Assignment 5: Merger Simulation with Rcpp

Kohei Kawaguchi

2019/4/9

Simulate data

We simulate data from a discrete choice model that is the same with in assignment 4 **except for that the price is derived from the Nash equlibrium**. 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:

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

Suppose that we only observe the (smooth) share data:

$$s_{jt}(p_t, x_t, \xi_t) = \frac{1}{N} \sum_{i=1}^N \sigma_{ijt}(p_t, x_t, \xi_t) = \frac{1}{N} \sum_{i=1}^N \frac{\exp(u_{ijt})}{1 + \sum_{k \in \mathcal{J}_t \cup \{0\}} \exp(u_{ikt})}.$$

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 price is determined by a Nash equilibrium. Let Δ_t be the $J_t \times J_t$ ownership matrix in which the (j, k)-th element δ_{tjk} is equal to 1 if product j and k are owned by the same firm and 0 otherwise. Assume that $\delta_{tjk} = 1$ if and only if $j = k$ for all $t = 1, \cdots, T$, i.e., each firm owns only one product. Next, define Ω_t be $J_t \times J_t$ matrix such that whose (j, k)-the element $\omega_{tjk}(p_t, x_t, \xi_t, \Delta_t)$ is:

$$\omega_{tjk}(p_t, x_t, \xi_t, \Delta_t) = -\frac{\partial s_{jt}(p_t, x_t, \xi_t)}{\partial p_{kt}} \delta_{tjk}.$$

Then, the equilibrium price vector p_t is determined by solving the following equilibrium condition:

$$p_t = c_t + \Omega_t(p_t, x_t, \xi_t, \Delta_t)^{-1} s_t(p_t, x_t, \xi_t).$$

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.

```r
# 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
)
```

```r
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}. For now, set $p_{jt} = 0$ and fill the equilibrium price later. After generating the variables, drop some products in each market. 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.

```r
# 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 = 0
```

```r
  )
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)
```

```r
M
```

```
## # A tibble: 689 x 5
##        j     t      xi     c     p
##    <dbl> <int>   <dbl> <dbl> <dbl>
##  1     0     1  0      0         0
##  2     1     1 -0.0779 0.951     0
##  3     2     1 -0.735  1.04      0
##  4     7     1  0.194  0.961     0
##  5    10     1 -0.207  1.02      0
##  6     0     2  0      0         0
##  7     8     2  0.278  0.955     0
##  8     0     3  0      0         0
##  9     3     3 -0.0562 1.02      0
## 10     0     4  0      0         0
## # ... with 679 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  0.559  -0.362   -0.707  0.594
##  2     2     1 -1.00   -0.306    0.324 -0.368
##  3     3     1  0.900   0.464    0.253 -0.994
##  4     4     1  0.152  -0.640   -0.622 -0.290
##  5     5     1 -0.301  -2.18     0.151  0.475
##  6     6     1  0.0512 -1.05     0.430  0.159
##  7     7     1  0.292   0.00469  1.29   0.761
##  8     8     1  0.245  -0.330   -0.420 -0.00911
##  9     9     1  0.0827 -0.00644 -1.59  -1.02
```

```
## 10    10      1 -0.0404 -0.0764   0.259 -0.911
## # ... with 49,990 more rows
```

5. We use `compute_indirect_utility(df, beta, sigma, mu, omega)`, `compute_choice_smooth(X, M, V, beta, sigma, mu, omega)`, and `compute_share_smooth(X, M, V, beta, sigma, mu, omega)` to compute $s_t(p_t, x_t, \xi_t)$. On top of this, we need a function `compute_derivative_share_smooth(X, M, V, beta, sigma, mu, omega)` that approximate:

$$\frac{\partial s_{jt}(p_t, x_t, \xi_t)}{\partial p_{kt}} = \begin{cases} \frac{1}{N}\sum_{i=1}^N \alpha_i \sigma_{ijt}(p_t, x_t, \xi_t)[1 - \sigma_{ijt}(p_t, x_t, \xi_t)] & \text{if } j = k \\ -\frac{1}{N}\sum_{i=1}^N \alpha_i \sigma_{ijt}(p_t, x_t, \xi_t)\sigma_{ikt}(p_t, x_t, \xi_t)] & \text{if } j \neq k. \end{cases}$$

The returned object should be a list across markets and each element of the list should be $J_t \times J_t$ matrix whose (j,k)-th element is $\partial s_{jt}/\partial p_{it}$ (do not include the outside option). The computation will be looped across markets. I recommend to use a parallel computing for this loop.

Now I rewrite `compute_indirect_utility`, `compute_choice_smooth`, `compute_derivative_share_smooth`, `update_price` in C++ using Rcpp and Eigen. However, some of these functions use R dataframes. Because it is not easy to handle dataframes in C++, we first rewrite these function so that work only with R matrices. I recommend to transform data into a list of matrices such that each element of the list represents information about a decision maker.

```r
# 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)
# make list of matrices
J_start <- 1
df_list <-
  foreach (tt = 1:T) %do% {
    df_ti <-
      df %>%
      dplyr::filter(t == tt, i == 1)
    J_t <- dim(df_ti)[1] - 1
    J_end <- J_start + J_t - 1
    df_list_t <-
      foreach (ii = 1:N) %dopar% {
        df_ti <-
          df %>%
          dplyr::filter(t == tt, i == ii)
        XX <- as.matrix(dplyr::select(df_ti, dplyr::starts_with("x_")))
        p <- as.matrix(dplyr::select(df_ti, p))
        v_x <- as.matrix(dplyr::select(df_ti, dplyr::starts_with("v_x")))
        v_p <- as.matrix(dplyr::select(df_ti, v_p))
        xi <- as.matrix(dplyr::select(df_ti, xi))
        c <- as.matrix(dplyr::select(df_ti, c))
        df_list_ti <-
          list(
```

```r
            XX = XX,
            p = p,
            v_x = v_x,
            v_p = v_p,
            xi = xi,
            c = c,
            j = seq(J_start, J_end)
          )
        return(df_list_ti)
      }
    J_start <- J_end + 1
    return(df_list_t)
  }
```

```r
# compute indirect utility
compute_indirect_utility_matrix <-
  function(df_list, beta, sigma, mu, omega) {
    value <-
      foreach (t = 1:length(df_list)) %dopar% {
        value_t <-
          foreach (i = 1:length(df_list[[t]])) %do% {
            # extrast matrices
            XX <- df_list[[t]][[i]]$XX
            p <- df_list[[t]][[i]]$p
            v_x <- df_list[[t]][[i]]$v_x
            v_p <- df_list[[t]][[i]]$v_p
            xi <- df_list[[t]][[i]]$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_ti <- as.matrix(rowSums(beta_i * XX) + p * alpha_i + xi)
            colnames(value_ti) <- "u"
            # return
            return(value_ti)
        }
        return(value_t)
      }
    return(value)
  }
# check
u_1 <- compute_indirect_utility(df, beta, sigma, mu, omega)
u_2 <- compute_indirect_utility_matrix(df_list, beta, sigma, mu, omega)
u_2 <- u_2 %>%
  unlist() %>%
  as.matrix()
max(abs(u_1 - u_2))
```

```
## [1] 0
```

```cpp
// src/A5_functions.cpp
// compute indirect utility
// [[Rcpp::export]]
Rcpp::List compute_indirect_utility_matrix_rcpp(
  Rcpp::List df_list,
```

```
  Eigen::VectorXd beta,
  Eigen::VectorXd sigma,
  double mu,
  double omega
) {
  Rcpp::List value;
  for (int t = 0; t < df_list.size(); t++) {
    Rcpp::List df_list_t = df_list[t];
    Rcpp::List value_t;
    for (int i = 0; i < df_list_t.size(); i++) {
      // extrast matrices
      Rcpp::List df_list_ti = df_list_t[i];
      Eigen::MatrixXd XX(Rcpp::as<Eigen::MatrixXd>(df_list_ti["XX"]));
      Eigen::MatrixXd p(Rcpp::as<Eigen::MatrixXd>(df_list_ti["p"]));
      Eigen::MatrixXd v_x(Rcpp::as<Eigen::MatrixXd>(df_list_ti["v_x"]));
      Eigen::MatrixXd v_p(Rcpp::as<Eigen::MatrixXd>(df_list_ti["v_p"]));
      Eigen::MatrixXd xi(Rcpp::as<Eigen::MatrixXd>(df_list_ti["xi"]));
      // random coefficients
      Eigen::MatrixXd beta_i = v_x * sigma.asDiagonal();
      beta_i = beta_i.rowwise() + beta.transpose();
      Eigen::MatrixXd alpha_i = - (mu + (omega * v_p).array()).exp();
      // conditional mean indirect utility
      Eigen::MatrixXd temp_1 = beta_i.array() * XX.array();
      Eigen::MatrixXd value = temp_1.rowwise().sum();
      value = value + (p.array() * alpha_i.array()).matrix() + xi;
      // return
      value_t.push_back(value);
    }
    // return
    value.push_back(value_t);
  }
  // return
  return(value);
}
```

```r
# check Rcpp function
u_3 <- compute_indirect_utility_matrix_rcpp(df_list, beta, sigma, mu, omega)
u_3 <- u_3 %>%
  unlist() %>%
  as.matrix()
max(abs(u_2 - u_3))
```

```
## [1] 3.552714e-15
```

```r
# compute choice
compute_choice_smooth_matrix <-
  function(df_list, beta, sigma,
           mu, omega) {
    # compute indirect utility
    u <- compute_indirect_utility_matrix(
      df_list, beta, sigma, mu, omega)
    # make choice
    q <-
      foreach (t = 1:length(u)) %dopar% {
        q_t <-
```

```r
          foreach (i = 1:length(u[[t]])) %do% {
            u_ti <- u[[t]][[i]]
            q_ti <- exp(u_ti) / sum(exp(u_ti))
            return(q_ti)
          }
        return(q_t)
      }
    # return
    return(q)
  }
# check
q_1 <- compute_choice_smooth(
  X, M, V, beta, sigma, mu, omega)
q_1 <- q_1$q
q_2 <- compute_choice_smooth_matrix(
  df_list, beta, sigma, mu, omega)
q_2 <- unlist(q_2)
max(abs(q_1 - q_2))
```

```
## [1] 0
```

```cpp
// src/A5_functions.cpp
// compute choice
// [[Rcpp::export]]
Rcpp::List compute_choice_smooth_matrix_rcpp(
    Rcpp::List df_list,
    Eigen::VectorXd beta,
    Eigen::VectorXd sigma,
    double mu,
    double omega
) {
  // compute indirect utility
  Rcpp::List u = compute_indirect_utility_matrix_rcpp(
    df_list, beta, sigma, mu, omega);
  // make choice
  Rcpp::List q;
  for (int t = 0; t < u.size(); t++) {
    Rcpp::List u_t = u[t];
    Rcpp::List q_t;
    for (int i = 0; i < u_t.size(); i++) {
      Eigen::MatrixXd u_ti(Rcpp::as<Eigen::MatrixXd>(u_t[i]));
      Eigen::MatrixXd q_ti = u_ti.array().exp();
      q_ti = q_ti / q_ti.sum();
      q_t.push_back(q_ti);
    }
    q.push_back(q_t);
  }
  // return
  return(q);
}
```

```r
q_3 <- compute_choice_smooth_matrix_rcpp(
  df_list, beta, sigma, mu, omega)
q_3 <- unlist(q_3)
max(abs(q_2 - q_3))
```

```
## [1] 9.992007e-16
```

```r
# compute share
compute_share_smooth_matrix <-
  function(df_list, beta, sigma,
           mu, omega) {
    # compute choice
    df_choice <-
      compute_choice_smooth_matrix(df_list, beta, sigma,
                                   mu, omega)
    # make share data
    df_share_smooth <-
      foreach (t = 1:length(df_choice)) %dopar% {
        q_t <- df_choice[[t]]
        q_t <- q_t %>%
          purrr::reduce(`+`)
        s_t <- q_t / sum(q_t)
        return(s_t)
      }

    return(df_share_smooth)
  }
s_1 <- compute_share_smooth(
  X, M, V, beta, sigma, mu, omega)
s_1 <- s_1$s
s_2 <- compute_share_smooth_matrix(
  df_list, beta, sigma, mu, omega)
s_2 <- unlist(s_2)
max(abs(s_1 - s_2))
```

```
## [1] 4.440892e-16
```

```cpp
// src/A5_functions.cpp
// compute share
// [[Rcpp::export]]
Rcpp::List compute_share_smooth_matrix_rcpp(
    Rcpp::List df_list,
    Eigen::VectorXd beta,
    Eigen::VectorXd sigma,
    double mu,
    double omega
) {
  // compute choice
  Rcpp::List df_choice = compute_choice_smooth_matrix_rcpp(
    df_list, beta, sigma, mu, omega);
  // make share data
  Rcpp::List df_share_smooth;
  for (int t = 0; t < df_choice.size(); t++) {
    Rcpp::List q_t = df_choice[t];
    Eigen::MatrixXd s_t(Rcpp::as<Eigen::MatrixXd>(q_t[0]));
    for (int i = 1; i < q_t.size(); i++) {
      Eigen::MatrixXd s_ti(Rcpp::as<Eigen::MatrixXd>(q_t[i]));
      s_t = s_t + s_ti;
    }
```

```
    s_t = s_t / q_t.size();
    s_t = s_t / s_t.sum();
    df_share_smooth.push_back(s_t);
  }
  return(df_share_smooth);
}
```

```r
s_3 <- compute_share_smooth_matrix_rcpp(
  df_list, beta, sigma, mu, omega)
s_3 <- unlist(s_3)
max(abs(s_2 - s_3))
```

```
## [1] 2.220446e-16
```

```r
# compute the derivatives of the smooth share
compute_derivative_share_smooth_matrix <-
  function(df_list, beta, sigma, mu, omega) {
    q <- compute_choice_smooth_matrix(df_list, beta, sigma, mu, omega)
    derivative_choice_smooth <-
      foreach (t = 1:length(q)) %dopar% {
        # extract data for market t
        q_t <- q[[t]]
        # compute the derivative matrix for each market
        derivative_choice_smooth_t <-
          foreach (i = 1:length(q_t)) %do% {
            # extract data for consumer i
            q_ti <- q_t[[i]]
            # drop the outside option
            s_ti <- q_ti[2:length(q_ti)]
            # extract alpha_i
            v_pi <- df_list[[t]][[i]]$v_p
            alpha_i <- - exp(mu + omega * v_pi[1])
            # compute the derivatice matrix for each consumer
            ss_ti <- tcrossprod(s_ti, s_ti)
            if (length(s_ti) > 1) {
              derivative_choice_smooth_ti <-
                diag(s_ti) - ss_ti
            } else {
              derivative_choice_smooth_ti <-
                s_ti - ss_ti
            }
            derivative_choice_smooth_ti <-
              alpha_i * derivative_choice_smooth_ti
            # return
            return(derivative_choice_smooth_ti)
          }
        # take average
        N <- length(derivative_choice_smooth_t)
        derivative_choice_smooth_t <-
          derivative_choice_smooth_t %>%
          purrr::reduce(`+`)
        derivative_choice_smooth_t <-
          derivative_choice_smooth_t / N
        # return
        return(derivative_choice_smooth_t)
```

```r
      }
    # return
    return(derivative_choice_smooth)
  }
ds_1 <- compute_derivative_share_smooth(
  X, M, V, beta, sigma, mu, omega)
ds_2 <- compute_derivative_share_smooth_matrix(
  df_list, beta, sigma, mu, omega)
max(abs(unlist(ds_1) - unlist(ds_2)))
```

```
## [1] 0
```

```cpp
// src/A5_functions.cpp
// compute the derivatives of the smooth share
// [[Rcpp::export]]
Rcpp::List compute_derivative_share_smooth_matrix_rcpp(
    Rcpp::List df_list,
    Eigen::VectorXd beta,
    Eigen::VectorXd sigma,
    double mu,
    double omega
) {
  //     q <- compute_choice_smooth_matrix(df_list, beta, sigma, mu, omega)
  Rcpp::List q = compute_choice_smooth_matrix_rcpp(
    df_list, beta, sigma, mu, omega);
  Rcpp::List derivative_choice_smooth;
  for (int t = 0; t < q.size(); t++) {
    // extract data for market t
    Rcpp::List q_t = q[t];
    Rcpp::List df_list_t = df_list[t];
    // compute the derivative matrix for each market
    Eigen::VectorXd q_ti(Rcpp::as<Eigen::VectorXd>(q_t[0]));
    int J = q_ti.size() - 1;
    Eigen::MatrixXd derivative_choice_smooth_t =
      Eigen::MatrixXd::Zero(J, J);
    for (int i = 0; i < q_t.size(); i++) {
      // extract data for consumer i
      Eigen::VectorXd q_ti(Rcpp::as<Eigen::VectorXd>(q_t[i]));
      // drop the outside option
      Eigen::MatrixXd s_ti = q_ti.tail(J);
      // extract alpha_i
      Rcpp::List df_list_ti = df_list_t[i];
      Eigen::VectorXd v_pi(Rcpp::as<Eigen::VectorXd>(df_list_ti["v_p"]));
      double alpha_i = - (mu + omega * v_pi.array()).exp()(0);
      // compute the derivative matrix for each consumer
      Eigen::MatrixXd ss_ti = s_ti * s_ti.transpose();
      Eigen::MatrixXd derivative_choice_smooth_ti;
      if (J > 1) {
        derivative_choice_smooth_ti = s_ti.asDiagonal();
        derivative_choice_smooth_ti = derivative_choice_smooth_ti - ss_ti;
      } else {
        derivative_choice_smooth_ti = s_ti - ss_ti;
      }
      derivative_choice_smooth_ti = alpha_i * derivative_choice_smooth_ti;
```

```
      // add
      derivative_choice_smooth_t = derivative_choice_smooth_t +
        derivative_choice_smooth_ti;
    }
    derivative_choice_smooth_t = derivative_choice_smooth_t / q_t.size();
    // return
    derivative_choice_smooth.push_back(derivative_choice_smooth_t);
  }
  // return
  return(derivative_choice_smooth);
}
```

```
ds_3 <- compute_derivative_share_smooth_matrix_rcpp(
  df_list, beta, sigma, mu, omega)
max(abs(unlist(ds_2) - unlist(ds_3)))
```

```
## [1] 2.220446e-16
```

```
derivative_share_smooth <-
  compute_derivative_share_smooth_matrix_rcpp(df_list, beta, sigma, mu, omega)
derivative_share_smooth[[1]]
```

```
##             [,1]        [,2]        [,3]        [,4]
## [1,] -0.55323517  0.07416782  0.22222078  0.24032215
## [2,]  0.07416782 -0.17952618  0.05347946  0.04757626
## [3,]  0.22222078  0.05347946 -0.47677137  0.18862102
## [4,]  0.24032215  0.04757626  0.18862102 -0.48803506
```

```
derivative_share_smooth[[T]]
```

```
##             [,1]        [,2]        [,3]        [,4]        [,5]
## [1,] -0.07358769  0.01724997  0.01539716  0.00728682  0.01037598
## [2,]  0.01724997 -0.18464980  0.03941164  0.02590020  0.05210013
## [3,]  0.01539716  0.03941164 -0.14947032  0.01760727  0.02950338
## [4,]  0.00728682  0.02590020  0.01760727 -0.11933630  0.04443484
## [5,]  0.01037598  0.05210013  0.02950338  0.04443484 -0.18480392
## [6,]  0.02286198  0.04875629  0.04659132  0.02338740  0.04568463
##             [,6]
## [1,]  0.02286198
## [2,]  0.04875629
## [3,]  0.04659132
## [4,]  0.02338740
## [5,]  0.04568463
## [6,] -0.18893897
```

6. Make a list `Delta` such that each element of the list is $J_t \times J_t$ matrix Δ_t.

```
Delta <-
  foreach (tt = 1:T) %do% {
    J_t <- M %>%
      dplyr::filter(t == tt) %>%
      dplyr::filter(j > 0)
    J_t <- dim(J_t)[1]
    Delta_t <- diag(rep(1, J_t))
    return(Delta_t)
  }
```

```r
Delta[[1]]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

```r
Delta[[T]]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    0    0    0    0    0
## [2,]    0    1    0    0    0    0
## [3,]    0    0    1    0    0    0
## [4,]    0    0    0    1    0    0
## [5,]    0    0    0    0    1    0
## [6,]    0    0    0    0    0    1
```

7. Write a function `update_price(logp, X, M, V, beta, sigma, mu, omega, Delta)` that receives a price vector $p_t^{(r)}$ and returns $p_t^{(r+1)}$ by:

$$p_t^{(r+1)} = c_t + \Omega_t(p_t^{(r)}, x_t, \xi_t, \Delta_t)^{-1} s_t(p_t^{(r)}, x_t, \xi_t).$$

The returned object should be a vector whose row represents the condition for an inside product of each market. To impose non-negativity constraint on the price vector, we pass log price and exponentiate inside the function. Iterate this until $\max_{jt} |p_{jt}^{(r+1)} - p_{jt}^{(r)}| < \lambda$, for example with $\lambda = 10^{-6}$. This iteration may or may not converge. The convergence depends on the parameters and the realization of the shocks. If the algorithm does not converge, first check the code.

```r
# set the initial price
p <- M[M$j > 0, "p"]
logp <- log(rep(1, dim(p)[1]))
p_1 <- update_price(logp, X, M, V, beta, sigma, mu, omega, Delta)
p_2 <- update_price_matrix(logp, df_list, beta, sigma, mu, omega, Delta)
p_2 <- purrr::reduce(p_2, rbind)
max(abs(p_1 - p_2))
```

```
## [1] 2.664535e-15
```

```cpp
// src/A5_functions.cpp
// evaluate the equilibrium condition
// [[Rcpp::export]]
Rcpp::List update_price_matrix_rcpp(
  Eigen::VectorXd logp,
  Rcpp::List df_list,
  Eigen::VectorXd beta,
  Eigen::VectorXd sigma,
  double mu,
  double omega,
  Rcpp::List Delta
) {
  // exponentiate
  Eigen::VectorXd p = logp.array().exp();
  // replace price
  for (int t = 0; t < df_list.size(); t++) {
    Rcpp::List df_list_t = df_list[t];
    for (int i = 0; i < df_list_t.size(); i++) {
```

```
      Rcpp::List df_list_ti = df_list_t[i];
      Eigen::ArrayXi j(Rcpp::as<Eigen::ArrayXi>(df_list_ti["j"]));
      j = j - 1;
      int start = j(0);
      Eigen::VectorXd p_j = p.segment(start, j.size());
      Eigen::MatrixXd p_j0 = Eigen::VectorXd::Zero(p_j.size() + 1, 1);
      p_j0.block(1, 0, p_j.size(), 1) = p_j;
      df_list_ti["p"] = Rcpp::wrap(p_j0);
      df_list_t[i] = Rcpp::wrap(df_list_ti);
    }
    df_list[t] = Rcpp::wrap(df_list_t);
  }
  Rcpp::List p_new;
  // compute the share and the derivative
  Rcpp::List share = compute_share_smooth_matrix_rcpp(df_list, beta, sigma, mu, omega);
  Rcpp::List ds = compute_derivative_share_smooth_matrix_rcpp(df_list, beta, sigma, mu, omega);
  // evaluate equilibrium condition
  for (int t = 0; t < ds.size(); t++) {
    // extract
    Rcpp::List df_list_t = df_list[t];
    Rcpp::List df_list_ti = df_list_t[0];
    Eigen::VectorXd s_t(Rcpp::as<Eigen::VectorXd>(share[t]));
    Eigen::VectorXd c_t(Rcpp::as<Eigen::VectorXd>(df_list_ti["c"]));
    Eigen::VectorXd p_t(Rcpp::as<Eigen::VectorXd>(df_list_ti["p"]));
    Eigen::VectorXd s_t0 = s_t.segment(1, s_t.size() - 1);
    Eigen::VectorXd c_t0 = c_t.segment(1, c_t.size() - 1);
    Eigen::VectorXd p_t0 = p_t.segment(1, p_t.size() - 1);
    // make Omega in market t
    Eigen::MatrixXd Delta_t(Rcpp::as<Eigen::MatrixXd>(Delta[t]));
    Eigen::MatrixXd ds_t(Rcpp::as<Eigen::MatrixXd>(ds[t]));
    Eigen::MatrixXd Omega_t = - Delta_t.array() * ds_t.array();
    // markup
    Eigen::VectorXd markup_t = Omega_t.colPivHouseholderQr().solve(s_t0);
    // equilibrium condition
    Eigen::MatrixXd p_new_t = c_t0 + markup_t;
    // return
    p_new.push_back(p_new_t);
  }
  // return
  return(p_new);
}
```

```r
p_3 <- update_price_matrix_rcpp(logp, df_list, beta, sigma, mu, omega, Delta)
p_3 <- purrr::reduce(p_3, rbind)
max(abs(p_2 - p_3))
```

```
## [1] 1.776357e-15
```

```r
# set the threshold
lambda <- 1e-6
# set the initial price
p <- M[M$j > 0, "p"]
logp <- log(rep(1, dim(p)[1]))
p_new <- update_price_matrix_rcpp(logp, df_list, beta, sigma, mu, omega, Delta)
p_new <- purrr::reduce(p_new, rbind)
```

```r
# iterate
distance <- 10000
while (distance > lambda) {
  p_old <- p_new
  p_new <- update_price_matrix_rcpp(log(p_old), df_list, beta, sigma, mu, omega, Delta)
  p_new <- purrr::reduce(p_new, rbind)
  distance <- max(abs(p_new - p_old))
  print(distance)
}
# save
p_actual <- p_new
save(p_actual, file = "data/A5_price_actual_rcpp.RData")
```