Assignment 3: Demand Function Estimation I

Kohei Kawaguchi

Simulate data

We simulate data from a discrete choice model. 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 $\xi_{jt}, \nu_{it}, \upsilon_{it}, \epsilon_{ijt}$.

In this assignment, we consider a model with $\xi_{jt} = 0$, i.e., the model without the unobserved fixed effects. However, the code to simulate data should be written for general ξ_{jt}, so that we can use the same code in the next assignment in which we consider a model with the unobserved fixed effects.

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:

$$\xi_{jt} = 0.$$

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
prop_jt <- 0.6
sd_x <- 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  0.244  -0.00810
##  3     2     1  0.369   0.472
##  4     3     1  0.288   0.411
##  5     4     1 -0.153   0.297
##  6     5     1  0.756   0.459
##  7     6     1  0.195   0.391
##  8     7     1 -0.311   0.0373
##  9     8     1 -1.11   -0.995
## 10     9     1  0.562   0.310
## 11    10     1 -0.0225 -0.0281
```

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 `1 - prop_jt` products from each market using `dplyr::sample_frac` function. 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 = 0 * 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(prop_jt) %>%
  dplyr::ungroup()
# add outside option
outside <- data.frame(j = 0, t = 1:T, xi = 0, c = 0, p = 0)
M <- rbind(
  M,
  outside
```

```r
) %>%
  dplyr::arrange(t, j)
```

```r
M
```

```
## # A tibble: 700 x 5
##        j     t    xi     c     p
##    <dbl> <int> <dbl> <dbl> <dbl>
##  1     0     1     0 0      0
##  2     1     1     0 0.951  1.93
##  3     2     1     0 1.04   1.96
##  4     3     1     0 0.937  1.89
##  5     4     1     0 1.03   2.07
##  6     6     1     0 0.980  1.96
##  7     7     1     0 0.961  1.94
##  8     0     2     0 0      0
##  9     4     2     0 1.00   2.05
## 10     6     2     0 1.01   2.03
## # ... with 690 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.

```r
# 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()
```

```r
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  1.13   -0.839   2.26   -0.325
##  2     2     1  1.24    0.217  -0.0313  0.0643
##  3     3     1 -0.106   0.232   0.0135  0.464
##  4     4     1  0.645   0.662   0.841   1.02
##  5     5     1  0.0842 -1.85   -0.440  -0.0970
##  6     6     1 -0.265  -0.800  -0.113  -1.13
##  7     7     1 -2.39    0.718   2.91   -0.999
##  8     8     1 -0.592   0.184  -0.504  -1.55
##  9     9     1  0.0328 -0.0566  0.840  -0.372
## 10    10     1  1.08    1.31   -1.19   -0.179
## # ... 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.

```r
# 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")) %>%
```

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

```r
df
```

```
## # A tibble: 350,000 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  1.13 -0.839  2.26   -0.325      0  0       0           0
##  2     1     1     1  1.13 -0.839  2.26   -0.325      1  0.244 -0.00810      0
##  3     1     1     2  1.13 -0.839  2.26   -0.325      1  0.369  0.472        0
##  4     1     1     3  1.13 -0.839  2.26   -0.325      1  0.288  0.411        0
##  5     1     1     4  1.13 -0.839  2.26   -0.325      1 -0.153  0.297        0
##  6     1     1     6  1.13 -0.839  2.26   -0.325      1  0.195  0.391        0
##  7     1     1     7  1.13 -0.839  2.26   -0.325      1 -0.311  0.0373       0
##  8     1     2     0  1.24  0.217 -0.0313  0.0643     0  0       0           0
##  9     1     2     1  1.24  0.217 -0.0313  0.0643     1  0.244 -0.00810      0
## 10     1     2     2  1.24  0.217 -0.0313  0.0643     1  0.369  0.472        0
## # ... with 349,990 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]  1.80380489 -1.71157721  0.16567015 -0.79664041  0.09569216  3.35804055
```

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.

```r
# compute indirect utility
compute_indirect_utility <-
  function(df, beta, 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))
    xi <- as.matrix(dplyr::select(df, xi))
    # random coefficients
    beta_i <- as.matrix(rep(1, dim(v_x)[1])) %*% t(as.matrix(beta)) + v_x %*% diag(sigma)
    alpha_i <- - exp(mu + omega * v_p)
    # conditional mean indirect utility
    value <- as.matrix(rowSums(beta_i * X) + p * alpha_i + xi)
    colnames(value) <- "u"
    return(value)
  }
u <-
  compute_indirect_utility(
    df, beta, sigma,
           mu, omega)
head(u)
```

```
##              u
## [1,] 0.000000
## [2,] 3.470073
## [3,] 3.917173
## [4,] 3.946269
## [5,] 3.656539
## [6,] 3.851153
```

8. Write a function `compute_choice(X, M, V, e, beta, sigma, mu, omega)` that first construct `df` from `X`, `M`, `V`, second call `compute_indirect_utility` to obtain the vector of mean indirect utilities `u`, third compute the choice vector `q` based on the vector of mean indirect utilities and `e`, and finally return the data frame to which `u` and `q` are added as columns.

```r
# compute choice
compute_choice <-
  function(X, M, V, e, 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 and e
    df_choice <- data.frame(df, u, e) %>%
      tibble::as_tibble()
    # make choice
    df_choice <- df_choice %>%
      dplyr::group_by(t, i) %>%
      dplyr::mutate(q = ifelse(u + e == max(u + e), 1, 0)) %>%
      dplyr::ungroup()
    # return
    return(df_choice)
  }
df_choice <-
  compute_choice(X, M, V, e, beta, sigma,
                 mu, omega)
df_choice
```

```
## # A tibble: 350,000 x 16
##        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  1.13 -0.839   2.26 -0.325     0 0      0              0
##  2     1     1     1  1.13 -0.839   2.26 -0.325     1 0.244 -0.00810        0
##  3     1     1     2  1.13 -0.839   2.26 -0.325     1 0.369  0.472          0
##  4     1     1     3  1.13 -0.839   2.26 -0.325     1 0.288  0.411          0
```

```
##  5     1     1     4  1.13 -0.839  2.26   -0.325      1 -0.153  0.297       0
##  6     1     1     6  1.13 -0.839  2.26   -0.325      1  0.195  0.391       0
##  7     1     1     7  1.13 -0.839  2.26   -0.325      1 -0.311  0.0373      0
##  8     1     2     0  1.24  0.217 -0.0313  0.0643     0  0      0           0
##  9     1     2     1  1.24  0.217 -0.0313  0.0643     1  0.244 -0.00810     0
## 10     1     2     2  1.24  0.217 -0.0313  0.0643     1  0.369  0.472       0
## # ... with 349,990 more rows, and 5 more variables: c <dbl>, p <dbl>, u <dbl>,
## #   e <dbl>, q <dbl>
```

```r
summary(df_choice)
```

```
##        t                i               j              v_x_1
##  Min.   :  1.00   Min.   :  1.0   Min.   : 0.000   Min.   :-4.471031
##  1st Qu.: 25.75   1st Qu.:125.8   1st Qu.: 2.000   1st Qu.:-0.678521
##  Median : 50.50   Median :250.5   Median : 5.000   Median : 0.000271
##  Mean   : 50.50   Mean   :250.5   Mean   : 4.716   Mean   :-0.001822
##  3rd Qu.: 75.25   3rd Qu.:375.2   3rd Qu.: 8.000   3rd Qu.: 0.678426
##  Max.   :100.00   Max.   :500.0   Max.   :10.000   Max.   : 3.876826
##      v_x_2              v_x_3               v_p                x_1
##  Min.   :-3.745932   Min.   :-4.229081   Min.   :-5.117722   Min.   :0.0000
##  1st Qu.:-0.673627   1st Qu.:-0.667369   1st Qu.:-0.675217   1st Qu.:1.0000
##  Median : 0.004978   Median :-0.002327   Median : 0.005292   Median :1.0000
##  Mean   : 0.000925   Mean   : 0.003736   Mean   : 0.000180   Mean   :0.8571
##  3rd Qu.: 0.672264   3rd Qu.: 0.668596   3rd Qu.: 0.676980   3rd Qu.:1.0000
##  Max.   : 4.193857   Max.   : 4.326689   Max.   : 4.267654   Max.   :1.0000
##       x_2                x_3                xi          c
##  Min.   :-1.10735   Min.   :-0.9947   Min.   :0   Min.   :0.0000
##  1st Qu.:-0.02247   1st Qu.: 0.0000   1st Qu.:0   1st Qu.:0.9412
##  Median : 0.19492   Median : 0.2970   Median :0   Median :0.9866
##  Mean   : 0.08733   Mean   : 0.1273   Mean   :0   Mean   :0.8566
##  3rd Qu.: 0.36916   3rd Qu.: 0.4106   3rd Qu.:0   3rd Qu.:1.0267
##  Max.   : 0.75589   Max.   : 0.4719   Max.   :0   Max.   :1.1996
##        p               u                  e                 q
##  Min.   :0.000   Min.   :-231.205   Min.   :-2.6364   Min.   :0.0000
##  1st Qu.:1.914   1st Qu.:  -2.213   1st Qu.:-0.3302   1st Qu.:0.0000
##  Median :1.985   Median :   0.000   Median : 0.3635   Median :0.0000
##  Mean   :1.715   Mean   :  -1.328   Mean   : 0.5761   Mean   :0.1429
##  3rd Qu.:2.043   3rd Qu.:   1.972   3rd Qu.: 1.2415   3rd Qu.:0.0000
##  Max.   :2.197   Max.   :  10.197   Max.   :14.0966   Max.   :1.0000
```

9. Write a function `compute_share(X, M, V, e, beta, sigma, mu, omega)` that first construct `df` from `X`, `M`, `V`, second call `compute_choice` 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
compute_share <-
  function(X, M, V, e, beta, sigma,
           mu, omega) {
    # constants
    T <- max(M$t)
    N <- max(V$i)
    J <- max(X$j)
    # compute choice
    df_choice <-
```

```r
      compute_choice(X, M, V, e, beta, sigma,
                     mu, omega)
    # make share data
    df_share <- df_choice %>%
      dplyr::select(-dplyr::starts_with("v_"), -u, -e, -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 <- df_share %>%
      dplyr::group_by(t) %>%
      dplyr::mutate(y = log(s/sum(s * (j == 0)))) %>%
      dplyr::ungroup()
    return(df_share)
  }
df_share <-
  compute_share(X, M, V, e, beta, sigma,
                mu, omega)
df_share
```

```
## # A tibble: 700 x 11
##        t     j   x_1    x_2       x_3    xi     c     p     q     s      y
##    <int> <dbl> <dbl>  <dbl>     <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>
##  1     1     0     0  0      0            0 0      0      163 0.326  0
##  2     1     1     1  0.244 -0.00810      0 0.951  1.93    92 0.184 -0.572
##  3     1     2     1  0.369  0.472        0 1.04   1.96    47 0.094 -1.24
##  4     1     3     1  0.288  0.411        0 0.937  1.89    64 0.128 -0.935
##  5     1     4     1 -0.153  0.297        0 1.03   2.07    32 0.064 -1.63
##  6     1     6     1  0.195  0.391        0 0.980  1.96    50 0.1   -1.18
##  7     1     7     1 -0.311  0.0373       0 0.961  1.94    52 0.104 -1.14
##  8     2     0     0  0      0            0 0      0      157 0.314  0
##  9     2     4     1 -0.153  0.297        0 1.00   2.05    31 0.062 -1.62
## 10     2     6     1  0.195  0.391        0 1.01   2.03    28 0.056 -1.72
## # ... with 690 more rows
```

```r
summary(df_share)
```

```
##        t                j               x_1              x_2          
##  Min.   :  1.00   Min.   : 0.000   Min.   :0.0000   Min.   :-1.10735  
##  1st Qu.: 25.75   1st Qu.: 2.000   1st Qu.:1.0000   1st Qu.:-0.02247  
##  Median : 50.50   Median : 5.000   Median :1.0000   Median : 0.19492  
##  Mean   : 50.50   Mean   : 4.716   Mean   :0.8571   Mean   : 0.08733  
##  3rd Qu.: 75.25   3rd Qu.: 8.000   3rd Qu.:1.0000   3rd Qu.: 0.36916  
##  Max.   :100.00   Max.   :10.000   Max.   :1.0000   Max.   : 0.75589  
##       x_3                xi          c                p        
##  Min.   :-0.9947   Min.   :0   Min.   :0.0000   Min.   :0.000  
##  1st Qu.: 0.0000   1st Qu.:0   1st Qu.:0.9412   1st Qu.:1.914  
##  Median : 0.2970   Median :0   Median :0.9866   Median :1.985  
##  Mean   : 0.1273   Mean   :0   Mean   :0.8566   Mean   :1.715  
##  3rd Qu.: 0.4106   3rd Qu.:0   3rd Qu.:1.0267   3rd Qu.:2.043  
```

```
##  Max.   : 0.4719   Max.   :0   Max.   :1.1996   Max.   :2.197
##        q                s                y
##  Min.   : 25.00   Min.   :0.0500   Min.   :-1.96851
##  1st Qu.: 44.00   1st Qu.:0.0880   1st Qu.:-1.35239
##  Median : 52.00   Median :0.1040   Median :-1.16761
##  Mean   : 71.43   Mean   :0.1429   Mean   :-1.00050
##  3rd Qu.: 71.00   3rd Qu.:0.1420   3rd Qu.:-0.84881
##  Max.   :193.00   Max.   :0.3860   Max.   : 0.01316
```

Estimate the parameters

1. Estimate the parameters assuming there is no consumer-level heterogeneity, i.e., by assuming:

$$\ln \frac{s_{jt}}{s_{0t}} = \beta' x_{jt} + \alpha p_{jt}.$$

   This can be implemented using `lm` function. Print out the estimate results.

```r
# logit regression
result_logit <- lm(
  data = df_share,
  formula = y ~ - 1 + x_1 + x_2 + x_3 + p
)
summary(result_logit)
```

```
##
## Call:
## lm(formula = y ~ -1 + x_1 + x_2 + x_3 + p, data = df_share)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.66624 -0.08915  0.00000  0.10364  0.45654
##
## Coefficients:
##     Estimate Std. Error t value Pr(>|t|)
## x_1  1.10817    0.18980   5.838 8.08e-09 ***
## x_2  0.19987    0.02908   6.873 1.40e-11 ***
## x_3 -0.88693    0.03480 -25.485  < 2e-16 ***
## p   -1.08146    0.09474 -11.415  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1718 on 696 degrees of freedom
## Multiple R-squared:  0.9769, Adjusted R-squared:  0.9768
## F-statistic:  7366 on 4 and 696 DF,  p-value: < 2.2e-16
```

We estimate the model using simulated share.

When optimizing an objective function that uses the Monte Carlo simulation, it is important to keep the realizations of the shocks the same across the evaluations of the objective function. If the realization of the shocks differ across the objective function evaluations, the optimization algorithm will not converge because it cannot distinguish the change in the value of the objective function due to the difference in the parameters and the difference in the realized shocks.

The best practice to avoid this problem is to generate the shocks outside the optimization algorithm as in the current case. If the size of the shocks can be too large to store in the memory, the second best practice is to make sure to set the seed inside the optimization algorithm so that the realized shocks are the same across function evaluations.

2. For this reason, we 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`.

```
# 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()
```

```
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 -0.901  -0.126 -0.0566  1.18
##  2     2     1  0.0850  0.495  0.813   0.112
##  3     3     1  0.305  -0.387 -1.29   -1.21
##  4     4     1 -0.0423  2.97  -0.982   1.15
##  5     5     1  1.02    0.521 -0.631  -1.93
##  6     6     1  0.375   0.281  0.191   0.420
##  7     7     1  0.129  -0.156  0.808   0.282
##  8     8     1  0.378   0.716  0.0401 -0.132
##  9     9     1  0.0196 -1.29  -1.22   -0.811
## 10    10     1 -0.156   2.72   0.280   0.709
## # ... 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] -0.6548845 -0.7244753 -1.1280240  0.5738110  1.1362857  2.0492881
```

3. Use `compute_share` to check the simulated share at the true parameter using the Monte Carlo shocks. Remember that the number of consumers should be set at `L` instead of `N`.

```
# compute predicted share
df_share_mcmc <-
  compute_share(X, M, V_mcmc, e_mcmc, beta, sigma,
                mu, omega)
```

```
df_share_mcmc
```

```
## # A tibble: 700 x 11
##        t     j   x_1    x_2      x_3    xi      c      p      q      s       y
```

```
##    <int> <dbl> <dbl>  <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>
##  1     1     0     0  0      0           0 0     0       161 0.322  0
##  2     1     1     1  0.244 -0.00810     0 0.951 1.93     76 0.152 -0.751
##  3     1     2     1  0.369  0.472      0 1.04  1.96     50 0.1   -1.17
##  4     1     3     1  0.288  0.411      0 0.937 1.89     52 0.104 -1.13
##  5     1     4     1 -0.153  0.297      0 1.03  2.07     53 0.106 -1.11
##  6     1     6     1  0.195  0.391      0 0.980 1.96     49 0.098 -1.19
##  7     1     7     1 -0.311  0.0373     0 0.961 1.94     59 0.118 -1.00
##  8     2     0     0  0      0           0 0     0       151 0.302  0
##  9     2     4     1 -0.153  0.297      0 1.00  2.05     40 0.08  -1.33
## 10     2     6     1  0.195  0.391      0 1.01  2.03     40 0.08  -1.33
## # ... with 690 more rows
```

5. 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  0.5000000
## [8]  1.0000000
```

6. Write a function `NLLS_objective_A3(theta, df_share, X, M, V_mcmc, e_mcmc)` that first computes the simulated share and then compute the mean-squared error between the share data.

```
# NLLS objective function
NLLS_objective_A3 <-
  function(theta, df_share, X, M, V_mcmc, e_mcmc) {
    # constants
    K <- length(grep("x_", colnames(X)))
    # extract parameters
    beta <- theta[1:K]
    sigma <- theta[(K + 1):(2 * K)]
    mu <- theta[2 * K + 1]
    omega <- theta[2 * K + 2]
    # compute predicted share
    df_share_mcmc <-
      compute_share(X, M, V_mcmc, e_mcmc, beta, sigma,
                    mu, omega)
    # compute distance
    distance <- mean((df_share_mcmc$s - df_share$s)^2)
    # return
    return(distance)
  }
NLLS_objective <- NLLS_objective_A3(theta, df_share, X, M, V_mcmc, e_mcmc)
```

```
NLLS_objective
```

```
## [1] 0.0004611771
```

7. Draw a graph of the objective function that varies each parameter from 0.5, 0.6, $\cdots$, 1.5 of the true value. First try with the actual shocks `V` and `e` and then try with the Monte Carlo shocks `V_mcmc` and `e_mcmc`. You will some of the graph does not look good with the Monte Carlo shocks. It will cause the approximation error.

Because this takes time, you may want to parallelize the computation using `%dopar` functionality of `foreach` loop. To do so, first install `doParallel` package and then load it and register the workers as follows:

```r
registerDoParallel()
```

This automatically detect the number of cores available at your computer and registers them as the workers. Then, you only have to change `%do%` to `%dopar` in the `foreach` loop as follows:

```r
foreach (i = 1:4) %dopar% {
  # this part is parallelized
  y <- 2 * i
  return(y)
}
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 6
##
## [[4]]
## [1] 8
```

In windows, you may have to explicitly pass packages, functions, and data to the worker by using `.export` and `.packages` options as follows:

```r
temp_func <- function(x) {
  y <- 2 * x
  return(y)
}
foreach (i = 1:4,
         .export = "temp_func",
         .packages = "magrittr") %dopar% {
  # this part is parallelized
  y <- temp_func(i)
  return(y)
}
```

```
## Warning in e$fun(obj, substitute(ex), parent.frame(), e$data): already exporting
## variable(s): temp_func
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 6
##
## [[4]]
## [1] 8
```

If you have called a function in a package in this way `dplyr::mutate`, then you will not have to pass `dplyr` by `.packages` option. This is one of the reasons why I prefer to explicitly call the package every time I call a function. If you have compiled your functions in a package, you will just have to pass the package as follows:

```r
# this function is compiled in the package EmpiricalIO
# temp_func <- function(x) {
#   y <- 2 * x
#   return(y)
# }
foreach (i = 1:4,
         .packages = c(
           "EmpiricalIO",
           "magrittr")) %dopar% {
  # this part is parallelized
  y <- temp_func(i)
  return(y)
}
```

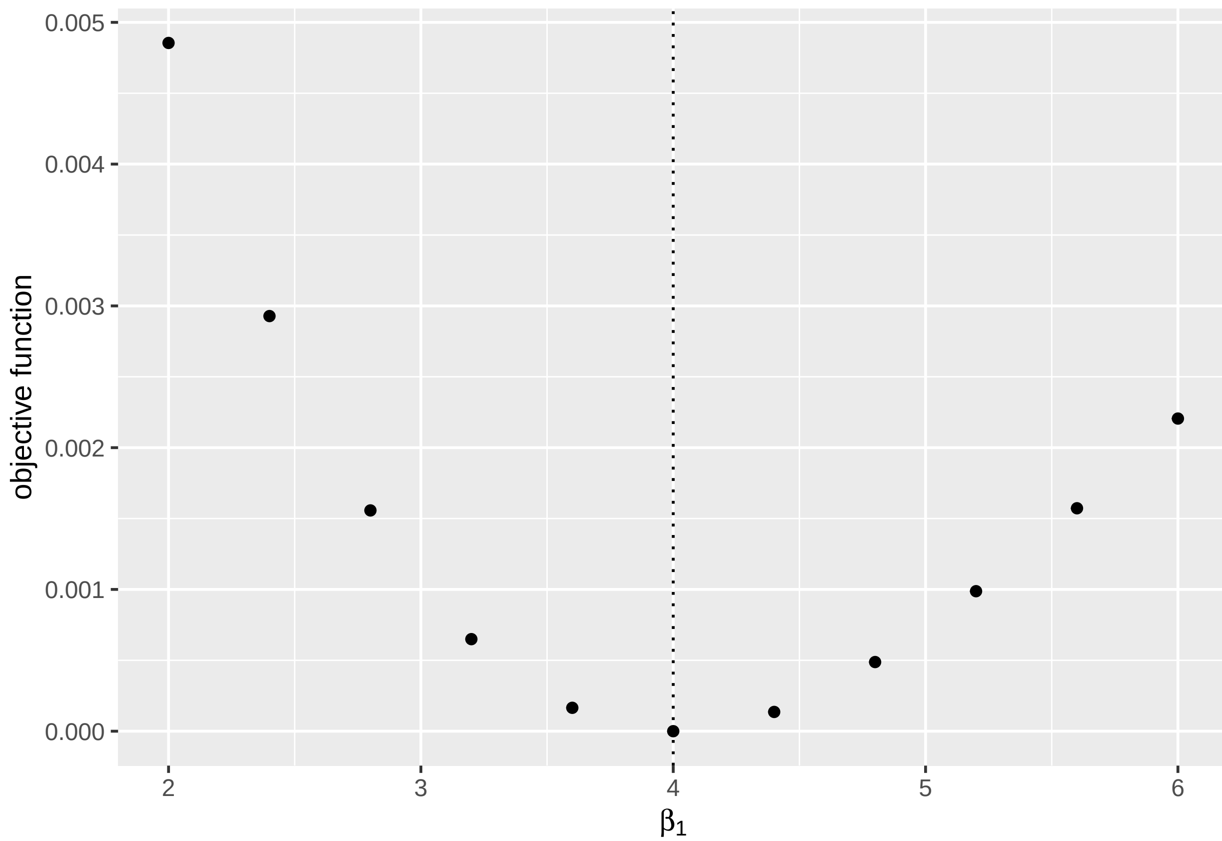
```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 6
##
## [[4]]
## [1] 8
```

The graphs with the true shocks:

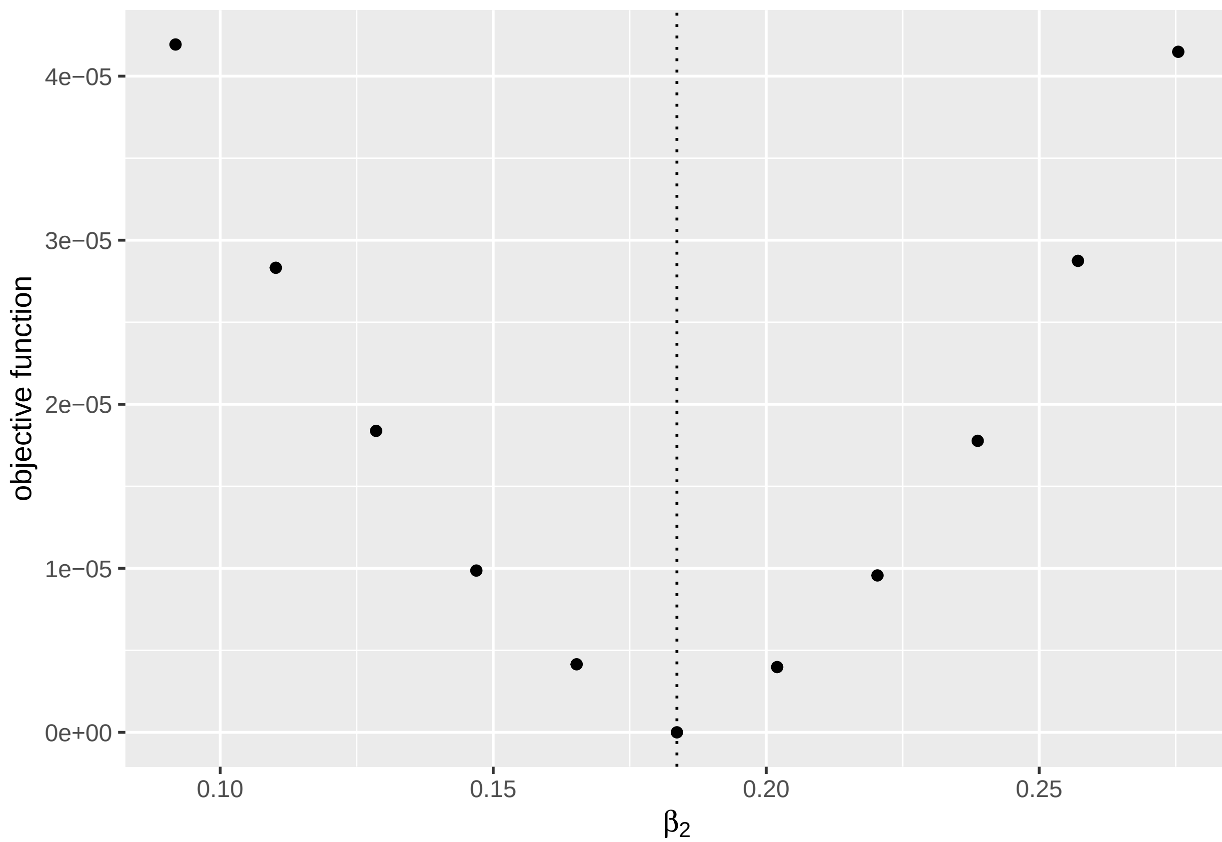
```r
label <- c(paste("\\beta_", 1:K, sep = ""),
           paste("\\sigma_", 1:K, sep = ""),
           "\\mu",
           "\\omega")
label <- paste("$", label, "$", sep = "")
graph_true <- foreach (i = 1:length(theta)) %do% {
  theta_i <- theta[i]
  theta_i_list <- theta_i * seq(0.5, 1.5, by = 0.1)
  objective_i <-
    foreach (theta_ij = theta_i_list,
             .combine = "rbind") %dopar% {
               theta_j <- theta
               theta_j[i] <- theta_ij
               objective_ij <-
                 NLLS_objective_A3(
                   theta_j, df_share, X, M, V, e)
               return(objective_ij)
             }
  df_graph <- data.frame(x = theta_i_list, y = objective_i)
  g <- ggplot(data = df_graph, aes(x = x, y = y)) +
    geom_point() +
    geom_vline(xintercept = theta_i, linetype = "dotted") +
    ylab("objective function") + xlab(TeX(label[i]))
  return(g)
}
save(graph_true, file = "data/A3_graph_true.RData")
```

```r
graph_true <- get(load(file = "data/A3_graph_true.RData"))
graph_true
```

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

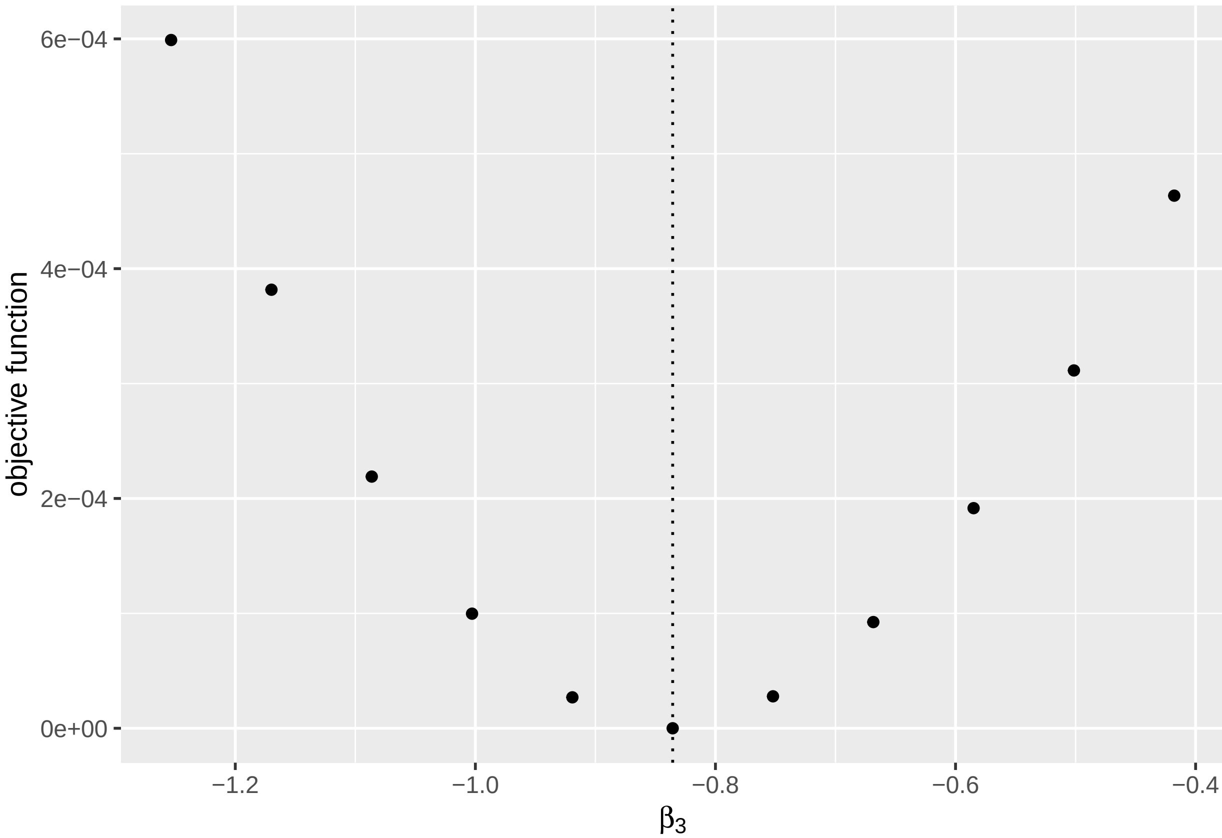



```
##
## [[2]]
```

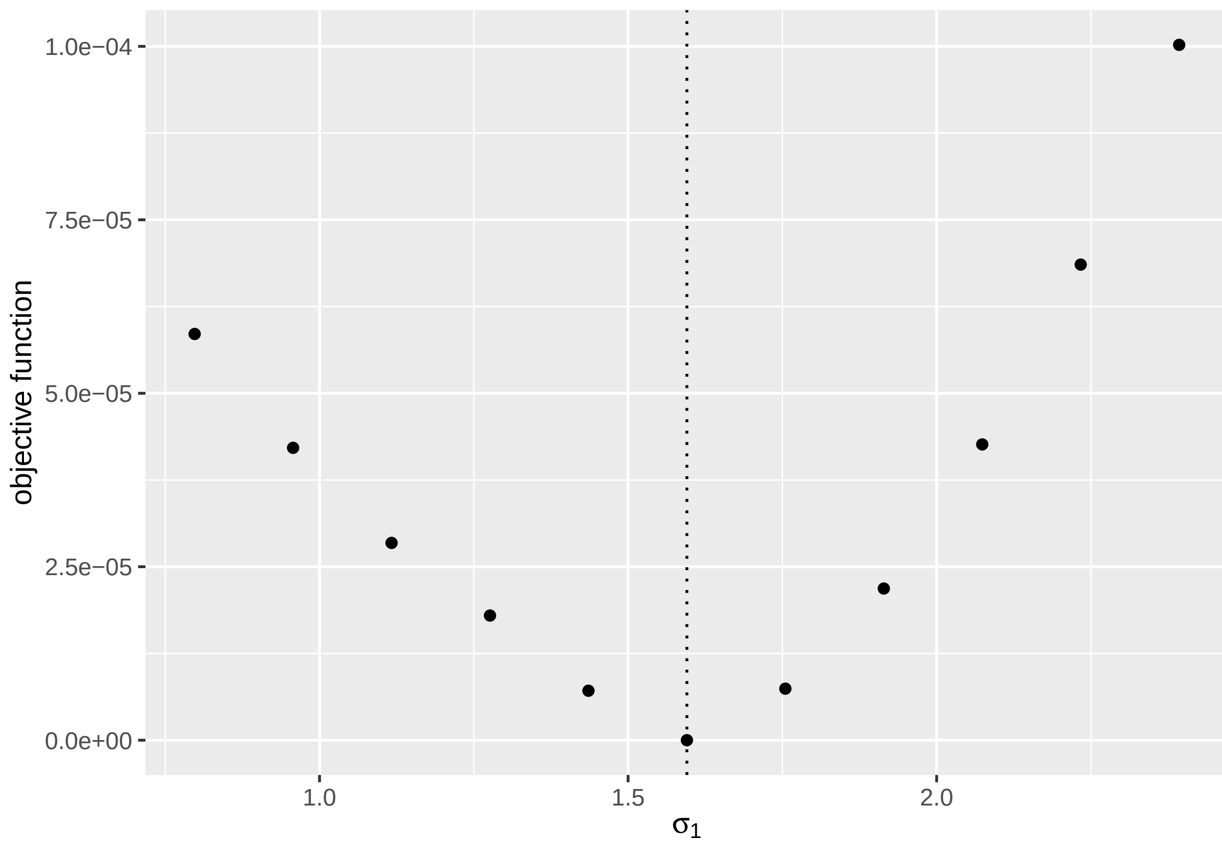


4e−05
3e−05
2e−05
1e−05
0e+00
objective function
0.10
0.15
0.20
0.25
β2

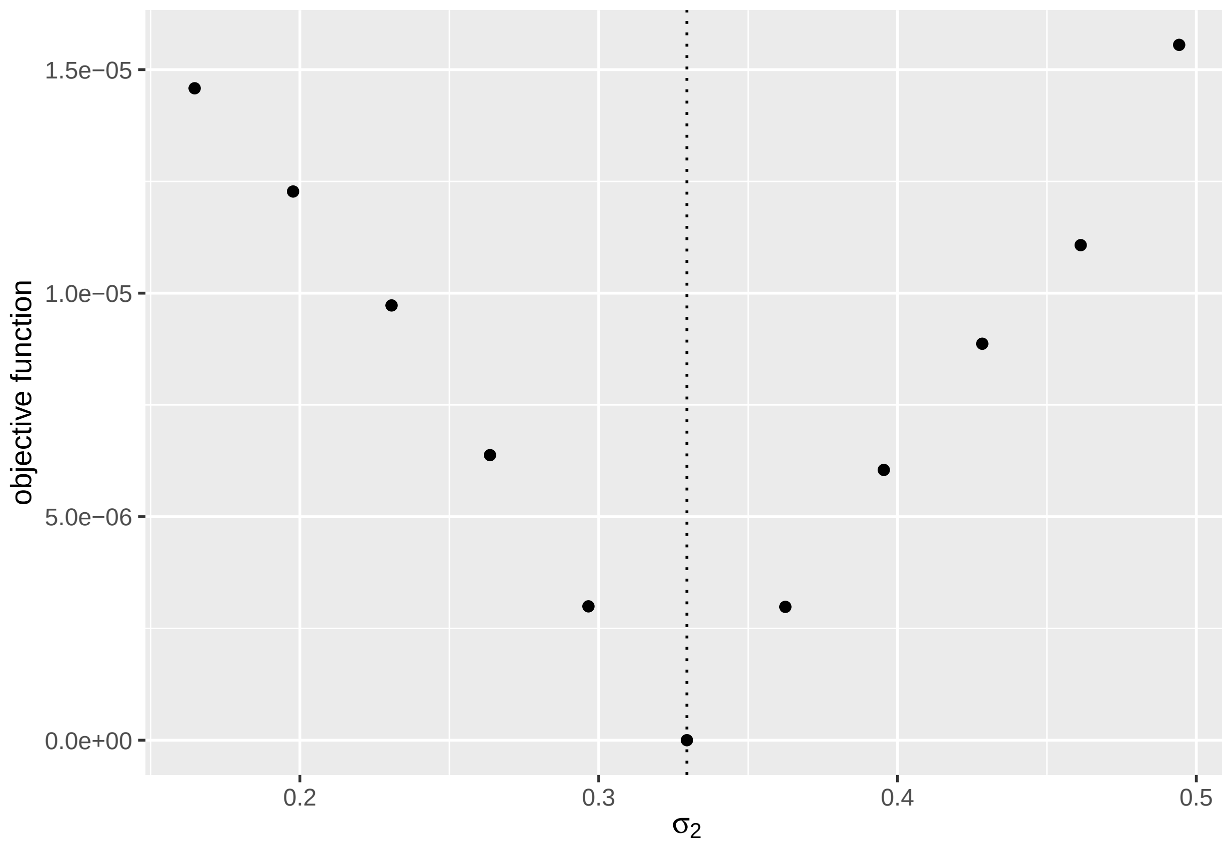

```
##
## [[3]]
```



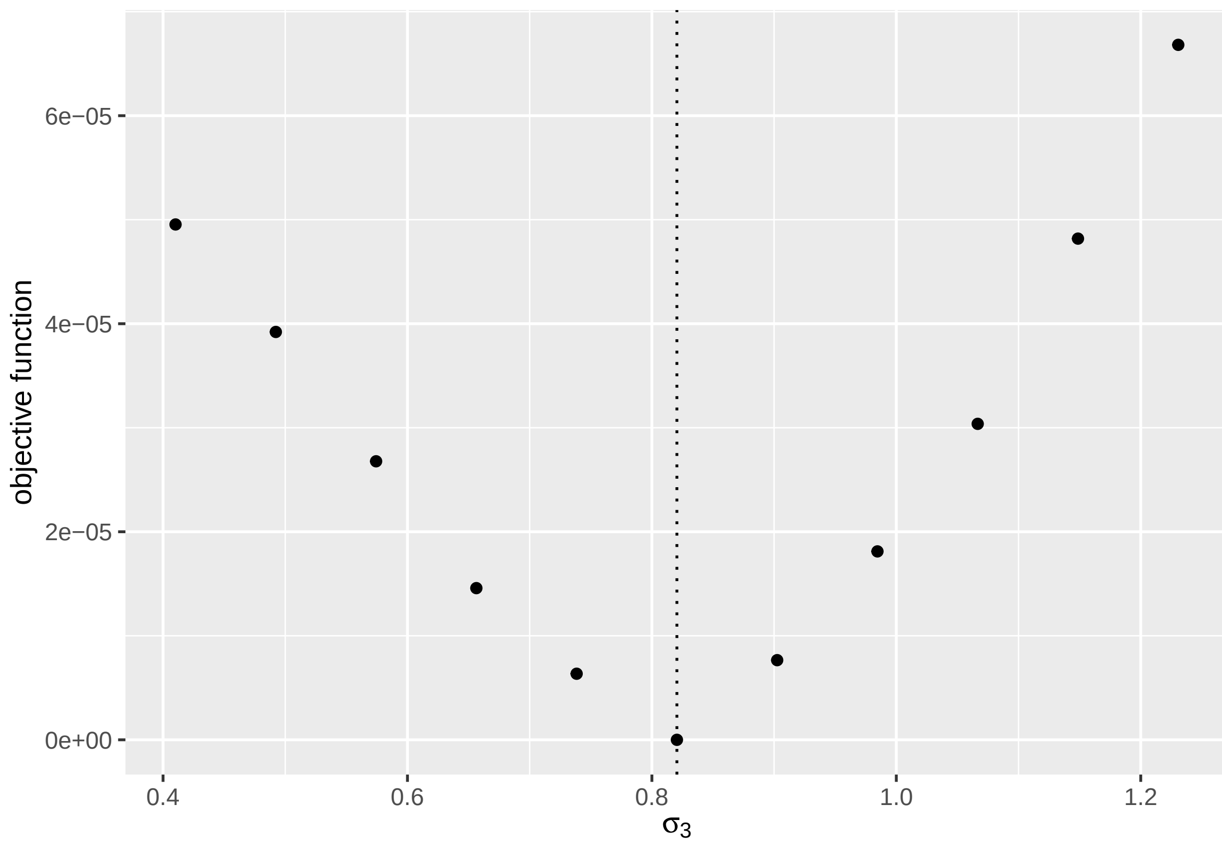

```
##
## [[4]]
```


1.0e−04
7.5e−05
5.0e−05
2.5e−05
0.0e+00
objective function
1.0
1.5
2.0
σ1

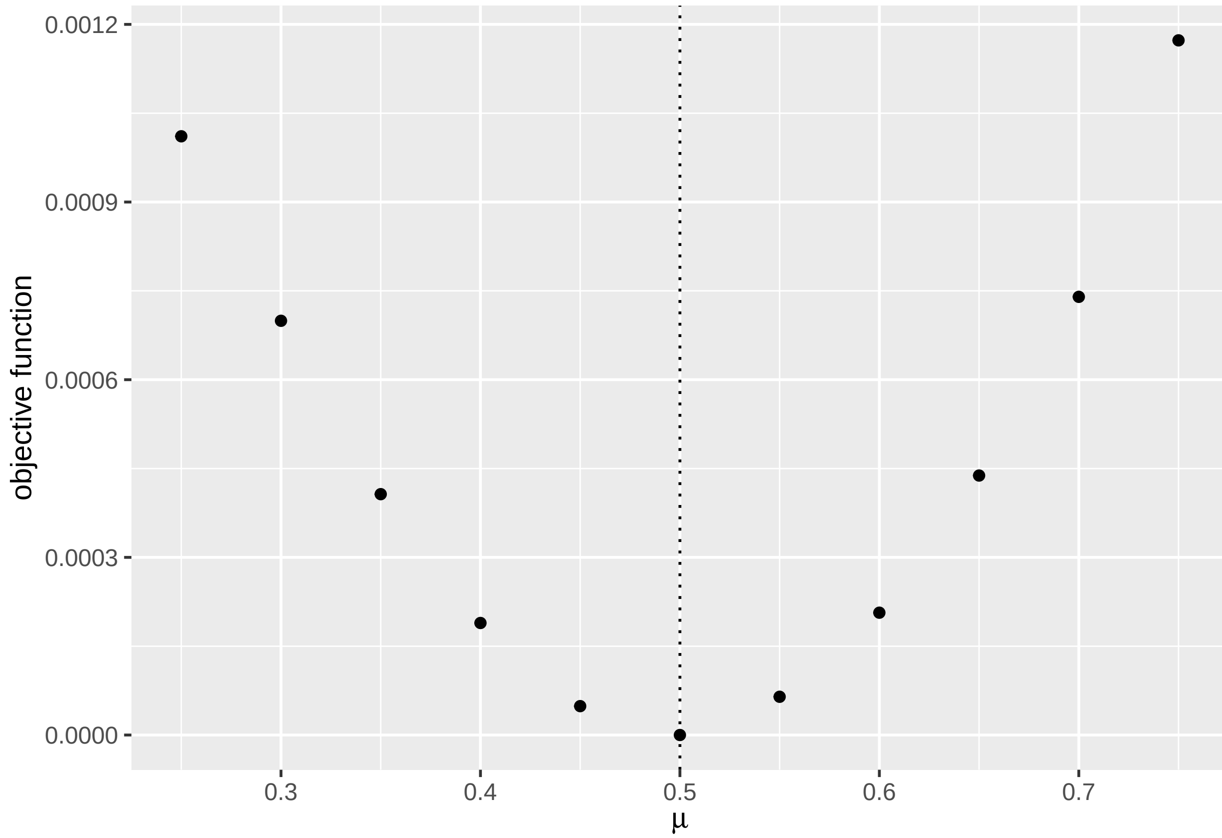
```
##
## [[5]]
```




```
##
## [[6]]
```

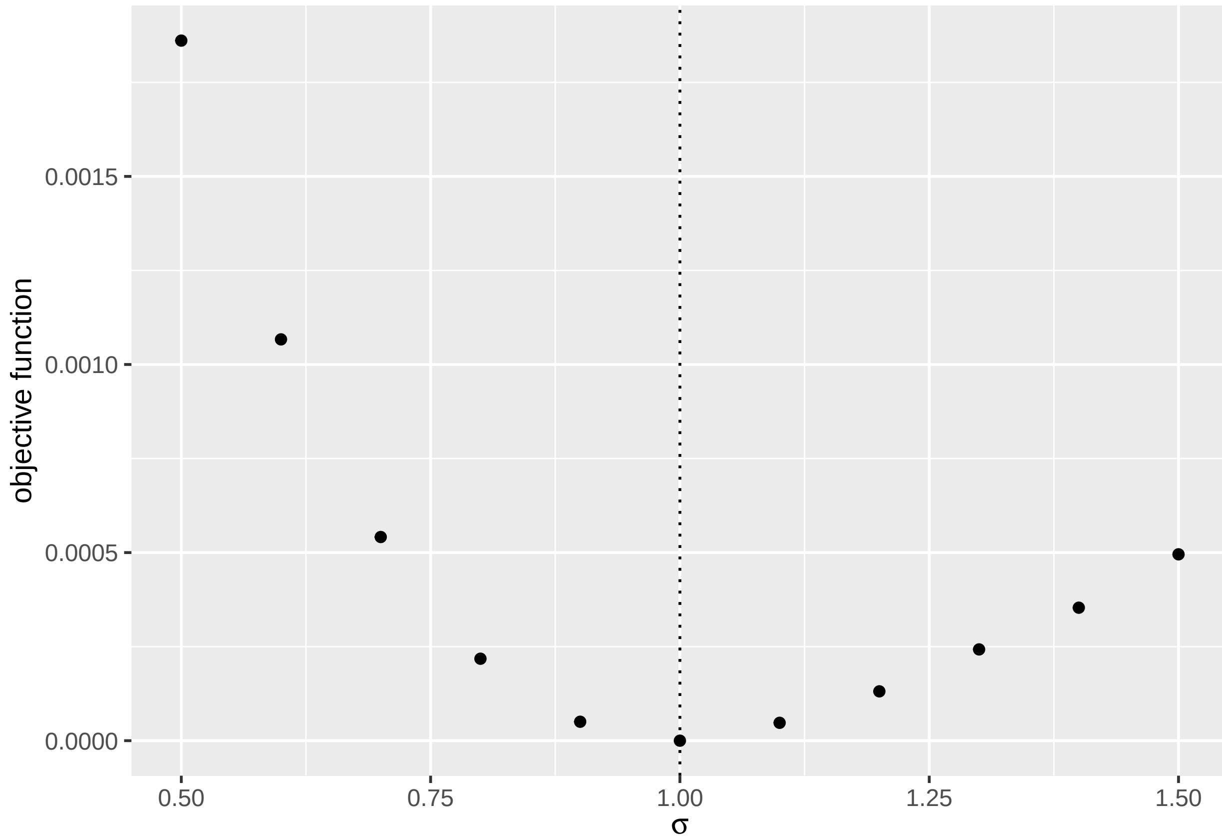



```
##
## [[7]]
```



0.0012
0.0009
0.0006
0.0003
0.0000
objective function
0.3
0.4
0.5
0.6
0.7
μ


```
##
## [[8]]
```



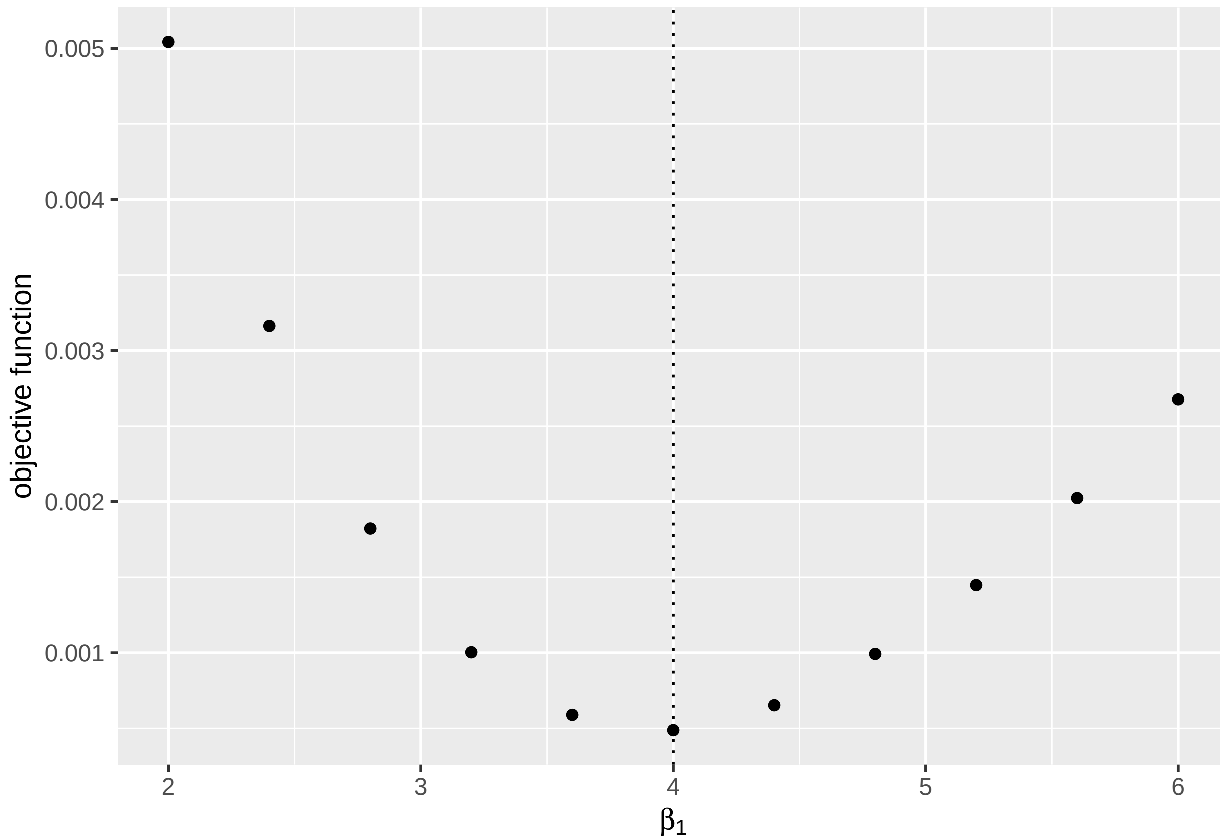
The graphs with the Monte Carlo shocks:

```
label <- c(paste("\\beta_", 1:K, sep = ""),
           paste("\\sigma_", 1:K, sep = ""),
           "\\mu",
           "\\omega")
label <- paste("$", label, "$", sep = "")
graph_mcmc <- foreach (i = 1:length(theta)) %do% {
  theta_i <- theta[i]
  theta_i_list <- theta_i * seq(0.5, 1.5, by = 0.1)
  objective_i <-
    foreach (theta_ij = theta_i_list,
             .combine = "rbind") %dopar% {
               theta_j <- theta
               theta_j[i] <- theta_ij
               objective_ij <-
                 NLLS_objective_A3(
                   theta_j, df_share, X, M, V_mcmc, e_mcmc)
               return(objective_ij)
             }
  df_graph <- data.frame(x = theta_i_list, y = objective_i)
  g <- ggplot(data = df_graph, aes(x = x, y = y)) +
    geom_point() +
    geom_vline(xintercept = theta_i, linetype = "dotted") +
    ylab("objective function") + xlab(TeX(label[i]))
  return(g)
}
```

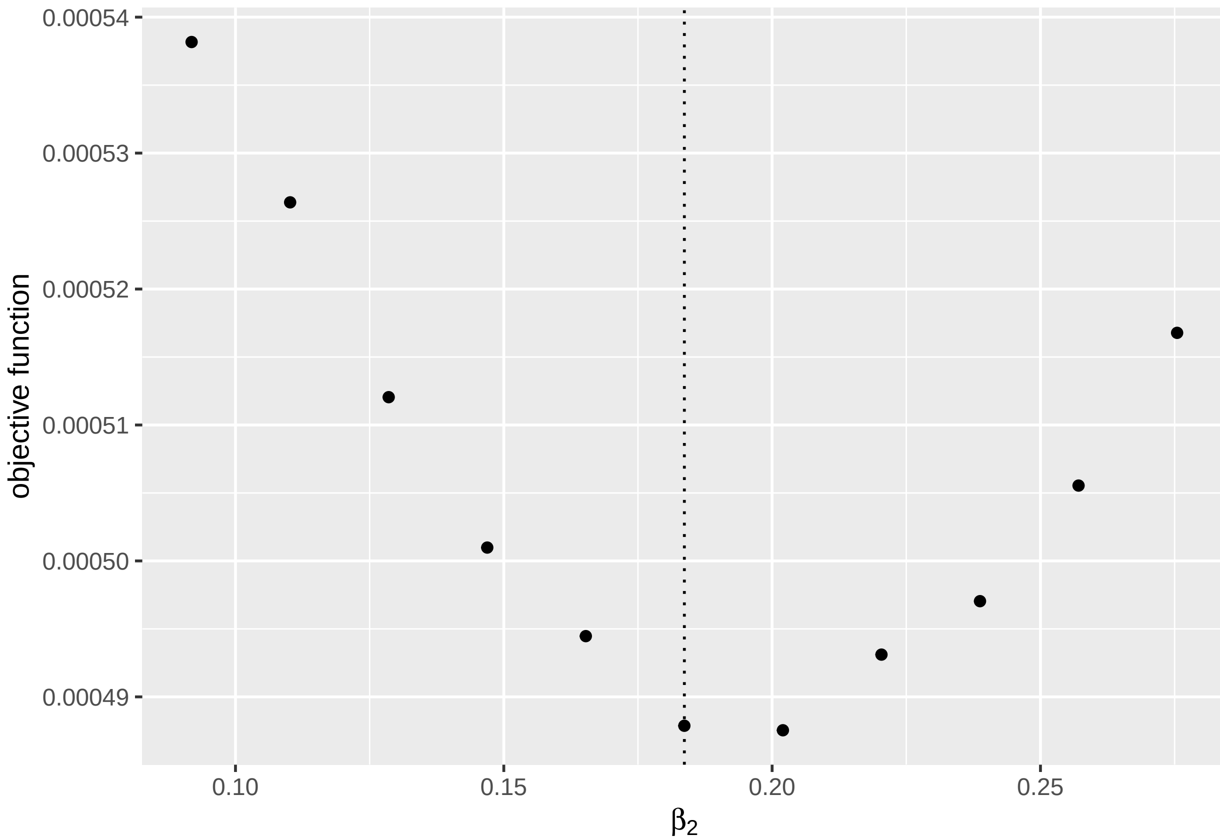
```r
save(graph_mcmc, file = "data/A3_graph_mcmc.RData")
```

```r
graph_mcmc <- get(load(file = "data/A3_graph_mcmc.RData"))
graph_mcmc
```

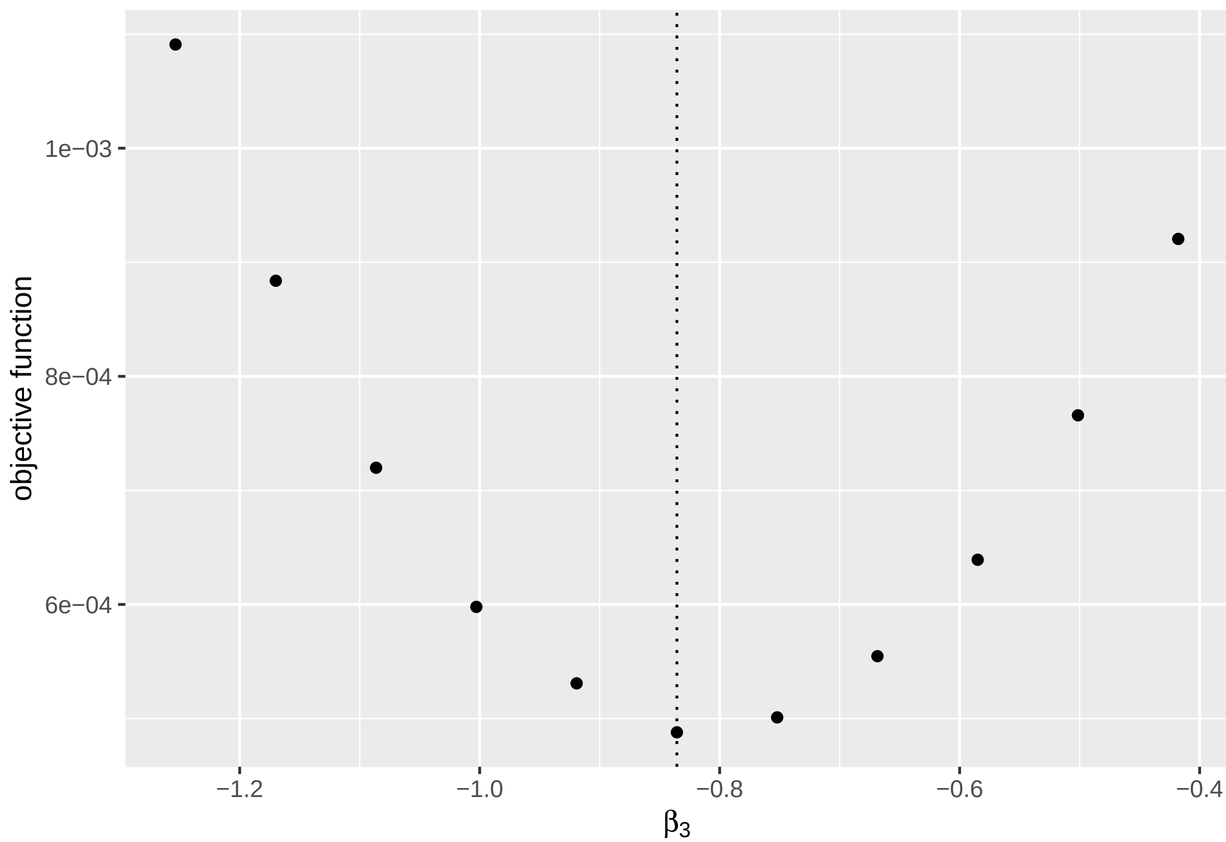
```
## [[1]]
```



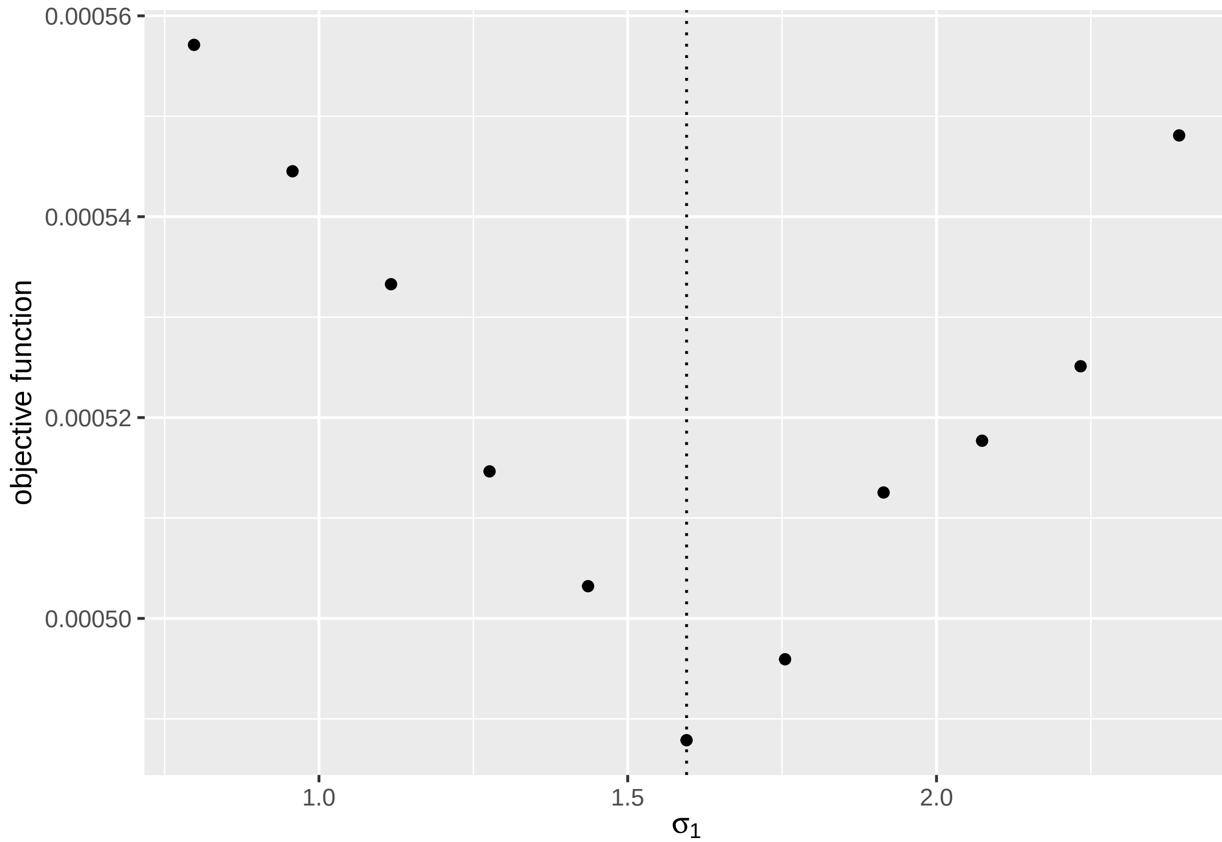

```
##
## [[2]]
```


0.00054
0.00053
0.00052
0.00051
0.00050
0.00049
objective function
0.10
0.15
0.20
0.25
β2

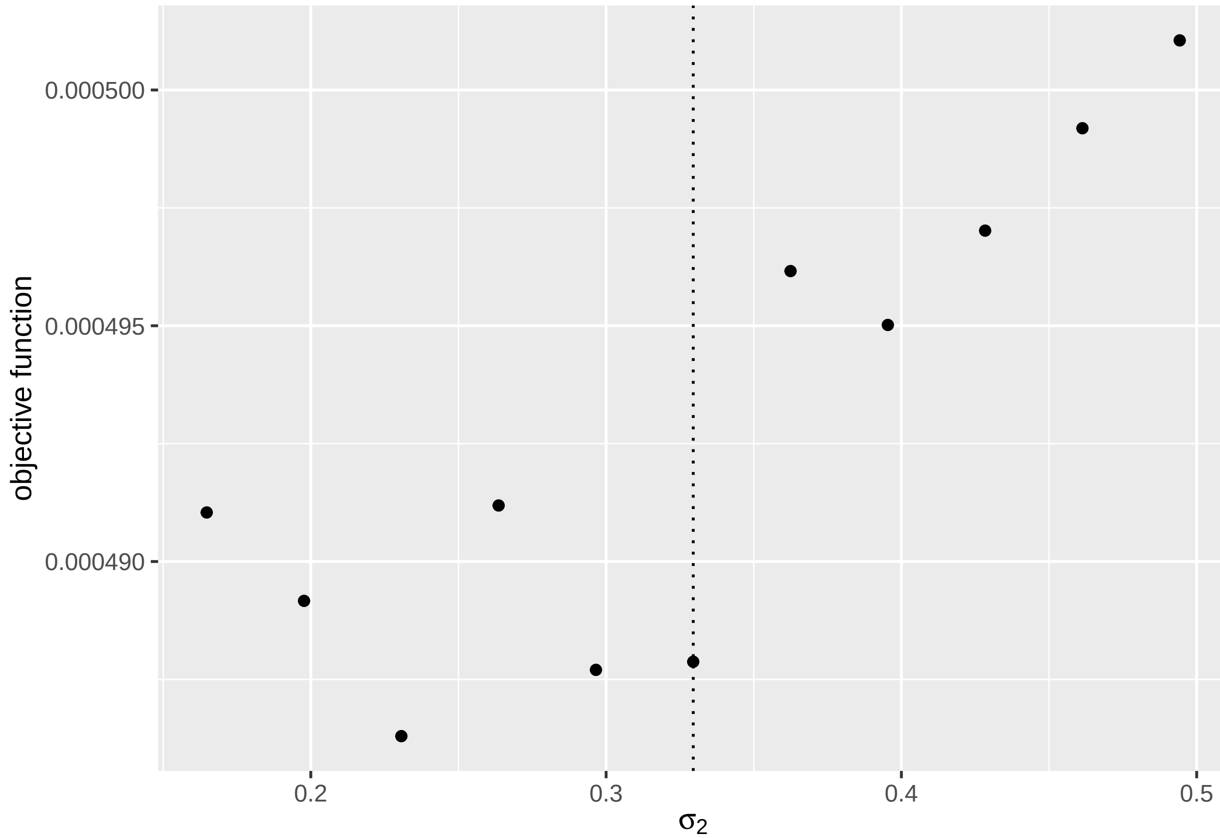
```
##
## [[3]]
```



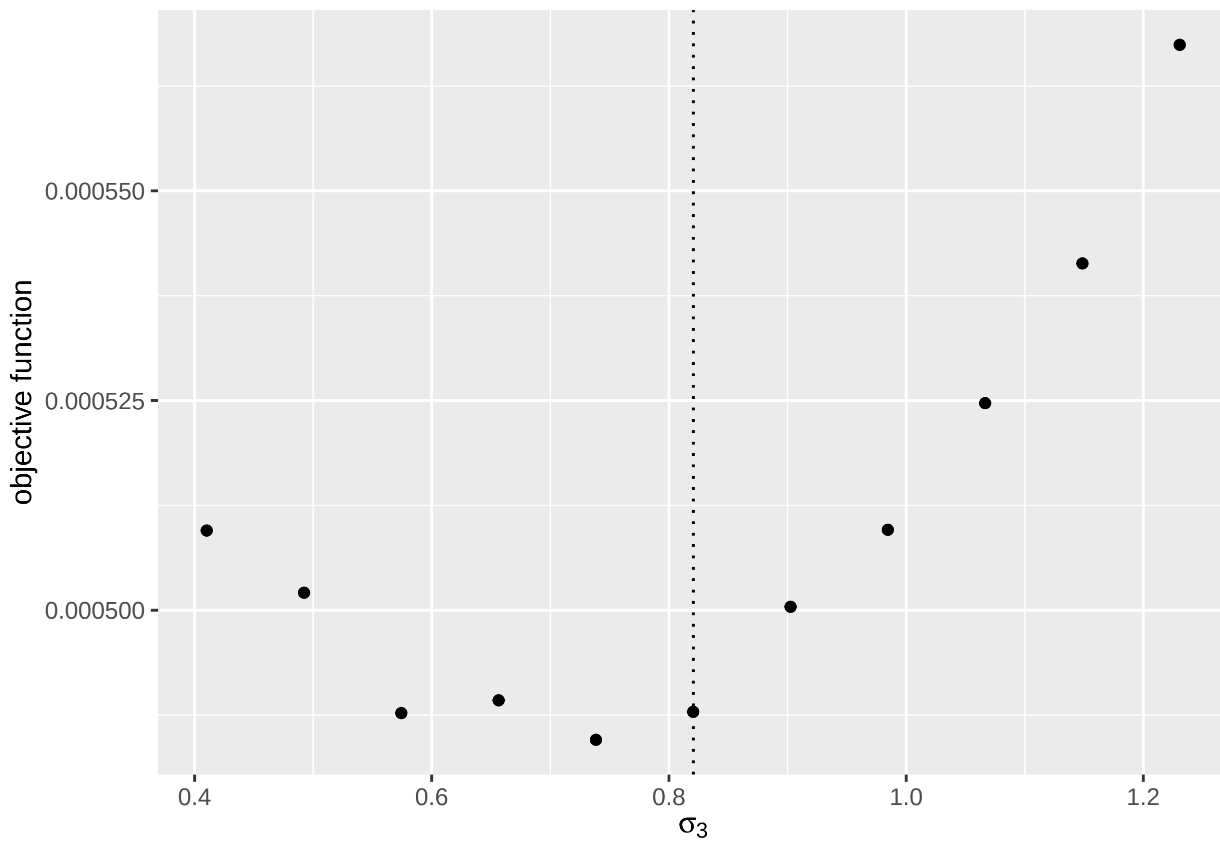

```
##
## [[4]]
```



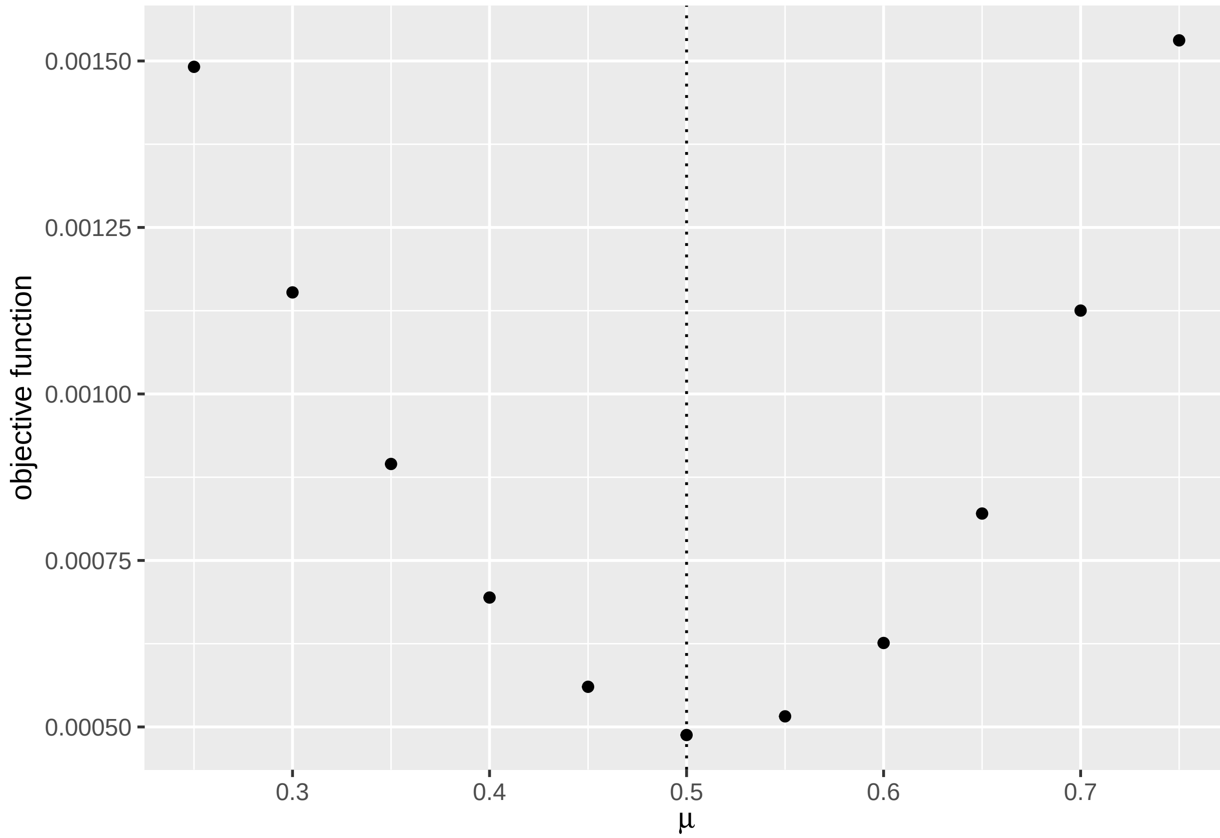

```
##
## [[5]]
```



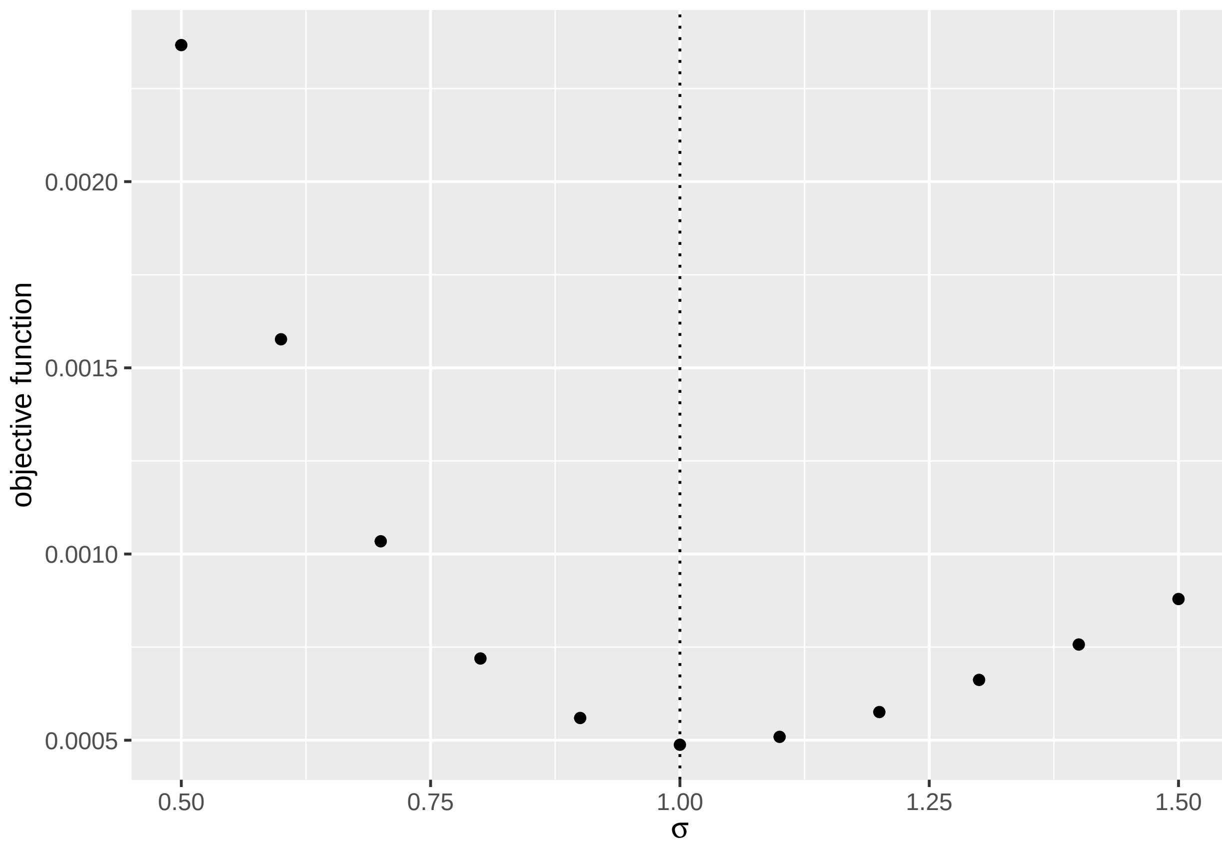

```
##
## [[6]]
```


objective function
0.000550
0.000525
0.000500
0.4
0.6
0.8
1.0
1.2
σ3

```
##
## [[7]]
```


0.00150
0.00125
0.00100
0.00075
0.00050
objective function
0.3
0.4
0.5
0.6
0.7
μ

```
##
## [[8]]
```



8. Use `optim` to find the minimizer of the objective function using `Nelder-Mead` method. You can start from the true parameter values. Compare the estimates with the true parameters.

```r
# find NLLS estimator
result_NLLS <-
  optim(par = theta, fn = NLLS_objective_A3,
        method = "Nelder-Mead",
        df_share = df_share,
        X = X,
        M = M,
        V_mcmc = V_mcmc,
        e_mcmc = e_mcmc)
save(result_NLLS, file = "data/A3_result_NLLS.RData")
```

```r
result_NLLS <- get(load(file = "data/A3_result_NLLS.RData"))
result_NLLS
```

```
## $par
## [1]  4.0425713  0.1841677 -0.8230489  1.6348574  0.3148886  0.7831262  0.4998710
## [8]  1.0138327
##
## $value
## [1] 0.0004760686
##
## $counts
## function gradient
##      263       NA
```

```
##
## $convergence
## [1] 0
##
## $message
## NULL
```

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

```
##         true  estimates
## 1  4.0000000  4.0425713
## 2  0.1836433  0.1841677
## 3 -0.8356286 -0.8230489
## 4  1.5952808  1.6348574
## 5  0.3295078  0.3148886
## 6  0.8204684  0.7831262
## 7  0.5000000  0.4998710
## 8  1.0000000  1.0138327
```