Università della Svizzera italiana | Faculty of Informatics | Institute of Computing CI

Numerical Computing 2022

Julia Essentials – Get Started with Julia

Malik Lechekhab

Overview

Why Julia?

Some of the reasons behind our choice of using Julia:

- **High Performance** Julia is fast. The code is compiled instead of interpreted. Julia outscores Python, R and Matlab in benchmarking tests.
- **High-Level** Julia is easy to use and quick to learn.
- **Purposeful** Julia features a wholesome catalog of science packages.
- **Dynamic Typing** Variables do not have types, values have. These types are determined at runtime.
- **Multiple Dispatch** Functions can have multiple version of themselves.
- **Just-in-time Compiler** Programs are compiled right after they have started executing.
- **Mutable Composite Types** Julia allows to specify composite types that can be modified throughout execution.

The Julia REPL

The Julia executable has a full-featured interactive command-line called REPL (read-eval-print-loop). In addition to allowing quick statements evaluation, it has a searchable history, tab-completion and many other helpful features. The REPL can be started by calling `julia` in a shell or by double-clicking on the executable:

```
(base) [malik@fedora ~]$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.7.2 (2022-02-06)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> print("Hello World")
Hello World
julia> 2+2
4

julia>
```

Installing and Managing Packages

The packages are installed, updated and removed from the Julia's builtin package manager `Pkg`. `Pkg` is included in the Julia's REPL. To enter the package manager interface, press the key] in the Julia's REPL. You should see a similar prompt:

```
(@v1.7) pkg>
```

You can install a package by using the keyword `add`:

```
(@v1.7) pkg> add Example
```

Installed package can be used in your program by adding the line:

```
using Example
```

It is a good practice to activate an environment before installing packages, so you keep a clear view on which package is used in which project. To do so, enter:

```
(@v1.7) pkg> activate .
```

Creating Scalars and Vectors

In order to assign the value 7 to variable `x`, we proceed as follows:

```
x = 7;
```

Notice that we have used the semi-column (;) to suppress the output in the command window. It is good practice to always do it, since printing all your results (e.g., in an iterative method) can have a negative impact on your code's performance.

In order to create a row vector `y` with 3 components, we use the square brackets, with the entries separated by a space:

```
y = [1 2 3];
```

If `y` is a column vector, we need instead to separate the components with a coma or semi-colon:

```
y = [1, 2, 3];   or   y = [1; 2; 3];
```

Creating Matrices and Strings

Suppose we now have a matrix $A \in \mathbb{R}^{3\times 3}$:

$$A = \begin{bmatrix} 1 & 3 & 2 \\ 1 & 1 & 2 \\ 3 & 1 & 4 \end{bmatrix}.$$

We can use the commands in the previous slide to create it, as follows:

```
A = [1 3 2; 1 1 2; 3 1 4];
```

A string is obtained by including the text in double quotes:

```
z = "string";
```

Sometimes, when importing your data, you could be dealing with strings and, in the next slides, you can find some useful commands to process them.

Useful Commands for Dealing with Strings

Let us suppose that you imported some integers and saved them in string called `S`:

```
S = ["1" "2" "3" "4"];
```

You can use the function `parse()` to convert `S` into a numeric array `X`:

```
X = parse.(Int, S);
```

Note that we use a dot. Dots allows Julia to recognize the vectorized nature of an operation at a syntactic level, guaranteeing loop fusion.

Transpose of a Matrix

The nonconjugate transpose of a vector (or matrix) can be obtained by using the function `transpose()`. Starting from the matrix `A` we defined before, if we write:

```
A_transpose = transpose(A);
```

we can verify that we obtain the transpose of matrix `A`:

$$\begin{bmatrix} 1 & 1 & 3 \\ 3 & 1 & 1 \\ 2 & 2 & 4 \end{bmatrix} = A^{\mathsf{T}}.$$

The conjugate transpose of a vector (or matrix) can be obtained by adding a single quote (') after the variable, as follows:

```
A_conjTranspose = A';
```

Inverse of a Matrix and System of Equations

The inverse of a matrix can be computed with the following command:

```
A_inverse = inv(A);
```

However, when solving a system of linear equations, it is better to use the *backslash operator*. In other words, the solution to the system $Ax = b$ can be computed either as:

```
x = inv(A)*b;  or  x = A\b;
```

The former method is slower and less accurate than the latter: this shortcoming of the `inv()` function is particularly evident when the matrix considered has a high condition number.

Matrix Product and Element-wise Matrix Operations

The matrix product between two matrices can be computed in the following two ways:

```
AB = *(A,B);  or  AB = A*B;
```

In practice, we normally use '*' to perform matrix multiplication. It is important to notice that, if we put a dot in front of the symbols used for the standard operations, they will be performed element-wise. Multiplication, division and matrix power will be, instead, computed element by element as:

```
A.*B;
A./B;
A.^B;
```

Selecting Elements Inside a Matrix

While with the square brackets we can create matrices and vectors, we can use the same brackets right next to the variable name to index the array. Let us consider again a matrix A defined as:

```
A = [1 3 2; 1 1 2; 3 1 4];
```

We can select the element in position [2,2] and change its value to 5 as follows:

```
A[2,2] = 5;
```

Please remember that the first argument refers to the row index, while the second to the column index. If we want instead to select the last element of the first row, we can write:

```
A[1,end];
```

In general, the value of `end` corresponds to the index of the last element in the row/column.

The Colon (:) Operator and `range()`

Colons (:) are used as a binary infix operator construct a range with fix step of size 1. For example, we can obtain a range from 1 to 10 (inclusive) by writing:

```
x = 1:10;
```

We can use `collect()` to return an array from the range x containing all the integers from 1 to 10:

```
v = collect(x);
```

We can also specify the distance between one element and the next one, as follows:

```
y = 1:0.01:10;
```

Here `y` will contain all the numbers from 1 to 10, in intervals of 0.1. A similar result can be obtained by using the function `range()`, which takes as third argument the number of points we want to consider:

```
z = range(1, 10, 1000);
```

Vector `z` will contain 1000 equally spaced elements between 1 and 10. Note that the function `LinRange()` should have less overhead than `range()` but won't try to correct for floating point errors.

Indexing with the Colon (:) Operator

An important use of the colon operator consists in the indexing of matrices. Let us consider the matrix A defined before and the following command:

```
A[1,:];
```

The command above selects the first row of A, while we can select the first column by writing:

```
A[:,1];
```

We can also select sub-matrices, like the 2×2 matrix in the lower right part of A:

```
A[2:end,2:end];
```

Please notice how we used the `end` operator to quickly access the last element, without necessary remembering that A is a 3×3 matrix. The column operator can also be used to vectorise a matrix: `A[:]` will return a column vector of length 9 with the elements of A concatenated vertically column by column.

Some Specific Matrices

A $n \times m$ **identity matrix** can be generated with the command (here, you need the `LinearAlgebra` package):

```
A = Matrix(I, n, m);
```

while a **matrix of zeros** can be intuitively obtained as:

```
A = zeros(n, m);
```

The `SparseArrays` package support sparse structures, so we can create a **sparse matrix** as:

```
A = spzeros(n, m);
```

Depending on your problem, using sparse matrices can be highly beneficial and can help you saving a lot of memory, but always remember to double check if the function you are planning to use supports sparse matrices.

Some Specific Matrices

A $n \times m$ **matrix of ones** can be generated with the command:

```
A = ones(n, m);
```

while a **random matrix** can be intuitively obtained as:

```
A = rand(n, m);
```

In case we are interested in having only **random integers** comprised between a and b, we can write:

```
A = rand(a:b, n, m);
```

If we want, instead, to extract the **diagonal** of a matrix and save it in a vector, we can write:

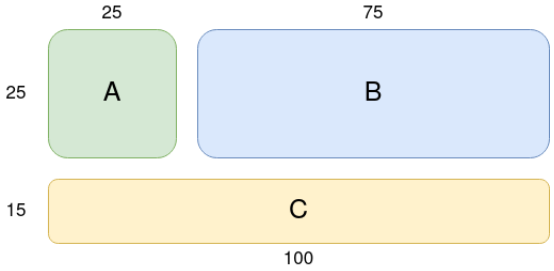
```
x = diag(A);
```

Concatenation of Matrices

The same syntax we used to create vectors and matrices can be used to concatenate matrices:

```
D = [A B; C];
```

for example, concatenates on the first row the 25×25 matrix `A` and the 25×75 matrix `B`, and then concatenates this result vertically with the 15×100 matrix `C`.



Always double check the dimensions of the matrices you are concatenating if you want to avoid nasty bugs in your code with `size(A)`.

Logical Operators

The logical operators return, as usual, the value 0 or 1, depending on whether or not a specific condition is verified. Some examples are: <, >, <=, >=, == (equal) and != (not equal). An important fact is that logical operators can be applied to matrices; so, for example:

```
B = A .> 0.5;
```

creates a logical matrix B, which contains in every position 0 or 1 depending on whether the condition of the entry being bigger than 0.5 is satisfied. Instead the commands:

```
c = sum(filter(x -> x > 0.5, A);  or  c = sum(A[A .> 0.5]);
```

returns a variable c, which counts all the elements of A bigger than 0.5. We introduced also the function `sum()`, which sum all the elements of an array. If we want column-wise or row-wise summation, we can write `sum(A, dims=1)` or `sum(A, dims=2)`.

Conditional Evaluation: If/else

The general structure of the if/else construct is the following:

```
if b == -1
    print("-1");
elseif b == 0
    print("0");
elseif b == 1
    print("1");
else
    print("other");
end
```

where we used the function `print()` to display the result as a string. Note that `&&` and `||` operators in Julia correspond to logical `and` and `or` operations.

Repeated Evaluation: While and For Loops

The general structure for a conditional loop with a while is:

```
while [condition]
    [body];
end
```

while a double for loop can be written as:

```
for [condition]
    for [condition]
        [body];
    end
end
```

Note that the `for` loop can also iterates over any container using the keyword `in`.

Scripts and Functions

You are free to organise your code in the way you prefer, but a tidy way to do it consists in creating a `main.jl` script in which you include the core of your computations and several functions files (maybe included in a '`utilities/`' folder). The general definition of a function in Julia is the following:

```
function functionName(input1, input2, ...)
    [body];
    return output1, output2, ...
end
```

and you normally save it in a file named `functionName.jl`. You can then easily call it inside your `main.jl` script by running the command:

```
include("./utilities/functionName.jl");
functionName(input1, input2, ...)
```

Data Visualization

Producing good plots is an art in itself and the visualisation of your results covers a significant role in all those situations in which you have to communicate with other people (and this includes your final project report and presentation!). Let us start with an example of figure and analyse the different elements:

```
using Plots
x = 1:10; y = rand(10, 2)
plot(x, y, label = ["Line 1" "Line 2"], lw = 3)
z = rand(10);
plot!(x, z)
xlabel!("My x label")
ylabel!("My y label")
title!("Three Lines")
```

Data Visualization

The command `plot()` creates a plot object in which data will be drawn. You can use `plot!()` to add to an existing plot.

A plot has attributes such as `label` to define the legends or `lw` to set the line width. Every attribute can also be applied by mutation the plot with a modifier function. For example, the `xlabel` attribute adds a label under the x-axis. We can either set its value inside the `plot` command or we can use the modifier function to add it after the plot has already been generated with: `xlabel!("My new x label")`.

Apart from the function `plot()`, which is widely used to plot a 2D lines, there are several other plot types. Among those, it is worth mentioning `scatter()`, `histogram()` and `heatmap()`. In addition to the default backend, which is called `gr()`, there are plenty of different backends such as `plotlyjs()`, `pyplot()` and many others!

Remember that Julia has many powerful visualisation tools, so explore and experiment!

Additional Material and References

This set of slides was aimed at introducing Julia and some of its functionalities, along with some practical advices and insights. However, it cannot cover every possible topic and, for this reason, you should check also other references, *in primis*:

https://julialang.org/learning/tutorials/

which consists in a collection of tutorials about all the basic functionalities of Julia. We also invite you to visit this page which gather tips to make your Julia code as fast as possible:

https://docs.julialang.org/en/v1/manual/performance-tips/

And, of course, the Julia documentation:

https://docs.julialang.org/en/v1.7/

Exercise 1: Plots and Functions

Consider the following function of x in the interval $[-2\pi, 2\pi]$:

$$f(x) = e^{-\frac{x^2}{5}} \sin(2x).$$

Create a script `exercise1.m` which solves the following tasks:

1. Plot the function for 200 linearly spaced points in the given interval using a for loop.
2. Define a function handle for $f(x)$ and plot the function for 1000 linearly spaced points in the given interval **without** using any for loop.
3. Add title and axis labels to the two plots and save them in SVG format.

Exercise 2: Taylor Series

The value of $\cos(x)$ can be approximated by using the following Taylor series:

$$\cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}.$$

Of course, for the purpose of implementation, we are interested in a truncated version of this series and we will indicate with N the number of terms considered.

Solve the following tasks:

1. Write a function `approxCos()` which approximates $\cos(x)$ according to the formula above. The function takes as inputs the number of points N and the point x in which we want to compute the approximation, and returns the approximated value of the function. Save this function in a folder called '`functions/`'.
2. Write a script `exercise_2.jl` in which you call `approxCos()` to compute the approximate value of $\cos(x)$ for $x = \pi$ and $x = 2\pi$. Compute the L1-norm of the difference between the approximation and the exact solution in both cases and plot it in logarithmic scale (use the plot attribute `yaxis=:log`) for $N = 1, \ldots, 20$ in the same figure.