# EXAM GRA 6670- Firm Dynamics using HopenHayn Models

## Contents:

**5. Literature list**

# 1. Introduction

This assignment contains three translations of Python Language code to Julia Language code, for the partial equilibrium models based on a GitHub published by *Jacob T.Hess*, which is a *Economics PhD Candidate at Universitat Autònoma de Barcelona & BSE*. The Python code is available at his public GitHub page here: **link text (https://github.com/hessjacob/Quantitative-Macro-Models/tree/main/Hopenhayn)**. This link will also be available in our literature list.

As an general introduction, the Hopnehyan firm dynamics models are a class of economic models that study the behaviour of firms in a dynamic environment. The models are characterized by:

1. *Heterogeneity*
2. *Entry* and *exit*
3. *Endogenous* productivity shocks.

We will in this paper present and explain all the three models, implement the translated Julia code and comment on the findings for each of the models.

## 1.1 Comments on Numerical Methods and the Julia Code

### Main Numerical Methods

Each of our three Hopenhayn Models utilize the same numerical methods, but each model has different notations and variables involved. However, these are the basis numerical methods which is applied in our project:

- **Value Function Iteration** used in the "Inumbent firm" function. This is used to solve the Bellman Equation, for finding the value function of incumbent firms, which represents the maximum value a firm can achieve given its current state and making optimal decisions mowing forward. The iteration continues until the change in the value function between iterations is below our specified tolerance level, indicating convergence.

- **Fixed Point Iteration** used in the 'solve_invariant_distribution' function. This is used to compute the invariant distribution of firms across different productivity states in the economy. It involves updating the distribution of firms iteratively intil it converges to a stable distribution, where the change in distribution between iterations is below our tolerance level.

- **Interpolation** is used to estimate values between discrete points. It is used to estimate the cumulative distribution functions for both the employment distribution and firm size. In addition to values between grid points for the policy functions and value functions.

- **Markov chain / Discrete Approximation of Continuous Processes** is implemented through our transition matrix, which represents the probabilities of transitioning from one state of productivity to another in successive periods. It allows the model to incorporate uncertainty and change in productivity levels. Used in the "Incumbent firm" function during the Value Function Iteration process, taking into account the probabilities of moving between different productivity states. All three models have used a AR(1) productivity process:
  - **AR(1)**: Is a type of AutoRegressive Model that assumes that the future value of a variable is linearly related to its past values plus some degree of randomness. In this context, it is used to model the evolution of a firms procductivity over time.

- **Bisection Method** is employed in the "find_equilibrium_price' function. Also called a root-finding algorithm that repeatedly bisects an interval and then selects a subinterval in which a root must lie for further processing. Is tasked with finding the equilibrium price that balances supply and demand in the market. Used to iteratively narrow down the range of possible prices until the equilibrium price is found, this point represents where the value of entering firms equals the cost of entry.

#### Comments on the Julia Code

What we have done in each of the three models are first constructing a *Module*, which is a way to organize our codes and name each of the three models. In our *Modules*, we have defined something called *Mutuable Struct* which is what we consider equivalent of a *Class* in the Python Language. All our functions with the numerical methods and variables are being connected to our *Module* and our *Mutuable Struct*, by doing this we only need to call on these two, in order to run our models.
In Julia we also need to define our *fields* with the corresponding *type*, which essentially are variables that belong to our struct and store data associated with that struct. After experimenting and getting known with the Julia Language, our codes are been gathered into one *Code Cell* in our Jupyter Notebook, the reason for this is that the Julia Language optimizes itself when running the same *Code Cell* multiple times. There are mainly two reasons for this:

- **JIT compilation:** The Julia Language uses a JIT complier to translate our code into machine code at runtime. This means that the compiler can take advantage of specific information about our code, such our fields and the values of our constants.
- **Caching:** Julia Language also caches the results of previous runs of our code. This means that if we run in the same *Code Cell* in Jupyter Notebook, Julia will not need to recompile the code each time.

Both of these are arguments to efficiently run our Julia code in the same *Code Cell*. We have also implemented the same models in their original language (Python), and discovered that after running our Julia a couple of times, the time it took to get our results was lower for the Julia Language compared to the Python Language.

**Overall Structure**

The order of how we have structured our code is:

1. Initilized our packages and initilized the module
2. Defined our Mutuable struct with both the base fields and additional fields we needed to define
3. Constructed our Setup functions, which are setup parameters, setup grid, interpol
4. The we need to Solve Incumbant and Entrant firm problem
5. We initialize the final function "solve_model" and we find the stationary equilibrium in our economy.
6. Plotting and printing the results

**PEP8 and Flake8**

What is important to take into account is that we have tried to follow the PEP8 standards for programming by following Flake8, in order to make it efficient to read and follow for people who want to use our code. This standard is mainly used in Python, however we saw the use of it in our project with the Julia Language as well.

# 2. Model 1 Hopenhayn 1992

## 2.1 Introduction to Model 1 with equations

The first version of the models, is the one developed by Hugo A.Hopenhayn in 1992, which is a partial equilibrium model that focuses on the firm side of the economy. In this model, firms are heterogenous in their productivty and they can enter and exit the market freely. Each period, incumbent / established firms decide whether to exit the market, and new firms can enter by paying a fixed entry cost. The model solves for the stationary equilibrium, in which the price and quantity of output are determined, as well as the amount of labor hired.

**Agents**

The model has three types of agents: firms, potential entrants and exiters:

- *Firms* are heterogenous in their productivity levels.
- *Potential entrants* are firms that have not yet entered the market. They must pay a sunk entry cost to become operating firms.
- *Exiters* are firms that are leaving the market.

# Equations

1. **Interpolation Formula**:

$$y_1 = y_l + \frac{y_r - y_l}{x_r - x_l} \cdot (x_1 - x_l)$$

   - **Purpose**: Used to estimate values between known data points in our grid.
   - ( $y_1$ ) is the interpolated y-coordinate.
   - ( $x_1$ ) is the x-coordinate of the point to interpolate.
   - ( $x_l, x_r$ ) are the left and right x-coordinates of the grid interval.
   - ( $y_l, y_r$ ) are the left and right y-coordinates of the data points.

2. **Labor Hiring Decision Policy Function ( $pol_n$ )**:

$$\text{pol}_n = \left( \frac{\theta \cdot \text{price} \cdot \text{grid}_z}{\text{wage}} \right)^{\frac{1}{1-\theta}}$$

- **Purpose**: Determines the optimal number of employees a firm should hire based on its productivity, the wage rate and the price of its ouput
- ( $\theta$ ) is the Labor Share parameter in the production function.
- ( $price$ ) is the price of the firm's output.
- ( $grid_z$ ) represents a grid of productivity values for firms.
- ( $wage$ ) is the wage rate.

3. **Firm Output**:

$$\text{firm\_output} = \text{grid}_z \cdot \text{pol}_n^{\theta}$$

- **Purpose**: Calculates the output of a firm given its productivity levl and the amount of labor it hires.
- ( $grid_z$ ) is the productivity level of the firm
- ($pol_n^{\theta}$ ) isd the optimal number of employees hired

4. **Firm Profits**:

$$\text{firm\_profit} = \text{price} \cdot \text{firm\_output} - \text{wage} \cdot \text{pol}_n - \text{cf}$$

- **Purpose**: Computes the profit of a firm by subtracting costs (wages and fixed costs) from revenue.

- ( $cf$ ) is a fixed cost.

5. **Bellman Equation for Incumbent Firms Dynamics Decision Making**

$$V(z) = \max\left\{\pi(z) + \beta\mathbb{E}[V(z')|z],\ 0\right\}$$

- **Purpose**: Represents the value function of a firm, determining its value based on current profits and the expected future value.
- ( $V(z)$ ): Value function for a firm with current productivity level ( z ).
- ( $\pi(z)$ ): Profit function for a firm with productivity level ( z ).
- ( $\beta$ ) is the discount factor.
- ( $\mathbb{E}[V(z')|z]$ ): Expected value of the firm's value function in the next period, given the current productivity level ( z ).
- The maximization ( $\max\{\cdot\}$ ): Represents the firm's decision to either continue operating or exit the market.

6. **Value Function Iteration**:

$$VF = \text{firm\_profit} + \beta \cdot \max(\pi \cdot VF_{\text{old}}, 0)$$

- **Purpose**: Iterative process to solve the Bellman Equation and find the firms value function
- ( $\beta$ ) is the discount factor.
- ( $\pi$ ) is the transition matrix.
- ( $VF_{old}$ ) is the value function from the previous iteration.

7. **Policy Function for Continuing in the Market**:

$$\text{pol\_continue} = 1 \text{ if } (\pi \cdot VF \geq 0) \text{ else } 0$$

- **Purpose**: Determines whether a firm should continue operating in the market or exit.
- ( $\pi$ ) represents the profit of the firm
- ( $VF$ ) is the value function of the firm, representing its expected future profits
- ( pol_continue ) is a binary indicator (1 or 0), where 1 means the firm continues in the market, and 0 means exits

8. **Productivity Exit Threshold**:

$$\text{exit\_cutoff} = \text{grid}_z[\text{idx}]$$

- **Purpose**: Identifies the productivity level below which firms decide to exit the market
- ( $grid_z$ ) is the productivity level of the firm
- ( $idx$ ) is the index where ( pol_continue ) switches from 1 to 0, indicating the threshold for productivity level for exiting the market.

9. **Bisection Method for Price Search**:

- Initial Conditions: ( $pmin$ = 1 ), ( $pmax$ = 100 )
- **Purpose**: Equilibrium Price is found by iteratively adjusting ( $pmin$ ) and ( $pmax$ ) based on the free entry condition:

$$\text{diff} = |\beta \cdot \text{dot}(VF, \nu) - \text{ce}|$$

- ( $\beta$ ) is the discount factor.
- ( $VF$ ) is the value function of the firm, representing its expected future profits
- ( $\nu$ ) is the initial distribution vector of firms across productivity levels.
- $ce$ is the cost of entry for new firms

10. **Invariant Distribution Calculation**:

$$\text{stat\_dist} = m \cdot \left((I - \pi_{\sim})^{-1} \cdot \nu\right)$$

- **Purpose**: Calculates the steady-state distribution of firms across different productivity levels.
- ( $m$ ) is a normalization factor.
- ( $\pi_{\sim}$ ) is the adjusted transition matrix incorporating the policy function.
- ( $I$ ) is the identity matrix.
- ( $\nu$ ) is the initial distribution vector of firms across productivity levels.

## 2.2 The code of Model 1

In [1]:
```
# Setting the necessary packages from Julia and initializing the main module
module HopenhaynModel1
using QuantEcon
using Distributions
using LinearAlgebra
using Statistics
using Plots


# Export all our used functions
export Hopenhayn,setup_parameters, setup_grid, interpol, static_profit_max, incumbent_firm, find_equilibrium_


# Main class or what is called Mutable Struct in Julia
mutable struct Hopenhayn
    # Our base variables / fields

    beta::Float64              # discount factor 5 year
    theta::Float64             # labor share
    cf::Float64                # fixed cost
    ce::Float64                # entry cost
    D::Float64                 # size of the market (exogenous)
    wage::Float64              # wage, normalized to one
    rho_z::Float64             # autocorrelation coefficient
    sigma_z::Float64           # std. dev. of shocks
    Nz::Int                    # number of discrete income states
    z_bar::Float64             # constant term in continuous income process (not the mean of the process)
    plott::Bool                # select true to make plots
    tol::Float64               # difference tolerance
    maxit::Int                 # maximum value function iterations
    mc::MarkovChain            # Markov chain for income states
    pi::Array{Float64,2}       # transition matrix
    grid_z::Vector{Float64}    # grid for income states
    nu::Vector{Float64} # stationary distribution



    # Additional variables / fields

    price_ss::Float64              # price steady state
    VF::Vector{Float64}            # value function
    firm_profit::Vector{Float64} # firm profit
    firm_output::Vector{Float64} # firm output
```

```
    pol_n::Vector{Float64}       # policy function for labor hiring
    pol_continue::Vector{Bool}   # policy for continuing in the market
    exit_cutoff::Float64         # exit cutoff productivity level
    distrib_stationary_0::Vector{Float64} # initial stationary distribution
    m_star::Float64                # equilibrium mass of entrants
    distrib_stationary::Vector{Float64} # invariant productivity distribution
    total_mass::Float64             # total mass of firms
    pdf_stationary::Vector{Float64} # stationary distribution PDF
    cdf_stationary::Vector{Float64} # stationary distribution CDF
    distrib_emp::Vector{Float64}    # employment distribution
    pdf_emp::Vector{Float64}        # employment distribution PDF
    cdf_emp::Vector{Float64}        # employment distribution CDF
    total_employment::Float64       # total employment
    average_firm_size::Float64      # average firm size
    exit_rate::Float64

    # Main function with defined parameters
    function Hopenhayn(beta=0.8, theta=2/3, cf=20.0, ce=40.0, D=100.0, wage=1.0, rho_z=0.9, sigma_z=0.2, Nz=2
        model = new(beta, theta, cf, ce, D, wage, rho_z, sigma_z, Nz, z_bar, plott, 1e-8, 2000)
        setup_parameters(model)
        setup_grid(model)
        return model
    end # Function

end # Mutable Struct


"""1. Setup functions"""
function setup_parameters(model::HopenhaynModel1.Hopenhayn)
    model.tol = 1e-8          # difference tolerance
    model.maxit = 2000        # maximum value function iterations
end # function


function setup_grid(model::HopenhaynModel1.Hopenhayn)
    model.mc = rouwenhorst(model.Nz, model.z_bar, model.sigma_z, model.rho_z)
    model.pi = model.mc.p
    model.grid_z = exp.(model.mc.state_values)
    model.nu = stationary_distributions(model.mc)[1]
end # function


function interpol(x, y, x1)
```

```
    N = length(x)
    i = clamp(searchsortedfirst(x, x1), 2, N)
    xl = x[i - 1]
    xr = x[i]
    yl = y[i - 1]
    yr = y[i]
    y1 = yl + (yr - yl) / (xr - xl) * (x1 - xl)

    # Handle extrapolation
    if x1 > x[end]
        y1 = y[end] + (x1 - x[end]) * (y[end] - y[end - 1]) / (x[end] - x[end - 1])
    elseif x1 < x[1]
        y1 = y[1]
    end # if/elif

    return y1, i
end # function


"""2. Solve incumbent and entrant firm problem"""
function static_profit_max(model::HopenhaynModel1.Hopenhayn, price::Float64)
    # a. Given prices, find the labor hiring decision policy function
    pol_n = ((model.theta * price * model.grid_z) / model.wage) .^ (1 / (1 - model.theta))

    # b. Given prices and hiring decision, find firm output
    firm_output = model.grid_z .* (pol_n .^ model.theta)

    # c. Given prices and hiring decision, find profits by solving static firm problem
    firm_profit = price .* firm_output - model.wage .* pol_n .- model.cf

    return firm_profit, firm_output, pol_n
end # function


function incumbent_firm(model::HopenhaynModel1.Hopenhayn, price::Float64)
    # Initialize value function arrays
    VF_old = zeros(model.Nz)
    VF = zeros(model.Nz)

    # Compute static firm profit, output, and policy function
    firm_profit, firm_output, pol_n = static_profit_max(model, price)
```

```julia
    # Initialize expected value function
    expected_VF = zeros(model.Nz)  # Add this line to initialize expected_VF

    # Value function iteration
    for it in 1:model.maxit
        # Compute the expected value function for the next period
        expected_VF = model.pi * VF_old

        # Update the value function with static profit and discounted expected value
        VF = firm_profit + model.beta * max.(expected_VF, 0)

        # Check for convergence
        dist = maximum(abs.(VF_old - VF))
        if dist < model.tol
            break
        end # if

        # Update old value function
        VF_old = copy(VF)
    end # for-loop

    # Determine policy for continuing in the market (true if expected value is non-negative)
    pol_continue = expected_VF .>= 0

    # Find the exit cutoff productivity level
    exit_cutoff_idx = findfirst(.!pol_continue) # Use broadcasting to negate pol_continue
    exit_cutoff = isnothing(exit_cutoff_idx) ? 0.0 : model.grid_z[exit_cutoff_idx]

    return VF, firm_profit, firm_output, pol_n, pol_continue, exit_cutoff
end # function


"""3. Find stationary equilibrium """
function find_equilibrium_price(model::HopenhaynModel1.Hopenhayn)
    # a. initial price interval
    pmin, pmax = 1.0, 100.0
    price = 0.0  # Initialize price outside the loop

    # b. iterate to find prices
    for it_p in 1:model.maxit
        # i. guess a price
        price = (pmin + pmax) / 2
```

```
        # ii. incumbent firm value function
        VF, _, _, _, _, _ = incumbent_firm(model, price)

        # iii. entrant firm value function
        VF_entrant = model.beta * dot(VF, model.nu)

        # iv. check if free entry condition is satisfied
        diff = abs(VF_entrant - model.ce)

        if diff < model.tol
            break
        end # if

        # v. update price interval
        if VF_entrant < model.ce
            pmin = price
        else
            pmax = price
        end # if/else
    end # for-loop

    return price
end # function

function solve_invariant_distribution(model::HopenhaynModel1.Hopenhayn, m::Float64, pol_continue::Vector{Bool
    # Adjust the transition matrix for firms that continue
    pi_tilde = (model.pi .* pol_continue')'

    # Identity matrix
    identity_matrix = Matrix{Float64}(I, model.Nz, model.Nz)

    # Solve for the invariant distribution
    return m * ((identity_matrix - pi_tilde) \ model.nu)
end # function


"""4. Main function"""
function solve_model(model::HopenhaynModel1.Hopenhayn)
    # Start the clock
    t0 = time()

    # a. Find the optimal price using bisection (algo steps 1-3)
    model.price_ss = find_equilibrium_price(model)
```

```
# b. Use the equilibrium price to recover incumbent firm solution
model.VF, model.firm_profit, model.firm_output, model.pol_n, model.pol_continue, model.exit_cutoff = incu

# c. Invariant (productivity) distribution with endogenous exit. Here assume m=1 which
# will come in handy in the next step.
model.distrib_stationary_0 = solve_invariant_distribution(model, 1.0, model.pol_continue)

# d. Rather than iterating on market clearing condition to find the equilibrium mass of entrants (m_star)
# we can compute it analytically (Edmond's notes ch. 3 pg. 25)
model.m_star = model.D / (dot(model.distrib_stationary_0, model.firm_output))

# e. Rescale to get invariant (productivity) distribution (mass of plants)
model.distrib_stationary = model.m_star * model.distrib_stationary_0
model.total_mass = sum(model.distrib_stationary)

# Invariant (productivity) distribution by percent
model.pdf_stationary = model.distrib_stationary / model.total_mass
model.cdf_stationary = cumsum(model.pdf_stationary)

# f. calculate employment distributions
model.distrib_emp = (model.pol_n .* model.distrib_stationary)

# invariant employment distribution by percent
model.pdf_emp = model.distrib_emp / sum(model.distrib_emp)
model.cdf_emp = cumsum(model.pdf_emp)

# g. calculate statistics
model.total_employment = dot(model.pol_n, model.distrib_stationary)
model.average_firm_size = model.total_employment / model.total_mass
model.exit_rate = model.m_star / model.total_mass

# h. plot (if plotting functionality is required, it should be implemented with Julia's plotting librarie
if model.plott
    plot_results(model)
    # Plotting code would go here, using Julia's plotting libraries like Plots.jl or Makie.jl
end # if

# Print results
println("\n-----------------------------------------")
println("Stationary Equilibrium")
println("-----------------------------------------")
println("ss price  = $(round(model.price_ss, digits=2))")
```

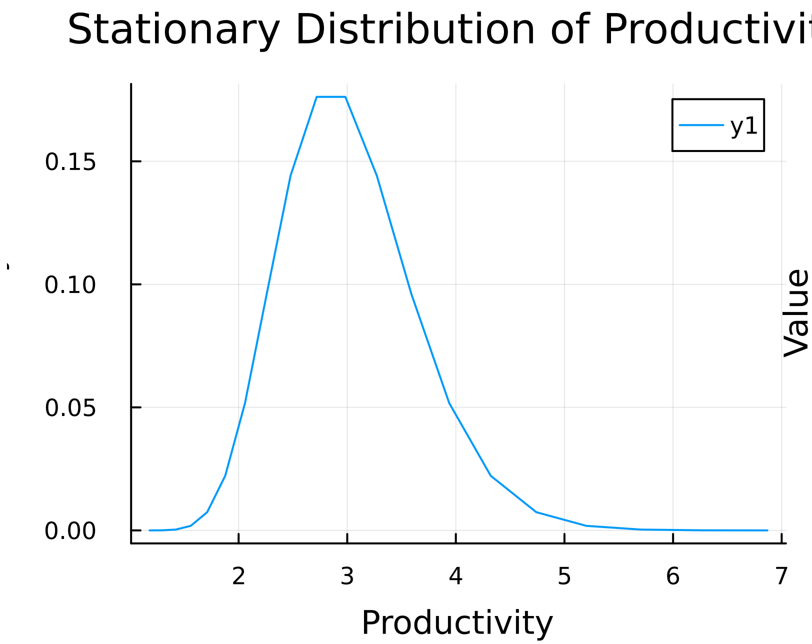
```
    println("entry/exit rate = $(round(model.exit_rate, digits=3))")
    println("avg. firm size = $(round(model.average_firm_size, digits=2))")

    t1 = time()
    println("\nTotal Run Time: $(round(t1-t0, digits=2)) seconds")
end # function


function plot_results(model::HopenhaynModel1.Hopenhayn)
    # Plot the stationary distribution of productivity
    p1 = plot(model.grid_z, model.pdf_stationary, title="Stationary Distribution of Productivity", xlabel="Pro

    # Plot the value function
    p2 = plot(model.grid_z, model.VF, title="Value Function", xlabel="Productivity", ylabel="Value", legend=t

    # Plot the employment distribution
    p3 = plot(model.grid_z, model.pdf_emp, title="Employment Distribution", xlabel="Productivity", ylabel="De


    categories = ["<20", "21-50", "51-100", "101-500", "501+"]

    # Initialize counters for each category
    num_firms_in_category = zeros(Int, length(categories))
    employment_in_category = zeros(Float64, length(categories))

    # Categorize firms and count
    for i in 1:length(model.pol_n)
        employees = model.pol_n[i]
        category_idx = if employees < 20
            1
        elseif employees <= 50
            2
        elseif employees <= 100
            3
        elseif employees <= 500
            4
        else
            5
        end # End of if-elseif-else block

        num_firms_in_category[category_idx] += 1
        employment_in_category[category_idx] += employees * model.distrib_stationary[i]
    end # for-loop
```

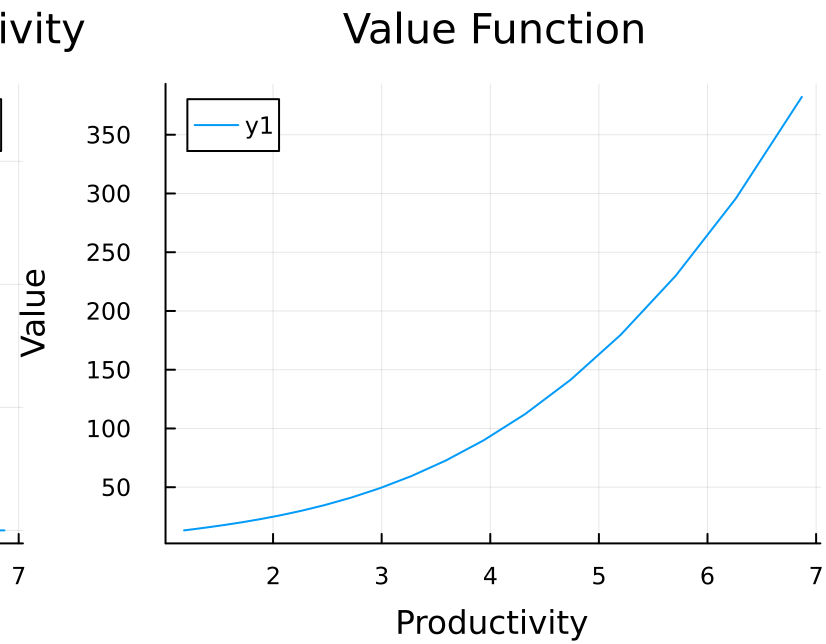
```
    # Normalize employment to get shares
    employment_share = employment_in_category / sum(employment_in_category)

    # Create pie charts
    p4 = pie(categories, num_firms_in_category, title="Size of Firm by Number of Employees")
    p5 = pie(categories, employment_share, title="Employment Share by Firm Size")

    # Combine all plots into a single figure
    combined_plot = plot(p1, p2, p3, p4, p5, layout=(4, 2), size=(800, 1200))

    # Display the combined plot
    display(combined_plot)

end # function


end # module
```

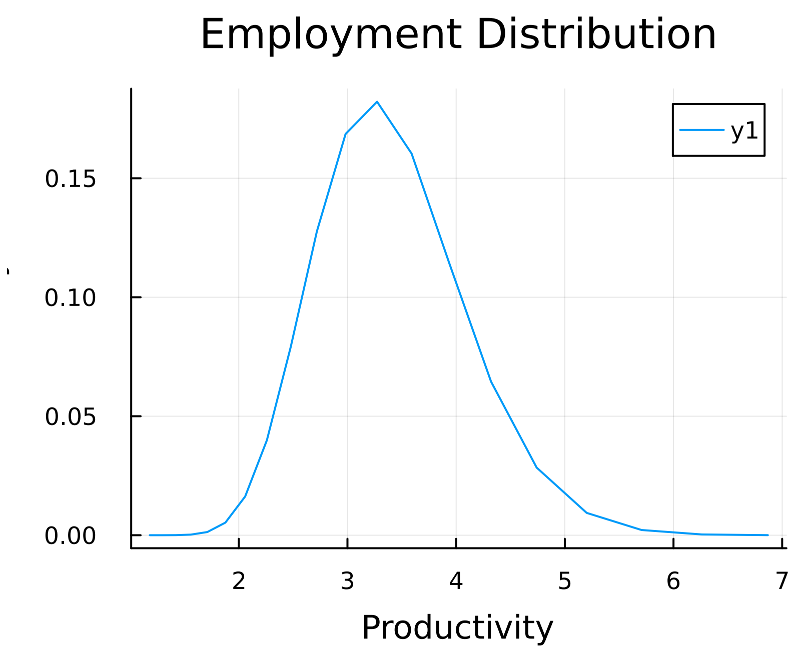
Out[1]:

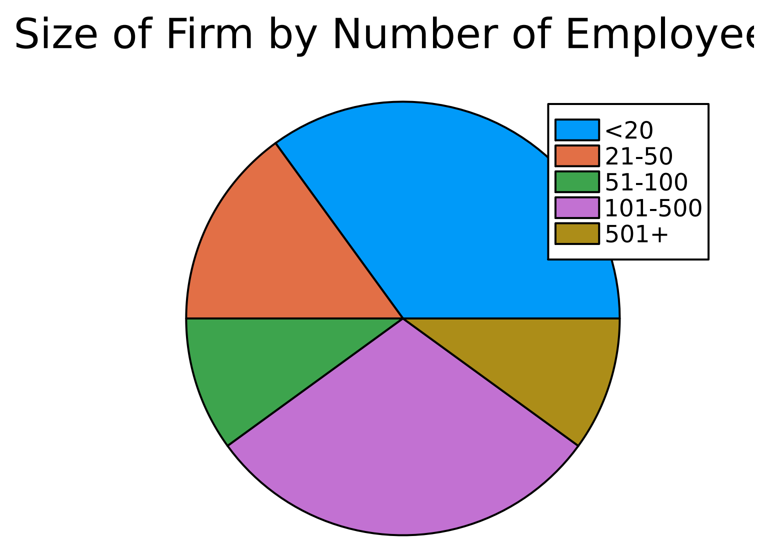
```
Main.HopenhaynModel1
```

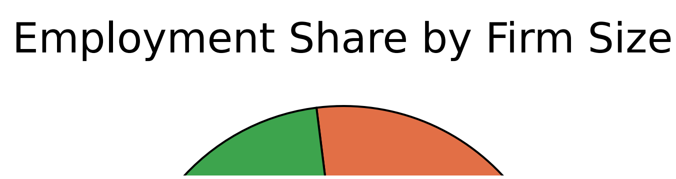
In [2]:
```
using .HopenhaynModel1  # Import the module

# Create an instance of the Hopenhayn model
model = HopenhaynModel1.Hopenhayn()

# Solve the model
HopenhaynModel1.solve_model(model)
```

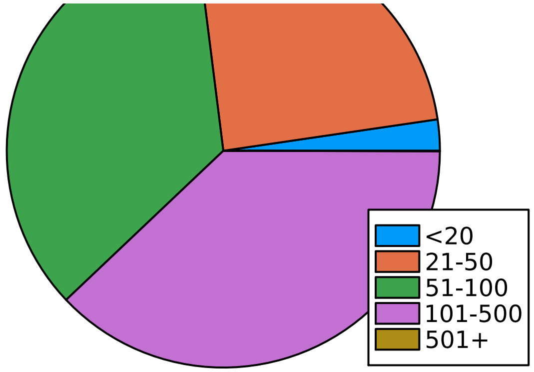

Stationary Distribution of Productivity
y1
0.00
0.05
0.10
0.15
2
3
4
5
6
7
Productivity


Value Function
y1
Value
50
100
150
200
250
300
350
2
3
4
5
6
7
Productivity


Employment Distribution
y1
0.00
0.05
0.10
0.15
2
3
4
5
6
7
Productivity


Size of Firm by Number of Employee
<20
21-50
51-100
101-500
501+


Employment Share by Firm Size




```
----------------------------------------
Stationary Equilibrium
----------------------------------------
ss price  = 1.94
entry/exit rate = 0.0
avg. firm size = 60.0

Total Run Time: 7.34 seconds
```

## 2.3 Commenting the result

**1. Equilibrium Price (ss_price) = 1.94:**

- The equilibrium price is 1.94. This price is the outcome of the interaction between various factors such as production costs, demand for products, and the competitive landscape.

**2. Entry/Exit Rate = 0.0%:**

- A 0.0% rate suggests an extremely stable market with no firms entering or exiting. This could indicate a market that has reached a very stable equilibrium where existing firms are well-established, and the barriers to entry are high enough to prevent new firms from entering.

**3. Average Firm Size = 60:**

- An average firm size of 60 suggests that, on average, firms in the industry employ 60 units of labor and capital.

**Short Plot Analysis:**

- *Stationary Distribution of Productivity Plot*: This plot visualizes the steady-state distribution of productivity levels across firms.
- *Value Function Plot*: This plot represents the value function of incumbent firms across different productivity levels. It typically increases with productivity, reflecting higher profits and viability for more productive firms.
- *Employment Distribution Plot*: This plot shows how employment is distributed across firms with different productivity levels. It can reveal whether employment is concentrated in more productive firms or if it's more evenly spread out.
- *Firm Size by Number of Employees (Pie Chart)*: This pie chart categorizes firms into different size classes based on the number of employees. It provides a visual representation of the proportion of small, medium, and large firms within the economy.

# 3. Model 2 Firm Dynamics and Neoclassical Growth Model

## 3.1 Introduction to Model 2

The second version elevates the model from Hopenhayn (1992), by integrating it with the standard neoclassical growth model. Here, firms rent capital from households, making a significant shift from the original model. This version introduces inelastic labor supply by households, contrasting with the more flexible labor market in Hopenhayn and Rogerson (1993). The agents are infinitely lived and identical, operating in complete markets. This integration provides a more comprehensve view of economy, linking firm dynamics directly with household decisions and capital markets. It allows for an exploration of how productivity shocks at the firm level can ripple through the entie economy, affecting investment decisions and capital allocation.

The primary goal is to find a stationary equilibrium where the entry and exit of firms balance out, and the economy reaches a steady state. This involves determining the equilibrium price, quantities and distributions of firms across different productivity levels. The model accounts for idiosyncratic productivity shocks to firms, influencing the calculation of optimal capital and labor demands for firms, based on their individual productivity levels.

On the household side, the model addresses the decsisons of households who supply labor, which is inelastically provided, and capital, which is rented out to firms. It delves into solving for the optimal investment decisions that households make in this economic environment.

A significant aspect of the model is its fouc on both exogenous and endogenous decision to leave based on their economic viability. Simultaneously, potential entrants evaluate their decision to enter the market, considering their initial productivity levels and the costs associated with entry.

**Agents**
There are three agents in the market:

- *Incumbent Firms*: Heterogenous in productivity and make decisions on capital and labor usage to maximize profits. Subject to idiosyncratic productivity shocks and a risk of exit.
- *Households*: Supply labor inelastically, provide capital to firms and represented as a single representative household due to complete markets and identical preferences.
- *Potential entrants*: Consider entering the market based on initial productivity and entry costs, their decisions contribute to the dynamic nature of the market.

# Equations

1. **Interpolation Formula**:

$$y_1 = y_l + \frac{y_r - y_l}{x_r - x_l} \cdot (x_1 - x_l)$$

   - **Purpose**: Used to estimate values between known data points in our grid.
   - ( $y_1$ ) is the interpolated y-coordinate.
   - ( $x_1$ ) is the x-coordinate of the point to interpolate.
   - ( $x_l, x_r$ ) are the left and right x-coordinates of the grid interval.
   - ( $y_l, y_r$ ) are the left and right y-coordinates of the data points.

2. **Optimal Capital Demand ( $pol_k$ )**:

$$\text{pol}_k = \text{grid}_z^{\frac{1}{\text{xx}}} \cdot \left(\frac{\alpha}{\text{rental\_rate}}\right)^{\frac{1-\gamma}{\text{xx}}} \cdot \left(\frac{\gamma}{\text{wage}}\right)^{\frac{\gamma}{\text{xx}}}$$

- **Purpose**: Calculates the opitmal amount of capital a firm should employ, given its productivity, the cost of capital (rental rate) and the cost of labor (wage)
- ( $grid_z$ ) represents a grid of productivity values.
- ( $xx$ ) is for factor demand solution.
- ( $\alpha$ ) is the amount of capital share
- ( $\gamma$) is the amount of labor share
- ( $rental_{rate}$ ) is the rental rate of capital.
- ( $wage$ ) is the wage rate.

3. **Optimal Labor Demand ( $pol_n$ )**:

$$\text{pol}_n = \text{grid}_z^{\frac{1}{\text{xx}}} \cdot \left(\frac{\alpha}{\text{rental\_rate}}\right)^{\frac{\alpha}{\text{xx}}} \cdot \left(\frac{\gamma}{\text{wage}}\right)^{\frac{1-\alpha}{\text{xx}}}$$

- **Purpose**: Determines the optimal number of workers a firm should hire, balancing the productivity of the firm with the costs of labor and capital
- ( $grid_z, xx, \alpha, \gamma, rental\_rate, wage$ ) as defined above in the Optimal Capital Demand function.

4. **Firm Revenues ( $firm\_output$ )**:

$$\text{firm\_output} = grid_z \cdot \text{pol}_k^{\alpha} \cdot \text{pol}_n^{\gamma}$$

- **Purpose**:Calculates trhe total revenue generated by a firm, based on its productivity, and the amounts of campital and labor employed.
- ( $grid_z, pol_k, \alpha, pol_n, \gamma$ ) as defined above in both Optimal Labor Demand and Optimal Capital Demand.

5. **Firm Profits ($firm\_profit$)**:

$$firm\_profit = \text{firm\_output} - \text{wage} \cdot \text{pol}_n - \text{rental\_rate} \cdot \text{pol}_k - \text{cf}$$

- **Purpose**: Calculates the profit of a firm by subtracting all costs (wages, rental rate of capital, and fixed costs) from its total revenue
- ( $firm\_output, wage, pol_n, rental\_rate, pol_k$ )
- ( $cf$ ) is the fixed costs among the firms

6. **Value Function Iteration**:

$$VF = \text{firm\_profit} + (1 - \lambda) \cdot \beta \cdot \max(\pi \cdot VF_{\text{old}}, 0)$$

- **Purpose**: Iterative process used to calculate the value function of a firm, which represents the present value of future profits
- ( $\lambda$ ) is the exogenous rate.
- ( $\beta$ ) is the discount factor.
- ( $\pi$ ) is the transition matrix.
- ( $VF_{old}$ ) is the value function from the previous iteration.

7. **Policy Function for Continuing in the Market**:

$$\text{pol\_continue} = 1 \text{ if } (\pi \cdot VF \geq 0) \text{ else } 0$$

- **Purpose**: Determines whether a firm should continue operating in the market or exit. Its based on the expected future profitability of the firm
- ( $\pi$ ) is the transition matrix, or transition probabilities for a firms productivity
- ( $VF$) is the value function
- If ( $\pi$ ) * ( $VF$) is non-negative, indicating expected future profitability, the firm continues (pol_continue = 1); otherwise the firm exits the market (pol_continue = 0)

8. **Productivity Exit Threshold**:

$$\text{exit\_cutoff} = \text{grid}_z[\text{idx}]$$

- **Purpose**: Identifies the productivity level below which firms decide to exit the market
- ( $grid_z$ ) is the productivity level of the firm
- ( $idx$ ) is the index where ( pol_continue ) switches from 1 to 0, indicating the threshold for productivity level for exiting the market.

9. **Bisection Method for Wage Search**:

- Initial Conditions: ( $wmin$ = 0.01 ), ( $wmax$ = 100 )
- Equilibrium Wage is found by iteratively adjusting ( $wmin$ ) and ( $wmax$ ) based on the free entry condition:
$$\text{free\_entry\_cond} = (1 - \lambda) \cdot \beta \cdot \text{dot}(VF, \nu) - \text{ce}$$
- The equilibrium wage is the wage level where new firms are indifferent between entering and not entering the market
- ( $\lambda$ ) is the exogenous rate.
- ( $\beta$ ) is the discount factor.
- ( $VF$) is the value function

- ( $\nu$ ) is the initial productivity distribution among new entrants.
- ( $ce$ ) is the entry cost for new firms

10. **Fixed Point Iteration for Stationary Distribution**:
$$\text{stat\_dist\_hat} = (1-\lambda) \cdot \text{dot}(stat\_dist\_0, \pi) \cdot \text{pol\_continue} + m \cdot \nu$$

- **Purpose**: This process fins the steady-state distribution of firms across different productivity levels in the market
- ( $\lambda$ ) is the exit rate of firms from the market.
- ( $\pi$ ) is the transition matrix for firm productivity.
- ( stat_dist_0 ) is the initial guess for the stationary distribution.
- ( $m$ ) is the normalized mass of entrants (set to 1).
- ( $\nu$ ) is the initial productivity distribution among new entrants.

## 2.3 The code of Model 2

In [3]:
```
# Setting the necessary packages from Julia and initializing the main module
module HopenhaynModel2
using QuantEcon
using Distributions
using LinearAlgebra
using Statistics
using Plots


# Export all our used functions
export Hopenhayn2, setup_parameters, setup_grid, interpol, static_profit_max, incumbent_firm, find_equilibr

# Main Class or what is called Mutable Struct in Julia

mutable struct Hopenhayn2
    # Our base variables

    beta::Float64
    alpha::Float64
    gamma::Float64
    delta::Float64
    lambdaa::Float64
    xx::Float64
    cf::Float64
    ce::Float64
    rho_z::Float64
    sigma_z::Float64
    Nz::Int
    z_bar::Float64
    tol::Float64
    maxit::Int
    interest_rate::Float64
    rental_rate::Float64
    mc::MarkovChain
    pi::Matrix{Float64}
    grid_z::Vector{Float64}
    mu_enter::Float64
    sigma_enter::Float64
    nu_cdf::Vector{Float64}
    nu::Vector{Float64}
    plott::Bool

    # Extra variables we need to define
```

```
    wage_ss::Float64   # Steady-state wage
    firm_profit::Vector{Float64}   # Vector of firm profits for each productivity level
    firm_output::Vector{Float64}   # Vector of firm outputs for each productivity level
    pol_k::Vector{Float64}   # Policy function for capital
    pol_n::Vector{Float64}   # Policy function for labor
    stat_dist_hat::Vector{Float64}   # Stationary distribution (unormalized)
    m_star::Float64   # Mass of entrants in steady state
    stat_dist::Vector{Float64}  # Stationary distribution (normalized)
    stat_dist_pdf::Vector{Float64}   # Stationary distribution probability density function
    stat_dist_cdf::Vector{Float64}   # Stationary distribution cumulative density function
    dist_emp::Vector{Float64}   # Distribution of employment
    dist_emp_pdf::Vector{Float64}   # Employment distribution probability density function
    dist_emp_cdf::Vector{Float64}   # Employment distribution cumulative density function
    Y_ss::Float64   # Steady-state output
    Yfc_ss::Float64   # Steady-state output net of fixed costs
    K_ss::Float64   # Steady-state capital
    N_ss::Float64   # Steady-state labor
    profit_ss::Float64   # Steady-state profit
    TFP_ss::Float64   # Total Factor Productivity in Steady-State
    C_ss::Float64   # Steady-state consumption
    average_firm_size::Float64   # Average firm size in Steady-state
    exit_rate::Float64   # Exit rate in steady state
    VF::Vector{Float64}   # Vector of Value-Functions
    pol_continue::Vector{Bool}

    # Setting the function
    function Hopenhayn2(; plott::Bool=true)
        instance = new()
        setup_parameters(instance)
        setup_grid(instance)
        instance.plott = plott
        return instance
    end # Function

end # Mutual Struct


"""1. Setup Functions"""
function setup_parameters(model::HopenhaynModel2.Hopenhayn2)
    model.beta = 0.9615 # Annual discount factor
    model.alpha = 0.85 / 3 # Capital share
    model.gamma = (0.85 * 2) / 3 # labor share
```

```
    model.delta = 0.08 # Annual depreciation rate
    model.lambdaa = 0.05  # exongenous exit rate
    model.xx = 1 - model.alpha - model.gamma # for factor demand solution
    model.cf = 1 # fixed cost
    model.ce = 10 #entry cost

    # AR(1) productivity rpocess
    model.rho_z = 0.6 # Autocorrelation coefficient
    model.sigma_z = 0.2 # Std.Dev of shocks
    model.Nz = 20 # Number of discrete income states
    model.z_bar = 0 # Constant term in continuous productivity process (not the mean of the process)

    # b.Iteration parameters
    model.tol = 1e-8  # Difference tolerance
    model.maxit = 2000 # Maximum value function iterations

    # c. hh solution
    model.interest_rate = 1 / model.beta - 1 # Steady-state interest rate
    model.rental_rate = model.interest_rate + model.delta  # Rental rate
end # Function


function setup_grid(model::HopenhaynModel2.Hopenhayn2)
    model.mc = QuantEcon.rouwenhorst(model.Nz, model.z_bar, model.sigma_z, model.rho_z)
    model.pi = model.mc.p
    model.grid_z = exp.(model.mc.state_values)
    mu_enter = 1.3
    sigma_enter = 0.22
    dist = Normal(mu_enter, sigma_enter)
    model.nu_cdf = cdf.(dist, model.grid_z)
    model.nu = diff(vcat(0.0, model.nu_cdf))
end # Function


"""2. One helper function"""
function interpol(x::Vector{Float64}, y::Vector{Float64}, x1::Float64)
        N = length(x)
        i = clamp(searchsortedfirst(x, x1), 2, N)
        xl = x[i - 1]
        xr = x[i]
        yl = y[i - 1]
        yr = y[i]
        y1 = yl + (yr - yl) / (xr - xl) * (x1 - xl)
```

```
        # We need to handle extrapolation
        if x1 > x[end]
            y1 = y[end] + (x1 - x[end]) * (y[end] - y[end-1]) / (x[end] - x[end-1])
        elseif x1 < x[1]
            y1 = y[1]
        end # If-statement

        return y1, i
end # Function


"""3. Solve Incumbent and entrant firm problem"""
function static_profit_max(model::HopenhaynModel2.Hopenhayn2, wage::Float64)

        # Optimal capital demand
        pol_k = model.grid_z .^ (1 / model.xx) .* (model.alpha / model.rental_rate) .^ ((1 - model.gamma) /

        # Optimal labor demand
        pol_n = model.grid_z .^ (1 / model.xx) .* (model.alpha / model.rental_rate) .^ (model.alpha / model

        # Firm revenues
        firm_output = model.grid_z .* pol_k .^ model.alpha .* pol_n .^ model.gamma

        # Firm profits
        firm_profit = firm_output - wage .* pol_n - model.rental_rate .* pol_k .- model.cf

        return firm_profit, firm_output, pol_k, pol_n
end # Function


function incumbent_firm(model::HopenhaynModel2.Hopenhayn2, wage::Float64)
        # Initialize
        VF_old = zeros(model.Nz)
        VF = zeros(model.Nz)

        # Solve the static firm problem
        firm_profit, _, _, _ = static_profit_max(model, wage)

        # Iterate in incumbent firm value function
        for it in 1:model.maxit
            VF = firm_profit + (1 - model.lambdaa) * model.beta * max.(model.pi * VF_old, 0)
```

```
            # Calculating the absolute distance
            dist = maximum(abs.(VF - VF_old))

            # Checking the distance up to our model tolerance
            if dist < model.tol
                break
            else
                VF_old = copy(VF)
            end # If-statement
        end # For-Loop

        # Continue / stay in the market policy function
        pol_continue = (model.pi * VF .>= 0)

        return VF, pol_continue
end # Function


function find_equilibrium_wage(model::HopenhaynModel2.Hopenhayn2)
        # Setting up the wage interval
        wmin, wmax = 0.01, 100.0

        # Ensuring the wmin variable is low enough
        VF_min, _ = incumbent_firm(model, wmin)
        VF_entrant_min = (1- model.lambdaa) * model.beta * dot(VF_min, model.nu) - model.ce

        # Setting the AssertionError equal to the string
        @assert VF_entrant_min > 0 "wmin is set too high"

        VF_entrant_max = Inf

        # Ensure wmax is high enough
        for i_pv in 1:model.maxit
            VF_max, _ = incumbent_firm(model, wmax)
            VF_entrant_max = (1 - model.lambdaa) * model.beta * dot(VF_max, model.nu) - model.ce

        # If statement for the wmax
            if VF_entrant_max < 0
                break
            else wmax += 100
            end # If- statement
        end # For-Loop
```

```
        # Setting the AssertionError equal to the string
        @assert VF_entrant_max < 0 "wmax is not high enough for bisection to work => No convergence"

        # We iterate to find the optimal wage
        wage_ss = 0.0

        # Starting the for loop
        for it_w in 1:model.maxit
            wage_guess = (wmin + wmax) / 2
            VF, _ =incumbent_firm(model, wage_guess)
            free_entry_cond = (1 - model.lambdaa) * model.beta * dot(VF, model.nu) - model.ce

            # Setting If-statement for the absolute value compared to tolerance value
            if abs(free_entry_cond) < model.tol
                wage_ss = wage_guess
                break
            elseif free_entry_cond < 0
                wmax = wage_guess
            else
                wmin = wage_guess
            end # If- statement
        end # For-loop

        return wage_ss
end # Function


"""4. Find Stationary (productivity) distribution"""
function solve_invariant_distribution(model::HopenhaynModel2.Hopenhayn2)
        # We initialize both variables
        stat_dist_hat = zeros(model.Nz)
        stat_dist_0 = zeros(model.Nz)
        m = 1.0 # Normlaize the mass of potential entrants to one

        # Fixed point iteration
        for it_d in 1:model.maxit
            stat_dist_hat = (1 - model.lambdaa) .* (model.pi * stat_dist_0) .* model.pol_continue .+ m .* m
            dist = maximum(abs.(stat_dist_hat - stat_dist_0))

            # Setting the if-statement for distribution value comapred to the model tolerance value
            if dist < model.tol
                break
            else
```

```
                stat_dist_0 = stat_dist_hat
            end # If-statement
        end # For-Loop

    return stat_dist_hat
end # Function


"""5. Main model with plotting"""
function solve_model(model::HopenhaynModel2.Hopenhayn2)
        # Start the clock
        t0 = time()

        # Find the steady state wage using bisection
        model.wage_ss = find_equilibrium_wage(model)

        # Use the equilibrium wage to recove incumbent firm solution
        model.firm_profit, model.firm_output, model.pol_k, model.pol_n = static_profit_max(model, model.wag
        model.VF, model.pol_continue = incumbent_firm(model, model.wage_ss)

        # Invariant (productivity) distribution with endogenous exit
        model.stat_dist_hat = solve_invariant_distribution(model)

        # Mass of entrants (m_star) in the steady state equilibrium
        model.m_star = 1.0 / dot(model.stat_dist_hat, model.pol_n)

        # Rescale to get invariant(productivity) distribution (mass of plants)
        model.stat_dist = model.m_star .* model.stat_dist_hat

        # Invariant (productivity) distribution by percent
        model.stat_dist_pdf = model.stat_dist ./ sum(model.stat_dist)
        model.stat_dist_cdf = cumsum(model.stat_dist_pdf)

        # Calculate employment distributions
        model.dist_emp = model.pol_n .* model.stat_dist

        # Invariant employment distribution by percent
        model.dist_emp_pdf = model.dist_emp ./ sum(model.dist_emp)
        model.dist_emp_cdf = cumsum(model.dist_emp_pdf)

        # Aggregate statistics
        model.Y_ss = dot(model.firm_output, model.stat_dist)
        model.Yfc_ss = dot(model.firm_output .- model.cf, model.stat_dist)
```

```
        model.K_ss = dot(model.pol_k, model.stat_dist)
        model.N_ss = dot(model.pol_n, model.stat_dist) # This should equal to 1
        model.profit_ss = dot(model.firm_profit, model.stat_dist)
        model.TFP_ss = model.Y_ss / (model.K_ss ^ model.alpha * model.N_ss^model.gamma)

        # Use resource constraint to get aggregated consumption
        model.C_ss = model.Yfc_ss - model.delta * model.K_ss - model.ce * model.m_star

        # Average firm size
        model.average_firm_size = model.N_ss / sum(model.stat_dist)

        # Exit rate
        model.exit_rate = 1.0 - sum((1 - model.lambdaa) .* (model.pi' * model.stat_dist_hat) .* model.pol_c

        t1 = time()
        println("\nTotal Run Time: $(round(t1-t0, digits=2)) seconds")

        return model
end # Function


function plot_and_print_results(model::HopenhaynModel2.Hopenhayn2)
        # Starting with if-statement on plott
        if model.plott
            idx = searchsortedfirst(model.pol_continue, false, rev=true) # Productivity threshold at which

            # Use a arbitrary productivity value as the cutoff
            if idx > length(model.grid_z)
                exit_cutoff = model.grid_z[17]
            else
                exit_cutoff = model.grid_z[idx]
            end # If-statement

            # Plotting the value function
            plot(model.grid_z, model.VF, label="Value Function", legend=:topright)
            vline!([exit_cutoff], color=:red, linestyle=:dash, label="Exit Threshold=$(round(exit_cutoff, c
            title!("Incumbent Firm Value Function")
            xlabel!("Productivity Level")
            # savefig("vf_hopenhayn2.pdf")
            display(plot!())

            # Plotting the PDF of productivity and employment
            plot(model.grid_z, model.stat_dist_pdf, label="Productivity")
```

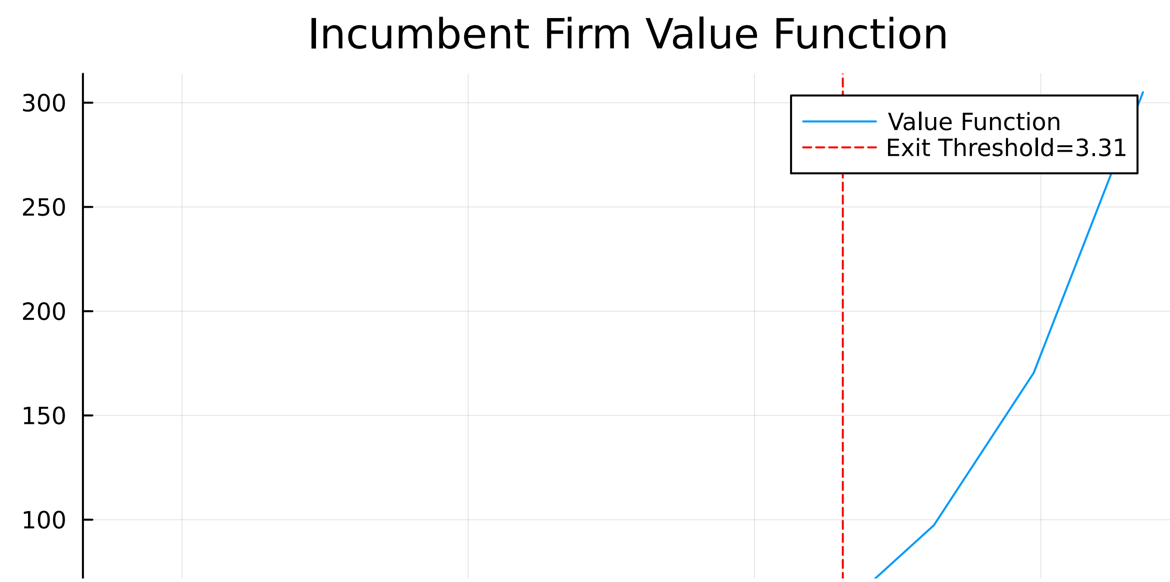
```
plot(model.grid_z, model.dist_emp_pdf, label="Employment")
title!("Stationary PDF")
xlabel!("Productivity level")
ylabel!("Percent")
# savefig("pdf_hopenhayn2.pdf")
display(plot!())

# Plotting the CDF of productivity and employment
plot(model.grid_z, model.stat_dist_cdf, label="Productivity")
plot(model.grid_z, model.dist_emp_cdf, label="Employment")
title!("Stationary CDF")
xlabel!("Productivity level")
ylabel!("Cumulative Sum")
# savefig("cdf_hopenhayn2.pdf")
display(plot!())


# Employment share pie charts
employed = [5, 10, 20, 100]
share_firms = zeros(length(employed) + 1)
share_employment = zeros(length(employed) + 1)
colors = [:blue, :red, :green, :orange, :purple]

# For-Loop for our Pie Charts
for i in 1:length(employed)
    summ = sum(share_firms)
    interpolate, _ = interpol(model.pol_n, model.stat_dist_cdf, Float64(employed[i]))
    share_firms[i] = interpolate - summ
end # For-Loop
share_firms[end] = 1 - sum(share_firms)

# Initializing pie chart for  firm size
p1 =  pie(share_firms, labels=["<=5", "6<=10", "11<=20", "21<=100"], colors=colors)
title!("Firm Size")

# savefig("firm_size_hopenhayn2.pdf")
display(p1)

# Initalizing pie chart for employment share
for i in 1:length(employed)
    summ = sum(share_employment)
    interpolate, _ = interpol(model.pol_n, model.dist_emp_cdf, Float64(employed[i]))
    share_employment[i] = interpolate - summ
```

```julia
            end # For-loop
            share_employment[end] = 1 - sum(share_employment)

            # Initializing pie chart for employment
            p2 = pie(share_employment, labels=["<=5", "6<=10", "11<=20", "21<=100"], colors=colors)
            title!("Share of Employment")
            # savefig("employment_share_hopenhayn2.pdf")
            display(p2)

        end # If-statement

        # All our printing lines
        println("\n------------------------------------------------------------------")
        println("Stationary Equilibrium")
        println("\n------------------------------------------------------------------")
        println("ss wage = $(round(model.wage_ss, digits=2))")
        println("entry/exit rate = $(round(model.exit_rate, digits=3))")
        println("avg. firm size = $(round(model.average_firm_size, digits=2))")
        println("\nss output = $(round(model.Y_ss, digits=2))")
        println("ss tfp = $(round(model.TFP_ss, digits=2))")
        println("ss consumption = $(round(model.K_ss, digits=2))")
        println("ss consumption = $(round(model.C_ss, digits=2))")
        println("ss labor = $(round(model.N_ss, digits=2))")
        println("ss profit = $(round(model.profit_ss, digits=2))")

    end # Function

end # Module
```

Out[3]: Main.HopenhaynModel2

In [4]:
```
# Create an instance of the Hopenhayn2 model
h_v2_ngm = HopenhaynModel2.Hopenhayn2(plott=true)

# Solve the model
HopenhaynModel2.solve_model(h_v2_ngm)

# Plot and print the results
HopenhaynModel2.plot_and_print_results(h_v2_ngm)
```

```
Total Run Time: 0.01 seconds
```




### 3.4 Comments on the result

These results indicate the values for our variables in the Stationary Equilibrium ( or called Steady-State in the model):

**1. Steady-State Wage (SS wage) = 1.57**

- The equilibrium wage rate is 1.57. This value reflects the price of labor in the industry. A higher wage rate could indicate a more skilled labor force or a higher demand for labor.

**2. Entry / Exit rate = 0.05:**

- The entry and exit rate of 0.05 suggests a relatively stable industry with low turnover in terms of firms entering and exiting the market. A low rate might indicate high barriers to entry or exit, or it could suggests that firms in the industry are relatively successful and thus have a lower propensity to exit.

**3. Average Firm Size = 74.68:**

- The average firm size of about 74.68 units (in terms of their labor force and capital usage), suggests a market dominated by relatively large firms. This could imply economics of scale playing a significant role in the industry, where larger firms might be more efficient and profitable.

**4. Steady-State Output (ss output) = 2.77**:

- The total output of the industry in steady state is 2.77. This is a measure of the industry's productivity and overall economic contribution.

**5. Total Factor Productivity (ss tfp) = 1.63**:

- A TFP of 1.63 indicates a relatively high level of industry productivity (higher than 1), considering both labor and capital inputs. TFP is a crucial measure in economics as it indicates how efficiently inputs are converted into outputs. A higher TFP suggests that the industry is good at turning labor and capital into productive output.

**6. Steady-State capital = 6.54 and consumption = 2.23:**

- The model reports a steady-state capital of 6.54 and a steady-state consumption of 2.23. The capital figure represents the total amount of capital employed by firms, while the consumption figure indicates the aggregate level of consumption in the economy.

**7. Steady-State Labor (ss labor) = 1**:

- The steady-state is normalized to 1.0 in this model set-up. It means that the total labor supply is used as the unit of measurement for labor inputs in our economy.

**8. Steady-State profit (ss profit) = 0.4**:

- This profit is the average earnings of firms after accounting for costs, including wages, capital rental and entry or exit costs.

**Short plot analysis**:

Our Julia code did not provide us with the prettiest plots, especially the pie-charts we had as output. It was therefore difficult to interpret the colors and the corresponding share size it wa supposed to present.

- *Incumbent Firm Value Function*: The plot shows a value function ranging from 0 to 300, with a productivity level where the exit threshold is 3.31. This threshold indicates the minimum productivity level required for a firm to remain profitable and continue operating in the market

- *Firm Size Distribution*: The largest number of firms falls into the smallest size category.

- *Employment share*: The largest share of employment is contributed by the largst or medium-sized firms. This implies that while

# 4. Model 3 Firm Dynamics with Firm-Owned Capital

## 4.1 Introduction to Model 3

This thrid version enchance the previous model 2, by the implications of firm-owned capital, a concept that brings new depth to the analysis. Now firms possess their capital stock, which grants firms the autonomy to make strategic investment decisions, influencing their growth trajectory and competitive stance in the market. Firms confront the challenge of determining the timing and scale of investments. The model accounts for the reversible nature of investments and the presence of both convex and non-convex adjustment capital costs, adding layers of complexity to these decisions. The model achieves a stationary equilibrium that includes the endogenous entry and exit of firms.

**Agents:**

- *Incumbent Firms*: These firms are already operating in the market. They decide on their capital for the next period, labor demand, whether to continue or exit the market, and their investment levels.
- *Entrant Firms*: New firms considering entering the market. They decide on their initial investment and expected firm value.
- *Households*: Supply labor inelastically, provide capital to firms and represented as a single representative household due to complete markets and identical preferences.

## Equations

1. **Interpolation Formula**:

$$y_1 = y_l + \frac{y_r - y_l}{x_r - x_l} \cdot (x_1 - x_l)$$

- **Purpose**: Used to estimate values between known data points in our grid.
- ( $y_1$ ) is the interpolated y-coordinate.

- ( $x_1$ ) is the x-coordinate of the point to interpolate.
- ( $x_l, x_r$ ) are the left and right x-coordinates of the grid interval.
- ( $y_l, y_r$ ) are the left and right y-coordinates of the data points.

2. **Discretization of the Continuous TFP Process**:

- **Purpose**: Converts a continous total factor productivity (TFP) process into a discrete form using Rouwenhorts method
- ( $\pi$ ): Transition matrix representing the probabilities of moving from one productivity state to another.
- ( $\pi_{\text{stat}}$ ): Stationary distribution of productivity states, indicating the long-term probabilities of being in each state.
- ( $\text{grid}_z$ ): A grid of productivity states, representing different levels of productivity that a firm can experience.

3. **Initial Productivity Distribution for Entrant Firm ( $\nu$ )**:

$$\nu = \frac{\text{stats.pareto.pdf}(\text{grid}_z, b, \text{loc} = -(1 - \text{grid}_z[0]))}{\sum \text{stats.pareto.pdf}(\text{grid}_z, b, \text{loc} = -(1 - \text{grid}_z[0]))}$$

- **Purpose**: Defines the initial distribution of productivity for new (entrant) firms, based on a Pareto distribution. This distribution influences how new firms are introduced into the market in terms of their productivity.
- ( $\nu$ ): The normalized distribution vector for the initial productivity of entrant firms.
- ( $\text{grid}_z$ )
- ( $b$ ): The annual discount factor

4. **Labor Demand ( $n$ )**:

$$n = \left( \frac{\gamma \cdot z \cdot k^{\alpha}}{w} \right)^{\frac{1}{1-\gamma}}$$

- **Purpose**: Calculates the labor demand of a firm based on its productivity, capital, and wage rate.
- ( $n$ ): Labor demand.
- ( $\gamma$ ): Is the amount of labor share
- ( $\alpha$ ): Is the amount of capital share
- ( $z$ ): Productivity level.
- ( $k$ ): Capital level.
- ( $w$ ): Wage rate.

5. **Firm Output ( $\text{output}$ )**:

$$\text{output} = z \cdot k^{\alpha} \cdot n^{\gamma}$$

- **Purpose**: Represents the production function of a firm, showing how output is generated from capital and labor inputs, modulated by productivity.
- ( $\gamma$ ): Is the amount of labor share
- ( $\alpha$ ): Is the amount of capital share
- ( $k$ ): Is the capital level
- ( $n$ ): Is the labor level, which we calculated above
- ( $\text{output}$ ): Total output of the firm.

6. **Firm Profit ( $\text{profit}$ )**:

$$\text{profit} = \text{output} - w \cdot n$$

- **Purpose**: Calculates the profit of a firm by subtracting labor costs from total output.
- ( $\text{output}$ ): The total output from the firm, as calculated above
- ( $n$ ): Is the labor level, which we calculated abov
- ( $w$ ): Wage rate.

7. **Value of Investing ( $\text{VF}$ )**:

$$VF_{\text{invest}} = \max\left(-k' + (1-\delta)\cdot k - \frac{\psi}{2}\cdot\left(\frac{k' - (1-\delta)\cdot k}{k}\right)^2 \cdot k - \xi \cdot k + \beta \cdot \pi \cdot VF_{\text{old}}\right)$$

- **Purpose**: Calculates the value of investing in new capital, considering costs and benefits, including adjustment costs and future values.
- ( $VF_{\text{invest}}$ ): Value of investing for the firm.
- ( $k'$ ): Future capital level.
- ( $\delta$ ): Annual Depreciation Rate
- ( $\psi$ ): Capital Adjustment Parameter
- ( $\xi$ ): Non-convex / fixed adjustment cost
- ( $\beta$ ): Annual Discount factor.
- ( $VF_{\text{old}}$ ): Value function of the firm in the previous period.

8. **Value of Waiting / Inaction ( $VF_{\text{inaction}}$ )**:

$$VF_{\text{inaction}} = \beta \cdot \sum \pi \cdot \text{interp\_func}((1-\delta)\cdot k)$$

- **Purpose**: Represents the value for a firm of not investing and continuing with its current capital stock, considering future possibilities.
- ( $\beta$ ): Annual Discount factor.
- ( $\pi$ ): Transition matrix representing the probabilities of moving from one productivity state to another.
- ( interp_func ): Interpolation function, as stated in number 1
- ( $\delta$ ): Annual Depreciation Rate
- ( $k$ ): Is the capital level

9. **Value of Exiting ( $VF_{\text{exit}}$ )**:

$$VF_{\text{exit}} = (1-\delta) \cdot k - \frac{\psi}{2} \cdot (\delta - 1)^2 \cdot k - \xi \cdot k$$

- **Purpose**: Calculates the value for a firm if it decides to exit the market, considering the salvage value of capital and costs.
- ( $k$ ): Is the capital level
- ( $\delta$ ): Annual Depreciation Rate
- ( $\psi$ ): Capital Adjustment Parameter
- ( $\xi$ ): Non-convex / fixed adjustment cost

10. **Value Function for Incumbent Firm ( $\text{VF}$ )**:

$$VF = \text{profit} + \max(VF_{\text{invest}} - c_f, VF_{\text{inaction}} - c_f, VF_{\text{exit}})$$

- **Purpose**: Determines the value of an incumbent firm by considering the maximum value from investing, remaining inactive, or exiting, adjusted for profit and fixed costs.
- ( $\text{profit}$ ): Is the profit we calculated above
- ( $VF_{\text{exit}}$ ): Value of exiting, as calculated above
- ( $VF_{\text{investing}}$ ): Value of investing, as calculated above
- ( $VF_{\text{inaction}}$ ): Value of waiting/inaction, as calculated above
- ( $c_f$ ): Fixed costs

11. **Entrant Firm Value Function ( $VF_{\text{entrant}}$ )**:

$$VF_{\text{entrant}} = \max\left(-c_k \cdot \text{grid}_k + \beta \cdot \text{dot}(\nu, VF)\right)$$

- **Purpose**: Determines the value of entering the market for a new firm, considering the costs and potential future earnings.
- ( $c_k$ ): Price of capital at entry
- ( $\text{grid}_k$ ): Grid of capital values.

- ( $\beta$ ): Annual Discount factor.
- ( $\nu$ ): The normalized distribution vector for the initial productivity of entrant firms.
- ( $\mathrm{VF}$ ): Value function for Incumbent Firms

12. **Optimal Initial Investment for Entrant Firm ( $k_e$ )**:

$$k_e = \mathrm{grid}_k[\mathrm{argmax}(RHS_{\mathrm{entrant}})]$$

- **Purpose**: Identifies the optimal initial capital investment for a new entrant firm based on the value function.

13. **Bisection Method for Wage Search**:

- Initial Conditions: ( $wmin$ = 0.01 ), ( $wmax$ = 100 )
- Equilibrium Wage is found by iteratively adjusting ( $wmin$ ) and ( $wmax$ ) based on the free entry condition:

$$\mathrm{free\_entry\_cond} = (1-\lambda)\cdot\beta\cdot\mathrm{dot}(VF,\nu) - \mathrm{ce}$$

- The equilibrium wage is the wage level where new firms are indifferent between entering and not entering the market
- ( $\lambda$ ) is the exogenous rate.
- ( $\beta$ ) is the discount factor.
- ( $VF$) is the value function
- ( $\nu$ ) is the initial productivity distribution among new entrants.
- ( $ce$ ) is the entry cost for new firms

14. **Fixed Point Iteration for Stationary Distribution**:

- Iterative Process:

$$\mathrm{stationary\_pdf}_{t+1}(z',k') = \sum_{z,k} \mathrm{stationary\_pdf}_t(z,k)\cdot\pi(z,z')\cdot\mathrm{pol\_continue}(z,k)\cdot\mathrm{Indicator}(k' \text{ is the next state for } k)$$

- Convergence Criterion:

$$\mathrm{dist} = \max_{z,k}\left|\mathrm{stationary\_pdf}_{t+1}(z,k) - \mathrm{stationary\_pdf}_t(z,k)\right|$$

- **Purpose**: An iterative process to find the stationary distribution of firms across different states in the model. This is crucial for

## 4.2 The code of Model 3

In [7]:
```
# Setting the necessary packages from Julia and initializing the main module
module HopenhaynModel3
using StatsBase
using QuantEcon
using Interpolations
using LinearAlgebra
using Plots
using Base.Threads
using Distributions


# Exporting our functions
export Hopenhayn3, setup_parameters, setup_grid, setup_discretization, pareto_pdf, make_grid, entrant_firm,


# Main Class or what is called Mutable Struct in Julia

mutable struct Hopenhayn3
    # Our base variables / fields

    plott::Bool
    beta::Float64
    alpha::Float64
    gamma::Float64
    delta::Float64
    psi::Float64
    xx::Float64
    cf::Float64
    ce::Float64
    xi::Float64
    ck::Float64
    interest_rate::Float64
    b::Float64
    rho_z::Float64
    sigma_z::Float64
    Nz::Int
    z_bar::Float64
    tol::Float64
    tol_w::Float64
    maxit::Int
    Nk::Int
    k_min::Float64
    k_max::Float64
```

```
    curv::Float64
    grid_k::Vector{Float64}
    mc::MarkovChain
    pi::Matrix{Float64}
    pi_stat::Vector{Float64}

    # Additional variables / fields

    grid_z::Vector{Float64}  # Grid for productivity levels
    nu::Vector{Float64}  # Vector of firm exit probabilities
    wage_ss::Float64  # Steady-state wage
    it_vfi::Int  # Iteration count for Value Function Iteration
    params_vfi::Tuple  # Parameters for Value Function Iteration
    params_dist::Tuple  # Parameters for distribution calculation
    VF::Matrix{Float64}  # Value Function matrix
    pol_kp::Matrix{Float64}  # Policy function for capital choice
    pol_n::Matrix{Float64}  # Policy function for labor choice
    pol_continue::Matrix{Float64}  # Policy function for continuation decision
    pol_inv::Matrix{Float64}  # Policy function for investment decision
    firm_output::Matrix{Float64}  # Matrix of firm outputs for each productivity level
    firm_profit::Matrix{Float64}  # Matrix of firm profits for each productivity level
    VF_entrant::Float64  # Value Function for entrant firms
    k_e::Float64  # Capital choice for entrant firms
    stationary_pdf_hat::Matrix{Float64}  # Unnormalized stationary probability density function
    it_pdf::Int  # Iteration count for PDF calculation
    dist_pdf::Float64  # Distribution probability density function
    m_star::Float64  # Mass of entrants in steady state
    stationary_pdf_star::Matrix{Float64}  # Normalized stationary probability density function
    capital_marginal_pdf_star::Vector{Float64}  # Marginal PDF of capital in steady state
    emp_pdf::Matrix{Float64}  # Employment probability density function
    emp_marginal_pdf::Vector{Float64}  # Marginal PDF for employment
    emp_marginal_cdf::Vector{Float64}  # Marginal CDF for employment
    Y_ss::Float64  # Steady-state output
    K_ss::Float64  # Steady-state capital
    N_ss::Float64  # Steady-state labor
    Inv_ss::Float64  # Steady-state investment
    TFP_ss::Float64  # Total Factor Productivity in steady state
    average_incumbent_firm_size::Float64  # Average size of incumbent firms
    average_entrant_firm_size::Float64  # Average size of entrant firms
    exit_rate::Float64  # Rate of firm exit
    wage_guess::Float64  # Initial guess for wage

    # Setting the main function
```

```
    function Hopenhayn3(; plott::Bool=true)
        instance = new()
        setup_parameters(instance)
        setup_grid(instance)
        setup_discretization(instance)
        instance.plott = plott
        return instance
    end # Function

end # Mutual Struct


"""1. Setup Functions"""
function setup_parameters(model::HopenhaynModel3.Hopenhayn3)
    model.beta = 0.9615   # Annual discount factor
    model.alpha = 0.85 / 3 # Capital share
    model.gamma = (0.85 * 2) / 3 # labor share
    model.delta = 0.1 # Annual depreciation rate
    model.psi = 0.5 # Capital adjustment parameter
    model.xx = 1 - model.alpha - model.gamma # for factor demand solution
    model.cf = 0.5 # fixed cost
    model.ce = 0.5 # entry cost
    model.psi = 0.25 # Convex adjustment cost parameter
    model.xi = 0.01 # Non-convex / fixed adjustment cost
    model.ck = 1 # Price of capital at entry
    model.interest_rate = 1 / model.beta - 1  # Steady-State interest rate
    model.b = 2 # Shape per parameter for pareto initial productivity distribution


    # AR(1) productivity process
    model.rho_z = 0.6 # Autocorrelation coefficient
    model.sigma_z = 0.2 # Std.Dev of shocks
    model.Nz = 10 # Number of discrete income states
    model.z_bar = 0 # Constant term in continuous productivity process (not the mean of the process)

    # b.Iteration parameters
    model.tol = 1e-8  # Default tolerance
    model.tol_w = 1e-4 # Tolerance for wage bisection
    model.maxit = 2000 # Maximum iterations

    # c. Capital grid
    model.Nk = 500 # number of capital grid points
    model.k_min = 0.01 # Minimum capital level
```

```
    model.k_max = 40 # Maximum capital level
    model.curv =  3 # Grid curvature parameter
end # Function


function setup_grid(model::HopenhaynModel3.Hopenhayn3)
    model.grid_k = make_grid(model.k_min, model.k_max, model.Nk, model.curv)
end # Function


function setup_discretization(model::HopenhaynModel3.Hopenhayn3)
    model.mc = QuantEcon.rouwenhorst(model.Nz, model.z_bar, model.sigma_z, model.rho_z)
    model.pi = model.mc.p
    model.pi_stat = reduce(vcat, stationary_distributions(model.mc))   # Flatten the matrix to a vector
    model.grid_z = exp.(model.mc.state_values)

    # Initialize productivity distribution for entrant firm
    model.nu = pareto_pdf(model.grid_z, model.b, (1 - model.grid_z[1]))

    # Used later to solve the model in the incumbent_firm function
    model.params_vfi = (model.alpha, model.beta, model.delta, model.gamma, model.cf, model.psi, model.xi, m
    model.params_dist = (model.grid_k, model.Nz, model.pi, model.pi_stat, model.nu, model.maxit, model.tol)
end # Function


function pareto_pdf(grid::Vector{Float64}, shape::Float64, shift::Float64)
    dist = Pareto(shape, shift)
    pdf_vals = pdf.(dist, grid)
    return pdf_vals / sum(pdf_vals)
end # Function


"""2. One helper function"""
function make_grid(min_val::Float64, max_val::Float64, num::Int, curv::Float64)
        grid = zeros(num)
        scale = max_val - min_val
        grid[1] = min_val
        grid[end] = max_val
        for i in 2:num - 1
            grid[i] = min_val + scale * ((i - 1) / (num - 1)) ^curv
        end # For-Loop
    return grid
end # Function
```

```
"""3. Solve incumbent and entrant firm problem"""
function entrant_firm(model::HopenhaynModel3.Hopenhayn3, VF::Matrix{Float64})
        # Entrant firm chooses its initial investment plus expected firm value over the initial productivit
        # Initialize an empty matrix to store the results of multiplication
        VF_nu = zeros(size(VF))

        # Multiply each row of VF by the corresponding element in model.nu
        Threads.@threads for i in 1:length(model.nu)
            VF_nu[i, :] = model.nu[i] * VF[i, :]
        end # For loop

        # Ensure model.grid_k is a row vector for broadcasting
        grid_k_row = reshape(model.grid_k, 1, length(model.grid_k))

        RHS_entrant = - model.ck .* grid_k_row .+ model.beta * VF_nu
        VF_entrant = maximum(RHS_entrant) # Value of the entrant
        k_e_index = argmax(RHS_entrant) # Index of optimal initial investment

        # Convert the linear index to Cartesian Indices
        indices = CartesianIndices(RHS_entrant)

        row_index, col_index = Tuple(indices[k_e_index])

        # Use the column index to access the correct value in model.grid_k

        k_e = model.grid_k[col_index]  # Optimal initial investment


        return VF_entrant, k_e
end # Function


function incumbent_firm(model::HopenhaynModel3.Hopenhayn3, wage::Float64, params_vfi::Tuple)
         # Unpack parameters from the model for easier syntax
        alpha, beta, delta, gamma, cf, psi, xi, pi, grid_k, grid_z, maxit, tol =
            model.alpha, model.beta, model.delta, model.gamma, model.cf, model.psi, model.xi, model.pi, moc
        Nz = length(grid_z)
        Nk = length(grid_k)

        # Initialize value function and policy functions
        VF_old = zeros(Nz, Nk)
```

```
        VF = copy(VF_old)
        pol_kp = copy(VF_old) # k prime, aka firms capital stock next period
        pol_n = copy(VF_old)
        pol_inv = copy(VF_old)
        pol_continue = copy(VF_old)
        firm_output = copy(VF_old)
        firm_profit = copy(VF_old)

        # Other value-function variables we need
        VF_invest = zeros(Nz, Nk)
        VF_inaction = zeros(Nz, Nk)
        VF_exit = zeros(Nz, Nk)

        # Creating our own Interpolation object / function
        interp_func = [LinearInterpolation(model.grid_k, VF_old[izz, :], extrapolation_bc=Flat()) for izz i

        # Initializing the iteration counter
        it = 0
        # Initialize it_vfi
        it_vfi = 0

        # Iterate over the value function
        for it in 1:maxit
            # Update interpolation function if VF_old changes
                interp_func = [LinearInterpolation(model.grid_k, VF_old[izz, :], extrapolation_bc=Flat()) f

            Threads.@threads for iz in 1:Nz
                for ik in 1:Nk
                    pol_n[iz, ik] = ((grid_z[iz] * gamma) / wage) ^ (1 / (1 - gamma)) * grid_k[ik] ^ (alpha

                    # Solution to the static problem
                    firm_output[iz, ik] = grid_z[iz] * grid_k[ik] ^ alpha * pol_n[iz, ik] ^ gamma
                    firm_profit[iz, ik] = firm_output[iz, ik] - wage * pol_n[iz, ik]

                    # Continuation values

                    # Value of investing
                    RHS_invest = -grid_k .+ (1 - delta) .* grid_k[ik] .- (psi / 2) .* ((grid_k .- (1 - delt

                    # Find the index of the maximum value in RHS_invest
                    max_index_invest = argmax(RHS_invest)

                    # Extract the correct index for a vector
```

```
                max_index_invest = max_index_invest[2]

                # Use the index to find the corresponding value in grid_k
                pol_kp_invest = grid_k[max_index_invest]

                VF_invest[iz, ik] = RHS_invest[max_index_invest]

                # Value of waiting (inaction)
                RHS_inaction = 0.0
                for izz in 1:Nz
                    RHS_inaction += pi[iz, izz] * interp_func[izz]((1 - delta) * grid_k[ik])
                end # For-Loop

                VF_inaction[iz, ik] = beta * RHS_inaction

                # Value of exiting
                VF_exit[iz, ik] = (1 - delta) * grid_k[ik] - (psi / 2) * (delta - 1)^2 * grid_k[ik] - x

                # Value of incumbent form
                vf_array = [VF_invest[iz, ik] - cf, VF_inaction[iz, ik] - cf, VF_exit[iz, ik]]
                VF[iz, ik] = firm_profit[iz, ik] + maximum(vf_array)

                # Determine the optimal policy
                max_index = argmax(vf_array)
                if max_index == 1  # Invest
                    pol_kp[iz, ik] = pol_kp_invest
                    pol_inv[iz, ik] = pol_kp[iz, ik] - (1 - delta) * grid_k[ik]
                    pol_continue[iz, ik] = 1
                elseif max_index == 2 # Inaction
                    pol_kp[iz, ik] = (1 - delta) * grid_k[ik]
                    pol_inv[iz, ik] = 0
                    pol_continue[iz, ik] = 1
                else # Exit
                    pol_kp[iz, ik] = 0
                    pol_inv[iz, ik] = -(1 - delta) * grid_k[ik]
                    pol_continue[iz, ik] = 0
                end # If statement
            end # For loop Nk
        end # For loop Nz

        # Check for convergence
        dist = maximum(abs.(VF - VF_old))
        if dist < tol
```

```julia
                break
        end # If statement

        # Increment it_vfi
        it_vfi += 1

        # Setting VF_old to copy of VF
        VF_old = copy(VF)

    end # For loop Maxit

    return VF, pol_kp, pol_n, pol_continue, pol_inv, firm_output, firm_profit, it_vfi
end # Function


function find_equilibrium_wage(model::HopenhaynModel3.Hopenhayn3)
        # Set up the wage interval
        wmin, wmax = 0.01, 100.0
        wage_ss = 0.0  # Initialze the steady-state wage variable
        it_w = 0  # Initialize the iteration variable

        # Starting the for-loop
        for it in 1:model.maxit
            it_w = it   # Update the iteration variable
            println("\n--------------------------------")
            println("Iteration #:", it_w)

            # Guess a wage
            wage_guess = it_w == 1 ? 0.5 : (wmin+wmax) / 2

            # Solve the incumbent firm problem
            println("Solving incumbent firm problem...")
            VF, pol_kp, pol_n, pol_continue, pol_inv, firm_output, firm_profit, it_vfi = incumbent_firm(moc

            println("Type of it_vfi", typeof(it_vfi))

            # Check for convergence of the incumbent firm problem
            if it_vfi < model.maxit
                println("Value function convergence in $it_vfi iterations")
            else
                error("No value function convergence")
            end # If statement
```

```
        # Solve the entrant firm problem using the value function from the incumbent firm
        println("Soving entrant firm problem...")
        VF_entrant, k_e = entrant_firm(model, VF)

        # Calculate the free entry condition
        free_entry_cond = VF_entrant - model.ce

        # Check if the free entry condition is satisfied
        if abs(free_entry_cond) < model.tol_w
            println("\n---------------------------------")
            println("Convergence!")
            wage_ss = wage_guess
            break
        else
            # Update the wage interval
            if free_entry_cond < 0
                wmax = wage_guess
            else
                wmin = wage_guess
            end # If statement
        end # If statement
        println("New wage guess = $wage_guess \t Free entry condition = $free_entry_cond")
    end # For loop

    if it_w >= model.maxit
        println("No convergence")
    end # If statement

    return wage_ss
end # Function


"""4. Find Stationary distribution"""
function discrete_stationary_denssity(model::HopenhaynModel3.Hopenhayn3, pol_kp, k_e, pol_continue)
    # Defining variables
    grid_k = model.grid_k
    Nz = model.Nz
    pi = model.pi
    pi_stat = model.pi_stat
    nu = model.nu
    maxit = model.maxit
    tol = model.tol
```

```
Nk = length(grid_k)
m = 1 # Normalize the mass of potential entrants to one

# Initial guess: uniform distribution
stationary_pdf_old = ones(Nk, Nz) ./ Nk
stationary_pdf_old .= (stationary_pdf_old .* transpose(pi_stat))  # Shoul be ' here or??

# Fixed point iteration
for it in 1:maxit
    stationary_pdf = zeros(Nz, Nk) # Distribution in period t+1

    Threads.@threads for iz in 1:Nz
        for ia in 1:Nk
            k_prime = pol_kp[iz, ia]

            # Obtain distribution in period t+1
            if k_prime <= grid_k[1]
                for izz in 1:Nz
                    stationary[izz, 1] += stationary_pdf_old[iz, ia] * pi[iz, izz] * pol_continue[iz, i
                end # For loop
            elseif k_prime >= grid_k[end]
                for izz in 1:Nz
                    stationary_pdf[izz, end] += stationary_pdf_old[iz, ia] * pi[iz, izz] * pol_continue
                end # Elseif
            else
                j = sum(grid_k .<= k_prime) # Grid index where k' is located
                for izz in 1:Nz
                    stationary_pdf[izz, j] += (stationary_pdf_old[iz, ia] * pi[iz, izz] * pol_continue[
                end # For loop
            end # If statement
        end # For loop Nk
    end # For loop Nz

    # Add on mass of entrants
    ike = sum(grid_k .<= k_e)  # Grid index where k_e is located
    stationary_pdf[:, ike] .+= m .* nu

    # Calculate supremum norm
    dist = maximum(abs.(stationary_pdf - stationary_pdf_old))

    if dist < tol
        break
    else
```

```
            stationary_pdf_old = copy(stationary_pdf)
        end # If statement
    end # For loop Maxit

    return stationary_pdf, it, dist
end # Function


"""5. Main function"""
function solve_model(model::HopenhaynModel3.Hopenhayn3)
    # a) Find the steady state wage using bisection
    println("\nFinding wage that satisfies free entry condition...")
    model.wage_ss = find_equilibrium_wage(model)

    # b) Use the equilibrium wage to recover incumbent and entrant firm solutions
    println("\nRecovering equilibrium solutions...")
    model.VF, model.pol_kp, model.pol_, model.pol_continue, model.pol_inv, model.firm_output, model.firm_pr
    model.VF_entrant, model.k_e = entrant_firm(model, model.VF)

    # c) Invariant joint distribution with endogenous exit
    println("\nFinding stationary density function by forward iteration...")
    model.stationary_pdf_hat, model.it_pdf, model.dist_pdf = discrete_stationary_density(model, model.pol_k

    if model.it_pdf < model.maxit - 1
        println("Convergence in $(model.it_pdf) iterations")
    else
        println("Maximum iteration number reached. Distance between last iteration: $(model.dist_pdf)")
    end # If statement

    # d) Mass of entrant (m_star) in the ss equilibrium
    model.m_star = 1 / sum(sum(model.stationary_pdf_hat .* model.pol_n))

    # e) Rescale to get invariant join distribution (mass of plants)
    model.stationary_pdf_star = model.stationary_pdf / sum(sum(model.stationary_pdf))

    # f) Marginal distributions
    println("\nCalculating aggregate statistics and marginal densities... ")

    # 1. Marginal capital density by percent
    model.capital_marginal_pdf_star = sum(model.stationary_pdf_star, dims=1)[:]

    # 2. Employment (Capital) density
    model.emp_pdf = model.pol_n .* model.stationary_pdf
```

```
# 3. Marginal (capital) employment density by percent
model.emp_marginal_pdf = sum(model.emp_pdf, dims=1)[:] / sum(model.emp_pdf)
model.emp.marginal_cdf = cumsum(model.emp_marginal_pdf)

# g) Aggregate statistics
model.Y_ss = sum(model.firm_output .* model.stationary_pdf)
model.K_ss = sum(model.stationary_pdf .* model.grid_k)
model.N_ss = sum(model.stationary_pdf .* model.pol_n)
model.Inv_ss = sum(model.stationary_pdf .* model.pol_inv)
model.TFP_ss = model.Y_ss / (model.K_ss^model.alpha * model.N_ss^model.gamma)

model.average_incumbent_firm_size = sum(model.stationary_pdf_star .* model.pol_n)
model.average_entrant_firm_size = sum(model.nu .* model.pol_n[:, sum(model.grid_k .<= model.k_e)])
model.exit_rate = 1 - sum((model.pi' * model.stationary_pdf_hat) .* model.pol_continue) / sum(model.sta

# h) Plot
if model.plott
    idx = [1, 3, 5, 7, 10]
    p1 = plot(model.grid_k, model.VF[idx, :], label=["V(k, z_$(idx[1]))" "V(k, z_$(idx[2]))" "V(k, z_$(

    p2 = plot(model.grid_k, model.pol_kp[idx, :], label=["k'(k,z_$(idx[1]))" "k'(k,z_$(idx[2]))" "k'(k,
    plot!(model.grid_k, (1-model.delta) .* model.grid_k, linestyle=:dash, label="k(1-delta)")
    title!("Capital Next Period Policy Function")
    xlabel!("Capital")

    p3 = plot(model.gid_k, model.pol_inv[idx, :], label=["i(k,z_$(idx[1]))" "i(k,z_$(idx[2]))" "i(k,z_$

    p4 = plot(model.grid_k, model.pol_n[idx, :], label=["n(k,z_$(idx[1]))" "n(k,z_$(idx[2]))" "n(k,z_$(

    p5 = plot(model.grid_k, [model.capital_marginal_pdf_star model.emp_marginal_pdf], label=["Productiv

    display(p1)
    display(p2)
    display(p3)
    display(p4)
    display(p5)
end # If statement

println("\n--------------------------------------------------")
println("Stationary Equilibrium")
println("\n--------------------------------------------------")
println("ss wage = $(round(model.wage_ss, digits=2))")
```

```
    println("Exit rate = $(round(model.exit_rate, digits=3))")
    println("Average incumbent firm size = $(round(model.average_incumbent_firm_size, digits=2))")
    println("Average entrant firm size = $(round(model.average_entrant_firm_size, digits=2))")
    println("\n ss output = $(round(model.Y_ss, digits=2))")
    println("ss investment = $(round(model.Inv_ss, digits=2))")
    println("ss TFP = $(round(model.TFP_ss, digits=2))")
    println("ss capital = $(round(model.K_ss, digits=2))")
    println("ss labor = $(round(model.N_ss))")
end # Function


end # Module
```

Out[7]:

```
Main.HopenhaynModel3
```

In [8]:
```
# Create an instance of the Hopenhayn3 model
h_v3_ngm = HopenhaynModel3.Hopenhayn3(plott=true)

# Solve the model
HopenhaynModel3.solve_model(h_v3_ngm)
```

```
Finding wage that satisfies free entry condition...

--------------------------------
Iteration #:1
Solving incumbent firm problem...
VF:Matrix{Float64}
(10, 500)
Type of it:Int64
Type of it_vfiInt64

Noe value function convergence

Stacktrace:
 [1] error(s::String)
   @ Base .\error.jl:35
 [2] find_equilibrium_wage(model::Main.HopenhaynModel3.Hopenhayn3)
   @ Main.HopenhaynModel3 .\In[7]:346
 [3] solve_model(model::Main.HopenhaynModel3.Hopenhayn3)
   @ Main.HopenhaynModel3 .\In[7]:447
 [4] top-level scope
   @ In[8]:5
```

## 4.3 Comments on the result

Our final model in Julia did not achieve convergence, and we were unable to resolve this issue. This model was significantly more complex and advanced than the first two.

# 5. Literature List


- Github: https://github.com/hessjacob/Quantitative-Macro-Models/tree/main/Hopenhayn (https://github.com/hessjacob/Quantitative-Macro-Models/tree/main/Hopenhayn)
- Julia Documentation: https://docs.julialang.org/en/v1/manual/types/ (https://docs.julialang.org/en/v1/manual/types/)
- Hopenhayn 1992: https://tomasrm.github.io/teaching/quantmacro/hopenhayn.pdf (https://tomasrm.github.io/teaching/quantmacro/hopenhayn.pdf)
- Markov Chain / AR(1) Process: https://stats.stackexchange.com/questions/23789/is-ar1-a-markov-process (https://stats.stackexchange.com/questions/23789/is-ar1-a-markov-process)