Monte Carlo Results

Settings

Setting	Value
String	Any
Number of Replications	2
Number of Simulations	10
Number of Markets	100
Market Size	10
Number of Moments	87
Optimization Method	adaptive_de_rand_1_bin_radiuslimited
Max Steps	25
Max Time	1000
Population Size	50

Estimation Results with Identity Weighting Matrix

Parameter Name	True Parameters	Bias	RMSE	Coverage (95%)	Step Size
String	Float64	Float64	Float64	Float64	Float64
Rho 1	0.2	0.316748	0.317387	0.0	1.0e-5
Rho 2	0.4	0.221386	0.233803	0.0	1.0e-5
Rho 3	0.6	-0.0483442	0.11566	0.0	1.0e-5
Sigma	2.0	1.7708	1.79231	0.0	1.0e-5
Gamma 1	1.0	0.892694	1.56179	0.0	1.0e-5
Gamma 2	-2.0	-1.42763	2.04707	0.0	1.0e-5
Gamma 3	2.0	-0.114044	0.688718	0.0	1.0e-5
Gamma 4	1.5	2.01991	2.02308	0.0	1.0e-5

Settings

Setting	Value
String	Any
Number of Replications	2
Number of Simulations	10
Number of Markets	100
Market Size	10
Number of Moments	87
Optimization Method	adaptive_de_rand_1_bin_radiuslimited
Max Steps	25
Max Time	1000
Population Size	50

Estimation Results with Optimal Weighting Matrix

Parameter Name	True Parameters	Bias	RMSE	Coverage (95%)	Step Size
String	Float64	Float64	Float64	Float64	Float64
Rho 1	0.2	0.165593	0.183407	0.0	1.0e-5
Rho 2	0.4	0.0318321	0.0344464	0.0	1.0e-5
Rho 3	0.6	0.0813213	0.0815426	0.0	1.0e-5
Sigma	2.0	0.205658	1.33208	0.0	1.0e-5
Gamma 1	1.0	1.73611	1.73616	0.0	1.0e-5
Gamma 2	-2.0	-1.36149	1.36612	0.0	1.0e-5
Gamma 3	2.0	1.20411	1.22978	0.0	1.0e-5
Gamma 4	1.5	1.40533	1.40983	0.0	1.0e-5

Explanation of Equations and Julia Implementations

Now I will provide a detailed mapping between the equations specified in the original instructions and their corresponding implementations in the provided Julia script. Each equation is presented alongside the relevant code snippet, with explanations elucidating how the code realizes the mathematical formulations.

Calculation of W_{ij}

$$W_{ij} = Z'_{ij}\gamma + e_{ij}$$
$$Z_{ij}(1)\gamma_1 + Z_{ij}(2)\gamma_2 + Z_i(1)Z_j(1)\gamma_3 + Z_i(2)Z_j(2)\gamma_4$$

Code Implementation

```
W_ij = sum(Z_ij .* reshape(gamma, (1, 1, length(gamma))); dims=3)
```

Explanation

This line computes the match-specific utility W_{ij} by taking the dot product of the match-specific characteristics Z_{ij} with the coefficient vector γ. The 'reshape' function ensures that γ is correctly broadcasted across the dimensions of Z_{ij}.

Correlations and Standard Deviation of $b_{u,d}$

$$b_{u,d} = c_{1,1,u,d} = e_{1,1} + e_{u,d} - (e_{1,d} + e_{u,1})$$
$$\text{Cor}\left(b_{u_1,d_1}, b_{u_2,d_2}\right) = \rho_1, \text{ if } u_1 \neq u_2, d_1 \neq d_2$$
$$\text{Cor}\left(b_{u_1,d_1}, b_{u_2,d_1}\right) = \rho_2, \text{ if } u_1 \neq u_2$$
$$\text{Cor}\left(b_{u_1,d_1}, b_{u_1,d_2}\right) = \rho_3, \text{ if } d_1 \neq d_2$$
$$\text{SD}\left(b_{u_1,d_1}\right) = \sigma.$$

Code Implementation

```
if u1 == u2 && d1 == d2
    cov = sigma^2
elseif u1 == u2 && d1 != d2
    cov = sigma^2 * rho3
elseif u1 != u2 && d1 == d2
    cov = sigma^2 * rho2
else  # u1 != u2 && d1 != d2
    cov = sigma^2 * rho1
end
```

Explanation

Within the **generate_covariance** function, this conditional structure assigns the appropriate covariance between different $b_{u,d}$ terms based on their indices. Specifically:

- If both u and d indices are different, the correlation ρ_1 is applied.
- If only the u indices differ, ρ_2 is used.
- If only the d indices differ, ρ_3 is applied.
- If both indices are the same, the variance σ^2 is assigned.

This directly mirrors the correlation and standard deviation specifications in the equation.

Method of Simulated Moments Objective Function

$$\bar{Q}_{g,T}(\theta) = \frac{1}{T}\sum_{t=1}^{T}\left[\int_B g\left(A\left(Z_t, B; \theta\right), Z_t\right) \tilde{dF}_B(B; \theta) - g\left(A_t, Z_t\right)\right]$$

$$Q_{g,T}(\theta) = \frac{1}{T}\sum_{t=1}^{T}\left[\frac{1}{S}\sum_{s=1}^{S} g\left(A\left(Z_t, B_{s,t}; \theta\right), Z_t\right) - g\left(A_t, Z_t\right)\right]$$

$$D(\theta) = Q_T(\theta)' W Q_T(\theta)$$

Code Implementation

```
function msm_objective(theta, replication_data::DataFrame, W, N::Int, T::Int, S::Int, r::Int)
    rho1::Float64 = theta[1]
    rho2::Float64 = theta[2]
    rho3::Float64 = theta[3]
    sigma::Float64 = theta[4]
    gamma::Vector{Float64} = theta[5:8]

    simulated_data = simulate_data(theta, S, T, N, r)

    # Check if simulated_data is empty (due to non-positive definite Sigma_B)
    if nrow(simulated_data) == 0
        return 1e10  # Return a large value for the objective function
    end

    replication_moments = compute_all_markets_moments(replication_data)

    running_market_sum = zeros(_NUM_MOMENTS)

    for (t, cross_simulation_market_data) in enumerate(groupby(simulated_data, :market_id))
        market_sim_moments = compute_all_simulation_moments_for_market(
                                    cross_simulation_market_data)
        temp_sum = zeros(_NUM_MOMENTS)
        for s in 1:S
            temp_sum .+= market_sim_moments[s] - replication_moments[t]
        end
        running_market_sum .+= temp_sum ./ S
    end
    total_avg_diff = running_market_sum ./ T

    Q = dot(total_avg_diff, W * total_avg_diff)
    return Q
end
```

Explanation

The **msm_objective** function calculates the objective function $\bar{Q}_{g,T}(\theta)$ as defined in the equation. It performs the following steps:

1. Parameter Extraction: Extracts the parameters $\rho_1, \rho_2, \rho_3, \sigma, \gamma$ from the input vector θ.
2. Data Simulation: Generates simulated data based on the current parameter estimates.
3. Moment Computation: Computes the moments for both the replicated data and the simulated data.
4. Difference Calculation: Calculates the difference between simulated moments and replicated moments across all simulations and markets.

5. Objective Function Evaluation: Computes the quadratic form $Q = \bar{Q}_{g,T}(\theta)$ by weighting the squared differences with the weighting matrix W.

If the simulated data is invalid (e.g., due to a non-positive definite covariance matrix), the function returns a large penalty value to discourage the optimizer from considering those parameter values.

Transformation of b_{ij}^w

$$\begin{aligned} b_{ij}^w &= W_{11} + W_{ij} - W_{1j} - W_{i1} \\ &= (Z_{11} + Z_{ij} - Z_{1j} - Z_{i1})\,\gamma + (e_{11} + e_{ij} - e_{1j} - e_{i1}) \\ &= (Z_{11} + Z_{ij} - Z_{1j} - Z_{i1})\,\gamma + b_{ij} \end{aligned}$$

Code Implementation

```
delta = W_ij[1,1] .+ W_ij .- W_ij[1, :] .- W_ij[:, 1]
b = [zeros(1, N); zeros(N - 1, 1) B]
b_w = delta + b
```

Explanation

The transformation b_{ij}^w is constructed by adjusting the weighted utility W_{ij} with the baseline utility W_{11} and subtracting the utilities W_{1j} and W_{i1}. In the code:

- **delta** computes $W_{11} + W_{ij} - W_{1j} - W_{i1}$.
- **b** constructs the matrix b_{ij} by combining zeros and the unobserved complementarities B.
- **b_w** represents the adjusted utility b_{ij}^w.

This transformation is essential for setting up the optimization problem to determine the optimal matching A.

Definitions of $L^{\text{match}}(k)$ and $L^{\text{agent}}(k)$

$$\begin{aligned} L^{\text{match}}(k) &= \left(z_{u=A(1),d=1}(k), \ldots, z_{u=A(N_D),d=N_D}(k)\right) \\ L^{\text{agent}}(k) &= \left(z_{u=A(1)}(k) z_{d=1}(k), \ldots, z_{u=A(N_D)}(k) z_{d=N_D}(k)\right) \end{aligned}$$

Code Implementation

```
function compute_market_moments(market_data)
    ...
    L_match = Dict{Symbol, Vector{Float64}}()
    L_agent = Dict{Symbol, Vector{Float64}}()
    L_agent_up = Dict{Symbol, Vector{Float64}}()
    L_agent_down = Dict{Symbol, Vector{Float64}}()

    for k in match_characters
        L_match[k] = observed_matches[!, k]
    end
    for k in agent_characters
        L_agent[k] = observed_matches[!, k] .* observed_matches[!,
                        Symbol(replace(string(k), "Z_u" => "Z_d"))]
        L_agent_up[k] = observed_matches[!, k]
        L_agent_down[k] = observed_matches[!, Symbol(replace(string(k), "Z_u" => "Z_d"))]
    end
    ...
end
```

Explanation

Within the **compute_market_moments** function:

- **L_match[k]** collects the match-specific characteristics $z_{u=A(i),d=i}(k)$ for each k.
- **L_agent[k]** computes the product $z_{u=A(i)}(k)z_{d=j}(k)$ for each agent-specific characteristic k.

These vectors are essential for computing quantile, correlation, regression, and other moments as defined in the subsequent equations.

Quantile Moments for Match-specific and Agent-specific Characteristics

$$g_{\text{quan}}^{\text{match}}(k,p) = Q\left(L^{\text{match}}(k), p\right)$$
$$g_{\text{quan}}^{\text{agent}}(k,p) = Q\left(L^{\text{agent}}(k), p\right)$$

Code Implementation

```
function compute_quantile_moments!(L_values::Dict{Symbol, Vector{Float64}},
        moment_quantiles::Vector{Float64}, moments::Vector{Float64},
        moment_index::Int, characteristics::Vector{Symbol})

    for k in characteristics
        for p in moment_quantiles
            moments[moment_index] = quantile(L_values[k], p)
            moment_index += 1
        end
    end
    return moment_index
end
```

Explanation

The **compute_quantile_moments!** function iterates over each characteristic k and each quantile p, computing the p-th quantile of the corresponding $L^{\text{match}}(k)$ or $L^{\text{agent}}(k)$ vector. These quantile moments g_{quan} are essential for capturing the distributional properties of the characteristics in the matching game.

Correlation Moments

$$g_{\text{cor}}^{\text{match}}(k_1, k_2) = \text{Cor}\left(L^{\text{match}}(k_1), L^{\text{match}}(k_2)\right)$$
$$g_{\text{cor}}^{\text{agent}}(k) = \text{Cor}\left(L_{\text{up}}^{\text{agent}}(k), L_{\text{down}}^{\text{agent}}(k)\right)$$
$$g_{\text{cor}}^{\text{match-agent}}(k^{\text{match}}, k^{\text{agent}}) = \text{Cor}\left(L^{\text{match}}(k^{\text{match}}), L^{\text{agent}}(k^{\text{agent}})\right)$$

Code Implementation

```
function compute_correlation_moments!(L_match::Dict{Symbol, Vector{Float64}},
        L_agent_up::Dict{Symbol, Vector{Float64}}, L_agent_down::Dict{Symbol, Vector{Float64}},
        L_agent::Dict{Symbol, Vector{Float64}}, moments::Vector{Float64},
        moment_index::Int, match_characters::Vector{Symbol}, agent_characters::Vector{Symbol})

    for i in eachindex(match_characters)
        for j in eachindex(match_characters)
            if j > i
                k1 = match_characters[i]
                k2 = match_characters[j]
                moments[moment_index] = cor(L_match[k1], L_match[k2])
                moment_index += 1
            end
        end
    end

    for k in agent_characters
        moments[moment_index] = cor(L_agent_up[k], L_agent_down[k])
        moment_index += 1
    end

    for k_match in match_characters
        for k_agent in agent_characters
            moments[moment_index] = cor(L_match[k_match], L_agent[k_agent])
            moment_index += 1
        end
    end
    return moment_index
end
```

Explanation

The **compute_correlation_moments!** function calculates the correlation moments as defined in the equation:

- Match-Match Correlations: Computes the correlation between pairs of match-specific characteristics k_1 and k_2.
- Agent-Agent Correlation: Computes the correlation between agent-specific up and down characteristics for each k.
- Match-Agent Correlations: Computes the correlation between match-specific characteristics and agent-specific characteristics.

These correlations capture the interdependencies between different characteristics.

Regression Moments

$$g_{\text{reg}} : \text{regress } \mathbf{1}(ij \in A) \text{ on } \{Z_{ij}(k)\}_k \text{ and } \{Z_i(k)Z_j(k)\}_k$$

Code Implementation

```
function compute_regression_moments!(market_data::DataFrame, moments::Vector{Float64},
        moment_index::Int, match_characters::Vector{Symbol}, agent_characters::Vector{Symbol})

    matches_indicator = market_data.matched

    regression_df = DataFrame(matches_indicator = matches_indicator)

    for k in match_characters
        regression_df[!, string(k)] = market_data[!, k]
    end
    for k in agent_characters
        interaction_term = market_data[!, k] .* market_data[!,
                        Symbol(replace(string(k), "Z_u" => "Z_d"))]
        regression_df[!, string(k) * "_interaction"] = interaction_term
    end

    if all(matches_indicator .== 1) || all(matches_indicator .== 0)
        coefficients = zeros(4)
    else
        X = hcat(
            regression_df[!, string(match_characters[1])],
            regression_df[!, string(match_characters[2])],
            regression_df[!, string(agent_characters[1]) * "_interaction"],
            regression_df[!, string(agent_characters[2]) * "_interaction"]
        )
        y = regression_df.matches_indicator
        X = hcat(ones(size(X, 1)), X)
        beta = inv(X' * X) * (X' * y)
        coefficients = beta[2:end]
    end

    for coef in coefficients
        moments[moment_index] = coef
        moment_index += 1
    end
    return moment_index
end
```

Explanation

The **compute_regression_moments!** function performs a regression of the indicator variable $\mathbf{1}(ij \in A)$ on the match-specific characteristics $Z_{ij}(k)$ and their interactions $Z_i(k)Z_j(k)$. The steps include:

1. Data Preparation: Constructs a DataFrame containing the match indicators and the relevant characteristics.
2. Interaction Terms: Creates interaction terms between agent-specific characteristics.
3. Regression Execution: If there is variation in the match indicators, it performs a linear regression and extracts the coefficients.
4. Moment Storage: Stores the regression coefficients as moments.

Opportunity Cost Moments for Match-specific Characteristics

$$g^{\text{match}}_{\text{opp\_up}}(k,p) = Q\left(L^{\text{match}}_{\text{opp\_up}}(k), p\right)$$
$$g^{\text{match}}_{\text{opp\_down}}(k,p) = Q\left(L^{\text{match}}_{\text{opp\_down}}(k), p\right)$$

Code Implementation

```
function compute_opportunity_cost_moments_match!(market_data::DataFrame,
        observed_matches::DataFrame, N::Int, moments::Vector{Float64},
        moment_index::Int, match_characters::Vector{Symbol},
        moment_quantiles::Vector{Float64})

    L_match_opp_up = Dict{Symbol, Vector{Float64}}()
    L_match_opp_down = Dict{Symbol, Vector{Float64}}()

    for k in match_characters
        L_match_opp_up_k = zeros(N)
        L_match_opp_down_k = zeros(N)
        for i in 1:N
            matched_j = observed_matches.downstream_id[observed_matches.upstream_id .== i][1]
            opp_values_up = market_data[(market_data.upstream_id .== i) .&&
                            (market_data.downstream_id .!= matched_j), k]
            if isempty(opp_values_up)
                L_match_opp_up_k[i] = NaN
            else
                L_match_opp_up_k[i] = mean(opp_values_up)
            end

            matched_i = observed_matches.upstream_id[observed_matches.downstream_id
                                                     .== matched_j][1]
            opp_values_down = market_data[(market_data.downstream_id .== matched_j) .&&
                              (market_data.upstream_id .!= matched_i), k]
            if isempty(opp_values_down)
                L_match_opp_down_k[i] = NaN
            else
                L_match_opp_down_k[i] = mean(opp_values_down)
            end
        end

        L_match_opp_up[k] = L_match_opp_up_k
        L_match_opp_down[k] = L_match_opp_down_k
    end

    for k in match_characters
        for p in moment_quantiles
            valid_values_up = filter(!isnan, L_match_opp_up[k])
            moments[moment_index] = isempty(valid_values_up) ? NaN
                                                             : quantile(valid_values_up, p)
            moment_index += 1

            valid_values_down = filter(!isnan, L_match_opp_down[k])
            moments[moment_index] = isempty(valid_values_down) ? NaN
                                                               : quantile(valid_values_down, p)
            moment_index += 1
        end
    end

    return moment_index
end
```

Explanation

The **compute_opportunity_cost_moments_match!** function calculates the opportunity cost moments for match-specific characteristics. For each characteristic k and each quantile p:

1. Upward Opportunity Cost ($L_{\text{opp\_up}}$): Computes the average $Z_{ij}(k)$ for each upstream agent i excluding their matched downstream agent.
2. Downward Opportunity Cost ($L_{\text{opp\_down}}$): Computes the average $Z_{ij}(k)$ for each downstream agent j excluding their matched upstream agent.
3. Quantile Calculation: Calculates the p-th quantile of the opportunity costs and stores them as moments.

Opportunity Cost Moments for Agent-specific Characteristics

$$g^{\text{agent}}_{\text{opp\_up}}(k,p) = Q\left(L^{\text{agent}}_{\text{opp\_up}}(k), p\right)$$
$$g^{\text{agent}}_{\text{opp\_down}}(k,p) = Q\left(L^{\text{agent}}_{\text{opp\_down}}(k), p\right)$$

Code Implementation

```
function compute_opportunity_cost_moments_agent!(market_data::DataFrame,
        observed_matches::DataFrame, N::Int, moments::Vector{Float64},
        moment_index::Int, agent_characters::Vector{Symbol}, moment_quantiles::Vector{Float64})

    L_agent_opp_up = Dict{Symbol, Vector{Float64}}()
    L_agent_opp_down = Dict{Symbol, Vector{Float64}}()

    for k in agent_characters
        L_agent_opp_up_k = zeros(N)
        L_agent_opp_down_k = zeros(N)
        for i in 1:N
            matched_j = observed_matches.downstream_id[observed_matches.upstream_id .== i][1]
            Z_i_k = observed_matches[observed_matches.upstream_id .== i, k][1]
            opp_values_up = market_data[(market_data.upstream_id .== i)
                            .&& (market_data.downstream_id .!= matched_j),
                            Symbol(replace(string(k), "Z_u" => "Z_d"))]
            if isempty(opp_values_up)
                L_agent_opp_up_k[i] = NaN
            else
                interaction_terms = Z_i_k .* opp_values_up
                L_agent_opp_up_k[i] = mean(interaction_terms)
            end

            matched_i = observed_matches.upstream_id[observed_matches.downstream_id
                                                     .== matched_j][1]
            Z_j_k = observed_matches[observed_matches.downstream_id .== matched_j,
                                     Symbol(replace(string(k), "Z_u" => "Z_d"))][1]
            opp_values_down = market_data[(market_data.downstream_id .== matched_j)
                                          .&& (market_data.upstream_id .!= matched_i), k]
            if isempty(opp_values_down)
                L_agent_opp_down_k[i] = NaN
            else
                interaction_terms = opp_values_down .* Z_j_k
                L_agent_opp_down_k[i] = mean(interaction_terms)
            end
        end
        L_agent_opp_up[k] = L_agent_opp_up_k
        L_agent_opp_down[k] = L_agent_opp_down_k
    end

    for k in agent_characters
        for p in moment_quantiles
            valid_values_up = filter(!isnan, L_agent_opp_up[k])
            moments[moment_index] = isempty(valid_values_up) ? NaN
                                          : quantile(valid_values_up, p)
            moment_index += 1

            valid_values_down = filter(!isnan, L_agent_opp_down[k])
            moments[moment_index] = isempty(valid_values_down) ? NaN
                                          : quantile(valid_values_down, p)
            moment_index += 1
        end
    end

    return moment_index
end
```

Explanation

The **compute_opportunity_cost_moments_agent!** function calculates opportunity cost moments for agent-specific characteristics. For each characteristic k and each quantile p:

1. Upward Opportunity Cost ($L_{\text{opp\_up}}$): Computes the average product $Z_i(k)Z_d(k)$ for each upstream agent i excluding their matched downstream agent.
2. Downward Opportunity Cost ($L_{\text{opp\_down}}$): Computes the average product $Z_i(k)Z_d(k)$ for each downstream agent j excluding their matched upstream agent.
3. Quantile Calculation: Calculates the p-th quantile of these opportunity costs and stores them as moments.

Regression Moments for Opportunity Cost

$$g_{\text{reg\_opp}} : \text{regress } \mathbf{1}(ij \in A) \text{ on}$$

$$\frac{1}{N-1}\sum_{d\neq j} z_{id}(k),$$

$$\frac{1}{N-1}\sum_{u\neq i} z_{uj}(k),$$

$$\frac{1}{N-1}\sum_{d\neq j} z_i(k)z_d(k), \text{ and}$$

$$\frac{1}{N-1}\sum_{u\neq i} z_u(k)z_j(k)$$

Code Implementation

```
function compute_regression_moments_opportunity_cost!(market_data::DataFrame,
        moments::Vector{Float64}, moment_index::Int, match_characters::Vector{Symbol},
        agent_characters::Vector{Symbol})

    regression_df_opp = DataFrame(matches_indicator = market_data.matched)
    for k in match_characters
        opp_up = zeros(size(market_data, 1))
        for idx_row in 1:size(market_data, 1)
            i = market_data.upstream_id[idx_row]
            j = market_data.downstream_id[idx_row]
            opp_values = market_data[(market_data.upstream_id .== i) .&&
                                     (market_data.downstream_id .!= j), k]
            opp_up[idx_row] = isempty(opp_values) ? NaN : mean(opp_values)
        end
        regression_df_opp[!, "opp_up_" * string(k)] = opp_up

        opp_down = zeros(size(market_data, 1))
        for idx_row in 1:size(market_data, 1)
            i = market_data.upstream_id[idx_row]
            j = market_data.downstream_id[idx_row]
            opp_values = market_data[(market_data.downstream_id .== j) .&&
                         (market_data.upstream_id .!= i), k]
            opp_down[idx_row] = isempty(opp_values) ? NaN : mean(opp_values)
        end
        regression_df_opp[!, "opp_down_" * string(k)] = opp_down
    end
    for k in agent_characters

        opp_agent_up = zeros(size(market_data, 1))
        for idx_row in 1:size(market_data, 1)
            i = market_data.upstream_id[idx_row]
            j = market_data.downstream_id[idx_row]
            Z_i_k = market_data[market_data.upstream_id .== i .&&
                                market_data.downstream_id .== j,
                                Symbol(replace(string(k), "Z_u" => "Z_d"))][1]
            opp_values = market_data[(market_data.upstream_id .== i) .&&
                                     (market_data.downstream_id .!= j),
                                     Symbol(replace(string(k), "Z_u" => "Z_d"))]
            if isempty(opp_values)
                opp_agent_up[idx_row] = NaN
            else
                interaction_terms = Z_i_k .* opp_values
                opp_agent_up[idx_row] = mean(interaction_terms)
            end
        end
        regression_df_opp[!, "opp_agent_up_" * string(k)] = opp_agent_up

        opp_agent_down = zeros(size(market_data, 1))
        for idx_row in 1:size(market_data, 1)
            i = market_data.upstream_id[idx_row]
            j = market_data.downstream_id[idx_row]
            Z_j_k = market_data[market_data.upstream_id .== i .&&
                                market_data.downstream_id .== j,
                                Symbol(replace(string(k), "Z_u" => "Z_d"))][1]
            opp_values = market_data[(market_data.downstream_id .== j) .&&
                         (market_data.upstream_id .!= i), k]
            if isempty(opp_values)
```

```
                opp_agent_down[idx_row] = NaN
            else
                interaction_terms = opp_values .* Z_j_k
                opp_agent_down[idx_row] = mean(interaction_terms)
            end
        end
        regression_df_opp[!, "opp_agent_down_" * string(k)] = opp_agent_down
    end

    complete_cases = dropmissing(regression_df_opp, disallowmissing=true)

    matches_indicator = complete_cases.matches_indicator
    if all(matches_indicator .== 1) || all(matches_indicator .== 0) || nrow(complete_cases) == 0
        coefficients_opp = zeros(4)
    else
        X_opp = hcat(
            complete_cases[!, "opp_up_" * string(match_characters[1])],
            complete_cases[!, "opp_down_" * string(match_characters[1])],
            complete_cases[!, "opp_agent_up_" * string(agent_characters[1])],
            complete_cases[!, "opp_agent_down_" * string(agent_characters[1])]
        )
        y_opp = complete_cases.matches_indicator
        X_opp = hcat(ones(size(X_opp, 1)), X_opp)
        beta_opp = inv(X_opp' * X_opp) * (X_opp' * y_opp)
        coefficients_opp = beta_opp[2:end]
    end
    for coef in coefficients_opp
        moments[moment_index] = coef
        moment_index += 1
    end
    return moment_index
end
```

Explanation

The **compute_regression_moments_opportunity_cost!** function performs regression analyses to estimate the effect of opportunity costs on the matching decisions. Specifically

1. Opportunity Cost Calculation: Computes the average opportunity costs for both upstream and downstream agents by aggregating $Z_{id}(k)$ and $Z_{uj}(k)$ terms.

2. Data Preparation: Constructs a DataFrame containing the match indicators and the calculated opportunity cost terms.

3. Regression Execution: Performs a linear regression of the match indicators on the opportunity cost terms, extracting the coefficients as moments.

Rank Moments

$$g_{\text{rank\_up}}(k) = \frac{1}{N^2} \sum_{ij \in A} \text{Rank}\left(Z_{ij}(k), \text{Column } j\right)$$

$$g_{\text{rank\_down}}(k) = \frac{1}{N^2} \sum_{ij \in A} \text{Rank}\left(Z_{ij}(k), \text{Row } i\right)$$

Code Implementation

```
function compute_rank_moments!(market_data::DataFrame, observed_matches::DataFrame,
                    N::Int, moments::Vector{Float64}, moment_index::Int,
                    match_characters::Vector{Symbol}, agent_characters::Vector{Symbol})

    L_rank_up = Dict{Symbol, Vector{Float64}}()
    L_rank_down = Dict{Symbol, Vector{Float64}}()

    for k in match_characters
        ranks_up = tiedrank(observed_matches[!, k])
        ranks_down = tiedrank(observed_matches[!, Symbol(replace(string(k), "Z_u" => "Z_d"))])
        rank_diff = abs.(ranks_up - ranks_down)

        moments[moment_index] = mean(rank_diff) / N
        moment_index += 1

        moments[moment_index] = var(rank_diff) / N
        moment_index += 1
    end

    return moment_index
end
```

Explanation

The **compute_rank_moments!** function calculates rank-based moments for both upstream and downstream characteristics:

1. Rank Calculation: Computes the ranks of $Z_{ij}(k)$ within each column and row.
2. Rank Differences: Calculates the absolute differences between upstream and downstream ranks.
3. Moment Calculation: Computes the mean and variance of these rank differences, normalized by N, and stores them as moments.

Asymptotic Distribution of the Estimator

$$\sqrt{T}\left(\hat{\theta}-\theta_0\right) \xrightarrow{d} \mathcal{N}\left[0, \left(1+\frac{1}{S}\right) G^\top W \Omega W^\top G \left(G^\top W G\right)^{-1}\right]$$

Code Implementation

```
function compute_variance_matrix(theta_hat::Vector{Float64}, data::DataFrame,
            W::Matrix{Float64}, S::Int, T::Int, N::Int, r::Int; h::Vector{Float64})
    ...
    AVAR = factor * inv_GWG * (G' * W * omega * W * G) * inv_GWG

    # Compute standard errors
    std_errors = sqrt.(diag(AVAR))

    # Compute 95% confidence intervals
    z_value = quantile(Distributions.Normal(0, 1), 0.975)
    lower_bounds = theta_hat .- z_value .* std_errors
    upper_bounds = theta_hat .+ z_value .* std_errors

    confidence_intervals = hcat(lower_bounds, upper_bounds)

    return std_errors, confidence_intervals
end
```

Explanation

The **compute_variance_matrix** function calculates the asymptotic variance-covariance matrix $\text{AVAR}(\hat{\theta})$ as specified in the equation. It performs the following steps:

1. Gradient Matrix G: Computes the numerical gradient of the moment conditions with respect to the parameters θ.
2. Covariance Matrix Ω: Estimates the covariance matrix of the moments across markets.
3. Variance-Covariance Matrix Calculation: Applies the formula to compute $\text{AVAR}(\hat{\theta})$ using the gradient matrix G, weighting matrix W, and covariance matrix Ω.
4. Standard Errors and Confidence Intervals: Derives the standard errors from the diagonal of AVAR and constructs 95% confidence intervals using the standard normal quantile.

Definitions of G and Ω

$$G = \mathrm{E}\left[\nabla_\theta g\left(Y_t, \theta_0\right)\right]$$
$$\Omega = \mathrm{E}\left[g\left(Y_t, \theta_0\right) g\left(Y_t, \theta_0\right)^\top\right]$$

$$G \approx \frac{1}{T}\sum_t \left(\nabla_\theta g\left(Y_t, \hat{\theta}\right)\right) = \nabla_\theta\left(\frac{1}{T}\sum_t g\left(Y_t, \hat{\theta}\right)\right)$$
$$\Omega = \mathrm{E}\left[g\left(Y_t, \theta_0\right) g\left(Y_t, \theta_0\right)^\top\right] \approx \frac{1}{T}\sum_t \left(g\left(Y_t, \hat{\theta}\right) g\left(Y_t, \hat{\theta}\right)^\top\right)$$

Code Implementation

```
G = zeros(num_moments, num_params)

for i in 1:num_params
    theta_plus = copy(theta_hat)
    theta_minus = copy(theta_hat)
    theta_plus[i] += h[i]
    theta_minus[i] -= h[i]

    data_plus = simulate_data(theta_plus, S, T, N, r)
    data_minus = simulate_data(theta_minus, S, T, N, r)
    moments_plus = compute_avg_moments(data_plus, S, T)
    moments_minus = compute_avg_moments(data_minus, S, T)

    G[:, i] = (moments_plus - moments_minus) / (2 * h[i])
end

moments_T = zeros(T, num_moments)
for t in 1:T
    market_data = data[data.market_id .== t, :]
    moments_T[t, :] = compute_market_moments(market_data)
end
omega = cov(moments_T, dims=1, corrected=false) + 1e-6 * I(num_moments)
```

Explanation

Within the **compute_variance_matrix** function:

- Gradient Matrix G: The loop over parameters i computes the numerical derivative of the moments $g(Y_t, \theta_0)$ with respect to each parameter θ_i using finite differences. This estimates the expectation $G = \mathrm{E}\left[\nabla_\theta g\left(Y_t, \theta_0\right)\right]$.
- Covariance Matrix Ω: The covariance matrix Ω is estimated by computing the sample covariance of the moments across all markets, with a small regularization term added to the diagonal to ensure numerical stability.

Asymptotic Variance-Covariance Matrix $\mathrm{AVAR}(\hat{\theta})$

$$\mathrm{AVAR}(\hat{\theta}) = \frac{1}{T}\left(1 + \frac{1}{S}\right)\left(G^\top W G\right)^{-1} G^\top W \Omega W^\top G \left(G^\top W G\right)^{-1}$$

Code Implementation

```
factor = (1 + 1/S) / T

GWG = G' * W * G
inv_GWG = pinv(GWG)

AVAR = factor * inv_GWG * (G' * W * omega * W * G) * inv_GWG
```

Explanation

The code calculates the asymptotic variance-covariance matrix AVAR($\hat{\theta}$) using the formula provided in the equation:

1. Factor Calculation: Computes the scaling factor $\frac{1}{T}\left(1+\frac{1}{S}\right)$.
2. Matrix Products: Calculates $G^\top WG$ and its pseudo-inverse to handle potential singularity.
3. Variance-Covariance Matrix AVAR($\hat{\theta}$): Combines these components to compute the final AVAR.

This computation is required to understand the precision of the estimated parameters and for constructing confidence intervals.

Bias of the Estimator

$$\text{Bias} \approx \frac{1}{R}\sum_{r}\left(\hat{\theta}_r^1 - \theta_0^1\right)$$

Code Implementation

```
function compute_statistics(theta_estimates::Array{Float64, 2},
        confidence_intervals::Array{Float64, 3}, theta_true::Vector{Float64})
    num_params = length(theta_true)
    bias = mean(theta_estimates, dims=1) - theta_true'
    ...
    return bias, rmse, coverage
end
```

Explanation

The **compute_statistics** function calculates the bias of the estimator by taking the average of the estimated parameters across all replications and subtracting the true parameter values.

Root Mean Squared Error (RMSE) of the Estimator

$$\text{RMSE} \approx \sqrt{\frac{1}{R}\sum_{r}\left(\hat{\theta}_r^1 - \theta_0^1\right)^2}$$

Code Implementation

```
rmse = sqrt.(mean((theta_estimates .- theta_true').^2, dims=1))
```

Explanation

Within the **compute_statistics** function, RMSE is computed by taking the square root of the average squared differences between the estimated parameters and the true parameters across all replications. This calculation provides a measure of the estimator's overall accuracy.

Coverage Probability of Confidence Intervals

$$\text{Coverage} = \frac{1}{R} \sum 1 \left[95\% \text{ CI for replication } r \text{ contains } \theta_0^1 \right]_r$$

Code Implementation

```
coverage = zeros(num_params)
for i in 1:num_params
    lower_bounds = confidence_intervals[:, i, 1]
    upper_bounds = confidence_intervals[:, i, 2]
    coverage[i] = mean((theta_true[i] .>= lower_bounds) .& (theta_true[i] .<= upper_bounds))
end
```

Explanation

The **compute_statistics** function assesses the coverage probability by checking, for each parameter, the proportion of confidence intervals that contain the true parameter value θ_0.

Asymptotic Normality of the MSM Estimator

$$\sqrt{T} \left(\hat{\theta} - \theta_0 \right) \xrightarrow{d} \mathcal{N} \left[0, \left(1 + \frac{1}{S} \right) G^\top W \Omega W^\top G \left(G^\top W G \right)^{-1} \right]$$

Code Implementation

```
# Compute standard errors
std_errors = sqrt.(diag(AVAR))

# Compute 95% confidence intervals
z_value = quantile(Distributions.Normal(0, 1), 0.975)
lower_bounds = theta_hat .- z_value .* std_errors
upper_bounds = theta_hat .+ z_value .* std_errors

confidence_intervals = hcat(lower_bounds, upper_bounds)
```

Explanation

After computing the asymptotic variance-covariance matrix $\text{AVAR}(\hat{\theta})$, the code derives the standard errors by taking the square roots of its diagonal elements. It then calculates the 95% confidence intervals using the standard normal quantile $z_{0.975}$. This process leverages the asymptotic normality of the MSM estimator to facilitate statistical inference.

Step By Step

1 Step 1: Generate Z_{ij} and B

Instruction: Write code to generate Z_{ij} for each match i, j and the matrix B for each of T markets in order to generate fake data.

Code Snippet:

```
function generate_covariance(N::Int64, sigma::Float64, rho1::Float64,
        rho2::Float64, rho3::Float64)
    sigma_B = zeros(num_elements, num_elements)
    ...
    return sigma_B
end

function generate_market_data(N::Int64, K::Int64)
    Z_u = randn(N, 2)
    Z_d = randn(N, 2)
    Z_match = randn(N, N, 2)

    Z_ij = Array{Float64, 3}(undef, N, N, K)
    for i in 1:N
        for j in 1:N
            Z_ij[i, j, 1] = Z_match[i, j, 1]
            Z_ij[i, j, 2] = Z_match[i, j, 2]
            Z_ij[i, j, 3] = Z_u[i, 1] * Z_d[j, 1]
            Z_ij[i, j, 4] = Z_u[i, 2] * Z_d[j, 2]
        end
    end
    return Z_u, Z_d, Z_ij
end
```

Explanation: The 'generate_covariance' function constructs the covariance matrix Σ_B using parameters σ, ρ_1, ρ_2, and ρ_3. The 'generate_market_data' function generates the Z_{ij} matrices and agent characteristics Z_u and Z_d for each market.

2 Step 2: Solve for Equilibrium Matches

Instruction: Write the linear programming code to solve for the equilibrium matches to a N by matching game as a function of γ, the Zs, and B.

Code Snippet:

```
function create_matching_model(N::Int64)
    model = Model(HiGHS.Optimizer)
    @variable(model, H[1:N, 1:N], Bin)
    @constraint(model, [i=1:N], sum(H[i,j] for j in 1:N) == 1)
    @constraint(model, [j=1:N], sum(H[i,j] for i in 1:N) == 1)
    @objective(model, Min, sum(b_w .* H))
    optimize!(model)
    ...
    return model, H
end

function solve_matching(N::Int64, gamma::Vector{Float64},
        Z_ij::Array{Float64, 3}, B::Array{Float64, 2},
        model::Model, H::Array{VariableRef, 2})
    W_ij = sum(Z_ij .* reshape(gamma, (1, 1, length(gamma))); dims=3)
    ...
    optimize!(model)
    matches = Tuple.(findall(value.(H) .> 0.5))
    return matches
end
```

Explanation: The ‘create_matching_model‘ function sets up the linear programming model using JuMP and HiGHS. The ‘solve_matching‘ function defines the objective based on γ, Z_{ij}, and B, then solves the model to determine equilibrium matches.

3 Step 3: Save Matches and Characteristics

Instruction: Write code to save the matches and the observed match characteristics for each market into a fake dataset.

Code Snippet:

```
function generate_market_replication(r::Int64, t::Int64, N::Int64, K::Int64,
        gamma::Vector{Float64}, chol_sigma_B::Cholesky{Float64, Matrix{Float64}},
        model::Model, H::Matrix{VariableRef})
    ...
    data = DataFrame(
        replication_id = r,
        market_id = t,
        upstream_id = i,
        downstream_id = j,
        Z_u1 = Z_ui[1],
        Z_u2 = Z_ui[2],
        Z_d1 = Z_dj[1],
        Z_d2 = Z_dj[2],
        Z_match1 = Z_ij_match[1],
        Z_match2 = Z_ij_match[2],
        W_ij = W_obs,
        matched = is_matched
    )
    ...
    return data
end

function save_replication_data(data, r::Int)
    ensure_replication_data_folder_exists()
    path = replication_data_path(r)
    CSV.write(path, sort!(data))
    return path
end
```

Explanation: The ‘generate_market_replication‘ function creates a DataFrame containing match information and characteristics for each market. The ‘save_replication_data‘ function saves this DataFrame to a CSV file for later use.

4 Step 4: Generate Simulated $B_{s,t}$ Matrices

Instruction: Write code to generate $S \cdot T$ simulated matrices $B_{s,t}$ of $(N-1)^2$ draws, which are held fixed during each Monte Carlo replication, but vary across replications.

Code Snippet:

```
function simulate_data(theta::Vector{Float64}, S::Int64,
        T::Int64, N::Int64, r::Int64, model::Model, H::Matrix{VariableRef})
    ...
    chol_sigma_B = cholesky(sigma_B; check=true)
    ...
    tasks = [(s, t) for s in 1:S for t in 1:T]
    results = pmap(simulate_market_task, tasks)
    return vcat(results...)
end
```

Explanation: The ‘simulate_data‘ function generates $S \times T$ simulated $B_{s,t}$ matrices by performing Cholesky decomposition on the covariance matrix Σ_B and creating tasks for parallel processing.

5 Step 5: Solve Linear Programs for Given Parameters

Instruction: Write code to solve the linear programming problem for a value of θ, the Zs for a particular market, and a matrix B of $(N-1)^2$ draws.

Code Snippet:

```
function solve_matching(N::Int64, gamma::Vector{Float64},
        Z_ij::Array{Float64, 3}, B::Array{Float64, 2},
        model::Model, H::Array{VariableRef, 2})
    W_ij = sum(Z_ij .* reshape(gamma, (1, 1, length(gamma))); dims=3)
    ...
    optimize!(model)
    matches = Tuple.(findall(value.(H) .> 0.5))
    return matches
end
```

Explanation: The 'solve_matching' function formulates and solves the linear program for given θ, Zs, and B, returning the set of equilibrium matches.

6 Step 6: Parallelize Linear Programs

Instruction: Parallelize the linear programs. Each linear program can be done on its own core using distributed computing.

Code Snippet:

```
using Distributed
addprocs(CPU_cores_to_use)

@everywhere begin
    ...
    function generate_market_simluation(r::Int64, s::Int64, t::Int64, N::Int64, K::Int64,
        gamma::Vector{Float64}, chol_sigma_B::Cholesky{Float64, Matrix{Float64}},
        model::Model, H::Matrix{VariableRef})
    ...
end
....
function simulate_data(theta::Vector{Float64}, S::Int64,
            T::Int64, N::Int64, r::Int64,
            model::Model, H::Matrix{VariableRef})
    ...
    tasks = [(s, t) for s in 1:S for t in 1:T]

    @everywhere begin
        CHOL_SIGMA_B = $chol_sigma_B
        GAMMA = $gamma
        K = length(GAMMA)
        N = $N
        r = $r
        model = $model
        H = $H
    end

    @everywhere begin
        function simulate_market_task(task)
            s = task[1]
            t = task[2]
            return generate_market_simluation(r, s, t, N, K, GAMMA, CHOL_SIGMA_B, model, H)
        end
    end
    results = pmap(simulate_market_task, tasks)
    return vcat(results...)
end
```

Explanation: The script adds worker processes based on available CPU cores and utilizes ‘pmap‘ for parallel execution of linear programs across multiple cores, enhancing computational efficiency.

7 Step 7: Compute Moments and MSM Objective Function

Instruction: Write code to compute the 87 moments described below and the MSM objective function with an identity weighting matrix.

Code Snippet:

```
function compute_market_moments(market_data::DataFrame)
    ...

    ## Quantile Moments for Match-specific Characteristics
    ## Moments 1-10
    moment_index = compute_quantile_moments!(L_match, moment_quantiles,
        moments, moment_index, match_characters)

    ## Quantile Moments for Agent-specific Characteristics
    ## Moments 11-20
    moment_index = compute_quantile_moments!(L_agent, moment_quantiles,
        moments, moment_index, agent_characters)

    ## Correlation Moments
    ## Moments 21-27
    moment_index = compute_correlation_moments!(L_match, L_agent_up,
        L_agent_down, L_agent, moments, moment_index, match_characters,
        agent_characters)

    ## Regression Moments
    ## Moments 28-31
    moment_index = compute_regression_moments!(market_data, moments,
        moment_index, match_characters, agent_characters)

    ## Opportunity Cost Moments for Match-specific Characteristics
    ## Moments 32-51
    moment_index = compute_opportunity_cost_moments_match!(market_data,
        observed_matches, N, moments, moment_index,
        match_characters, moment_quantiles)

    ## Opportunity Cost Moments for Agent-specific Characteristics
    ## Moments 52-71
    moment_index = compute_opportunity_cost_moments_agent!(market_data,
        observed_matches, N, moments, moment_index,
        agent_characters, moment_quantiles)

    ## Regression Moments - Opportunity Cost
    ## Moments 72-75
    moment_index = compute_regression_moments_opportunity_cost!(
        market_data, moments, moment_index, match_characters, agent_characters)

    ## Rank Moments
    ## Moments 76-87
    moment_index = compute_rank_moments!(market_data, observed_matches,
        N, moments, moment_index, match_characters, agent_characters)

    return moments
end
```

Explanation: The ‘compute_market_moments‘ function aggregates various types of moments across markets. Most of the interesting points here are in the section on equations.

8 Step 8: Optimize the Objective Function with Global Solver

Instruction: Write code to use a global optimization solver to optimize the objective function.

Code Snippet:

```
function estimate_theta(data::DataFrame, W::Matrix{Float64},
        theta_initial::Vector{Float64}, N::Int64, T::Int64, S::Int64,
        r::Int64, optimization_settings::Dict, model::Model, H::Matrix{VariableRef})
    ...
    obj(theta) = msm_objective(theta, replication_moments, W, N, T, S, r, model, H)
    res = bboptimize(obj; optimization_settings...)
    theta_hat = best_candidate(res)
    ...
    return theta_hat, solver_info
end

function main(R::Int64, N::Int64, T::Int64, S::Int64,
        theta::Vector{Float64}; regenerate_replication_data::Bool = false)
    ...
    optimization_settings = Dict(
        :Method => :adaptive_de_rand_1_bin_radiuslimited,
        :SearchRange => collect(zip(_SEARCH_RANGE_LB, _SEARCH_RANGE_UB)),
        :MaxSteps => _BBOPTIMIZE_MAX_STEPS,
        :MaxTime => 1000,
        :TraceMode => :compact,
        :PopulationSize => _BBOPTIMIZE_POPULATION_SIZE,
        :NumDimensions => length(_THETA_0)
    )
    ...
end
```

Explanation: The ‘estimate_theta‘ function employs the ‘bboptimize‘ function from the BlackBoxOptim.jl package to globally optimize the MSM objective, returning the estimated parameters and solver information.

9 Step 9: Ensure Solver Reliability

Instruction: Ensure the solver finds the global optimum reliably by adjusting solver settings and verifying outcomes.

Code Snippet:

```
function estimate_theta(...)
    ...
    res = bboptimize(obj; optimization_settings...)
    theta_hat = best_candidate(res)
    solver_info = Dict(
        "stop_reason" => BlackBoxOptim.stop_reason(res),
        "best_fitness" =>  BlackBoxOptim.best_fitness(res),
        "best_candidate" => BlackBoxOptim.best_candidate(res),
    )
    return theta_hat, solver_info
end
```

Explanation: The ‘estimate_theta‘ function captures solver exit reasons and fitness values to assess the reliability of finding the global optimum, enabling adjustments to optimization settings as needed.

10 Step 10: Compute Optimal Weighting Matrix

Instruction: Write code to compute the GMM optimal weighting matrix for a given dataset.

Code Snippet:

```
function compute_optimal_weighting_matrix(replication_data)
    small_delta = 1e-6
    replication_moments = compute_all_markets_moments(replication_data)
    Omega_hat = cov(replication_moments, corrected=false)
    Omega_hat += small_delta * I(size(Omega_hat, 2))
    W = inv(Omega_hat)
    return W
end
```

Explanation: The 'compute_optimal_weighting_matrix' function calculates the optimal weighting matrix W by inverting the covariance matrix of replication moments, ensuring numerical stability with a small regularization term.

11 Step 11: Optimize MSM Objective with Optimal Weighting Matrix

Instruction: Write code to optimize the MSM objective function using the optimal weighting matrix.

Code Snippet:

```
function main(...)
    ...
    W = compute_optimal_weighting_matrix(rep_data)
    theta_initial = theta_true .+ randn(length(theta_true)) * 0.1
    theta_hat, solver_info = estimate_theta(rep_data, W,
        theta_initial, N, T, S, r, optimization_settings, model, H)
    ...
end
```

Explanation: Within the 'main' function, the optimal weighting matrix W is computed and then used in the 'estimate_theta' function to optimize the MSM objective, yielding parameter estimates.

12 Step 12: Compute MSM Variance Matrix and Confidence Intervals

Instruction: Write code to compute the MSM variance matrix and the 95% confidence intervals for each parameter.

Code Snippet:

```
function compute_variance_matrix(theta_hat::Vector{Float64}, data::DataFrame,
        W::Matrix{Float64}, S::Int64, T::Int64, N::Int64, r::Int64,
        h::Vector{Float64}, model::Model, H::Matrix{VariableRef})
    ...
    AVAR = factor * inv_GWG * (G' * W * omega * W * G) * inv_GWG
    std_errors = sqrt.(diag(AVAR))
    z_value = quantile(Distributions.Normal(0, 1), 0.975)
    lower_bounds = theta_hat .- z_value .* std_errors
    upper_bounds = theta_hat .+ z_value .* std_errors
    confidence_intervals = hcat(lower_bounds, upper_bounds)
    return std_errors, confidence_intervals
end
```

Explanation: The ‘compute_variance_matrix‘ function calculates the asymptotic variancecovariance matrix AVAR($\hat{\theta}$), derives standard errors, and constructs 95% confidence intervals based on the normal distribution.

13 Step 13: Loop Over Monte Carlo Replications

Instruction: Wrap the entire procedure inside a loop over Monte Carlo replications, saving necessary outputs for each replication.

Code Snippet:

```
function main(R::Int64, N::Int64, T::Int64, S::Int64,
        theta::Vector{Float64}; regenerate_replication_data::Bool = false)
    ...
    for r in 1:R
        rep_data = load_replication_data(r)
        W = compute_optimal_weighting_matrix(rep_data)
        theta_initial = theta_true .+ randn(length(theta_true)) * 0.1

        theta_hat, solver_info = estimate_theta(rep_data, W,
                theta_initial, N, T, S, r, optimization_settings, model, H)

        std_errs, conf_intervals = compute_variance_matrix(theta_hat,
                rep_data, W, S, T, N, r, h, model, H)

        append!(all_solver_info, DataFrame(solver_info))
        theta_estimates[r, :] = theta_hat
        std_errors[r, :] = std_errs
        confidence_intervals[r, :, :] = conf_intervals
    end
    ...
end
```

Explanation: The ‘main‘ function iterates over R Monte Carlo replications, generating or loading replication data, computing the weighting matrix, estimating parameters, and storing solver information, estimates, standard errors, and confidence intervals for each replication.

14 Step 14: Compute Bias, RMSE, and Coverage

Instruction: At the end of the Monte Carlo study, compute the bias, RMSE, and coverage of your estimates and confidence intervals.

Code Snippet:

```
function compute_statistics(theta_estimates::Array{Float64, 2},
        confidence_intervals::Array{Float64, 3}, theta_true::Vector{Float64})

    bias::Vector{Float64} = vec(mean(theta_estimates, dims=1) - theta_true')
    rmse::Vector{Float64} = vec(sqrt.(mean((theta_estimates .- theta_true').^2, dims=1)))
    coverage = zeros(num_params)
    for i in 1:num_params
        lower_bounds = confidence_intervals[:, i, 1]
        upper_bounds = confidence_intervals[:, i, 2]
        coverage[i] = mean((theta_true[i] .>= lower_bounds) .& (theta_true[i] .<= upper_bounds))
    end
    return bias, rmse, coverage
end

function present_results(...)
    ...
    results_table = DataFrame(
        "Parameter Name" => parameter_names,
        "True Parameter" => theta_true,
        "Bias" => bias,
        "RMSE" => rmse,
        "Coverage (95\%)" => coverage,
        "Step Size" => h,
    )
    ...
    present_results(results_table_settings, theta_true, bias, rmse, coverage, h, "Optimal")
    ...
end
```

Explanation: The ‘compute_statistics‘ function calculates the bias, RMSE, and coverage probabilities for each parameter across all replications. The ‘present_results‘ function then organizes and displays these statistics in a formatted table.

15 Step 15: Report Additional Optimization Studies

Instruction: If you have more to say about optimization routines, feel free to report more Monte Carlo studies.

16 Step 16: Ensure Solver Performance

Instruction: Ensure that your main solver and solver settings work in the sense of having an estimator with low bias, low RMSE, and coverage around 95%.

Code Snippet:

```
function compute_statistics(...)
    ...
    coverage = zeros(num_params)
    for i in 1:num_params
        ...
        coverage[i] = mean((theta_true[i] .>= lower_bounds) .& (theta_true[i] .<= upper_bounds))
    end
    ...
end

function present_results(...)
    ...
    println("Coverage probabilities:")
    println(coverage)
    ...
end
```

Explanation: The script computes coverage probabilities to verify that confidence intervals achieve the desired 95% coverage, ensuring the solver and variance estimation are performing correctly.