Appendix B

# Brief introduction to Julia

In this chapter, we very briefly present some of the basic features and functions of Julia. Most of the information contained in this chapter can be found in the online manual, to which we provide pointers in each section.

## Installing Julia

The suggested programming environment for this course is the open-source text editor Visual Studio Code. You may also use *Vim* or *Emacs*, if you are familiar with any of these.

💻 **Task 1.** *Install Visual Studio Code. Install also the Julia and Jupyter Notebook extensions.*

## Obtaining documentation

To find documentation on a function from the Julia console, type "?" to access "help mode", and then the name of the function. Tab completion is helpful for listing available function names.

💻 **Task 2.** *Read the help pages for* **`if`**, **`while`** *and* **`for`**. *More information on these keywords is available in the online documentation.*

*Remark* B.1 (Shorthand **`if`** notation). If there is no **`elseif`** clause, it is sometimes convenient to use the following shorthand notations instead of an **`if`** block.

```
condition = true

# Assign x = 0 if `condition` is true, else assign x = 2
x = condition ? 0 : 2

# Print "true" if `condition` is true
condition && println("true")
```

```julia
# Print "false" if `condition` is false
condition || println("false")
```

### Installing and using a package [link to relevant manual section]

To install a package from the Julia REPL (Read Evaluate Print Loop, also more simply called the Julia console), first type "`]`" to enter the package REPL, and then type `add` followed by the name of the package to install. After it has been added, a package can be used with the `import` keyword. A function `fun` defined in a package `pack` can be accessed as `pack.fun`. For example, to plot the cosine function from the Julia console or in a script, write

```julia
import Plots
Plots.plot(cos)
```

Alternatively, a package may be imported with the `using` keyword, and then functions can be accessed without specifying the package name. While convenient, this approach is less descriptive; it does not explicitly show what package a function comes from. For this reason, it is often recommended to use `import`, especially in a large codebase.

💻 **Task 3.** *Install the* `Plots` *package, read the documentation of the* `Plots.plot` *function, and plot the function* $f(x) = \exp(x)$*. The tutorial on plotting available at this link may be useful for this exercise.*

*Remark* B.2. We have seen that `?` and `]` enable to access "help mode" and "package mode", respectively. Another mode which is occasionally useful is "shell mode", which is accessed with the character `;` and allows to type `bash` commands, such as `cd` to change directory. See this part of the manual for additional documentation on Julia modes.

### Printing output

The functions `println` and `print` enable to display output. The former adds a new line at the end and the latter does not. The symbol `$`, followed by a variable name or an expression within brackets, can be employed to perform *string interpolation*. For instance, the following code prints `a = 2, a^2 = 4`.

```julia
a = 2
println("a = $a, a^2 = $(a*a)")
```

To print a matrix in an easily readable format, the `display` function is very useful.

### Defining functions [link to relevant manual section]

Functions can be defined using a `function` block. For example, the following code block defines a function that prints "Hello, NAME!", where NAME is the string passed as argument.

```julia
function hello(name)
    # Here * is the string concatenation operator
    println("Hello, " * name)
end

# Call the function
hello("Bob")
```

If the function definition is short, it is convenient to use the following more compact syntax:

```
hello(name) = println("Hello, " * name)
```

Sometimes, it is useful to define a function without giving it a name, called an *anonymous function*. This can be achieved in Julia using the arrow notation `->`. For example, the following expressions calculate the squares and cubes of the first 5 natural numbers. Here, the function `map` enables to transform the collection passed as second argument by applying the function passed as first argument to each element.

```
squares = map(x -> x^2, [1, 2, 3, 4, 5])
cubes = map(x -> x^3, [1, 2, 3, 4, 5])
```

The `return` keyword can be used for returning a value to the function caller. Several values, separated by commas, can be returned at once. For instance, the following function takes a number $x$ and returns a tuple $(x, x^2, x^3)$.

```
function powers(x)
    return x, x^2, x^3
end

# This is an equivalent definition in short notation
short_powers(x) = x, x^2, x^3

# This assigns a = 2, b = 4, c = 8
a, b, c = powers(2)
```

Like many other languages, including Python and Scheme, Julia follows a convention for argument-passing called “pass-by-sharing”: values passed as arguments to a function are not copied, and the arguments act as new bindings within the function body. It is possible, therefore, to modify a value passed as argument, provided this value is of mutable type. Functions that modify some of their arguments usually end with an exclamation mark `!`. For example, the following code prints first `[4, 3, 2, 1]`, because the function `sort` does not modify its argument, and then it prints `[1, 2, 3, 4]`, because the function `sort!` does.

```
x = [4, 3, 2, 1]
y = sort(x)  # y is sorted
println(x); sort!(x); println(x)
```

Similarly, when displaying several curves in a figure, we first start with the function `plot`, and then we use `plot!` to modify the existing figure.

```julia
import Plots
Plots.plot(cos)
Plots.plot!(sin)
```

As a final example to illustrate argument-passing, consider the following code. Here two arguments are passed to the function `test`: an array, which is a mutable value, and an integer, which is immutable. The instruction `arg1[1] = 0` modifies the array to which both `a` and `arg1` are bindings. The instruction `arg2 = 2`, on the other hand, just causes the variable `arg2` to point to a new immutable value (3), but it does not change the destination of the binding `b`, which remains the immutable value 2. Therefore, the code prints `[0, 2, 3]` and `3`.

```julia
function test(arg1, arg2)
    arg1[1] = 0
    arg2 = 2
end
a = [1, 2, 3]
b = 3
test(a, b)
println(a, b)
```

💻 **Task 4** (Euler–Mascheroni constant for the harmonic series)**.** *Euler showed that*

$$\lim_{N\to\infty}\left(-\ln(N)+\sum_{n=1}^{N}\frac{1}{n}\right)=\gamma:=0.577...$$

*Write a function that returns an approximation of the Euler–Mascheroni constant $\gamma$ by evaluating the expression between brackets at a finite value of $N$.*

```julia
function euler_constant(N)
    # Your code comes here
end
```

💻 **Task 5** (Ancient algorithms)**.** *The goal of this exercise is to explore three of the oldest algorithms ever invented.*

- *Circa 1600 BC, the Babylonians invented an iterative method for calculating the square root of a number. Read the relevant information on the associated Wikipedia page and write a function that calculates the square root of the argument using this algorithm.*

  ```julia
  function babylonian_square_root(n)
      # Your code comes here
  end
  # The function should return the square root of n
  ```

- *Circa 300 BC, the Greek mathematician Euclid of Alexandria published the* Elements*, his famous mathematical treatise. In one of the books, he proposes an algorithm for calculating* *the greatest common divisor of two numbers. This algorithm, which is still in common use today, is based on the observation that if $a > b \geqslant 0$ are natural numbers, then*

$$\gcd(a, b) = \gcd(b, r), \tag{B.1}$$

*where $r$ is the remainder of the division of $a$ by $b$. Indeed, in view of the equation*

$$a = qb + r,$$

*the common divisors of $\{a, b\}$ coincide with those of $\{b, r\}$. Using (B.1), write a function to calculate the greatest common divisor of two numbers.*

```
function euclid_gcd(a, b)
    # Your code comes here
end
```

- *Circa 200 BC, the Greek mathematician Eratosthenes of Cyrene invented a method for efficiently calculating the prime numbers, which is now known as the* sieve of Eratosthenes. *Read the associated Wikipedia page and write a function implementing this algorithm.*

```
function eratosthenes_sieve(n)
    # Your code comes here
end
# The function should return an array containing all the prime
# numbers less than or equal to n.
```

**Task 6** (Tower of Hanoi). *We consider a variation on the classic Tower of Hanoi problem, in which the number $r$ of pegs is allowed to be larger than 3. We denote the pegs by $p_1, \ldots, p_r$, and assume that the problem includes $n$ disks with radii 1 to $n$. The tower is initially constructed in $p_1$, with the disks arranged in order of decreasing radius, the largest at the bottom. The goal of the problem is to reconstruct the tower at $p_r$ by moving the disks one at the time, with the constraint that a disk may be placed on top of another only if its radius is smaller.*

*It has been conjectured that the optimal solution, which requires the minimum number of moves, can always be decomposed into the following three steps, for some $k \in \{1, n-1\}$:*

- *First move the top $k$ disks of the tower to peg $p_2$;*
- *Then move the bottom $n-k$ disks of the tower to $p_r$ without using $p_2$;*
- *Finally, move the top of the tower from $p_2$ to $p_r$.*

*This suggests a recursive procedure for solving the problem, known as the Frame-Stewart algorithm. Write a Julia function* `T(n, r)` *returning the minimal number of moves necessary.*

## Local and global scopes [link to relevant manual section]

Some constructs in Julia introduce scope blocks, notably `for` and `while` loops, as well as `function` blocks. The variables defined within these structures are not available outside them. For example

```
if true
    a = 1
end
println(a)
```

prints `1`, because **`if`** does not introduce a scope block, but

```
for i in [1, 2, 3]
    a = 1
end
println(a)
```

produces `ERROR: LoadError: UndefVarError: a not defined`. The variable `a` defined within the **`for`** loop is said to be in the *local scope* of the loop, whereas a variable defined outside of it is in the *global scope*. In order to modify a global variable from a local scope, the **`global`** keyword must be used. For instance, the following code

```
a = 1
for i in [1, 2, 3]
    global a += 1
end
println(a)
```

modifies the global variable `a` and prints 4.

## Multi-dimensional arrays [link to relevant manual section]

A working knowledge of multi-dimensional arrays is important for this course, because vectors and matrices are ubiquitous in numerical algorithms. In Julia, a two-dimensional array can be created by writing its lines one by one, separating them with a semicolon `;`. Within a line, elements are separated by a space. For example, the instruction

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

creates the matrix

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

More generally, the semicolon enables vertical concatenation while space concatenates horizontally. For example, `[M M]` defines the matrix

$$\begin{pmatrix} 1 & 2 & 3 & 1 & 2 & 3 \\ 4 & 5 & 6 & 4 & 5 & 6 \end{pmatrix}$$

The expression `M[r, c]` gives the $(r, c)$ matrix element of $M$, located at row $r$ and column $c$. The special entry **`end`** can be used to access the last row or column. For instance, `M[end-1, end]` gives the matrix entry in the second to last row and the last column. From the matrix $M$ above, the submatrix `[2 3; 5 6]` can be obtained with `M[:, 2:3]`. Here the row index `:` means "select all lines" and the column index `2:3` means "select columns 2 to 3". Likewise, the submatrix `[1 3; 4 6]` may be extracted with `M[:, [1; 3]]`.

*Remark* B.3 (One-dimensional arrays). The comma `,` can also be employed for creating one-dimensional arrays, but its behavior differs slightly from that of the vertical concatenation operator `;`. For example, `x = [1, [2; 3]]` creates a `Vector` object with two elements, the first one being `1` and the second one being `[1; 3]`, which is itself a `Vector`. In contrast, the instruction `x = [1; [1; 2]]` creates the same `Vector` as `[1; 2; 3]` would.

We also mention that the expression `x = [1 2 3]` produces not a one-dimensional `Vector` but a two-dimensional `Matrix`, with one row and three columns. This can be checked using the `size` function, which for `x = [1 2 3]` returns the tuple `(1, 3)`.

There are many built-in functions for quickly creating commonly used arrays. For example,

- `transpose(M)` gives the transpose of $M$, and `adjoint(M)` or `M'` gives the transpose conjugate. For a matrix with real-valued entries, both functions deliver the same result.
- `zeros(Int, 4, 5)` creates a $4 \times 5$ matrix of zeros of type `Int`;
- `ones(2, 2)` creates a $2 \times 2$ matrix of ones of type `Float64`;
- `range(0, 1, length=101)`, or `LinRange(0, 1, 101)`, creates an array of size 101 with elements evenly spaced between 0 and 1 included. More precisely, `range` returns an array-like object, which can be converted to a vector using the `collect` function.
- `collect(reshape(1:9, 3, 3))` creates a $3 \times 3$ matrix with elements

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

Let us also mention the following shorthand notation, called *array comprehension*, for creating vectors and matrices:

- `[i^2 for i in 1:5]` creates the vector `[1, 4, 9, 16, 25]`.
- `[i + 10*j for i in 1:4, j in 1:4]` creates the matrix

$$\begin{pmatrix} 11 & 21 & 31 & 41 \\ 12 & 22 & 32 & 42 \\ 13 & 23 & 33 & 43 \\ 14 & 24 & 34 & 44 \end{pmatrix}.$$

- `[i for i in 1:10 if ispow2(i)]` creates the vector `[1, 2, 4, 8]`. The same result can be achieved with the `filter` function: `filter(ispow2, 1:10)`.

In contrast with Matlab, array assignment in Julia does not perform a copy. For example the following code prints `[1, 2, 3, 4]`, because the instruction `b = a` defines a new binding to the array `a`.

```julia
a = [2; 2; 3]
b = a
b[1] = 1
append!(b, 4)
println(a)
```

A similar behavior applies when passing an array as argument to a function, as we saw previously. The `copy` function can be used to perform a copy.

💻 **Task 7.** *Create a 10 by 10 diagonal matrix with the $i$-th entry on the diagonal equal to $i$.*

### Broadcasting

To conclude this chapter, we briefly discuss *broadcasting*, which enables to apply functions to array elements and to perform operations on arrays of different sizes. Julia really shines in this area, with syntax that is both explicit and concise. Rather than providing a detailed definition of broadcasting, which is available in this part of the official documentation, we illustrate the concept using examples. Consider first the following code block:

```julia
function welcome(name)
    return "Hello, " * name * "!"
end
result = broadcast(welcome, ["Alice", "Bob"])
```

Here `broadcast` returns an array with elements `"Hello, Alice!"` and `"Hello, Bob!"`, as would the `map` function. Broadcasting, however, is much more flexible because it can handle arrays with different sizes. For instance, `broadcast(gcd, 24, [10, 20, 30])` returns an array of size 3 containing the greatest common divisors of the pairs $(24, 10)$, $(24, 20)$ and $(24, 30)$. Similarly, the instruction `broadcast(+, 1, [1, 2, 3])` returns `[2, 3, 4]`. To understand the latter example, note that `+` (as well as `*`, `-` and `/`) can be called like any other Julia functions; the notation `a + b` is just syntactic sugar for `+(a, b)`.

Since broadcasting is so often useful in numerical mathematics, Julia provides a shorthand notation for it: the instruction `broadcast(welcome, ["Alice", "Bob"])` can be written compactly as `welcome.(["Alice", "Bob"])`. Likewise, the line `broadcast(+, 1, [1, 2, 3])` can be shortened to `(+).(1, [1, 2, 3])`, or to the more readable expression `1 .+ [1, 2, 3]`.

💻 **Task 8.** *Explain in words what the following instructions do.*

```julia
reshape(1:9, 3, 3) .* [1 2 3]
reshape(1:9, 3, 3) .* [1; 2; 3]
reshape(1:9, 3, 3) * [1; 2; 3]
```