Julia BLP

Raymond Chiasson

June 2021

Econometrica, Vol. 63, No. 4 (July, 1995), 841–890

AUTOMOBILE PRICES IN MARKET EQUILIBRIUM

BY STEVEN BERRY, JAMES LEVINSOHN, AND ARIEL PAKES[1]

Summary

Implementation of BLP in Julia

- Extension of assignments in introductory graduate course on empirical IO.
- Basis of a yet-to-be-created beginner tutorial on BLP.
- Code and model are based on Knittel and Metaxoglou 2008.

Improvements on existing code

- Focuses on clarity rather than speed.
- Provides extensive documentation of every calculation.
- Does not use global variables and uses as few custom functions as possible.
- Takes advantage of Julia syntax to write easy-to-read equations.

Julia offers fast and beautiful code

- Has Python's simple syntax for general computing.
- Uses Matlab's syntax for math (inv(X'X), f.(X), A = [B C D], etc.).
- Supports Greek letters, scripts, and subscripts using LaTeX syntax.

Code remains fast and concise

- Less than 100 lines of code to estimate the demand side coefficients.
- Minimizing the objective function takes less than 5 minutes.
- Can be solved with or without using the gradient function.

```
# vector of observations and instruments
IV = [X IV_others IV_rivals]
# restate Y for clarity
Y = log.(share) - log.(outshr)

# estimate 2SLS coefficients
𝒳 = IV*inv(IV'IV)*IV'x₁      # (stage 1)
θ₁_IV = inv(𝒳'𝒳)*𝒳'Y         # (stage 2)
```

```
# 1. Contraction mapping to find δ
Φ(δ) = δ + log.(s) - log.(σ(δ,θ₂,X,v,market_id))
```

```
# GMM objective function value
Q = (Z'ξ)' * W * (Z'ξ)
```

```
# gradient calculation
# ∂Q_∂θ₂ = (2*(Z'ξ)'W) * Z' * -∂σ_∂δ⁻¹ * ∂σ_∂θ₂
∂Q_∂θ₂ = ∂Q_∂ξ * (-∂σ_∂δ⁻¹ * ∂σ_∂θ₂)
```

```
Own price elasticity:
∂σⱼ/∂pⱼ = ∫(α + vᵢₖσᵛₚ) 𝒯ⱼ (1 - 𝒯ⱼ) f(vᵢ)dvᵢ

Cross price elasticity:
∂σⱼ/∂pₘ = ∫ -(α + vᵢₖσᵛₚ) 𝒯ⱼ 𝒯ₘ f(vᵢ)dvᵢ
```

Key Files

		File type	File	Description
Scripts to execute the model estimation	(1)	Script	Demand – OLS and 2SLS	Provides baseline estimates using basic techniques that are expected to be biased. Also calculates BLP instruments.
	(2)	Script	Demand – BLP	Minimizes the BLP objective function to get the demand side parameter estimates. Relies on (4) (5) and (6).
	(3)	Script	Supply	Estimates the pricing equation parameters. Relies on (7) and results from (2).
Modules containing the functions used in model estimation	(4)	Module	BLP_functions	Two functions: the main BLP objective function and a function $\sigma()$ used to estimate the market share of a given product. This implements the essential parts of the BLP method: the contraction mapping and assembly of GMM condition.
	(5)	Module	BLP_derivatives	A function to calculate the gradient of the BLP objective function.
	(6)	Module	BLP_instruments	A function to calculate the table of 10 instruments. Performs the same calculation in (1).
	(7)	Module	supply_price_elasticities	A function to calculate the price elasticities.
Data files	(8)	Data table	BLP_product_data.csv	Observations of all observable variables.
	(9)	Data table	Random_draws_50_individuals.csv ... Random_draws_5000_individuals.csv	Predetermined random draws from standard joint normal distribution. Files to simulate between 50 and 5000 individuals.

Model and Dataset

Original BLP dataset

- Approximately 100 products (cars) observed in 20 markets (20 different years).
- 5 observable characteristics and price.

Separate estimation of demand and supply sides

- Demand side random effects coefficients estimated with the classic BLP method.
- Supply side parameters estimated with OLS.

Key assumptions

- Random coefficients are uncorrelated and normally distributed.
- Uses the "BLP instruments".
- Firms are in equilibrium and set prices simultaneously to maximize (static) profits.
- Constant marginal cost pricing.

Benefits of Julia

Clean syntax

- Similar to Matlab for calculations:
  - inv(X'X) rather than np.linalg.inv(np.dot (X.T, X))
  - Simple broadcasting: f.(X) rather than map(f, X)
- Similar to Python otherwise.
- Implements Unicode characters (Greek letters, script letters, subscripts) using LaTeX syntax.

High computational speed

- Base speed supposedly rivalling C and other fast languages.
- Fast for loops.
- Easy implementation of parallelization.

Open source with rapidly growing user base

- Already has large library of packages.
- Capable of calling packages in other languages.

```
# vector of observations and instruments
IV = [X IV_others IV_rivals]
# restate Y for clarity
Y = log.(share) - log.(outshr)

# estimate 2SLS coefficients
𝑿 = IV*inv(IV'IV)*IV'x₁      # (stage 1)
θ₁_IV = inv(𝑿'𝑿)*𝑿'Y         # (stage 2)
```

```
# 1. Contraction mapping to find δ
Φ(δ) = δ + log.(s) - log.(σ(δ,θ₂,X,v,market_id))
```

```
# GMM objective function value
Q = (Z'ξ)' * W * (Z'ξ)
```

```
# gradient calculation
# ∂Q_∂θ₂ = (2*(Z'ξ)'W) * Z' * -∂σ_∂δ⁻¹ * ∂σ_∂θ₂
∂Q_∂θ₂ = ∂Q_∂ξ * (-∂σ_∂δ⁻¹ * ∂σ_∂θ₂)
```

```
Own price elasticity:
∂σⱼ/∂pⱼ = ∫(α + vᵢₖₚσ'ₚ) 𝒯ⱼ (1 - 𝒯ⱼ) f(vᵢ)dvᵢ

Cross price elasticity:
∂σⱼ/∂pₘ = ∫ -(α + vᵢₖₚσ'ₚ) 𝒯ⱼ 𝒯ₘ f(vᵢ)dvᵢ
```

Results - comparison of minimization algorithms

Method	Nelder-Mead	BFGS	L-BFGS	Gradient Descent	Conjugate Gradient
Seconds run	223	99	148	53	23
Gradient	No	Yes	Yes	Yes	Yes
Objective Function Value	205	205	205	273	260
X tolerance	0.001	0.01	0.01	0.01	0.01
Iterations	397	15	20	8	6
Function calls	672	49	69	27	16
Gradient calls		49	69	27	9
θ_2 guess	Ones	Ones	Ones	Ones	Ones
θ_2 estimates	0.172	0.172	0.172	0.118	0.134
	-2.523	-2.523	-2.523	0.880	0.279
	0.762	0.762	0.762	0.951	0.766
	0.587	0.587	0.587	0.981	0.742
	0.595	0.595	0.595	1.190	1.830

Gradient Descent and Conjugate Gradient did not produce the same estimates despite various starting points and tolerance levels.

Results – number of simulated individuals

N individuals	50	100	200	500	1000	5000
Objective function value	205	182	241	216	206	225
Calls	566	637	671	330	828	846
Iterations	329	370	405	194	500	500
Seconds run	245	434	437	1058	6836	30351
θ_2 estimates	0.17	0.16	0.08	0.16	0.18	0.16
	-2.53	-4.41	0.89	-3.65	4.52	-4.74
	0.76	-2.03	-4.30	-6.63	-6.56	-1.39
	0.59	0.88	0.15	-1.33	0.07	-0.33
	0.60	0.96	0.26	-0.08	-0.34	0.22
θ_1 estimates	-0.43	-0.40	-0.27	-0.47	-0.54	-0.46
	-10.00	-11.31	-9.78	-11.48	-10.76	-13.05
	2.80	3.14	-1.47	0.97	-2.19	2.68
	1.10	1.05	1.01	0.99	1.84	1.53
	-0.43	-1.01	0.15	0.10	-0.13	0.07
	2.79	2.52	2.84	2.97	3.00	2.99

Considerable variation and lack of convergence in parameter estimates

Estimates somewhat robust to more simulated individuals

Criticisms and Future Work

Code is slower than Matlab code from Knittel and Metaxoglou 2008

- Achieves speed by relying on global variables and using computational steps that make the underlying calculation hard to follow.

Separate demand and supply side estimates

- Leads to some negative marginal cost estimates which must be dropped in order to estimate supply side parameters.

I do not provide a calculation for the Hessian nor for standard errors.

I may eventually add simultaneous estimation of parameters, a hessian function, and a standard error calculation in the future.

Code Snippet Comparisons

Matlab (white) versus Julia (black)

Matlab code from Knittel and Metaxoglou 2008

Contraction mapping to find δ

main.m | gmmobj2.m | meanval.m | mktsh.m | ind_sh.m | mufunc.m

```
function f = meanval(theta2)

global x2 s_jt mvalold oldt2 mvalolds mval_track

% s_jt is vector of observed shares

if max(abs(theta2-oldt2)) < 0.01;
    tol = 1e-9;
    flag = 0;
else
    tol = 1e-6;
    flag = 1;
end

theta2w = zeros(5,5); theta2w(:,1)=theta2;
expmu = exp(mufunc(x2,theta2w));
norm = 1;
avgnorm = 1;

i = 0;
mvalolds=[mvalold];

% this is the contraction mapping
while norm > tol*10^(flag*floor(i/50)) && avgnorm > 1e-3*tol*10^
%while (norm > 1e-16 && i<=5000)
      mval = mvalold.*s_jt./mktsh(mvalold,expmu); % market share
      t = abs(mval-mvalold);
      norm = max(t);
      avgnorm = mean(t);
      mvalold = mval;
      i = i + 1;
end

if flag == 1 && max(isnan(mval)) < 1;
   mvalold = mval;
   oldt2 = theta2;
end

f = log(mval);

mvalolds=[mvalolds,mvalold];
mval_track=[mval_track,mvalold];
```

```
function demand_objective_function(θ₂,X,s,Z,v,market_id)

# initialze δ
δ = zeros(size(s))

# 1. Contraction mapping to find δ
# set up contraction mapping for δ.
# use first output of σ() which is the predicted share. second output is th
Φ(δ) = δ + log.(s) - log.(σ(δ,θ₂,X,v,market_id)[1])

# contraction mapping parameters
tolerance = 1e-6                    # Matlab code uses 1e-6 or 1e-9
largest_dif = tolerance + 1         # track difference from first value.
max_iterations = 1000               # reasonable limit
counter = 0                         # track iterations

while (largest_dif > tolerance)
    # recalculate delta
    δ = Φ(δ)
    # check if all estimates (elements) are within tolerance
    largest_dif = maximum(abs.( δ - Φ(δ) ))
    # check if max iterations is exceeded
    counter += 1
    if counter == max_iterations
        break
    end
end

# get the shares for each individual to use for calculating the gradient.
# vector of 2217 products x 50 individuals
𝓘 = σ(δ,θ₂,X,v,market_id)[2]
```

Objective function gradient

```
function df = gradobj(theta2)
% calculates and returns gradient object.

global invA1 IV1 ns cdid cdindex x2 v gmmresid mvalold

mval=mvalold; % delta

theta2w = zeros(5,5);
theta2w(:,1)=theta2; % theta2 value

expmu = exp(mufunc(x2,theta2w));
shares = ind_sh(mval,expmu);
clear expmu

% f1 seems to be (partial sigma)/(partial theta2)
f1 = zeros(size(cdid,1),size(x2,2));

for ii=1:size(x2,2)
    xv = (x2(:,ii)*ones(1,ns)).*v(cdid,ns*(ii-1)+1:ns*ii);
    temp = cumsum(xv.*shares);
    suml = temp(cdindex,:);
    suml(2:size(suml,1),:) = diff(suml);
    f1(:,ii) = mean((shares.*(xv-suml(cdid,:)))')';
    clear xv temp suml
end
rel=size(theta2,1);

% f seems to be inv(partial sigma/partial delta)*f1
f = zeros(size(cdid,1),rel);
n = 1;
for i = 1:size(cdindex,1)
    temp = shares(n:cdindex(i),:);
    H1 = temp*temp';
    H = (diag(sum(temp')) - H1)/ns;
    f(n:cdindex(i),:) = - inv(H)*f1(n:cdindex(i),1:rel);
    n = cdindex(i) + 1;
end

df = 2*f'*IV1*invA1*IV1'*gmmresid; % gradient of gmm objective function
```

```
# 1 - Take derivative of Q with respect to ξ
# ∂Q/∂θ₂ = ∂Q/∂ξ * ∂ξ/∂θ₂
W = inv(Z'Z)
∂Q_∂ξ = 2*(Z'ξ)'W*Z'


# 2 - find the derivatives ∂σ/∂δ and ∂σ/∂θ₂

    # 2.1 ∂σ/∂δ:
    # array to store the derivatives
    # each slice [:,:,i] is the 2217x2217 matrix for individual i
    ∂σᵢ_∂δ = zeros(n_products, n_products, n_individuals)

    # get the index of the diagonal for slice of ∂σᵢ_∂δ[:,:,individual]
    diagonal_index = CartesianIndex.(1:n_products, 1:n_products) # object of (1,1) (2,2) ..

    # calculate the derivative given 𝓘(j,i) values from objective function
    for individual in 1:n_individuals

        # derivative for off-diagonal elements: -𝓘ⱼᵢ * 𝓘ₘᵢ
        ∂σᵢ_∂δ[:,:,individual] = -𝓘[:,individual] * 𝓘[:,individual]'

        # derivative for diagonal elements: 𝓘ⱼᵢ * (1 - 𝓘ⱼᵢ)
        ∂σᵢ_∂δ[diagonal_index, individual] = 𝓘[:,individual] .* (1 .- 𝓘[:,individual])

    end

    # calculate mean over all individuals (Monty Carlo integration)
    ∂σ_∂δ = mean(∂σᵢ_∂δ, dims=3)[:,:] # semicolon to remove the dropped third dimension

    # calculate inverse
    ∂σ_∂δ⁻¹ = zeros(size(∂σ_∂δ))
    # must be done market-by-market: products outside of given market do not affect shares
    for market in unique(market_id)
        ∂σ_∂δ⁻¹[market_id.==market, market_id.==market] = inv(∂σ_∂δ[market_id.==market, mar
    end
```

Calculate price elasticities and marginal costs

```
function calc_mc(incomeDif, p_ijt, params)
    mc_all = zeros(delta_0)
    alpha = abs(params[1])
    s_jt = p_ijt * unobs_weight' #/ ns

    for m in marks
        firm_yr = firms[find(markets .== m),:]
        price = p[find(markets .== m)]
        income = incomeDif[find(markets .== m),:]
        sameFirm = convert(Array{Float64, 2}, firm_yr .== firm_yr')
        yr = p_ijt[find(markets .== m),:]

        nobs = size(yr)[1]
        grad = zeros(nobs, nobs)
        for i=1:ns
            grad .+= alpha ./ income[:,i] .* sameFirm .* unobs_weight[i] .* (yr[:,i].*yr[:,i]' - diagm(yr[
        end
        subMatrix = - grad #/ ns
        b = inv(subMatrix) * s_jt[find(markets .== m),:]

        mc = price - b
        mc[mc.<0] = .001
        mc_all[find(markets .== m), :] = mc
    end
    return mc_all
end
```

```
# loop through all products
Threads.@threads for j in 1:n_products # run loop in parallel with Threads. reduced time ~75x.

    # get market id for product j
    market = market_id[j]

    # get observables and indiviudals
    xⱼ = X[j,:]                    # observables for product j
    xₘ = X[market_id.==market,:]   # observables of all products in market with product j
    vₘ = v_diag[market,:,:]        # matrix of 5000x5 pre-selected random draws (=> 5000 individuals)

    # function defining the interior of the sigma function integral
    𝒯(vᵢ) = exp(xⱼ'θ₁ + xⱼ[Not(6)]'*(θ₂.*vᵢ)) / (1 + sum(exp.(xₘ*θ₁ + xₘ[:,Not(6)]*(θ₂.*vᵢ))))

    # interior of the own price elasticity function
    integral_interior(vᵢ) = (α + vᵢ[1]*σᵛₚ) * 𝒯(vᵢ) * (1 - 𝒯(vᵢ))

    # estimate with Monty Carlo integration over all individuals in vₘ
    # integral_interior() is applied to each of the ~5000 sets of 5 vᵢ values in vₘ
    ∂σⱼ_∂pⱼ = mean(integral_interior.(vₘ))
    # equivalently: ∂σⱼ_∂pⱼ = sum(integral_interior.(vₘ)) * 1 / length(vₘ)

    # assign own price elasticitiy to matrix of price elasticities (along the diagonal)
    Δ[j,j] = -∂σⱼ_∂pⱼ

end
```

Parallelization in Python versus Julia

Julia

- Add "Threads.@threads" decorator to any suitable for loop.
- Part of the base implementation
- No modification of the for loop

```julia
# loop through all products
Threads.@threads for j in 1:n_products
```

Python

- Import a dedicated package
- Enclose the loop in a function
- Re-write the loop to work element-by-element

```python
import multiprocessing
from joblib import Parallel, delayed

# key loop for the function call
results = Parallel(n_jobs=num_cores) (delayed(extract_text_parallel)(i, drug_list, transcripts_folder) for i in inputs)
```