Tasks:

Task 1

Solve both versions of the model presented above. Use the parameter values in the calibration section of section 1 for both versions.

Answer: Our code is attached in the appendix. ∎

Task 2

Compute the following model moments and fill in the table. Are they any different across model specifications? If yes, try to explain intuitively what drives the differences.

Answer: The table is the following.

variable	Standard	TV1 Shock $\alpha = 2.0$	TV1 Shock $\alpha = 3.0$	TV1 Shock $\alpha = 1.0$
Price Level	0.739	0.725	0.728	0.713
Mass of Incumbents	8.321	8.829	8.207	10.336
Mass of Entrants	2.639	3.033	3.218	2.756
Mass of Exits	1.662	2.132	2.09	2.132
Aggregate Labor	179.834	183.024	181.198	187.775
Labor of Incumbents	142.627	142.497	137.744	152.541
Labor of Entrants	37.207	40.527	43.455	35.234
Fraction of Labor Entrants	0.207	0.221	0.24	0.188

∎

Task 3

Plot the decision rules of exit in all model specifications you have solved. Are they any different? If yes, try to explain intuitively what drives the differences.

Answer: Figure 1 displays the results. ∎

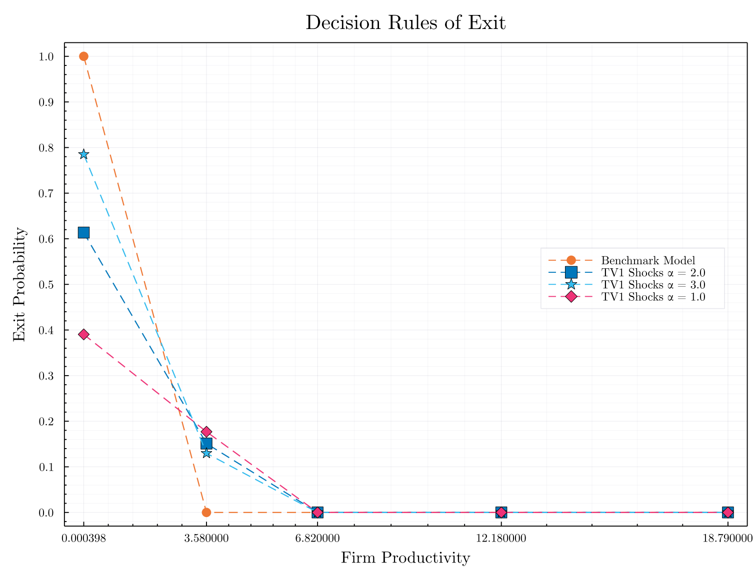

Figure 1: Decision Rules across Model Specifications

Task 4

How does the exit decision rule change if cf rises from 10 to 15?

Answer:

-

Appendix

The first code file runs the code.

```
# Loading Packages
using Parameters, LinearAlgebra, Plots, Latexify, DataFrames, LaTeXStrings

# Loadifng Programs
include("./hopenhayn_rogerson.jl")

# Set plot theme
theme(:vibrant)
default(fontfamily="Computer Modern", framestyle=:box) # LaTex-style

# Initialize the model's parameters and results struct
prim, res = Initialize();

# Create the structure for experiments 1 and 2 with random disturbances
_, res_1 = Initialize();
_, res_2 = Initialize();
_, res_3 = Initialize();

# First we solve for the case with no random disturbances
solve_model_no_dist(prim, res)

# and then we add random disturbances to the model
# Fist we will create a dictionary of that will index the result structure with the random disturbance
results = Dict(0.0 => res, 1.0 => res_1, 2.0 => res_2, 3.0 => res_3 ) # α = 0.0 means no disturbances
# then we iterate over the random disturbances and solve the model
for (α, res_struct) in results
    if α  != 0.0
        find_equilibrium(prim, res_struct, α)
    end
end


# Plot the results
p2 = plot(prim.s_vals, res.x_opt, size=(800,600),
            title="Decision Rules of Exit", label="Benchmark Model",
            linestyle =:dash, markershape = :auto, legend = :right)
xticks!(prim.s_vals)
yticks!(0:0.1:1)
xlabel!("Firm Productivity")
ylabel!("Exit Probability")


for (α, res_struct) in results
    if α  != 0.0
        plot!(prim.s_vals, res_struct.x_opt, size=(800,600),
                title="Decision Rules of Exit", label="TV1 Shocks α = $α ",
                linestyle =:dash, markershape = :auto)
    end
end
current()

# savefig(p2, "./PS6/Document/Figures/decision_rules_2.pdf")
savefig(p2, "../Document/Figures/decision_rules_2.pdf")

# Save results to a table
## Error for i.
n_opt = [prim.n_optim.(prim.s_vals, r.p) for (_, r) in results]
n_incumbents = [ prim.n_optim.(prim.s_vals, r.p)' * r.μ for (_, r) in results]
n_entrants = [ prim.n_optim.(prim.s_vals, r.p)' * prim.ν * results[2].M for (_, r) in results]
n_total = [n_incumbents[i] + n_entrants[i] for i in 1:length(n_incumbents) ]
fraction_labor_entrants = [n_entrants[i] ./ n_total[i] for i in 1:length(n_incumbents) ]

df = DataFrame(["Price Level" => [r.p for (_, r) in results]
                "Mass of Incumbents" => [sum(r.μ) for (_, r) in results]
                "Mass of Entrants" => [sum(r.M .* prim.ν) for (_, r) in results]
                "Mass of Exits" => [sum(r.μ .* r.x_opt) for (_, r) in results]
                "Aggregate Labor" => n_total
                "Labor of Incumbents" => n_incumbents
                "Labor of Entrants" => n_entrants
                "Fraction of Labor Entrants" => fraction_labor_entrants]
                )
                df = round.(df, digits=3)
                colnames = names(df)
                df[!, :id] = [( α == 0.0 ) ?  "Standard" : "TV1 Shock α = $α" for (α,
```

```
_) in results]
df = unstack(stack(df, colnames), :variable, :id, :value)

laetx_table = latexify(df; env=:table, latex=false)

open("./PS6/Document/Tables/table_1.tex", "w") do file
    write(file, laetx_table)
#open("../Document/Tables/table_2.tex", "w") do file
#    write(file, laetx_table)

end
```

The second code file contains the relevant functions.

```
using Parameters, LinearAlgebra

# Structure that holds the parameters for the model
@with_kw struct Primitives
    β           ::Float64          = 0.8
    θ           ::Float64          = 0.64
    s_vals      ::Array{Float64}   = [3.98e-4, 3.58, 6.82, 12.18, 18.79]
    nS          ::Int64            = length(s_vals)
    x_vals      ::Array{Int64}     = [0, 1]
    emp_lev     ::Array{Float64}   = [1.3e-9, 10, 60, 300, 1000]
    trans_mat   ::Array{Float64,2} =[0.6598  0.2600  0.0416  0.0331  0.0055 ;
                                     0.1997  0.7201  0.0420  0.0326  0.0056 ;
                                     0.2000  0.2000  0.5555  0.0344  0.0101 ;
                                     0.2000  0.2000  0.2502  0.3397  0.0101 ;
                                     0.2000  0.2000  0.2500  0.3400  0.0100]
    ν           ::Array{Float64}   = [0.37, 0.4631, 0.1102, 0.0504, 0.0063]
    A           ::Float64          = 1/200
    c_f         ::Int64            = 10
    c_e         ::Int64            = 5

    # Price grid
    p_min       ::Float64 = 0.01
    p_max       ::Float64 = 3.0
    # nP          ::Int64   = 10
    # p_grid      ::Array{Float64}   = range(p_min, stop = p_max, length = nP)

    # Optimal decision rules
    n_optim     ::Function         = ( s , p ) -> (θ * p * s) ^ (1/(1 - θ))

    Π           ::Function         = ( s , p,  n ) -> (n > 0) ? p*s*( n )^θ - n
- p * c_f : -p * c_f

    # Limits for the mass of entrants
    M_min       ::Float64 = 1.0
    M_max       ::Float64 = 10.0


end

# Structure that stores results
mutable struct Results
    W_val   ::Array{Float64}         # Firm value given state variables
    n_opt   ::Array{Float64}         # Optimal labor demand for each possible state
    x_opt   ::Array{Float64}           # Optimal firm decicion for each state
    p       ::Float64                # Market clearing price
    μ       ::Array{Float64}         # Distribution of Firms
    M       ::Float64                # Mass of entrants
end

# Initialize model
function Initialize()
    prim  = Primitives()

    W_val = zeros(prim.nS)
    n_opt = zeros(prim.nS)
    x_opt = zeros(prim.nS)
    p = (prim.p_max + prim.p_min)/2
    μ = ones(prim.nS) / prim.nS # Uniform distribution is the initial guess
    M = 5.0

    res = Results(W_val, n_opt, x_opt, p, μ, M)

    return prim, res
```

```
end

# Bellman operator for W
function W(prim::Primitives, res::Results)
    @unpack Π, n_optim,s_vals, nS, trans_mat, c_f, β = prim
    @unpack p = res

    temp_val = zeros(size(res.W_val))

    n_opt = prim.n_optim.(s_vals, p)
    profit_state = Π.(s_vals, p, n_opt)

    # Iterate over all possible states
    for s_i ∈ 1:nS

        prof = profit_state[s_i]

        # Calculate expected continuation value

        exp_cont_value = trans_mat[s_i, :]' * res.W_val

        # Firm exit the market if next period's expected value of stay is negative
        x = ( exp_cont_value > 0 ) ? 0 : 1

        temp_val[s_i] = prof + β * (1 - x) * (exp_cont_value )
        res.x_opt[s_i] = x

    end

    res.W_val = temp_val

end # W

#Value function iteration for W operator
function TW_iterate(prim::Primitives, res::Results; tol::Float64 = 1e-4)

    n = 0 #counter
    err = 100.0 #initialize error
    while  (err > tol) & (n < 4000)#begin iteration
        W_val_old = copy(res.W_val)
        W(prim, res)
        err = maximum(  abs.(W_val_old - res.W_val ) ) #reset error level
        n+=1
        if n % 100 == 0
            println("Iter =", n , " Error = ", err)
        end
    end
end # TW_iterate


function market_clearing(prim::Primitives, res::Results; tol::Float64 = 1e-3,  n_max::Int64 =
1000)
    @unpack Π, nS, trans_mat, ν, c_e, p_min, p_max = prim

    θ = 0.99
    n = 0

    while n < n_max
        TW_iterate(prim, res)
        # W(prim, res)
        # Calculate EC
        #EC = sum(res.W_val .* ν) - res.p * c_e
        EC = sum(res.W_val .* ν) /res.p - c_e

        # println("p = ", res.p," EC = ", EC, " tol = ", tol)
        if abs(EC) > tol * 10000
        # adjust tuning parameter based on EC
            θ = 0.5
        elseif abs(EC) > tol * 5000
            θ = 0.75
        elseif abs(EC) > tol * 1000
            θ = 0.9
        else
            θ = 0.99
        end
        if n % 10 == 0
            println(n+1, " iterations; EC = ", EC, ", p = ", res.p, ", p_min = ", p_min, ", p_max = ", p_max, ",
= ", θ)
        end
        if abs( EC ) < tol
```

```
            println("Price converged in $(n+1) iterations, p = $(res.p)")
            break
        end

        # adjust price toward bounds according to tuning parameter
        if EC > 0
            p_old = res.p
            res.p = θ*res.p + (1-θ)*p_min
            p_max = p_old
        else
            p_old = res.p
            res.p = θ*res.p + (1-θ)*p_max
            p_min = p_old
        end

        n += 1

    end
end


function Tμ(prim::Primitives, x::Array{Float64}, M::Float64)
    @unpack ν, nS, trans_mat = prim
    # Calculate B Matrix
    B = repeat((1 .- x)', nS) .* trans_mat'
    return M* (I - B)^(-1) * B * ν
end

# Calculate aggregate labor supply and demand for a given mass of entrants
function labor_supply_demand(prim::Primitives, res::Results; M::Float64=res.M)
    @unpack c_e, ν = prim

    res.μ = Tμ(prim, res.x_opt, M)

    # Calculate optimal labor demand for each firm (for each productivity level)
    n_opt = prim.n_optim.(prim.s_vals, res.p)
    # Calculate profit for each firm (for each productivity level)
    prof =  prim.Π.( prim.s_vals, res.p, n_opt)
    # Calculate  mass of firms in the market (for each productivity level)
    mass = res.μ  + M *  prim.ν

    # Calculate Total labor demand
    tot_labor_demand = n_opt' * mass

    # Calculate total profits
    tot_profit = prof' * mass
    # Calculate total supply of labor
    tot_labor_supply = 1/prim.A - tot_profit

    return tot_labor_supply, tot_labor_demand

end

# Iterate until labor market clears
function Tμ_iterate_until_cleared(prim::Primitives, res::Results;  tol::Float64 =
1e-3, n_max::Int64 = 1000)
    @unpack Π, n_optim , s_vals, ν, A, M_min, M_max = prim

    θ = 0.5
    n = 0 #counter
    err = 100.0 #initialize error


    while  (abs(err) > tol) & (n < n_max)#begin iteration
        # Calculate optimal labor demand for a given mass of entrants

        tot_labor_supply, tot_labor_demand = labor_supply_demand(prim::Primitives, res::Results)

        # Labor MarketClearing condition
        LMC = tot_labor_demand - tot_labor_supply

        # adjust tuning parameter based on LMC
        if abs(LMC) > tol * 10000
            θ = 0.5
        elseif abs(LMC) > tol * 5000
            θ = 0.75
        elseif abs(LMC) > tol * 1000
            θ = 0.9
        else
            θ = 0.99
        end
```

```
        if (n+1) % 10 == 0
            println(n+1, " iterations; LMC = ", LMC, ", M = ", res.M, ", M_min = ", M_min, ", M_max = ", M_max, "
= ", θ)
        end

        if abs( LMC ) < tol
            println("Labor Market Cleared in $(n+1) iterations, Mass of entrants = $(res.M)")
            break
        end
        # adjust price toward bounds according to tuning parameter
        if LMC > 0
            M_old = res.M
            res.M = θ*res.M + (1-θ)*M_min
            M_max = M_old
        else
            M_old = res.M
            res.M = θ*res.M + (1-θ)*M_max
            M_min = M_old
        end

        n += 1
    end

end # Tμ_iterate_until_cleared

# Solve model withoug random disturbances
function solve_model_no_dist(prim::Primitives, res::Results)

    println("\n",'='^135, "\n",'='^135, "\n", "Solving for price such that entrants make 0 profits, no random dis
    market_clearing(prim, res)
    println('='^135, "\n", "Solving for optimal mass of entrants, no random disturbances", "\n", '='^135)
    Tμ_iterate_until_cleared(prim, res)
    println('='^135, "\n", "Model Solved without random disturbances", "\n", '='^135, "\n", '='^135, "\n")

end

# Obtain values assocued with exit decicion for a given random disturvance variance
function find_Vx(prim::Primitives, res::Results,  α::Float64 ; tol::Float64 = 1e-3, n_max::Int64 =
100)
    @unpack Π, n_optim , nS, s_vals, ν, A, M_min, M_max, β, trans_mat = prim

    nX = 2

    # Initialize error and counter
    err = 100.0
    n = 0

    # Make initial guess of U(s;p)
    U₀ = zeros(nS)
    # Optimal labor demand and profits by productivity
    n_opt = n_optim.(s_vals, res.p)
    prof = Π.(s_vals, res.p, n_opt)

    # Initialize V_x
    V_x = ones(nS, nX) .* prof
    σ_x = zeros(nS, nX)

    while (err > tol ) & (n < n_max)
        # Compute V_0(s;p), V_1(s;p) wont change
        V_x[:, 1] = prof + β * (trans_mat * U₀)

        c = maximum(α * V_x, dims=2)  # Define normalization constant
        log_sum = c .+ log.( sum( exp.( α * V_x .- c), dims = 2 ) )

        # Find U₁
        U₁ = 1/α * ( 0.5772156649 .+ log_sum )

        err = maximum( abs.( U₁ - U₀ ) )
        # if n % 10 == 0
        #     println("Iter $n err = $err")
        # end
        U₀ = copy(U₁)
        n += 1

        # We can also calculate and return σ at this point
        σ_1 = exp.(α*V_x[:, 2] .- log_sum)
        σ_0 = 1 .- σ_1
        σ_x = hcat( σ_0, σ_1 )

    end # end while
```

```
    # println("Iter $n err = $err")
    return V_x, σ_x
end # find_Vx

# Find equilibrium objects given a  variance indexer α for the shocks
function find_equilibrium(prim::Primitives, res::Results, α::Float64; tol::Float64 =
1e-3, n_max::Int64 = 100)

    @unpack Π, n_optim , nS, s_vals, ν, p_min, p_max, c_e = prim

    θ = 0.99
    n = 0

    println("\n",'='^135, "\n",'='^135, "\n", "Solving for price such that entrants make 0 profits, TV1 Shocks α
= $α", "\n", '='^135)
    while n < n_max
        V_x, σ_x = find_Vx(prim, res, α);


        # Calculate value of each firm
        n_opt  = n_optim.(s_vals, res.p)
        W_vals = Π.(s_vals, res.p, n_opt) + sum(σ_x .* V_x, dims=2)

        #EC = sum(W_vals .* ν) - res.p * c_e
        EC = sum(W_vals .* ν) /res.p - c_e


        # adjust tuning parameter based on EC
        if abs(EC) > tol * 10000
            θ = 0.5
        elseif abs(EC) > tol * 5000
            θ = 0.75
        elseif abs(EC) > tol * 1000
            θ = 0.9
        else
            θ = 0.99
        end
        if n % 10 == 0
            println(n+1, " iterations; EC = ", EC, ", p = ", res.p, ", p_min = ", p_min, ", p_max = ", p_max, ",
= ", θ)
        end
        if abs( EC ) < tol
            # println("Market Cleared in $(n+1) iterations.")
            break
        end

        # adjust price toward bounds according to tuning parameter
        if EC > 0
            p_old = res.p
            res.p = θ*res.p + (1-θ)*p_min
            p_max = p_old
        else
            p_old = res.p
            res.p = θ*res.p + (1-θ)*p_max
            p_min = p_old
        end
        n += 1
        res.x_opt = copy(σ_x[:,2])
    end # end while
    println("Price converged in $(n+1) iterations, p = $(res.p)")

    println('='^135, "\n", "Solving for optimal mass of entrants, TV1 Shocks α = $α", "\n", '='^135)
    Tμ_iterate_until_cleared(prim, res)
    println('='^135, "\n", "Model Solved with random disturbances, TV1 Shocks α = $α", "\n", '='^135, "\n",'='^13
end # find_equilibrium
```