Online Appendix

Figure OA1: Sweep of True Demand Idiosyncratic Demand Parameter with β_1 *=0, -2.5, -5, and -7.5 and initial prior* $b_1 = b_2 = \beta_1$

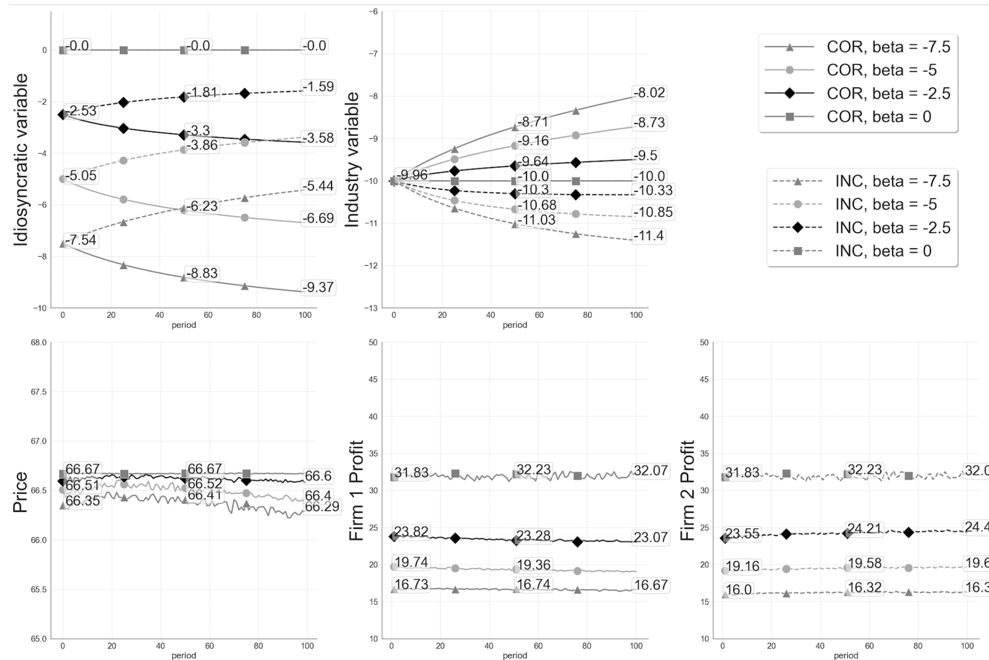
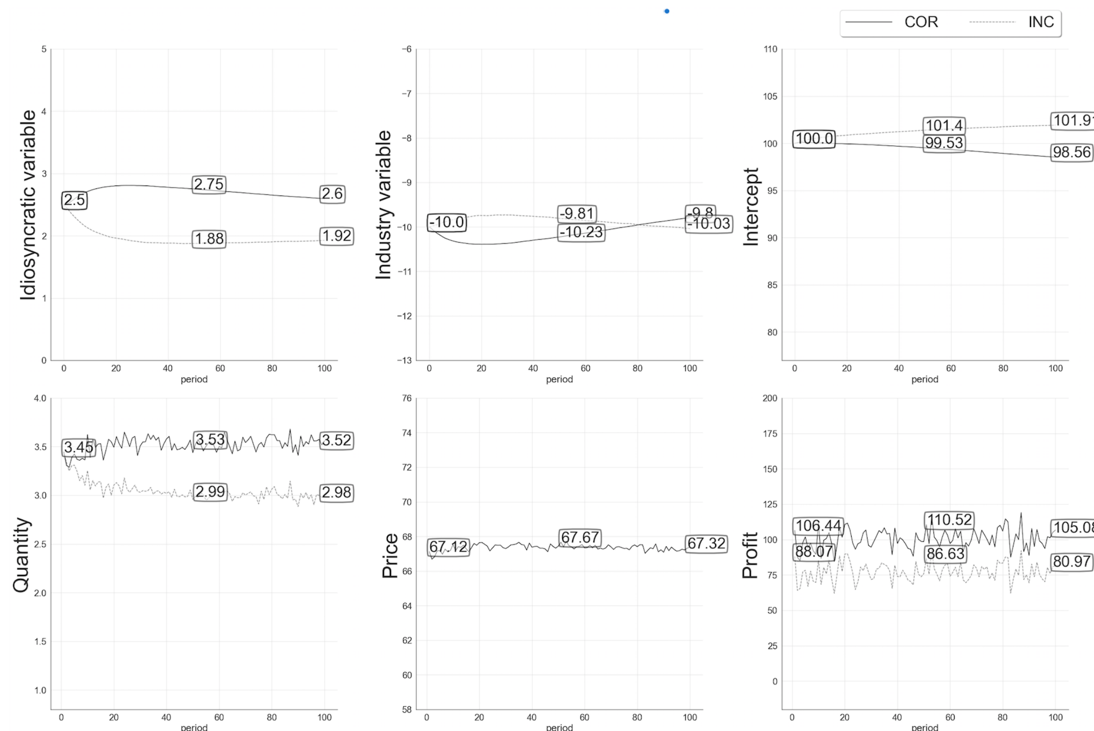


Figure OA2: Base Demand Structure Simulation Run Results of the Positive Idiosyncratic Case with β_1 *=+2.5.*



Simplified Python Code for the Demand Structure Model (for β_1 nonpositive)

This code consists of four modules: S3simplified_runsim.py, S3_simfunctions.py, S3_graphfunctions.py, and S3_paramfunctions.py. Use S3simplified_runsim.py to set parameters and run program.

S3simplified_runsim.py

```
# -*- coding: utf-8 -*-
"""
April 6, 2023
# Use this file to set parameters and run simulation using Python 3.8
# This set of programs is a simplified version of the simulation of the
base demand structure model with
#    correct and incorrect MR firms and self-projection.
# This simplified version works for negative-starting-valued idiosyncratic
terms.
# The simulation also requires S3_simfunctions.py, S3_paramfunctions.py,
and S3_graphfunctions.py files.

"""
import os
import S3_paramfunctions as pm

'''important for file path'''
version_no = 'S3'

if __name__ == '__main__':


    iterations = 10000

    idio_beta = -5
    ind_beta = -10

    #True demand parameters
    true_alpha_0 = 100
    true_idio_beta = -5
    true_ind_beta = -10

    #structurally correct firm's demand parameters
    cor_alpha_0 = 100
    cor_idio_beta = -5
    cor_ind_beta = -10

    #structurally incorrect firm's demand parameters
    inc_alpha_0 = 100
    inc_idio_beta = -5
    inc_ind_beta = -10


    coeffs = {'true_alpha_0': true_alpha_0, 'true_idio_beta':
true_idio_beta, 'true_ind_beta': true_ind_beta,
```

```
                 'cor_alpha_0': cor_alpha_0, 'cor_idio_beta':
cor_idio_beta, 'cor_ind_beta': cor_ind_beta,
                 'inc_alpha_0': inc_alpha_0, 'inc_idio_beta':
inc_idio_beta, 'inc_ind_beta': inc_ind_beta}


    phx_var = 0.5
    phq_var = 0.5
    x_var = 0.3
    phlen = 20
    max_q = 50
    periods = 100

    last_fold = 'outputgraphfolder' + str(iterations) + 'it'
    directory = os.path.join(os.getcwd(), str(version_no),

last_fold,str([true_alpha_0,true_idio_beta,0,true_ind_beta]))


    # %% Creating the parameter list
    param_list = pm.generate_param_list(iterations, coeffs,
                      phx_var, phq_var, x_var, phlen, max_q, periods,
                      last_fold, directory)


    # %% Running the simulation on the parameter list to create csv's.
    csv_name, dir_list, initial_weights =
pm.run_through_parameters(param_list, directory, version_no, last_fold,
                                                rerun_sim = True) #rerun_sim =
True if you want sim csv files to be generated even if an one already
exists at the file path.
```

simfunctions.py

```
# -*- coding: utf-8 -*-
"""
Created Feb. 4 2022

"""

import os
import numpy as np
import pandas as pd

import multiprocessing as mp
from multiprocessing import Pool
import concurrent.futures


def olsfunction(x, y):
    '''
```

```
    Returns coefficients and adjusted r-squared for OLS regression in
variable results.
    Results is a dictionary that contains coefficients as the first term
and adjusted r-square, adjrsq, as the second term.
    '''
    results={}
    A=np.c_[np.ones(len(y)),x] # generate matrix containing column of 1's
(for intercept term) appended to X matrix
    olsresults = np.linalg.lstsq(A, y, rcond = -100) # find coefficient
vector that minimizes squared residuals
    results['coefficients']=olsresults[0] # store coefficient vector in
'results' list
    ss_res=0 # initialize sum of squared residuals
    for row in range(len(y)):
        ss_res+=(y[row]-sum(A[row,:]*olsresults[0]))**2 # find sum of
squared residuals
    df_reg=len(y)-int(x.shape[1])-1 # save degrees of freedom of
regression
    df_tot = len(y)-1 # total degrees of freedom
    adjrsq=1-((ss_res/df_reg)/((y.size*y.var())/df_tot)) # find adjusted r
squared
    results['adjrsq']=adjrsq # store adjusted r q=squared value in results
    return results

def truncate(f, n):
    '''Truncates/pads a float f to n decimal places without rounding'''
    s = '{}'.format(f)
    if 'e' in s or 'E' in s:
        return '{0:.{1}f}'.format(f, n)
    i, p, d = s.partition('.')
    return '.'.join([i, (d+'0'*n)[:n]])

class Firm:
    def __init__(self, mental_model, init_betas, global_cost):
        self.mm = mental_model
        self.init_b = init_betas
        self.cost = global_cost
        self.cap = 0
        self.cap_maxedout = 0
        self.cap_posdemand = 0
        self.Q_lessthan_zero = 0


    def gen_init_history(self, init_hist_length, prehistmean, prehistsd,
                         prehistmean_Q, prehistsd_Q, directory, filebegin,
runsim_it, firm_num):

        weights_mm = self.init_b

        pre_hist_cov, pre_hist_covx1x2, pre_hist_covx1x3, pre_hist_covx2x3
= [0,0,0,0]
        pre_hist_means = [prehistmean, prehistmean, prehistmean]
        pre_hist_sd = prehistsd
        pre_hist_mean_Q = prehistmean_Q
```

```
        pre_hist_sd_Q = prehistsd_Q
        covar_matrix = [[pre_hist_sd**2, pre_hist_covx1x2,
pre_hist_covx1x3],
                        [pre_hist_covx1x2, pre_hist_sd**2,
pre_hist_covx2x3],
                        [pre_hist_covx1x3, pre_hist_covx2x3,
pre_hist_sd**2]]

        Q = {}
        H = []
        X_shocks = []
        hist_len = init_hist_length

        for i in range(hist_len):
            #Generates list of randoms of length N using the X shock means
and covariances given:
            X = np.random.multivariate_normal(pre_hist_means,
covar_matrix)
            #If any of the x shocks are zero, redraw:
            while (X[0] < 0) or (X[1] < 0) or (X[2] < 0):
                X = np.random.multivariate_normal(pre_hist_means,
covar_matrix)

            # Select first three random numbers to be demand shocks x1,
x2, x3
            x1 = X[0] #random shock x1
            x2 = X[1] #random shock x2
            x3 = X[2] #random shock x3

            #In the prehistory, there are random draws for quantities as
well.
            Q[i] = np.random.normal(pre_hist_mean_Q, pre_hist_sd_Q)
            values = [x1*Q[i], x2*Q[i], x3*Q[i]]

            #Price function.
            p = weights_mm[0] + (weights_mm[1] * values[0]) +
(weights_mm[2] * values[1]) + (weights_mm[3] * values[2])

            #Check if demand is upward sloping or flat (throws error)
            if ((weights_mm[1] * x1) + (weights_mm[2] * x2) +
(weights_mm[3] * x3)) >= 0:
                raise ValueError('Demand not downward sloping in period 0
in iteration' + str(runsim_it))

            #If price is negative, redraw the quantity. (previously price
was set to 0 if price was negative.)
            while (p <= 0):
                Q[i] = np.random.normal(pre_hist_mean_Q, pre_hist_sd_Q)
                values = [x1*Q[i], x2*Q[i], x3*Q[i]]
                p = weights_mm[0] + (weights_mm[1] * values[0]) +
(weights_mm[2] * values[1]) + (weights_mm[3] * values[2])

            #Another check for upward sloping demand
```

```
            if ((weights_mm[1] * x1) + (weights_mm[2] * x2) +
(weights_mm[3] * x3)) >= 0:
                raise ValueError('Demand upward sloping in pre-hist after
Q redraw in it. ' + str(runsim_it))

            #H is a list which contains a list [price, x1*Q, x2*Q, x3*Q]
for each period.
            H.append([p]+values)
            X_shocks.append(X)

        return H

    def calibrate(self, H):

        MM = self.mm
        H = np.array(H)
        indexes = [0]

        for j in range(len(MM)):
            if MM[j] == 1:
                if j == 0:
                    pass
                else:
                    indexes.append(j)
        H_firm = H[:, indexes]

        #Store price and whichever explanatory variables are stored H_firm
(x1*Q, x2*Q, and/or x3*Q) separately:
        vals = H_firm[: ,1:]
        price = H_firm[: ,0]
        ##Perform OLS on values and price
        output = olsfunction(vals, price)
        ##Store estimated coefficients of demand:
        weights = output['coefficients']
        adj_r2 = output['adjrsq']

        for j in range(len(MM)):
            if MM[j] == 0:
                weights = np.insert(weights, j, 0)

        return weights, adj_r2



    def chooseQ(self, periods, currentP, xvars, max_q, mean, sd):

        #Firm characteristics
        mm = self.mm
        c = self.cost
        weights = self.weights
        max_q = max_q
        x1, x2, x3 = xvars[0], xvars[1], xvars[2]
        period = currentP
```

```
        #Set Q to 0 to initialize Q.
        Q = 0

        #Upward sloping or flat:
        if (weights[1] * x1 * mm[1]) + (weights[2] * x2 * mm[2]) +
(weights[3] * x3 * mm[3]) >= 0:
            p_exp = weights[0] + (mm[1]*weights[1]*x1 +
mm[2]*weights[2]*x2 + mm[3]*weights[3]*x3) *2*max_q

            profit_exp = (p_exp - c) * max_q

            if profit_exp < 0:
                Q = 0
            else:
                Q = max_q
                self.cap += 1
                self.cap_posdemand += 1

        #Downward sloping:
        else:
            if weights[0]<0:
                    Q = 0
            else:
                Q = (2*c - c - weights[0])/(3*(weights[1]*x1*mm[1] +
weights[2]*mm[2]*x2 + mm[3]*weights[3]*x3))
                p_exp = weights[0] + (mm[1]*weights[1]*x1 +
mm[2]*weights[2]*x2 + mm[3]*weights[3]*x3) *2*Q
            profit_exp = (p_exp - c) * Q
            if profit_exp < 0:
                raise ValueError('Neg profit')

        #Boundary conditions: Quantity
        if Q < 0:
            Q = 0
            self.Q_lessthan_zero += 1
        if Q > max_q:
            Q = max_q
            self.cap += 1
            self.cap_maxedout += 1

        return Q, p_exp, profit_exp

def run_iteration(firmnum, unknown, truebetas, init_betas,
                  mental_models, prehistlen, dist_params, cost, periods,
periods_examind,
                  max_q, iterations, directory, filebegin, param_num, Q,
p_exp, profit_exp, ideal_Q, firm_profits, firm_betas,
                  truevalues, pricevalues, init_weights, dfs, iter):
        """Runs a single iteration of the simulation"""

        #Bringing in variables
        param_num, filebegin, directory = param_num, filebegin, directory
        mental_models, init_betas, truebetas = mental_models, init_betas,
truebetas
```

```
        c, max_q, periods, firmnum, iterations  = cost, max_q, periods,
firmnum, iterations
        prehistmean, prehistsd, mean, sd, prehistmean_Q, prehistsd_Q =
dist_params[0], dist_params[1], dist_params[2], dist_params[3],
dist_params[4], dist_params[5]

        covx1x2, covx1x3, covx2x3 = [0,0,0]
        mean = mean
        means = [mean, mean, mean]
        sd = sd
        covar_matrix = [[sd**2, covx1x2, covx1x3],
                        [covx1x2, sd**2, covx2x3],
                        [covx1x3, covx2x3, sd**2]]

        runsim_it = iter #added v2_5 to use as an index to unique the pre-
hist graphs

        Firms = {}
        for i in range(firmnum):
            mm = mental_models[i]
            b = init_betas[i]
            Firms[i] = Firm(mm, b, c)

            Firms[i].H = Firms[i].gen_init_history(prehistlen,
prehistmean, prehistsd,
                                                        prehistmean_Q,
prehistsd_Q, directory, filebegin, runsim_it, i)
            Firms[i].ideal_H = Firms[i].H[:]
            Firms[i].expectations = []

            Firms[i].weights = Firms[i].calibrate(Firms[i].H)[0] ##saves
the firm's initial weights
            weights = Firms[i].weights
            weights[0] = round(weights[0], 2)
            init_weights[i, iter] = Firms[i].weights

            Firms[i].firmvals = [] #initiate a list where the firm will
store the history of the interaction

            #comparing beginning of sim weights (which were arrived at by
calibrating) to the init_betas (or true betas)
            init_b = np.array(init_betas[i], dtype=np.float32)
            for weight in weights:
                if weight not in init_b:
                    break

        Firms[0].weights_opp = Firms[1].weights
        Firms[0].mm_opp = Firms[1].mm

        for t in range(periods):

            X = np.random.multivariate_normal(means, covar_matrix)

            #If any of the x shocks are zero, redraw:
```

```
            while (X[0] < 0) or (X[1] < 0) or (X[2] < 0):
                X = np.random.multivariate_normal(means, covar_matrix)

            x1 = X[0]
            x2 = X[1]
            x3 = X[2]
            xvars = [x1,x2,x3]

            #if (truebetas[1] * x1) + (truebetas[3] * x3) >= 0:
                #output_file.write('upward or flat slope in period ' +
str(t))

            for i in range(firmnum):

                weights = Firms[i].weights
                mm = Firms[i].mm

                #When the firm perceives demand to be upward sloping
                #if (weights[1] * x1 * mm[1]) + (weights[2] * x2 * mm[2])
+ (weights[3] * x3 * mm[3]) >= 0:
                   # output_file.write('\n firm ' + str(i) + ' perceived
upward or flat sloping before choosing Q in period ' + str(t))
                   # output_file.write('\n' + str(weights[1]) + ', ' +
str(weights[2]) + ', ' + str(weights[3]))

                #Firm i chooses a quantity for this period t.
                Q[t, i], p_exp[t,i], profit_exp[t,i] =
Firms[i].chooseQ(periods, t, xvars, max_q, mean, sd)

                Firms[i].Qperceived = Q[t,i] * 2

                #Add values to the firm's history of what it thinks
happened in this interaction
                Qper = Firms[i].Qperceived
                vals = [x1*Qper, x2*Qper, x3*Qper]
                Firms[i].firmvals.append(vals) #Store firm i's perceived
values of X*Q

            #Calculate total actual Q per period:
            Qtotal = np.sum(Q[t,:])

            #Fill dictionary with true values per period:
            truevalues[t] = [x1*Qtotal, x2*Qtotal, x3*Qtotal]

            #calculate price per period and save in pricevalues array
            p = np.inner(truebetas, [1] + truevalues[t])
            pricevalues[t] = p

            '''Create history of explanatory vars, prices, profits.'''
            '''Calibrate to fit OLS to what has happened in past
periods.'''

            comb_hist = []
            for i in range(firmnum):
```

```
                Firms[i].H.append([p] + Firms[i].firmvals[t])
                Firms[i].profit = (p-c)*Q[t,i]
                Firms[i].expectations.append([p_exp[t,i],
profit_exp[t,i]])
                firm_profits[t,i] = Firms[i].profit
                comb_hist.append(Firms[i].H)



            for i in range(firmnum):
                weights, adjr2 = Firm.calibrate(Firms[i], H = Firms[i].H)
                firm_betas[t,i] = weights
                Firms[i].weights = weights

        return Firms


def run_period_pooled(firmnum, unknown, truebetas, init_betas,
                      mental_models, prehistlen, dist_params, cost,
periods, periods_examind,
                      max_q, iterations,
                      directory, filebegin, param_num, Q, p_exp,
profit_exp, ideal_Q, firm_profits, firm_betas,
                      truevalues, pricevalues, init_weights, dfs, iter):

        #Bringing in variables
        param_num, filebegin, directory = param_num, filebegin, directory
        mental_models, init_betas, truebetas = mental_models, init_betas,
truebetas
        c, max_q, periods, firmnum, iterations  = cost, max_q, periods,
firmnum, iterations
        prehistmean, prehistsd, mean, sd, prehistmean_Q, prehistsd_Q =
dist_params[0], dist_params[1], dist_params[2], dist_params[3],
dist_params[4], dist_params[5]


        covx1x2, covx1x3, covx2x3 = [0,0,0]
        mean = mean
        means = [mean, mean, mean]
        sd = sd
        covar_matrix = [[sd**2, covx1x2, covx1x3],
                        [covx1x2, sd**2, covx2x3],
                        [covx1x3, covx2x3, sd**2]]

        runsim_it = iter #added v2_5 to use as an index to unique the pre-
hist graphs

        Firms = run_iteration(firmnum, unknown, truebetas, init_betas,
                   mental_models, prehistlen, dist_params, cost, periods,
periods_examind,
                   max_q, iterations,
                   directory, filebegin, param_num, Q, p_exp, profit_exp,
ideal_Q, firm_profits, firm_betas,
                   truevalues, pricevalues, init_weights, dfs, iter)
```

```
        '''Saving simulation data to csv.'''
        data = []
        for n in range(firmnum):
            firmhist = np.array(Firms[n].H[prehistlen:]) #excludes the
prehistory
            for l in range(len(firmhist)):
                data.append(firmhist[l])
        firmhist_df = pd.DataFrame(data, columns = ['price', 'x1*Q',
'x2*Q', 'x3*Q'])
        del firmhist_df['price']
        #split into 2 so they can be concatenated side-by-side; before
this, the two firm histories are stacked on top of one another.
        firmhist_df1 = firmhist_df.iloc[:periods,:]
        firmhist_df2 = firmhist_df.iloc[periods:,:]
        firmhist_df2.index = firmhist_df2.index - periods #corrects index

        #firm expectations dataframe:
        exp_data = {}
        exp_dict = {}
        for n in range(firmnum):
            exp_data[n] = []
            firmexp = np.array(Firms[n].expectations)
            for l in range(len(firmexp)):
                exp_data[n].append(firmexp[l])
            exp_dict[n] = pd.DataFrame(exp_data[n], columns =
['exp_price', 'exp_profit'])
        exp_df = pd.concat([exp_dict[0], exp_dict[1]], axis=1, sort =
True)
        exp_df.columns = ['exp_price_1', 'exp_profit_1', 'exp_price_2',
'exp_profit_2']

        #make firm profit dataframe:
        firmprof_df = pd.DataFrame(firm_profits, columns = ['prof1',
'prof2'])
        #make quantity dataframe:
        Q_df = pd.DataFrame(Q, columns = ['Q1', 'Q2'])
        Q_df['total_Q'] = Q_df.Q1 + Q_df.Q2

        #make price dataframe (Don't need this..same as first column of
firmhist_df):
        price_df = pd.DataFrame(pricevalues, index = [0]).transpose()
        price_df.columns = ['price'] # # # 100 rows of 24 vars each
iteration...

        #dataframe of firm betas... want to separate firm 1 and firm 2
using multi-index...
        firmb_df = pd.DataFrame.from_dict(firm_betas)
        firmb_df= firmb_df.transpose()
        firmb_df = firmb_df.reset_index(level=[1])
        firmb_df.columns = ['firm', 'int', 'b1', 'b2', 'b3']
        #separate into betas by firm, so can be concatenated side-by-side.
        firmb_df1 = firmb_df[firmb_df['firm']==0]
        firmb_df1.columns = ['firm', 'f1_int', 'f1_b1', 'f1_b2', 'f1_b3']
```

```
        del(firmb_df1['firm'])
        firmb_df2 = firmb_df[firmb_df['firm']==1]
        firmb_df2.columns = ['firm', 'f2_int', 'f2_b1', 'f2_b2', 'f2_b3']
        del(firmb_df2['firm'])

        #dataframe of truevales (a dictionary)
        truevals_df = pd.DataFrame.from_dict(truevalues).transpose()
        truevals_df.columns = ['true_x1Q', 'true_x2Q', 'true_x3Q']

        sim_data =
pd.concat([price_df,Q_df,truevals_df,firmprof_df,firmb_df1,firmb_df2,firmh
ist_df1,firmhist_df2, exp_df], axis=1, sort = True)
        dfs.append(sim_data)

        return sim_data

def run_simulation(firmnum, unknown, truebetas, init_betas,
                   mental_models, prehistlen, dist_params, cost, periods,
periods_examind,
                   max_q, iterations, directory, filebegin, param_num,
file_id):
    """
    Runs the simulation given parameters. Returns the filepath of the csv
containing the simulation data and a dataframe containing the initial
parameters.
    Calls run_period_pooled() in parallel process.
    """

    #Create directory if it does not already exist
    if not os.path.exists(directory):
        os.makedirs(directory)

    #Distribution parameters for the x demand shocks
    prehistmean, prehistsd, mean, sd, prehistmean_Q, prehistsd_Q =
*dist_params,
    covx1x2, covx1x3, covx2x3 = [0,0,0]
    means = [mean, mean, mean]
    sd = sd
    covar_matrix = [[sd**2, covx1x2, covx1x3],
                    [covx1x2, sd**2, covx2x3],
                    [covx1x3, covx2x3, sd**2]]

    #Create empty dictionaries to hold simulation info
    Q = np.empty((periods, firmnum))
    p_exp = np.empty((periods, firmnum))
    profit_exp = np.empty((periods, firmnum))
    ideal_Q = np.empty((periods, firmnum))
    firm_profits = np.empty((periods, firmnum))
    firm_betas = {}
    truevalues = {}
    pricevalues = {}
    init_weights = {}
```

```
    #Create a master list of dataframes to hold the dataframe created by
each iteration.
    dfs = []

    #parallel processing: runs multiple iterations in parallel.
    with concurrent.futures.ProcessPoolExecutor() as executor:
        periods_to_run = range(iterations)

        for iter in periods_to_run:
            sim_data = run_period_pooled(firmnum, unknown, truebetas,
init_betas,
                   mental_models, prehistlen, dist_params, cost, periods,
periods_examind,
                   max_q, iterations, directory, filebegin, param_num, Q,
p_exp, profit_exp, ideal_Q, firm_profits, firm_betas,
                   truevalues, pricevalues, init_weights, dfs, iter)

    #Initial weights dataframe: contains average initial weights to check
that firms begin with intended parameters
    initw_df = pd.DataFrame(init_weights).T
    init_totals = np.zeros((2,4))
    for i in range(firmnum):
        for j in range(iterations):
            init_totals[i] = init_totals[i] + initw_df.loc[i,j]
        init_totals[i] = init_totals[i]/iterations
    inits1 = pd.DataFrame(init_totals[0]).T
    inits2 = pd.DataFrame(init_totals[1]).T
    initav_df = pd.concat([inits1,inits2],axis = 1, sort = True)
    initav_df.columns = ['f1_int', 'f1_b1', 'f1_b2', 'f1_b3', 'f2_int',
'f2_b1', 'f2_b2', 'f2_b3']
    initav_df['period'] = 0
    initial_weights = initav_df

    #Create dataframe containing average results over all iterations:
    totals = np.zeros_like(sim_data)
    totals_df = pd.DataFrame(totals)
    for df in dfs:
        totals_df.columns = sim_data.columns
        totals_df.index = sim_data.index
        totals_df = df.add(totals_df, fill_value = 0)
    av_data = totals_df/iterations
    av_data['period'] = sim_data.index + 1
    av_data.index = sim_data.index
    av_df = pd.DataFrame.from_dict(av_data) #unnecessary step?

    #Storing run information in csv and return csv name (in addition to
initial weights)
    csv_name = file_id + '_sim_data.csv'
    av_df.to_csv(os.path.join(directory, csv_name), header = True)

    return csv_name, initial_weights
```

S3_graphfunctions.py

```
# -*- coding: utf-8 -*-
"""
Friday, Feb. 4 2022



"""
import os
import numpy as np
import pandas as pd
import matplotlib
matplotlib.rcParams['figure.max_open_warning'] = 0
import matplotlib.pyplot as plt
import seaborn as sns

def graph_label(figure, df, graph_content, graph_colors, label_params):

    figure, df = figure, df

    ##Rounding data and finding averages
    f1_data_round = round(graph_content[0], 2)
    f2_data_round = round(graph_content[1], 2)
    av_f2_data = np.mean(f2_data_round)
    av_f1_data = np.mean(f1_data_round)

    F1_inc, F2_inc, zero_inc, zero_horiz = label_params[0],
label_params[1], label_params[2], label_params[3]
    f1_edgecolor, f2_edgecolor, f1_boxcolor, f2_boxcolor =
graph_colors[0], graph_colors[1], graph_colors[2], graph_colors[3]

    #Loop through zipped points to graph.#
    #Firm 1#
    for pt_zip in zip(df.period, f1_data_round):
        period = pt_zip[0]
        if av_f1_data < av_f2_data:
            #if f1 is smaller, adjust it up
            if period % 25 == 0  and period != 0 and period !=100:
                figure.annotate(str(pt_zip[1]), xy=(pt_zip[0], pt_zip[1] +
F1_inc), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5", edgecolor =
f1_edgecolor, fc=f1_boxcolor))
            if period == 0:
                figure.annotate(str(pt_zip[1]), xy=(pt_zip[0] + 8,
pt_zip[1] + zero_inc), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5",
edgecolor = f1_edgecolor, fc=f1_boxcolor))
            if period == 100:
                figure.annotate(str(pt_zip[1]), xy=(pt_zip[0] - 5,
pt_zip[1] + F1_inc), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5",
edgecolor = f1_edgecolor, fc=f1_boxcolor))
        if av_f1_data > av_f2_data:
            #if f1 is larger, adjust it down
            if period % 25 == 0  and period != 0 and period !=100:
```

```
                figure.annotate(str(pt_zip[1]), xy=(pt_zip[0], pt_zip[1] -
F1_inc), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5", edgecolor =
f1_edgecolor, fc=f1_boxcolor))
            if period == 0:
                figure.annotate(str(pt_zip[1]), xy=(pt_zip[0] + 8,
pt_zip[1] - zero_inc), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5",
edgecolor = f1_edgecolor, fc=f1_boxcolor))
            if period == 100:
                figure.annotate(str(pt_zip[1]), xy=(pt_zip[0] - 5,
pt_zip[1] - F1_inc), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5",
edgecolor = f1_edgecolor, fc=f1_boxcolor))

    #Firm 2#
    for pt_zip in zip(df.period, f2_data_round):
        period = pt_zip[0]
        if av_f2_data > av_f1_data:
            #if f2 is larger, adjust it down
            if period % 25 == 0 and period != 0 and period != 100:
                figure.annotate(str(pt_zip[1]), xy=(pt_zip[0], pt_zip[1] -
F2_inc), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5", edgecolor =
f2_edgecolor, fc=f2_boxcolor))
            if period == 100:
                figure.annotate(str(pt_zip[1]), xy=(pt_zip[0] - 5,
pt_zip[1] - F2_inc), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5",
edgecolor = f2_edgecolor, fc=f2_boxcolor))
        if av_f2_data < av_f1_data:
            #if f2 is smaller, adjust it up
            if period % 25 == 0 and period != 0 and period != 100:
                figure.annotate(str(pt_zip[1]), xy=(pt_zip[0], pt_zip[1] +
F2_inc), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5", edgecolor =
f2_edgecolor, fc=f2_boxcolor))
            if period == 100:
                figure.annotate(str(pt_zip[1]), xy=(pt_zip[0] - 5,
pt_zip[1] + F2_inc), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5",
edgecolor = f2_edgecolor, fc=f2_boxcolor))

def graph_sns(csv_name, initial_weights, firmnum, unknown, truebetas,
init_betas,
                  mental_models, prehistlen, dist_params, cost, periods,
periods_examind, max_q, iterations,
                  directory, filebegin, param_num, parameters_dict):


    ##graphing_params = [csv_name, initial_weights, firmnum, unknown,
truebetas, init_betas,
    ##               mental_models, prehistlen, dist_params, cost,
periods, periods_examind, max_q, iterations,
    ##               directory, filebegin, param_num]

    directory = directory
    mm = mental_models
    truebetas=truebetas
    b = truebetas[1]
    periods = periods
```

```
    iw_df = initial_weights

    #bring in data saved in csv:
    df = pd.read_csv(os.path.join(directory, csv_name))

    #Make a beta data frame to include period 0
    dfbeta = df.reindex(columns = ['period', 'f1_int', 'f1_b1', 'f1_b2',
'f1_b3', 'f2_int', 'f2_b1', 'f2_b2', 'f2_b3', 'exp_price_1',
'exp_price_2', 'exp_profit_1', 'exp_profit_2'])
    dfbeta = pd.concat([dfbeta, iw_df], axis = 0, sort =
True).sort_values(by=['period']).reset_index()

    sns.set_style('white')
    sns.set_context('paper', font_scale=2)

    ##bbox variables:
    f1_edgecolor = 'black'
    f2_edgecolor = 'grey'
    f1_boxcolor = '1.0'
    f2_boxcolor = '0.8'
    graph_colors = [f1_edgecolor, f2_edgecolor, f1_boxcolor, f2_boxcolor]

    b1 = np.round(dfbeta.f1_b1[0], 0)

    ##################################################
    # Plot 1 - idiosyncratic slope estimate by firm #
    ##################################################

    ## Plotting lines. ##
    #Graph beta1 for firm 1.#
    graph_content = {}
    if mm[0] == [1,1,0,1]:
        fig1 = sns.lineplot(x = 'period', y = 'f1_b1', data = dfbeta,
label = 'Firm 1', color = 'black')
        graph_content[0] = dfbeta.f1_b1
    elif mm[0] == [1,0,1,1]:
        fig1 = sns.lineplot(x = 'period', y = 'f1_b2', data = dfbeta,
label = 'Firm 1', color = 'black')
        graph_content[0] = dfbeta.f1_b2
    if mm[1] == [1,1,0,1]:
        sns.lineplot(x = 'period', y = 'f2_b1', data = dfbeta, label =
'Firm 2', color = 'grey')
        f2_beta = dfbeta.f2_b1
        graph_content[1] = dfbeta.f2_b1
    elif mm[1] == [1,0,1,1]:
        sns.lineplot(x = 'period', y = 'f2_b2', data = dfbeta, label =
'Firm 2', color = 'grey')
        f2_beta = dfbeta.f2_b2
        graph_content[1] = dfbeta.f2_b2

    #Set the label params depending on model type.#
    label_params = [0.25, 0.04, 0, 0]
```

```
    #graphing_label function takes: figure name, all graphing params, and
then label_params list
    graph_label(fig1, dfbeta, graph_content, graph_colors, label_params)

    fig1.set(ylabel='Idiosyncratic variable')
    fig1.lines[1].set_linestyle("--")
    fig1.legend(loc='upper center', bbox_to_anchor=(0.5, -0.2),
shadow=True, ncol=2)
    plt.savefig(os.path.join(directory, 'plot1_idio' + str(filebegin) +
'b1_is_' + str(b1) + 'seaborn_plt1_3' + '.png'), bbox_inches='tight', dpi
= 600)
    plt.close()
    plt.cla()
    plt.clf()


    ###############################################
    # Plot 2 - industry variable estimate by firm #
    ###############################################

    ##Plotting Lines##
    # Plot beta3 for both firms. #
    fig2 = sns.lineplot(x = 'period', y = 'f1_b3', data = dfbeta, label =
'Firm 1', color = 'black')
    sns.lineplot(x = 'period', y = 'f2_b3', data = dfbeta, label = 'Firm
2', color = 'grey')
    graph_content = {0: dfbeta.f1_b3, 1: dfbeta.f2_b3}

    #Set different positional variables depending on model type.#
    label_params = [0.15, 0.15, 0, 5]

    graph_label(fig2, dfbeta, graph_content, graph_colors, label_params)

    fig2.set(ylabel='Industry variable')
    fig2.lines[1].set_linestyle("--")
    fig2.legend(loc='upper center', bbox_to_anchor=(0.5, -0.2),
shadow=True, ncol=2)
    plt.savefig(os.path.join(directory, 'plot2_ind_' + str(filebegin) +
'b1_is_' + str(b1) + 'seaborn_plt1_3' + '.png'), bbox_inches='tight', dpi
= 600)
    plt.close()
    plt.cla()
    plt.clf()

    #######################################
    # Plot 3 - intercept estimate by firm #
    #######################################

    ##Plotting Lines##
    # Plot intercept for both firms. #
    fig3 = sns.lineplot(x = 'period', y = 'f1_int', data = dfbeta, label =
'Firm 1', color = 'black')
    sns.lineplot(x = 'period', y = 'f2_int', data = dfbeta, label = 'Firm
2', color = 'grey')
```

```
    graph_content = {0: dfbeta.f1_int, 1: dfbeta.f2_int}

    #Set different positional variables depending on model type.#
    label_params = [0.15, 0.15, 0, 5]

    graph_label(fig3, dfbeta, graph_content, graph_colors, label_params)

    fig3.set(ylabel='Intercept')
    fig3.lines[1].set_linestyle("--")
    fig3.legend(loc='upper center', bbox_to_anchor=(0.5, -0.2),
shadow=True, ncol=2)
    plt.savefig(os.path.join(directory, 'plot3_int_' + str(filebegin) +
'b1_is_' + str(b1) + 'seaborn_plt1_3' + '.png'), bbox_inches='tight', dpi
= 600)
    plt.close()
    plt.cla()
    plt.clf()

    #############################
    # Plot 4 - Quantity by firm #
    #############################

    fig4 = sns.lineplot(x = 'period', y = 'Q1', data = df, label = 'Firm
1', color = 'black')
    sns.lineplot(x = 'period', y = 'Q2', data = df, label = 'Firm 2',
color = 'grey')
    graph_content = {0: df.Q1, 1: df.Q2}

    biggestQ = max(max(df.Q1), max(df.Q2))
    smallestQ = min(min(df.Q1), min(df.Q2))
    fig4.set_ylim([smallestQ - 0.4, biggestQ + 0.4])

    label_params = [0.25, 0.25, 0, 5]

    graph_label(fig4, df, graph_content, graph_colors, label_params)

    fig4.set(ylabel='Quantity')
    fig4.lines[1].set_linestyle("--")
    fig4.legend(loc='upper center', bbox_to_anchor=(0.5, -0.2),
shadow=True, ncol=2)
    plt.savefig(os.path.join(directory, 'plot4_q_' + str(filebegin) +
'b1_is_' + str(b1) + 'seaborn_plt1_3' + '.png'), bbox_inches='tight', dpi
= 600)
    plt.close()
    plt.cla()
    plt.clf()

    ##################
    # Plot 5 - Price #
    ##################
    fig5 = sns.lineplot(x = 'period', y = 'price', data = df, label =
'price', color = 'black')
    fig5.set(ylabel='Price')
```

```
    fig5.legend(loc='upper center', bbox_to_anchor=(0.5, -0.2),
shadow=True, ncol=1)

    fig5.set_ylim([min(df.price) - 0.5, max(df.price) + 0.5])

    price_round = df.price.round(2)
    mean_p = np.mean(price_round)
    for pt_zip in zip(df.period, price_round):
        period = pt_zip[0]
        if (period == 1 or period % 25 == 0) and period != 100:
            if pt_zip[1] > mean_p:
                #Adjust up if the point is above the mean
                fig5.annotate(str(pt_zip[1]), xy = (pt_zip[0], pt_zip[1] +
0.2), fontsize = 8, bbox=dict(boxstyle="round4,pad=.5", edgecolor =
'grey', fc='1.0'))
            if pt_zip[1] < mean_p:
                #Adjust down if the point is below the mean
                fig5.annotate(str(pt_zip[1]), xy = (pt_zip[0], pt_zip[1] -
0.2), fontsize = 8, bbox=dict(boxstyle="round4,pad=.5", edgecolor =
'grey', fc='1.0'))
        if period == 100:
            if pt_zip[1] > mean_p:
                fig5.annotate(str(pt_zip[1]), xy = (pt_zip[0] - 5,
pt_zip[1] + 0.2), fontsize = 8, bbox=dict(boxstyle="round4,pad=.5",
edgecolor = 'grey', fc='1.0'))
            if pt_zip[1] < mean_p:
                fig5.annotate(str(pt_zip[1]), xy = (pt_zip[0] - 5,
pt_zip[1] - 0.2), fontsize = 8, bbox=dict(boxstyle="round4,pad=.5",
edgecolor = 'grey', fc='1.0'))
    plt.savefig(os.path.join(directory, 'plot5_p_' + str(filebegin) +
'b1_is_' + str(b1)+ 'seaborn_plt1_3' + '.png'), bbox_inches='tight', dpi =
600)
    plt.close()
    plt.cla()
    plt.clf()


    ###########################
    # Plot 6 - Profit by firm #
    ###########################
    fig6 = sns.lineplot(x = 'period', y = 'prof1', data = df, label =
'Firm 1', color = 'black')
    sns.lineplot(x = 'period', y = 'prof2', data = df, label = 'Firm 2',
color = 'grey')
    fig6.set(ylabel='Profit')
    fig6.lines[1].set_linestyle("--")
    fig6.legend(loc='upper center', bbox_to_anchor=(0.5, -0.2),
shadow=True, ncol=1)

    biggestprof = max(max(df.prof1), max(df.prof2))
    smallestprof = min(min(df.prof1), min(df.prof2))
    fig6.set_ylim([smallestprof - 3, biggestprof + 3])

    prof1_round = df.prof1.round(2)
```

```
    for pt_zip in zip(df.period, prof1_round):
        period = pt_zip[0]
        if pt_zip[1] > df.prof2[period-1]:
            #If F1 profits are higher, annotate them higher
            if (period == 1 or period % 25 == 0) and period != 100:
                fig6.annotate(str(pt_zip[1]), xy=(pt_zip[0], biggestprof +
1), fontsize = 8, bbox=dict(boxstyle="round4,pad=.5", edgecolor =
f1_edgecolor, fc=f1_boxcolor))
            if period == 100:
                fig6.annotate(str(pt_zip[1]), xy=(pt_zip[0] - 5,
biggestprof + 1), fontsize = 8, bbox=dict(boxstyle="round4,pad=.5",
edgecolor = f1_edgecolor, fc=f1_boxcolor))
        else:
            #If F1 profits are lower, annotate them lower
            if (period == 1 or period % 25 == 0) and period != 100:
                fig6.annotate(str(pt_zip[1]), xy=(pt_zip[0], smallestprof
- 1), fontsize = 8, bbox=dict(boxstyle="round4,pad=.5", edgecolor =
f1_edgecolor, fc=f1_boxcolor))
            if period == 100:
                fig6.annotate(str(pt_zip[1]), xy=(pt_zip[0] - 5,
smallestprof - 1), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5",
edgecolor = f1_edgecolor, fc=f1_boxcolor))
    prof2_round = df.prof2.round(2)
    for pt_zip in zip(df.period, prof2_round):
        period = pt_zip[0]
        if pt_zip[1] > df.prof1[period-1]:
            #If F2 profits are higher, annotate them higher
            if (period == 1 or period % 25 == 0) and period != 100:
                fig6.annotate(str(pt_zip[1]), xy=(pt_zip[0], biggestprof +
1), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5", edgecolor =
f2_edgecolor, fc=f2_boxcolor))
            if period == 100:
                fig6.annotate(str(pt_zip[1]), xy=(pt_zip[0] - 5,
biggestprof + 1), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5",
edgecolor = f2_edgecolor, fc=f2_boxcolor))
        else:
            #If F2 profits are lower, annotate them lower
            if (period == 1 or period % 25 == 0) and period != 100:
                fig6.annotate(str(pt_zip[1]), xy=(period, smallestprof -
1), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5", edgecolor =
f2_edgecolor, fc=f2_boxcolor))
            if period == 100:
                fig6.annotate(str(pt_zip[1]), xy=(period - 5, smallestprof
- 1), fontsize = 8,  bbox=dict(boxstyle="round4,pad=.5", edgecolor =
f2_edgecolor, fc=f2_boxcolor))
    plt.savefig(os.path.join(directory, 'plot6_profit_' + str(filebegin) +
'b1_is_' + str(b1)+ 'seaborn_plt1_3' + '.png'), bbox_inches='tight', dpi =
600)
    plt.close()
    plt.cla()
    plt.clf()
```

S3_paramfunctions.py

```
# -*- coding: utf-8 -*-
"""
Feb. 4 2022

"""

import os
import pandas as pd
import matplotlib
matplotlib.rcParams['figure.max_open_warning'] = 0

import S3_simfunctions as sim
import S3_graphfunctions as gr

def generate_param_list(iterations, coeffs, phx_var, phq_var, x_var, 
phlen, max_q, periods,
                    last_fold, directory, param_list = []):

    true_alpha_0, true_idio_beta, true_ind_beta = coeffs['true_alpha_0'], 
coeffs['true_idio_beta'], coeffs['true_ind_beta']
    cor_alpha_0, cor_idio_beta, cor_ind_beta = coeffs['cor_alpha_0'], 
coeffs['cor_idio_beta'], coeffs['cor_ind_beta']
    inc_alpha_0, inc_idio_beta, inc_ind_beta = coeffs['inc_alpha_0'], 
coeffs['inc_idio_beta'], coeffs['inc_ind_beta']

    param_list = [2, 'q',[true_alpha_0,true_idio_beta,0,true_ind_beta],
                  
[[cor_alpha_0,cor_idio_beta,0,cor_ind_beta],[inc_alpha_0,0,inc_idio_beta,i
nc_ind_beta]],
              [[1,1,0,1],[1,0,1,1]],phlen,[1,phx_var,1,x_var,2,phq_var],
               50,periods, periods,max_q,iterations, 
               os.path.join(directory, 'cm=sr_base_duop') ,'base_sr_',1]

    return param_list


def run_through_parameters(param_list, directory, version_no, last_fold, 
rerun_sim):

    csv_name = {}
    initial_weights = {}

    if not os.path.exists(directory):
        os.makedirs(directory)

    param = param_list
    parameters_dict = {'true_alpha_0': param[2][0], 'true_idio_beta': 
param[2][1], 'true_ind_beta': param[2][3],
                       'cor_alpha_0': param[3][0][0], 'cor_idio_beta': 
param[3][0][1], 'cor_ind_beta': param[3][0][3],
                       'inc_alpha_0': param[3][1][0], 'inc_idio_beta': 
param[3][1][1], 'inc_ind_beta': param[3][1][3],
```

```
                       'phlen': param[5], 'phx_var': param[6][1], 'x_var':
param[6][3], 'phq_var': param[6][5],
                       'periods': param[8], 'max_q': param[10],
'iterations': param[11]}

    var_index = parameters_dict['true_idio_beta']
    file_id = 'sr_base_duop' + str(var_index)

    directory = param[12]
    sim_csv_name = file_id + '_sim_data.csv'

    if not os.path.exists(os.path.join(directory, sim_csv_name)):
        print('found no csv already existing at: ')
        print(os.path.join(directory, sim_csv_name))
        csv_name[var_index], initial_weights[var_index] =
sim.run_simulation(*param, file_id)
    else:
        print('This csv exists: ' + str(sim_csv_name))
        if rerun_sim == False:
          iwf1 = pd.DataFrame(param[4][0]).T
          iwf2 = pd.DataFrame(param[4][1]).T
          initial_weights[var_index] = pd.concat([iwf1, iwf2], axis=1,
sort=True)
          initial_weights[var_index].columns = ['f1_int', 'f1_b1',
'f1_b2','f1_b3', 'f2_int', 'f2_b1', 'f2_b2', 'f2_b3']
          initial_weights[var_index]['period'] = 0
          csv_name[var_index] = os.path.join(directory, sim_csv_name)
        else:
          print('rerunning simulation')
          csv_name[var_index], initial_weights[var_index] =
sim.run_simulation(*param, file_id)

    gr.graph_sns(csv_name[var_index], initial_weights[var_index], *param,
parameters_dict=parameters_dict)

    dir_list = [os.path.join(os.getcwd(), str(version_no), last_fold,

str([parameters_dict['true_alpha_0'],parameters_dict['true_idio_beta'],0,p
arameters_dict['true_ind_beta']]),
        'sr_base_duop'), [*param]]

    return csv_name, dir_list, initial_weights
```