Julia Tutorial

Motoaki Takahashi

December 1, 2025

1 Installing Julia and VS Code

The first step to compute a spatial model with Julia is to install Julia. It is convenient to use Visial Studio Code (henceforth, VS Code) to write Julia codes. So we also install VS Code. Both software packages are free. And another important step is to install the Julia extension to VS Code. Because I'm using Windows, I primarily explain how to install Julia, VS Code, and the Julia extension for VS Code in Windows.

Here we install the latest version of Julia (as of December 1, 2025). Access
https://julialang.org/install/
If you are using Windows, the webpage recommend to install Julia using the MSIX App Installer.

VS Code is downloadable at
https://code.visualstudio.com/Download

After you install VS Code, you need to set up the Julia extension on VS Code. The steps for this is detailed at
https://code.visualstudio.com/docs/languages/julia
Usually the Julia extension on VS Code automatically find your Julia exe file, but you may need to specify the path of the Julia exe file on VS Code.

2 A very brief introduction to Julia

2.1 Basic matrix algebra

What we need to do to compute an equilibrium of the EK model is calculations of vectors and matrices.

Before anything, set a current directory

```
cd("C:\Users\takah\Dropbox\handai\")
```

Make a three-dimensional vector of ones. Name it `a`.

```
a = ones(3)
size(a)
```

Notice that in Julia, vectors are column vectors. So `a` is a 3×1 vector. Let's make a vector whose elements are all different.

```
b = [1; 2; 3]
c = [1, 2, 3]
b == c
a + b
```

To make a column vector, you put `;` or `,` after numbers.

How about row vectors? Let's make a three-dimensional row vector of ones and a row vector consisting of 1, 2, 3.

```
d = ones(3)'
e = [1 2 3]
```

In the first line, `'` after `ones(3)` means transposition. Note that to define row vector `e`, we write neither `;` nor `,`. Just empty spaces to align numbers.

To compute the EK model, you need not only vectors but also matrices. Let's make a 2×3 matrix whose elements are all zero.

```
A = zeros(2, 3)
size(A)
```

Notice that the first element, 2, in the `zeros` function specifies the number of *rows* and the second element, 3, the number of *columns*. Let's make a matrix whose elements are all different.

```
B = [1 2 3; 4 5 6]
```

This matrix B means

$$B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}.$$

So, the first three elements before `;` is the first row, the next three elements after `;` the second. Let's make a different matrix.

```
C = [-1 -3 -5; -7 -9 -11]
```

Then let's premultiply a matrix to a vector.

```
A * b
B * b
```

Notice that `A` and `B` are both 2×3 vectors, and `b` is a 3×1 vector, so these two multiplications are well-defined.[1]

Now we introduce a very important concept (lingo?) in Julia. It is "broadcast."

```
abs(C)
abs.(C)
```

I think the first line yields an error, the second line does not. The difference is only `.` between `abs` and `(C)`. This `.` is a broadcast. In short, broadcasts apply specified manipulation to a vector or a matrix *element-by-element*. `abs` takes an absolute value of a scalar and is a function whose argument must be a scalar. But the argument, `C`, is a matrix. To apply the `abs` to each element of matrix `C`, we need to explicitly write the broadcast symbol `.`.

Another usage of broadcasts is to get element-by-element division or multiplications of vectors or matrices.

```
D = B ./ C
E = B .* C
G = B * C
```

The first two lines yield 2×3 matrices, and the last line errors. The first line computes

$$D = \begin{bmatrix} -1 & -2/3 & -3/5 \\ -4/7 & -5/9 & -6/11 \end{bmatrix}.$$

Since matrices `B` and `C` have the same dimensions, `D` and `E` are well-defined. But `B*C` is not well-defined.[2]

2.2 Loop

Sometimes you want to do the same operation to multiple objects. The loop helps you do so. We discuss two kinds of loops: the for loop and the while loop.

```
H = zeros(size(D))
for i in 1:3
    H[1, i] = abs(D[1, i])
end
```

In the first line, we made a matrix whose elements are all zeros and which have the same dimensions as `D`. We named this matrix `H`. Then, for $i = 1, 2, 3$, we replaced the $(1, i)$ element of `H` with the absolute value of the $(1, i)$ element of `D`.

We can insert a for loop in a for loop.

[1] If dimensions are not aligned to produce well-defined multiplications, Julia spells an error message.

[2] See a basic textbook of linear algebra if you do not know the reason.

```
J = zeros(size(D))
for i in 1:3
    for j in 1:2
        J[j, i] = abs(D[j, i])
    end
end
J == abs.(D)
```

We first made matrix `J` whose elements are all zeros and which have the same dimensions as `D`. Then we replaced each element in `H` with the absolute value of the corresponding element in `D`. By "corresponding", I meant the same indices for the row and the column.

We move on to the while loop.

```
count = 0
tol = 0.2
maxit = 1000

K = [1, 2]
L = [3, 7]

dif = minimum(abs.(K - L))

while dif > tol && count < maxit
    K = K + fill(0.1, 2)
    dif = minimum(abs.(K - L))
    count = count + 1
end
```

This while loop means that we keep adding 0.1 to any element in `K` if `dif` (in this case, the absolute distance between `K` and `L`) is greater than the predetermined tolerance value `tol` (0.2), and if the number of iteration `count` is smaller than the predetermined upper bound for iterations `maxit` (1000).

2.3 Function

It is necessary to define a function that maps a vector to a vector to solve the EK model.

```
function my_function(a)
    output = a + ones(size(a))
    return output
end
```

```
a_1 = [1, 3, 4]
a_2 = my_function(a_1)
```

In the code above, we define a new function named `new_function`. There, we add the vector whose element is all one to a given input vector, say `a`. If $a_1 = (1, 3, 4)^T$ is an input,[3] then the function yields an object named `a_2`.

2.4 Plots

Sometimes you want to make graphs. To do so, you need to install a package.

```
using Pkg
Pkg.add("Plots")
using Plots
```

The second line is to install package "Plots." The third line declares that we use the package "Plots" from now on. Once you installed the package, you do not have to run the first line again.

Let's draw a simple graph.

```
x = 0.1:0.1:1

y1 = x .^ 2
y2 = x .^ (1/2)

plot(x, y1)
plot!(x, y2)
```

For the domain of `x` from 0.1 to 1 with the step size 0.1, we've drawn $y_1 = x^2$ and $y_2 = \sqrt{x}$. In the last line, the exclamation `!` after `plot` declares that you add a line plot to your existing graph. You may want to add legends and save the graph as a pdf file in your current directory.

```
plot(x, y1, label = "x squared", xlabel = "x", ylabel = "y")
plot!(x, y2, label = "sqrt x")
savefig("figure1.pdf")
```

If you want to dive deep into computation with Julia, I recommend you to read https://julia.quantecon.org/intro.html

[3] For a vector x, x^T denotes the transpose of x.