# Comparison Between Matlab and Julia

Han Wang, Jianping He

June 22, 2020

**Abstract**

This article gives some basic usage of Julia. The illustration is mainly demonstrated by comparison with Matlab due to the high similarity between them. Benchmarks that do test compiler performance of common used function with Julia, Matlab and C++ are also given.

## 1 Introduction

Nowadays, some universities in China has been forbidden to use product by MATHWORK, e.g. Matlab. Many professors and students were complaining about the decision because they use Matlab daily for their research and homework. Well, although we have alternative programming choice like C++ and python, Matlab do really has its own irreplacable good characteristics in computation and simulation. Under this circumstance, Julia was developed as an open source, free computation language to replace Matlab. Users can easily use it with different IDE like vscode and ATOM. In this article, we want to introduce some basic usage of it and compare the performance with different languages.

## 2 Basic Computation

In this section, we want to illustrate some basic usage in computation like matrix multiply, change entries in loop, etc. It should be noted that most of the operations are similar to Matlab so we will omit them in this article.

### 2.1 Variable Declaration

Variable types are similar in Matlab and Julia, e.g. int64, float32 etc. One can simply declare a variable by assignment: a=1 or b=[1 2;3 4]. It should be noted that default data type are different in Matlab and Julia. Here we give an example to help with illustration:

```
julia> a=1
1

julia> typeof(a)
```

```
Int64

julia> a=1.0
1.0

julia> typeof(a)
Float64
```

In Matlab:

```
matlab> a=1;
1

matlab> class(a)
double
```

In Matlab, number will be type ”double” in default, and if you want to get ”int” you will need to declare explicitly:

```
matlab> a=int64(1);
1

matlab> class(a);
int64
```

If you want to convert the variable from one type to another, convert() can help you with that:

```
julia> a=1
1

julia> convert(Float64,a)
1.0

julia> typeof(convert(Float64,a))
Float64
```

## 2.2 Matrix Declaration

Matrix operation can be one of the most important application in the field of automation. Matlab has many useful and fast matrix operation functions inserted. Julia has realized many basic functions with similar name and usage to Matlab. Well...we will start with matrix declaration. In Matlab, you can simplily declare a matrix by value assignment:

```
matlab> a=[1  2;
    3  4;
    5  6];
```

Specific entries in matrix can be selected by brackets:

```
matlab> a(1,1);
1
```

However, in Julia you will need square brackets to do so:

```
julia> a[1,1]
1
```

Some people make a joke on this characteristic, if you know Matlab, you will just need to change your brackets into square brackets, then you also know Julia. Julia also support undefined matrix and blank matrix:

```
julia> a=Matrix{Nothing}(nothing,2,3)
2*3 Array{Nothing,2}:
 nothing  nothing  nothing
 nothing  nothing  nothing

julia> a=[]
0-element Array{Any,1}
```

One interesting feature is one dimension vector and one dimension matrix is different in Julia, you can have similar row vector in different data type:

```
julia> a=Array{Float64,1}(undef,3)
3-element Array{Float64,1}:
 8.1520942e-316
 1.25931918e-315
 1.2498317e-315

 julia> a=Array{Float64,2}(undef,3,1)
3*1 Array{Float64,2}:
 8.19715696e-316
 1.091240025e-315
 1.091354727e-315
```

This feature can cause errors in some operations like transpose. In Matlab if you want to get a transpose of a matrix or vector, you can simply use quote to do so, but in Julia you will need function transpose():

```
julia> a=ones(3,1)
3*1 Array{Float64,2}:
 1.0
 1.0
 1.0

julia> c=zeros(1,3)
1*3 Array{Float64,2}:
 0.0   0.0   0.0

julia> transpose!(c,a)
```

```
1*3 Array{Float64,2}:
 1.0  1.0  1.0

julia> b=[1,1,1]
3-element Array{Int64,1}:
 1
 1
 1

julia> transpose!(b,a)
ERROR: DimensionMismatch("transpose")
```

## 2.3 Basic Functions

Julia has amount of functions and operators with similar name to Matlab. Details can be reffered to Julia's official tutorial. In this subsection we only give some notes to the functions. In Julia, input data type are more strict than that in Matlab. For example ones():

```
julia> ones(3.0,3)
ERROR: MethodError: no method matching ones(::Float64, ::Int64)

matlab> ones(3.0,3)

ans =

     1     1     1
     1     1     1
     1     1     1
```

This problem sounds like weird but in many cases it will happen, especially when the dimension of matrix is calculated through some equations. Similarly, when selecting entries in matrix this problem will also throw.

## 2.4 User Defined Functions

The biggest difference between Julia and Matlab in user defined functions lie in return value. In Matlab, you will have to create a file with similar name to functions and declare the return value in the front. In Julia, the feature is more similar to c++, functions can be created in one file and you can use return to get return value. Here we give an example:

```
function print_some_thing(name::String,character::String)
    return name*""*" is so "*""*character
end

julia> a=print_some_thing("Han","handsome")
"Han is so handsome"
```

Julai also allows unknown number of parameter inputs. In Matlab, this function is realized through ”varargin” as input. Here we give an example in Julia:

```
function multi_input(x...)
    nargin=length(x)
    a=nargin>2 ? x[2] : x[1]
    return a
end

julia> a=multi_input(1,2,3)
2

julia> a=multi_input(1)
1
```

### 2.5 Variable Scope

There are many differences in this part. In Matlab, value assignment is value passing but in Julia, for matrix is pointer passing:

```
a=[1 2 3]
b=[2 3 4]
a=b
b[1]=0

julia> a
1*3 Array{Int64,2}:
 0  3  4
```

As to the variable living scope, scalars are regarded as local variable which cannot be changed in loops without a global declaration:

```
matrixA=[1 1;1 1]
a=1
for i=1:2
    matrixA[i,1]=0
    global a=a+1
end
```

## 3 Optimization

The author has done some work on optimization with Julia in Prof. Francesco Bullo's group. In Matlab, multiple optimization toolbox can be used like fmincon, CVX, etc. JuMP is a collector of Julia's optimization toolbox. Here we give a brief guide for using JuMP.

### 3.1 Solver

Multi solvers has been developed in JuMP like SQP, Ipopt(interior point), one can chose proper optimizer with:

```
model=Model(with_optimizer(Ipopt.Optimizer))
```

Here we use Ipopt as our solver because we want to solve a non-convex problem. Comparison and introduction to solvers can be found here: https://jump.dev/JuMP.jl/v0.20.0/solvers/index.html.

### 3.2 Variable

JuMP supports multiple variables in optimization problem. It should be noted that dislike fmincon, initial input value is required.

```
@variable(model,x[i=1:n],start=x0[i])
```

### 3.3 Object

One can set a function as optimization object.

```
register(model, :op_myfun, n, op_myfun; autodiff = true)
@NLobjective(model,Min,op_myfun(x...))
```

### 3.4 constraints

JuMP support both linear and non-linear constraints:

```
@constraint(model,A_ineq*x.<=b_ineq)
@constraint(model,A_eq*x.==b_eq)
@NLexpression(model,my_expr,nonlcon1(X...))
@NLconstraint(model,my_constr,nonlcon1(X...)==0)
```

Details of the code can be found in: https://github.com/HanWang99/RoboSurv/tree/master/Robot-Surveillance-Julia

# 4 Misc

### 4.1 Package Management

Dislike Matlab, Julia does not have an official software. Every package except for Base.jl need to be added by users. For example, if you want to use linear algebra function like diag(), you will need to add the package firstly:

```
Pkg.add("LinearAlgebra")
```

Then, every time when you need to use functions included in package "LinearAlgebra", you will need to add

```
using LinearAlgebra
```

in the front of your file. You can add our package to your computer, and use

```
Pkg.add("https://github.com/HanWang99/RoboSurv.git")
using MarkovChain
```

to use our package.

## 4.2 Start Up

The start up speed of Julia is very slow...Just be patient and wait.