>>> Introduction to Programming with Julia

Name: Georgios Vretinaris

Numerical Analysis Group
Eberhard Karls University of Tübingen
Date: February 23-27, 2025

>>> Schedule

* Monday to Friday, 9am-12pm and 1:30pm-4pm
    * Theoretical explanations
    * Live programming of small programs
    * Independent work on exercise sheet tasks
    * Open format with room for questions

>>> Outline

>>> Why Numerical Analysis?

* Many analytical problems cannot be solved explicitly or
* explicit representation of the solution is not suitable for fast computation.

Examples:

$$\int_{-1}^{1} e^{-x^2} dx = ? \qquad \frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \pm \dots$$

>>> Why Numerical Analysis?

Numerical Mathematics / Numerical Analysis:

* Construction of algorithms for (approximate) computation of solutions to continuous problems
* Analysis of the accuracy of the computed solution (error analysis)

>>> Numerical Analysis and Programming?

* **Algorithm:** Unambiguous set of instructions for solving a problem; consists of finitely many well-defined individual steps. Example:

  Input: A number x
  Instruction 1: Add 3 to the number
  Instruction 2: Multiply the result by 2
  Output: ?

* Mathematically formulated algorithms are translated into programming language.

>>> What Does Programming Mean?

* Translating colloquial instructions into computer language
* Creating computer programs

Julia ...

* ... is a programming language in which algorithms are implemented
* ... can be used for various different things but has a scientific/mathematical and HPC focus
* ... provides a vast number of packages with pre-built algorithms

>>> **Installing Julia**

You can go to Juliaup's github page: https://github.com/JuliaLang/juliaup

Windows

On Windows Julia and Juliaup can be installed directly from the Windows store here. One can also install exactly the same version by executing

```
winget install --name Julia --id 9NJNWW8PVKMN -e -s msstore
```

on a command line.

If the Windows Store is blocked on a system, we have an alternative MSIX App Installer based setup. Note that this is currently experimental, please report back successes and failures here. To use the App Installer version, download this file and open it by double clicking on it.

>>> Installing Julia

You can go to Juliaup's github page: https://github.com/JuliaLang/juliaup

Mac, Linux, and FreeBSD

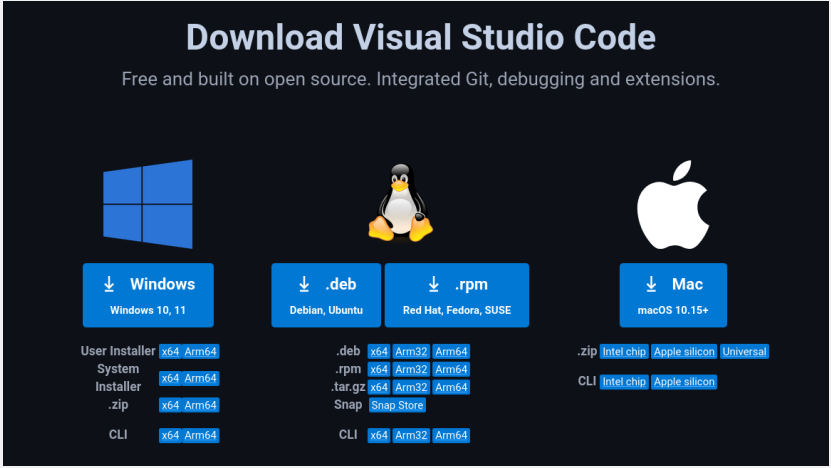
Juliaup can be installed on Unix-like platforms (currently Linux, Mac, or FreeBSD) by executing

```
curl -fsSL https://install.julialang.org | sh
```

in a shell.

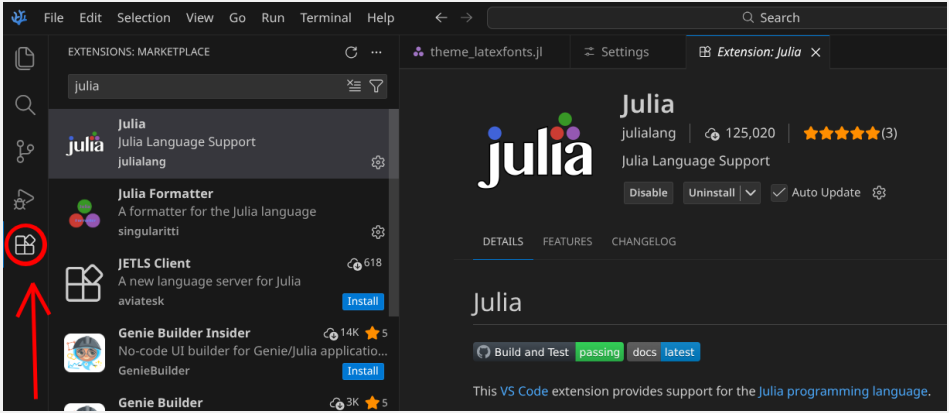
>>> (Optional) Install VSCode + Julia Extension

https://code.visualstudio.com/download



>>> (Optional) Install VSCode + Julia Extension

Click on the Extensions tab on the left side bar
(or Ctrl/Cmd+Shift+X) and search for Julia



There are multiple equivalent ways to use Julia.

* Write a document and call Julia to execute it.

>>> Using Julia

There are multiple equivalent ways to use Julia.

* Write a document and call Julia to execute it.

```
 ~/Dow/w/I/slides/Introduction to Programming/scripts julia hello_world.jl
The latest version of Julia in the `release` channel is 1.11.5+0.x64.linux.gnu. You currently have `1.11.3+0.x64.linux.gnu` installed. Run:

  juliaup update

in your terminal shell to install Julia 1.11.5+0.x64.linux.gnu and update the `release` channel to that version.
Hello World
 ~/Dow/w/I/slides/Introduction to Programming/scripts
```

>>> Using Julia

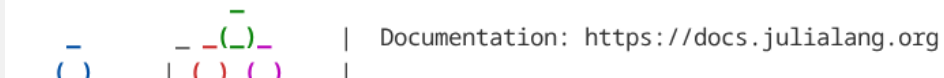
There are multiple equivalent ways to use Julia.

* Write a document and call Julia to execute it.
* or use the REPL (Read-Eval-Print-Loop) interface

>>> Using Julia

There are multiple equivalent ways to use Julia.

* Write a document and call Julia to execute it.
* or use the REPL (Read-Eval-Print-Loop) interface




```
julia> include("hello_world.jl")
Hello World
```

>>> Using Julia

There are multiple equivalent ways to use Julia.

* Write a document and call Julia to execute it.
* or use the REPL (Read-Eval-Print-Loop) interface
* or use Jupyter lab/notebooks

>>> Using Julia

There are multiple equivalent ways to use Julia.

* Write a document and call Julia to execute it.
* or use the REPL (Read-Eval-Print-Loop) interface
* or use Jupyter lab/notebooks

Jupyter allows one to write and style text (and images) in between code.

It uses Markdown, which is a language to format text, and supports $\LaTeX$ and HTML.

[2]:
```
i = 1
while true
    i += 1
    if i >= 10
        break
    end
end
println(i)
```

```
10
```

>>> **Do it yourself!**

Exercise 0

* Start the Julia REPL.
    * For VSCode, either click on View>Terminal or press Ctrl/Cmd + `.
    * Then write julia and press Enter.
* Enter the command display("Hello world") in the REPL. What happens?

>>> Outline

>>> Syntax

* **Syntax:** Rule system for composing elementary characters.
* Computers cannot understand natural language.
* Unlike (most) humans, computers cannot understand sentences with incorrect structure.
* Syntax defines rules about permissible language elements of the programming language.

>>> Syntax

* For a programming command to be executed, the correct syntax must be used.
* Example multiplication:
  a = 3
  b = 2
  c = ab ⇐ Not executable!
  Multiplication sign * must be used.
* Example sine function:
  x = 3.1415
  y = sin x ⇐ Not executable. Parentheses must be used.
  y = sin(x) ⇐ Executable.

>>> Syntax

* When executing a sequence of commands, often only the final result is desired.
* Each command is typically terminated with a semicolon to suppress the output of the result in the REPL .
* Comments can be added to your program using the hashtag sign:
  `p = 3.14; # p is an approximation for pi`
  ⇒ Everything from the hashtag sign to the end of the line is ignored by Julia during program execution.

If one writes a script or a function, only the last assignment/operation will be shown.

>>> **Do it yourself!**

Exercise 1

(a) Start the Julia REPL.

(b) Declare some variables, e.g., `n=3` and `m=5`.

(c) Enter the following commands: `n+m`, `n+m;`, and `k=n+m`. Understand the differences by noting which values are returned and which are stored.

>>> Syntax

* Predefined constants, e.g., **pi**, **eps()** (machine epsilon), **im** (imaginary unit). Not: **e** (use **exp(1)**).
* Predefined functions, e.g., **sin**, **exp**, etc.
* Strings (character sequences) are enclosed in double quotes:
  `string = "Hello."`
* Code can be spread across multiple lines with some care, parentheses are recommended yet not necessary.

```
sum = (1 + 2 + 3 + 4 + 5 +
+ 6 + 7 + 8 + 9 + 10);
```

>>> Variables

* Variables and operators (+, -, *, /, %...) are the most important building blocks in programming.
* Variables are declared through expressions like a = 3 + 5, v = [1 2 3], etc.
* The equals sign should be understood like := in mathematical texts: The expression on the right is evaluated (if possible) and then stored as a variable with the name on the left.
  ⇒ n = n+1 increases the value of the variable n by 1.
* The equal sign is known as the assignment operator.

>>> Rules for Variable Names

* Variable names may consist of lowercase and uppercase letters, numbers, underscores and some special characters and should be one continuous sequences of characters without a break (i.e. a space).
* Variable names **cannot** start with a number or have symbols with predefined operations (e.g. +).
* Variable names **cannot** have a dot (.) in them as this has a special meaning in Julia (later).
* Almost all programming languages distinguish between uppercase and lowercase letters!
* Good programming style involves using largely self-explanatory, intuitive, and as short as possible variable names.
* Try to follow conventions you know from mathematics: e.g., $n, m, \ldots$ for natural numbers, v, w for vectors, and A for matrices, etc.

>>> **Do it yourself!**

Exercise 2

(a) Which of the following variable names are permissible?

- **(a)** SumX
- **(b)** Variable 1
- **(c)** 12345
- **(d)** Numerics12_34
- **(e)** ∇_x_WS_25_26
- **(f)** 😄

(b) Invent some permissible (and fancy) variable names and test them in Julia.

>>> **Do it yourself!**

Exercise 2

(a) Which of the following variable names are permissible?

- **(a)** SumX
- **(b)** Variable 1
- **(c)** 12345
- **(d)** Numerics12_34
- **(e)** ∇_x_WS_25_26
- **(f)** 😄

(b) Invent some permissible (and fancy) variable names and test them in Julia.

>>> **Do it yourself!**

Exercise 2

(a) Which of the following variable names are permissible?

- **(a)** SumX
- **(b)** Variable 1
- **(c)** 12345
- **(d)** Numerics12_34
- **(e)** ∇_x_WS_25_26
- **(f)** 😄

(b) Invent some permissible (and fancy) variable names and test them in Julia.

>>> **Do it yourself!**

Exercise 2

(a) Which of the following variable names are permissible?

- **(a)** SumX
- **(b)** Variable 1
- **(c)** 12345
- **(d)** Numerics12_34
- **(e)** ∇_x_WS_25_26
- **(f)** 😄

(b) Invent some permissible (and fancy) variable names and test them in Julia.

>>> **Do it yourself!**

Exercise 2

(a) Which of the following variable names are permissible?

- **(a)** SumX
- **(b)** Variable 1
- **(c)** 12345
- **(d)** Numerics12_34
- **(e)** ∇_x_WS_25_26
- **(f)** 😄

(b) Invent some permissible (and fancy) variable names and test them in Julia.

>>> **Do it yourself!**

Exercise 2

(a) Which of the following variable names are permissible?

- **(a)** SumX
- **(b)** Variable 1
- **(c)** 12345
- **(d)** Numerics12_34
- **(e)** ∇_x_WS_25_26
- **(f)** 😄

(b) Invent some permissible (and fancy) variable names and test them in Julia.

>>> **Do it yourself!**

Exercise 2

(a) Which of the following variable names are permissible?

- **(a)** SumX
- **(b)** Variable 1
- **(c)** 12345
- **(d)** Numerics12_34
- **(e)** ∇_x_WS_25_26
- **(f)** 😄 <- This is actually fine and can be used with by typing \:smile:

(b) Invent some permissible (and fancy) variable names and test them in Julia.

Take a look at https://github.com/JuliaWTF/WatchJuliaBurn.jl

>>> Scripts

* Simple Julia programs consist of scripts and functions.
* A function bundles a series of commands.
* A script bundles a series of functions and commands.
* A function is part of a script, and should do one thing (e.g. add argument a with b).
* A script uses the collection of functions to deal with a larger problem.
* When executing the script or a function, the commands entered there are executed in order.

>>> Scripts

* Julia can access any file if provided the exact path to it,
  e.g. "/home/User/Documents/directory/" - Linux
  or "C:\\Users\User\Documents\directory" - Windows.
* Tip: Create a separate folder for each programming problem, each project.

>>> **Do it yourself!**

Exercise 3

(a) Open VSCode, click on the topmost icon in the left bar and then click on Open Folder.

(b) Navigate to whichever place you feel suits you the best, and create a new folder called Intro2Julia, and use that one.

(c) If you hover your mouse on the name of the folder (top-left bar with all the files), and you then hover it over the icons that will appear, the first one from the left will create a new file.

(d) Call it new.jl, and then click on it, it will be empty.

(e) Declare three variables a, b, and c with values of your choice, then enter sum=a+b+c and prod=a*b*c.

(f) Press Ctrl + ` to open the terminal, then type "julia new.jl" and press Enter.

(g) Change the values of a, b, and c and execute the script again.

>>> Relational Operators

* Besides the assignment operator = and the arithmetic operators + - * / ^, there are other operators.
* The relational operators are == != > >= < <=.

```
n = (3 != 4);
display(n); # Output: true
```

* These are typically used to query certain conditions.
* Here too: The operation to the right of the assignment operator, here the query whether $3 \neq 4$ holds, is performed first. The result true is stored in n.

>>> Logical Operators

* Besides relational operators, there are also logical operators: && || ~ stand for and, or, and not, respectively.
* Logical operators link logical variables (i.e., true and false). || is an inclusive or.
  Example:

```
x = 5;
y = (x < 6) || (x > 4);
display(y); # Output true
z = (x > 0) && ~(x > 3);
display(z); # Output false
```

>>> **Do it yourself!**

Exercise 4

1. Define a variable z that queries whether a number x is greater than 5 and less than 7. Do this for several values of x. Display the result in the REPL.
2. Try some additional logical queries in your script.

>>> Outline

>>> if statements

* In many programs, there are individual steps that are only performed under certain conditions.
* For this, if statements paired with logical expressions are used. The structure is always as follows:

```
if condition_1
    Instruction(s)
else if condition_2
    Instruction(s)
⋮
else
Instruction(s)
end
```

>>> if statements

* The last else must not have a logical expression.
* The lines with if, elseif, else, and end are **not** terminated with a semicolon!
* elseif and else are not required:

```
if x != 0
  y = 1/x;
end
```

  ⇒ Since no alternative with else is given, these lines are simply skipped if $x = 0$.
* if-else structures can be nested arbitrarily. One indents the instructions between if and else by four spaces. Maintain this indentation! It makes the code more readable.

>>> **Do it yourself!**

Exercise 5

Write a script in which you declare two numbers x and y. The program should calculate $z = \frac{x}{y}$ if $y \neq 0$ and display an error message if $y = 0$.

Exercise 6

Write a script in which you declare two numbers x and y. The program should output whether $x > y$, $x = y$, or $x < y$.

>>> Practice Exercises!

1. Work on exercises 1 to 8 on the exercise sheet. Ask questions if needed—I'll come around!

>>> Outline

>>> Loops

* For repeated, iterative execution of certain operations, e.g., for:
  * Partial sums
  * Integration with Riemann sums
  * Gaussian elimination
* Loops, along with if queries, are the most important control structures in programming.

>>> for Loops

* for loops have the structure

```
for variable in Iterable (e.g. start:step:end)
    Instruction(s)
end
```

* Example: Calculation of $S = \sum_{k=0}^{n} \frac{(-1)^k}{2k+1}$ for given n, e.g., $n = 5$:

```
1: let[1]
2:     n = 5;
3:     S = 0; # Variable for the sum
4:     for k in 0:n
5:         S = S + (-1)^k / (2*k+1);
6:     end
7: end
```

[1] This is to avoid anything be defined in the global memory scope.

>>> Explanations

* In line 2, the index n up to which the partial sum is to be calculated is first declared.
* In line 3, a variable S is declared that should contain the value of the sum at the end. This is necessary because the individual summands in line 5 must be added to an already existing variable.
  (For products, one would initialize with the value 2.)

Explanation:

* In line 4, a variable k=0 is first defined.
* All instructions between this line and line 5 (end) are executed one after the other, with the variable k starting with the value 0.
* Then k is increased by 1, so it now has the value 1.
* All instructions are executed one after the other, this time with k=1
* Then k is increased by 1 and the instructions are executed again
* …until k=6, where the instructions of the for loop are executed one last time.

* Sometimes you don't want the loop variable to be increased by 1 in each step.
* Example: Calculation of $7! = 7 \cdot 6 \cdots 2 \cdot 1$

```
let
  n = 7;
  x = n;
  for i ∈ (n-1):-1:1
    x = x*i;
  end
end
```

* Explanation: i initially has the value n-1 = 6 and is increased by -1 in each loop iteration until the loop is executed for the last time with i = 1. The step doesn't need to be positive, but if the starting value plus the step is larger than the stopping value then the loop isn't executed.

>>> **Do it yourself!**

Exercise 7

(a) Write a script in which you define two integers n and N. The program should output all integers from n up to and including N.
(b) What happens if you choose n greater than N?
(c) Improve your program so that it outputs an error message and the loop is not called if $n > N$.

Exercise 8

Write a script that calculates the double factorial n!! for a given number n.

* Sometimes it's not clear beforehand, how many repetitions are needed.
* Example: It can be shown that the recursively defined sequence

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{2}{x_k}\right), \quad x_1 = 2$$

  converges to $\sqrt{2}$ (Heron's method).
* To implement this method, one computes this sequence successively until two consecutive terms no longer differ significantly, i.e., until $|x_{k+1} - x_k| < \texttt{tol}$ with a given tolerance tol.
  $\Longrightarrow$ It is not clear how many iterations are required.

>>> while Loops

* A while loop has the structure
  ```
  while condition
      Instruction(s)
  end
  ```
* Before each loop iteration, it is checked whether logical expression returns true. If yes, the instructions between while and end are executed and logical expression is checked again.
* Warning: It is possible for logical expression to always return true. In this case, the loop does not terminate. This is called an **infinite loop**.

This is also possible with any type of loop and is a programming error.

Exercise 9

Write a program that calculates the recursive sequence $x_{k+1} = \frac{1}{2}(x_k + \frac{2}{x_k})$ with starting value $x_1 = 2$ until the difference between two consecutive values is smaller than a given tolerance tol. Use tolerances smaller than 10^{-2}.
In a while loop, a xNew should be calculated from xOld according to the above formula.

>>> break and continue in Loops

* continue: The current loop iteration is interrupted, and the program jumps back to the start of the loop.
* break: The entire loop is interrupted, and execution continues after the loop.

```
for i=1:6
    if i == 2
        continue
    end
    if i == 4
        break
    end
    display(i)
end
```

Which numbers does the program output?

Practice Exercises

1. Work on exercises 9 to 12 on the exercise sheet. Ask questions if needed—I'll come around!

>>> Outline

>>> Functions

* Function are a collection of commands to compute something.
* Functions typically receive input parameters (variables) and return output parameters (function values) as results.
* Functions create their own local (memory) scope. This means:
  * When a function is called, it does not know the variables already declared elsewhere and therefore cannot modify them (almost).
  * If new variables are defined in a function, they are not accessible after the function is evaluated, if not returned.

>>> Example Function

The following function calculates the area and perimeter of a rectangle:

```
"""
Calculate area and perimeter
of a rectangle with side lengths a,b.
"""
function rectangle_area(a,b)
   area = a*b;
   perim = 2*a+2*b;
   return area, perim
end
rect_area, rect_perim = rectangle_area(1,2)
```

Here, **a** and **b** are the input parameters, and **area** and **perim** are the output parameters, which when called, can be stored in any other variable named with a different way like **rect_area** and **rect_perim**.

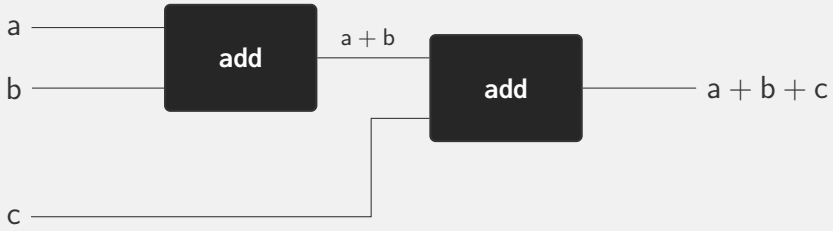
>>> Important Notes About Functions

* If a function requires no input values, the parentheses after the function name are empty: ().
* It is a good programming habit to have your functions do one thing and not a hundred.
* Think of functions as black boxes that abstract some operations, keep them simple and then use them together for further abstractions.

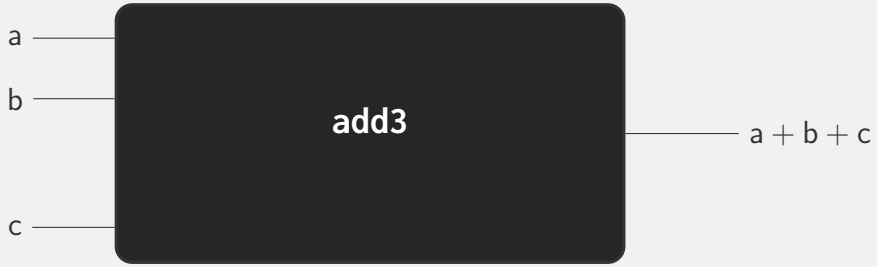
>>> Function Abstraction

From a talk of Gabriele Bozzola https://tinyurl.com/abstractionPdf

>>> Function Abstraction



From a talk of Gabriele Bozzola https://tinyurl.com/abstractionPdf



From a talk of Gabriele Bozzola https://tinyurl.com/abstractionPdf

Exercise 10

Write a function circ_geom(r), which returns the area and the perimeter of a circle with radius r.

Then call the function (in the REPL) in different ways.

* circ_area,circ_perim = circ_geom(r)
* x = circ_geom(r)

What value does x get? Understand the different meanings of the calls.

>>> Important Functions in Julia

* Trigonometric functions: **sin, cos, tan**
* Exponential function and natural logarithm: **exp, log**
* Square root function **sqrt**
* Absolute value **abs**
* Property querying **length, size**
* Loading, saving **load, save** (not covered)
* Graphical functions **plot** (later)
* More powerful functions **lu, qr, norm**…(not covered)

>>> Scripts - Summary

- A script bundles a series of functions and commands.
- Executing a script is equivalent to entering each line written in it one after the other in the REPL. This means in particular:
  - It has access to the global (memory) scope, i.e. it knows the variables that were declared before it was executed.
  - If existing variables are modified or new ones are declared, these changes are present after the script is executed.

>>> Outline

>>> What Is a Type?

A **type** describes what kind of value something is and what you can do with it.

Value	Type	Legal operations
42	Integer	+ − × ÷
3.14	Float	+ − × ÷
"hello"	String	concatenate, slice, search
true	Boolean	and, or, not
[1,2,3]	List	index, append, sort

Types are a **contract**:

* An integer can be added to another integer. ✓
* A string can be searched or sliced. ✓
* Dividing a string by a number? **Nonsense.** Types catch that.

>>> Static vs. Dynamic Typing

Static Typing (C)

Types checked **before** the program runs.

```c
int x = 5;
char *name = "Alice";
int bad = x + name;
// COMPILE ERROR — caught early
```

++ Bugs caught early

-- Less flexible

Dynamic Typing (Julia)

Types checked **while** the program runs.

```julia
x = 5        # Int64
x = "hello"  # now a String!
result = "hello" + 3
# MethodError — only at RUNTIME
```

++ Flexible, quick to write

-- Bugs hide until execution

>>> Type System

* Julia is a dynamically typed language, this means that you do not have to specify the type of variable you declare explicitly, but Julia will take care of it and infer its type.
* Which is good if you want ease of coding but makes things slow because Julia still has to run something to infer the type.
* E.g. `x = 1; typeof(x)` (you can try it yourselves), returns `Int64` but `x=1.0` would return `Float64`.

>>> Exercise

Type Tree for Numbers

```
using AbstractTrees
AbstractTrees.children(d::DataType) = subtypes(d)
print_tree(Number)
```

>>> Exercise

Type Tree for Arrays

```
using AbstractTrees
AbstractTrees.children(d::Type) = subtypes(d)
print_tree(AbstractArray)
```

>>> Types

* Types are abstractions which help programmers to provide certain information to the computer about the variables, so that they can write specialized code for each specific type.
* Julia's type system plays an important role in how one develops their code.
* We can specify the type we want in the following way: **`x::Int8 = 1`**.

>>> Vectors and Matrices

* Scalars (numbers) are subtypes of Number.
* Vectors are subtypes of AbstractArray. Specifically the Vector type is just an alias for Array{eltype,1}.
* Likewise, matrices are also subtypes of AbstractArray and are an alias for Array{eltype,2}.
* Try running just Vector or Matrix to verify.

>>> Vectors: Creation

* Vectors are declared in square brackets [ ].
* Columns are separated by commas, or semicolons.
* a = [1, 2, 3, 4]; creates a vector of with 4 elements.
* c = [1;2;3;4]; is the same as a.
* d = start:step:end; creates a range, which we can access as a vector with collect(d).
* e = []; creates an empty vector.
* f = zeros(10); creates a 10-element vector filled with zeros of type Float64.
* h = zeros(ComplexF64,10); creates a 10-element vector filled with zeros of type ComplexF64.

>>> Vectors: Access

* a = Float64[1, 2, 5, 9, 3, 7,]; creates a vector with Float64 type elements.
* Element-wise access:
  g = a[1]; creates a scalar with the value $a_1 = 1$.
  i = 3; g = a[i]; creates a scalar with the value $a_i = a_3 = 5$.
  g = a[end]; creates a scalar with the last value of a, i.e., $g = 7$, we can do the same with begin.
* Range selection:
  g = a[1:3]; contains the first three values of a.
  v = [1, 2, 4]; g = a[v]; $\Rightarrow$ What is g?
* Unlike other programming languages, Julia starts indexing at 1, not 0.

>>> Vectors: Operations

Warning: Indexing calls a get_index function, and for normal indexing only input parameters of type Int are accepted!

* Transpose: If c is a column vector, then d = c' is a row vector and vice versa.
* By default all vectors are column vectors.
* Determine the length of a vector, e.g., for loops:

```
n = length(a);
for i = 1:n
    display(a[i]);
end
```

Exercise 11

Write a function function add(a,b), which adds two vectors a and b. The addition should be implemented element-wise using a **for** loop. First, determine the lengths of the vectors a and b.

Operators may work differently on vectors than on scalars.

* Assignment operator: =
  a = b: $a[i] = b[i]$ for all i.
* Addition and Subtraction operators: + -
  a + b: $a[i] + b[i]$ for all i.
* Multiplication operator: *
  a * b only works if a is a row vector and b is a column vector (dot product) or vice versa (dyadic product).
  $\Rightarrow$ Corresponds to matrix multiplication.

>>> Broadcasting

* Julia has a specific syntactic sugar, which allows an operation to be applied to all of the elements of a vector (or an iterable in general).
* Writing a dot before any function e.g. `.*` applies the operation in an element-wise manner. For example:

```
a = 2;
b = [1, 2, 3];
c = [4, 5, 6];
a * b # returns [2, 4, 6]
b * c
b .* c
```

>>> Broadcasting

* Julia has a specific syntactic sugar, which allows an operation to be applied to all of the elements of a vector (or an iterable in general).
* Writing a dot before any function e.g. `.*` applies the operation in an element-wise manner. For example:

```
a = 2;
b = [1, 2, 3];
c = [4, 5, 6];
a * b # returns [2, 4, 6]
b * c # return an error
b .* c
```

>>> Broadcasting

* Julia has a specific syntactic sugar, which allows an operation to be applied to all of the elements of a vector (or an iterable in general).
* Writing a dot before any function e.g. `.*` applies the operation in an element-wise manner. For example:

```
a = 2;
b = [1, 2, 3];
c = [4, 5, 6];
a * b # returns [2, 4, 6]
b * c # return an error
b .* c # returns [4, 10, 18]
```

>>> Vectors: Operations

* Relational operators are applied either element-wise: a .== b and get a vector whose entries are true or false, or to the whole vector as an object a == b:

```
a = [1, 2, 3, 4];
b = [1, 9, 3, 4];
c = (a == b);
d = (a .== b);
```

* The same applies to >, >=, etc.

>>> Vectors: Operations

* Relational operators are applied either element-wise: a .== b and get a vector whose entries are true or false, or to the whole vector as an object a == b:

```
a = [1, 2, 3, 4];
b = [1, 9, 3, 4];
c = (a == b); # c = false
d = (a .== b);
```

* The same applies to >, >=, etc.

>>> Vectors: Operations

* Relational operators are applied either element-wise: a .== b and get a vector whose entries are true or false, or to the whole vector as an object a == b:

```
a = [1, 2, 3, 4];
b = [1, 9, 3, 4];
c = (a == b); # c = false
d = (a .== b); # d = [1, 0, 1, 1]
```

* The same applies to >, >=, etc.

We know how to separate columns, for rows one uses just a empty space between the elements.

* `A = [1 2 3 4; 5 6 7 8]; # 2 x 4 - Matrix{Int64}`

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

* `B = [1 5; 2 6; 3 7; 4 8]'; # identical to A (!)`
* `B[2,3] = 15; # changes an element of B`

$$B = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 15 & 8 \end{pmatrix}$$

>>> Matrices: Creation

* `Id = I(n); # n x n - Identity matrix`
* `A = zeros(n,m); # n x m - Matrix with Float64 zeros`
* `B = ones(Int8,n,m); # n x m - Matrix with Int8 ones`
* `v = [1, 2, 3]; D = diagm(v);` creates the matrix

$$\mathsf{D} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

* `M = diagm(1 => v);` creates the matrix

$$\mathsf{D} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Needs the LinearAlgebra package which comes with Julia.

>>> Matrices: Access

* As a whole: `B = A;`.
* Element-wise:
  `b = A[3,1];`
  `i = 2; j = 3; g = A[i,j];`
  `g = A[end,begin];` g is a scalar containing the first element from the last row of A.
* Range selection:
  `B = A[2:4,2:4];` is a (3×3) submatrix

$$B = \begin{pmatrix} a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} \\ a_{42} & a_{43} & a_{44} \end{pmatrix}$$

For vectors v, w with positive integers, $A[v, w]$ returns the matrix with entries a_{v_i, w_j} (advanced).

>>> Matrices: Operations

* Transpose:
  ```
  B = A'; % B is the transpose of A
  ```
* Determine size:
  ```
  n, m = size(A); # n rows, m columns
  ```
  (according to the notation $A \in \mathbb{R}^{n \times m}$)

Exercise 14

Write a function function addMat(A,B), which adds two matrices A and B. First, determine the sizes of A and B, and implement the addition using two nested for loops.

>>> Matrices: Arithmetic Operations

* `A ± B` computes the matrix with entries `A[i,j] ± B[i,j]`.
* `A * B` has three meanings:
  1. If `A` and `B` are matrices, then `A*B` corresponds to standard matrix multiplication (if dimensions match).
  2. If `A` is an $(m \times n)$-matrix and `B` is an $(n \times 1)$-column vector, then `A*B` is a matrix-vector multiplication (falls under (1)).
  3. If `A` is a matrix and `B` is a scalar, then `A * B` is the matrix with entries `A[i,j]*B`.

Practice Exercises

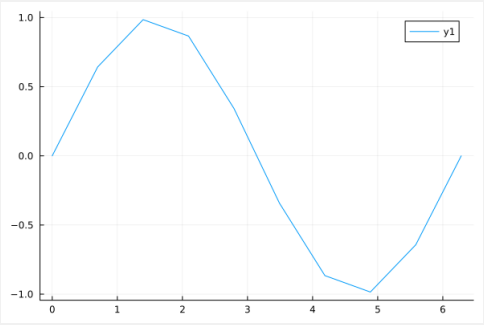
1. Work on exercises 13 to 21 on the exercise sheet. Ask questions if needed—I'll come around!

>>> Outline

>>> 2D Plots

* Julia doesn't come with a plotting functions, so we'll use the simplest package for the course.

```
using Plots
x = LinRange(0,2*pi,10);
y = sin(x);
plot(x,y); # Plots vectors
```



>>> Plotting Procedure

Given a function $f : [a, b] \to \mathbb{R}$ to plot:

1. Create vector x with support points (evaluation points). Example:
   ```
   x = -1:0.1:1;
   % Interval [-1,1], with stepsize 0.1
   x2 = LinRange(-1,1,21); % equivalent to x
   ```
2. Evaluate the function at all support points, e.g., with a loop or broadcasting.
   ```
   y = f.(x)
   ```
3. The point set is passed to Julia for plotting
   ```
   plot(x,y,marker=(:x,6))
   ```
4. Optionally, various settings can be adjusted on the plot.

>>> Plotting: Example

```
# Plot sin(x) for x in [0,2pi]

using Plots
using LaTeXStrings
# Create vector with 50 equidistant support points
x = LinRange(0,2*pi,50);
y = zeros(50); # Initialization

# Calculate sin(x(i)) for all x(i)
for i ∈ 1:length(x)
     y[i] = sin(x[i]);
end

# Plot in red color
# Points (x[i],y[i]) are marked with a circle
p=plot(x,y,marker=(:circle,"red"));
```

>>> Plotting: Example

```
18 # Title and axis labels! Julia can also handle LaTeX
19 title!(p,"My first plot");
20 xlabel!(p, L"0 \leq x \leq 2 \pi");
21 ylabel!('sin(x)');
22
23 # Set axes
24 xlims!(p,0,2pi)
25 ylims!(p,-1,1)
26 # Save plot as png file
27 savefig(p,"sin.png");
```

---

with some extra help from the LaTeXStrings package.

>>> Multiple Plots in One Figure

```
# x1, y1 vectors with n elements
# x2, y2 vectors with m elements
p1 = plot(x1,y1,label="first");
plot!(p1,x2,y2,label="second");
```

Practice Exercises

1. Work on exercises 22 and 23 on the exercise sheet. Ask questions if needed—I'll come around!

>>> Outline

>>> Structure of a Julia Program

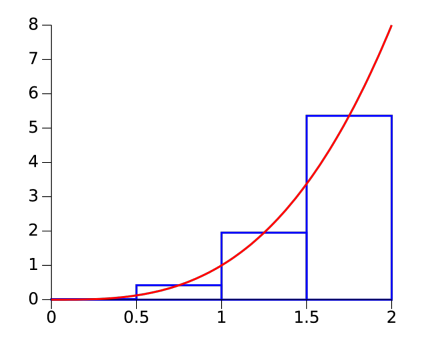
* We have learned how to work with scripts and functions.
* A program typically consists of one script and several functions and can be divided into three blocks:
  1. **Initialization:** Here, we prepare our variables and our vectors/matrices so that we can use them.
  2. **Processing:** This part of the code handles the operations that use our variables in some way.
  3. **Postprocessing:** Output, storage, visualization of results.

>>> Structure of a Julia Program

* It is a matter of style how many and how large functions are used. That being said, programming is the art of controlling complexity.
* In a finished program, the user should only need to enter the input parameters in the script and execute it. The processing and postprocessing should then run completely automatically.

>>> Numerical Integration

The integral $\int_a^b f(x)dx$ of a function is by definition the limit of Riemann sums. Thus, an integral can be approximately calculated by computing Riemann sums.



Procedure:

* Divide the interval $[a, b]$ into N subintervals with boundaries

$$a < a + h < a + 2h < \ldots < a + Nh = b\,,$$

  i.e., $h = \frac{b-a}{N}$.

* It holds that

$$\int_a^b f(x)dx = \sum_{k=0}^{N-1} \int_{a+kh}^{a+(k+1)h} f(x)dx\,.$$

The idea is that on a small interval $[c, c+h]$

$$\int_c^{c+h} f(x)dx \approx hf(c)\,.$$

This yields the (left) **Rectangle Rule**

$$I := \int_a^b f(x)dx \approx \sum_{k=0}^{N-1} hf(a+kh) =: I(h)\,.$$

The value $I(h)$, which depends on h, is an approximation of the exact value I of the integral.

Numerical aspects:

- Does the approximated value converge to the exact value for $h \to 0$, i.e., does $\lim_{h \to 0} I(h) = I$ hold?
- Does the approximated value converge for every (integrable) function?
- If yes, how fast does $I(h)$ converge to I, i.e., can something be said about $|I(h) - I|$?

$\to$ This is covered in Numerics 1, but the ideas carry over to the other courses too!

Spoiler: There is a constant c such that $|I(h) - I| \leq ch$ for every continuously differentiable function.

* What are the input parameters? The function f, the interval boundaries a, b, and the number N of subintervals. $\Rightarrow$ Initialization.
* What must the algorithm compute? Function evaluations $f(a + kh)$ for $k = 0, \ldots, N - 1$ and the sum $\sum hf(a + kh)$. $\Rightarrow$ Processing.
* What results should the algorithm provide and how should they be visualized? Here only the approximated result $I(h)$. $\Rightarrow$ Postprocessing.

>>> Numerical Integration: Pseudocode

1. Input: f, a, b, N.
2. Compute the function values $f_k = f(a + kh)$ for $k = 0, \ldots, N-1$.
3. Compute the products hf_k.
4. Form the sum $I(h) = \sum_{k=0}^{N-1} hf_k$.
5. Output the result $I(h)$.

>>> Logarithmic Error Plots

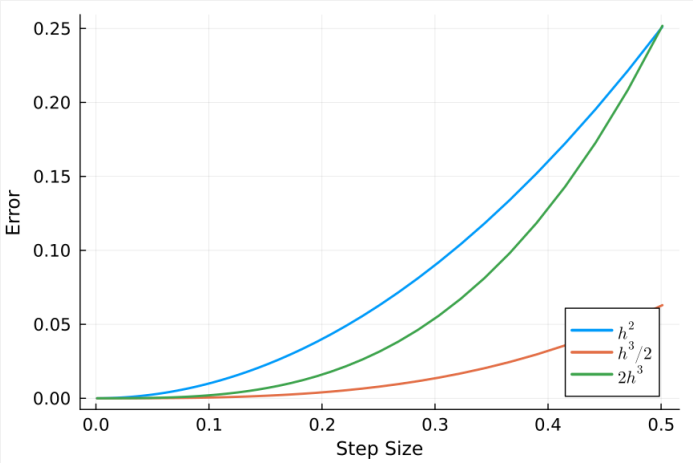
* Numerical results often depend on the chosen step size h.
* Examples: Width h of rectangles in integration; difference quotient, step size in differential equations ...
* One wants to know whether the approximated value converges to the exact value.
* For integrals: Does $I(h) \to I$ hold for $h \to 0$ or $|I(h) - I| \to 0$?
* If $I(h)$ is the approximated value depending on h, one can often show that $|I(h) - I| \leq ch^k$ with a natural number k (for the rectangle rule, $k = 1$).

>>> Logarithmic Plots in One Figure

* Logarithmic plots are very important in numerical analysis. For example, errors of algorithms are often given in terms of a step size h, Error $\leq h^2$. This means that halving the step size h reduces the error by a factor of $\frac{1}{2}^2 = \frac{1}{4}$.
* Question: How can we recognize this in Julia plots?

>>> Logarithmic Error Plots

* Here are three error curves given. How do the errors behave? Like h^2, h^3, h^4,... ?

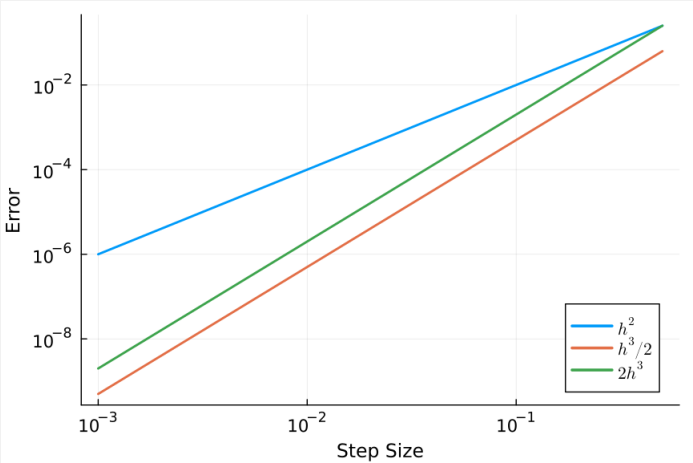


* If $I(h)$ is the approximated value depending on h, one can often show that $|I(h) - I| \leq ch^k$ with a natural number k (for the rectangle rule, $k = 1$).
* k is called the convergence order. The larger k, the faster the method converges.
* Because
$$\log(ch^k) = \log c + k \log h$$
the function $h \mapsto ch^k$ becomes a straight line with slope k after taking the logarithm.
* With a (doubly) logarithmic plot, the order of the method can be read from the slope of the "line" $h \mapsto |I(h) - I|$.

* The arguments xscale=:log10 and yscale=:log10 transform each scale to logarithmic.
* Everything else is kept the same!
* The x- and y-axes are then scaled logarithmically, not linearly.
* This allows verification that the method has the correct order.

>>> Logarithmic Error Plots

* It is very easy to read this in a loglog plot! Error curves with the same exponent are parallel to each other in loglog plots.



>>> Outline

>>> Error Finding

* When finishing writing a code, it will not work correctly, in most cases, on the first try.
* It is normal to spend a significant amount of time on debugging.
* There are different types of errors:
  * Syntax errors, e.g., `A = * A`: Easy to fix since Julia tells you what's wrong.
  * Logical errors: Hard to find because the program is error-free from the computer's perspective but doesn't produce the correct result.

>>> Typical Error Sources

* Wrong syntax, e.g., y = xy+1;
* Incomplete program, e.g., not considering negative numbers, wrong dimensions, row and column vectors, etc.
* Unexpected arguments for functions, e.g., passing vectors to a function that expects scalars.
* Unexpected data state, e.g., a variable needed later is accidentally overwritten.
* Logical errors, e.g., wrong conditions in if statements.

>>> Error Finding

Some tips:

* Stay calm: Errors in programming are absolutely normal.
* Systematic approach: Proceed slowly and methodically!
* Error messages: Read the error messages! At least the line numbers are understandable. Many errors can be fixed easily this way.
* Error assessment: Is it more likely a simple syntax error?
* Error isolation: Isolate the error:
    * Comment out parts of the code that aren't strictly necessary. Does the error still occur?
    * Did the code work before? What exactly did you change?
* Ask others! This is usually the fastest method.

>>> Performance

* Performance refers to the time behavior of a program.
* The faster a program runs and the less memory it requires, the better.
* Here: Performance is important when explicitly required in the programming task.
* Performance can be measured with the amount of memory allocations, or how much compiling is done or just by timing the entire execution of the code.

* There is no uniform definition of when a program is called efficient enough.
* Often, the runtime or number of required computations depends on a size N:
  * (Quadratic) equation systems with $(N \times N)$-matrices,
  * Numerical integration with N subintervals,
  * Calculation of the first N terms of a (recursive) sequence.
* The goal is that the runtime of the program depending on N does not grow too quickly with N.

* Sometimes an algorithm is called efficient if it has polynomial runtime, i.e., if the runtime is roughly proportional to cN^k with a constant c and an exponent k.
* Example (Numerics 1): Computational effort for calculating the LU decomposition (= Gaussian elimination) of an $(N \times N)$-matrix $\approx \frac{1}{3}N^3$.
* The above definition is useless if k is large, but it excludes algorithms whose runtime grows exponentially.

>>> Example: Fibonacci Sequence

The Fibonacci sequence is defined by

$$f_{n+2} = f_{n+1} + f_n\,, \quad f_1 = f_2 = 1\,.$$

A recursive function can be easily written:

```
function fib_recursive(n)
# Calculates the n-th element of the Fibonacci sequence recursively
if n <= 2
       fn = 1;
else
       fn = fib_recursive(n-1)+fib_recursive(n-2);
end
```

>>> Example: Fibonacci Sequence

* If you call the above function with n $= 3$, the if block is skipped and line 6 is executed.
* The function calls itself twice with n $= 2$ and n $= 1$.
* In these calls, fn = 1 is returned.
* In the original function call, fn = 1+1 is then calculated with the returned values.

>>> Example: Fibonacci Sequence

The above program already takes about a minute to compute f_{45}. Do you think the Fibonacci sequence is complicated enough to justify this?
Why is this program so slow? Hint: Think about how often the above function is called with an $n \in 1, 2$ when you want to compute f_4, f_5, f_6, etc.!

>>> Example: Fibonacci Sequence

Explanation: Only for $n = 1$ and $n = 2$ can the function directly return a result; otherwise, it must call itself. For example, f_6 is calculated as follows:

$$
\begin{aligned}
f_6 &= f_5 + f_4 \\
&= (f_4 + f_3) + (f_3 + f_2) \\
&= ((f_3 + f_2) + (f_2 + f_1)) + ((f_2 + f_1) + f_1) \\
&= (((f_2 + f_1) + 1) + (1 + 1)) + ((1 + 1) + 1) \\
&= (((1 + 1) + 1) + 2) + (2 + 1) \\
&= ((2 + 1) + 2) + 3 \\
&= (3 + 2) + 3 = 5 + 3 = 8
\end{aligned}
$$

This leads to an unnecessarily high number of function calls, and sequence terms are computed multiple times (e.g., f_3 is computed three times).

$\Rightarrow$ An iterative program is more efficient (exercise).

>>> Performance and Efficiency

* The performance of a program should generally only be optimized once the program already works.
* Julia provides a package BenchmarkTools, which contains the macro `@benchmark` or `@btime` that precisely measures the execution time and number of memory allocations that a function call needs.
* Rather irrelevant for Numerical Analysis, important for larger programs.

>>> Outline

>>> Key Takeaways

* Julia is a dynamically typed language.
* With an =, the expression on the right is evaluated and stored in the variable on the left.
* Loops (for and while) and if-statements.
* **Not helpful**: Memorizing all commands.
  Instead: Know what possibilities exist and use Google when needed!

Practice Exercises

1. Work on the remaining exercises on the exercise sheet. Ask questions if needed—I'll come around!

Questions?