Assignment 4: Demand Function Estimation II

Kohei Kawaguchi

2019/3/25

Simulate data

Be carefull that some parameters are changed from assignment 3. We simulate data from a discrete choice model that is the same with in assignment 3 except for the existence of unobserved product-specific fixed effects. There are T markets and each market has N consumers. There are J products and the indirect utility of consumer i in market t for product j is:

$$u_{itj} = \beta_{it}' x_j + \alpha_{it} p_{jt} + \xi_{jt} + \epsilon_{ijt},$$

where ϵ_{ijt} is an i.i.d. type-I extreme random variable. x_j is K-dimensional observed characteristics of the product. p_{jt} is the retail price of the product in the market.

ξ_{jt} is product-market specific fixed effect. p_{jt} can be correlated with ξ_{jt} but x_{jt}s are independent of ξ_{jt}. $j = 0$ is an outside option whose indirect utility is:

$$u_{it0} = \epsilon_{i0t},$$

where ϵ_{i0t} is an i.i.d. type-I extreme random variable.

β_{it} and α_{it} are different across consumers, and they are distributed as:

$$\beta_{itk} = \beta_{0k} + \sigma_k \nu_{itk},$$

$$\alpha_{it} = -\exp(\mu + \omega \upsilon_{it}) = -\exp(\mu + \frac{\omega^2}{2}) + [-\exp(\mu + \omega \upsilon_{it}) + \exp(\mu + \frac{\omega^2}{2})] \equiv \alpha_0 + \tilde{\alpha}_{it},$$

where ν_{itk} for $k = 1, \cdots, K$ and υ_{it} are i.i.d. standard normal random variables. α_0 is the mean of α_i and $\tilde{\alpha}_i$ is the deviation from the mean.

Given a choice set in the market, $\mathcal{J}_t \cup \{0\}$, a consumer chooses the alternative that maximizes her utility:

$$q_{ijt} = 1\{u_{ijt} = \max_{k \in \mathcal{J}_t \cup \{0\}} u_{ikt}\}.$$

The choice probability of product j for consumer i in market t is:

$$\sigma_{jt}(p_t, x_t, \xi_t) = \mathbb{P}\{u_{ijt} = \max_{k \in \mathcal{J}_t \cup \{0\}} u_{ikt}\}.$$

Suppose that we only observe the share data:

$$s_{jt} = \frac{1}{N} \sum_{i=1}^{N} q_{ijt},$$

along with the product-market characteristics x_{jt} and the retail prices p_{jt} for $j \in \mathcal{J}_t \cup \{0\}$ for $t = 1, \cdots, T$. We do not observe the choice data q_{ijt} nor shocks $\xi_{jt}, \nu_{it}, \upsilon_{it}, \epsilon_{ijt}$.

We draw ξ_{jt} from i.i.d. normal distribution with mean 0 and standard deviation σ_ξ.

1. Set the seed, constants, and parameters of interest as follows.

```r
# set the seed
set.seed(1)
# number of products
J <- 10
# dimension of product characteristics including the intercept
K <- 3
# number of markets
T <- 100
# number of consumers per market
N <- 500
# number of Monte Carlo
L <- 500
```

```r
# set parameters of interests
beta <- rnorm(K);
beta[1] <- 4
beta
```

```
## [1]  4.0000000  0.1836433 -0.8356286
```

```r
sigma <- abs(rnorm(K)); sigma
```

```
## [1] 1.5952808 0.3295078 0.8204684
```

```r
mu <- 0.5
omega <- 1
```

Generate the covariates as follows.

The product-market characteristics:

$$x_{j1} = 1, x_{jk} \sim N(0, \sigma_x), k = 2, \cdots, K,$$

where σ_x is referred to as `sd_x` in the code.

The product-market-specific unobserved fixed effect:

$$\xi_{jt} \sim N(0, \sigma_\xi),$$

where $\sigma_x i$ is referred to as `sd_xi` in the code.

The marginal cost of product j in market t:

$$c_{jt} \sim \text{logNormal}(0, \sigma_c),$$

where σ_c is referred to as `sd_c` in the code.

The retail price:

$$p_{jt} - c_{jt} \sim \text{logNorm}(\gamma \xi_{jt}, \sigma_p),$$

where γ is referred to as `price_xi` and σ_p as `sd_p` in the code. This price is not the equilibrium price. We will revisit this point in a subsequent assignment.

The value of the auxiliary parameters are set as follows:

```r
# set auxiliary parameters
price_xi <- 1
sd_x <- 2
sd_xi <- 0.5
sd_c <- 0.05
sd_p <- 0.05
```

2. `X` is the data frame such that a row contains the characteristics vector x_j of a product and columns are product index and observed product characteristics. The dimension of the characteristics K is specified above. Add the row of the outside option whose index is 0 and all the characteristics are zero.

```
# make product characteristics data
X <- matrix(sd_x * rnorm(J * (K - 1)), nrow = J)
X <- cbind(rep(1, J), X)
colnames(X) <- paste("x", 1:K, sep = "_")
X <- data.frame(j = 1:J, X) %>%
  tibble::as_tibble()
# add outside option
X <- rbind(
  rep(0, dim(X)[2]),
  X
)
```

```
X
```

```
## # A tibble: 11 x 4
##        j   x_1     x_2     x_3
##    <dbl> <dbl>   <dbl>   <dbl>
##  1     0     0  0        0
##  2     1     1  0.975  -0.0324
##  3     2     1  1.48    1.89
##  4     3     1  1.15    1.64
##  5     4     1 -0.611   1.19
##  6     5     1  3.02    1.84
##  7     6     1  0.780   1.56
##  8     7     1 -1.24    0.149
##  9     8     1 -4.43   -3.98
## 10     9     1  2.25    1.24
## 11    10     1 -0.0899 -0.112
```

3. `M` is the data frame such that a row contains the price ξ_{jt}, marginal cost c_{jt}, and price p_{jt}. After generating the variables, drop some products in each market. **In this assignment, we drop products in a different way from the last assignment**. In order to change the number of available products in each market, for each market, first draw J_t from a discrete uniform distribution between 1 and J. Then, drop products from each market using `dplyr::sample_frac` function with the realized number of available products. The variation in the available products is important for the identification of the distribution of consumer-level unobserved heterogeneity. Add the row of the outside option to each market whose index is 0 and all the variables take value zero.

```
# make market-product data
M <- expand.grid(j = 1:J, t = 1:T) %>%
  tibble::as_tibble() %>%
  dplyr::mutate(
    xi = sd_xi * rnorm(J*T),
    c = exp(sd_c * rnorm(J*T)),
    p = exp(price_xi * xi + sd_p * rnorm(J*T)) + c
  )
M <- M %>%
  dplyr::group_by(t) %>%
  dplyr::sample_frac(size = purrr::rdunif(1, J)/J) %>%
  dplyr::ungroup()
# add outside option
outside <- data.frame(j = 0, t = 1:T, xi = 0, c = 0, p = 0)
```

```
M <- rbind(
  M,
  outside
) %>%
  dplyr::arrange(t, j)
```

```
M
```

```
## # A tibble: 746 x 5
##        j     t      xi     c     p
##    <dbl> <int>   <dbl> <dbl> <dbl>
##  1     0     1  0       0     0
##  2     6     1 -0.0514 0.980  1.91
##  3     0     2  0       0     0
##  4     1     2 -0.197  0.988  1.90
##  5    10     2 -0.354  1.05   1.82
##  6     0     3  0       0     0
##  7     1     3  0.182  1.04   2.22
##  8     2     3  0.384  1.01   2.61
##  9     3     3 -0.0562 1.02   2.01
## 10     4     3  0.441  1.01   2.62
## # ... with 736 more rows
```

4. Generate the consumer-level heterogeneity. `V` is the data frame such that a row contains the vector of shocks to consumer-level heterogeneity, (ν_i', υ_i). They are all i.i.d. standard normal random variables.

```
# make consumer-market data
V <- matrix(rnorm(N * T * (K + 1)), nrow = N * T)
colnames(V) <- c(paste("v_x", 1:K, sep = "_"), "v_p")
V <- data.frame(
  expand.grid(i = 1:N, t = 1:T),
  V
) %>%
  tibble::as_tibble()
```

```
V
```

```
## # A tibble: 50,000 x 6
##        i     t   v_x_1  v_x_2   v_x_3    v_p
##    <int> <int>   <dbl>  <dbl>   <dbl>  <dbl>
##  1     1     1  1.16   -1.40   0.0786 -1.15
##  2     2     1 -1.05   -0.149  1.08    0.623
##  3     3     1 -0.426  -4.21   0.625  -1.14
##  4     4     1 -0.235   0.463  0.470   0.241
##  5     5     1  1.19   -0.342  0.169   0.160
##  6     6     1  0.541   0.525  0.305   1.72
##  7     7     1 -0.0893 -0.434  2.18   -0.432
##  8     8     1 -0.712   0.747 -0.306  -0.527
##  9     9     1  0.504   1.11   1.70   -1.75
## 10    10     1 -0.107   1.83  -0.841   0.693
## # ... with 49,990 more rows
```

5. Join `X`, `M`, `V` using `dplyr::left_join` and name it `df`. `df` is the data frame such that a row contains variables for a consumer about a product that is available in a market.

```
# make choice data
df <- expand.grid(t = 1:T, i = 1:N, j = 0:J) %>%
```

```r
  tibble::as_tibble() %>%
  dplyr::left_join(V, by = c("i", "t")) %>%
  dplyr::left_join(X, by = c("j")) %>%
  dplyr::left_join(M, by = c("j", "t")) %>%
  dplyr::filter(!is.na(p)) %>%
  dplyr::arrange(t, i, j)
```

```r
df
```

```
## # A tibble: 373,000 x 13
##        t     i     j  v_x_1  v_x_2  v_x_3    v_p   x_1   x_2   x_3      xi
##    <int> <int> <dbl>  <dbl>  <dbl>  <dbl>  <dbl> <dbl> <dbl> <dbl>   <dbl>
##  1     1     1     0  1.16  -1.40  0.0786 -1.15      0 0       0      0
##  2     1     1     6  1.16  -1.40  0.0786 -1.15      1 0.780   1.56 -0.0514
##  3     1     2     0 -1.05  -0.149 1.08    0.623     0 0       0      0
##  4     1     2     6 -1.05  -0.149 1.08    0.623     1 0.780   1.56 -0.0514
##  5     1     3     0 -0.426 -4.21  0.625  -1.14      0 0       0      0
##  6     1     3     6 -0.426 -4.21  0.625  -1.14      1 0.780   1.56 -0.0514
##  7     1     4     0 -0.235  0.463 0.470   0.241     0 0       0      0
##  8     1     4     6 -0.235  0.463 0.470   0.241     1 0.780   1.56 -0.0514
##  9     1     5     0  1.19  -0.342 0.169   0.160     0 0       0      0
## 10     1     5     6  1.19  -0.342 0.169   0.160     1 0.780   1.56 -0.0514
## # ... with 372,990 more rows, and 2 more variables: c <dbl>, p <dbl>
```

6. Draw a vector of preference shocks `e` whose length is the same as the number of rows of `df`.

```r
# draw idiosyncratic shocks
e <- evd::rgev(dim(df)[1])
```

```r
head(e)
```

```
## [1]  0.2262454  1.3417639 -0.1693913  0.8906905  0.5558130  1.9909058
```

7. Write a function `compute_indirect_utility(df, beta, sigma, mu, omega)` that returns a vector whose element is the mean indirect utility of a product for a consumer in a market. The output should have the same length with e. (This function is the same with assignment 3. You can use the function.)

```r
# compute indirect utility
u <-
  compute_indirect_utility(
    df, beta, sigma,
           mu, omega)
head(u)
```

```
##               u
## [1,]  0.0000000
## [2,]  3.3750542
## [3,]  0.0000000
## [4,] -3.3983588
## [5,]  0.0000000
## [6,]  0.8235142
```

In the previous assingment, we computed predicted share by simulating choice and taking their average. Instead, we compute the actual share by:

$$s_{jt} = \frac{1}{N} \sum_{i=1}^{N} \frac{\exp[\beta_{it}' x_j + \alpha_{it} p_{jt} + \xi_{jt}]}{1 + \sum_{k \in \mathcal{J}_t} \exp[\beta_{it}' x_k + \alpha_{it} p_{kt} + \xi_{jt}]}$$

and the predicted share by:

$$\widehat{\sigma}_j(x, p_t, \xi_t) = \frac{1}{L} \sum_{l=1}^{L} \frac{\exp[\beta_t^{(l)\prime} x_j + \alpha_t^{(l)} p_{jt} + \xi_{jt}]}{1 + \sum_{k \in \mathcal{J}_t} \exp[\beta_t^{(l)\prime} x_k + \alpha_t^{(l)} p_{kt} + \xi_{jt}]}.$$

8. To do so, write a function `compute_choice_smooth(X, M, V, beta, sigma, mu, omega)` in which the choice of each consumer is not:

$$q_{ijt} = 1\{u_{ijt} = \max_{k \in \mathcal{J}_t \cup \{0\}} u_{ikt}\},$$

but

$$\tilde{q}_{ijt} = \frac{\exp(u_{ijt})}{1 + \sum_{k \in \mathcal{J}_t} \exp(u_{ikt})}.$$

```r
# compute choice
compute_choice_smooth <-
  function(X, M, V, beta, sigma,
           mu, omega) {
    # constants
    T <- max(M$t)
    N <- max(V$i)
    J <- max(X$j)
    # make choice data
    df <- expand.grid(t = 1:T, i = 1:N, j = 0:J) %>%
      tibble::as_tibble() %>%
      dplyr::left_join(V, by = c("i", "t")) %>%
      dplyr::left_join(X, by = c("j")) %>%
      dplyr::left_join(M, by = c("j", "t")) %>%
      dplyr::filter(!is.na(p)) %>%
      dplyr::arrange(t, i, j)
    # compute indirect utility
    u <- compute_indirect_utility(df, beta, sigma,
                                  mu, omega)
    # add u
    df_choice <- data.frame(df, u) %>%
      tibble::as_tibble()
    # make choice
    df_choice <- df_choice %>%
      dplyr::group_by(t, i) %>%
      dplyr::mutate(q = exp(u)/sum(exp(u))) %>%
      dplyr::ungroup()
    # return
    return(df_choice)
  }
df_choice_smooth <-
  compute_choice_smooth(X, M, V, beta, sigma, mu, omega)
summary(df_choice_smooth)
```

```
##        t                i                j               v_x_1
##  Min.   :  1.00   Min.   :  1.0   Min.   : 0.000   Min.   :-4.302781
##  1st Qu.: 26.00   1st Qu.:125.8   1st Qu.: 2.000   1st Qu.:-0.685447
##  Median : 51.50   Median :250.5   Median : 5.000   Median :-0.000461
##  Mean   : 51.26   Mean   :250.5   Mean   : 4.807   Mean   :-0.005284
##  3rd Qu.: 77.00   3rd Qu.:375.2   3rd Qu.: 8.000   3rd Qu.: 0.665219
##  Max.   :100.00   Max.   :500.0   Max.   :10.000   Max.   : 3.809895
```

```
##      v_x_2               v_x_3               v_p
##  Min.   :-4.542122   Min.   :-3.957618   Min.   :-4.218131
##  1st Qu.:-0.678377   1st Qu.:-0.676638   1st Qu.:-0.672011
##  Median : 0.001435   Median : 0.006281   Median : 0.002166
##  Mean   :-0.001076   Mean   : 0.003433   Mean   :-0.001402
##  3rd Qu.: 0.671827   3rd Qu.: 0.679273   3rd Qu.: 0.674681
##  Max.   : 4.313621   Max.   : 4.244194   Max.   : 4.074300
##       x_1             x_2                x_3                  xi
##  Min.   :0.000   Min.   :-4.4294   Min.   :-3.97870   Min.   :-1.444460
##  1st Qu.:1.000   1st Qu.:-0.6108   1st Qu.:-0.03238   1st Qu.:-0.286633
##  Median :1.000   Median : 0.7797   Median : 1.18780   Median : 0.000000
##  Mean   :0.866   Mean   : 0.2713   Mean   : 0.44050   Mean   : 0.009578
##  3rd Qu.:1.000   3rd Qu.: 1.4766   3rd Qu.: 1.64244   3rd Qu.: 0.317352
##  Max.   :1.000   Max.   : 3.0236   Max.   : 1.88767   Max.   : 1.905138
##        c                p                u                   q
##  Min.   :0.0000   Min.   :0.000   Min.   :-284.515   Min.   :0.0000000
##  1st Qu.:0.9425   1st Qu.:1.562   1st Qu.:  -3.157   1st Qu.:0.0009063
##  Median :0.9886   Median :1.902   Median :   0.000   Median :0.0225375
##  Mean   :0.8670   Mean   :1.871   Mean   :  -1.927   Mean   :0.1340483
##  3rd Qu.:1.0322   3rd Qu.:2.394   3rd Qu.:   1.623   3rd Qu.:0.1203147
##  Max.   :1.1996   Max.   :8.211   Max.   :  19.907   Max.   :1.0000000
```

9. Next, write a function `compute_share_smooth(X, M, V, beta, sigma, mu, omega)` that calls `compute_choice_smooth` and then returns the share based on above $\tilde{q}_{ijt}$. If we use these functions with the Monte Carlo shocks, it gives us the predicted share of the products.

```
# compute share
compute_share_smooth <-
  function(X, M, V, beta, sigma,
           mu, omega) {
    # constants
    T <- max(M$t)
    N <- max(V$i)
    J <- max(X$j)
    # compute choice
    df_choice <-
      compute_choice_smooth(X, M, V, beta, sigma,
                    mu, omega)
    # make share data
    df_share_smooth <- df_choice %>%
      dplyr::select(-dplyr::starts_with("v_"), -u, -i) %>%
      dplyr::group_by(t, j) %>%
      dplyr::mutate(q = sum(q)) %>%
      dplyr::ungroup() %>%
      dplyr::distinct(t, j, .keep_all = TRUE) %>%
      dplyr::group_by(t) %>%
      dplyr::mutate(s = q/sum(q)) %>%
      dplyr::ungroup()
    # log share difference
    df_share_smooth <- df_share_smooth %>%
      dplyr::group_by(t) %>%
      dplyr::mutate(y = log(s/sum(s * (j == 0)))) %>%
      dplyr::ungroup()
    return(df_share_smooth)
  }
```

```r
df_share_smooth <- compute_share_smooth(X, M, V, beta, sigma, mu, omega)
summary(df_share_smooth)
```

```
##        t                j               x_1              x_2         
##  Min.   :  1.00   Min.   : 0.000   Min.   :0.000   Min.   :-4.4294  
##  1st Qu.: 26.00   1st Qu.: 2.000   1st Qu.:1.000   1st Qu.:-0.6108  
##  Median : 51.50   Median : 5.000   Median :1.000   Median : 0.7797  
##  Mean   : 51.26   Mean   : 4.807   Mean   :0.866   Mean   : 0.2713  
##  3rd Qu.: 77.00   3rd Qu.: 8.000   3rd Qu.:1.000   3rd Qu.: 1.4766  
##  Max.   :100.00   Max.   :10.000   Max.   :1.000   Max.   : 3.0236  
##       x_3                xi                  c                p        
##  Min.   :-3.97870   Min.   :-1.444460   Min.   :0.0000   Min.   :0.000  
##  1st Qu.:-0.03238   1st Qu.:-0.286542   1st Qu.:0.9427   1st Qu.:1.564  
##  Median : 1.18780   Median : 0.000000   Median :0.9886   Median :1.902  
##  Mean   : 0.44050   Mean   : 0.009578   Mean   :0.8670   Mean   :1.871  
##  3rd Qu.: 1.62290   3rd Qu.: 0.316516   3rd Qu.:1.0322   3rd Qu.:2.392  
##  Max.   : 1.88767   Max.   : 1.905138   Max.   :1.1996   Max.   :8.211  
##        q                  s                  y          
##  Min.   :  5.912   Min.   :0.01182   Min.   :-2.9837  
##  1st Qu.: 17.628   1st Qu.:0.03526   1st Qu.:-1.9109  
##  Median : 28.025   Median :0.05605   Median :-1.5011  
##  Mean   : 67.024   Mean   :0.13405   Mean   :-1.2219  
##  3rd Qu.:100.446   3rd Qu.:0.20089   3rd Qu.:-0.5556  
##  Max.   :337.118   Max.   :0.67424   Max.   : 1.1167  
```

Use this `df_share_smooth` as the data to estimate the parameters in the following section.

Estimate the parameters

1. First draw Monte Carlo consumer-level heterogeneity `V_mcmc` and Monte Carlo preference shocks `e_mcmc`. The number of simulations is `L`. This does not have to be the same with the actual number of consumers `N`.

```r
# mixed logit estimation
## draw mcmc V
V_mcmc <- matrix(rnorm(L*T*(K + 1)), nrow = L*T)
colnames(V_mcmc) <- c(paste("v_x", 1:K, sep = "_"), "v_p")
V_mcmc <- data.frame(
  expand.grid(i = 1:L, t = 1:T),
  V_mcmc
) %>%
  tibble::as_tibble()
```

```r
V_mcmc
```

```
## # A tibble: 50,000 x 6
##        i     t  v_x_1   v_x_2  v_x_3     v_p
##    <int> <int>  <dbl>   <dbl>  <dbl>   <dbl>
##  1     1     1  0.865 -0.142  -0.667  1.17  
##  2     2     1  0.316  1.29   -1.56  -0.691 
##  3     3     1  0.673  1.35   -0.203  0.388 
##  4     4     1  0.295  0.613   1.31   0.698 
##  5     5     1  0.214 -0.0878 -0.343 -0.0642
##  6     6     1 -1.06  -0.240   0.373 -0.631 
```

```
## 7      7      1 -0.556  1.05   -1.21  -2.22
## 8      8      1  0.376 -2.47    1.77  -0.333
## 9      9      1 -0.872  1.36    0.508 -0.834
## 10    10      1 -0.895 -1.26   -3.04   0.821
## # ... with 49,990 more rows
```

```
## draw mcmc e
df_mcmc <- expand.grid(t = 1:T, i = 1:L, j = 0:J) %>%
  tibble::as_tibble() %>%
  dplyr::left_join(V_mcmc, by = c("i", "t")) %>%
  dplyr::left_join(X, by = c("j")) %>%
  dplyr::left_join(M, by = c("j", "t")) %>%
  dplyr::filter(!is.na(p)) %>%
  dplyr::arrange(t, i, j)
# draw idiosyncratic shocks
e_mcmc <- evd::rgev(dim(df_mcmc)[1])
```

```
head(e_mcmc)
```

```
## [1] 1.4664583 0.9890441 1.2502808 0.7103677 0.7433964 1.9116964
```

2. Vectorize the parameters to a vector `theta` because `optim` requires the maximiand to be a vector.

```
# set parameters
theta <- c(beta, sigma, mu, omega)
theta
```

```
## [1]  4.0000000  0.1836433 -0.8356286  1.5952808  0.3295078  0.8204684
## [7]  0.5000000  1.0000000
```

3. Estimate the parameters assuming there is no product-specific unobserved fixed effects ξ_{jt}, i.e., using the functions in assignment 3. To do so, first modify `M` to `M_no` in which `xi` is replaced with 0 and estimate the model with `M_no`. Otherwise, your function will compute the share with the true `xi`.

```
M_no <- M %>%
  dplyr::mutate(xi = 0)
```

```
# find NLLS estimator
result_NLLS <-
  optim(par = theta, fn = NLLS_objective_A3,
        method = "Nelder-Mead",
        df_share = df_share_smooth,
        X = X,
        M = M_no,
        V_mcmc = V_mcmc,
        e_mcmc = e_mcmc)
save(result_NLLS, file = "data/A4_result_NLLS.RData")
```

```
result_NLLS <- get(load(file = "data/A4_result_NLLS.RData"))
result_NLLS
```

```
## $par
## [1]  3.1292612  0.1834495 -0.9357865  1.5678279  0.3768438  1.1698887
## [7] -0.1108147  1.9156776
##
## $value
## [1] 0.0004291768
##
```

```
## $counts
## function gradient
##      297       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
result <- data.frame(true = theta, estimates = result_NLLS$par)
result
```

```
##         true  estimates
## 1  4.0000000  3.1292612
## 2  0.1836433  0.1834495
## 3 -0.8356286 -0.9357865
## 4  1.5952808  1.5678279
## 5  0.3295078  0.3768438
## 6  0.8204684  1.1698887
## 7  0.5000000 -0.1108147
## 8  1.0000000  1.9156776
```

Next, we estimate the model allowing for the product-market-specific unobserved fixed effect ξ_{jt} using the BLP algorithm. To do so, we slightly modify the `compute_indirect_utility`, `compute_choice_smooth`, and `compute_share_smooth` functions so that they receive δ_{jt} to compute the indirect utilities, choices, and shares. Be careful that the treatment of α_i is slightly different from the lecture note, because we assumed that α_is are log-normal random variables.

4. Compute and print out δ_{jt} at the true parameters, i.e.:

$$\delta_{jt} = \beta_0' x_j + \alpha_0' p_{jt} + \xi_{jt}.$$

```
XX <- as.matrix(dplyr::select(df_share_smooth, dplyr::starts_with("x_")))
pp <- as.matrix(dplyr::select(df_share_smooth, p))
xi <- as.matrix(dplyr::select(df_share_smooth, xi))
alpha <- - exp(mu + omega^2/2)
delta <- XX %*% as.matrix(beta) + pp * alpha + xi
delta <- dplyr::select(df_share_smooth, t, j) %>%
  dplyr::mutate(delta = as.numeric(delta))
```

```
delta
```

```
## # A tibble: 746 x 3
##        t     j delta
##    <int> <dbl> <dbl>
##  1     1     0  0
##  2     1     6 -2.40
##  3     2     0  0
##  4     2     1 -1.15
##  5     2    10 -1.22
##  6     3     0  0
##  7     3     1 -1.65
##  8     3     2 -4.03
##  9     3     3 -2.69
## 10     3     4 -3.78
## # ... with 736 more rows
```

5. Write a function `compute_indirect_utility_delta(df, delta, sigma, mu, omega)` that returns a vector whose element is the mean indirect utility of a product for a consumer in a market. The output should have the same length with e. Print out the output with δ_{jt} evaluated at the true parameters. Check if the output is close to the true indirect utilities.

```
# compute indirect utility from delta
compute_indirect_utility_delta <-
  function(df, delta, sigma,
           mu, omega) {
    # extract matrices
    X <- as.matrix(dplyr::select(df, dplyr::starts_with("x_")))
    p <- as.matrix(dplyr::select(df, p))
    v_x <- as.matrix(dplyr::select(df, dplyr::starts_with("v_x")))
    v_p <- as.matrix(dplyr::select(df, v_p))
    # expand delta
    delta_ijt <- df %>%
      dplyr::left_join(delta, by = c("t", "j")) %>%
      dplyr::select(delta) %>%
      as.matrix()
    # random coefficients
    beta_i <- v_x %*% diag(sigma)
    alpha_i <- - exp(mu + omega * v_p) - (- exp(mu + omega^2/2))
    # conditional mean indirect utility
    value <- as.matrix(delta_ijt + rowSums(beta_i * X) + p * alpha_i)
    colnames(value) <- "u"
    return(value)
  }

# compute indirect utility from delta
u_delta <-
  compute_indirect_utility_delta(df, delta, sigma,
                                 mu, omega)
head(u_delta)
```

```
##               u
## [1,]  0.0000000
## [2,]  3.3750542
## [3,]  0.0000000
## [4,] -3.3983588
## [5,]  0.0000000
## [6,]  0.8235142
```

```
summary(u - u_delta)
```

```
##        u
##  Min.   :-5.684e-14
##  1st Qu.:-4.441e-16
##  Median : 0.000e+00
##  Mean   :-1.279e-17
##  3rd Qu.: 3.331e-16
##  Max.   : 5.684e-14
```

6. Write a function `compute_choice_smooth_delta(X, M, V, delta, sigma, mu, omega)` that first construct `df` from `X`, `M`, `V`, second call `compute_indirect_utility_delta` to obtain the vector of mean indirect utilities `u`, third compute the (smooth) choice vector `q` based on the vector of mean indirect utilities, and finally return the data frame to which `u` and `q` are added as columns. Print out the output

with δ_{jt} evaluated at the true parameters. Check if the output is close to the true (smooth) choice vector.

```
# compute choice from delta
compute_choice_smooth_delta <-
  function(X, M, V, delta, sigma,
           mu, omega) {
    # constants
    T <- max(M$t)
    N <- max(V$i)
    J <- max(X$j)
    # make choice data
    df <- expand.grid(t = 1:T, i = 1:N, j = 0:J) %>%
      tibble::as_tibble() %>%
      dplyr::left_join(V, by = c("i", "t")) %>%
      dplyr::left_join(X, by = c("j")) %>%
      dplyr::left_join(M, by = c("j", "t")) %>%
      dplyr::filter(!is.na(p)) %>%
      dplyr::arrange(t, i, j)
    # compute indirect utility
    u <- compute_indirect_utility_delta(df, delta, sigma,
                                        mu, omega)
    # add u
    df_choice <- data.frame(df, u) %>%
      tibble::as_tibble()
    # make choice
    df_choice <- df_choice %>%
      dplyr::group_by(t, i) %>%
      dplyr::mutate(q = exp(u)/sum(exp(u))) %>%
      dplyr::ungroup()
    # return
    return(df_choice)
  }

# compute choice
df_choice_smooth_delta <-
  compute_choice_smooth_delta(X, M, V, delta, sigma, mu, omega)
df_choice_smooth_delta
```

```
## # A tibble: 373,000 x 15
##        t     i     j  v_x_1  v_x_2  v_x_3    v_p   x_1   x_2   x_3      xi
##    <int> <int> <dbl>  <dbl>  <dbl>  <dbl>  <dbl> <dbl> <dbl> <dbl>   <dbl>
##  1     1     1     0  1.16  -1.40  0.0786 -1.15      0 0      0     0
##  2     1     1     6  1.16  -1.40  0.0786 -1.15      1 0.780  1.56 -0.0514
##  3     1     2     0 -1.05  -0.149 1.08    0.623     0 0      0     0
##  4     1     2     6 -1.05  -0.149 1.08    0.623     1 0.780  1.56 -0.0514
##  5     1     3     0 -0.426 -4.21  0.625  -1.14      0 0      0     0
##  6     1     3     6 -0.426 -4.21  0.625  -1.14      1 0.780  1.56 -0.0514
##  7     1     4     0 -0.235  0.463 0.470   0.241     0 0      0     0
##  8     1     4     6 -0.235  0.463 0.470   0.241     1 0.780  1.56 -0.0514
##  9     1     5     0  1.19  -0.342 0.169   0.160     0 0      0     0
## 10     1     5     6  1.19  -0.342 0.169   0.160     1 0.780  1.56 -0.0514
## # ... with 372,990 more rows, and 4 more variables: c <dbl>, p <dbl>,
## #   u <dbl>, q <dbl>
```

```r
summary(df_choice_smooth_delta)
```

```
##        t                 i               j              v_x_1
##  Min.   :  1.00   Min.   :  1.0   Min.   : 0.000   Min.   :-4.302781
##  1st Qu.: 26.00   1st Qu.:125.8   1st Qu.: 2.000   1st Qu.:-0.685447
##  Median : 51.50   Median :250.5   Median : 5.000   Median :-0.000461
##  Mean   : 51.26   Mean   :250.5   Mean   : 4.807   Mean   :-0.005284
##  3rd Qu.: 77.00   3rd Qu.:375.2   3rd Qu.: 8.000   3rd Qu.: 0.665219
##  Max.   :100.00   Max.   :500.0   Max.   :10.000   Max.   : 3.809895
##      v_x_2              v_x_3                v_p
##  Min.   :-4.542122   Min.   :-3.957618   Min.   :-4.218131
##  1st Qu.:-0.678377   1st Qu.:-0.676638   1st Qu.:-0.672011
##  Median : 0.001435   Median : 0.006281   Median : 0.002166
##  Mean   :-0.001076   Mean   : 0.003433   Mean   :-0.001402
##  3rd Qu.: 0.671827   3rd Qu.: 0.679273   3rd Qu.: 0.674681
##  Max.   : 4.313621   Max.   : 4.244194   Max.   : 4.074300
##       x_1             x_2                x_3                 xi
##  Min.   :0.000   Min.   :-4.4294   Min.   :-3.97870   Min.   :-1.444460
##  1st Qu.:1.000   1st Qu.:-0.6108   1st Qu.:-0.03238   1st Qu.:-0.286633
##  Median :1.000   Median : 0.7797   Median : 1.18780   Median : 0.000000
##  Mean   :0.866   Mean   : 0.2713   Mean   : 0.44050   Mean   : 0.009578
##  3rd Qu.:1.000   3rd Qu.: 1.4766   3rd Qu.: 1.64244   3rd Qu.: 0.317352
##  Max.   :1.000   Max.   : 3.0236   Max.   : 1.88767   Max.   : 1.905138
##        c                 p               u                  q
##  Min.   :0.0000   Min.   :0.000   Min.   :-284.515   Min.   :0.0000000
##  1st Qu.:0.9425   1st Qu.:1.562   1st Qu.:  -3.157   1st Qu.:0.0009063
##  Median :0.9886   Median :1.902   Median :   0.000   Median :0.0225375
##  Mean   :0.8670   Mean   :1.871   Mean   :  -1.927   Mean   :0.1340483
##  3rd Qu.:1.0322   3rd Qu.:2.394   3rd Qu.:   1.623   3rd Qu.:0.1203147
##  Max.   :1.1996   Max.   :8.211   Max.   :  19.907   Max.   :1.0000000
```

```r
summary(df_choice_smooth$q - df_choice_smooth_delta$q)
```

```
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max. 
## -1.166e-15 -6.939e-18  0.000e+00  4.540e-20  1.084e-18  1.221e-15
```

7. Write a function `compute_share_delta(X, M, V, delta, sigma, mu, omega)` that first construct `df` from `X`, `M`, `V`, second call `compute_choice_delta` to obtain a data frame with `u` and `q`, third compute the share of each product at each market `s` and the log difference in the share from the outside option, $\ln(s_{jt}/s_{0t})$, denoted by `y`, and finally return the data frame that is summarized at the product-market level, dropped consumer-level variables, and added `s` and `y`.

```r
# compute share from delta
compute_share_smooth_delta <-
  function(X, M, V, delta, sigma,
           mu, omega) {
    # constants
    T <- max(M$t)
    N <- max(V$i)
    J <- max(X$j)
    # compute choice
    df_choice <-
      compute_choice_smooth_delta(X, M, V, delta, sigma,
                                  mu, omega)
    # make share data
```

```r
    df_share_smooth <- df_choice %>%
      dplyr::select(-dplyr::starts_with("v_"), -u, -i) %>%
      dplyr::group_by(t, j) %>%
      dplyr::mutate(q = sum(q)) %>%
      dplyr::ungroup() %>%
      dplyr::distinct(t, j, .keep_all = TRUE) %>%
      dplyr::group_by(t) %>%
      dplyr::mutate(s = q/sum(q)) %>%
      dplyr::ungroup()
    # log share difference
    df_share_smooth <- df_share_smooth %>%
      dplyr::group_by(t) %>%
      dplyr::mutate(y = log(s/sum(s * (j == 0)))) %>%
      dplyr::ungroup()
    return(df_share_smooth)
  }

# compute share
df_share_smooth_delta <-
  compute_share_smooth_delta(X, M, V, delta, sigma, mu, omega)
df_share_smooth_delta
```

```
## # A tibble: 746 x 11
##        t     j   x_1     x_2     x_3      xi     c     p      q      s
##    <int> <dbl> <dbl>   <dbl>   <dbl>   <dbl> <dbl> <dbl>  <dbl>  <dbl>
##  1     1     0     0  0       0       0      0     0     307.   0.614
##  2     1     6     1  0.780   1.56   -0.0514 0.980  1.91 193.   0.386
##  3     2     0     0  0       0       0      0     0     208.   0.415
##  4     2     1     1  0.975  -0.0324 -0.197  0.988  1.90 158.   0.317
##  5     2    10     1 -0.0899 -0.112  -0.354  1.05   1.82 134.   0.268
##  6     3     0     0  0       0       0      0     0     113.   0.225
##  7     3     1     1  0.975  -0.0324  0.182  1.04   2.22  25.6  0.0513
##  8     3     2     1  1.48    1.89    0.384  1.01   2.61  14.4  0.0288
##  9     3     3     1  1.15    1.64   -0.0562 1.02   2.01  18.3  0.0366
## 10     3     4     1 -0.611   1.19    0.441  1.01   2.62   9.46 0.0189
## # ... with 736 more rows, and 1 more variable: y <dbl>
```

```r
summary(df_share_smooth_delta)
```

```
##        t                j               x_1             x_2
##  Min.   :  1.00   Min.   : 0.000   Min.   :0.000   Min.   :-4.4294
##  1st Qu.: 26.00   1st Qu.: 2.000   1st Qu.:1.000   1st Qu.:-0.6108
##  Median : 51.50   Median : 5.000   Median :1.000   Median : 0.7797
##  Mean   : 51.26   Mean   : 4.807   Mean   :0.866   Mean   : 0.2713
##  3rd Qu.: 77.00   3rd Qu.: 8.000   3rd Qu.:1.000   3rd Qu.: 1.4766
##  Max.   :100.00   Max.   :10.000   Max.   :1.000   Max.   : 3.0236
##       x_3                 xi                   c                p
##  Min.   :-3.97870   Min.   :-1.444460   Min.   :0.0000   Min.   :0.000
##  1st Qu.:-0.03238   1st Qu.:-0.286542   1st Qu.:0.9427   1st Qu.:1.564
##  Median : 1.18780   Median : 0.000000   Median :0.9886   Median :1.902
##  Mean   : 0.44050   Mean   : 0.009578   Mean   :0.8670   Mean   :1.871
##  3rd Qu.: 1.62290   3rd Qu.: 0.316516   3rd Qu.:1.0322   3rd Qu.:2.392
##  Max.   : 1.88767   Max.   : 1.905138   Max.   :1.1996   Max.   :8.211
##        q                 s                  y
```

```
##  Min.   :  5.912   Min.   :0.01182   Min.   :-2.9837
##  1st Qu.: 17.628   1st Qu.:0.03526   1st Qu.:-1.9109
##  Median : 28.025   Median :0.05605   Median :-1.5011
##  Mean   : 67.024   Mean   :0.13405   Mean   :-1.2219
##  3rd Qu.:100.446   3rd Qu.:0.20089   3rd Qu.:-0.5556
##  Max.   :337.118   Max.   :0.67424   Max.   : 1.1167
```

```r
summary(df_share_smooth$s - df_share_smooth_delta$s)
```

```
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## -1.388e-16 -1.388e-17  0.000e+00  7.046e-19  6.939e-18  2.220e-16
```

8. Write a function `solve_delta(df_share_smooth, X, M, V, delta, sigma, mu, omega)` that finds δ_{jt} that equates the actua share and the predicted share based on `compute_share_smooth_delta` by the fixed-point algorithm with an operator:

$$T(\delta_{jt}^{(r)}) = \delta_{jt}^{(r)} + \kappa \cdot \log\left(\frac{s_{jt}}{\sigma_{jt}[\delta^{(r)}]}\right),$$

where s_{jt} is the actual share of product j in market t and $\sigma_{jt}[\delta^{(r)}]$ is the predicted share of product j in market t given $\delta^{(r)}$. Multiplying κ is for the numerical stability. I set the value at $\kappa = 1$. Adjust it if the algorithm did not work. Set the stopping criterion at $\max_{jt} |\delta_{jt}^{(r+1)} - \delta_{jt}^{(r)}| < \lambda$. Set λ at 10^{-3}. Make sure that δ_{i0t} is always set at zero while the iteration.

Start the algorithm with the true δ_{jt} and check if the algorithm returns (almost) the same δ_{jt} when the actual and predicted smooth share are equated.

```r
# solve delta by the fixed-point algorithm
solve_delta <-
  function(df_share_smooth, X, M, V, delta, sigma, mu, omega, kappa, lambda) {
    # initial distance
    distance <- 10000
    # fixed-point algorithm
    delta_old <- delta
    while (distance > lambda) {
      # save the old delta
      delta_old$delta <- delta$delta
      # compute the share with the old delta
      df_share_smooth_predicted <-
        compute_share_smooth_delta(X, M, V, delta_old, sigma, mu, omega)
      # update the delta
      delta$delta <- delta_old$delta +
        (log(df_share_smooth$s) - log(df_share_smooth_predicted$s)) * kappa
      delta <- delta %>%
        dplyr::mutate(delta = ifelse(j == 0, 0, delta))
      # update the distance
      distance <- max(abs(delta$delta - delta_old$delta))
    }
    return(delta)
  }


kappa <- 1
lambda <- 1e-3
delta_new <-
  solve_delta(df_share_smooth, X, M, V, delta, sigma, mu, omega, kappa, lambda)
head(delta_new)
```

```
## # A tibble: 6 x 3
##       t     j delta
##   <int> <dbl> <dbl>
## 1     1     0  0
## 2     1     6 -2.40
## 3     2     0  0
## 4     2     1 -1.15
## 5     2    10 -1.22
## 6     3     0  0
```

```r
summary(delta_new$delta - delta$delta)
```

```
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## -3.553e-15 -4.441e-16  0.000e+00 -5.745e-17  0.000e+00  1.776e-15
```

9. Check how long it takes to compute the limit δ under the Monte Carlo shocks starting from the true δ to match with `df_share_smooth`. This is approximately the time to evaluate the objective function.

```r
delta_new <-
  solve_delta(df_share_smooth, X, M, V_mcmc, delta, sigma, mu, omega, kappa, lambda)
save(delta_new, file = "data/A4_delta_new.RData")
```

```r
delta_new <- get(load(file = "data/A4_delta_new.RData"))
summary(delta_new$delta - delta$delta)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1.30836 -0.30499  0.00000 -0.06422  0.16800  1.36169
```

10. We use the marginal cost c_{jt} as the excluded instrumental variable for p_{jt}. Let Ψ be the weighing matrix for the GMM estimator. For now, let it be the identity matrix. Write a function `compute_theta_linear(df_share_smooth, delta, mu, omega, Psi)` that returns the optimal linear parameters associated with the data and δ. Notice that we only obtain β_0 in this way because α_0 is directly computed from the non-linear parameters by $-\exp(\mu + \omega^2/2)$. The first order condition for β_0 is:

$$\beta_0 = (X'W\Phi^{-1}W'X)^{-1}X'W\Phi^{-1}W'[\delta - \alpha_0 p], \tag{1}$$

where

$$X = \begin{pmatrix} x'_{11} \\ \vdots \\ x'_{J_1 1} \\ \vdots \\ x'_{1T} \\ \vdots \\ x_{J_T T} \end{pmatrix} \tag{2}$$

$$W = \begin{pmatrix} x'_{11} & c_{11} \\ \vdots & \vdots \\ x'_{J_1 1} & c_{J_1 1} \\ \vdots & \vdots \\ x'_{1T} & c_{1T} \\ \vdots & \vdots \\ x_{J_T T} & c_{J_T T} \end{pmatrix}, \tag{3}$$

$$\delta = \begin{pmatrix} \delta_1 1 \\ \vdots \\ \delta_{J_1 1} \\ \vdots \\ \delta_1 T \\ \vdots \\ \delta_{J_T T} \end{pmatrix} \tag{4}$$

,

where $\alpha_0 = -\exp(\mu + \omega^2/2)$. Notice that X and W does not include rows for the outwide option.

```
# compute the optimal linear parameters
compute_theta_linear <-
  function(df_share_smooth, delta, mu, omega, Psi) {
    # extract matrices
    X <- df_share_smooth %>%
      dplyr::filter(j != 0) %>%
      dplyr::select(dplyr::starts_with("x_")) %>%
      as.matrix()
    p <- df_share_smooth %>%
      dplyr::filter(j != 0) %>%
      dplyr::select(p) %>%
      as.matrix()
    W <- df_share_smooth %>%
      dplyr::filter(j != 0) %>%
      dplyr::select(dplyr::starts_with("x_"), c) %>%
      as.matrix()
    delta_m <- delta %>%
      dplyr::filter(j != 0) %>%
      dplyr::select(delta) %>%
      as.matrix()
    alpha <- - exp(mu + omega^2/2)
    # compute the optimal linear parameters
    theta_linear_1 <-
      crossprod(X, W) %*%
      solve(Psi, crossprod(W, X))
    theta_linear_2 <-
      crossprod(X, W) %*%
      solve(Psi, crossprod(W, delta_m - alpha * p))
    theta_linear <- solve(theta_linear_1, theta_linear_2)
    return(theta_linear)
  }

Psi <- diag(length(beta) + 1)
theta_linear <-
  compute_theta_linear(df_share_smooth, delta, mu, omega, Psi)
cbind(theta_linear, beta)
```

```
##          delta       beta
## x_1  3.9935855  4.0000000
## x_2  0.1552469  0.1836433
## x_3 -0.7838712 -0.8356286
```

11. Write a function `solve_xi(df_share_smooth, delta, beta, mu, omega)` that computes the values of ξ that are implied from the data, δ, and the linear parameters. Check that the (almost) true values are returned when true δ and the true linear parmaeters are passed to the function. Notice that the returend ξ should not include rows for the outside option.

```r
# solve xi associated with delta and linear parameters
solve_xi <-
  function(df_share_smooth, delta, beta, mu, omega) {
    # extract matrices
    X1 <- df_share_smooth %>%
      dplyr::filter(j != 0) %>%
      dplyr::select(dplyr::starts_with("x_"), p) %>%
      as.matrix()
    delta_m <- delta %>%
      dplyr::filter(j != 0) %>%
      dplyr::select(delta) %>%
      as.matrix()
    alpha <- - exp(mu + omega^2/2)
    theta_linear <- c(beta, alpha)
    # compute xi
    xi <- delta_m - X1 %*% theta_linear
    colnames(xi) <- "xi"
    # return
    return(xi)
  }

xi_new <- solve_xi(df_share_smooth, delta, beta, mu, omega)
head(xi_new)
```

```
##               xi
## [1,] -0.05139386
## [2,] -0.19714498
## [3,] -0.35374758
## [4,]  0.18229098
## [5,]  0.38426646
## [6,] -0.05617311
```

```r
xi_true <-
  df_share_smooth %>%
  dplyr::filter(j != 0) %>%
  dplyr::select(xi)
summary(xi_true - xi_new)
```

```
##        xi
##  Min.   :-4.441e-16
##  1st Qu.:-8.327e-17
##  Median : 0.000e+00
##  Mean   :-4.134e-18
##  3rd Qu.: 6.765e-17
##  Max.   : 6.661e-16
```

11. Write a function `GMM_objective_A4(theta_nonlinear, delta, df_share_smooth, Psi, X, M, V_mcmc, kappa, lambda)` that returns the value of the GMM objective function as a function of non-linear parameters `mu`, `omega`, and `sigma`:

$$\min_{\theta} \xi(\theta)' W \Phi^{-1} W' \xi(\theta),$$

where $\xi(\theta)$ is the values of ξ that solves:

$$s = \sigma(p, x, \xi),$$

given parameters θ. Note that the row of $\xi(\theta)$ and W do not include the rows for the outside options.

```
# non-linear parmaeters
theta_nonlinear <- c(mu, omega, sigma)
```

```
# compute GMM objective function
GMM_objective_A4 <-
  function(theta_nonlinear, delta, df_share_smooth, Psi,
           X, M, V_mcmc, kappa, lambda) {
    # exctract parameters
    mu <- theta_nonlinear[1]
    omega <- theta_nonlinear[2]
    sigma <- theta_nonlinear[3:length(theta_nonlinear)]
    # extract matrix
    W <- df_share_smooth %>%
      dplyr::filter(j != 0) %>%
      dplyr::select(dplyr::starts_with("x_"), c) %>%
      as.matrix()
    # compute the delta that equates the actual and predicted shares
    delta <-
      solve_delta(df_share_smooth, X, M, V_mcmc,
                  delta, sigma, mu, omega, kappa, lambda)
    # compute the optimal linear parameters
    beta <-
      compute_theta_linear(df_share_smooth, delta, mu, omega, Psi)
    # compute associated xi
    xi <- solve_xi(df_share_smooth, delta, beta, mu, omega)
    # compute objective
    objective <- crossprod(xi, W) %*% solve(Psi, crossprod(W, xi))
    # return
    return(objective)
  }

# compute GMM objective function
objective <-
  GMM_objective_A4(theta_nonlinear, delta, df_share_smooth, Psi,
                   X, M, V_mcmc, kappa, lambda)
save(objective, file = "data/A4_objective.RData")
```

```
objective <- get(load(file = "data/A4_objective.RData"))
objective
```

```
##           xi
## xi 0.1368324
```

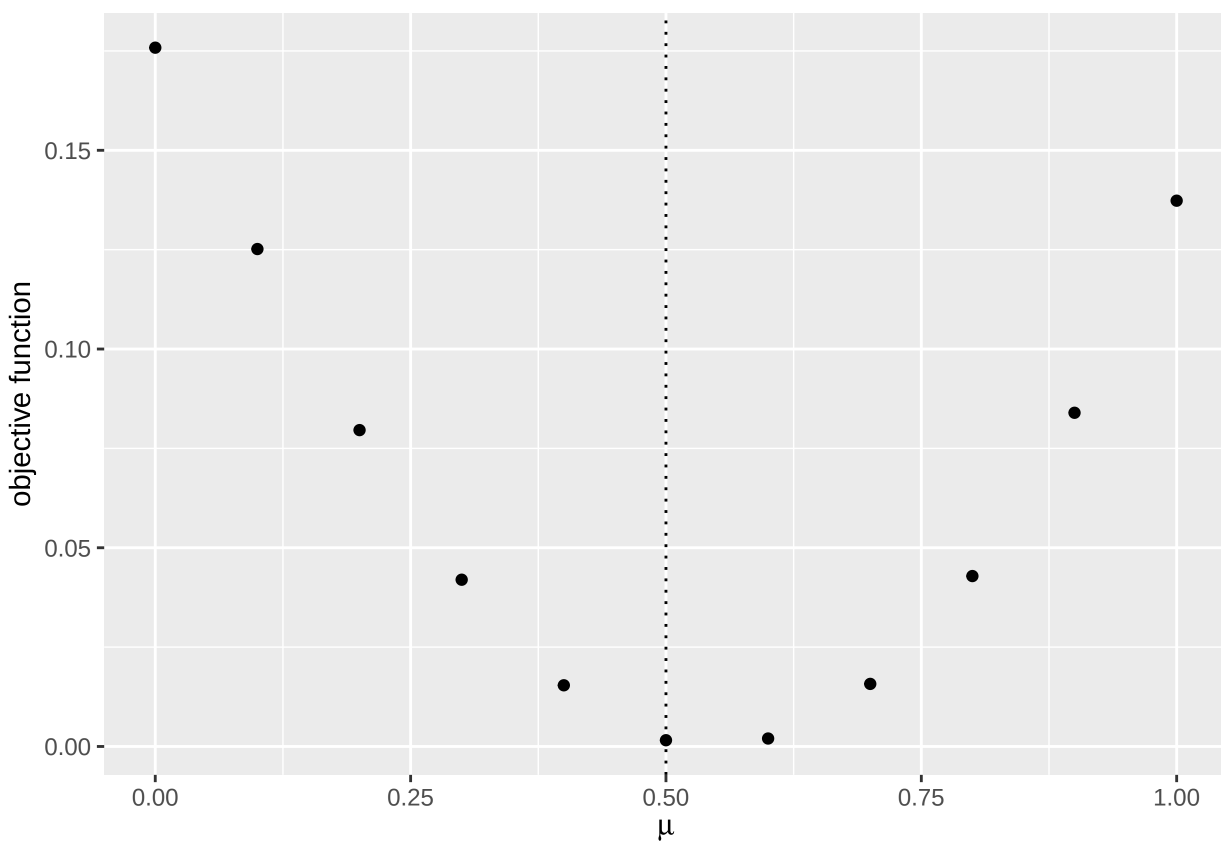
12. Draw a graph of the objective function that varies each non-linear parameter from 0, 0.2, $\cdots$, 2.0 of the true value. Try with the actual shocks `V`.

```
label <- c( "\\mu",
           "\\omega",
           paste("\\sigma_", 1:K, sep = ""))
label <- paste("$", label, "$", sep = "")
graph_true <- foreach (i = 1:length(theta_nonlinear)) %do% {
```

```
  theta_nonlinear_i <- theta_nonlinear[i]
  theta_nonlinear_i_list <- theta_nonlinear_i * seq(0, 2, by = 0.2)
  objective_i <-
    foreach (theta_nonlinear_ij = theta_nonlinear_i_list,
             .combine = "rbind") %dopar% {
               theta_nonlinear_j <- theta_nonlinear
               theta_nonlinear_j[i] <- theta_nonlinear_ij
               objective_ij <-
                 GMM_objective_A4(theta_nonlinear_j, delta, df_share_smooth, Psi,
                                  X, M, V, kappa, lambda)
               return(objective_ij)
             }
  df_graph <-
    data.frame(x = theta_nonlinear_i_list, y = as.numeric(objective_i))
  g <- ggplot(data = df_graph, aes(x = x, y = y)) +
    geom_point() +
    geom_vline(xintercept = theta_nonlinear_i, linetype = "dotted") +
    ylab("objective function") + xlab(TeX(label[i]))
  return(g)
}
save(graph_true, file = "data/A4_graph_true.RData")
```

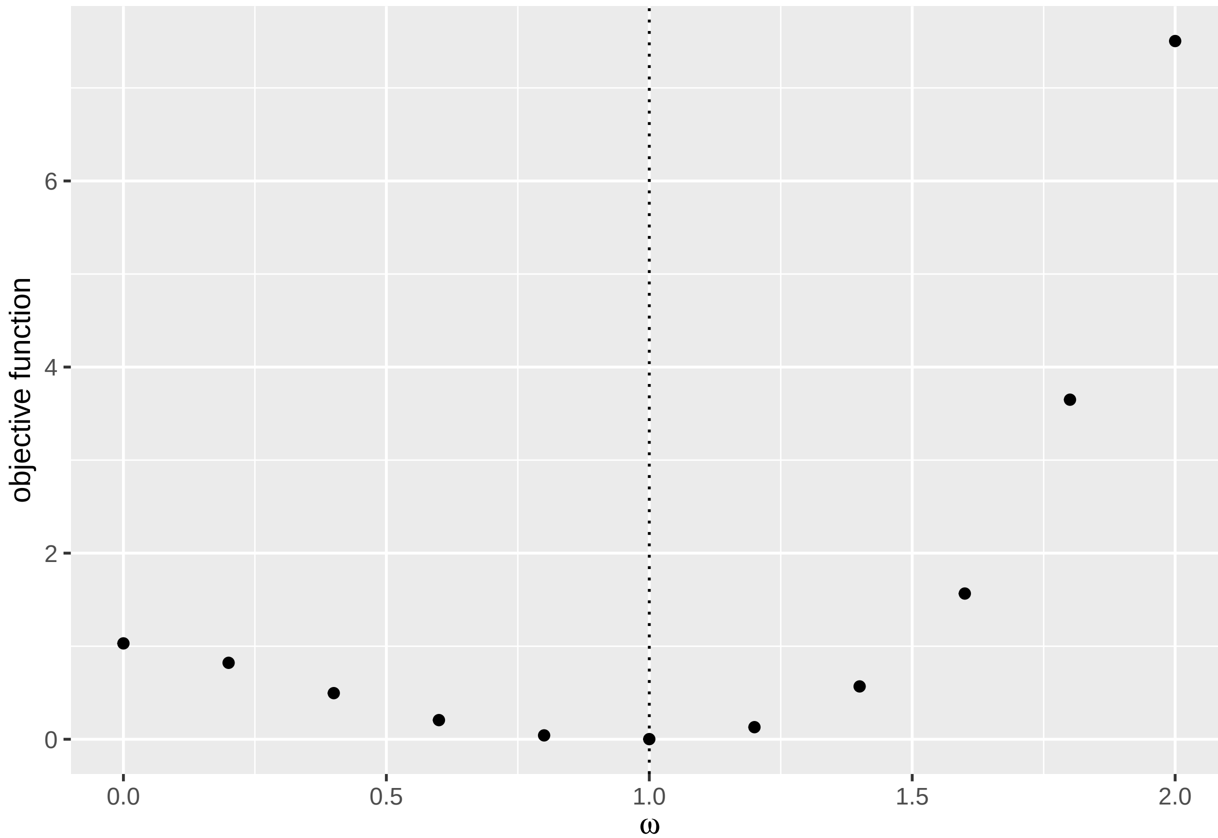
```
graph_true <- get(load(file = "data/A4_graph_true.RData"))
graph_true
```

```
## [[1]]
```

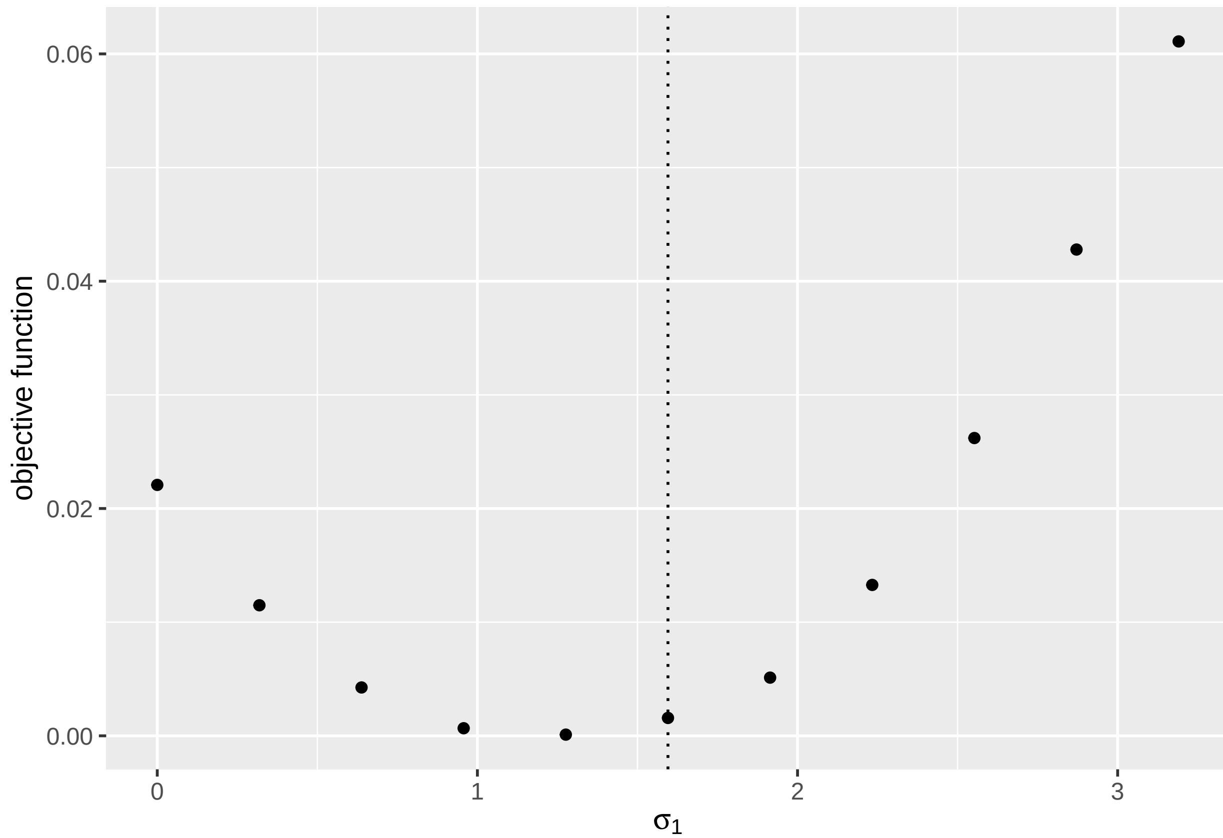



```
##
```

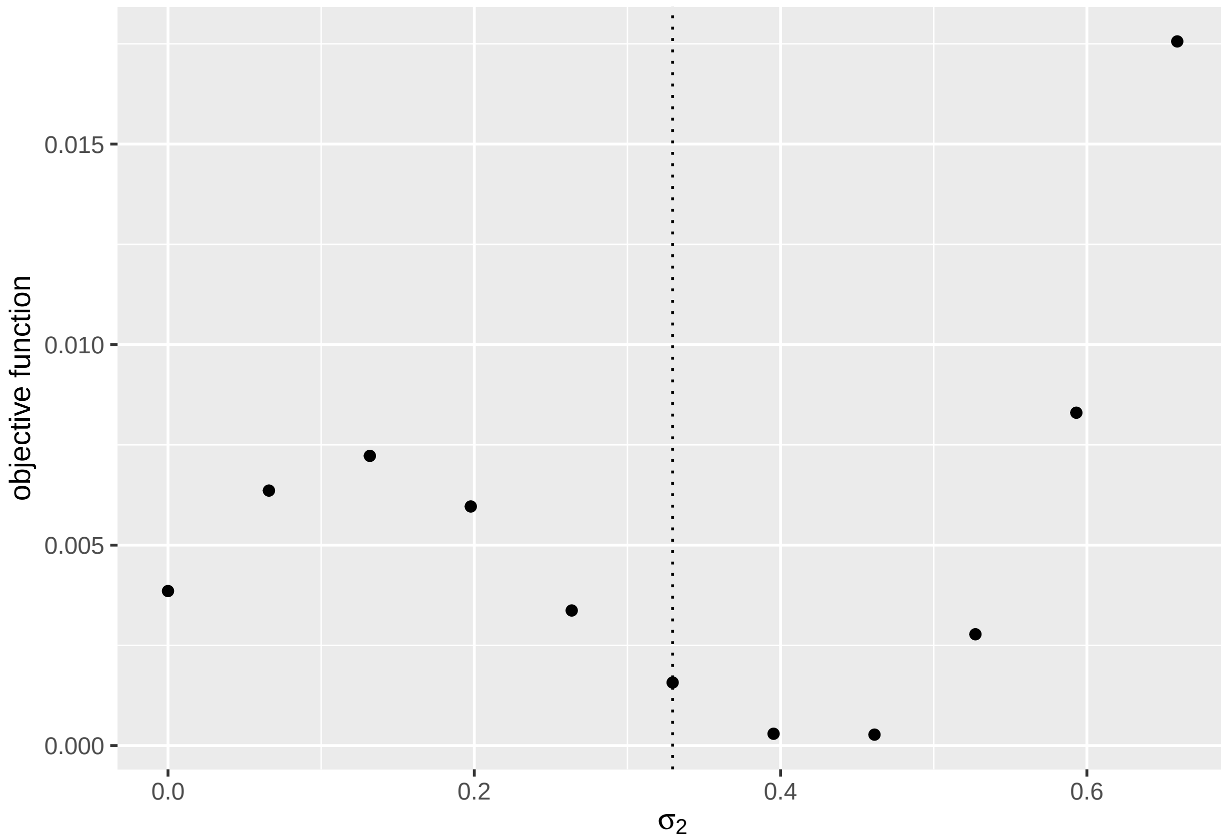
```
## [[2]]
```



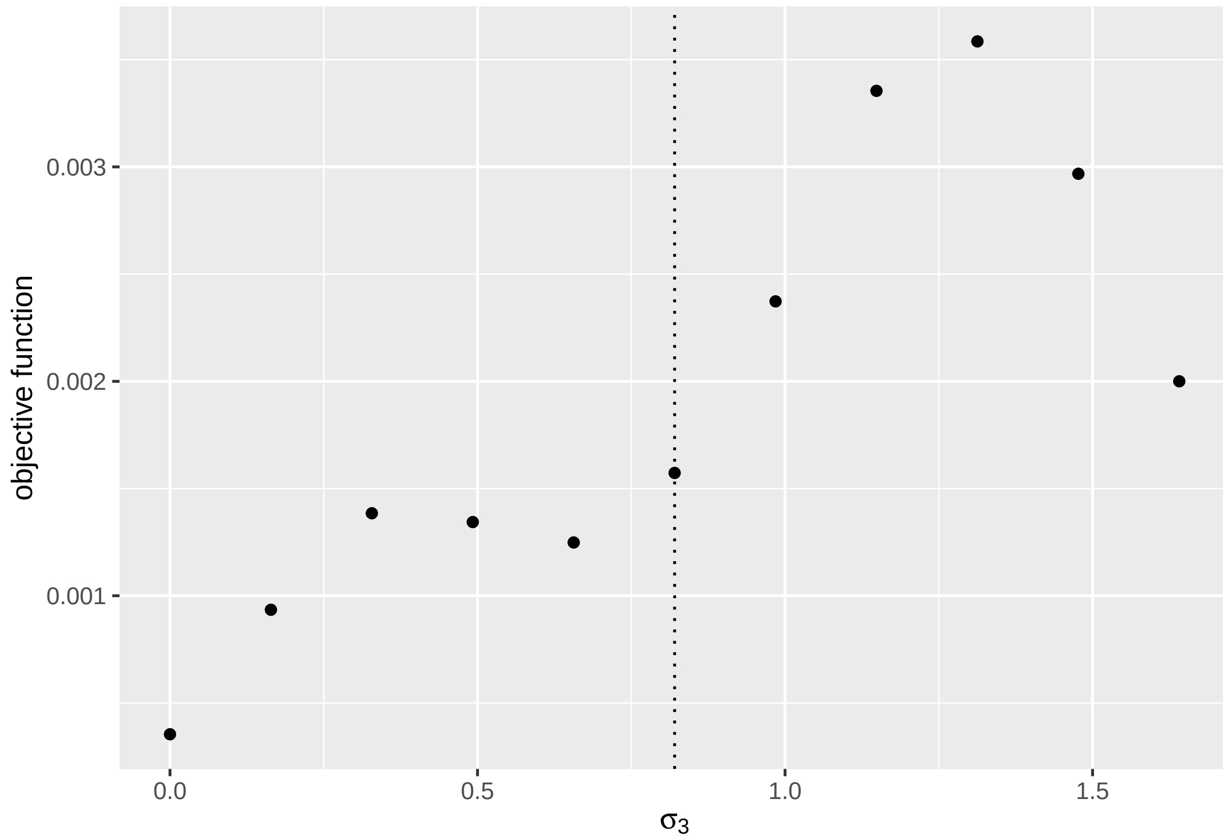

```
##
## [[3]]
```




```
##
## [[4]]
```




```
##
## [[5]]
```



13. Find non-linear parameters that minimize the GMM objective function. Because standard deviations of the same absolute value with positive and negative values have almost the same implication for the data, you can take the absolute value if the estimates of the standard deviations happened to be negative (Another way is to set the non-negativity constraints on the standard deviations).

```
result <-
  optim(par = theta_nonlinear,
        fn = GMM_objective_A4,
        method = "BFGS",
        delta = delta,
        df_share_smooth = df_share_smooth,
        Psi = Psi,
        X = X, M = M,
        V_mcmc = V_mcmc,
        kappa = kappa,
        lambda = lambda)
save(result, file = "data/A4_result.RData")
```

```
result <- get(load(file = "data/A4_result.RData"))
result
```

```
## $par
## [1] 0.3635051 0.7502512 1.5852184 0.4223104 0.8757237
##
## $value
## [1] 2.064906e-18
##
## $counts
```

```
## function gradient
##       54       11
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```r
comparison <- cbind(theta_nonlinear, abs(result$par))
colnames(comparison) <- c("true", "estimate")
comparison
```

```
##           true  estimate
## [1,] 0.5000000 0.3635051
## [2,] 1.0000000 0.7502512
## [3,] 1.5952808 1.5852184
## [4,] 0.3295078 0.4223104
## [5,] 0.8204684 0.8757237
```