

Julia as a calculator and CAS

In Major Rough Draft Stage

Daniel Eric Smith

January 7, 2025

Contents

I	The Julia Language	5
1	Starting Julia	7
1.1	Installing Julia	7
1.2	Starting Julia	7
2	Calculator	9
2.0.1	Arithmetic	9
2.0.2	Math Functions	11
2.1	Types	13
2.1.1	Integers	13
2.1.2	Floats	14
2.1.3	Rationals	14
2.2	Matrices	14
2.2.1	Arithmetic	15
2.2.2	Transpose	17
II	CAS	19
3	VS Code	23
4	Dash	25

Part I

The Julia Language

Chapter 1

Starting Julia

1.1 Installing Julia

In case you want to install Julia on your own machine (it's on all the lab computers) go to the following webpage: [Download Julia](#)

As of 2024 a great series of videos are from the Dabbling Doggo's YouTube channel: [doggo dot jl](#). In particular the videos of interest are:

- Installing Julia
- Installing VS Code
- How to use Julia in VS Code

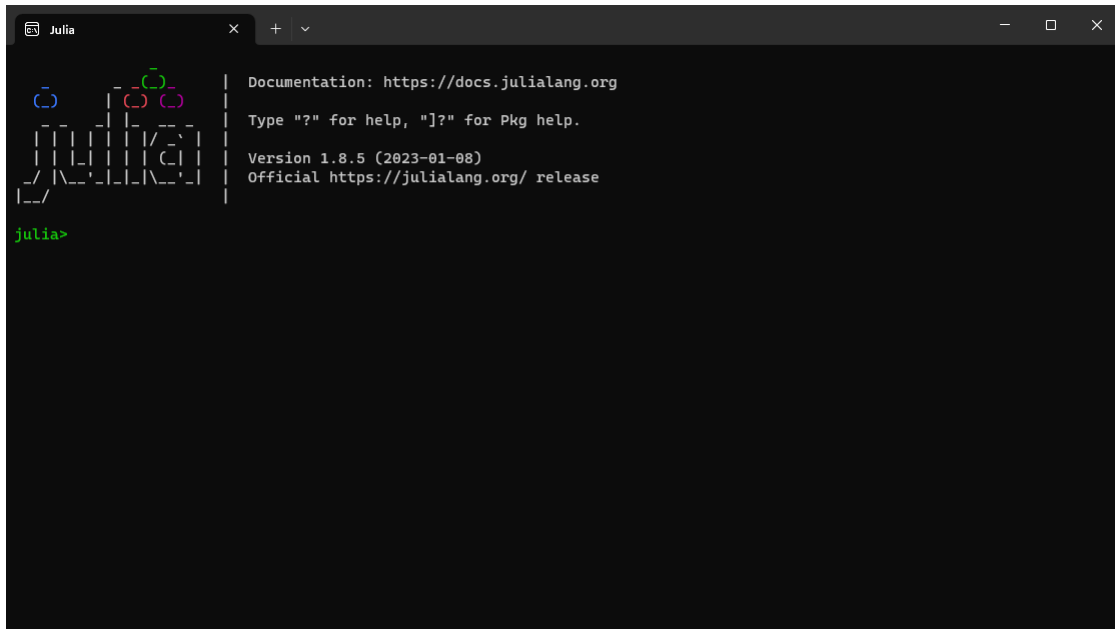
Warning: he has older videos on how to setup Julia so make sure you look at the latest playlist.

1.2 Starting Julia

To start Julia either double click the Julia icon on the desktop:



or hit the start button on Windows and type “Julia”. This will start a new Julia session which will look something like this:



```
Documentation: https://docs.julialang.org
Type "?" for help, "]" for Pkg help.
Version 1.8.5 (2023-01-08)
Official https://julialang.org/ release

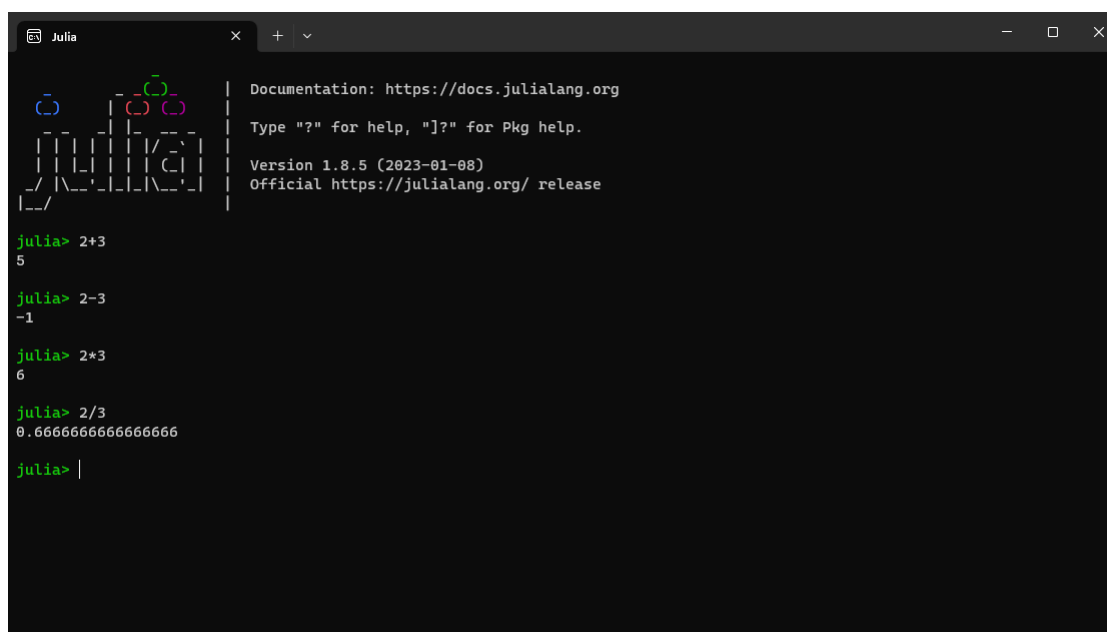
julia>
```


Chapter 2

Calculator

2.0.1 Arithmetic

We can use the standard symbols to do basic arithmetic in Julia: $+$, $-$, $*$, $/$.

A screenshot of the Julia REPL (Read-Eval-Print Loop) window. The window has a dark theme. On the left, there's a logo for Julia. The main area shows the following text:

```
julia> 2+3
5
julia> 2-3
-1
julia> 2*3
6
julia> 2/3
0.6666666666666666
julia> |
```

On the right side of the window, there's a sidebar with the following text:

```
Documentation: https://docs.julialang.org
Type "?" for help, "]?" for Pkg help.
Version 1.8.5 (2023-01-08)
Official https://julialang.org/ release
```

Suppose we want to add one to $2/3$ and have already evaluated $2/3$:

```
julia> 2/3
0.6666666666666666
```

We can get the last result by typing:

```
julia> ans
0.6666666666666666
```

So to add 1 to it we can just type:

```
julia> ans + 1
1.6666666666666665
```

We can also raise numbers to exponents using the `^` character:

```
julia> 2^3
8
```

And as usual, we can use parenthesis to group expressions:

```
julia> 3*(1+1)
6
```

Note that if we start with two integers certain operators return an integer:

```
julia> 2+3
5
```

```
julia> 2-3
-1
```

```
julia> 2*3
6
```

But sometimes we will get a floating point number back instead:

```
julia> 1/2
0.5
```

```
julia> 1/3
0.3333333333333333
```

```
julia> 1/2 + 1/3
0.8333333333333333
```

Suppose we need to work with exact rational number arithmetic. Then Julia has another operator denoted `//` to denote that you want exact rational arithmetic:

```
julia> 1//2
1//2
```

```
julia> 1//3
1//3
```

Now if we add these numbers we get an exact answer:

```
julia> 1//2 + 1//3
5//6
```

2.0.2 Math Functions

Julia can handle Unicode characters. This is nice since we can write code that looks more like our math notation. As an example, Julia knows that π is:

```
julia>  $\pi$ 
 $\pi$  = 3.1415926535897...
```

To type π in the terminal (it works the same in VS Code) you type `\pi` and then hit the TAB key (before hitting the SPACE key) and the REPL will convert the code `\pi` to the Unicode character π .

Since Julia is a computational language many of the common math functions are built into the core language. So you don't need to import packages or qualify them with a namespace. For example the square root of 2 can be found as shown below:

```
julia> sqrt(2)
1.4142135623730951
```

```
# this is a comment, to get  $\sqrt{\phantom{x}}$  you type \sqrt then hit TAB
julia>  $\sqrt{2}$ 
1.4142135623730951
```

For those of you familiar with Python this would require you to do the following:

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
```

While there is a cube root function `cbrrt(x)`, all other roots will require you to rewrite the root with fractional exponents:

```
julia> cbrt(8)
2.0
```

```
julia> 8^(1/3)
2.0
```

```
julia> 16^(1/4)
2.0
```

There are two other main functions that we use in this class: the natural log and the exponential functions. The exponential function is `exp(x)`. Unfortunately the way almost (if not) all programming languages handle logarithms can be confusing when switching between the programming language and written math. The natural log is written as $\ln x$ or $\ln(x)$, but programming languages write this as `log(x)`. But recall that in written mathematics $\log(x) = \log_{10}(x)$ and $\ln(x) = \log_e(x)$.

```
julia> exp(1)
2.718281828459045
```

```
julia> log(1)
0.0
```

```
julia> e = exp(1)
2.718281828459045
```

```
julia> log(e)
1.0
```

```
#to get the common log you put the base first
# and the argument second
julia> log(10, 100)
2.0
```

Note that in the third interaction I bound the result of `exp(1)` to the variable `e`. This allowed us to use this result in a later calculation `log(e)` which resulted in `1.0` since `e` is the base for the natural logarithm.

2.1 Types

We will not have to pay attention to types for most of this class, but there are two reason we need to talk about them:

1. there will be a few occasions where we will need to think about how the computer is representing our numbers
2. Julia keeps track of types and sometimes reports them so it would be nice to know what it's talking about.

2.1.1 Integers

The Integer type is how the computer represents the mathematical concept of an integer. The mathematical concept of an integer are numbers such as

$$\dots, -3, -2, -1, 0, 1, 2, 3, \dots$$

There are an infinite number of integers. But since computer memory is finite, computers can only represent a finite number of integers.

If your computer is a 64-bit computer then Julia will automatically use 64-bit integers. These have a range of -2^{63} to $2^{63} - 1$. These should be more than large/small enough for what we will need for this class.

If we add, subtract, or multiply integers then we get back integers:

```
julia> 2+3
5
```

```
julia> 5-7
-2
```

```
julia> 5*7
35
```

But if we try to divide integers we should get an integer back

```
julia> 10/5
2.0
```

But wait 2.0 isn't an integer. The operator `/` returns floating point numbers. (We will discuss floating point numbers below.) There is an operator that does return integers when you divide 10 by 5:

```
julia> 10 ÷ 5
2
julia> 10 ÷ 3
3
```

In the second example we get 10 divided by 3 is somehow 3. So what \div is doing is divide 10 by 3 and truncating the result. We shouldn't need this so let's move onto floating point numbers.

2.1.2 Floats

2.1.3 Rationals

2.2 Matrices

A matrix is a rectangular array of numbers. Suppose we have the 2×3 matrix (2 rows and 3 columns)

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

We make this matrix in Julia by separating each entry in a row by a space and each row by a semicolon:

```
julia> M = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

Julia returns the matrix but it also returns 2×3 **Matrix{Int64}**. This tells us that what we entered is a 2×3 matrix of type **Int64**.

Let us make two more matrices but this time they will both be 2×2 :

$$F = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 1.4 \end{bmatrix}, \quad R = \begin{bmatrix} 1/1 & 1/2 \\ 1/3 & 1/4 \end{bmatrix}$$

```
julia> F = [0.1 0.2; 0.3 0.4]
2×2 Matrix{Float64}:
0.1  0.2
0.3  0.4

julia> R = [1//1 1//2; 1//3 1//4]
```

```
2×2 Matrix{Rational{Int64}}:
1//1  1//2
1//3  1//4
```

2.2.1 Arithmetic

Since matrices F and R are both 2×2 we can add them:

```
julia> F + R
2×2 Matrix{Float64}:
1.1      0.7
0.633333 0.65
```

subtract them:

```
julia> F - R
2×2 Matrix{Float64}:
-0.9      -0.3
-0.0333333 0.15
```

and multiply them:

```
julia> F*R
2×2 Matrix{Float64}:
0.166667  0.1
0.433333  0.25
```

But since M and F are not the same size we cannot add them:

```
julia> M + F
ERROR: DimensionMismatch: dimensions must match: a has dims
↳ (Base.OneTo(2), Base.OneTo(3)), b has dims (Base.OneTo(2),
↳ Base.OneTo(2)), mismatch at 2
Stacktrace:
 [1] promote_shape
@ .\indices.jl:178 [inlined]
 [2] promote_shape(a::Matrix{Int64}, b::Matrix{Float64})
@ Base .\indices.jl:169
 [3] +(A::Matrix{Int64}, Bs::Matrix{Float64})
@ Base .\arraymath.jl:14
 [4] top-level scope
@ REPL[7]:1
```

The key thing to notice in this error message is:

```
julia> M + F
ERROR: DimensionMismatch: dimensions must match: ...
```

Recall that we can only add or subtract matrices if they are exactly the same size. So if we wanted to add a matrix to M , which is 2×3 , we would need another matrix N that is also 2×3 .

```
julia> N = [10 11 12; 13 14 15]
2×3 Matrix{Int64}:
10  11  12
13  14  15
```

```
julia> M + N
2×3 Matrix{Int64}:
11  13  15
17  19  21
```

```
julia> N - M
2×3 Matrix{Int64}:
9   9   9
9   9   9
```

Can we multiply M and F ? Lets try:

```
julia> M*F
ERROR: DimensionMismatch: matrix A has dimensions (2,3), matrix B has
↪ dimensions (2,2)
```

doesn't look like it, but what if we try the other way around?

```
julia> F*M
2×3 Matrix{Float64}:
0.9  1.2  1.5
1.9  2.6  3.3
```

That worked.

2.2.2 Transpose

Another operation that we may not need very often is the transpose of a matrix. But it does come up in your homework on occasion, so let's see how to make Julia do this for us.

What is the *transpose* of a matrix? It is a new matrix where we take the the rows from the old matrix and make them into columns of the new matrix. Recalling that we defined the matrix M as

```
julia> M = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

then it's transposed matrix M^T is

```
julia> transpose(M)
3×2 transpose(::Matrix{Int64}) with eltype Int64:
 1  4
 2  5
 3  6
```

And for F and R

```
julia> F = [0.1 0.2; 0.3 0.4]
2×2 Matrix{Float64}:
 0.1  0.2
 0.3  0.4

julia> R = [1//1 1//2; 1//3 1//4]
2×2 Matrix{Rational{Int64}}:
 1//1  1//2
 1//3  1//4
```

Their transposed matrices F^T and R^T are:

```
julia> transpose(F)
2×2 transpose(::Matrix{Float64}) with eltype Float64:
 0.1  0.3
 0.2  0.4

julia> transpose(R)
2×2 transpose(::Matrix{Rational{Int64}}) with eltype Rational{Int64}:
 1//1  1//3
 1//2  1//4
```

```
1      1//3
1//2  1//4
```

So if the homework were to ask you to do the following: Let

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 6 & 7 \\ 8 & 9 \end{bmatrix}$$

Find $A + B^\top$. We would just need to enter the following into Julia

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
julia> B = [6 7; 8 9]
2×2 Matrix{Int64}:
 6  7
 8  9
```

```
julia> A + transpose(B)
2×2 Matrix{Int64}:
 7  10
10  13
```

Mathematically what's happening is

$$A + B^\top = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 6 & 7 \\ 8 & 9 \end{bmatrix}^\top = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 6 & 8 \\ 7 & 9 \end{bmatrix} = \begin{bmatrix} 1+6 & 2+8 \\ 3+7 & 4+9 \end{bmatrix} = \begin{bmatrix} 7 & 10 \\ 10 & 13 \end{bmatrix}$$

Part II

CAS

The acronym CAS stand for Computer Algebra System. These packages/programs allow the computer to solve many equations exactly. It also allows them to do some calculus. This will allow us to take derivatives and integrals of the functions we are interested in in this class. Note there are some functions that even the CAS can't solve or integrate exactly. By using a CAS we can push some (or a lot of) the grunt work off onto the computer and concentrate more of interpreting the results.

Chapter 3

VS Code

VS Code should already be installed on all of the lab computers. If you want to install this on your own computer the Dabbling doggo has videos on how to install VS Code and on how to use Julia in VS Code:

doggo dot jl: [10x05] How to install VS Code

doggo dot jl: [10x06] How to use Julia in VS Code

Chapter 4

Dash

JSUMS120

$f(x) =$ Domain:

Limits	Sign Charts	Plot Function	Function Summary	Calculus	Help
--------	-------------	---------------	------------------	----------	------

Limits

Choose type of limit:

Limit value: Direction:

Format: Rows: Variable:

