negative) in order to rationalize the chosen prices.

We choose the joint estimation using oligopoly conduct as our preferred estimation procedure. The standard errors and bias are close to those from the other methods using 100 markets (except for the one with the incorrect use of price as an instrument), so this choice is somewhat arbitrary. However, it is comforting that this procedure gets very close to the correct parameters and marginal costs, and the additional data provided by the supply-side moments could make it more robust than other methods. It also converges faster than the demand-side estimation (with or without w^{cost} as a moment).

5.5 Demand Elasticities

The elasticity of demand is unaffected by conduct, so we have only one estimated martrix of mean elastricities. These are identical to those calculated using demand-side estimation with w^{cost} as an instrument.

Table 18: Comparison of True and Estimated Elasticities: ϵ_{ij}

	Estimated			True		
	p_1	p_2	p_3	p_1	p_2	p_3
s_1	-4.9771	1.3587	0.9736	-4.9793	1.3620	0.9764
s_2	0.5780	-2.5665	0.9618	0.5799	-2.5690	0.9648
s_3	0.5869	1.4061	-3.3321	0.5889	1.4105	-3.3358

6 Merger Exercise

We use the joint estimation for this exercise. Although all of our 100-markets estimates are very close and have similar standard errors, the joint estimation converges the fastest and is able to match more moments in the data, so we prefer it.

Table 19: Mean Markups Pre- and Post-Merger

Product	**Pre-Merger**	**Post-Merger**
Product 1	0.8113	1.6176
Product 2	1.2230	1.4383
Product 3	0.9517	0.9748

Markups increase for all firms. The increases are largest for Products 1 and 2, which are now sold by the merged entity. Product 3 has a small increase in its markup, as the third firm now faces softer competition.

Table 20: Mean Prices Pre- and Post-Merger

Product	**Pre-Merger**	**Post-Merger**
Product 1	3.8858	4.6921
Product 2	2.4579	2.6732
Product 3	2.7359	2.7590

Like markups, the prices increase most for the products owned by the merged entity, but Product 3 also experiences a small price increase.

Table 21: Average Profits by Product (across markets) Pre- and Post-Merger

Product	**Pre-Merger**	**Post-Merger**
Product 1	0.1013	0.1099
Product 2	0.5476	0.5696
Product 3	0.2936	0.3100

For calculating the average profits by product, we weight markups by shares in each market, then average across markets. Profits increase for all firms, as competition has decreased.

Table 22: Mean Consumer Surplus Pre- and Post-Merger

	Pre-Merger	**Post-Merger**
Consumer Surplus	1.3384	1.2490

Consumer surplus declines after the merger, as the market is less competitive and firms charge higher prices.

A Methods

This section presents the methods we use in estimation. Code is provided in the last section of the Appendix.

A.1 Consumers

Consumer $i's$ preference for product j in market m is assumed to take the following form:

$$U_{ijm} = X_{jm}\beta - \alpha_i p_{jm} + \xi_{jm} + \epsilon_{ijm},$$

where:

$$\alpha_i = \alpha + \sigma_\alpha \nu_{im},$$

and:

$$U_{i0} = 0.$$

Product j in market m has mean utility

$$\delta_{jm} = X_{jm}\beta - \alpha p_{jm} + \xi_{jm}.$$

The random component of the consumer's utility, excluding the epsilon, is given by:

$$\mu_{ijm} = -\sigma_\alpha \nu_{im} p_{jm}.$$

A.2 The Share Equation

Given the mean utility of each product-market, δ_{jm}, the random component of utility, μ_{ijm}, and the distributional assumption on ϵ_{ijm} (Type 1 EV), the market share equation is given by:

$$s(\delta, \sigma_\alpha) = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{\exp(\delta_{jm} + \mu_{ijm})}{1 + \sum_{k=1}^{J}\exp(\delta_{km} + \mu_{ikm})}\right).$$

A.3 Instrumental Variables:

First, we redefine the equation for mean utilities in stacked vector-matrix form:

$$\delta = \bar{X}\beta + \xi,$$

where:

$$\bar{X} = \begin{bmatrix} \mathbf{1} & \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{p} \end{bmatrix}$$

and:

$$\beta = \begin{bmatrix} \beta_0 & \beta_1 & \beta_2 & -\alpha \end{bmatrix}^\intercal .$$

Price is endogenous, but we have a matrix of instruments Z satisfying $E\left[\xi|Z\right] = 0$. The instruments are:

$$\bar{Z} = \begin{bmatrix} \mathbf{1} & \mathbf{x}_1 & \mathbf{x}_2 & \bar{\mathbf{x}}_{\mathbf{1,j}} & \bar{\mathbf{x}}_{\mathbf{2,j}} & \bar{\mathbf{x}}_{\mathbf{1,m}} & \bar{\mathbf{x}}_{\mathbf{2,m}} \end{bmatrix}$$

where the last four are the BLP instruments. The IV estimator is:

$$\hat{\beta}^{IV} = \left[\left(\bar{X}^\intercal P_Z \bar{X}\right)^{-1} \bar{X}^\intercal P_Z \right] \delta,$$

and the residuals are given by:

$$\hat{\xi}^{IV} = \underbrace{\left[\mathbb{I} - \bar{X} \left(\bar{X}^\intercal P_Z \bar{X}\right)^{-1} \bar{X}^\intercal P_Z \right]}_{\equiv A_Z} \delta.$$

A.4 The MPEC for the Demand-Side

We solve the following MPEC formulation:

$$\min_{\theta \equiv [\sigma_\alpha, \eta, \delta]} \eta' W \eta$$
$$s\left(\theta\right) - \text{shares} = 0, \quad \bar{g}(\theta) - \eta = 0$$

Here, $\bar{g}(\theta) = \frac{1}{JM}\sum_{jm} g_{jm}(\theta)$ and $g_{jm}(\theta) = \left[\mathbf{z}_{jm}\xi_{jm}\right]_{n_I \times 1}$, where n_I is the number of instruments. Using *jax*, we automatically differentiate the constraint functions with respect to θ to obtain Jacobians. To solve the MPEC, we use SciPy's minimize with the constrained trust region (`trust-constr`) algorithm.

A.5 Standard Errors

The variance-covariance matrix (using $W = \mathbb{I}$) is:

$$V_{GMM} = \left[G^\intercal G\right]^{-1} G^\intercal \bar{B} G \left[G^\intercal G\right]^{-1},$$

where:

$$G \approx \frac{1}{JM} \sum_{j,m} \frac{\partial g_{jm}(\theta)}{\partial \theta},$$

and:

$$\bar{B} \approx \frac{1}{JM} \sum_{j,m} g_{jm}(\theta) g_{jm}(\theta)^{\intercal}.$$

G will have rows and columns of zeros corresponding to η, so we use the pseudo-inverse of $G^{\intercal}G$. Here, the function $g(\theta)$ represents the vector of moment conditions (before averaging):

$$g_{jm}(\theta) = [\mathbf{z}_{jm}\xi_{jm}]_{n_I \times 1}.$$

This formula gives the variances for the predicted parameters:

$$\begin{bmatrix} \sigma_\alpha & \eta & \delta \end{bmatrix}.$$

To obtain the variance-covariance matrix of $\beta = [\beta_0, \beta_1, \beta_2, -\alpha]$, we use the property that $\beta = \mathcal{M}\delta$, where $\mathcal{M} = (\bar{X}^{\intercal} P_z \bar{X})^{-1} \bar{X}^{\intercal} P_z$:

$$\text{Var}(\beta) = \mathcal{M}\,\text{Var}(\delta)\mathcal{M}^{\intercal},$$

where $\text{Var}(\delta)$ is the last JM rows and columns of V_{GMM}. Standard errors are then given by:

$$\text{se}(\beta) = \sqrt{\frac{\text{Var}(\beta)}{JM}}, \quad \text{se}(\sigma_\alpha) = \sqrt{\frac{V_{GMM[1,1]}}{JM}}.$$

A.5.1 Note on deriving $dg/d\theta$

Note that:

$$\frac{\partial g_{jm}(\theta)}{\partial \theta} = \frac{\partial z_{jm} \cdot \xi_{jm}}{\partial \theta} = z_{jm} \cdot \frac{\partial\left(\delta_{jm} - \beta X_{jm} + \alpha p_{jm}\right)}{\partial \theta} = z_{jm} \cdot \frac{\partial \delta_{jm}}{\partial \theta}$$

where z_{jm} is the vector of moment conditions of product j and market m. Then

$$\frac{d\delta}{d\theta} = \begin{bmatrix} \frac{d\delta}{d\sigma}_{JM\times 1} & \mathbf{0}_{JM \times n_I} & \mathbb{I}_{JM \times JM} \end{bmatrix},$$

where the matrix of zeros comes from derivatives with respect to η, and the big identity matrix $\mathbb{I}_{JM\times JM}$ comes from the derivatives with respect to δ. Next, to derive $d\delta/d\sigma$, totally differentiate the share-matching constraint $S(\delta,\sigma) - s = 0$ to get:

$$\begin{aligned} & 0 = \nabla_\theta S(\delta, \sigma) \\ \iff\ & 0 = \underbrace{D_\delta S(\delta,\sigma)}_{JM\times JM} \underbrace{d\delta}_{JM\times 1} + \underbrace{\frac{\partial S(\delta,\sigma)}{\partial \sigma}}_{JM \times 1} \underbrace{d\sigma}_{1\times 1} \\ \iff\ & \frac{d\delta}{d\sigma} = -\left[D_\delta S(\delta,\sigma)\right]^{-1} \left[\frac{\partial S(\delta,\sigma)}{\partial \sigma}\right]. \end{aligned}$$

We can get these easily from pieces of our Jacobian of $S(\delta, \sigma)$, constructed using *jax*.

A.6 Estimating Price Elasticity of Demand

The price-elasticity of demand for product j with respect to the price of product k in a given market is:

$$\mathcal{E}_{jk} = \frac{\partial s_j p_k}{\partial p_k s_j} = \begin{cases} -\frac{p_j}{s_j} \int \alpha_i s_{ij} \left(1 - s_{ij}\right) dP_v(v) & \text{if } j = k, \\ \frac{p_k}{s_j} \int \alpha_i s_{ij} s_{ik} dP_v(v) & \text{otherwise.} \end{cases}$$

This is straightforward to estimate by averaging across the simulated consumers, that is, using Monte-Carlo integration.

A.7 Estimating Marginal Costs

Assuming the firms' conduct is Nash-in-prices, we can back out the marginal costs from the firms' first-order conditions. The FOC within a given market is then:

$$0 = \frac{\partial \pi_f}{\partial p_k} = \frac{\partial}{\partial p_k} \sum_{j \in F_f} s_j \left(p_j - mc_j\right) = s_k + \sum_{j \in F_f} \frac{\partial s_j}{\partial p_k} \left(p_j - mc_j\right),$$

where F_f denotes the set of products (in the market) that are produced by firm f. This can then be rewritten in vector form:

$$0 = s + \Delta(p - mc)$$

where Δ is a $J \times J$ matrix with $\Delta_{j,k}$ equal to $\left(\frac{\partial s_k}{\partial p_j}\right)$ if both j and k owned by the same firm, and equal to zero otherwise. The vector of marginal costs for all products is then:

$$mc = \Delta^{-1} s + p.$$

To see how the marginal costs are estimated for the perfect collusion and oligopoly cases, we can decompose Δ:

$$\Delta = \Omega \odot \mathcal{S}.$$

In the notation above, Δ equals the Hadamard product ($\odot$) between the ownership matrix Ω and the $J \times J$ matrix $\mathcal{S}$ of share partial derivatives with respect to price, where $\mathcal{S}_{j,k}$ equals $\left(\frac{\partial s_k}{\partial p_j}\right)$. The ownership matrix elements Ω_{jk} equal one if the products j and k are owned by the same firm and zero otherwise.

Now, if the conduct is perfect collusion, the firms act as one merged entity. In other words, they internalize the fact that raising prices in one product leads some consumers to choose a "competing" firms product. The profit from this diversion under collusion isn't lost, so prices have upward pressure. In this case (with three products in the market), marginal costs are given by the equation above with the ownership matrix being full of ones:

$$\Omega_{collusion} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

If the pricing is oligopoly pricing, the firms have markups, but they only consider the demand of their own product when setting prices. That is, the ownership matrix takes the following form:

$$\Omega_{oligopoly} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Finally, if the pricing is marginal cost pricing, marginal costs simply equal prices:

$$mc_{jm} = p_{jm}.$$

A.8 Estimating Consumer Surplus

The consumer surplus for consumer i in a given market is:

$$CS_i = \frac{1}{\alpha_i} \cdot \underbrace{\max_j u_{ij}}_{\text{utility from choice}}.$$

The division by α_i is to transform the units from utils to monetary terms. In the random-coefficient logit model, the mean of the utility from the best choice has a closed-form expression:

$$E\left[\max_j u_{ij}\right] = \sum_j E\left[u_{ij} \mid u_{ij} \geq u_{ik} \text{ for all } k\right] \cdot s_{ij} = \log\left(1 + \sum_j \exp\left(\delta_{jm} + \mu_{ijm}\right)\right).$$

We use this expression to calculate the consumer surplus under different parameter values.

A.9 Supply Side

The supply side is defined by the marginal cost equation:

$$MC_{jm} = \gamma_0 + \gamma_1 w_j^{\text{cost}} + \gamma_2 z_{jm}^{\text{cost}} + \eta_{jm}.$$

A.9.1 Additional Moment Using the Cost Shifter

We use the moment condition $E\left[\xi w^{\text{cost}}\right] = 0$. This will hold if:

- *Relevance:* w_j^{cost} is correlated with prices p_{jm}.
- *Exogeneity:* w_j^{cost} is uncorrelated with the unobserved demand shock ξ_{jm}.

Relevance holds, because the cost shifter affects pricing decisions via the marginal costs equation. Exogeneity likely holds as well. For example, if w_j^{cost} represents steel prices and the products represent cars, then the price of steel should not affect consumer preferences over cars (other than through prices). The new matrix of instruments will then be:

$$\bar{Z} = \begin{bmatrix} \mathbf{1} & \mathbf{x}_1 & \mathbf{x}_2 & \bar{\mathbf{x}}_{\mathbf{1,j}} & \bar{\mathbf{x}}_{\mathbf{2,j}} & \bar{\mathbf{x}}_{\mathbf{1,m}} & \bar{\mathbf{x}}_{\mathbf{2,m}} & \mathbf{w}^{\text{cost}} \end{bmatrix}.$$

A.10 Joint Estimation

A.10.1 New Moments

For the demand side, we continue to use the instruments from earlier:

$$\bar{Z}^D = \begin{bmatrix} \mathbf{1} & \mathbf{x}_1 & \mathbf{x}_2 & \bar{\mathbf{x}}_{\mathbf{1,j}} & \bar{\mathbf{x}}_{\mathbf{2,j}} & \bar{\mathbf{x}}_{\mathbf{1,m}} & \bar{\mathbf{x}}_{\mathbf{2,m}} & \mathbf{w}^{\text{cost}} \end{bmatrix}.$$

For the supply side, recall that:

$$mc(\theta) = \gamma_0 + \gamma_1 w_j^{\text{cost}} + \gamma_2 z_{jm}^{\text{cost}} + \omega_{jm} = X^s \gamma + \omega,$$

where $X^s = \begin{bmatrix} \mathbf{1} & w_j^{\text{cost}} & z_{jm}^{\text{cost}} \end{bmatrix}$. Using the exogeneity of the cost shifters, $E\left[\omega | w_j^{\text{cost}}, z_{jm}^{\text{cost}}\right] = 0$, we can get the OLS estimate of γ:

$$\hat{\gamma} = \left((X^s)^\intercal X^s\right)^{-1} (X^s)^\intercal mc(\theta).$$

Using this, we concentrate out γ, so our parameter vector is the same as in the demand-side case, $\theta = [\sigma_\alpha, \eta, \delta]$. To obtain marginal costs $mc(\theta)$, we first calculate the price elasticity of demand, then calculate marginal costs as a function of that price elasticity using the equation(s) in Section A.7.

A.10.2 Joint Estimation: Additional Moments and Standard Errors

In the joint estimation, we now have 3 additional moments. The moment conditions are:

$$g_{jm}(\theta) = \begin{bmatrix} z_{jm}\xi_{jm} \\ X_{jm}^s \omega_{jm} \end{bmatrix} = \begin{bmatrix} z_{jm} A_z^D \delta_{jm} \\ X_{jm}^s \cdot [A^s mc(\theta)]_{jm} \end{bmatrix} \equiv \begin{bmatrix} g_{jm}^{\text{demand}}(\theta) \\ g_{jm}^{\text{supply}}(\theta) \end{bmatrix}.$$

The additional moments are included in the constraint for $\bar{g}$ in the MPEC and in the moment conditions.

For the standard errors, differentiating the supply-side moments with respect to θ, we get:

$$X_{jm}^s \cdot [A^s D_\theta mc(\theta)]_{jm} = X_{jm}^s \left[A^s \frac{\partial mc}{\partial \theta} \right],$$

and:

$$\frac{\partial g_{jm}^{\text{supply}}(\theta)}{\partial \theta} = X_{jm,1\times 3}^s \cdot \left[A_{3\times JM}^s \left(\frac{\partial mc}{\partial \theta} \right)_{JM \times \mathcal{N}_\theta} \right]_{jm}.$$

There are JM observations of $\frac{\partial}{\partial \theta} g_{jm}^{\text{supply}}$, which is a $1 \times \mathcal{N}_\theta$ object. To obtain $\frac{\partial mc}{\partial \theta}$, we use *jax.* Summing over rows, we get:

$$\frac{\partial}{\partial \theta} \bar{g}^{\text{supply}}(\theta) = \frac{1}{JM} \sum_{jm} g_{jm}^{\text{supply}}(\theta),$$

which is a $3 \times \mathcal{N}_\theta$ object. Now, let $G \equiv \begin{bmatrix} \frac{\partial}{\partial\theta}\bar{g}^{\text{demand}} \\ \frac{\partial}{\partial\theta}\bar{g}^{\text{supply}} \end{bmatrix}$ and proceed as in the demand-only case to get V_{GMM} using all 11 moment conditions. We can use V_{GMM} to get the standard errors for $\beta, \alpha, \sigma_\alpha$ as usual. To get the standard errors for γ, we use the delta method:

$$\begin{aligned} \text{Var}(\gamma) &= \mathcal{M}_s \, \text{Var}\left(mc(\theta)\right) \mathcal{M}_s^\intercal \\ &\approx \mathcal{M}_s \left[\nabla_\theta mc(\theta)\right] V_{GMM} \left[\nabla_\theta mc(\theta)\right]^\intercal \mathcal{M}_s^\intercal \\ &= \mathcal{M}_s \left[\frac{\partial mc}{\partial \theta}\right] V_{GMM} \left[\frac{\partial mc}{\partial \theta}\right]^\intercal \mathcal{M}_s^\intercal, \end{aligned}$$

where $\mathcal{M}_s = \left(X_s^\intercal X_s\right)^{-1} X_s^\intercal$. The delta method gives a local approximation, so it may underestimate the true standard errors of a nonlinear function such as the BLP estimator.

A.11 Merger Analysis

To predict the prices post-merger, we follow the method in the PyBLP documentation which cites Morrow and Skerlos (2011). After firms 1 and 2 merge, the ownership matrix becomes:

$$\Omega = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Denote marginal costs as c. Markups are then given by:

$$p - c = \underbrace{\Lambda^{-1}(\Omega \odot \Gamma)'(p - c) - \Lambda^{-1} s}_{\zeta},$$

where Λ is a diagonal $J_m \times J_m$ matrix approximated by:

$$\Lambda_{jj} \approx \sum_{i \in I_m} w_{im} s_{ijm} \frac{\partial U_{ijm}}{\partial p_{jm}},$$

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

$$\Gamma_{jk} \approx \sum_{i \in I_m} w_{it} s_{ijm} s_{ikm} \frac{\partial U_{ikm}}{\partial p_{km}}.$$

Equilibrium prices are computed by iterating over the ζ-markup equation above:

$$p \leftarrow c + \zeta(p),$$

which is a contraction. Iteration terminates when the norm of firms' first order conditions, $\|\Lambda(p)(p - c - \zeta(p))\|_\infty$, is less than a small number (we use $1e-8$).

B Code

All code in this problem set is written in Python. We use *jax* for automatic differentiation to get Jacobians for the MPEC constraints. We first give the Jupyter notebook that runs the estimation routine by calling the relevant functions.

```
[1]: from source_functions import *
     from estimate_BLP import *
     from utils import *
```

Load data and set function arguments

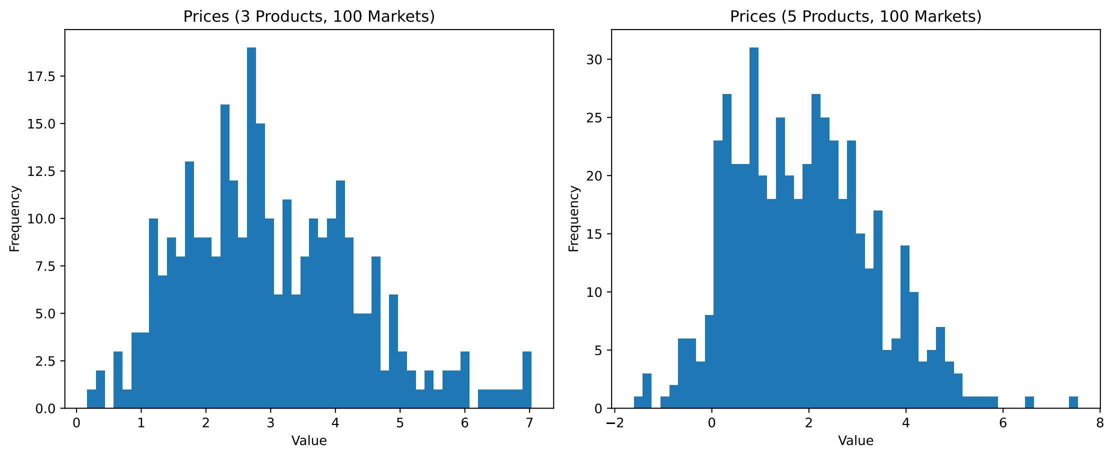
```
[2]: df_3p, alphas_3p = load_mat_data('Simulation Data/100markets3products.mat', 3,␣
      ↪100)
     df_5p, alphas_5p = load_mat_data('Simulation Data/100markets5products.mat', 5,␣
      ↪100)

     sigma_alpha_init = 1.0 #initial guess
     df = df_3p #data
     alphas = alphas_3p #random coefficients on prices
     mode = "demand_side" #demand-side moments only
```

Part 1: Create Graphs

```
[3]: # Set true parameter values
     beta = (5, 1, 1)
     gamma = (2, 1, 1)

     #Draw epsilons
     epsilons_3p = draw_epsilons(alphas_3p)
     epsilons_5p = draw_epsilons(alphas_5p)

     # Calculate marginal costs for the firms
     df_3p['mc'] = gamma[0] + gamma[1]*df_3p['w'] + gamma[2]*df_3p['Z'] + df_3p['eta']
     df_5p['mc'] = gamma[0] + gamma[1]*df_5p['w'] + gamma[2]*df_5p['Z'] + df_5p['eta']

     # Calculate profits per purchase for the firms
     df_3p['Pi'] = df_3p['P_opt'] - df_3p['mc']
     df_5p['Pi'] = df_5p['P_opt'] - df_5p['mc']

     # Calculate consumer welfare
     consumer_welfare_3p = calculate_welfare(df_3p, alphas_3p, beta, epsilons_3p)
     consumer_welfare_5p = calculate_welfare(df_5p, alphas_5p, beta, epsilons_5p)
```

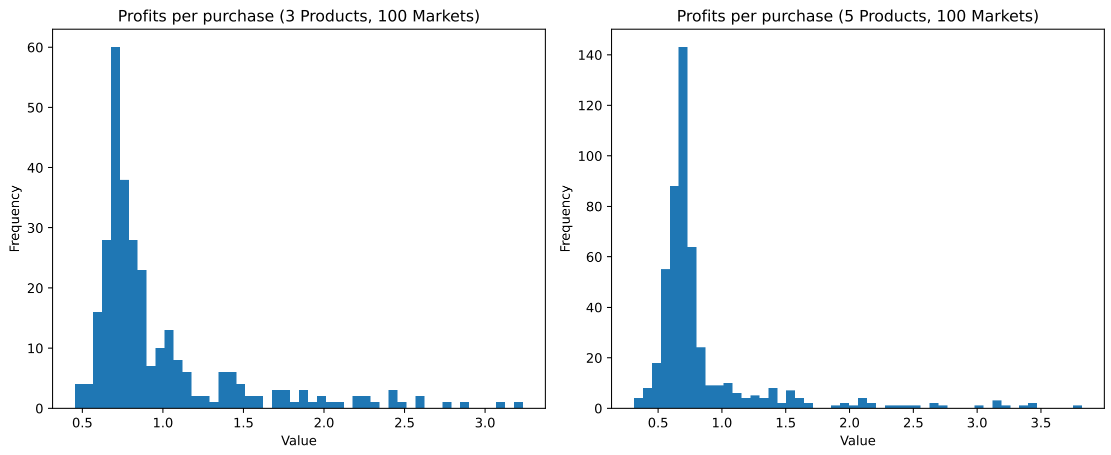
Plot prices

```
[9]: plot_two_histograms(df_3p['P_opt'].values, df_5p['P_opt'].values, bins=50,␣
      ↪labels=('Prices (3 Products, 100 Markets)', 'Prices (5 Products, 100␣
      ↪Markets)'), path="Results/prices_dist.pdf")
```



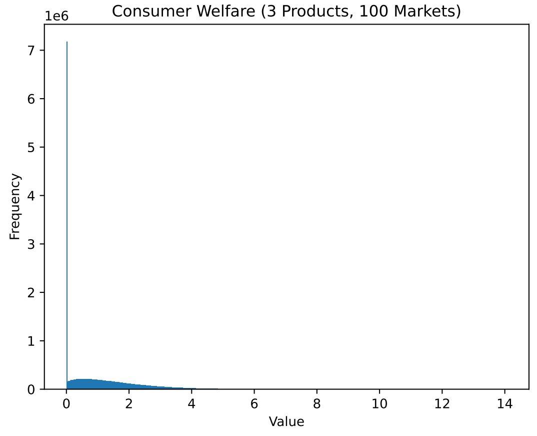
Plot profits per purchase

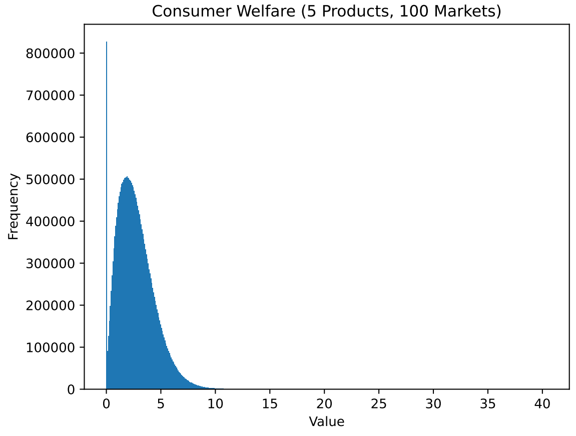
```
[10]: plot_two_histograms(df_3p['Pi'].values, df_5p['Pi'].values, bins=50,␣
       ↪labels=('Profits per purchase (3 Products, 100 Markets)', 'Profits per␣
       ↪purchase (5 Products, 100 Markets)'), path="Results/profits_dist.pdf")
```



Plot consumer welfare

```
[11]: plot_two_histograms(consumer_welfare_3p, consumer_welfare_5p, labels=('Consumer␣
       ↪Welfare (3 Products, 100 Markets)', 'Consumer Welfare (5 Products, 100␣
       ↪Markets)'), path="Results/cs_dist.pdf")
```





Part 2.1: BLP and Hausman Instruments

```
[10]: # Set the number of firms, markets, and simulated consumers in each market
J = 3
M = 100
N = 500
# Set the number of BLP instruments
N_instruments = 7
```

Using X as regressors

```
[69]: df_3p['P_other_markets'] = df_3p.groupby('product_id')['P_opt'].transform(
        lambda x: (x.sum() - x) / (len(x) - 1)
    )
xi = df_3p['xi_all'].values.reshape(J*M, 1)
X = df_3p[['x1', 'x2', 'x3']].values.reshape(J*M, 3)
shares = df_3p['shares'].values.reshape(J*M, 1)
price = df_3p['P_opt'].values.reshape(J*M, 1)
price_other_mkt = df_3p['P_other_markets'].values.reshape(J*M, 1)
```

```
[12]: (xi*X).mean(axis=0)
```

```
[12]: array([0.04346104, 0.02074192, 0.0355315 ])
```

Hausman Instruments

```
[13]: (xi*price).mean()
```

```
[13]: np.float64(0.2949581371489729)
```

```
[14]: (xi*price_other_mkt).mean()
```

```
[14]: np.float64(0.1436795476746802)
```

Part 2.2: Demand-side estimation, 100 markets

```
[15]: out_demandside = estimate_BLP(df, alphas, sigma_alpha_init, mode, verbose_print␣
       ↪= 3, scale_delta_guess = 3)
```

```
#==============================================================================#
#== Solving the MPEC optimization routine
#==============================================================================#
| niter |f evals|CG iter|  obj func   |tr radius |   opt    |  c viol  | penalty
|CG stop|
|-------|-------|-------|-------------|----------|----------|----------|--------
--|-------|
|   1   |   1   |   0   | +7.0000e+00 | 1.00e+00 | 2.32e-04 | 1.00e+00 |
1.00e+00 |   0   |
|   2   |   2   |   1   | +3.5180e+00 | 5.60e+00 | 1.54e-04 | 7.09e-01 |
1.00e+00 |   1   |
|   3   |   3   |   2   | +1.5079e-02 | 3.14e+01 | 6.02e-06 | 3.48e-01 |
1.74e+01 |   1   |
...
|  154  |  226  |  101  | +7.8585e-03 | 6.54e-21 | 4.68e-14 | 1.67e-16 |
2.75e+04 |   4   |

`xtol` termination condition is satisfied.
Number of iterations: 154, function evaluations: 226, CG iterations: 101,
optimality: 4.68e-14, constraint violation: 1.67e-16, execution time: 2.1e+01 s.
Optimal solution found.
#==============================================================================#
#== Optimal parameters found. Next, calculating standard errors:
#== Calculating standard errors, elasticities, profits, and consumer surplus
#==============================================================================#
#==============================================================================#
#== BLP Estimation Complete
#==============================================================================#
```

```
[ ]: store_results(out_demandside, "Results/q2_demandside_100markets.xlsx")
     store_results(out_demandside, "Results/q2_demandside_100markets.tex")
     save_mc_to_excel(out_demandside, file_name="Results/
      ↪q2_demandside_100markets_profits.xlsx", mode = "profits")
```

Part 2.2g: Demand-side instruments, 10 markets

Load the data and set function arguments:

```
[19]: df_3p10m, alphas_3p10m = load_mat_data('Simulation Data/10markets3products.mat',␣
      ↪3, 10)
      #### Function arguments
      mode = "demand_side" #demand-side moments only
```

```
[20]: out_demandside_10m = estimate_BLP(df_3p10m, alphas_3p10m, sigma_alpha_init,␣
      ↪mode, verbose_print = 1, scale_delta_guess = 3, max_iter=10000)
```

```
#==============================================================================#
#== Solving the MPEC optimization routine
#==============================================================================#
The maximum number of function evaluations is exceeded.
Number of iterations: 10000, function evaluations: 10000, CG iterations: 9999,
optimality: 5.78e-05, constraint violation: 5.64e-12, execution time: 4.1e+01 s.
Optimization failed: The maximum number of function evaluations is exceeded.
#==============================================================================#
#== Optimal parameters found. Next, calculating standard errors:
#== Calculating standard errors, elasticities, profits, and consumer surplus
#==============================================================================#
#==============================================================================#
#== BLP Estimation Complete
#==============================================================================#
```

```
[59]: store_results(out_demandside_10m, "Results/
      ↪q2_demandside_10markets_delta_equals_3.xlsx")
      store_results(out_demandside_10m, "Results/
      ↪q2_demandside_10markets_delta_equals_3.tex")
```

```
Excel file saved to Results/q2_demandside_10markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_3.xlsx
TeX file saved to Results/q2_demandside_10markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_3.tex
```

```
[60]: out_demandside_10m_v2=estimate_BLP(df_3p10m, alphas_3p10m, sigma_alpha_init,␣
      ↪mode, verbose_print = 1, scale_delta_guess = 1, max_iter=10000)
```

```
#==============================================================================#
```

```
#== Solving the MPEC optimization routine
#==============================================================================#
The maximum number of function evaluations is exceeded.
Number of iterations: 10000, function evaluations: 10000, CG iterations: 9999,
optimality: 4.67e-05, constraint violation: 1.01e-11, execution time: 1.1e+02 s.
Optimization failed: The maximum number of function evaluations is exceeded.
#==============================================================================#
#== Optimal parameters found. Next, calculating standard errors:
#== Calculating standard errors, elasticities, profits, and consumer surplus
#==============================================================================#
#==============================================================================#
#== BLP Estimation Complete
#==============================================================================#
```

[61]:
```
store_results(out_demandside_10m_v2, "Results/
↪q2_demandside_10markets_delta_equals_1.xlsx")
store_results(out_demandside_10m_v2, "Results/
↪q2_demandside_10markets_delta_equals_1.tex")
```

```
Excel file saved to Results/q2_demandside_10markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_10markets_delta_equals_1.xlsx
TeX file saved to Results/q2_demandside_10markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_10markets_delta_equals_1.tex
```

Part 2.3: Using price as a moment.

Estimate θ assming incorrectly that $E[\xi|p] = 0$ within each market.

Load the data and set function arguments:

[63]:
```
out_demandside_pmoment_delta1 = estimate_BLP(df, alphas, sigma_alpha_init,␣
↪"p_exercise", verbose_print = 1, scale_delta_guess = 1, max_iter=10000)
out_demandside_pmoment_delta3 = estimate_BLP(df, alphas, sigma_alpha_init,␣
↪"p_exercise", verbose_print = 1, scale_delta_guess = 3, max_iter=10000)
```

```
#==============================================================================#
#== Solving the MPEC optimization routine
```

```
#==============================================================================#
`xtol` termination condition is satisfied.
Number of iterations: 96, function evaluations: 81, CG iterations: 0,
optimality: 3.14e-31, constraint violation: 1.11e-16, execution time: 2.3e+01 s.
Optimal solution found.
#==============================================================================#
#== Optimal parameters found. Next, calculating standard errors:
#== Calculating standard errors, elasticities, profits, and consumer surplus
#==============================================================================#
#==============================================================================#
#== BLP Estimation Complete
#==============================================================================#
#==============================================================================#
#== Solving the MPEC optimization routine
#==============================================================================#
`xtol` termination condition is satisfied.
Number of iterations: 63, function evaluations: 48, CG iterations: 0,
optimality: 6.00e-31, constraint violation: 1.11e-16, execution time: 1.4e+01 s.
Optimal solution found.
#==============================================================================#
#== Optimal parameters found. Next, calculating standard errors:
#== Calculating standard errors, elasticities, profits, and consumer surplus
#==============================================================================#
#==============================================================================#
#== BLP Estimation Complete
#==============================================================================#
```

[64]:
```
store_results(out_demandside_pmoment_delta1, "Results/
 ↪q2_demandside_pmoment_100markets_delta_equals_1.xlsx")
store_results(out_demandside_pmoment_delta1, "Results/
 ↪q2_demandside_pmoment_100markets_delta_equals_1.tex")
store_results(out_demandside_pmoment_delta3, "Results/
 ↪q2_demandside_pmoment_100markets_delta_equals_3.xlsx")
store_results(out_demandside_pmoment_delta3, "Results/
 ↪q2_demandside_pmoment_100markets_delta_equals_3.tex")
```

```
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.xlsx
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_1.tex
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.xlsx
Excel file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.xlsx
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.tex
TeX file saved to Results/q2_demandside_pmoment_100markets_delta_equals_3.tex
```

Part 3.1: Using the cost shifter as a moment

Estimate assuming $E[\xi|X, w^{\text{cost}}] = 0$.

Load the data and set function arguments:

[66]:
```
out_demandside_wcost = estimate_BLP(df, alphas, sigma_alpha_init, "supply_W",␣
↪verbose_print=1)
```

```
#==============================================================================#
#== Solving the MPEC optimization routine
#==============================================================================#
`xtol` termination condition is satisfied.
Number of iterations: 6144, function evaluations: 6123, CG iterations: 6088,
optimality: 5.35e-14, constraint violation: 1.67e-16, execution time: 1.7e+03 s.
Optimal solution found.
#==============================================================================#
#== Optimal parameters found. Next, calculating standard errors:
#== Calculating standard errors, elasticities, profits, and consumer surplus
#==============================================================================#
#==============================================================================#
#== BLP Estimation Complete
#==============================================================================#
```

[107]:
```
store_results(out_demandside_wcost, "Results/q3_demandside_wcost_100markets.
↪xlsx")
store_results(out_demandside_wcost, "Results/q3_demandside_wcost_100markets.tex")
```

```
save_mc_to_excel(out_demandside_wcost, file_name="Results/
 ↪q3_demandside_wcost_100markets_profits.xlsx", mode = "profits")
```

```
Excel file saved to Results/q3_demandside_wcost_100markets.xlsx
Excel file saved to Results/q3_demandside_wcost_100markets.xlsx
Excel file saved to Results/q3_demandside_wcost_100markets.xlsx
Excel file saved to Results/q3_demandside_wcost_100markets.xlsx
Excel file saved to Results/q3_demandside_wcost_100markets.xlsx
Excel file saved to Results/q3_demandside_wcost_100markets.xlsx
Excel file saved to Results/q3_demandside_wcost_100markets.xlsx
Excel file saved to Results/q3_demandside_wcost_100markets.xlsx
TeX file saved to Results/q3_demandside_wcost_100markets.tex
TeX file saved to Results/q3_demandside_wcost_100markets.tex
TeX file saved to Results/q3_demandside_wcost_100markets.tex
TeX file saved to Results/q3_demandside_wcost_100markets.tex
TeX file saved to Results/q3_demandside_wcost_100markets.tex
TeX file saved to Results/q3_demandside_wcost_100markets.tex
TeX file saved to Results/q3_demandside_wcost_100markets.tex
TeX file saved to Results/q3_demandside_wcost_100markets.tex
Comparison file saved to Results/q3_demandside_wcost_100markets_profits.xlsx.
```

```
[68]: out_demandside_wcost_10m = estimate_BLP(df_3p10m, alphas_3p10m,␣
 ↪sigma_alpha_init, "supply_W", scale_delta_guess=3, verbose_print=1)
```

```
#==========================================================================#
#== Solving the MPEC optimization routine
#==========================================================================#
The maximum number of function evaluations is exceeded.
Number of iterations: 10000, function evaluations: 10000, CG iterations: 9999,
optimality: 9.15e-05, constraint violation: 1.34e-11, execution time: 1.1e+02 s.
Optimization failed: The maximum number of function evaluations is exceeded.
#==========================================================================#
#== Optimal parameters found. Next, calculating standard errors:
#== Calculating standard errors, elasticities, profits, and consumer surplus
#==========================================================================#
#==========================================================================#
#== BLP Estimation Complete
#==========================================================================#
```

```
[69]: store_results(out_demandside_wcost_10m, "Results/q3_demandside_wcost_10markets.
 ↪xlsx")
store_results(out_demandside_wcost_10m, "Results/q3_demandside_wcost_10markets.
 ↪tex")
```

```
Excel file saved to Results/q3_demandside_wcost_10markets.xlsx
Excel file saved to Results/q3_demandside_wcost_10markets.xlsx
Excel file saved to Results/q3_demandside_wcost_10markets.xlsx
Excel file saved to Results/q3_demandside_wcost_10markets.xlsx
Excel file saved to Results/q3_demandside_wcost_10markets.xlsx
Excel file saved to Results/q3_demandside_wcost_10markets.xlsx
Excel file saved to Results/q3_demandside_wcost_10markets.xlsx
Excel file saved to Results/q3_demandside_wcost_10markets.xlsx
TeX file saved to Results/q3_demandside_wcost_10markets.tex
TeX file saved to Results/q3_demandside_wcost_10markets.tex
TeX file saved to Results/q3_demandside_wcost_10markets.tex
TeX file saved to Results/q3_demandside_wcost_10markets.tex
TeX file saved to Results/q3_demandside_wcost_10markets.tex
TeX file saved to Results/q3_demandside_wcost_10markets.tex
TeX file saved to Results/q3_demandside_wcost_10markets.tex
TeX file saved to Results/q3_demandside_wcost_10markets.tex
```

Joint estimation

Oligopoly (true) case

[21]:
```
out_joint_oligopoly = estimate_BLP(df, alphas, sigma_alpha_init, "supply_joint",␣
↪verbose_print=3, scale_delta_guess=3)
```

```
#==============================================================================#
#== Solving the MPEC optimization routine
#==============================================================================#
| niter |f evals|CG iter|  obj func   |tr radius |   opt    |  c viol  | penalty
|CG stop|
|-------|-------|-------|-------------|----------|----------|----------|--------
--|-------|
|   1   |   1   |   0   | +1.1000e+01 | 1.00e+00 | 2.87e-04 | 1.00e+00 |
1.00e+00 |   0   |
|   2   |   2   |   1   | +6.4326e+00 | 5.60e+00 | 1.98e-04 | 7.65e-01 |
1.00e+00 |   1   |
|   3   |   3   |   2   | +1.8552e-02 | 3.92e+01 | 3.99e-05 | 3.57e-01 |
1.00e+00 |   2   |
...
|  62   |  48   |  17   | +3.2076e-03 | 1.18e-20 | 4.81e-14 | 2.22e-16 |
5.40e+00 |   4   |
|  63   |  49   |  17   | +3.2076e-03 | 5.88e-21 | 4.81e-14 | 2.22e-16 |
5.40e+00 |   4   |

`xtol` termination condition is satisfied.
Number of iterations: 63, function evaluations: 49, CG iterations: 17,
optimality: 4.81e-14, constraint violation: 2.22e-16, execution time: 2.5e+01 s.
Optimal solution found.
#==============================================================================#
#== Optimal parameters found. Next, calculating standard errors:
#== Calculating standard errors, elasticities, profits, and consumer surplus
#==============================================================================#
#==============================================================================#
```

```
#== BLP Estimation Complete
#==============================================================================#
```

```
[72]: store_results(out_joint_oligopoly, "Results/q3_joint_oligopoly_100markets.xlsx")
      store_results(out_joint_oligopoly, "Results/q3_joint_oligopoly_100markets.tex")

      save_mc_to_excel(out_joint_oligopoly, file_name="Results/q3_mc_oligopoly.xlsx")
```

```
Excel file saved to Results/q3_joint_oligopoly_100markets.xlsx
Excel file saved to Results/q3_joint_oligopoly_100markets.xlsx
Excel file saved to Results/q3_joint_oligopoly_100markets.xlsx
Excel file saved to Results/q3_joint_oligopoly_100markets.xlsx
Excel file saved to Results/q3_joint_oligopoly_100markets.xlsx
Excel file saved to Results/q3_joint_oligopoly_100markets.xlsx
Excel file saved to Results/q3_joint_oligopoly_100markets.xlsx
Excel file saved to Results/q3_joint_oligopoly_100markets.xlsx
TeX file saved to Results/q3_joint_oligopoly_100markets.tex
TeX file saved to Results/q3_joint_oligopoly_100markets.tex
TeX file saved to Results/q3_joint_oligopoly_100markets.tex
TeX file saved to Results/q3_joint_oligopoly_100markets.tex
TeX file saved to Results/q3_joint_oligopoly_100markets.tex
TeX file saved to Results/q3_joint_oligopoly_100markets.tex
TeX file saved to Results/q3_joint_oligopoly_100markets.tex
TeX file saved to Results/q3_joint_oligopoly_100markets.tex
Comparison file saved to Results/q3_mc_oligopoly.xlsx.
```

Collusion case

```
[74]: out_joint_collusion = estimate_BLP(df, alphas, sigma_alpha_init, "supply_joint",␣
       ↪conduct = "collusion", verbose_print=3, scale_delta_guess=3)
```

```
#==============================================================================#
#== Solving the MPEC optimization routine
#==============================================================================#
| niter |f evals|CG iter|  obj func   |tr radius |   opt    |  c viol  | penalty
|CG stop|
|-------|-------|-------|-------------|----------|----------|----------|--------
--|-------|
|   1   |   1   |   0   | +1.1000e+01 | 1.00e+00 | 2.38e-04 | 1.00e+00 |
1.00e+00 |   0   |
|   2   |   2   |   1   | +6.4314e+00 | 5.60e+00 | 1.72e-04 | 7.65e-01 |
1.00e+00 |   1   |
|   3   |   3   |   2   | +1.2285e-02 | 3.92e+01 | 1.30e-05 | 3.56e-01 |
1.00e+00 |   2   |
...
|  70   |  56   |  17   | +8.3887e-03 | 6.18e-21 | 4.01e-14 | 1.11e-16 |
4.75e+00 |   4   |

`xtol` termination condition is satisfied.
```

```
Number of iterations: 70, function evaluations: 56, CG iterations: 17,
optimality: 4.01e-14, constraint violation: 1.11e-16, execution time: 6.5e+01 s.
Optimal solution found.
#==============================================================================#
#== Optimal parameters found. Next, calculating standard errors:
#== Calculating standard errors, elasticities, profits, and consumer surplus
#==============================================================================#
#==============================================================================#
#== BLP Estimation Complete
#==============================================================================#
```

[75]:
```
store_results(out_joint_collusion, "Results/q3_joint_collusion_100markets.xlsx")
store_results(out_joint_collusion, "Results/q3_joint_collusion_100markets.tex")

save_mc_to_excel(out_joint_collusion, file_name="Results/q3_mc_collusion.xlsx")
```

```
Excel file saved to Results/q3_joint_collusion_100markets.xlsx
Excel file saved to Results/q3_joint_collusion_100markets.xlsx
Excel file saved to Results/q3_joint_collusion_100markets.xlsx
Excel file saved to Results/q3_joint_collusion_100markets.xlsx
Excel file saved to Results/q3_joint_collusion_100markets.xlsx
Excel file saved to Results/q3_joint_collusion_100markets.xlsx
Excel file saved to Results/q3_joint_collusion_100markets.xlsx
Excel file saved to Results/q3_joint_collusion_100markets.xlsx
TeX file saved to Results/q3_joint_collusion_100markets.tex
TeX file saved to Results/q3_joint_collusion_100markets.tex
TeX file saved to Results/q3_joint_collusion_100markets.tex
TeX file saved to Results/q3_joint_collusion_100markets.tex
TeX file saved to Results/q3_joint_collusion_100markets.tex
TeX file saved to Results/q3_joint_collusion_100markets.tex
TeX file saved to Results/q3_joint_collusion_100markets.tex
TeX file saved to Results/q3_joint_collusion_100markets.tex
Comparison file saved to Results/q3_mc_collusion.xlsx.
```

[76]:
```
out_joint_perfect = estimate_BLP(df, alphas, sigma_alpha_init, "supply_joint",␣
 ↪conduct = "perfect", verbose_print=3, scale_delta_guess=3)
```

```
#==============================================================================#
#== Solving the MPEC optimization routine
#==============================================================================#
| niter |f evals|CG iter|  obj func   |tr radius |   opt    |  c viol  | penalty
|CG stop|
|-------|-------|-------|-------------|----------|----------|----------|--------
--|-------|
|   1   |   1   |   0   | +1.1000e+01 | 1.00e+00 | 2.38e-04 | 1.00e+00 |
1.00e+00 |   0   |
|   2   |   2   |   1   | +6.4314e+00 | 5.60e+00 | 1.72e-04 | 7.65e-01 |
1.00e+00 |   1   |
|   3   |   3   |   2   | +1.1978e-02 | 3.14e+01 | 1.16e-05 | 3.57e-01 |
6.76e+00 |    1    |
|   4    |   4    |   3    | +5.0880e-01 | 1.26e+02 | 1.93e-03 | 6.09e-02 |
6.76e+00 |    1    |
...
|  193   |  286   |  129   | +8.3887e-03 | 5.66e-21 | 4.25e-14 | 2.22e-16 |
2.73e+04 |    4    |

`xtol` termination condition is satisfied.
Number of iterations: 193, function evaluations: 286, CG iterations: 129,
optimality: 4.25e-14, constraint violation: 2.22e-16, execution time: 8.2e+01 s.
Optimal solution found.
#==============================================================================#
#== Optimal parameters found. Next, calculating standard errors:
#== Calculating standard errors, elasticities, profits, and consumer surplus
#==============================================================================#
#==============================================================================#
#== BLP Estimation Complete
#==============================================================================#
```

[77]:
```
store_results(out_joint_perfect, "Results/q3_joint_perfect_100markets.xlsx")
store_results(out_joint_perfect, "Results/q3_joint_perfect_100markets.tex")

save_mc_to_excel(out_joint_perfect, file_name="Results/q3_mc_perfect.xlsx")
```

```
Excel file saved to Results/q3_joint_perfect_100markets.xlsx
Excel file saved to Results/q3_joint_perfect_100markets.xlsx
Excel file saved to Results/q3_joint_perfect_100markets.xlsx
Excel file saved to Results/q3_joint_perfect_100markets.xlsx
Excel file saved to Results/q3_joint_perfect_100markets.xlsx
Excel file saved to Results/q3_joint_perfect_100markets.xlsx
Excel file saved to Results/q3_joint_perfect_100markets.xlsx
Excel file saved to Results/q3_joint_perfect_100markets.xlsx
TeX file saved to Results/q3_joint_perfect_100markets.tex
TeX file saved to Results/q3_joint_perfect_100markets.tex
TeX file saved to Results/q3_joint_perfect_100markets.tex
TeX file saved to Results/q3_joint_perfect_100markets.tex
TeX file saved to Results/q3_joint_perfect_100markets.tex
TeX file saved to Results/q3_joint_perfect_100markets.tex
TeX file saved to Results/q3_joint_perfect_100markets.tex
TeX file saved to Results/q3_joint_perfect_100markets.tex
Comparison file saved to Results/q3_mc_perfect.xlsx.
```

Bonus: Merger analysis

Use the parameter estimates from the join estimation case (discussion in the answer pdf)

[96]:
```python
mc = out_joint_oligopoly['mc']['hat'].reshape(-1, 1)
betas = out_joint_oligopoly['beta_hat']
alpha = out_joint_oligopoly['alpha_hat']
sigma_alpha = out_joint_oligopoly['sigma_alpha_hat']
xi = out_joint_oligopoly['xi_hat']
```

[97]:
```python
MJN = (100, 3, 8, 500)
```

[98]:
```python
ownership_merger = np.array(
    [
        [1.0, 1.0, 0.0],
        [1.0, 1.0, 0.0],
        [0.0, 0.0, 1.0]
    ]
)
```

Prices after the merger

[99]:
```python
merger_prices, merger_shares = predict_prices_and_shares(ownership_merger, mc,␣
 ↪betas, alpha, sigma_alpha, xi, X, MJN)
```

```
Iteration 0. Norm: 3.2491641698801064.
Iteration 1. Norm: 1.2649923295137064.
Iteration 2. Norm: 1.2665241527758957.
Iteration 3. Norm: 1.2574030657234723.
...
Iteration 34. Norm: 1.9784320239898597e-08.
Iteration 35. Norm: 9.913446950172242e-09.
Contraction mapping converged, found prices that satisfy the FOC.
Iterations: 35
```

[100]:
```python
# Reshape to match the dimensions of what's required for prices and shares
merger_prices = merger_prices.reshape(-1, 1)
merger_shares = merger_shares.reshape(-1, 1)
```

Markup comparison

[101]:
```python
# Define indices for each product
idx_p1 = [i for i in range(0, 300, 3)]
idx_p2 = [i for i in range(1, 300, 3)]
idx_p3 = [i for i in range(2, 300, 3)]
```

[102]:
```python
markup_pre = price - mc
markup_post = merger_prices - mc
```

[103]:
```python
print("Mean markup pre-merger, product 1:", markup_pre[idx_p1, :].mean())
print("Mean markup pre-merger, product 2:", markup_pre[idx_p2, :].mean())
print("Mean markup pre-merger, product 3:", markup_pre[idx_p3, :].mean())
```

```
Mean markup pre-merger, product 1: 0.8112508888765517
Mean markup pre-merger, product 2: 1.2230171901026956
Mean markup pre-merger, product 3: 0.9517142736119993
```

```
[104]: print("Mean markup post-merger, product 1:", markup_post[idx_p1, :].mean())
print("Mean markup post-merger, product 2:", markup_post[idx_p2, :].mean())
print("Mean markup post-merger, product 3:", markup_post[idx_p3, :].mean())
```

```
Mean markup post-merger, product 1: 1.6175575459997558
Mean markup post-merger, product 2: 1.4382918135071117
Mean markup post-merger, product 3: 0.9748140644645866
```

Consumer surplus comparison

```
[105]: cs_pre = calculate_consumer_surplus(betas, alpha, sigma_alpha, xi, X, price, MJN)
cs_post = calculate_consumer_surplus(betas, alpha, sigma_alpha, xi, X,␣
 ↪merger_prices, MJN)
```

```
[106]: print("Mean CS pre-merger accross all markets:", cs_pre.mean())
```

```
Mean CS pre-merger accross all markets: 1.338376498397264
```

```
[107]: print("Mean CS post-merger accross all markets:", cs_post.mean())
```

```
Mean CS post-merger accross all markets: 1.249033370252179
```

Prices comparison

```
[108]: print("Mean price pre-merger, product 1:", price[idx_p1, :].mean())
print("Mean price pre-merger, product 2:", price[idx_p2, :].mean())
print("Mean price pre-merger, product 3:", price[idx_p3, :].mean())
```

```
Mean price pre-merger, product 1: 3.885773837354908
Mean price pre-merger, product 2: 2.4579350030224396
Mean price pre-merger, product 3: 2.735937439635178
```

```
[109]: print("Mean price post-merger, product 1:", merger_prices[idx_p1, :].mean())
print("Mean price post-merger, product 2:", merger_prices[idx_p2, :].mean())
print("Mean price post-merger, product 3:", merger_prices[idx_p3, :].mean())
```

```
Mean price post-merger, product 1: 4.692080494478112
Mean price post-merger, product 2: 2.6732096264268557
Mean price post-merger, product 3: 2.7590372304877655
```

Profit comparison

```
[110]: profit_pre = markup_pre * shares
profit_post = markup_post * merger_shares
```

```
[111]: print("Mean profit pre-merger, product 1:", profit_pre[idx_p1, :].mean())
print("Mean profit pre-merger, product 2:", profit_pre[idx_p2, :].mean())
print("Mean profit pre-merger, product 3:", profit_pre[idx_p3, :].mean())
```

```
Mean profit pre-merger, product 1: 0.10133274803280334
Mean profit pre-merger, product 2: 0.5475679509416372
Mean profit pre-merger, product 3: 0.2935782041286019
```

```
[112]: print("Mean profit post-merger, product 1:", profit_post[idx_p1, :].mean())
       print("Mean profit post-merger, product 2:", profit_post[idx_p2, :].mean())
       print("Mean profit post-merger, product 3:", profit_post[idx_p3, :].mean())
```

```
Mean profit post-merger, product 1: 0.10989816955150225
Mean profit post-merger, product 2: 0.5695504689753006
Mean profit post-merger, product 3: 0.3099750551384529
```

B.1 Estimation of BLP

```
# -*- coding: utf-8 -*-
"""
Function to estimate BLP
"""

from source_functions import *

def estimate_BLP(df, alphas, sigma_alpha_init, mode, conduct =
↪  "oligopoly", verbose_print = 1, scale_delta_guess = 1,
↪  delta_solve_init = False, max_iter = 10000, beta_true = (5, 1, 1),
↪  gamma_true = (2, 1, 1), alpha_true=1, sigma_alpha_true=1):

    ###
    J=3
    M = alphas.shape[0] #Number of markets
    N = alphas.shape[1] #Number of consumers



    ### Choose conduct
    if conduct == "oligopoly":
        ownership = jnp.eye(J)
    elif conduct == "collusion":
        ownership = jnp.ones((J, J))
    elif conduct == "perfect":
        ownership = jnp.zeros((J, J)) #Never used, just to avoid errors.
         ↪  In perfect competition, price=marginal cost
    else:
        raise Exception("Invalid conduct choice.")


    ### Choose mode of estimation run
    if mode == "demand_side":
        N_instruments = 7
    elif mode == "p_exercise":
        N_instruments = 4
    elif mode == "supply_W":
        N_instruments = 8
    elif mode == "supply_joint":
        N_instruments = 11
```

```
#Initial guesses for eta and delta
eta_init =  jnp.ones((N_instruments, 1))
delta_init = jnp.ones((J*M, 1))  * scale_delta_guess ### For trying
↪  different initial guesses of delta to test stability

#Initial guess for all parameters
params_init =
↪  jnp.concatenate([np.array([sigma_alpha_init]).reshape(-1,1),
↪  eta_init, delta_init], axis=0).flatten()

MJN = (M, J, N_instruments, N) #Stack all shape-related parameters
↪  together

#========== Constructing correct values of parameters, for use later
↪  ===============================================================================#
deltas_correct = (5 + df['x2'] + df['x3'] - df['P_opt'] +
↪  df['xi_all']).values.reshape(J*M, -1)
# Prepend the new element
first_elems = jnp.ones(1+N_instruments).reshape(-1, 1)
params_correct = jnp.vstack([first_elems, deltas_correct]).reshape(-1)

xi_true = (df['xi_all']).values.reshape(J*M, -1)

#========== Constructing commonly used variables
↪  ================================================================================
X = jnp.array(df[['x1', 'x2', 'x3']].values) #Matrix of product
↪  characteristics
prices = jnp.array(df[['P_opt']].values)
shares = jnp.array(df[['shares']].values)
W_costs = jnp.array(df[['w']].values)
Z_costs = jnp.array(df[['Z']].values)
omegas_true = np.array(df[['eta']].values)


# For speed, compute this outside of the function and pass it later
alphas_repeat = jnp.repeat(np.array(alphas.values), repeats=J, axis=0)

# Random coefficients nu on the prices
nus = (np.array(alphas.values)-1)
nus_on_prices = (alphas_repeat-1) * prices.reshape(-1, 1)

# Get matrix of regressors
Xbar = df[['x1', 'x2', 'x3', 'P_opt']].values.reshape(J*M, 4)
```

```
# Get the matrix of instruments (including x1, x2, and the BLP
↪  moments).
Z_everything = jnp.array(blp_instruments_all(X, W_costs, Z_costs,
↪  prices, MJN))

if mode == "demand_side":
    Z = Z_everything[:, 0:7]
elif mode == "p_exercise":
    Z = Z_everything[:, [0, 1, 2, 8]] #Includes x1, x2, BLP moments in
    ↪  X, and p
elif mode == "supply_W":
    Z = Z_everything[:, 0:8] #Includes x1, x2, BLP moments in X, and
    ↪  w^cost
elif mode == "supply_joint":
    Z = Z_everything[:, 0:8] #Includes x1, x2, BLP moments in X, and
    ↪  w^cost

#Projection matrix onto the instruments
Pz = Z @ jnp.linalg.inv(Z.T @ Z) @ Z.T
#Annihiliator matrix to get xi from delta
Az = jnp.eye(Pz.shape[0]) - Xbar @ jnp.linalg.inv(Xbar.T @ Pz @ Xbar)
↪  @ Xbar.T @ Pz
#This thing gets the value of beta.
M_iv_est = (np.linalg.inv(Xbar.T @ Pz @ Xbar) @ Xbar.T @ Pz)
# GMM weighting matrix
W = np.eye(N_instruments)

# Joint estimation, supply-side case.
Xs = Z_everything[:, [0, 7, 9]] #### Keep vector of ones, W_cost, and
↪  Z_cost.

As = jnp.eye(Xs.shape[0]) - Xs @ jnp.linalg.inv(Xs.T@Xs) @ Xs.T

if mode == "supply_joint":
    constraint_g_joint_jac = jacobian(constraint_g_joint)


if delta_solve_init:
    #### Solve initial deltas for improved initial guess.

    ↪  print("#=============================================================
    print("#== Getting feasible initial guess using modified MPEC")

    ↪  print("#=============================================================
    res = solve_init_deltas(params_init, shares, nus_on_prices, MJN)

    ### Replace the deltas with the new ones
    params_init0 = params_init
    params_init = res.x
    #params_init[1+N_instruments:] = res.x


#========== Define the constraints for the MPEC
↪   ================================================================================
if mode == "supply_joint":
    constraints = [
        {
            'type': 'eq',  # Equality constraint g(x) = eta
            'fun': lambda x: np.asarray(constraint_g_joint(x, X, Z,
            ↪  Az, M_iv_est, Xs, As, prices, shares, nus,
            ↪  nus_on_prices, MJN, ownership)),  # Convert to NumPy
            'jac': lambda x: np.asarray(constraint_g_joint_jac(x, X,
            ↪  Z, Az, M_iv_est, Xs, As, prices, shares, nus,
            ↪  nus_on_prices, MJN, ownership))  # Convert Jacobian to
            ↪  NumPy
        },
        {
            'type': 'eq',  # Equality constraint s(x) = shares
            'fun': lambda x: np.asarray(constraint_s(x, shares,
            ↪  nus_on_prices, MJN)),
            'jac': lambda x: np.asarray(constraint_s_jac(x, shares,
            ↪  nus_on_prices, MJN))
        }
    ]
else:
    constraints = [
        {
            'type': 'eq',  # Equality constraint g(x) = eta
            'fun': lambda x: np.asarray(constraint_g(x, Z, Az, MJN)),
            ↪  # Convert to NumPy
            'jac': lambda x: np.asarray(constraint_g_jac(x, Z, Az,
            ↪  MJN))  # Convert Jacobian to NumPy
        },
        {
            'type': 'eq',  # Equality constraint s(x) = shares
            'fun': lambda x: np.asarray(constraint_s(x, shares,
            ↪  nus_on_prices, MJN)),
            'jac': lambda x: np.asarray(constraint_s_jac(x, shares,
             ↪  nus_on_prices, MJN))
        }
    ]


#========== Running the MPEC optimization routine
 ↪  ================================================================================


 ↪  print("#================================================================================
print("#== Solving the MPEC optimization routine")

 ↪  print("#================================================================================


# Silence warning about delta_grad==0.0
warnings.filterwarnings("ignore", message="delta_grad == 0.0. Check if
 ↪  the approximated function is linear.")

# Solve optimization
tolerance = 1e-20
W = np.eye(N_instruments)

result = minimize(
    fun = lambda x: objective_mpec(x, W, MJN),
    x0 = params_init,
    constraints = constraints,
    method = 'trust-constr',
    jac = lambda x: objective_jac(x, W, MJN),
    hess = lambda x: objective_hess(x, W, MJN),
    options = {
             'xtol': tolerance,
             'gtol': tolerance,
             'barrier_tol': tolerance,
             'sparse_jacobian': True,
             'disp': True,
             'verbose': verbose_print,
             'maxiter':max_iter
            },
)

# Output results
if result.success:
    print("Optimal solution found.")
else:
    print("Optimization failed:", result.message)


#========== Calculate the parameters of interest
↪  ===================================================================
theta_hat = result.x # Get the estimated value of theta
delta_hat = theta_hat[N_instruments + 1:].reshape(J*M, 1)
beta_and_alpha_hat = np.array(M_iv_est @ delta_hat)
xi_hat = np.array(Az@delta_hat)

beta_hat = beta_and_alpha_hat[:3]
alpha_hat = -beta_and_alpha_hat[3].item()
sigma_alpha_hat = theta_hat[0].item()

#Calculate gamma if we're solving the supply-side joint version
if mode == "supply_joint":
    if conduct == "perfect":
        gamma_hat = (np.linalg.inv(Xs.T@Xs)@Xs.T)@prices
    else:
        ### Calculate gamma_hat
        mc = calculate_marginal_costs(theta_hat, ownership, xi_hat, X,
         ↪  M_iv_est, prices, nus, nus_on_prices, MJN)
        gamma_hat = (np.linalg.inv(Xs.T@Xs)@Xs.T)@mc
else:
    gamma_hat = np.array([np.nan, np.nan, np.nan])

    #elas_old = calculate_price_elasticity_old(beta_hat, alpha_hat,
    ↪  sigma_alpha_hat, xi_hat, X, prices, shares, nus, MJN)
    #elas_old=elas_old.flatten()
    #mc_old = calculate_marginal_costs(elas_old, "oligopoly", prices,
    ↪  shares, MJN)

#(beta_hat)
#print(alpha_hat)
#print(sigma_alpha_hat)


↪  print("#==================================================================
print("#== Optimal parameters found. Next, calculating standard
↪  errors:")
print("#== Calculating standard errors, elasticities, profits, and
↪  consumer surplus")
↪  print("#==================================================================
    
#========== Calculate the standard errors
↪  ===================================================================
if mode == "supply_joint":
    #se_sigma, se_betas = standard_errors_joint(theta_hat, Z, Az,
     ↪  M_iv_est, shares, nus_on_prices, MJN)
    se_sigma, se_betas, se_gamma = standard_errors_joint(theta_hat, X,
     ↪  Z, Az, M_iv_est, Xs, prices, shares, nus_on_prices, nus, MJN,
     ↪  ownership)
    se = np.concatenate([[se_sigma], se_betas, se_gamma])
else:
    se_sigma, se_betas = standard_errors(theta_hat, Z, Az, M_iv_est,
     ↪  shares, nus_on_prices, MJN)
    se = np.concatenate([[se_sigma], se_betas, gamma_hat])

#========== Calculate the elasticities
↪  ===================================================================
#Predicted deltas, xis, and moment conditions
beta_true_array = np.array(beta_true)
#True value of the elasticities
elasticities_true = calculate_price_elasticity(params_correct,
↪  xi_true, X, M_iv_est, prices, shares, nus, nus_on_prices, MJN)
#Mean of the true value of elasticities
mean_elasticities_true = elasticities_true.reshape(J, J,
↪  M).mean(axis=2)
#######
#Predicted value of the elasticities
elasticities_hat = calculate_price_elasticity(theta_hat, xi_hat, X,
↪  M_iv_est, prices, shares, nus, nus_on_prices, MJN)
#Mean of the true value of elasticities
mean_elasticities_hat = elasticities_hat.reshape(J, J, M).mean(axis=2)

#========== Calculate the marginal costs
↪  ===================================================================
#True marginal cost
gamma_true_array = np.array(gamma_true).reshape(-1, 1)



if conduct == "perfect":
    mc_true=prices
    mc_hat=prices
```

```
else:
    mc_true = (Xs@gamma_true_array + omegas_true).flatten()
    mc_hat = calculate_marginal_costs(theta_hat, ownership, xi_hat, X,
     ↪  M_iv_est, prices, nus, nus_on_prices, MJN).flatten()

mean_mc_true = mc_true.reshape(J, M).mean(axis=1)
mean_mc_hat = mc_hat.reshape(J, M).mean(axis=1)


#========== Calculate the profits per purchase
 ↪  ===========================================================================
profits_true = prices.flatten() - mc_true.flatten()
profits_hat = prices.flatten() - mc_hat.flatten()
mean_profits_true = profits_true.reshape(J, M).mean(axis=1)
mean_profits_hat = profits_hat.reshape(J, M).mean(axis=1)

#========== Calculate the consumer surplus
 ↪  ===========================================================================
cs_true = calculate_consumer_surplus(beta_true_array, alpha_true,
 ↪  sigma_alpha_true, xi_true, X, prices, MJN)
cs_hat = calculate_consumer_surplus(beta_hat, alpha_hat,
 ↪  sigma_alpha_hat, xi_hat, X, prices, MJN)

mean_cs_true = cs_true.mean()
mean_cs_hat = cs_hat.mean()



#========== Create a dictionary to return all output
 ↪  ===========================================================================
OUT = {
    'delta_hat': delta_hat,
    'xi_hat': xi_hat,
    'beta_hat': beta_hat,
    'alpha_hat': alpha_hat,
    'sigma_alpha_hat': sigma_alpha_hat,
    'gamma_hat': gamma_hat,
    'se': se,
    'se_names': ["sigma_alpha", "beta0", "beta1", "beta2", "alpha",
     ↪  "gamma0", "gamma1", "gamma2"],
    'elasticities': {'true': elasticities_true, 'hat':
     ↪  elasticities_hat},
    'mean_elasticities': {'true': mean_elasticities_true, 'hat':
     ↪  mean_elasticities_hat},
    'mc': {'true': mc_true, 'hat': mc_hat},
    'mean_mc': {'true': mean_mc_true, 'hat': mean_mc_hat},
    'profits': {'true': profits_true, 'hat': profits_hat},
    'mean_profits': {'true': mean_profits_true, 'hat':
     ↪  mean_profits_hat},
    'cs': {'true': cs_true, 'hat': cs_hat},
    'mean_cs': {'true': mean_cs_true, 'hat': mean_cs_hat},
    'optim_results': result
}


 ↪  print("#==========================================================================
print("#== BLP Estimation Complete")

 ↪  print("#==========================================================================


return OUT
```

B.2 Source Functions Used in Estimation and Other Parts

```
# -*- coding: utf-8 -*-
"""
Source functions
"""
import pandas as pd
import numpy as np
from scipy.io import loadmat  # this is the SciPy module that loads
 ↪  mat-files
from scipy.optimize import root, minimize
import matplotlib.pyplot as plt
from jax import grad, jacobian, hessian, config, jit, lax, jacfwd
import jax.numpy as jnp
import warnings
from functools import partial


config.update("jax_enable_x64", True)



#================================================================================#
# Arguments common among all functions
#================================================================================#
```

```
# params: parameter vector [sigma, etas, deltas]
# M: number of markets
# J: number of products per market
# N_instruments: number of instruments/moment conditions
# N: Number of consumers in the random draw.


#=====================================================================================#
# s: Function to calculate market shares
#=====================================================================================#
#Arguments:
# nus_on_prices: The random component of utility from prices.
@partial(jit, static_argnums=(2,))
def s(params, nus_on_prices, MJN):

    M, J, N_instruments, N = MJN
    sigma = params[0]

    # Use lax.dynamic_slice for dynamic slicing
    deltas_start = N_instruments + 1
    deltas = lax.dynamic_slice(params, (deltas_start,), (params.shape[0] -
    ↪  deltas_start,)).reshape(-1, 1)

    # Compute utilities
    utilities = deltas - sigma * nus_on_prices  # Shape: (M*J, N)

    # Reshape utilities for markets and products
    utilities_reshaped = utilities.reshape(M, J, N)  # Shape: (M, J, N)

    # Compute the stabilization constant (max utility per market per
    ↪  individual)
    max_utilities = jnp.max(utilities_reshaped, axis=1, keepdims=True)  #
    ↪  Shape: (M, 1, N)

    # Stabilized exponentials
    exp_utilities = jnp.exp(utilities_reshaped - max_utilities)  # Shape:
    ↪  (M, J, N)

    # Adjust the "outside option" (1 becomes exp(-max_utilities))
    outside_option = jnp.exp(-max_utilities)  # Shape: (M, 1, N)

    # Compute the stabilized denominator
    sum_exp_utilities = outside_option + exp_utilities.sum(axis=1,
    ↪  keepdims=True)  # Shape: (M, 1, N)
    # Compute shares
    shares = exp_utilities / sum_exp_utilities  # Shape: (M, J, N)

    # Average across individuals
    avg_shares = shares.mean(axis=2)  # Shape: (M, J)

    # Flatten output to match the original function's shape
    return avg_shares.flatten()

#================================================================================#
# solve_init_deltas: function to get a feasible initial guess for deltas.
↪
#================================================================================#
# shares: the vector of true market shares
def solve_init_deltas(params, shares, nus_on_prices, MJN):

    M, J, N_instruments, N = MJN

    constraint_func = lambda x: (s(x, nus_on_prices, MJN).reshape(J*M, -1)
    ↪  - shares).reshape(-1)
    constraint_jac = jacobian(constraint_func)

    # Define the constraint dictionary
    constraints = {
        'type': 'eq',
        'fun': constraint_func,
        'jac': constraint_jac
    }

    # Perform optimization
    result = minimize(
        fun=lambda x: 0,
        x0=params,
        method='SLSQP',
        constraints=constraints
    )

    # Return results
    if result.success:
        return result
    else:
        print("Optimization failed:", result.message)
        print("Returning the original passed parameters.")
        return params

#==============================================================================#
# blp_instruments_all: gets matrix of instruments used in moment
↪   conditions
#==============================================================================#
def blp_instruments_all(X, W_costs, Z_costs, prices, MJN):
    """
    Computes the matrix of instruments for all (j, m) pairs in a
    ↪   vectorized manner.

    Parameters:
    ----------
    X : jnp.ndarray
        Input matrix of shape (J * M, features).

    Returns:
    -------
    instruments : jnp.ndarray
        Matrix of instruments for all (j, m), shape (J * M, 6).
    """
    M, J, _, N = MJN

    # Reshape X into (M, J, features)
    # Note: for instruments, use only the nonconstant product
    ↪   characteristics.
    X_reshaped = X[:, 1:].reshape(M, J, -1)  # Shape: (M, J, features)

    # First two elements: Features of product j in market m
    X_jm = X_reshaped  # Shape: (M, J, features)

    # Next two elements: Sum of product j's features in all other markets
    X_j_sum = jnp.sum(X_reshaped, axis=0, keepdims=True) - X_reshaped  #
    ↪   Shape: (M, J, features)

    # Next two elements: Sum of all other products' features in the same
    ↪   market
    X_m_sum = jnp.sum(X_reshaped, axis=1, keepdims=True) - X_reshaped  #
    ↪   Shape: (M, J, features)

    # Next element: W (in the marginal cost function)
    W_costs_reshaped = W_costs.reshape(M, J, -1)

    # Next element: W (in the marginal cost function)
    Z_costs_reshaped = Z_costs.reshape(M, J, -1)

    # Final element: Prices
    prices_reshaped = prices.reshape(M, J, -1)

    # Concatenate results along the last dimension and add a column of
     ↪   ones
    instruments = jnp.concatenate([
        jnp.ones((M, J, 1)),
        X_jm,
        X_j_sum,
        X_m_sum,
        W_costs_reshaped,
        prices_reshaped,
        Z_costs_reshaped
    ], axis=-1)  # Shape: (M, J, 7)

    # Reshape back to (J * M, 6)
    return instruments.reshape(J * M, -1)


#================================================================================#
# blp_moment: gets matrix of instruments used in moment conditions
#================================================================================#
# Z: matrix of instruments used to calculate moment conditions
# Az: Annihilator matrix, used to recover xis from deltas.
def blp_moment(params, Z, Az, MJN):
    """
    Computes the BLP moment vector using vectorized instruments.

    Parameters:
    ----------
    params : array-like
        Model parameters.
    X : jnp.ndarray
        Input data matrix of shape (J * M, features).

    Returns:
    -------
    sum_vec : jnp.ndarray
        The moment vector divided by the number of market and products,
         ↪   shape (instrument_features,).
    """
    M, J, N_instruments, N = MJN
    deltas = params[1+N_instruments:].reshape(-1, 1)  # Shape: (J * M, 1)

    xis = Az @ deltas # Use the annihilator matrix to recover xi
    sum_vec = jnp.sum(xis*Z, axis=0)  # Shape: (instrument_features,)
    return (sum_vec / (J*M))


#================================================================================#
# blp_moment: gets matrix of instruments used in moment conditions
#================================================================================#
# Z: matrix of instruments used to calculate moment conditions
# Az: Annihilator matrix for the demand-side, used to recover xis from
↪  deltas.
# Xs: matrix of supply-side regressors: [1, wcost, zcost]
# As: Annihilator matrix for the suppply side, used to recover the
↪  marginal cost residual, omega.
def blp_moment_joint(params, X, Z, Az, M_iv_est, Xs, As, prices, shares,
↪  nus, nus_on_prices, MJN, ownership):
    """
    Computes the BLP moment vector using vectorized instruments.

    Parameters:
    -----------
    params : array-like
        Model parameters.
    X : jnp.ndarray
        Input data matrix of shape (J * M, features).

    Returns:
    --------
    sum_vec : jnp.ndarray
        The moment vector divided by the number of market and products,
        ↪  shape (instrument_features,).
    """
    ###### First: Demand-side moments
    M, J, N_instruments, N = MJN
    deltas = params[1+N_instruments:].reshape(-1, 1)  # Shape: (J * M, 1)
    xis = Az @ deltas # Use the annihilator matrix to recover xi
    moment_demand_side = jnp.sum(xis*Z, axis=0)  # Shape:
    ↪  (instrument_features,)

    ###### Next: Supply-side moments
    if not ownership.any(): ## I am coding perfect competition as an
    ↪  ownership matrix of zeros.
        mc=prices ### Perfect competition case.
    else:
        mc = calculate_marginal_costs(params, ownership, xis, X, M_iv_est,
         ↪  prices, nus, nus_on_prices, MJN)
    # Find the residual of the marginal cost equation
    omegas = As @ mc
    moment_supply_side = jnp.sum(omegas*Xs, axis=0)  # Shape:
    ↪  (instrument_features,)
    #Put the moments together
    moments_all = jnp.concatenate([moment_demand_side,
    ↪  moment_supply_side], axis = 0)
    return (moments_all / (J*M))


#==============================================================================#
# objective_mpec
#==============================================================================#
def objective_mpec(params, W, MJN):
    M, J, N_instruments, N = MJN
    eta = jnp.array(params[1:1+N_instruments])
    out = eta.T @ W @ eta
    return out
#==============================================================================#
# constraint_g
#==============================================================================#
def constraint_g(params, Z, Az, MJN):
    _, _, N_instruments, _ = MJN
    g_xi = blp_moment(params, Z, Az, MJN)
    eta = params[1:1+N_instruments]
    return g_xi - eta


#==============================================================================#
# constraint_g_joint
#==============================================================================#
#@partial(jit, static_argnums=(10,))
def constraint_g_joint(params, X, Z, Az, M_iv_est, Xs, As, prices, shares,
↪  nus, nus_on_prices, MJN, ownership):
    _, _, N_instruments, _ = MJN
    g_xi = blp_moment_joint(params, X, Z, Az, M_iv_est, Xs, As, prices,
    ↪  shares, nus, nus_on_prices, MJN, ownership)
    eta = params[1:1+N_instruments]
    return g_xi - eta

#==============================================================================#
# constraint_s
#==============================================================================#
def constraint_s(params, shares, nus_on_prices, MJN):
    return s(params, nus_on_prices, MJN) - shares.flatten()


# Compute the Jacobian of the constraints using Jax
constraint_g_jac = jacobian(constraint_g)
constraint_s_jac = jacobian(constraint_s)


#==============================================================================#
# objective_jac
#==============================================================================#
def objective_jac(params, W, MJN):
    _, _, N_instruments, _ = MJN
    # Extract etas
    eta = params[1:1+N_instruments]
    gradient_eta = 2 * W @ eta  # Gradient for eta

    # Build the full gradient vector
    gradient = np.zeros_like(params)  # Use NumPy for the full gradient
    gradient[1:1+N_instruments] = np.array(gradient_eta)  # Convert JAX
     ↪  array to NumPy
    return gradient

#==============================================================================#
# objective_hess
#==============================================================================#
def objective_hess(params, W, MJN):
    _, _, N_instruments, _ = MJN
    # Initialize the full Hessian matrix
    hess = np.zeros((len(params), len(params)))
    # Fill in the block corresponding to etas
    hess[1:1+N_instruments, 1:1+N_instruments] = 2 * W  # Constant Hessian
     ↪  for etas
    return hess


#==============================================================================#
# Calculate standard errors
```

```
#==============================================================================#
def standard_errors(thetastar, Z, Az, M_iv_est, shares, nus_on_prices,
↪  MJN):

    M, J, N_instruments, N = MJN

    # Predicted deltas, xis, and moment conditions
    deltahat = thetastar[1+N_instruments:].reshape(-1, 1)
    xihat = np.array(Az@deltahat)
    g0 = np.array(Z*xihat)

    # Covariance matrix of moment conditions ("meat" of the sandwich
    ↪  formula)
    Bbar = np.cov(g0.T)

    # Gradient of shares evaluated at the solution
    grad_s_star = np.array(constraint_s_jac(thetastar, shares,
    ↪  nus_on_prices, MJN))

    # Calculate derivative terms
    ds_ddelta = grad_s_star[:, 1+N_instruments:]
    ds_dsigma = grad_s_star[:, 0]
    ddelta_dsigma = -np.linalg.solve(ds_ddelta, ds_dsigma)

    # Constructing the gradient matrix, G
    # This thing represents dg_dtheta, which we sum over j, m to get G.
    dg0 = np.zeros((J*M, 1+N_instruments+J*M))
    dg0[:, 0] = ddelta_dsigma
    dg0[:, 1+N_instruments:] = np.eye(J*M)
    # Reshape it by product and market
    dg = dg0.reshape(M, J, 1+N_instruments+J*M)
    # Reshape the isnturments
    Z_reshaped = Z.reshape(M, J, N_instruments)
    #Sum over products and markets.
    G = np.zeros((N_instruments, 1+N_instruments+J*M))
    for i in range(dg.shape[0]):
        for j in range(dg.shape[1]):
            G += np.outer(Z_reshaped[i, j, :], dg[i, j, :])
    G = G/(J*M)
    GTG = G.T @ G

    # Using pseudoinverse because there's a bunch of zero columns and
    ↪  rows, corresponding with eta, which make the matrix
    ↪  non-invertible
    GTG_inv = np.linalg.pinv(GTG)

    # Variance-covariance matrix of the GMM estimates
    V_gmm = np.array(GTG_inv @ (G.T) @ Bbar @ G @ GTG_inv)

    # Get the parts of the VCV we care about
    v_sigma = V_gmm[0,0]
    V_delta = V_gmm[1+N_instruments:, 1+N_instruments:]

    # Get the variance covariance matrix of beta
    V_beta = np.array(M_iv_est @ V_delta @ (M_iv_est.T))

    # Get the standard errors
    se_betas = np.sqrt(np.diag(V_beta)/(J*M))
    se_sigma = np.sqrt(v_sigma/(J*M))

    return se_sigma, se_betas


#==================================================================================#
# Calculate standard errors, joint estimation
#==================================================================================#
def standard_errors_joint(theta_hat, X, Z, Az, M_iv_est, Xs, prices,
↪   shares, nus_on_prices, nus, MJN, ownership):

    M, J, N_instruments, N = MJN

    Nd = Z.shape[1]         #Number of demand-side instruments
    Ns = Xs.shape[1] #Number of supply-side instruments


    # Predicted deltas, xis
    delta_hat = theta_hat[1+N_instruments:].reshape(-1, 1)
    xi_hat = np.array(Az@delta_hat)

    ### This means we are a conduct case other than perfect competition
    if ownership.any():
        ### Demand-side moment conditions
        g0_demand = np.array(Z*xi_hat) # Vector of moment conditions, (J*M
        ↪   x 7)

        ### Supply-side moments
        As = np.eye(Xs.shape[0]) - Xs@np.linalg.inv(Xs.T@Xs)@Xs.T
        ↪   #supply-side annihilator matrix
```

```
#elas_hat = calculate_price_elasticity(theta_hat, xi_hat, X,
↪  M_iv_est, prices, shares, nus, nus_on_prices, MJN) #
↪  Elasticities
mc_hat = calculate_marginal_costs(theta_hat, ownership, xi_hat, X,
↪  M_iv_est, prices, nus, nus_on_prices, MJN)
↪  # Marginal Costs

### Supply-side moment conditions
g0_supply = np.array(Xs*mc_hat) # Vector of moment conditions,
↪  (J*M x 7)

### All moments (JM x 10)
g0 = np.concatenate([g0_demand, g0_supply], axis=1)

# Covariance matrix of moment conditions ("meat" of the sandwich
↪  formula)
Bbar = np.cov(g0.T)

#####-------------- Now, calculating demand-side standard errors
# Gradient of shares evaluated at the solution
grad_s_star = np.array(constraint_s_jac(theta_hat, shares,
↪  nus_on_prices, MJN))

# Calculate derivative terms
ds_ddelta = grad_s_star[:, 1+N_instruments:]
ds_dsigma = grad_s_star[:, 0]
ddelta_dsigma = -np.linalg.solve(ds_ddelta, ds_dsigma)

# Constructing the gradient matrix, G
dgd0 = np.zeros((J*M, 1+N_instruments+J*M))
dgd0[:, 0] = ddelta_dsigma
dgd0[:, 1+N_instruments:] = np.eye(J*M)

# Reshape it by product and market
dgd = dgd0.reshape(M, J, 1+N_instruments+J*M)
Z_reshaped = Z.reshape(M, J, Nd)

#####-------------- Preparing supply-side errors
#Jacobian of marginal costs evaluated at theta_hat
mc_jac = jacobian(calculate_marginal_costs)
Jmc = mc_jac(theta_hat, ownership, xi_hat, X, M_iv_est, prices,
↪  nus, nus_on_prices, MJN).reshape(J*M, J*M+N_instruments+1)
# Combined Jacobian of marginal costs with respect to theta
dmc_dtheta = Jmc
```

```
# This thing (As@dmc_dtheta) gives domega_dtheta, the derivative
↪  of the supply-side residual.
dgs0 = As@dmc_dtheta
dgs = dgs0.reshape(M, J, 1+N_instruments+J*M)
Xs_reshaped = Xs.reshape(M, J, Ns)

# Final "gradient matrix G" used in calculation of standard
↪  errors.
Gd = np.zeros((Nd, 1+N_instruments+J*M))
Gs =  np.zeros((Ns, 1+N_instruments+J*M))
# Loop through and calculate standard errors
for i in range(dgd.shape[0]):
    for j in range(dgd.shape[1]):
        Gd += np.outer(Z_reshaped[i, j, :], dgd[i, j, :])
for i in range(dgs.shape[0]):
    for j in range(dgs.shape[1]):
        Gs += np.outer(Xs_reshaped[i, j, :], dgs[i, j, :])

#Combine gradient of supply and demand moment conditions
G = np.concatenate([Gd, Gs], axis=0)/(J*M)
GTG = G.T @ G

# Using pseudoinverse because there's a bunch of zero columns and
↪  rows, corresponding with eta, which make the matrix
↪  non-invertible
GTG_inv = np.linalg.pinv(GTG)

# Variance-covariance matrix of the GMM estimates
V_gmm = np.array(GTG_inv @ (G.T) @ Bbar @ G @ GTG_inv)

# Get the parts of the VCV we care about
v_sigma = V_gmm[0,0]
V_delta = V_gmm[1+N_instruments:, 1+N_instruments:]

# Get the variance covariance matrix of beta
V_beta = np.array(M_iv_est @ V_delta @ (M_iv_est.T))

#Next, use delta method to get standard errors for gamma.
Ms = np.linalg.inv(Xs.T@Xs)@Xs.T
V_mc = (dmc_dtheta)@(V_gmm)@(dmc_dtheta).T
V_gamma = (Ms)@(V_mc)@(Ms.T)

# Get the standard errors
se_betas = np.sqrt(np.diag(V_beta)/(J*M))
```

```
        se_sigma = np.sqrt(v_sigma/(J*M))
        se_gamma = np.sqrt(np.diag(V_gamma)/(J*M))

    else: ### Perfect competition case
        #Demand side moment not dependent on gammas.

        ### Demand-side moment conditions
        g0 = np.array(Z*xi_hat) # Vector of moment conditions, (J*M x 7)

        # Covariance matrix of moment conditions ("meat" of the sandwich
        ↪  formula)
        Bbar = np.cov(g0.T)

        #####-------------- Now, calculating demand-side standard errors
        # Gradient of shares evaluated at the solution
        grad_s_star = np.array(constraint_s_jac(theta_hat, shares,
        ↪  nus_on_prices, MJN))

        # Calculate derivative terms
        ds_ddelta = grad_s_star[:, 1+N_instruments:]
        ds_dsigma = grad_s_star[:, 0]
        ddelta_dsigma = -np.linalg.solve(ds_ddelta, ds_dsigma)

        # Constructing the gradient matrix, G
        dgd0 = np.zeros((J*M, 1+N_instruments+J*M))
        dgd0[:, 0] = ddelta_dsigma
        dgd0[:, 1+N_instruments:] = np.eye(J*M)

        # Reshape it by product and market
        dgd = dgd0.reshape(M, J, 1+N_instruments+J*M)
        Z_reshaped = Z.reshape(M, J, Nd)

        #####-------------- Preparing supply-side errors
        # Final "gradient matrix G" used in calculation of standard
        ↪  errors.
        Gd = np.zeros((Nd, 1+N_instruments+J*M))
        # Loop through and calculate standard errors
        for i in range(dgd.shape[0]):
            for j in range(dgd.shape[1]):
                Gd += np.outer(Z_reshaped[i, j, :], dgd[i, j, :])

        #Combine gradient of supply and demand moment conditions
        G = Gd/(J*M)
        GTG = G.T @ G
```

```
# Using pseudoinverse because there's a bunch of zero columns and
↪  rows, corresponding with eta, which make the matrix
↪  non-invertible
GTG_inv = np.linalg.pinv(GTG)

# Variance-covariance matrix of the GMM estimates
V_gmm = np.array(GTG_inv @ (G.T) @ Bbar @ G @ GTG_inv)

# Get the parts of the VCV we care about
v_sigma = V_gmm[0,0]
V_delta = V_gmm[1+N_instruments:, 1+N_instruments:]

# Get the variance covariance matrix of beta
V_beta = np.array(M_iv_est @ V_delta @ (M_iv_est.T))

se_betas = np.sqrt(np.diag(V_beta)/(J*M))
se_sigma = np.sqrt(v_sigma/(J*M))


#Supply-side moment standard errors come directly from the SE
↪  formulas of linear regression
#mc=p is no longer a random variable.
XsTXs_inv = np.linalg.inv(Xs.T @ Xs)
#gamma_hat = XTX_inv @ (Xs.T) @ prices
#residuals
#omega_hat = prices - Xs @ gamma_hat
#n, k = Xs.shape
# Outer product of residuals and rows of X
#robust_sum = np.zeros((k, k))
#for i in range(n):
#    Xi = X[i, :].reshape(-1, 1)  # Row vector of X as column
#    robust_sum += (omega_hat[i] ** 2) * (Xi @ Xi.T)
# Robust variance-covariance matrix
#var_gamma_robust = XTX_inv @ robust_sum @ XTX_inv
#robust_standard_errors = np.sqrt(np.diag(var_gamma_robust))
### Try non-robust SEs
# Step 1: Calculate regression coefficients (gamma)
gamma_hat = XsTXs_inv @ Xs.T @ prices
# Step 2: Calculate residuals
omega_hat = prices - Xs @ gamma_hat
# Step 3: Estimate variance of residuals (sigma^2)
n, k = X.shape
sigma2 = (omega_hat.T @ omega_hat) / (n - k)
```

```
        # Step 4: Calculate variance-covariance matrix of gamma
        var_gamma = sigma2 * np.linalg.inv(Xs.T @ Xs)
        # Step 5: Standard errors of gamma
        se_gamma = np.sqrt(np.diag(var_gamma))

    return se_sigma, se_betas, se_gamma


#==============================================================================#
# Calculate price elasticities
#==============================================================================#
##### New version for easier Jacobian calculation in standard errors.
#@partial(jit, static_argnums=(6,))  --------- non-hashable static
↪  arguments are not supported. One of these things is not a Jax array.
@partial(jit, static_argnums=(8,))
def calculate_price_elasticity(params, xi, X, M_iv_est, prices, shares,
↪  nus, nus_on_prices, MJN):

    M, J, N_instruments, N = MJN

    ### Extract parameters
    sigma = params[0]
    #deltas = params[1+N_instruments:]
    # Use lax.dynamic_slice for dynamic slicing
    deltas_start = N_instruments + 1
    deltas = lax.dynamic_slice(params, (deltas_start,), (params.shape[0] -
    ↪  deltas_start,)).reshape(-1, 1)

    ### Calculate betas and alphas
    betas_and_alpha_hat = (M_iv_est @ deltas)
    betas = betas_and_alpha_hat[:3]
    alpha = -betas_and_alpha_hat[3]

    # Take the drawn alphas and calculate the utilities for each consumer
    alphas = (sigma*nus + alpha).reshape(M, N)


    # Compute utilities
    utilities = deltas - sigma*nus_on_prices
    # Reshape by markets and products
    utilities_reshaped = utilities.reshape(M, J, N)  # Shape: (M, J, N)
    # Compute the stabilization constant (max utilities per market per
    ↪  individual)
    max_utilities = jnp.max(utilities_reshaped, axis=1, keepdims=True)
```

```
# Stabilized exponentials
exp_utilities = jnp.exp(utilities_reshaped-max_utilities)
#Adjust the outside option (1 becomes exp(-max_utilities))
outside_option = jnp.exp(-max_utilities)  # Shape: (M, 1, N)
#Compute the stabilized denominator
sum_exp_utilities = outside_option + exp_utilities.sum(axis=1,
↪  keepdims=True)  # Shape: (M, 1, N)
# Compute individual-level market shares (before averaging)
ind_shares = exp_utilities / sum_exp_utilities  # Shape: (M, J, N)
ind_shares = ind_shares.reshape(J*M, N)

#### Trying to vectorize
# Reshaping alphas
alphas_repeat0 = jnp.repeat(alphas, repeats=J*J, axis=0)
alphas_mat = alphas_repeat0.reshape(J, J, M, N, order='F') #(shape:J,
↪  J, M, N).
# Reshaping prices for j-and k-indexing
# Here, "j" varies by row, "k" varies by column
prices_rs = prices.reshape(M, J)
prices_repeat = jnp.repeat(prices_rs, repeats=J, axis=1)
prices_j = prices_repeat.reshape(M, J, J).transpose(1, 2, 0)
prices_j = prices_j[..., np.newaxis]    #(shape: J, J, M, 1)
prices_k = prices_j.transpose(1,0,2,3)  #(shape: J, J, M, 1)
# Do the same for shares
ind_shares_rs = ind_shares.reshape(M, J, N)
ind_shares_repeat = jnp.repeat(ind_shares_rs, repeats=J, axis=0)
ind_shares_k = ind_shares_repeat.reshape(M, J, J, N).transpose(1, 2,
↪  0, 3) #(shape: J, J, M, N)
ind_shares_j = ind_shares_k.transpose(1,0,2,3)
↪  #(shape: J, J, M, N)
# Average of shares
shares_j = jnp.mean(ind_shares_j, axis=3, keepdims=True)

#### Elasticities
# Own-price elasticity (we will only need the diagonals of this
↪  matrix.)
elas_own_price =
↪  -(prices_j/shares_j)*alphas_mat*ind_shares_j*(1-ind_shares_j)
↪  #(shape: J, J, M, N)
#Cross-price elasticity (we will only need the off-diagonals.)
elas_cross_price =
↪  (prices_k/shares_j)*alphas_mat*ind_shares_j*ind_shares_k
↪  #(shape: J, J, M, N)
```

```
    # Average across elasticities (i.e., evaluate the Monte Carlo
    ↪  integral)
    elas_own_price_mean = jnp.mean(elas_own_price, axis=3)     #(shape: J,
    ↪  J, M)
    elas_cross_price_mean = jnp.mean(elas_cross_price, axis=3) #(shape: J,
    ↪  J, M)

    ## Combine the off-diagonal elements of the cross-price elasticities
    ↪  with the diagonal elements of the own-price elasticities.
    diag_indices = jnp.arange(J)
    diag_mask = (diag_indices[:, None] == diag_indices[None, :])[:, :,
    ↪  None]
    elas_mean = jnp.where(diag_mask, elas_own_price_mean,
    ↪  elas_cross_price_mean)

    return elas_mean.flatten()

#================================================================================#
# Calculate marginal costs
#================================================================================#

def calculate_marginal_costs(params, ownership, xi, X, M_iv_est, prices,
↪  nus, nus_on_prices, MJN):

    M, J, N_instruments, N = MJN

    # Calculate shares and reshape
    shares = s(params, nus_on_prices, MJN)
    shares_reshaped = shares.reshape(M, J).T

    # Reshape prices
    prices_reshaped = prices.reshape(M, J).T

    # Calculate elasticities and reshape to (J, J, M)
    elasticities = calculate_price_elasticity(params, xi, X, M_iv_est,
    ↪  prices, shares, nus, nus_on_prices, MJN)
    elasticities_reshaped = elasticities.reshape(J, J, M)

    mc = jnp.zeros(J*M).reshape(J*M, -1)

    for m in range(M):
            elast_mkt = elasticities_reshaped[:, :, m].reshape(J, J)
            shares_mkt = shares_reshaped[:, m]
            prices_mkt = prices_reshaped[:, m]
            p_over_s = (1/shares_mkt).reshape(J, -1) @
             ↪  prices_mkt.reshape(-1, J)
            s_partial_p = elast_mkt/p_over_s
            mc_mkt = jnp.linalg.inv(ownership*s_partial_p.T) @
             ↪  shares_mkt.reshape(J, -1) + prices_mkt.reshape(J, -1)
            mc = mc.at[J*m:J*m + J].add(mc_mkt)

    return mc

#==============================================================================#
# Calculate consumer surplus
#==============================================================================#

def calculate_consumer_surplus(betas, alpha, sigma_alpha, xi, X, prices,
 ↪  MJN):

    M, J, N_instruments, N = MJN

    # Draw alphas and calculate the utilities for each consumer
    alphas = (sigma_alpha*np.random.lognormal(0.0, 1.0, M*N) +
     ↪  alpha).reshape(M, N)

    utilities = (betas.reshape(1, 3) @ X.T).reshape(J*M, -1) -
     ↪  prices*np.repeat(alphas, repeats=J, axis=0) + xi
    utilities_exp = np.exp(utilities)

    # Create an array of shape (M, N) that will store consumer surplus
     ↪  for each consumer in all markets
    cs = np.zeros((M, N))

    for m in range(M):
        utilities_exp_mkt = utilities_exp[J*m:J*m + J, :]
        cs[m, :] = np.log(1 + utilities_exp_mkt.sum(axis=0))/alphas[m, :]

    cs = cs.sum(axis=1)/N
    return cs



#==============================================================================#
#==============================================================================#
#==============================================================================#
#==============================================================================#
#==============================================================================#
```

```
#==============================================================================#
# Functions for the first part: making graphs, loading the data, etc.
#==============================================================================#
#==============================================================================#
#==============================================================================#
#==============================================================================#
#==============================================================================#
#==============================================================================#

#==============================================================================#
# Loading data from MATLAB
#==============================================================================#
def load_mat_data(datapath, nrProducts, nrMarkets):
    """
    Purpose: Loads a .mat data file and returns a Pandas DataFrame.

    Parameters
    ----------
    datapath : str
        The path to the .mat file.

    nrProducts : int
        The number of products in the market data.

    nrMarkets : int
        The number of markets in the market data.

    Returns
    -------
    pd.DataFrame
        The market level data in the .mat data file converted to a Pandas
        ↪  DataFrame.
    pd.DataFrame
        The simulated alphas in the .mat data file converted to a Pandas
        ↪  DataFrame.

    Description
    -----------
    This function loads the .mat data using scipy.io.loadmat and collects
    ↪  the variable names and the
    data (in numpy arrays), ignoring other items in the dictionary (such
    ↪  as the header). It then converts
    the cleaned dictionary into two DataFrames, one for the market level
    ↪  data and one for the simulated alphas in each market.
    """

    # Load the .mat data and format the X's appropriately
    mat = loadmat(datapath)
    mat = {k:v for k, v in mat.items() if k[0] != '_'}
    mat['x2'] = mat['x1'][:, 1]
    mat['x3'] = mat['x1'][:, 2]
    mat['x1'] = mat['x1'][:, 0]

    # Get the simulated alphas into one DataFrame
    alphas = mat['alphas']
    column_names = [i for i in range(alphas.shape[1])]
    df_alphas = pd.DataFrame(alphas, columns=column_names)
    mat.pop('alphas')

    # Store the market level data to a DataFrame
    df_mkt = pd.DataFrame({k: np.array(v).flatten(order='F') for k, v in
     ↪  mat.items()})

    # Add market and product ids to the market level data
    product_ids = [i+1 for i in range(nrProducts)] * nrMarkets
    market_ids = [i+1 for i in range(nrMarkets) for _ in
     ↪  range(nrProducts)]
    df_mkt['market_id'] = market_ids
    df_mkt['product_id'] = product_ids

    return df_mkt, df_alphas

#==============================================================================#
# Drawing logit shocks (epsilons)
#==============================================================================#
def draw_epsilons(alphas):
    """
    Purpose: Draws epsilons from the specified distribution, to the same
     ↪  shape as alphas.

    Parameters
    ----------
    alphas : pd.DataFrame
        The DataFrame of alphas, in the form (Number of Consumers) x
         ↪  (Number of Markets).

    Returns
    -------
    pd.DataFrame
        The simulated epsilons in a Pandas DataFrame.

    Description
    -----------
    This function takes the simulated alphas for all consumers in all
    ↪  markets and simulates epsilons for each of them.
    """
    draws = np.random.gumbel(size=(alphas.shape[0], alphas.shape[1]))
    column_names = [i+1 for i in range(draws.shape[1])]
    df_epsilons = pd.DataFrame(draws, columns=column_names)
    return df_epsilons

#==============================================================================#
# calculate consumer welfare for the true data.
#==============================================================================#
def calculate_welfare(data, alphas, beta, epsilons):
    """
    Purpose: Calculates the consumer welfares based on the given market
    ↪  level data and the simulated consumers.

    Parameters
    ----------
    data : pd.DataFrame
        The DataFrame of market level data.

    alphas : pd.DataFrame
        The DataFrame of alphas, in the form (Number of Consumers) x
        ↪  (Number of Markets).

    epsilons : pd.DataFrame
        The DataFrame of epsilons, in the form (Number of Consumers) x
        ↪  (Number of Markets).

    Returns
    -------
    np.array
        The (Number of Consumers) x (Number of Markets) array of
        ↪  utilities (welfare) of each consumer in each market,
        ↪  conditional
        on them choosing optimally based on their utility function.

    Description
    -----------
```

```
This function takes the market level data, and simulated alphas and
↪  epsilons and calculates the welfare for each consumer in each
↪  market,
conditional on them choosing optimally based on their utility
↪  function parameters.
"""
# Check that the alphas and epsilons agree on the number of markets
assert len(alphas) == len(epsilons)

# Store number of markets
nrMarkets = len(alphas)

# Store number of products
nrProducts = data['product_id'].max()

for market_id in range(1, nrMarkets+1):
    # Get the data for the market at hand
    mkt_data = data.loc[data['market_id']==market_id].copy()

    # Calculate the part of the utility that is independent from the
    ↪  consumer
    mkt_data['common_util'] = beta[0]*mkt_data['x1'] +
    ↪  beta[1]*mkt_data['x2'] + beta[2]*mkt_data['x3'] +
    ↪  mkt_data['xi_all']

    # Calculate consumer utilities
    utils = {}
    for product in range(1, nrProducts+1):
        utils[product] = (

              ↪  -alphas.iloc[market_id-1].values*mkt_data.loc[mkt_data['product_id']
             + epsilons.iloc[market_id-1].values
             +
              ↪  mkt_data.loc[mkt_data['product_id']==product]['common_util'].iloc[0]
        )

    # Stack utilities for each product in the market for each
    ↪  consumers into a matrix
    product_utilities = np.stack(tuple(utils.values()), axis=1)

    # Create a column of zeros with the same number of rows as there
    ↪  are consumers
    zero_column = np.zeros((product_utilities.shape[0], 1))
```

```
        # Concatenate the zero column to 'product_utilities', to store
        ↪  the utility from the outside option (zero)
        product_utilities = np.concatenate([product_utilities,
        ↪  zero_column], axis=1)

        mkt_welfare = np.amax(product_utilities, axis=1).reshape(500, -1)

        # Divide by alpha_i's to get the surplus in monetary units
        mkt_welfare = mkt_welfare/alphas.iloc[market_id-1].values

        if market_id == 1:
            market_welfares = mkt_welfare
        else:
            market_welfares = np.hstack([market_welfares, mkt_welfare])

    return market_welfares

#==================================================================================#
# Plot the histograms for the true data.
#==================================================================================#
def plot_two_histograms(data1, data2, bins=500, labels=('Data 1', 'Data
↪  2'), path=""):
    """
    Purpose: Plots two histograms side by side to compare the
    ↪  distributions of two different datasets.

    Parameters
    ----------
    data1 : np.array
        The first dataset, which can be a multi-dimensional NumPy array.
        ↪  All elements will be flattened for the histogram.

    data2 : np.array
        The second dataset, which can also be a multi-dimensional NumPy
        ↪  array. All elements will be flattened for the histogram.

    bins : int, optional
        The number of bins for each histogram (default is 50).

    labels : tuple of str, optional
        Labels for each dataset, used in the titles of the histograms
        ↪  (default is ('Data 1', 'Data 2')).

    Description
    -----------
    This function takes two datasets, flattens them into 1-dimensional
    ↪  arrays if necessary, and plots them as two
    histograms side by side in a single figure. It provides a visual
    ↪  comparison of the distributions in both datasets.
    """
    # Create a figure with two subplots side by side
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Plot the first histogram
    axes[0].hist(data1.flatten(), bins=bins)
    axes[0].set_title(f'{labels[0]}')
    axes[0].set_xlabel('Value')
    axes[0].set_ylabel('Frequency')

    # Plot the second histogram
    axes[1].hist(data2.flatten(), bins=bins)
    axes[1].set_title(f'{labels[1]}')
    axes[1].set_xlabel('Value')
    axes[1].set_ylabel('Frequency')

    # Show the plots
    plt.tight_layout()

    if path != "":
        plt.savefig(path, dpi=400)

    plt.show()

def predict_prices_and_shares(ownership, mc, betas, alpha, sigma_alpha,
↪  xi, X, MJN):

    M, J, N_instruments, N = MJN

    # Draw alphas and calculate the utilities for each consumer
    alphas = (sigma_alpha*np.random.lognormal(0.0, 1.0, M*N) +
    ↪  alpha).reshape(M, N)

    def predict_ind_shares(p):
        utilities = (betas.reshape(1, 3) @ X.T).reshape(J*M, -1) -
        ↪  p.reshape(-1, 1)*np.repeat(alphas, repeats=J, axis=0) + xi

        # Reshape utilities for markets and products
        utilities_reshaped = utilities.reshape(M, J, N)  # Shape: (M, J,
        ↪  N)

        # Compute the stabilization constant (max utility per market per
        ↪  individual)
        max_utilities = jnp.max(utilities_reshaped, axis=1, keepdims=True)
        ↪  # Shape: (M, 1, N)

        # Stabilized exponentials
        exp_utilities = jnp.exp(utilities_reshaped - max_utilities)  #
        ↪  Shape: (M, J, N)

        # Adjust the "outside option" (1 becomes exp(-max_utilities))
        outside_option = jnp.exp(-max_utilities)  # Shape: (M, 1, N)

        # Compute the stabilized denominator
        sum_exp_utilities = outside_option + exp_utilities.sum(axis=1,
        ↪  keepdims=True)  # Shape: (M, 1, N)

        # Compute shares
        ind_shares = exp_utilities / sum_exp_utilities  # Shape: (M, J, N)

        return ind_shares.reshape(J*M, N)

    def zeta(p):

        ind_shares = predict_ind_shares(p)
        mkt_shares = ind_shares.mean(axis=1)

        lambda_diag = (ind_shares*np.repeat(-alphas, repeats=J,
        ↪  axis=0)).sum(axis=1)/N

        # Reshape into a (J*M, J) matrix with block diagonal structure
        blocks = []
        blocks_inv = []
        for i in range(0, len(lambda_diag), J):
            diag_matrix = np.diag(lambda_diag[i:i+J])
            diag_matrix_inv = np.linalg.inv(diag_matrix)
            blocks.append(diag_matrix)
            blocks_inv.append(diag_matrix_inv)

        # Stack the blocks vertically
        lambda_mat = np.vstack(blocks)
        lambda_mat_inv = np.vstack(blocks_inv)
        gamma_mat = np.zeros((M, J, J))
        for m in range(M):
            for j in range(J):
                for k in range(J):
                    gamma_mat[m, j, k] = (ind_shares[J*m + j,
                     ↪  :]*ind_shares[J*m + k, :]*(-alphas[m, :])).mean()

        gamma_mat = gamma_mat.reshape(J*M, J)

        zeta = np.zeros(J*M)
        for m in range(M):
            zeta[m*J:m*J + J] = (lambda_mat_inv[m*J:m*J + J, :] @
                ↪  (ownership*gamma_mat[m*J:m*J + J, :]) @ (p[m*J:m*J +
                ↪  J].reshape(-1, 1) - mc[m*J:m*J + J])
                                - lambda_mat_inv[m*J:m*J + J, :] @
                                    ↪  mkt_shares[m*J:m*J + J].reshape(-1,
                                    ↪  1)).reshape(-1)

        return zeta, lambda_mat

    def contraction_mapping(zeta, p0, tol=1e-8, max_iter=1000):
        """
        Implements the contraction mapping: p <- c + zeta(p).

        Parameters:
        - zeta: function zeta(p), mapping p to a vector or scalar of the
         ↪  same shape as p.
        - p0: initial guess for p (scalar or array).
        - tol: convergence tolerance (default 1e-8).
        - max_iter: maximum number of iterations (default 1000).

        Returns:
        - p: converged value of p.
        """
        p = p0
        for n_iter in range(max_iter):
            zeta_vec, lambda_mat = zeta(p)
            p_new = mc.reshape(-1) + zeta_vec

            # Reshape into groups of 3x3 matrices and 3x1 vectors
            lambda_mat_grouped = lambda_mat.reshape(-1, J, J)  # Shape:
                ↪  (100, 3, 3)
```

```
            p_grouped = (p_new - p).reshape(-1, 3, 1)  # Shape: (100, 3,
            ↪  1)

            # Perform batch matrix-vector multiplication
            norm_grouped = np.matmul(lambda_mat_grouped, p_grouped)  #
            ↪  Shape: (100, 3, 1)

            # Flatten the result back into a vector
            norm_vector = norm_grouped.reshape(-1)  # Shape: (300,)
            norm = np.linalg.norm(norm_vector, np.inf)
            print(f"Iteration {n_iter}. Norm: {norm}.")
            if norm < tol:  # Check for convergence
                print("Contraction mapping converged, found prices that
                ↪  satisfy the FOC.")
                print("Iterations:", n_iter)
                return p_new
            p = p_new
        raise RuntimeError("Contraction mapping did not converge within
        ↪  the maximum number of iterations.")

    p_init = np.ones(J*M)

    res_prices = contraction_mapping(zeta, p_init)
    res_shares = predict_ind_shares(res_prices).sum(axis=1)/N

    return res_prices, res_shares
```

B.3 Functions for Storing Estimation Results

```
import numpy as np
import pandas as pd

def store_results(results, file_path):

    table = pd.DataFrame(columns=['Parameter', 'Estimate', 'Std. Error',
    ↪  'True Value', 'Bias'])

    sigma_alpha_true=1
    beta_true = np.array([5, 1, 1])
    alpha_true=1
    gamma_true = np.array([2, 1, 1])

    for i, name in enumerate(results['se_names']):
        if i == 0:
            estimate = results['sigma_alpha_hat']
            true = sigma_alpha_true
        elif i >= 1 and i <= 3:
            estimate = (results['beta_hat'][i-1]).item()
            true = beta_true[i-1].item()
        elif i == 4:
            estimate = results['alpha_hat']
            true = alpha_true
        elif i >= 5 and i <= 7:
            estimate = (results['gamma_hat'][i-5]).item()
            true = gamma_true[i-5].item()

        bias = estimate-true
        #print(estimate)

        se = results['se'][i]

        table.loc[len(table)] = [name, estimate, se, true, bias]

        file_type = file_path.split(".")[-1]

        if file_type == "xlsx":
            table.to_excel(file_path, index=False)
            print(f"Excel file saved to {file_path}")
        elif file_type == "tex":
            # Generate LaTeX code for the table
            latex_code = table.to_latex(index=False, caption="Parameter
             ↪  Estimates", float_format="%.4f")
            # Save the LaTeX code to a file
            with open(file_path, "w") as file:
                file.write(latex_code)
            print(f"TeX file saved to {file_path}")
        else:
            print(f"File type {file_type} is not supported. Options are:
             ↪  .xlsx and .tex.")

def save_mc_to_excel(results, file_name="mc_comparison.xlsx", mode =
 ↪  "mc"):
    """
    Extracts true and predicted marginal costs from the dictionary 'mc',
    creates a comparison DataFrame, and saves it to an Excel file.
    Parameters:
    mc (dict): A dictionary with keys 'true' and 'hat', each containing a
    ↪  3x1 vector.
    file_name (str): Name of the Excel file to save. Default is
    ↪  'mc_comparison.xlsx'.

    Returns:
    None
    """
    if mode == "mc":
        mc = results['mean_mc']
    else:
        mc = results['mean_profits']

    # Extract true and predicted values from the dictionary
    true_values = mc.get('true', [])
    predicted_values = mc.get('hat', [])

    # Ensure they are the same length
    if len(true_values) != len(predicted_values):
        raise ValueError("Length of 'true' and 'hat' values in 'mc'
        ↪  dictionary must be the same.")

    # Create a DataFrame for comparison
    df = pd.DataFrame({
        "Index": range(1, len(true_values) + 1),
        "True Value": true_values.flatten(),
        "Predicted Value": predicted_values.flatten(),
        "Difference": (predicted_values - true_values).flatten()
    })

    # Save the DataFrame to an Excel file
    df.to_excel(file_name, index=False, sheet_name="MC Comparison")
    print(f"Comparison file saved to {file_name}.")
```