In [1]:

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
import pyblp as blp
import torch
from torch.autograd import Variable
import torch.optim as optim
from linearmodels.iv import IV2SLS
from HomogenousDemandEstimation import HomDemEst
from GaussHermiteQuadrature import GaussHermiteQuadrature

blp.options.digits = 2
blp.options.verbose = False
nax = np.newaxis
```

The file `ps1_ex4.csv` contains aggregate data on $T = 600$ markets in which $J = 6$ products compete between each other together with an outside good $j = 0$. The utility of consumer i is given by:

$$\begin{aligned} u_{ijt} &= \widetilde{\mathbf{x}}_{jt}'\boldsymbol{\beta} + \xi_{jt} + \widetilde{\mathbf{x}}_{jt}'\boldsymbol{\Gamma}\boldsymbol{v}_i + \epsilon_{ijt} \quad j = 1, \ldots, 6 \\ u_{i0t} &= \epsilon_{i0t} \end{aligned}$$

where x_{jt} is a vector of observed product characteristics including the price, ξ_{jt} is an unobserved product characteristic, v_i is a vector of unobserved taste shocks for the product characteristics and ϵ_{ijt} is i.i.d T1EV $(0, 1)$. Our goal is to to estimate demand parameters $(\boldsymbol{\beta}, \boldsymbol{\Gamma})$ using the BLP algorithm. As you can see from the data there are only two characteristics $\widetilde{\mathbf{x}}_{jt} = \begin{pmatrix} p_{jt} & x_{jt} \end{pmatrix}$, namely prices and an observed measure of product quality. Moreover, there are several valid instruments $\mathbf{z}_{jt}$ that you will use to construct moments to estimate $(\boldsymbol{\beta}, \boldsymbol{\Gamma})$. Finally, you can assume that Γ is lower triangular e.g.,

$$\boldsymbol{\Gamma} = \begin{pmatrix} \gamma_{11} & 0 \\ \gamma_{21} & \gamma_{22} \end{pmatrix}$$

such that $\boldsymbol{\Gamma}\boldsymbol{\Gamma}' = \boldsymbol{\Omega}$ is a positive definite matrix and that v_i is a 2 dimensional vector of i.i.d random taste shocks distributed $\mathcal{N}(\mathbf{0}, \mathbf{I}_2)$.

In [2]:

```
# Load the dataset.
data_ex4 = pd.read_csv('ps1_ex4.csv')
data_ex4['const'] = 1.0      # Add a constant term

num_prod = data_ex4.choice.nunique()    # Number of products to choose from.
num_T = data_ex4.market.nunique()

# Create outside option shares and merge into dataset.
share_total = data_ex4.groupby(['market'])['shares'].sum().reset_index()
share_total.rename(columns={'shares': 's0'}, inplace=True)
share_total['s0'] = 1 - share_total['s0']
data_ex4 = pd.merge(data_ex4, share_total, on='market')

# Create natural log of share ratios
data_ex4['sr'] = np.log(data_ex4['shares']/data_ex4['s0'])
```

```python
# Create constant term
data_ex4['const'] = 1
```

The market shares can be expressed as a function of individual characteristics as shown below.

$$
\begin{aligned}
s_j &\simeq \mathbb{E}[\Pr(i \text{ Chooses } j)] \\
&= \int_{\mathbf{v}_i} \Pr(i \text{ Chooses } j)\, \mathrm{d}\, F(\mathbf{v}_i) \\
&= \int_{\mathbf{v}_i} \frac{\exp\left(\widetilde{\mathbf{x}}'_{jt}\boldsymbol{\beta} + \xi_{jt} + \widetilde{\mathbf{x}}'_{jt}\boldsymbol{\Gamma}\boldsymbol{v}_i\right)}{1 + \sum_{k\in\mathcal{J}_t} \exp\left(\widetilde{\mathbf{x}}'_{kt}\boldsymbol{\beta} + \xi_{kt} + \widetilde{\mathbf{x}}'_{kt}\boldsymbol{\Gamma}\boldsymbol{v}_i\right)} \mathrm{d}\, F(\mathbf{v}_i)
\end{aligned}
$$

However, due to the heterogeneity in individual preferences, we do not have a neat solution to back out the preference parameters from using logarithms of share-ratios.

In [3]:

```python
# Obtain initial guess for β using the homogenous model.

est = HomDemEst(data_dict={
    'Data': data_ex4,
    'Choice Column': 'choice',
    'Market Column': 'market',
    'Log Share Ratio Column': 'sr',
    'Endogenous Columns': ['p'],
    'Exogenous Columns': ['x'],
    'Instrument Columns': ['z1', 'z2', 'z3', 'z4', 'z5', 'z6'],
    'Add Constant': True
})

beta_guess = torch.tensor(np.array(est.one_step_gmm().detach()), dtype=torch.dou
beta_guess
```

Out[3]:

```
tensor([-1.9158,  0.7115, -0.3054], dtype=torch.float64)
```

In [4]:

```python
# Set parameters for the optimization procedure.
gamma = Variable(3 * torch.rand((2,2), dtype=torch.double), requires_grad=True)
beta = Variable(beta_guess, requires_grad=False)

print(gamma)
```

```
tensor([[2.4525, 2.1547],
        [0.2082, 0.1282]], dtype=torch.float64, requires_grad=True)
```

In [5]:

```python
ghq = GaussHermiteQuadrature(2, 9)
ghq_node_mat = ghq.X.T
```

In [6]:

```python
# Save data as Pytorch tensors.
shares = torch.tensor(np.array(data_ex4['shares']),
                      dtype=torch.double)
covars = torch.tensor(np.array(data_ex4[['const', 'x', 'p']]),
                      dtype=torch.double)
num_covar = covars.size()[1]
instruments = torch.tensor(np.array(data_ex4[['const', 'x', 'z1', 'z2',
                                              'z3', 'z4', 'z5', 'z6']]),
```

```
                                             dtype=torch.double)
x_mat = covars.reshape((num_T, num_prod, num_covar))
s_mat = shares.reshape((num_T, num_prod))

x_random_mat = x_mat[:, :, 1:-1]
```

Part A - Nested Fixed Point Approach

We solve for the model parameters β and $\color{red}{\text{\Gamm}}$ using the NFXP algorithm outlined in BLP (1995) and Nevo (2001).

In [7]:

```
def mean_utility(b, xi):

    return covars @ b[:, None] + xi

def market_share_val(delta, g):

    # Evaluate the expression for every market, product and
    # Gauss-Hermite node.
    # Returns a matrix of size (num_T, num_prod, GHQ_size).

    numer = torch.exp(delta[:, :, None] + torch.einsum('tjk,kl,lm -> tjm', x_ran
    denom = 1 + numer.sum(axis=1)

    # Compute the share matrix for every value of unobserved individual characte
    share_mat = numer.div(denom[:, None])

    # Take the expected value of the above matrix using a GH integral
    # approximation.
    exp_share = torch.einsum('m, tjm -> tj', ghq.W, share_mat)

    return exp_share

def blp_contraction(b, g, res):

    # Initial guess for mean utility
    delta = mean_utility(b, res).reshape((num_T, num_prod))

    error, tol = 1, 1e-12

    while error > tol:

        exp_delta_new = torch.exp(delta) * s_mat.div(market_share_val(delta, g))

        delta_new = torch.log(exp_delta_new)

        error = torch.linalg.norm(delta_new - delta)
        delta = delta_new

        if error % 20 == 0:
            print('Inner Loop Error = {}'.format(error))

    return delta
```

In [8]:

```
def blp_gmm_loss(b, g):
```

```
    xi = torch.zeros((num_prod * num_T, 1), dtype=torch.double, requires_grad=Fa

    # Obtaining the BLP contraction solution for the mean utilities.
    delta = blp_contraction(b, g, xi).reshape((num_T * num_prod, 1))

    # Run 2SLS of mean utilities on covariates (including prices).
    blp_2sls = IV2SLS(dependent=np.array(delta.detach()),
                      exog=data_ex4[['const', 'x']],
                      endog=data_ex4['p'],
                      instruments=data_ex4[['z1', 'z2', 'z3', 'z4', 'z5', 'z6']]

    # Use 2SLS coefficients.
    b_2sls = torch.tensor(np.array(blp_2sls.params))

    # Derive residuals using 2SLS coefficients.
    res = delta - covars @ b_2sls[:, None]

    # Derive moment conditions required for BLP.
    moment_eqns = res * instruments
    moments = moment_eqns.mean(axis=0)

    loss_gmm = moments[None, :] @ weight_matrix @ moments[:, None]

    print('beta = {}, gamma = {}, loss = {}'.format(np.array(b_2sls.clone().deta
                                                    np.array(g.clone().detach())
                                                    loss_gmm.clone().detach())
          )

    return loss_gmm, moment_eqns, b_2sls
```

In [ ]:

```
opt_gmm = optim.Adam([gamma], lr=0.01)
weight_matrix = Variable(torch.eye(instruments.shape[1], dtype=torch.double), re

# Optimizing over the GMM loss function
for epoch in range(500):

    opt_gmm.zero_grad()   # Reset gradient inside the optimizer

    # Compute the objective at the current parameter values.
    loss, moment_x, new_beta = blp_gmm_loss(beta, gamma)
    loss.backward()    # Gradient computed.

    opt_gmm.step()     # Update parameter values using gradient descent.

    with torch.no_grad():
        gamma[0,1] = gamma[0,1].clamp(0.00, 0.00)
        beta[1] = beta[1].clamp(0.00, np.inf)
        # gamma[0,0] = gamma[0,0].clamp(0.00, np.inf)
        # gamma[1,1] = gamma[1,1].clamp(0.00, np.inf)

    weight_matrix = torch.inverse(1/(num_T * num_prod) * (moment_x.T @ moment_x)
    beta = new_beta.detach()
    # beta = beta2.detach().clone()

    # if epoch % 10 == 0:
    #
    #     loss_val = np.squeeze(loss.detach())
    #     print('Iteration [{}]: Loss = {:2.4e}'.format(epoch, loss_val))
```

In [10]:
```
gamma
```

Out[10]:
```
tensor([[ 2.0276,  0.0000],
        [-0.2168,  0.1702]], dtype=torch.float64, requires_grad=True)
```

In [11]:
```
beta
```

Out[11]:
```
tensor([-4.5439,  0.5183, -0.3859], dtype=torch.float64)
```

In [12]:
```
ω = np.array((gamma @ gamma.T).detach())
ω
```

Out[12]:
```
array([[ 4.1112315 , -0.43952497],
       [-0.43952497,  0.07597359]])
```

In [13]:
```
ω[1,0] / np.sqrt(ω[0,0] * ω[1, 1])
```

Out[13]:
```
-0.7864411888049343
```

In [14]:
```
np.sqrt([ω[0,0] , ω[1,1]])
```

Out[14]:
```
array([2.0276172 , 0.27563308])
```

We find that

$$\widehat{\boldsymbol{\beta}} = \begin{pmatrix} -4.5439 \\ 0.5183 \\ -0.3859 \end{pmatrix}, \qquad \widehat{\boldsymbol{\Gamma}} = \begin{pmatrix} 2.0276 & 0.0000 \\ -0.2168 & 0.1702 \end{pmatrix}$$

In [15]:
```
beta, gamma = beta.detach(), gamma.detach()
```

Part B

To compute market-specific elasticities, we need to first predict individual level market shares for various realizations of $\mathbf{v}_i$ and then average these across all individuals. For each realization of $\mathbf{v}_i$, the predicted market share for product j is given by

$$s_{ijt} = \frac{\exp\left(\widetilde{\mathbf{x}}_{jt}'\boldsymbol{\beta} + \xi_{jt} + \widetilde{\mathbf{x}}_{jt}'\boldsymbol{\Gamma}\boldsymbol{v}_i\right)}{1 + \sum_{k \in \mathcal{J}_t} \exp\left(\widetilde{\mathbf{x}}_{kt}'\boldsymbol{\beta} + \xi_{kt} + \widetilde{\mathbf{x}}_{kt}'\boldsymbol{\Gamma}\boldsymbol{v}_i\right)}$$

The individual coefficients are given by

$$\widehat{\boldsymbol{\beta}}_i = \widehat{\boldsymbol{\beta}} + \boldsymbol{\Gamma}\boldsymbol{v_i}$$

We can put these together to compute the own-price and cross-price elasticities for each market using the following equations:

$$\varepsilon_{jk,t} = \frac{\partial \pi_{j,t}}{\partial p_{k,t}} \frac{p_{k,t}}{\pi_{j,t}} = \begin{cases} -\frac{p_{j,t}}{\pi_{j,t}} \int_{\mathbf{v}_i} \alpha_i \pi_{i,j,t} \left(1 - \pi_{i,j,t}\right) \mathrm{d}F\left(\mathbf{v}_i\right) & \text{if } j = k \\ \frac{p_{k,t}}{\pi_{j,t}} \int_{\mathbf{v}_i} \alpha_i \pi_{i,j,t} \pi_{i,k,t} \, \mathrm{d}F\left(\mathbf{v}_i\right) & \text{otherwise.} \end{cases}$$

We again rely on Gauss-Hermite quadratures to evaluate the integrals.

In [16]:
```
def generate_ind_params(b, g):

    # Predicts individual market shares and individual price coefficients for ea
    # Returns a matrix of size (num_T, num_prod, GHQ_size) and (2, GHQ_size)

    numer = torch.exp(torch.einsum('tjk,kl->tjl', x_mat, b[:, None]) + torch.ein
    denom = 1 + numer.sum(axis=1)

    # Compute the share matrix for every value of unobserved individual characte
    share_mat = numer.div(denom[:, None])

    beta_mat = b[1:-1][:, None] + torch.einsum('kl,lm -> km', g, ghq_node_mat)

    return share_mat, beta_mat
```

In [17]:
```
data_wide = pd.pivot(data_ex4, values='p', index='market', columns='choice')

data_wide.rename(columns={1: "price_1", 2: "price_2",
                          3: "price_3", 4: "price_4",
                          5: "price_5", 6: "price_6"}, inplace=True)

data_ex4 = pd.merge(data_ex4, data_wide, on='market')
```

In [18]:
```
beta[-1] = -beta[-1]

share_mat, beta_mat = generate_ind_params(beta, gamma)

own_price_integral = torch.einsum('tjm, tjm, m, m -> tj', share_mat, 1 - share_m

cross_price_integral = torch.einsum('tjm, tkm, m, m -> tjk', share_mat, share_ma

data_ex4['own'] = - own_price_integral.reshape((num_T * num_prod)) * data_ex4['p

data_ex4['cross_1'] = cross_price_integral[:, :, 0].reshape((num_T * num_prod))
data_ex4['cross_2'] = cross_price_integral[:, :, 1].reshape((num_T * num_prod))
data_ex4['cross_3'] = cross_price_integral[:, :, 2].reshape((num_T * num_prod))
data_ex4['cross_4'] = cross_price_integral[:, :, 3].reshape((num_T * num_prod))
data_ex4['cross_5'] = cross_price_integral[:, :, 4].reshape((num_T * num_prod))
data_ex4['cross_6'] = cross_price_integral[:, :, 5].reshape((num_T * num_prod))
```

In [19]:
```
average_elasticity = data_ex4.groupby('choice')[['own', 'cross_1', 'cross_2', 'c
e_mat = np.array(average_elasticity[['cross_1', 'cross_2', 'cross_3', 'cross_4',
np.fill_diagonal(e_mat, np.array(average_elasticity['own']))
```

j/k	Product 1	Product 2	Product 3	Product 4	Product 5	Product 6
Product 1	-0.00058851	1.01873e-05	0.0383455	0.023619	0.111892	0.103909

j/k	Product 1	Product 2	Product 3	Product 4	Product 5	Product 6
Product 2	2.5978e-05	-0.000799222	0.0416366	0.0259857	0.157375	0.210676
Product 3	0.000215098	0.000272143	-21.8507	4.99784	4.66744	4.99098
Product 4	0.000215119	0.000143814	0.609505	-18.32	4.39132	4.94391
Product 5	6.38342e-05	6.19015e-05	0.273356	0.288736	-5.41066	2.6977
Product 6	4.42949e-05	7.32514e-05	0.215868	0.197864	2.42652	-5.01205

We see that own price and cross price elasticities are not driven solely by functional form, but by the heterogeneity in the price sensitivity across consumers who purchase the various products. This creates the difference between the results here and in Exercise 3. The absurdly low elasticities associated with products 1 and 2 could be driven by the extremely low prices for these products across all markets as seen in the table below for Part 3.

Part 3

The difference in prices and market shares could be attributed to certain products having much lower quality on average (especially products 3 and 4) compared to products 5 and 6. The impact of quality on customer preferences might be heterogenous, but the coefficient related to quality is strictly positive with low variance, which implies that customers will tend to shift away from these products in unison.

In [21]:

```
data_ex4.groupby('choice')[['p', 'x', 'shares']].mean()
```

Out[21]:

choice	p	x	shares
1	0.002439	-0.019330	0.098810
2	0.002286	-0.026036	0.089131
3	2.019113	-0.081252	0.043009
4	1.751616	-0.180135	0.039323
5	3.576978	1.692610	0.151714
6	4.442894	2.002366	0.193238

In [21]:

```
```