An Introduction to Julia in Economics

Yang Guang

Department of Economics
Nankai University

2024.07.10 / Shenzhen

Contents

What is Julia

- Julia is a very young but promising programming language focused on scientific computing. Its development began in 2009 while the first version was provided in 2012. Julia is a free and open source programming language with a license provided by MIT.
- The main attractiveness of the Julia comes from combining the high level syntax of languages like Python and Matlab with the speed of low level languages like Fortran or C/C++.
- For macroeconomists, Fortran is still quite used since it provides a fast solution to computational intensive problems. Julia provides two solutions. First, it can easily embed Fortran (as well as C/C++ code). Second, it also makes possible to write the entire code in Julia, while keeping the execution speed comparable to the ones by Fortran or C/C++.

Installation

- Julia can be run under Windows, Linux and OSX. As this slides was finished, Julia was at version 1.10.4.)
- It is not strictly required, but I will strongly encourage installing and using Visual Studio Code (VS Code) to perform Julia.
- For instructions in Chinese for installation of Julia and VS Code, please refer to: https://zhuanlan.zhihu.com/p/163809924
- For instructions in English for installation of Julia and VS Code, please refer to: https://code.visualstudio.com/docs/languages/julia

Installation

- I here simply introduce the installation:
  1. Download Julia from the official website: http://julialang.org/downloads/
  2. Install VS Code from the official website: https://code.visualstudio.com/download.
  3. Start VS Code. Inside VS Code, go to the Extensions view by clicking View on the top menu bar and then selecting Extensions.
  4. In the Extensions view, search for the term "julia" in the Marketplace search box, then select the Julia extension (julialang.language-julia) and select the Install button.
  5. Restart VS Code.

Understanding REPL

- REPL stands for Read/Evaluate/Print/Loop. When we start the Julia command prompt, it will provide a REPL window that is the most direct way to access the above named operations.
- To use REPL, we simply type statements and press enter so that Julia can execute them. For example, by writing:

```
Julia> x=0
0
```

- We can access help, pretty much like in other programming languages, by writing:

```
Julia> ?
```

After accessing the help mode, we can search functions by their names. For example, by writing:

```
Help?> sum
```

Script

We can write the program in a script and then run it.

```
# In Julia script, the contents following '#' is
    comment

# Load the package we need
using LinearAlgebra, Random # , and all installed
    package you need

# By default, the run script show only the result of
    the last expression. If you want to check results
    in some specific steps, you should ask for
    functions 'println', 'show', etc. If you want to
    hide even the last expression, you can append the
    ';' at the end of command line.

pos_response="No problem."
println(pos_response)
neg_response="Sorry."
```

Running

- We can set and use a function in script:

```
# script.jl

function greet(name)
    println("Hello, $name!")
end

greet("Julia")
```

- We can change the directory in REPL to the script located in:

```
julia> cd("/path/to/new/directory")
```

- Then, we can run the scrpit in REPL:

```
julia> include("/path/to/script.jl")
Hello, Julia!
```

Packages

- Once we download and install Julia, we can access the so-called standard library in Julia.
- Similarly to other free software, like R or Python, Julia allows external contributors to build and make available third party packages. These are available through Git, see https://github.com/.
- The list of currently available Julia packages can be found at: http://pkg.julialang.org.

Packages

- Some useful packages for economists from Julia's Standard Library (using without adding):
  - LinearAlgebra: Provides essential linear algebra operations such as matrix factorizations, eigenvalues, and solving linear systems.
  - Statistics: Offers basic statistical functions like mean, median, variance, and correlation coefficients.
  - Random: Facilitates random number generation and sampling from various distributions, crucial for Monte Carlo simulations and stochastic processes.
  - Dates: Handles date and time calculations, useful for time series analysis and economic modeling involving temporal data.
- To start using these packages, we can write:

```
using LinearAlgebra, Statistics
```

Packages

- The baseline Julia installation comes with a built-in package manager called Pkg. We can use when in Julia's package management mode (enterying by pressing ']' and exiting by pressing 'Backspace'):
  1. Add a package:

```
(@v1.10) pkg> add PackageName
```

  2. Update all packages:

```
(@v1.10) pkg> Update
```

  3. Remove a package:

```
(@v1.10) pkg> remove PackageName
```

  4. List installed packages:

```
(@v1.10) pkg> status
```

Packages

- Some useful packages for economists from Julia's Standard Library (using after adding):
  - Distributions.jl: Provides a wide range of probability distributions and functions for generating random variables, essential for statistical modeling and hypothesis testing.
  - DataFrames.jl: Offers tools for working with tabular data, including data manipulation, merging, and statistical operations, critical for data-intensive economic analysis.
  - Optim.jl: Provides optimization algorithms for nonlinear and linear optimization problems, which are frequently encountered in economic modeling and parameter estimation.
  - Plots.jl: A versatile plotting package that supports various plot types and customization options, crucial for visualizing economic data and model outputs effectively.

References

- The most faithful companion:
  - Chat GPT
- Textbook:
  - Caraiani P. Introduction to Quantitative Macroeconomics Using Julia: From Basic to State-of-the-Art Computational Techniques[M]. Academic Press, 2018.
  - ——also there is Chinese version.
- Website for tutorials:
  - https://julia.quantecon.org/
  - https://juliaeconomics.com/
- BBS in Chinese for deeper programming demand:
  - https://cn.julialang.org/JuliaZH.jl/latest/
  - https://discourse.juliacn.com/

Data

Bool	false, true
Int, Int8, ..., Int128	123, 1_000_000, UInt128(2)^127
UInt, UInt8, ..., UInt128	0xff, 0x0012, 0b1011, 0o377
Float64, Float32, Float16	.5, 1.0, 3e6, 2.3f9, NaN, -Inf16
ComplexFloat64	0.0 + 1.0im
RationalInt64	3//4 + 1//2 == 5//4
Char	'a', '\n', '\u20ac'
String	"hi", "I am \"$name \"", "1+1=$(1+1)"
Symbol	:test
VectorInt = ArrayInt,1	[1, 2, 3]
MatrixInt = ArrayInt,2	[1 2; 3 4]
TupleInt64,Char,Bool	(1, 'a' , false)
Nothing	nothing
Missing	missing + 1 == missing

Data

- DataFrame:

```
using DataFrames
df = DataFrame(Name = ["Alice", "Bob", "Charlie"],
    Age = [30, 24, 35], Salary = [70000, 40000,
   90000])
```

- Dictionary Types:

```
params = Dict("alpha" => 0.05, "beta" => 0.9)
```

- Statistical Distributions:

```
using Distributions
dist = Normal(0, 1)
sample = rand(dist, 100)
```

LinearAlgebra

- 'dot': Computes the dot product of two vectors.

```
v1 = [1, 2, 3]
v2 = [4, 5, 6]
dot_product = dot(v1, v2)
```

- cross: Computes the cross product of two 3-dimensional vectors.

```
v1 = [1, 0, 0]
v2 = [0, 1, 0]
cross_product = cross(v1, v2)
```

- norm: Computes the norm (magnitude) of a vector.

```
v = [3, 4]
vector_norm = norm(v)
```

LinearAlgebra

- det: Computes the determinant of a matrix.

```
A = [1 2; 3 4]
determinant = det(A)
```

- inv: Computes the inverse of a matrix.

```
inverse_A = inv(A)
```

- rank: Computes the rank of a matrix.

```
matrix_rank = rank(A)
```

- trace: Computes the trace of a matrix (sum of diagonal elements).

```
matrix_trace = tr(A)
```

LinearAlgebra

- eigvals and eigvecs: Computes the eigenvalues and eigenvectors of a matrix.

```
A = [1 2; 2 3]
eigenvalues = eigvals(A)
eigenvectors = eigvecs(A)
```

- svd: Computes the Singular Value Decomposition (SVD) of a matrix.

```
SVD = svd(A)
U, S, V = SVD.U, SVD.S, SVD.V
```

- cholesky: Computes the Cholesky decomposition of a positive definite matrix.

```
chol = cholesky(A)
L = chol.L
```

LinearAlgebra

- \ (Backslash Operator): Solves the linear system Ax = b.

```
A = [1 2; 3 4]
b = [5, 6]
x = A \ b
```

- transpose (or adjoint): Computes the transpose (or Hermitian transpose) of a matrix.

```
At = transpose(A)   # Transpose
Ah = adjoint(A)     # Hermitian transpose
```

- Computes the QR decomposition of a matrix.

```
Q, R = qr(A)
```

CSV

- Write the DataFrame to a CSV file

```
df = DataFrame(Name = ["Alice", "Bob", "Charlie"],
     Age = [30, 24, 35], Salary = [70000, 40000,
    90000])
CSV.write("data.csv", df)
```

- Read the CSV file

```
df = CSV.read("data.csv", DataFrame)
```

Statistics

- Descriptive statistics unction from Library

```
df_ds_function = describe(df)
show(df_ds_function)
```

```
julia> show(df_ds_function)
3×7 DataFrame
 Row │ variable  mean     min     median   max      nmissing  eltype
     │ Symbol    Union…   Any     Union…   Any      Int64     DataType
─────┼──────────────────────────────────────────────────────────────────
   1 │ Name               Alice             Charlie         0  String7
   2 │ Age       29.6667  24      30.0     35              0  Int64
   3 │ Salary    66666.7  40000   70000.0  90000           0  Int64
```

Statistics

- Descriptive statistics Calculated by ourselves

```
mean_age = mean(df.Age) # Calculate Mean
median_age = median(df.Age) # Calculate Median
std_age = std(df.Age) # Calculate Standard Errors
var_age = var(df.Age) # Calculate Variance
min_age = minimum(df.Age) # Calculate Minimum
max_age = maximum(df.Age) # Calculate Maximum
q1_age, q2_age, q3_age = quantile(df.Age, [0.25,
    0.5, 0.75])  # Calculate Quantile
...
df_ds_myself = DataFrame(variables = ["Age", "
    Salary"], mean = [mean_age, mean_salary],
    median = [median_age, median_salary], std = [
    std_age, std_salary], variance = [var_age,
    var_salary], minimum = [min_age, min_salary],
    maximum = [max_age, max_salary])
```

Random

```
x = rand()  # Generates a random float in the range
    [0.0, 1.0)
y = randn()  # Generates a random float from a
    standard normal distribution (mean=0, std=1)
arr_normal = randn(5)  # Generates an array of 5
    random floats from a standard normal distribution
matrix_normal = randn(2, 3)  # Generates a 2x3 matrix
    of random floats from a standard normal
    distribution
perm = randperm(10)  # Generates a random permutation
    of integers from 1 to 10
original_arr = [1, 2, 3, 4, 5]
shuffled_arr = shuffle(original_arr)  # Randomly
    shuffles the elements of the array
Random.seed!(1234)  # Sets the seed for the random
    number generator
```

Dates

There are some useful functions in Random library:

```
# Returns the current date and time
time_now = now()

# Creates a Date object for July 10, 2024
d = Date(2024, 7, 10)

# Creates a DateTime object for July 10, 2024, 10:30
dt = DateTime(2024, 7, 10, 10, 30, 00)

# Subtracts 10 days from the date d
d_minus_10 = d - Day(10)

# Adds 5 hours to the DateTime dt
dt_plus_5hrs = dt + Hour(5)
```

Dates

There are some useful functions in Dates library:

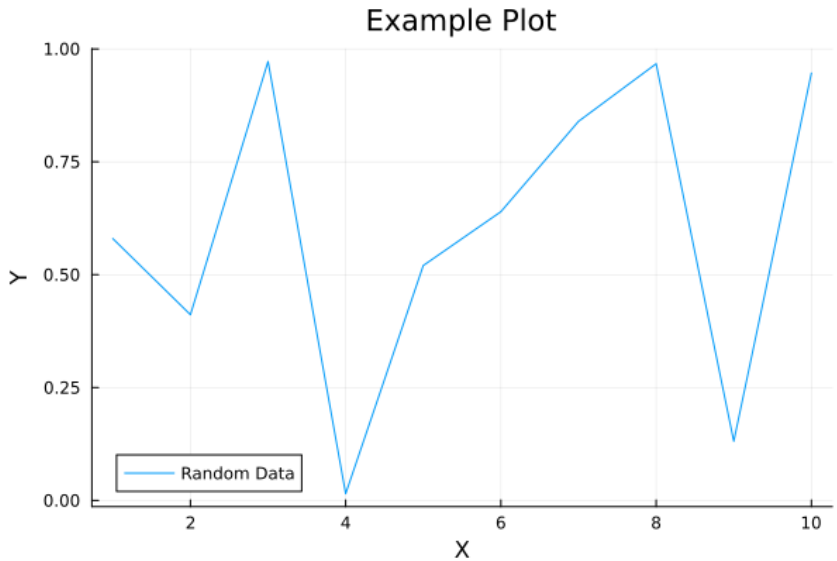
```
# Date, DateTime and string
formatted_datetime = Dates.format(dt, "yyyy-mm-dd HH:
    MM:SS")  # Formats the datetime dt as "2024-07-10
    10:30:00"

parsed_datetime = DateTime("2024-07-10 10:30:00", "
    yyyy-mm-dd HH:MM:SS")  # Parses the string
    "2024-07-10 10:30:00" into a Date object


# Get the part from a Date or DateTime
d_year = year(d)  # Extracts the year
d_month = month(d)  # Extracts the month
d_day = day(d)  # Extracts the day
d_hour = hour(dt)  # Extracts the hour
d_minute = minute(dt)  # Extracts the minute
d_second = second(dt)  # Extracts the second
```

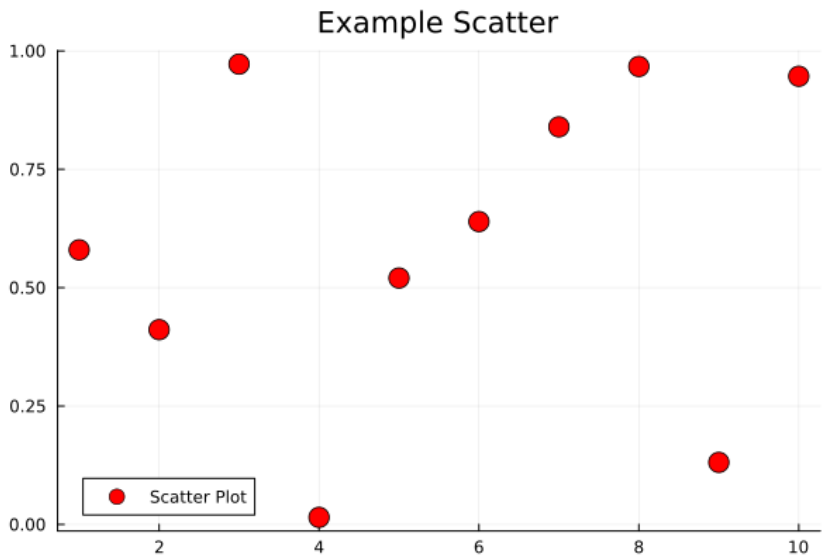
Plots

There are examples about ploting.

```
# Generate some example data
x = 1:10
y = rand(10)

# Plot a line graph
plotxy=plot(x, y, label="Random Data", xlabel="X",
    ylabel="Y", title="Example Plot")
# Plot a scatter plot
scatterxy=scatter(x, y, label="Scatter Plot",
    markersize=8, marker=:circle, color=:red, title="
    Example Scatter")

# Save the plot as a PNG file
savefig("plot_example.png")

# Display the plot
display(plotxy)
```

Plots


Example Plot
1.00
0.75
0.50
0.25
0.00
Y
Random Data
2
4
6
8
10
X


Example Scatter
1.00
0.75
0.50
0.25
0.00
Scatter Plot
2
4
6
8
10

Example

- Consider a Neoclassical growth (RCK) model, the representative household maximizes present as well as expected future utility.

$$\max \quad \sum_{t=0}^{\infty} \beta^t u(c(t)), \tag{1}$$

where β denonting the discount factor.

- The budget constraint of the household as follow:

$$k_{t+1} = f(k_t) - c_t - \delta k_t, \tag{2}$$

where δ is depreciate rate.

Example

- Without loss of generality, we set utility function and production function be:

$$u(c_t) = \frac{{c_t}^{1-\sigma}}{1-\sigma}$$

$$f(k_t) = {k_t}^{\alpha}$$

- The Bellman function and first-order conditions are:

$$v(k_t) = \max_{k_{t+1}} \left\{ \frac{({k_t}^{\alpha} - \delta k_t - k_{t+1})^{1-\sigma}}{1-\sigma} + \beta v(k_{t+1}) \right\}$$

- According to mapping contraction theorem, we can solve this problem by iterating computation program.

Code

The computation can be performed in Julia:

```
# Define parameters
numits = 240 # number of interation
beta = 0.98
sigma = 0.5
delta = 0.1
alpha = 0.4
kss=((1-beta*(1-delta))/(alpha*beta))^(1/(alpha-1))

# Define array of variables
numk = 1000 # number of k
k0 = range(0.01*kss,2*kss,numk)
vlast = zeros(numk) # value
k1 = similar(k0) # the next period capital
```

Code

```
# Define Value function
function valfun(kt1,kt,alpha=alpha,delta=delta,sigma=
    sigma,k0=k0,vlast=vlast)
    # kt1 is control variable
    # kt is state variable
    itp = LinearInterpolation(k0, vlast)
    vkt1 = itp(kt1)
    ct=kt^alpha-kt1+(1-delta)*kt;
    if ct<=0
        val=-888-800*abs(ct); # keep consumption from
            going negative
    else
        val=ct^(1-sigma)/(1-sigma)+beta*vkt1;
    end
    return -val;
    # change value to negative since "optimize" finds
        minimum
end
```

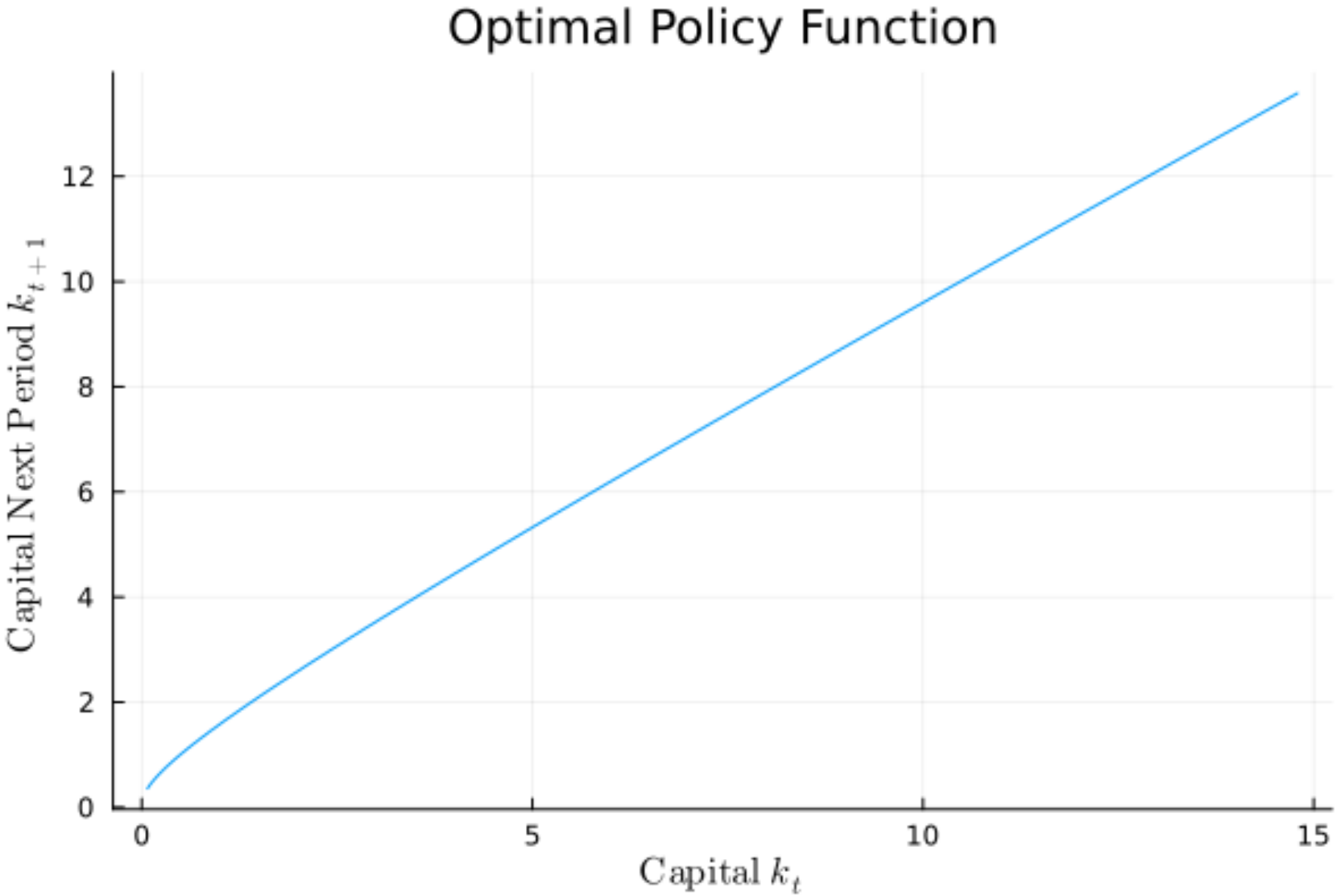
Code

```
# Calculate recursively
for i = 1:numits
    for j = 1:numk
        kt = k0[j]
        result = optimize(kt1 -> valfun(kt1,kt),
            minimum(k0), maximum(k0), GoldenSection())
        k1[j] = result.minimizer
        vlast[j] = -result.minimum
    end
end
```

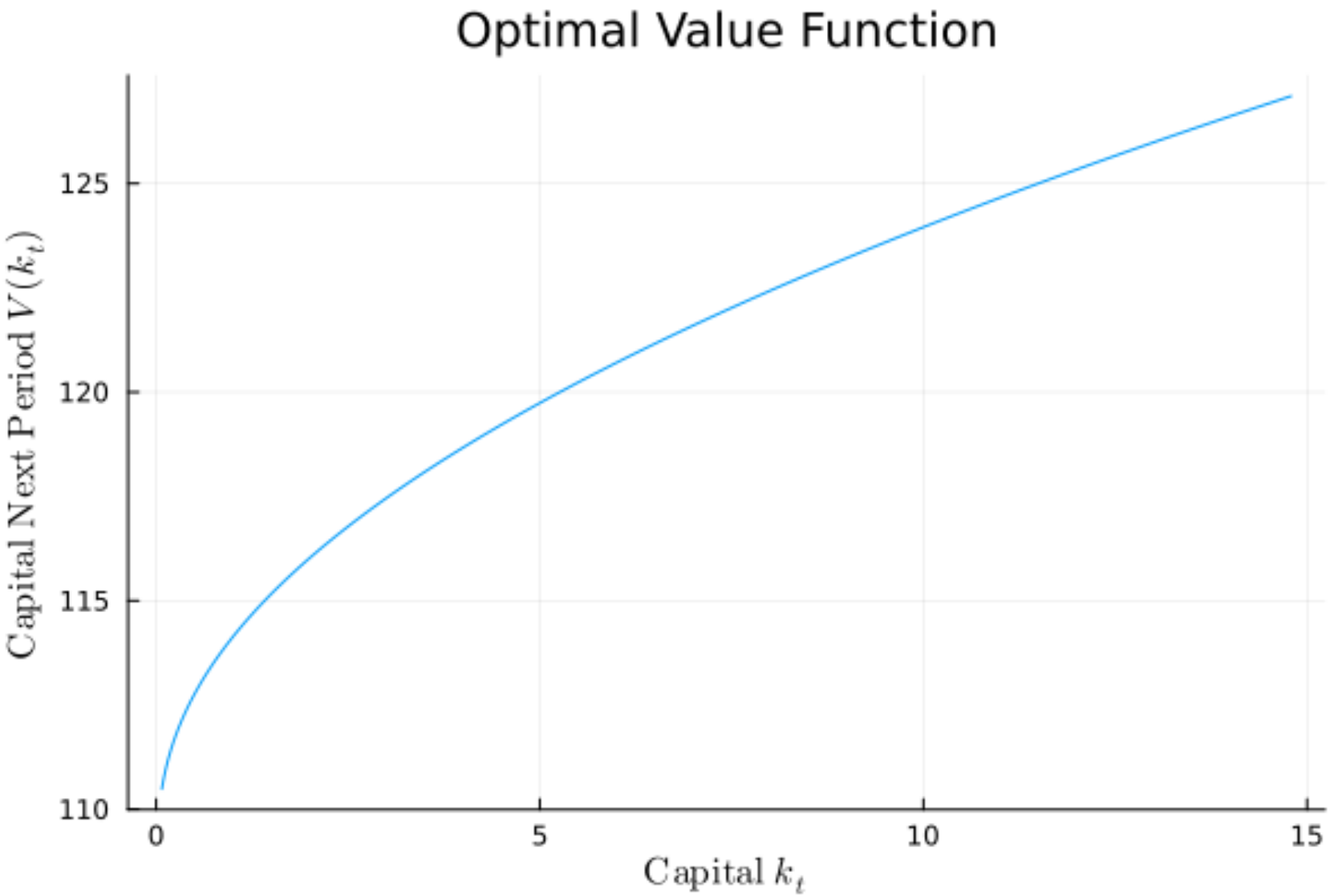
Code

```
# Plot policy function and value fucntion
v = vlast
c1 = k0.^alpha .-k1 .+ (1-delta)*k0;

figure_policy_function=plot(k0, k1, xlabel="Capital",
    ylabel="Capital Next Period", title="Optimal
    Policy Function")
figure_value_function=plot(k0, v, xlabel="Capital",
    ylabel="Value", title="Optimal Value Function")

display(figure_policy_function)
display(figure_value_function)
```

Results


Optimal Policy Function
Capital Next Period k_{t+1}
0
2
4
6
8
10
12
0
5
10
15
Capital k_t

Results


Optimal Value Function
Capital Next Period $V(k_t)$
125
120
115
110
0
5
10
15
Capital k_t

Dynare

- Now, Dynare can run on Julia.
- The most part of .mod file is same as on Matlab.
- Run .mod file in Julia REPL:

```
pkg> add Dynare
Julia> using Dynare
Julia> context = @dynare "RBC.mod"
```

Summary

- Compared to traditional programming languages used in economics (such as Matlab, R, and Python), Julia processes large datasets and computationally intensive tasks more quickly.
- Julia has a concise and intuitive syntax, making it easy to learn and use, allowing economics researchers to write and debug code quickly.
- Julia's interactive programming environment (such as VS Code) enhances research flexibility and reproducibility, facilitating the sharing and presentation of economic research results.