Econ 450-1: Industrial Organization
Problem Set #3

Johannes Hirvonen, Russell Miles

November 3, 2024

Problem 1:

1.

Given the assumptions, we can write Equation (9) in GNR (2017) as:

$$\begin{aligned} E\left[y_t \mid \Gamma_t\right] &= E\left[f\left(k_t, l_t, m_t\right) \mid \Gamma_t\right] + E\left[\omega_t \mid \Gamma_t\right] \\ &= \delta_0 + \alpha_k k_t + \alpha_l l_t + \alpha_m E\left[m_t \mid \Gamma_t\right] \\ &+ \delta_1\left(\phi(k_{t-1}, l_{t-1}, m_{t-1}) + d_{t-1} - \alpha_k k_{t-1} - \alpha_l l_{t-1} - \alpha_m m_{t-1}\right). \end{aligned}$$

2.

Given the prices of output P_t and materials ρ_t, we can write the firm's optimization problem with respect to materials M_t as:

$$\max_{M_t} P_t K_t^{\alpha_k} L_t^{\alpha_l} M_t^{\alpha_m} E\left[e^{\omega_t + \epsilon_t}\right] - \rho_t M_t.$$

Following the notation in GNR, we denote $E\left[e^{\epsilon_t}\right] \equiv \mathcal{E}$, which then leads to the FOC:

$$\alpha_m P_t K_t^{\alpha_k} L_t^{\alpha_l} M_t^{\alpha_m - 1} e^{\omega_t} \mathcal{E} = \rho_t.$$

Taking logs of both sides and rearranging:

$$\begin{aligned} m_t &= \frac{1}{\alpha_m - 1}\left(\log \rho_t - \log P_t - \log \mathcal{E} - \log \alpha_m - \omega_t - \alpha_k k_t - \alpha_l l_t\right) \\ &= \frac{1}{\alpha_m - 1}\left(d_t - \log \alpha_m - \omega_t - \alpha_k k_t - \alpha_l l_t\right) \qquad (1) \\ &= \frac{1}{1 - \alpha_m}\left(\alpha_k k_t + \alpha_l l_t + \log \alpha_m + \omega_t - d_t\right) \end{aligned}$$

where $d_t \equiv \log \rho_t - \log P_t - \log \mathcal{E}$ as in GNR. Taking the expectation conditional on Γ_t and substituting into the expression in the previous part we get:

$$\begin{aligned} E\left[y_t \mid \Gamma_t\right] = \delta_0 + \frac{\alpha_m}{1 - \alpha_m} \log \alpha_m + \frac{\alpha_k}{1 - \alpha_m} k_t + \frac{\alpha_l}{1 - \alpha_m} l_t + \frac{\alpha_m}{1 - \alpha_m} E\left[\omega_t \mid \Gamma_t\right] - \frac{\alpha_m}{1 - \alpha_m} d_t \\ + \delta_1\left(\phi(k_{t-1}, l_{t-1}, m_{t-1}) + d_{t-1} - \alpha_k k_{t-1} - \alpha_l l_{t-1} - \alpha_m m_{t-1}\right), \end{aligned}$$

where we have used the following for $x_t \in \{k_t, l_t\}$ and $\alpha_x \in \{\alpha_k, \alpha_l\}$:

$$\frac{\alpha_m \alpha_x}{1-\alpha_m} x_t + \alpha_x x_t = \frac{\alpha_m \alpha_x}{1-\alpha_x} x_t + \frac{\alpha_x - \alpha_x \alpha_m}{1-\alpha_m} x_t = \frac{\alpha_x}{1-\alpha_m} x_t.$$

Next, note that:

$$E\left[\omega_t \mid \Gamma_t\right] = \delta_0 + \delta_1 \left(\phi(k_{t-1}, l_{t-1}, m_{t-1}) + d_{t-1} - \alpha_k k_{t-1} - \alpha_l l_{t-1} - \alpha_m m_{t-1}\right),$$

which then implies:

$$\begin{aligned} E\left[y_t \mid \Gamma_t\right] = {} & \frac{1}{1-\alpha_m}\delta_0 + \frac{\alpha_m}{1-\alpha_m}\log\alpha_m + \frac{\alpha_k}{1-\alpha_m}k_t + \frac{\alpha_l}{1-\alpha_m}l_t - \frac{\alpha_m}{1-\alpha_m}d_t + \frac{\delta_1}{1-\alpha_m}d_{t-1} \\ & + \frac{\delta_1}{1-\alpha_m}\left(\phi(k_{t-1}, l_{t-1}, m_{t-1}) - \alpha_k k_{t-1} - \alpha_l l_{t-1} - \alpha_m m_{t-1}\right). \end{aligned}$$

Finally, denoting $C \equiv \frac{1}{1-\alpha_m}\delta_0 + \frac{\alpha_m}{1-\alpha_m}\log\alpha_m$ we have:

$$\begin{aligned} E\left[y_t \mid \Gamma_t\right] = {} & C + \frac{\alpha_k}{1-\alpha_m}k_t + \frac{\alpha_l}{1-\alpha_m}l_t - \frac{\alpha_m}{1-\alpha_m}d_t + \frac{\delta_1}{1-\alpha_m}d_{t-1} \\ & + \frac{\delta_1}{1-\alpha_m}\left(\phi(k_{t-1}, l_{t-1}, m_{t-1}) - \alpha_k k_{t-1} - \alpha_l l_{t-1} - \alpha_m m_{t-1}\right). \end{aligned}$$

3.

Starting from Equation (1) in the previous part, we have:

$$\begin{aligned} & m_t = \frac{1}{1-\alpha_m}\left(\alpha_k k_t + \alpha_l l_t + \log\alpha_m + \omega_t - d_t\right) \\ \iff & \omega_t = (1-\alpha_m)m_t - \alpha_k k_t - \alpha_l l_t - \log\alpha_m + d_t \equiv \mathbb{M}^{-1}(k_t, l_t, m_t) + d_t. \end{aligned}$$

Now, since $\phi(k_t, l_t, m_t) = f(k_t, l_t, m_t) + \mathbb{M}^{-1}(k_t, l_t, m_t)$, we have:

$$\phi(k_{t-1}, l_{t-1}, m_{t-1}) = m_{t-1} - \log\alpha_m.$$

Replacing this into the expression for $E\left[y_t \mid \Gamma_t\right]$ and rearranging we get:

$$\begin{aligned} E\left[y_t \mid \Gamma_t\right] = {} & \tilde{C} + \frac{\alpha_k}{1-\alpha_m}k_t + \frac{\alpha_l}{1-\alpha_m}l_t - \frac{\delta_1\alpha_k}{1-\alpha_m}k_{t-1} - \frac{\delta_1\alpha_l}{1-\alpha_m}l_{t-1} + \delta_1 m_{t-1} \\ & - \frac{\alpha_m}{1-\alpha_m}d_t + \frac{\delta_1}{1-\alpha_m}d_{t-1}, \end{aligned}$$

where $\tilde{C} \equiv C - \frac{\delta_1}{1-\alpha_m}\log\alpha_m$.

4.

Since output and input prices are assumed to be fixed, we have $d_t = d_{t-1} = d$. Thus, the terms with d_t and d_{t-1} are subsumed into the constant. That is, with $\hat{C} \equiv \tilde{C} - \frac{\alpha_m}{1-\alpha_m} d_t + \frac{\delta_1}{1-\alpha_m} d_{t-1} = \tilde{C} - \frac{\delta_1 - \alpha_m}{1-\alpha_m} d$ we have:

$$E\left[y_t \mid \Gamma_t\right] = \hat{C} + \frac{\alpha_k}{1-\alpha_m} k_t + \frac{\alpha_l}{1-\alpha_m} l_t - \frac{\delta_1 \alpha_k}{1-\alpha_m} k_{t-1} - \frac{\delta_1 \alpha_l}{1-\alpha_m} l_{t-1} + \delta_1 m_{t-1}. \tag{2}$$

From this, it is easy to see that none of the parameters of the production function are identified. Variation in k_t identifies $\frac{\alpha_k}{1-\alpha_m}$ and variation in l_t identifies $\frac{\alpha_l}{1-\alpha_m}$. In other words, the parameters α_k, α_m, and α_m are identified only up to scale. Note that while δ_1 is identified (it is the coefficient on m_{t-1}), it doesn't help us identify the other parameters, since the coefficients on k_{t-1} and l_{t-1} are equal to the coefficients on k_t and l_t after dividing both by δ_1.

5.

To find the share equation in this context, we start with the FOC of the firm derived in part 2:

$$\alpha_m P_t K_t^{\alpha_k} L_t^{\alpha_l} M_t^{\alpha_m - 1} e^{\omega_t} \mathcal{E} = \rho_t.$$

Taking logs of both sides gives:

$$\log \alpha_m + \log P_t + \alpha_k k_t + \alpha_l l_t + (\alpha_m - 1) m_t + \omega_t + \log \mathcal{E} - \log \rho_t = 0.$$

Next, subtracting the production function $f(k_t, l_t, m_t) + \omega_t + \epsilon_t$ from both sides we have:

$$\begin{aligned}
m_t + \log \rho_t - \log P_t - f(k_t, l_t, m_t) + \omega_t + \epsilon_t &= \log \alpha_m + \log \mathcal{E} - \epsilon_t \\
\iff m_t + \log \rho_t - \log P_t - y_t &= \log \alpha_m + \log \mathcal{E} - \epsilon_t \\
\iff \log M_t \rho_t - \log P_t Y_t &= \log \alpha_m + \log \mathcal{E} - \epsilon_t \\
\iff s_t \equiv \log \frac{M_t \rho_t}{P_t Y_t} &= \log \alpha_m + \log \mathcal{E} - \epsilon_t.
\end{aligned}$$

Since $E\left[\epsilon_t\right] = 0$ by assumption, the term $\log \alpha_m + \log \mathcal{E}$ is trivially identified. Next, since $\mathcal{E} = E\left[e^{\epsilon_t}\right]$ by definition, we have:

$$\mathcal{E} = E\left[e^{\log \alpha_m + \log \mathcal{E} - s_t}\right],$$

which then allows us to identify $\mathcal{E}$. With that, we can identify $\log \alpha_m$ (and α_m) by subtracting $\log \mathcal{E}$ from the term $\log \alpha_m + \log \mathcal{E}$ we identified earlier.

Given α_m we can identify α_k and α_l by multiplying the coefficients on k_t and l_t by $1 - \alpha_m$ in Equation (2), so all the parameters of the production function are identified. Similarly, as noted before, δ_1 is identified since it equals the coefficient on m_{t-1} in Equation (2). Finally, since ω_t is identified, δ_0 in the production process can also be identified by estimating the AR(1) process.

Problem 2:

1.

The sample statistics are reported in Tables 1 and 2.

Table 1: Summary Statistics of Key Variables (Unbalanced)

Statistic	Count	Mean	Std	Min	25%	50%	75%	Max
Output	11393.0	13.8	1.8	8.4	12.3	13.6	15.3	20.5
Investment	11393.0	8.8	4.2	0.0	7.6	9.8	11.7	17.1
Capital	11393.0	11.9	2.1	8.0	10.1	11.8	13.7	18.8
Total Effective Hours	11393.0	4.9	1.4	2.2	3.7	4.6	6.1	10.1
Intermediate Consumption	11393.0	13.3	2.0	6.4	11.7	13.2	14.8	20.3

Table 2: Number of Firms and Zero Usage by Industry-Year (Unbalanced)

Industry	Year	Total Firms	Zero Investment	Zero Labor	Zero Materials
1	1990	14	1	0	0
1	1991	18	0	0	0
1	1992	23	1	0	0
1	1993	24	3	0	0
1	1994	29	4	0	0
1	1995	32	4	0	0
1	1996	34	2	0	0
1	1997	35	3	0	0
1	1998	30	1	0	0
1	1999	29	1	0	0
2	1990	36	4	0	0
2	1991	60	11	0	0
2	1992	75	17	0	0
2	1993	74	15	0	0
2	1994	85	16	0	0
2	1995	86	18	0	0
2	1996	91	16	0	0
2	1997	91	12	0	0
2	1998	88	6	0	0
2	1999	71	7	0	0
3	1990	50	5	0	0
3	1991	68	3	0	0
3	1992	86	7	0	0
3	1993	84	11	0	0
3	1994	92	9	0	0
3	1995	92	9	0	0
3	1996	88	2	0	0
3	1997	88	6	0	0
3	1998	85	1	0	0
3	1999	71	1	0	0
4	1990	61	7	0	0
4	1991	102	16	0	0
4	1992	124	24	0	0
4	1993	112	24	0	0
4	1994	119	32	0	0
4	1995	123	19	0	0
4	1996	125	24	0	0
4	1997	163	20	0	0
4	1998	160	16	0	0
4	1999	137	13	0	0
5	1990	35	3	0	0
5	1991	53	8	0	0
5	1992	58	9	0	0
5	1993	63	19	0	0
5	1994	75	17	0	0
5	1995	80	10	0	0
5	1996	84	13	0	0
5	1997	93	16	0	0
5	1998	90	9	0	0
5	1999	76	6	0	0

Note: Only the first five industries are reported due to space constraints. For a full list of industries, see the attached code file.

2.

The sample statistics are reported in Tables 3 and 4. Output, investment, capital, total effective hours, and intermediate consumption are all on average higher for the balanced

sample than for the unbalanced table. This suggests that there is selection going on in the sample of firms: Firms that enter or exit within the panel are on average smaller (as determined by their input use and output). The balanced panel has fewer firms in general (as expected, since there is exit) and has fewer firms with zero investment periods than the unbalanced panel.[1]

Table 3: Summary Statistics of Key Variables (Balanced)

Statistic	Count	Mean	Std	Min	25%	50%	75%	Max
Output	2470.0	14.0	1.6	8.9	12.6	14.1	15.4	18.1
Investment	2470.0	9.4	3.8	0.0	8.5	10.3	11.9	16.0
Capital	2470.0	12.3	1.9	8.0	10.8	12.2	13.9	17.6
Total Effective Hours	2470.0	5.2	1.3	2.9	4.0	5.1	6.3	9.0
Intermediate Consumption	2470.0	13.5	1.8	6.8	12.1	13.6	14.9	17.8

[1]The exiting firms are also likely to have lower productivities, though that is not apparent from simply comparing the summary statistics.

Table 4: Number of Firms and Zero Usage by Industry-Year (Balanced)

Industry	Year	Total Firms	Zero Investment	Zero Labor	Zero Materials
1	1990	7	0	0	0
1	1991	6	0	0	0
1	1992	7	0	0	0
1	1993	7	0	0	0
1	1994	7	0	0	0
1	1995	7	1	0	0
1	1996	7	0	0	0
1	1997	6	1	0	0
1	1998	6	0	0	0
1	1999	6	0	0	0
2	1990	21	2	0	0
2	1991	21	3	0	0
2	1992	21	2	0	0
2	1993	21	3	0	0
2	1994	21	3	0	0
2	1995	21	4	0	0
2	1996	21	2	0	0
2	1997	21	4	0	0
2	1998	21	1	0	0
2	1999	21	1	0	0
3	1990	15	1	0	0
3	1991	14	0	0	0
3	1992	14	0	0	0
3	1993	14	0	0	0
3	1994	14	0	0	0
3	1995	14	0	0	0
3	1996	14	0	0	0
3	1997	14	0	0	0
3	1998	14	0	0	0
3	1999	14	0	0	0
4	1990	27	1	0	0
4	1991	26	4	0	0
4	1992	26	6	0	0
4	1993	26	5	0	0
4	1994	26	4	0	0
4	1995	27	6	0	0
4	1996	27	7	0	0
4	1997	28	3	0	0
4	1998	27	5	0	0
4	1999	27	4	0	0
5	1990	15	0	0	0
5	1991	15	1	0	0
5	1992	14	0	0	0
5	1993	14	0	0	0
5	1994	14	1	0	0
5	1995	14	1	0	0
5	1996	14	0	0	0
5	1997	14	0	0	0
5	1998	14	1	0	0
5	1999	13	1	0	0

Note: The within-industry sample is not balanced due to firms switching industries over time. Only the first five industries are reported due to space constraints. For a full list of industries, see the attached code file.

3.

Table 5: Estimates for α_k, α_l, α_m, and the Constant Across Estimators (Balanced)

Estimator	α_k	α_l	α_m	Constant
OLS	0.106	0.283	0.624	2.780
	(0.010)	(0.016)	(0.016)	(0.120)
Fixed Effects	0.080	0.190	0.608	3.816
	(0.030)	(0.055)	(0.029)	(0.405)
First Difference	0.071	0.205	0.438	0.009
	(0.040)	(0.101)	(0.043)	(0.008)
Long Difference	0.060	0.200	0.670	0.051
	(0.032)	(0.060)	(0.029)	(0.015)
Random Effects	0.093	0.279	0.624	2.914
	(0.040)	(0.042)	(0.042)	(0.251)

For all results in this problem set focusing on one industry, we use industry number 13.

There is quite a lot of variation across the estimates when using different models. The OLS model is very likely to lead to wrong results, since it doesn't account for the endogeneity problem inherent in production function estimation. The fixed effects, first difference, and long difference estimates are broadly similar for α_k and α_l, as expected. All three methods are valid if the productivity term is constant across time for each firm, which is still unlikely to be true in reality.[2] The random effects estimator aims to improve efficiency over the fixed effects estimator, relying on the assumption that unobservable time-invariant factors (that include productivity, if it is assumed to be invariant over time) are mean independent with the covariates (capital, labor, materials) at all time periods. While the constant-over-time productivity assumption is unrealistic in the first place, the additional assumption required by the random effects estimator seems even more unrealistic.

With the balanced table, the Hausmann test cannot reject the null ($p = 0.0534$) of the difference in the coefficients not being systematic, which suggests that the random effects model might be appropriate here (if productivity doesn't change over time in the first place), and since it is more efficient than the FE estimator, it should be preferred. Still, the p-value is very low, so this interpretation is likely to be overly optimistic. It should be noted that even applying the Hausmann test implicitly assumes that the underlying model is correct, which in this case means that the firm productivities do not vary over time. Again, this is highly unrealistic in practice.

[2]Note that the estimation sample changes across these estimators, since the first differenced variables cannot be defined for observations in the first year of the panel (and similarly for the first four years in the long difference version).

4.

Table 6: Estimates for α_k, α_l, α_m, and the Constant Across Estimators (Unbalanced)

Estimator	α_k	α_l	α_m	Constant
OLS	0.061	0.340	0.594	3.440
	(0.007)	(0.011)	(0.010)	(0.077)
Fixed Effects	0.073	0.305	0.491	5.067
	(0.019)	(0.039)	(0.027)	(0.380)
First Difference	0.088	0.265	0.368	0.014
	(0.028)	(0.064)	(0.030)	(0.006)
Long Difference	0.049	0.302	0.615	0.086
	(0.024)	(0.050)	(0.041)	(0.014)
Random Effects	0.078	0.350	0.552	3.677
	(0.017)	(0.025)	(0.020)	(0.143)

Comparing the estimates obtained using the balanced and the unbalanced panels, we can see that estimates for α_l are larger, and the estimates for α_m and α_k smaller in the latter. For the capital coefficient, larger estimates in the balanced panel could suggest that firms that stay in the sample have more stable and larger capital (and are more capital-intensive). The larger coefficients on labor and smaller on materials might imply that for entering and exiting firms, the production process is more labor-intensive and less material-intensive.

With the unbalanced table, the Hausmann test strongly rejects ($p < 0.001$) the null of the difference in the coefficients not being systematic, which implies that the random effects model is not appropriate here and leads to inconsistent estimates (even if productivity is time-invariant).

5.

Arellano and Bond (1991), Blundell and Bond (1999), Olley and Pakes (1996), Levinsohn and Petrin (2006)

The coefficients estimated using Arellano and Bond (1991), Blundell and Bond (1999), Olley and Pakes (1996), and Levinsohn and Petrin (2006) are included in Table 7. We discuss the differences in the results across all models in the last part of our answers.

Table 7: Estimates for α_k, α_l, α_m, and the Constant Across Estimators (Unbalanced)

Estimator	α_k	α_l	α_m	Constant
Arellano and Bond (1991)	0.289	0.575	-	3.621
	(0.078)	(0.112)	-	(1.646)
Blundell and Bond (1999)	0.226	0.561	-	3.641
	(0.066)	(0.082)	-	(1.097)
Olley and Pakes (1996)	0.080	0.326	0.596	-
	(0.037)	(0.021)	(0.018)	-
Levinsohn and Petrin (2006)	0.291	0.612	-	-
	(0.075)	(0.037)	-	-

Note: The dependent variable for the Olley and Pakes (1996) estimator is log output. For all other estimators, it is log value added output. Some estimates cannot be identified for all the methods and are thus omitted.

Ackerberg, Caves, and Frazer (2015)

All of these estimates are created using our script `main.py`, which calls the ACF and GNR routines in `ACF_GNR_estimation_functions.py` and helper functions in `source_functions.py` For bootstrapping, we use 500 samples in which we draw the **firms** with replacement (not the firm-years). Drawing by firm preserves the time series properties of the data.[3] We create confidence intervals by calculating quantiles of the estimated coefficients from the bootstrap samples.

Table 8: ACF Estimation Results

Parameter	2.5%	25%	Point Estimate	75%	97.5%	Standard Error
β_0	-3.76e-15	-1.23e-15	-6.23e-17	1.14e-15	3.50e-15	9.16e-17
β_k	0.195	0.282	0.324	0.389	0.501	0.00357
β_l	0.431	0.635	0.732	0.798	0.911	0.00612
ρ	0.794	0.860	0.882	0.907	0.943	0.00168
$E\omega$	4.260	4.788	5.071	5.289	5.767	0.01744

The coefficients for $\beta_k = 0.32$ and $\beta_l = 0.73$ are quite close to the ACF estimates from the lecture slides (Table 1 of GNR). Adding up our point estimates for β_k and β_l gives us 1.056, indicating returns to scale are approximately constant. The estimate of $\rho = 0.88$ suggests a relatively persistent productivity process. We estimate an average productivity coefficeint of $\omega = 5.071$, which seems high. All coefficients are statistically significant according to the bootstrap 95% confidence interval.

[3]We first tried the bootstrap by sampling firm-years before realizing this, and ρ was biased downwards.

Gandhi, Navarro, and Rivers (2017)

These estimates also come from running `main.py`.

Table 9: GNR Estimation Results

Parameter	2.5%	25%	Point Estimate	75%	97.5%	Standard Error
$\beta_{0,cd}$	-7.552	-2.847	-0.211	2.044	6.156	0.164
$\beta_{k,cd}$	0.139	0.172	0.196	0.209	0.245	0.00124
$\beta_{l,cd}$	0.247	0.290	0.312	0.340	0.383	0.00157
$\beta_{m,cd}$	0.436	0.456	0.468	0.483	0.511	0.00090
$E_{df/dm}$	0.541	0.560	0.570	0.578	0.597	0.00063
$E\omega$	-2.437	1.532	3.840	6.499	11.141	0.16521

In the table, the last two rows $E_{df/dm}$ and $E\omega$ are the sample averages of the estimated materials-output elasticity and of productivity, respectively. $E_{df/dm}$ is quite close to our esimtates from OLS, the dynamic panel methods, and Olley and Pakes. The point estimate for $E\omega$ is smaller in GNR than in ACF, but the confidence interval is quite wide.

For easy comparison with our other results, and fast calculation, we assume Cobb-Douglas form and run OLS of the nonparametrically-estimated f on $1, k, l, m$. This gives us $\beta_{0,cd}, \beta_{k,cd}$, $\beta_{l,cd}, \beta_{m,cd}$.[4] Our estimates for capital and labor are larger than those of the dynamic panel methods. Again, returns to scale are approximately constant at 0.976.

Comparison

One key difference in the assumptions of dynamic panel methods and structural methods such as ACF and GNR is that the structural methods do not allow for unobserved price shocks to intermediate inputs m_{it}, while dynamic panel methods do allow for these shocks.

We are reluctant to read too much into differences in these models, since we don't know the details of the industry we are analyzing. If in this industry, we expect idiosyncratic input price shocks, and think it is reasonable to assume productivity follows an AR(1) process, then dynamic panel methods would be preferred. If the materials market is such that firms tend to face very similar prices, or we expect productivity to follow a flexible process that is different from AR(1), then the structural methods would be preferred.[5] The structural methods are also more useful for counterfactual analysis.

In general, our ACF/GNR codes have larger coefficients on capital than the OLS, dynamic panel, and OP/LP estimates. OP and LP suffer from the functional dependence problem,

[4] We considered using some kind of numerical differentiation method to get the nonparametric average elasticities. This slowed down the estimation, which would have made bootstrapping very slow.

[5] In our ACF code, we assume productivity process is an AR(1). In GNR, it is a flexible polynomial (with no intercept) in k and l.

that is, the labor coefficient is not identified if labor is perfectly flexible. This could drive some of the difference if labor markets in this industry are relatively flexible.

Overview of the ACF and GNR Methods

For a more detailed exposition, see the Jupyter notebooks in the folder attached to the submission. All scripts are included at the end of this document as well, but those do not attempt to explain the code in detail.

ACF

We used the version of ACF with two stages and two moments to estimate the coefficients. In the first stage, we approximate Φ by regressing y_{it} on (k_{it}, l_{it}, m_{it}) on a degree-3 polynomial, to obtain $\hat{\tilde{\Phi}}_t(k_{it}, l_{it}, m_{it})$. After calculating $\hat{\tilde{\Phi}}$ and its lagged value, we use the method of concentrating out additional moments. Let:

$$\tilde{\beta}_0 + \widehat{\omega_{it}(\beta_k, \beta_l)} = \widehat{\tilde{\Phi}_t}(k_{it}, l_{it}, m_{it}) - \beta_k k_{it} - \beta_l l_{it}$$

Then, we regress $\tilde{\beta}_0 + \widehat{\omega_{it}(\beta_k, \beta_l)}$ on $\tilde{\beta}_0 + \widehat{\omega_{it-1}(\beta_k, \beta_l)}$, noting that the residuals of this regression are the implied values of the innovations to productivity, that is, $\hat{\xi}_{it}(\beta_k, \beta_l)$.
Notice that this regression implicitly makes these innovations mean zero and uncorrelated with $\omega_{it-1}(\beta_k, \beta_l)$, so it is similar to enforcing the first and fourth moments in the "four moments" version of this etimation. Then, we optimize over β_k and β_l to satisfy the moment conditions:

$$E\left[\hat{\xi}_{it}(\beta_k, \beta_l) \otimes \begin{pmatrix} k_{it} \\ l_{it-1} \end{pmatrix}\right] = 0.$$

In the optimization routine, we use Autograd to calculate the gradient. From here, we can recover β_k and β_l. Finally, c and ρ are the coefficients of the regression of $\tilde{\beta}_0 + \widehat{\omega_{it}(\beta_k, \beta_l)}$ on $\tilde{\beta}_0 + \widehat{\omega_{it-1}(\beta_k, \beta_l)}$, and we can use these to recover ω_{it} and β_0.

ACF Joint Estimation Attempt

We attempted ACF joint estimation, but it does not converge. The attached notebook `ACF_joint.ipynb` in the `jupyter_notebooks_for_exposition` folder shows that we calculated the analytic gradient and set up the optimization problem. The joint estimation script is close to being complete, but there is probably a minor bug somewhere.

GNR

First, we want to estimate the elasticity $D^{\mathcal{E}}(k_{jt}, l_{jt}, m_{jt}) = \text{Polynomial}(k_{jt}, l_{jt}, m_{jt})$. To find the coefficients of the polynomials, we run the estimator:

$$\min_{\gamma'} \sum_{j,t} \left\{ s_{jt} - \ln \underbrace{\begin{pmatrix} \gamma'_0 + \gamma'_k k_{jt} + \gamma'_l l_{jt} + \gamma'_m m_{jt} + \gamma'_{kk} k_{jt}^2 + \gamma'_{ll} l_{jt}^2 \\ \gamma'_{mm} m_{jt}^2 + \gamma'_{kl} k_{jt} l_{jt} + \gamma'_{km} k_{jt} m_{jt} + \gamma'_{lm} l_{jt} m_{jt} \end{pmatrix}}_{D^{\mathcal{E}}} \right\}^2$$

where $s_{jt} \equiv \ln\left(\frac{\rho_t M_{jt}}{P_t Y_t}\right)$ is the log intermediate share of output. Then using the gamma coefficients, we get:

$$\hat{D}_{jt} \equiv D^{\mathcal{E}}(k_{jt}, l_{jt}, m_{jt}) = \text{Polynomial}(k_{jt}, l_{jt}, m_{jt} | \vec{\gamma}')$$

From there, we get the residuals

$$\hat{\epsilon}_{jt} = \ln \hat{D}_{jt} - s_{jt}$$

Next, we estimate $\widehat{\mathcal{E}} = \frac{1}{JT} \sum_{j,t} e^{\hat{\epsilon}_{jt}}$. Then:

$$\begin{aligned} \ln \hat{D}^{\mathcal{E}}_{jt} &= \ln \hat{\mathcal{E}} + \ln\left(\frac{\partial}{\partial m_{jt}} f_{jt}\right) \\ \iff \hat{D}^{\mathcal{E}}_{jt} \cdot \frac{1}{\hat{\mathcal{E}}} &= \frac{\partial}{\partial m_{jt}} f_{jt} \\ \iff \hat{X}_{\text{poly}} \cdot \frac{\vec{\gamma}'}{\hat{\mathcal{E}}} &= \frac{\partial}{\partial m_{jt}} f_{jt} \end{aligned}$$

That is, we can use the same polynomial design matrices used to estimate $\hat{D}^{\mathcal{E}}_{jt}$, but just divide the gamma coefficients by $\widehat{\mathcal{E}}$, to get the $\frac{\partial}{\partial m_{jt}} f_{jt}$. Next we want to recover the constant of integration in:

$$\int \frac{\partial}{\partial m_{jt}} f(k_{jt}, l_{jt}, m_{jt}) dm_{jt} = f(k_{jt}, l_{jt}, m_{jt}) + \mathcal{C}(k_{jt}, l_{jt})$$

This amounts to integrating the polynomial using $\gamma = \gamma'/\hat{\mathcal{E}}$:

$$\text{Integral} \equiv \mathcal{D}(k_{jt}, l_{jt}, m_{jt}) = \begin{pmatrix} \gamma_0 + \gamma_k k_{jt} + \gamma_l l_{jt} + \frac{\gamma_m}{2} m_{jt} + \gamma_{kk} k_{jt}^2 + \gamma_{ll} l_{jt}^2 \\ \frac{\gamma_{mm}}{3} m_{jt}^2 + \gamma_{kl} k_{jt} l_{jt} + \frac{\gamma_{km}}{2} k_{jt} m_{jt} + \frac{\gamma_{lm}}{2} l_{jt} m_{jt} \end{pmatrix} m_{jt}$$

Using the above $\mathcal{D}(\cdot)$, we can calculate $\mathcal{Y}_{jt} \equiv y_{jt} - \epsilon_{jt} - \mathcal{D}_{jt} = -\mathcal{C}(k_{jt}, l_{jt}) + \omega_{jt}$:

$$\hat{\mathcal{Y}}_{jt} = \ln\left(\frac{Y_{jt}}{e^{\hat{\epsilon}_{jt}} e^{\hat{\mathcal{D}}_{jt}}}\right).$$

Moments

We approximate ω_{it} with a function $h(\cdot)$:

$$\mathcal{Y}_{jt} = -\mathcal{C}(k_{jt}, l_{jt}) + h(\mathcal{Y}_{jt-1} + \mathcal{C}(k_{jt-1}, l_{jt-1})) + \eta_{jt}.$$

In this case we use:

$$\mathcal{C}(k_{jt}, l_{jt}) = \text{Polynomial}(k_{jt}, l_{jt}, \text{degree}_C),$$

and

$$h_A(\omega_{jt-1}) = \text{Polynomial}(\omega_{jt-1}, \text{degree}_\omega) = \text{Polynomial}(\mathcal{Y}_{jt-1} + \mathcal{C}(k_{jt-1}, l_{jt-1})), \text{degree}_\omega) + \eta_{jt}.$$

All together we have:

$$\hat{\mathcal{Y}}_{jt} = - \sum_{0<\tau_k+\tau_l\leq\tau} \alpha_{\tau_k,\tau_l} k_{jt}^{\tau_k} l_{jt}^{\tau_l} + \sum_{0\leq a\leq A} \delta_a \left(\hat{\mathcal{Y}}_{jt-1} + \sum_{0<\tau_k+\tau_l\leq\tau} \alpha_{\tau_k,\tau_l} k_{jt-1}^{\tau_k} l_{jt-1}^{\tau_l} \right)^a + \eta_{jt}$$

The moments to estimate are:

$$\begin{aligned} E\left[\varepsilon_{jt} \frac{\partial \ln D_r\left(k_{jt}, l_{jt}, m_{jt}\right)}{\partial \gamma}\right] &= 0 \\ E\left[\eta_{jt} k_{jt}^{\tau_k} l_{jt}^{\tau_t}\right] &= 0, \\ E\left[\eta_{jt} \mathcal{Y}_{jt-1}^a\right] &= 0, \end{aligned} \tag{3}$$

The "Concentrate Out" Method

We can "concentrate out" the first moment in Equation (3) by running the nonlinear least squares regression of of the share equation s_{jt} on $\ln D_{jt}$. We can concentrate out the third moment with the following procedure:

1. Guess α.
2. Form $\omega_{jt-1}(\alpha) = \hat{\mathcal{Y}}_{jt-1} + \text{Poly}^{\alpha}_{\text{no intercept}}(k_{jt-1}, l_{jt-1})$.
3. Calculate $\hat{\omega}_{jt} = \hat{\mathcal{Y}}_{jt} + \text{Poly}^{\alpha}_{\text{no intercept}}(k_{jt-1}, l_{jt-1})$, plugging in the guessed regression coeficients α and multiplying them with the polynomial design matrix to get the predicted omega.
4. Polynomial-regress $\hat{\omega}_{jt} \sim \hat{\omega}_{jt-1}$ to get the innovation $\hat{\eta}_{jt}$.
5. Use the moments $\mathbb{E}\left[\hat{\eta}_{jt} X_{jt}\right] = 0$ for all $X \in [k, l, k^2, l^2, kl, ...]$.

We solved this using the L-BFGS-B algorithm, supplying a gradient from Autograd. Note that there is no intercept in the approximation of the constant of integration. Even if we put an intercept in, it would not be identified.

Now we have α, which lets us recover the constant of integration, and we have δ (which is just the regression coefficients from regression ω on the polynomial of lagged ω), which lets us recover the productivity ω_{jt}. The production function is non-parametrically identified by:

$$f(k_{jt}, l_{jt}, m_{jt}) = \mathcal{D}_{jt} - \mathcal{C}_{jt}$$

that is, it's equal to the "integral" minus the "constant of integration."

Elasticity Estimates

We have the nonparametric $\frac{\partial}{\partial m_{jt}} f_{jt}$ from previous parts of the estimation. For comparability with ACF and panel methods, we can assume Cobb-Douglas form to get predicted elasticities for capital, labor, and materials:

$$f(k_{jt}, l_{jt}, m_{jt}) = \beta_0 + \beta_k k_{jt} + \beta_l l_{jt} + \beta_m m_{jt}.$$

Then, we can get the elasticities by running OLS of $f(k_{jt}, l_{jt}, m_{jt})$ on $1, k_{jt}, l_{jt}, m_{jt}$.

Outline of ACF/GNR code submission

Main code

- `main.py`: Runs the ACF and GNR estimation, including bootstrapping.
- `ACF_GNR_estimation_functions.py`: Contains functions for estimating the ACF and GNR models.
- `source_functions.py`: Contains helper functions called by the ACF and GNR estimation functions, as well as bootstrapping and summary statistics.

Jupyter Notebooks (for exposition, not used to produce results)

- See the folder `jupyter_notebooks_for_exposition`.
- `ACF_notebook_2step_2moments.ipynb`: Contains the version of ACF we used for our results in the form of a Jupyter notebook
- `ACF_notebook_joint.ipynb`: Contains the failed attempt to run ACF jointly (we got pretty close, but there's probably a small bug).
- `GNR_notebook.ipynb`: Contains the version of GNR we used for our results, in the form of a Jupyter notebook.

See a copy of our code on the next pages.

Code

Producing the Summary Statistics (Python)

```python
#!/usr/bin/env python
# coding: utf-8

# # Problem Set 3

# ### Preliminaries

# Load the needed libraries and the data.

import pandas as pd
import numpy as np
import re

df = pd.read_stata('PS3_data.dta')


# Rename variables

# Use the given codebook to rename variables.

labels = {
    "X03": "Output",
    "X04": "Industry_1_dummy",
    "X05": "Industry_2_dummy",
    "X06": "Industry_3_dummy",
    "X07": "Industry_4_dummy",
    "X08": "Industry_5_dummy",
    "X09": "Industry_6_dummy",
    "X10": "Industry_7_dummy",
    "X11": "Industry_8_dummy",
    "X12": "Industry_9_dummy",
    "X13": "Industry_10_dummy",
    "X14": "Industry_11_dummy",
    "X15": "Industry_12_dummy",
    "X16": "Industry_13_dummy",
    "X17": "Industry_14_dummy",
    "X18": "Industry_15_dummy",
    "X19": "Industry_16_dummy",
    "X20": "Industry_17_dummy",
    "X21": "Industry_18_dummy",
```

```
    "X22": "200_workers_and_fewer",
    "X23": "More_than_200_workers",
    "X24": "Year_1990_dummy",
    "X25": "Year_1991_dummy",
    "X26": "Year_1992_dummy",
    "X27": "Year_1993_dummy",
    "X28": "Year_1994_dummy",
    "X29": "Year_1995_dummy",
    "X30": "Year_1996_dummy",
    "X31": "Year_1997_dummy",
    "X32": "Year_1998_dummy",
    "X33": "Year_1999_dummy",
    "X34": "Merger_dummy",
    "X35": "Scission_dummy",
    "X36": "RD_expenditure",
    "X37": "Process_innovation_dummy",
    "X38": "Product_innovation_dummy",
    "X39": "Investment",
    "X40": "Capital",
    "X41": "Number_of_workers",
    "X42": "Effective_hours_per_worker",
    "X43": "Total_effective_hours",
    "X44": "Intermediate_consumption",
    "X45": "Output_price_index",
    "X46": "Consumer_price_index",
    "X47": "Region_of_industrial_employment",
    "X48": "Hourly_wage",
    "X49": "Materials_price_index",
    "X50": "Proportion_of_temporary_workers",
    "X51": "Proportion_of_white_collar_workers",
    "X52": "Proportion_of_engineers_and_graduates",
    "X53": "Proportion_of_non_graduates",
    "X54": "Technological_sophistication",
    "X55": "Market_dynamism_index",
    "X56": "Incorporated",
    "X57": "Ownership_control_identification",
    "X58": "Firm_age",
    "X59": "NACE_code",
    "X60": "Entrant_firm_dummy",
    "X61": "Exiting_firm_dummy"
}

df.rename(columns=labels, inplace=True)
```

```python
# Sample Statistics

ind_dummies = [c for c in df.columns if c.startswith('Industry_')]
years = [y for y in range(1990, 2000)]

def print_sample_stats(varlist, balanced=False):
    if balanced:
        data = df[df['obs'] == 10].copy()
    else:
        data = df.copy()
    for var in varlist:
        print('**************************')
        print(var.replace('_', ' '))
        print('**************************\n')
        print(data[var].describe())
        print()

# Industry-year firm and non-zero variable counts

def print_industry_year_stats(balanced=False):
    if balanced:
        data = df[df['obs'] == 10].copy()
    else:
        data = df.copy()
    for ind in ind_dummies:
        print(f'Industry: {re.sub("[^0-9]", "", ind)}')
        for year in years:
            print(
                f"Year {str(year)}: \
                No. of observations: {str(len(data[(data[ind] == 1) &
                ↪  (data['year'] == year)]))}, \
                Non-zero investment: {str(len(data[(data[ind] == 1) &
                ↪  (data['year'] == year) & (data['Investment'] > 0)]))},
                ↪  \
                Non-zero hours: {str(len(data[(data[ind] == 1) &
                ↪  (data['year'] == year) &
                ↪  (data['Total_effective_hours'] > 0)]))}, \
                Non-zero materials: {str(len(data[(data[ind] == 1) &
                ↪  (data['year'] == year) &
                ↪  (data['Intermediate_consumption'] > 0)]))}"
            )

# Variables of interest
```

```
variables = ['Output', 'Investment', 'Capital', 'Total_effective_hours',
↪  'Intermediate_consumption']


#### Unbalanced panel

print_sample_stats(variables)

print_industry_year_stats()


### Balanced panel

print_sample_stats(variables, balanced=True)

print_industry_year_stats(balanced=True)


# NOTE: Seems like firms are switching industries, which causes the panel
↪  not to be balanced within industries!

# Export the data to .dta for Stata parts of the code

df.to_stata('PS3_data_clean.dta', write_index=False)
```

Estimating the Production Function Using "Simpler" Methods (Stata)

```
* Set path
cd "../PS3/"

* Load the cleaned data (with variable names)
clear all
use "PS3_data_clean.dta"

        * Pick industry 13
        keep if Industry_13_dummy == 1

        * Rename the variables to shorter versions
        rename Output Y
        rename Capital K
        rename Total_effective_hours L
        rename Intermediate_consumption M
        rename Investment I
```

```
* NOTE: The variables are already logged, so we can just proceed with
↪  estimation

* BALANCED

        preserve

        * Since there's industry switching, ensure that the panel is
        ↪  balanced within the industry
        bysort firm_id: keep if _N == 10

        * OLS

        reg Y K L M, vce(robust)

        * Fixed Effects

        reg Y K L M i.firm_id i.year, vce(robust)

        * First Differences

        xtset firm_id year

        reg d.Y d.K d.L d.M, vce(robust)

        * Long Differences

        foreach var in Y K L M {
                gen `var'_ld = `var' - L5.`var'
        }

        reg Y_ld K_ld L_ld M_ld, vce(robust)

        * Random Effects

        xtreg Y K L M i.year, re robust

        * Hausman Test
        qui xtreg Y K L M i.year, fe
        estimates store fixed_effects
        qui xtreg Y K L M i.year, re
        hausman fixed_effects ., sigmamore
```

```
        restore

* UNBALANCED

        * OLS

        reg Y K L M, vce(robust)

        * Fixed Effects

        reg Y K L M i.firm_id i.year, vce(robust)

        * First Differences

        xtset firm_id year

        reg d.Y d.K d.L d.M, vce(robust)

        * Long Differences

        foreach var in Y K L M {
                gen `var'_ld = `var' - L5.`var'
        }

        reg Y_ld K_ld L_ld M_ld, vce(robust)

        * Random Effects

        xtreg Y K L M i.year, re robust

        * Hausman Test
        qui xtreg Y K L M i.year, fe
        estimates store fixed_effects
        qui xtreg Y K L M i.year, re
        hausman fixed_effects ., sigmamore

* ARELLANO AND BOND (1991)

        * Create value added output
        gen Y_va = ln(exp(Y)*exp(Output_price_index) -
        ↪  exp(M)*exp(Materials_price_index))

        xtabond Y_va K L, lags(1) maxldep(1) maxlags(1) vce(robust)
```

```
* BLUNDELL AND BOND (1999)

        xtdpdsys Y_va K L, lags(1) maxldep(1) maxlags(1) vce(robust)

* OLLEY AND PAKES (1996)

        * Check whether there are gaps for firms
        bysort firm_id (year): gen last_year = year[_N]
        bysort firm_id (year): gen first_year = year[1]
        bysort firm_id: gen has_gaps = (last_year - first_year + 1) !=
        ↪  (_N)

        bysort firm_id: gen exit = (obs != 10) & (has_gaps == 0) & (year
        ↪  == last_year)
        replace exit = 0 if year == 1999

        opreg Y, exit(exit) state(K) proxy(I) free(L M) vce(bootstrap,
        ↪  seed(999) rep(250))

* LEVINSOHN AND PETRIN (2006)

        levpet Y_va, free(L) proxy(M) capital(K)
```

Estimating the Production Function Using ACF and GNR (Python)

Main script (main.py)

```
# -*- coding: utf-8 -*-
"""
Main script to create point estimates and bootstrapped standard errors
↪  for ACF and GNR
For bootstrapping, we sample FIRMS with replacement in order to preserve
↪  the time series properties of the data.

Some things are hard-coded within the estimation functions -- without
↪  time constraints, we would code this more elegantly.
"""
from ACF_GNR_estimation_functions import *


#==== Options
↪  ================================================================#
#Number of bootstrap samples
```

```
np.random.seed(9)
n_boot_samples = 100

#==== load_data
 ↪  ================================================================#

df_ACF = load_data("ACF")
df_GNR = load_data("GNR")

#==== point estimate for ACF
 ↪  ================================================#
print("-------------------------------------------------")
print("-----------ACF: Getting Point Estimates -----------")
print("-------------------------------------------------")
theta0 = np.array([1,1])/2 #initial guess for ACF parameters
coeffs_ACF, convergence = ACF_estimation(df_ACF, theta0, print_results=1)
theta_ACF = coeffs_ACF[1:3]

#==== point estimate for GNR
 ↪  ================================================#
#initial guesses for GNR parameters.
print("-------------------------------------------------")
print("-----------GNR: Getting Point Estimates -----------")
print("-------------------------------------------------")

alpha0 = np.ones(5)/2 #This is the required size to have coefficeints for
 ↪  k, l, kl, k**2, l**2. Need to change if the degree is changed
gammaprime0 = np.ones(10)/2  #Also needs to change if the degree is
 ↪  changed
initial_guesses0 = (alpha0, gammaprime0)

results_params_GNR, results_convergence_GNR, alpha_GNR, gammaprime_GNR =
 ↪  GNR_estimation(df_GNR, initial_guesses0, print_results = 1)
initial_guesses_GNR = (alpha_GNR, gammaprime_GNR)

#==== Run bootstrap for ACF, save results
 ↪  ==================================#
print("-------------------------------------------------")
print("-----------ACF: Bootstrapping Standard Errors------")
print("-------------------------------------------------")
bootstrap_results_ACF, convergence_ACF = bootstrap(ACF_estimation,
 ↪  theta_ACF, df_ACF, n_boot_samples)

#Summarize array
```

```
ACF_row_names = np.array(["beta_0", "beta_k", "beta_l", "rho", "Eomega",
↪  "gmm_error"])
ACF_summary = summarize_array(coeffs_ACF, bootstrap_results_ACF,
↪  ACF_row_names[:-1])

boot_full_ACF = pd.DataFrame(np.hstack((bootstrap_results_ACF,
↪  convergence_ACF)), columns = ACF_row_names)

#Save to CSV
ACF_summary.to_csv("../Results/summary_stats_ACF.csv")
boot_full_ACF.to_csv("../Results/full_bootstrap_ACF.csv")

#==== Run bootstrap for GNR, save results
↪  ====================================#
#Use the initial condition from the true data to improve speed and
↪  convergence.
print("-----------------------------------------------------")
print("-----------GNR: Bootstrapping Standard Errors------")
print("-----------------------------------------------------")
bootstrap_results_GNR, convergence_GNR = bootstrap(GNR_estimation,
↪  initial_guesses_GNR, df_GNR, n_boot_samples, columns = 6)

#Summarize array
GNR_row_names = np.array(["beta_0_cd", "beta_k_cd", "beta_l_cd",
↪  "beta_m_cd", "Edf_dm", "Eomega", "gmm_error"])
GNR_summary = summarize_array(results_params_GNR, bootstrap_results_GNR,
↪  GNR_row_names[:-1])

boot_full_GNR = pd.DataFrame(np.hstack((bootstrap_results_GNR,
↪  convergence_GNR)), columns = GNR_row_names)

#Save to CSV
GNR_summary.to_csv("../Results/summary_stats_GNR.csv")
boot_full_GNR.to_csv("../Results/full_bootstrap_GNR.csv")
```

ACF_GNR_estimation_functions.py

```
# -*- coding: utf-8 -*-
"""
ACF Estimation
"""

from source_functions import *
```

```
def ACF_estimation(df, theta0, print_results = 0):

#=== options
 ↪  ==========================================================================#
    degree= 3 #polynomial fit degree
    #theta0 = np.array([1,1])     #Initial guess for parameters beta_k,
     ↪  beta_l
    W0 = np.eye(2)     #Weight matrix -- use the identity for now.

#=== Fit Phi
 ↪  ==========================================================================#
 ↪

    xvars = df[['k', 'l', 'm']].to_numpy()
    y = df[['y']].to_numpy()
    X_poly = poly_design_matrix(xvars, degree)

    Phi = regress(y, X_poly)[1]

    df["Phi"] = Phi
    #Add into the dataframe
    df['Phiprev'] = df.groupby('firm_id')['Phi'].shift(1)

#=== Drop NaNs after creating the lagged Phi. Define GMM arguments
 ↪  ===========#
    df_nonans = df.dropna()
    #Get all the variables out of the dataframe -- This allows me to use
     ↪  Autograd
    y = df_nonans['y'].to_numpy()
    k = df_nonans['k'].to_numpy()
    l = df_nonans['l'].to_numpy()
    Phi = df_nonans['Phi'].to_numpy()
    kprev = df_nonans['kprev'].to_numpy()
    lprev = df_nonans['lprev'].to_numpy()
    Phiprev = df_nonans['Phiprev'].to_numpy()
    #Run GMM
    #(2) Get matrix of variables used in exogeneity restrictions
    Vex = moment_ex_restrictions_ACF(k, lprev)

#=== Run GMM
 ↪  ==========================================================================#
 ↪
```

```
autogradient = grad(gmm_obj_ACF)

gmm_args = (y, k, l, kprev, lprev, Phi, Phiprev, Vex, W0)

tolerance = 1e-25

#theta_results = opt.minimize(gmm_obj_ACF, theta0, args=gmm_args,
#                              tol=tolerance, method='Nelder-Mead',
↪  options={'maxiter': 10000})

theta_results_grad = opt.minimize(gmm_obj_ACF, theta0, args=gmm_args,
                        tol=tolerance, jac=autogradient,
                        ↪  method='L-BFGS-B',
                         options={'ftol': tolerance, 'gtol': tolerance,
                         ↪  'maxiter': 20000})

theta=theta_results_grad.x
#Get the slope, rho. It's the slope of the regression used to find
↪  the moments.
rho = moment_error_ACF(theta, y, k, l, kprev, lprev, Phi,
↪  Phiprev)[1][1]

if print_results == 1:
    print("The gradient at the optimum is: ", autogradient(theta, y,
    ↪  k, l, kprev, lprev, Phi, Phiprev, Vex, W0))
    print("The GMM error using the gradient is:", gmm_obj_ACF(theta,
    ↪  y, k, l, kprev, lprev, Phi, Phiprev, Vex, W0))
    print("The estimates using autograd: [beta_k, beta_l] = ", theta)
    print("The slope of the AR(1) of productivity is: rho = ", rho)


#Calculate omega and beta0
xi, Rho, b0_plus_omega, b0_plus_omega_prev = moment_error_ACF(theta,
↪  y, k, l, kprev, lprev, Phi, Phiprev)

omegaprev = (b0_plus_omega-b0_plus_omega_prev - Rho[0] -
↪  xi)/(Rho[1]-1)
omega = Rho[0] + Rho[1]*omegaprev + xi

Eomega = np.mean(omega)

Ebeta0 = np.mean(b0_plus_omega-omega)
```

```
    results_coefficients = np.array([Ebeta0, theta[0], theta[1], rho,
    ↪  Eomega])
    results_convergence = gmm_obj_ACF(theta, y, k, l, kprev, lprev, Phi,
    ↪  Phiprev, Vex, W0)

    #df_nonans['omega'] = omega
    #df_nonans['omegaprev'] = omegaprev

    return results_coefficients, results_convergence

#============================================================================#
↪
def GNR_estimation(df, initial_guesses, print_results = 0):


    alpha0, gammaprime0 = initial_guesses

#=== options
↪  ============================================================#
    degree= 2
    degree_omega = 2

#=== Fit D_{jt}
↪  ============================================================#

    #Make the polynomial design matrix
    xvars = df[['k', 'l', 'm']].to_numpy()
    s = df[['s']].to_numpy()
    X_poly_D = poly_design_matrix(xvars, degree)
    #calculate the gradient of the objective function using AutoGrad
    autogradient_nlls = grad(nlls_share_obj)
    autohessian_nlls = hessian(nlls_share_obj)
    #initial guess
    #gammaprime0 = np.ones(X_poly_D.shape[1])/2

    #minimize to fit the coefficients gammaprime
    #Enforce that X@gamma is nonnegative, otherwise we get negative
    ↪  values in the log
    nonnegative_b = {'type': 'ineq', 'fun': lambda b: (X_poly_D@b)}

    gammaprime_results = opt.minimize(nlls_share_obj, gammaprime0,
    ↪  args=(X_poly_D, s),
                                      constraints = [nonnegative_b],
```

```
                            tol=1e-12, jac=autogradient_nlls, hess =
                            ↪  autohessian_nlls, method='trust-constr'
    )

    #print("The error is:",  gammaprime_results.fun)
    #print("The gradient is:",  gammaprime_results.grad)
    #print("The coefficients in the degree-1 fit are:",
    ↪  gammaprime_results.x)

    #shat = np.log(X_poly_D@gammaprime_results.x)
    gammaprime = gammaprime_results.x
    #Get Dhat, the elasticities
    df['Dhat'] = X_poly_D@gammaprime
    #Back out the residuals, epsilons
    df['epsilonhat'] = np.log(df['Dhat']) - df['s']
    # mean of epsilon is 1e-12 --- good sign
    CurlyEhat = (np.mean(np.exp(df['epsilonhat'])))
    #The theoretial guess for CurlyEhat given epsilon ~ N(0, sigma^2) is
    ↪  very close to the actual curlyEhat,
    #suggesting the epsilons are approximately normally distributed.
    #It follows from the math above that ...
    gamma = gammaprime/CurlyEhat
    df['df_dm'] = X_poly_D@gamma

#=== Evaluate the integral, CurlyD
↪  ==========================================#
#Then calculate some more objects needed for GMM estimation.
    #Get the design matrix associated with the integral of the polynomial
    X_poly_D_integral =  poly_integral_design_matrix(xvars, degree, w_r_t
    ↪  = 2)
    #Evaluate it to get curlyD, which is the integral of the log
    ↪  elasticities
    df['CurlyD'] = X_poly_D_integral@gamma
    #from here, get CurlyY
    df['CurlyY'] = df['y'] - df['epsilonhat'] - df['CurlyD']
    df['CurlyYprev'] = df.groupby('firm_id')['CurlyY'].shift(1)

    #Now, drop all NaNs
    df_nonans = df.dropna().copy()

    xvars_omega = df_nonans[["k", "l"]].to_numpy()
    xvars_prev_omega = df_nonans[["kprev", "lprev"]].to_numpy()
```

```
    #This polynomial fit has NO INTERCEPT. Even if we wanted an intercept
    ↪  it would not be identified because we end up taking first
    ↪  differences of omega.
    X_poly_omega = poly_design_matrix(xvars_omega, degree_omega)[:, 1:]
    Xprev_poly_omega = poly_design_matrix(xvars_prev_omega,
    ↪  degree_omega)[:, 1:]

    #Previous CurlyY
    CurlyY = df_nonans['CurlyY'].to_numpy()
    CurlyYprev = df_nonans['CurlyYprev'].to_numpy()

    #alpha0 = np.ones(X_poly_omega.shape[1])/2
    W0 = np.eye(len(alpha0))

#=== Run the GMM estimation
↪  ================================================#

    args_GNR = (X_poly_omega, Xprev_poly_omega, CurlyY, CurlyYprev, W0)

    tolerance = 1e-24

    gmm_results_GNR = opt.minimize(gmm_obj_fcn_GNR, alpha0, args=args_GNR,
                           tol=1e-24, jac=autogradient_GNR,
                           ↪  method='L-BFGS-B',
                           options={'ftol': tolerance, 'gtol': tolerance,
                           ↪  'maxiter': 20000}
    )

    alpha = gmm_results_GNR.x
    delta, eta = gmm_stage2_error_GNR(alpha, X_poly_omega,
    ↪  Xprev_poly_omega, CurlyY, CurlyYprev)[1:3]

    #Calculate the omegas
    df_nonans['ConstantC'] = X_poly_omega@alpha
    df_nonans['omega'] = df_nonans['ConstantC'] + df_nonans['CurlyY']

    Eomega = np.mean(df_nonans['omega'])
    Edf_dm = np.mean(df_nonans['df_dm'])


    #Assuming Cobb-Douglas, we can get elasticities with OLS
    df_nonans['f'] = df_nonans['CurlyD'] - df_nonans['ConstantC']
    #df_nonans['f_plus_epsilon'] = df_nonans['y'] - df_nonans['omega']
    f = df_nonans['f']
```

```
    #Run OLS
    klm = df_nonans[['k', 'l', 'm']]
    Xklm = np.hstack((np.ones((klm.shape[0],1)), klm.to_numpy()))
    fbeta_cobbdouglas, _, _ = regress(f, Xklm)

    if print_results == 1:
        print("The error is:",  gmm_results_GNR.fun)
        print("The gradient is:",  gmm_results_GNR.jac)
        print("The coefficients for the integration constant [alpha]
        ↪  are:",  gmm_results_GNR.x)
        print("The coefficients for productivity omega [delta] are:",
        ↪  delta)
        print("the average productivity [omega] is:", Eomega)
        print("the average elasticity [df/dm] is:", Edf_dm)
        print("----Assuming Cobb-Douglas----")
        print("[beta_0, beta_k, beta_l, beta_m] = ",
        ↪  fbeta_cobbdouglas.flatten())

    results_params = np.concatenate((fbeta_cobbdouglas.flatten(),
    ↪  [Edf_dm], [Eomega]))

    results_convergence = gmm_results_GNR.fun

    return results_params, results_convergence, alpha, gammaprime
```

source_functions.py

```
"""
Source functions for GNR and ACF estimations
Detailed Jupyter notebooks describing the logic of these functions are
↪  attached.
"""

#Source functions
import autograd.numpy as np
from autograd import grad, hessian
import pandas as pd
import scipy.optimize as opt
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import math
```

```
from itertools import combinations_with_replacement, chain #used for
↪  constructing polynomials
from numba import jit



#==== Used in ACF and GNR
↪  ==================================================#

#Load the data
def load_data(model):

    filename = "../PS3_data_changedtoxlsx.xlsx"
    df0 = pd.read_excel(filename)
    #Remove missing materials columns
    df = df0[['year', 'firm_id', 'X03', 'X04', 'X05', 'X16', 'X40', 'X43',
    ↪  'X44', 'X45', 'X49']]
    #new_names = ["year", "firm_id", "obs", "ly", "s01", "s02", "lc",
    ↪  "ll", "lm"]
    new_names = ["t", "firm_id", "y_gross", "s01", "s02", "s13", "k", "l",
    ↪  "m", 'py', 'pm']
    df.columns = new_names
    #Drop missing materials data
    df=df[df['m']!=0]
    #Keep industry 1 only
    df=df[df['s13']==1]

    if model == "ACF":
        #Creating value-added y
        df['y'] = np.log(np.exp(df['y_gross'] + df['py']) - np.exp(df['m']
        ↪  + df['pm']))
    elif model == "GNR":
        #in GNR, we simply use gross y
        df['y'] = df['y_gross']
        df['s'] = df['pm']+df['m'] - df['py'] - df['y']
    else:
        print("Please enter the string ACF or GNR" )

    #Creating lagged variables
    df = df.sort_values(by=['firm_id', 't'])
    df['kprev'] = df.groupby('firm_id')['k'].shift(1)
    df['lprev'] = df.groupby('firm_id')['l'].shift(1)
    df['mprev'] = df.groupby('firm_id')['m'].shift(1)
```

```
    return df



#Creates an iterator of tuples, useful for constructing polynomial
↪  regression design matrices.
def poly_terms(n_features, degree):
    #This thing creates an iterator structure of tuples, used to create
    ↪  polynomial interaction terms.
    #It looks something like this: (0,), (1,), (2,), (0, 0), (0, 1)
    polynomial_terms = chain(
        *(combinations_with_replacement(range(n_features), d) for d in
        ↪  range(1, degree+1))
    )
    return(polynomial_terms)

#Contructs a polynomial regression design matrix.
def poly_design_matrix(xvars, degree):
    if xvars.ndim == 1:
        xvars = xvars.reshape(1, -1)
    # Get the number of samples (n) and number of features (m) from X
    n_samples, n_features = xvars.shape
    # Start with a column of ones for the intercept term
    X_poly = np.ones((n_samples, 1))
    #Create iterator used to construct polynomial terms
    polynomial_terms = poly_terms(n_features, degree)
    # Generate polynomial terms and interaction terms up to 4th degree
    for terms in  polynomial_terms:  # For degrees 1 to 4
            #print(terms)
            X_poly = np.hstack((X_poly, np.prod(xvars[:, terms],
            ↪  axis=1).reshape(-1, 1)))
    # Compute the coefficients using the normal equation: beta = (X.T *
    ↪  X)^(-1) * X.T * y
    return X_poly

#Runs a regression
def regress(y, X):
    beta = np.linalg.solve(X.T@X, X.T@y)
    yhat = X@beta
    resids = y-yhat
    return beta, yhat, resids

#==== Used in ACF only
↪  ========================================================#
```

```
#Calculates the error term, h(theta, y, k, l)
def moment_error_ACF(theta, y, k, l, kprev, lprev, Phi, Phiprev):
    #get the innovations to omega
    beta_k = theta[0]
    beta_l = theta[1]
    b0_plus_omega = Phi - beta_k*k - beta_l*l
    b0_plus_omega_prev = Phiprev - beta_k*kprev - beta_l*lprev
    #Regress them to get the innovations
    yvar = b0_plus_omega#.reshape(-1, 1)
    xvar = b0_plus_omega_prev.reshape(-1, 1)
    #Degree of the Omega polynomial
    omega_degree = 1
    Xdesign = poly_design_matrix(xvar, omega_degree)
    #coeffs will contain rho, the AR(1) slope of productivity
    #b0_plus_omega_hat is the predicted value.
    coeffs, b0_plus_omega_hat = regress(yvar, Xdesign)[:2]
    #Get residual
    xi = b0_plus_omega -  b0_plus_omega_hat
    return xi, coeffs, b0_plus_omega, b0_plus_omega_prev

def moment_ex_restrictions_ACF(k, lprev):
    #Moment conditions include exogeneity restrictions for 1, k_{it},
    ↪  l_{it-1}, and Phi.
    #Put them all in one matrix for easy access, called Vexc (short for
    ↪  vectors for exogeneity restrictions)
    #Replace all nans with zeros -- this is ok, because we're just taking
    ↪  a dot product over each row of this matrix, and want to remove
    ↪  the nans
    Vex = np.vstack([
        k,
        lprev])
    return Vex

def gmm_obj_ACF(theta, y, k, l, kprev, lprev, Phi, Phiprev, Vex, W):
    #Arguments
    #Get the vector h(theta, y, k, l)
    xi = moment_error_ACF(theta, y, k, l, kprev, lprev, Phi, Phiprev)[0]
    #Calculate the "error" -- exogenous terms (dotproduct) h(theta, y, k,
    ↪  l)
    err = (Vex@xi)/len(xi)
    #Calculate the weighted sum of the error using the weight matrix, W
    obj = err.T@W@err
    return obj
```

```
#==== Used in GNR only
 ↪  ========================================================#
#Function for finding the nonlinear least squares objective function
#This is used to fit the regression of log shares on the log of the
 ↪  polynomial approximation of D_{jt}
def nlls_share_obj(gamma, X_poly, s):
    #gamma is the vector of coefficients
    #X_poly is the design matrix containing all of the polynomial
     ↪  coefficients
    Dhat = X_poly@gamma
    #Evaluate the objective -- the sum of squared residuals
    obj = np.sum((s.flatten() - np.log(Dhat))**2) #/(X_poly.shape[0])
    return obj

#Used to integrate the polynomial, in order to get CurlyD.
#by default,integrate with respect to to xvars[2] = m
def poly_integral_design_matrix(xvars, degree, w_r_t = 2):
    #Get number of observations (n) and number of independent variables
     ↪  (k)
    if xvars.ndim == 1:
        xvars = xvars.reshape(1, -1)
    # Get the number of samples (n) and number of features (m) from X
    n_samples, n_features = xvars.shape
    # Start with a column of ones for the intercept term
    X_poly_integral0 = np.ones((n_samples, 1))
    #Create iterator used to construct polynomial terms
    polynomial_terms = poly_terms(n_features, degree)
    # Generate polynomial terms and interaction terms up to 4th degree
    for terms in  polynomial_terms:  # For degrees 1 to 4
            integration_divisor = terms.count(w_r_t) + 1 #count the number
             ↪  of xvars[2] (i.e. m) appearing in the term and add 1.
                                                          #Divide the
                                                           ↪  column by
                                                           ↪  that term to
                                                           ↪  campute the
                                                           ↪  "integration
                                                           ↪  scalar"
            xcolumn = np.prod(xvars[:, terms], axis=1).reshape(-1, 1) /
             ↪  integration_divisor
            X_poly_integral0 = np.hstack((X_poly_integral0, xcolumn))
    #Elementwise-multiply all columns in the resulting matrix by m
    X_poly_integral = X_poly_integral0 *
     ↪  xvars[:,w_r_t].reshape(xvars.shape[0],1)
    return X_poly_integral
```

```
#Moment error function
def gmm_stage2_error_GNR(alpha, X_poly_omega, Xprev_poly_omega, CurlyY,
↪  CurlyYprev):
    #Given alpha, the previous omega is Curly Y + the integration
    ↪  constant curlyC, which is a polynomial fit on lagged k and l
    #Note that there's NO INTERCEPT in this polynomial fit
    #Even if we included an intercept, it would not be identified.
    omegaprev = CurlyYprev + Xprev_poly_omega@alpha
    #Then, calculate current omega = curlyY
    omega = CurlyY + X_poly_omega@alpha
    #Regress omega on omegaprev
    degree_omega = 2
    #xvars
    Xo = poly_design_matrix(omegaprev.reshape(-1, 1), degree_omega)
    #Fit the regression omegaprev ~ polynomial(omega)
    #delta is the coefficients, eta are the residuals
    delta, _, eta = regress(omega, Xo)
    #calculate the moment errors
    #The moments are simply the polynomial terms in poly(alpha) used to
    ↪  approximate the constant of integration.
    moment_error = (X_poly_omega.T @ eta)/len(eta)
    return moment_error, delta, eta

def gmm_obj_fcn_GNR(alpha, X_poly_omega, Xprev_poly_omega, CurlyY,
↪  CurlyYprev, W):
    #Get the moment errors
    moment_error = gmm_stage2_error_GNR(alpha, X_poly_omega,
    ↪  Xprev_poly_omega, CurlyY, CurlyYprev)[0]
    #caluclate GMM error objective function
    obj = moment_error.T@W@moment_error
    return obj

#Define autogradient
autogradient_GNR = grad(gmm_obj_fcn_GNR)



#==== functions for bootstrapping
↪  ==========================================#


def bootstrap(func, param0, df, n_samples = 1000, columns=5):
    # Store results
```

```
    coefficients = np.zeros((n_samples, columns))  # 3 coefficients
    convergence = np.zeros((n_samples, 1))

    list_df_boot = bootstrap_sample_panel(df, n_samples)


    # Perform bootstrap sampling
    for i in range(n_samples):
        # Sample with replacement
        # Get coefficients from the provided function
        printflag = 0
        param0_temp = param0
        for tries in range(50):
            coefs, conv = func(list_df_boot[i], param0_temp)[:2]
            if conv < 1e-10:
                if printflag == 1:
                    print("convergence succeeded on sample", i)
                break
            else:  #Run again if no convergence (occurs rarely)
                print("convergence failed on sample", i, "; trying new
                ↪  initial guess.")
                printflag = 1
                perturb = np.random.uniform(0.3, 3)
                if isinstance(param0_temp, tuple):
                    a, b = param0
                    param0_temp = (a*perturb, b)
                else:
                    param0_temp = param0*perturb


        coefficients[i,:] =coefs
        convergence[i] =conv

    # Return the bootstrap results
    return coefficients, convergence

# Bootstrap sampling function
def bootstrap_sample_panel(data, n_samples, id_col='firm_id'):
    # Create an empty list to store bootstrap samples
    bootstrap_samples = []
    original_firms = data[id_col].unique()
    num_firms = len(original_firms)

    for _ in range(n_samples):
```

```
        # Sample 'firm_id's with replacement
        sampled_firms = np.random.choice(original_firms, size=num_firms,
        ↪  replace=True)

        # Initialize a list to store the sampled data with unique firm
        ↪  IDs
        sample_data_list = []

        for i, firm in enumerate(sampled_firms):
            # Extract the firm data block
            firm_data = data[data[id_col] == firm].copy()

            # Assign a new unique firm ID by offsetting with the current
            ↪  sample index
            firm_data[id_col] = f"{firm}_{i}"  # e.g., '1_0', '2_1', etc.

            # Append to the list for this sample
            sample_data_list.append(firm_data)

        # Concatenate all firm blocks in this sample
        sample_data = pd.concat(sample_data_list).reset_index(drop=True)

        # Append to list of samples
        bootstrap_samples.append(sample_data)

    return bootstrap_samples

def summarize_array(point_estimate, arr, row_names):
    # Ensure arr is a NumPy array
    arr = np.asarray(arr)

    # Validate row_names length
    if len(row_names) != arr.shape[1]:
        raise ValueError("The length of row_names must match the number of
        ↪  columns in the array.")

    # Compute summary statistics
    mean = np.mean(arr, axis=0)
    p2_5 = np.percentile(arr, 2.5, axis=0)
    p25 = np.percentile(arr, 25, axis=0)
    median = np.median(arr, axis=0)
    p75 = np.percentile(arr, 75, axis=0)
    p97_5 = np.percentile(arr, 97.5, axis=0)
    std_error = np.std(arr, axis=0, ddof=1) / np.sqrt(arr.shape[0])
```

```
    # Create a DataFrame to organize results
    summary_df = pd.DataFrame({
        'Point Estimate': point_estimate,
        'Bootstrap Mean': mean,
        '2.5th Percentile': p2_5,
        '25th Percentile': p25,
        'Median': median,
        '75th Percentile': p75,
        '97.5th Percentile': p97_5,
        'Standard Error': std_error
    }, index=row_names)

    return summary_df
```