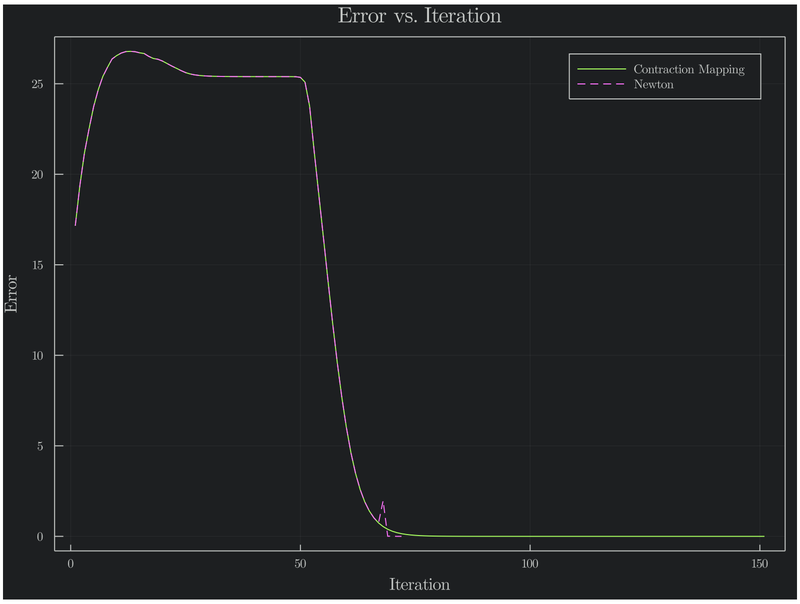
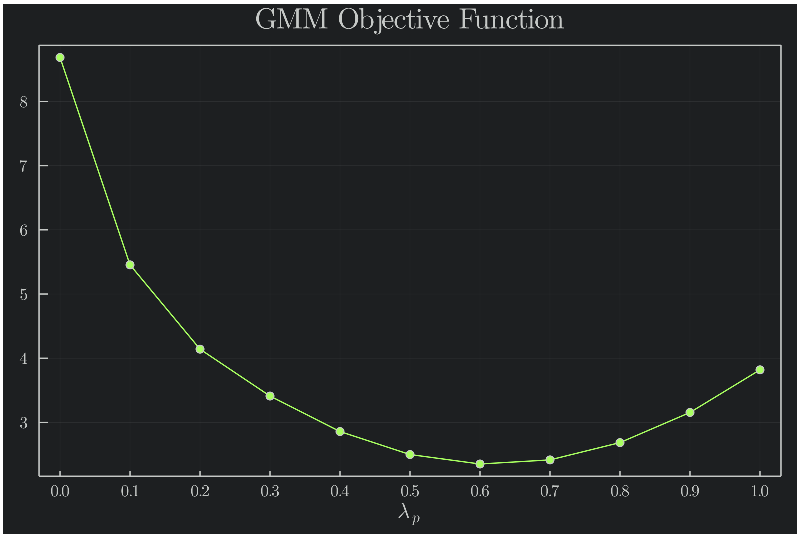
The code used to complete this problem set is attached in the appendix below.[1]

1. Problem 1: See inverse_demand() in functions.jl. The function converges in 73 iterations, and the errors are plotted below.



2. Problem 2: The GMM objective function using the 2SLS weighting matrix is plotted below for $\lambda_p \in [0, 1]$. As you can see, the function is smooth, continuous, and concave with a unique minimum in this range.

[1]We gratefully acknowledge a conversation with Michael Nattinger that helped us to find an error in applying the Newton method.



3. Problem 3: Minimizing the GMM objective function to estimate λ with 2-step GMM yields an estimate of $\lambda_p = 0.564$.

Appendix

The first codefile named “runfile.jl" runs the code.

```
#==
    This file conducts the analyses for JF's PS3
==#

# Load the necessary packages
using StatFiles, DataFrames, Plots, LaTeXStrings, Printf
theme(:juno)
# Un-comment the following line when compiling the document
# theme(:vibrant)
default(fontfamily = "Computer Modern", framestyle = :box) # LaTex-style


# Indlude the functions
include("./functions.jl")
include("./manipulate_data.jl")
include("./aux_vars.jl")

#car_data, instruments, income = load_data("./PS3b/data/")
# car_data, instruments, income = load_data("../data/")
car_data, instruments, income = load_data("data")



##############################################################################
####                            Problem 1
##############################################################################
model = construct_model(model_specs, car_data, instruments, income)
market, λₚ = 1985, 0.6

err_list_cm = inverse_demand(model, λₚ, market; method = "Contraction Mapping", max_iter =
Inf)
# No longer need to reset the model. Funcion will recalculate δ from scratch if not told otherwise
err_list_n = inverse_demand(model, λₚ, market; method = "Newton", max_iter = Inf)

plot(err_list_cm[2:end], xlabel = "Iteration", size = (800, 600),
    ylabel = "Error", title = "Error vs. Iteration", label = "Contraction Mapping")
plot!(err_list_n[2:end], line = (:dash), label = "Newton")
    #savefig("./PS3b/Document/Figures/Problem1.pdf")
    savefig("Document/Figures/Problem1.pdf")


# markets = unique(car_data.Year)

# err_list = Dict()
# for market in markets
#     err_list[market] = inverse_demand(model, λₚ, market)
# end
# err_list

# p = plot([err_list[market][2:end] for market in markets], label="")
# scatter(err_list[2000][2:end])

#inverse_demand(model, λₚ, 1992)

##############################################################################
####                            Problem 2
##############################################################################


grid = 0:0.1:1
```

```
#? Experimemt
# I'll try different values of the parameter and check if using previous demand estimates improves speed of con

model = construct_model(model_specs, car_data, instruments, income)

num_iter_recalc = []
for λ ∈ grid
    err = inverse_demand(model, λ, market; recalc_δ = true)
    push!(num_iter_recalc, length(err))
end

num_iter_no_recalc = []
for λ ∈ grid
    err = inverse_demand(model, λ, market; recalc_δ = false)
    push!(num_iter_no_recalc, length(err))
end


sum(num_iter_recalc) / length(num_iter_recalc)
sum(num_iter_no_recalc) / length(num_iter_no_recalc)

#? Experimemt
# What is faster Newton or Contraction Mapping?
using BenchmarkTools
@btime begin
    model = construct_model(model_specs, car_data, instruments, income)
    for λ ∈ grid
        err = inverse_demand(model, λ, market; recalc_δ = false, method = "Newton")
    end
end

@btime begin
    model = construct_model(model_specs, car_data, instruments, income)
    for λ ∈ grid
        err = inverse_demand(model, λ, market; recalc_δ = false, method = "Contraction Mapping")
    end
end

function ReturnData(model, grid)
    data = []
    for λ in grid
        println(λ)
        data = push!(data, gmm(model, λ))
    end
    return data
end

data = ReturnData(model, grid)

plot(grid, data, title = "GMM Objective Function", xlabel = L"\lambda_{p}", legend =
false, markershape = :auto)
xticks!(grid)
#savefig("./PS3b/Document/Figures/Problem2.pdf")
savefig("Document/Figures/Problem2.pdf")

##############################################################################
####                            Problem 3
##############################################################################

λhat_GMM = TwoStage_gmm(model)

# print result
#fname = "./PS3b/Document/Problem3.tex";
fname = "Document/Problem3.tex";
open(fname, "w") do io
    str = @sprintf "\$\\lambda_p=%1.3f\$" λhat_GMM[1]
    write(io, str)
```

```
end;



#==
##Testing
opt = optimize(λ -> gmm(model, λ[1]), [.6], method = BFGS(), f_tol = 1e-5, g_tol = 1e-5)
    λ_hat = opt.minimizer[]
    ξ_hat=gmm(model, λ_hat, Return_ρ=true)
    OptimalW=inv( (model.Z .* ξ_hat)*transpose(model.Z .* ξ_hat) )
    λ_hat_2=gmm(model, λ_hat,SpecifyW=true,SpecifiedW=OptimalW)


Z = model.Z

temp = (Z' * ξ_hat) * (Z' * ξ_hat)'



det(temp .+ rand(0.9:0.001:1.1, size(temp)))

size(temp .+ rand(0.9:0.001:1.1, size(temp)))



OptimalW=inv( inv(temp .+ rand(0.9:0.001:1.1, size(temp))) )

Z

Zopt_scondstage = optimize(λ -> gmm(model, λ[1],SpecifyW=true,SpecifiedW=OptimalW),
                [λ_hat], method = BFGS(), f_tol = 1e-5, g_tol = 1e-5).minimizer

TwoStage_gmm(model)
==#
```

The next codefiles, named “functions.jl,” “manipulate_data.jl,” and “aux_vars.jl,” are referenced by “runfile.jl.”

```
# Load packages
using LinearAlgebra, Parameters, Optim

# Auxiliary functions
# Choice probability function
function choice_probability(δ::Array{Float64}, μ::Array{Float64}; eval_jacobian::Bool =
false)

    # number of individuals and choicesm
    J, R = size(μ)

    # Compute choice probabilities
    Λ = exp.(δ .+ μ)
    Σ = Λ ./ (1 .+ sum(Λ, dims = 1))
    σ = sum(Σ, dims = 2) / R

    if eval_jacobian
        # Compute Jacobian
        Δ = (1 / R) * ((I(J) .* (Σ * (1 .- Σ)')) - ((1 .- I(J)) .* (Σ * Σ'))) ./
σ
        return σ, Δ
    else
        return σ, nothing
    end
```

```
end

# segment_data by market for demand estimation
function segment_data(model, market)

    # Get market id column
    market_id_col = model.market_id

    # Filter data by market
    data = model.inv_dem_est[model.inv_dem_est[!, market_id_col] .== market, :]

    # Get the observed market shares
    S = data.share
    # Get the observed prices
    P = data.price
    # Get the income levels
    Y = model.Y
    # Get the inital guess for the inverse demand
    δ = data.δ

    return S, P, Y, δ
end

# Model Structures
# Primitives
@with_kw struct Primitives
    λₚ_range :: Array{Float64} = [0, 1]
end


# Model
mutable struct Model
    # Parameters
    parameters      ::Primitives                # Parameters of the model

    # Data
    market_id       :: Any                      # Market id column
    product_id      :: Any                      # Product id column
    X               :: Array{Float64, 2}        # Matrix of covariates
    Z               :: Array{Float64, 2}        # Matrix of instruments
    Y               :: Array{Float64, 2}        # Matrix of simulated data
    inv_dem_est     :: DataFrame                # DataFrame of for demand estimation

    # GMM estimation
    ρ               :: Array{Float64}           # GMM Residuals
end



# Demand inverter
function inverse_demand(model::Model, λₚ::Float64, market; method::String="Newton", max_iter =
Inf, verbose::Bool = false, recalc_δ::Bool = true)
    if method=="Newton" #Otherwise this prints too often when we do the contraction mapping
        print("\n")
        print("Market: $(market)")
        print("\n")
    end
    # Check the method
    valid_methods = ["Newton", "Contraction Mapping"]
    @assert (method ∈ valid_methods)

    # Get the data
    S, P, Y, δ = segment_data(model, market)

    if recalc_δ
        # Recalculate the initial guess for the inverse demand
```

```
        δ = zeros(size(δ))
    end

    # Compute the matrix μ[i,j] = λ_p * Y[i] * P[j]
    μ = λ_p * repeat(Y', length(S), 1) .* P

    # Compute the inverse demand

    # Initial guess for the inverse demand
    δ_0 = copy( δ )
    δ_1 = copy( δ )

    # Initialize the iteration counter
    iter = 0

    # Initialize the error
    err = 100
    eval_jacobian = false

    ε = 1e-12
    ε_1 = ( method == "Newton" ) ? 1 : -Inf

    # Iterate until convergence

    err_list = []
    method_flag = "Contraction Mapping"

    while (err > ε) && (iter < max_iter)
        # Compute the choice probability and the Jacobian
        if (method == "Newton") && (err < ε_1)
            eval_jacobian = true
            method_flag = "Newton"
        else # This will bring it back to contraction mapping if it diverges
            eval_jacobian = false
            method_flag = "Contraction Mapping"
        end

        σ, Δ = choice_probability(δ_0, μ, eval_jacobian=eval_jacobian)

        # Compute the inverse demand
        if (method == "Newton") && (err < ε_1)
            #I added the ./S after talking with Michael Nattinger
            #It also lines up with JF's ox code in blp_func_ps.ox
            #(From Danny) JF divides by "Shat", or predicted shares,
            # aka σ
            #δ_1 = δ_0 + inv(Δ ./ S) * (log.(S) - log.(σ))
            δ_1 = δ_0 + inv(Δ) * (log.(S) - log.(σ))
        else
            δ_1 = δ_0 + log.(S) - log.(σ)
        end

        # Update the error
        #err = maximum( abs.(δ_1 - δ_0) )
        err = norm(δ_1 - δ_0)

        push!(err_list, err)
        # Update the inverse demand
        δ_0 = copy(δ_1)

        # Update the iteration counter
        iter = iter + 1
        if (iter % 10 == 0) && verbose
            println("Iteration = $iter, Method = $method_flag , error = $err, tolerance = $ε, error > tolerance

        end

    end
    # println("Iteration = $iter, Method = $method_flag, error = $err, tolerance = $ε, error > tolerance = $(er
```

```
= $θ")
    market_id_col = model.market_id
    model.inv_dem_est[model.inv_dem_est[!, market_id_col] .== market, :δ] .= δ₁[:,
1]

    return err_list

end


function gmm(model, λ; Return_ρ::Bool=false, SpecifyW::Bool=false,
        SpecifiedW::Array{Float64, 2}=zeros(2,2))
    markets = unique(model.inv_dem_est[!, model.market_id])
    for  market in markets
        inverse_demand(model, λ, market, method = "Contraction Mapping")
    end

    # Iv regression
    X = model.X
    Z = model.Z
    W = inv(Z'Z)
    if SpecifyW
        W=SpecifiedW
    end
    δ = model.inv_dem_est.δ
    β_iv = inv((X'Z)*W*(Z'X))*(X'Z)*W*(Z'δ)

    ρ = (δ - X*β_iv)
    if ~Return_ρ
        return ρ'Z*W*Z'*ρ/100
    else
        return ρ/100
    end
end


function TwoStage_gmm(model)
    λhat = optimize(λ -> gmm(model, λ[1]), [.6],
                method = BFGS(), f_tol = 1e-5, g_tol = 1e-5).minimizer
    ξhat=gmm(model, λhat[1],Return_ρ=true)
    #OptimalW=inv( (model.Z * ξhat)*transpose(model.Z * ξhat) )
    #Maybe it ought to be this?
    OptimalW=inv( (model.Z .* ξhat)'*(model.Z .* ξhat) )
    #OptimalW=inv( dot(model.Z, ξhat)transpose(dot(model.Z, ξhat) ))
    #print(OptimalW)
    print(λhat)
    λhat_SecondStage=optimize(λ -> gmm(model, λ[1],SpecifyW=true,SpecifiedW=OptimalW),
               λhat, method = BFGS(), f_tol = 1e-5, g_tol = 1e-5).minimizer
    return λhat_SecondStage
end
```

```
# Load packages
using StatFiles, DataFrames

include("./functions.jl")

# Function to load data
function load_data(path_to_dir::String)
    # Load the data
    # car characteristics
    car_data = DataFrame(StatFiles.load(path_to_dir*"/Car_demand_characteristics_spec1.dta"))
    dropmissing!(car_data)
    car_data[!, :Year] = Int.(car_data.Year)
```

```
    car_data[!, :Model_id] = Int.(car_data.Model_id)
    # Create unique identifier for each product_marker
    car_data[!, :id] = string.(car_data[!, :Year]) .* "_" .* lpad.( car_data[:, :Model_id], 4, "0")
    # Sort by id
    sort!(car_data, [:id])
    # instruments
    instruments = DataFrame(StatFiles.load(path_to_dir*"/Car_demand_iv_spec1.dta"))
    dropmissing!(instruments)
    instruments[!, :Year] = Int.(instruments.Year)
    instruments[!, :Model_id] = Int.(instruments.Model_id)
    # Create unique identifier for each product_marker
    instruments[!, :id] = string.(instruments[!, :Year]) .* "_" .* lpad.( instruments[:, :Model_id], 4, "0")
    # Sort by id
    sort!(instruments, [:id])
    # simulated income (random coefficient)
    income = DataFrame(StatFiles.load(path_to_dir*"/Simulated_type_distribution.dta"))
    dropmissing!(income)

    return car_data, instruments, income
end

# Function to construct model from data
function construct_model(model_specs::Dict, car_data::DataFrame, instruments::DataFrame, income::DataFrame)


    parameters = Primitives()

    # Get id's of the products and markets
    market_id = model_specs[:ids][:market]
    product_id = model_specs[:ids][:product]


    # Create co-variate matrix
    X = car_data[!, model_specs[:covariates]] |> Matrix

    # Create exogenous variables matrix
    Z_exo = car_data[!, model_specs[:exogenous]] |> Matrix

    # Create instruments matrix
    Z_inst = instruments[!, model_specs[:instruments]] |> Matrix

    # Merge exogenous and instruments
    Z = hcat(Z_exo, Z_inst)

    # Create income vector
    Y = income[:, model_specs[:sim_data]] |> Matrix

    # Create a DataFrame for inverse demand estimation

    inv_dem_est = car_data[!, [:id, :Year, :share, :price]]

    # Initial guess for the inverse demand
    inv_dem = zeros(size(car_data)[1])

    inv_dem_est[:, :δ] = inv_dem

    # Initialize GMM Residuals
    gmm_residuals = zeros(size(car_data)[1])

    return  Model(parameters, market_id, product_id, X, Z, Y, inv_dem_est, gmm_residuals)
end
```

```
# Identifiers
```

```
indentifiers = Dict(:market => :Year, :product => :id)

# Name of covariates to use for the regression
covariates_names = [:price,
                    :dpm,
                    :hp2wt,
                    :size,
                    :turbo,
                    :trans,
                    :Year_1986,
                    :Year_1987,
                    :Year_1988,
                    :Year_1989,
                    :Year_1990,
                    :Year_1991,
                    :Year_1992,
                    :Year_1993,
                    :Year_1994,
                    :Year_1995,
                    :Year_1996,
                    :Year_1997,
                    :Year_1998,
                    :Year_1999,
                    :Year_2000,
                    :Year_2001,
                    :Year_2002,
                    :Year_2003,
                    :Year_2004,
                    :Year_2005,
                    :Year_2006,
                    :Year_2007,
                    :Year_2008,
                    :Year_2009,
                    :Year_2010,
                    :Year_2011,
                    :Year_2012,
                    :Year_2013,
                    :Year_2014,
                    :Year_2015,
                    :model_class_2,
                    :model_class_3,
                    :model_class_4,
                    :model_class_5,
                    :cyl_2,
                    :cyl_4,
                    :cyl_6,
                    :cyl_8,
                    :drive_2,
                    :drive_3,
                    :Intercept]


exo_varlist         =[:dpm,
                     :hp2wt,
                     :size,
                     :turbo,
                     :trans,
                     :Year_1986,
                     :Year_1987,
                     :Year_1988,
                     :Year_1989,
                     :Year_1990,
                     :Year_1991,
                     :Year_1992,
                     :Year_1993,
                     :Year_1994,
                     :Year_1995,
```

```
                      :Year_1996,
                      :Year_1997,
                      :Year_1998,
                      :Year_1999,
                      :Year_2000,
                      :Year_2001,
                      :Year_2002,
                      :Year_2003,
                      :Year_2004,
                      :Year_2005,
                      :Year_2006,
                      :Year_2007,
                      :Year_2008,
                      :Year_2009,
                      :Year_2010,
                      :Year_2011,
                      :Year_2012,
                      :Year_2013,
                      :Year_2014,
                      :Year_2015,
                      :model_class_2,
                      :model_class_3,
                      :model_class_4,
                      :model_class_5,
                      :cyl_2,
                      :cyl_4,
                      :cyl_6,
                      :cyl_8,
                      :drive_2,
                      :drive_3,
                      :Intercept]

# Name of the instruments
instruments_names = [:i_import,
                     :diffiv_local_0,
                     :diffiv_local_1,
                     :diffiv_local_2,
                     :diffiv_local_3,
                     :diffiv_ed_0]

sim_data_names = [:Var1]

model_specs = Dict( :ids         => indentifiers,
                    :covariates  => covariates_names,
                    :exogenous   => exo_varlist,
                    :instruments => instruments_names,
                    :sim_data    => sim_data_names)
```