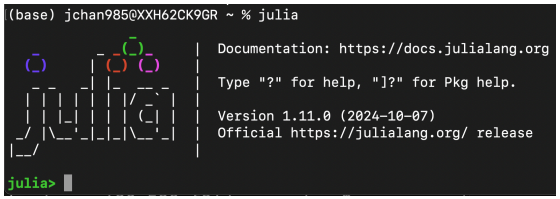
CMOR 420/520
Computational Science
Programming in Julia

An overview of Julia

- Modern programming languages like Python introduce:
  - Portability of an interpreted language
  - Dynamic typing (or at least non-explicit typing)
  - Fast JIT (just-in-time) compilation
  - Interactivity via the REPL (read-eval-print loop) or terminal
- Many of these features are incompatible with what makes C/C++ fast.
- Note: Julia has ***a lot*** of features. I won't be able to cover everything, but the Julia manual, the Julia Discourse forum, and the Slack/Zulip pages are good places to look for answers.

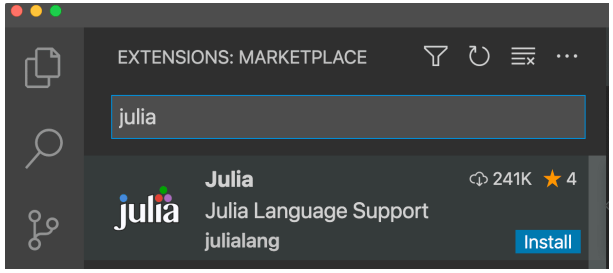
Installing and running the Julia REPL

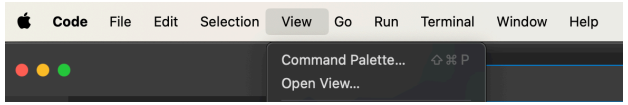
- Go to https://julialang.org/install/
  - Follow instructions to install the Juliaup version manager (manages multiple versions, alerts you when new versions are available).
  - Can also download a binary installer.
- Open up a Julia REPL through the terminal once installed.
- You can also execute a Julia file directly (easier for scripting, not covered)

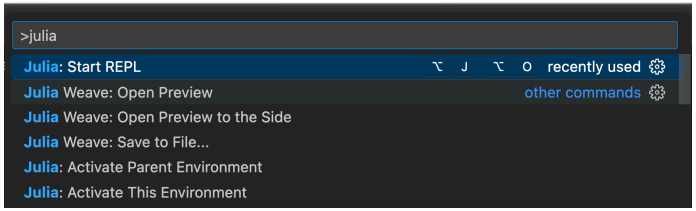


Setting up VSCode

- I use Julia through VSCode nowadays (VSCode also works with C, C++, and even LaTeX).
  - Download VSCode and install the Julia extension
  - To use the Julia REPL in VScode, use the “Command Palette” and execute “Julia: Start REPL”
- You can edit, run, profile, and analyze code in VSCode (also plotting and inline execution).







Executing hello world in Julia

- The Julia REPL is similar to the Matlab terminal. You can run a file using “include(“filename.jl”)”.
- Some observations:
  - You can define a function three different ways in Julia
  - “#” comments a line
  - Autocomplete and emacs bindings (e.g., ctrl-a, ctrl-e, ctrl-k) work in Julia VSCode

```julia
function hello_v1()
    println("Hello world")
end

hello_v2() = println("hello world")

hello_v3 = () -> println("hello world")

# four different ways to
# run "hello world"
println("hello world")
hello_v1()
hello_v2()
hello_v3()
```

hello.jl

```
julia> include("hello.jl")
```

Running the code in the REPL

Exercise

- Write a ’hello_world’ function and run it

The Julia REPL

- Running code leaves behind variables and functions in the workspace.
  - When we run “hello.jl”, we defined three functions. Using “varinfo()” shows all functions and variables currently in the workspace.
- To clear these, you need to restart the REPL. You usually don’t need to clear these variables (can just redefine them).
- You can also access the Linux shell through the REPL; this is one of three REPL environments (the REPL, the shell, and the Package environment)

```
julia> include("hello.jl")
hello world
Hello world
hello world
hello world

julia> varinfo()
  name        size summary
  -------- -------- ------------------------------------------------
  Base             Module
  Core             Module
  Main             Module
  hello_v1 0 bytes hello_v1 (generic function with 1 method)
  hello_v2 0 bytes hello_v2 (generic function with 1 method)
  hello_v3 0 bytes #15 (generic function with 1 method)
```

```
julia> # press ; to enter the shell

shell> echo "I'm now in the Linux shell"
I'm now in the Linux shell

julia> # backspace returns you to the REPL
```

Modules

- Modules contain functions and macros
- Main is the top-level module, and Julia starts with Main set as the current module
- Core is the module that contains all identifiers considered ”built in” to the language, i.e. part of the core language and not libraries
- Base is a module that contains basic functionality
- You will learn more about modules later

Accessing the Linux terminal in the REPL

- You can also access the Linux shell through the REPL by pressing ";"
- This is one of three REPL environments (the REPL, the shell, and the Package environment)
- We'll go more into the Package environment (accessed by "]") later.

```
julia> # press ; to enter the shell

shell> echo "I'm now in the Linux shell"
I'm now in the Linux shell

julia> # backspace returns you to the REPL
```

Matlab and Python-like functionality

- We won't go too much into Julia's feature set; generally, if you can do something in Matlab and Python, you can do it in Julia.
- You don't need to use any fancy features to write good, efficient Julia code! The most efficient Julia code tends to look like C or Fortran.

Loops and control flow that are available in Matlab, Python, C, C++ are also available in Julia.

```julia
for i in 1:100
    if (i % 2 == 0)
        # do something
    elseif (i < 10)
        # do something else
    else
        # ...
    end
end
```

like_matlab_python.jl

```julia
# ranges in Julia
i = 0:10
a = LinRange(-1, 2, 1000)

# an array comprehension
b = [2 * i for i in 1:10]

# tuples in Julia
c = ("hi", 1, 3.0)

# named tuples
d = (; a = "hi", b = 1,
       c = 3.0)
d.a # returns "hi"
```

Julia has a lot of Python-like features

Why is Julia sometimes faster than Python?

- Julia is marketed as a high performance code that is Python-like
  - This doesn't mean that any code written in Julia is fast; it means that certain codes written in native Julia can be significantly faster than the same code written in native Python.
  - Many operations will not be faster; Python outsources most computationally intensive tasks to optimized libraries written in C (e.g., non-native Python).
- To understand why Julia can be fast (and more importantly, when it is not!) we have to understand how Julia combines aspects of both Python and C++.

C/C++ vs Python: AOT vs JIT compilation

- C/C++ use Ahead-Of-Time (AOT) compilation.
  - The compiler analyzes the code and applies optimizations based on code structure and type information.
  - C/C++ is *statically typed*, so all variable types are known at *compile time*.
- Python code is Just-In-Time (JIT) compiled: the code is interpreted (compiled and linked with libraries) when you run it in a Python REPL.
  - Python is *dynamically typed,* so it figures out what variable types are while the code is running (at *run time*). However, because the code is compiled on-the-spot, expensive compiler optimizations would make execution feel laggy.
  - Efficient Python code typically relies on libraries written in C/C++.

Why Julia is (usually) fast

- Julia calls itself a Just-Ahead-Of-Time (JAOT) compiled language.
  - The first time a code is run, it is compiled, and the compiled code is cached.
  - The second time a code is run, the cached machine code is loaded, and no compilation is necessary *if the input types are the same as the first time.*

```
julia> x = randn(10, 10);

julia> @time x * x;
  0.335731 seconds (4.41 M allocations: 218.362 MiB, 4.87% gc time, 99.93% compilation time)

julia> @time x * x;
  0.000029 seconds (2 allocations: 944 bytes)

julia> x = ones(Int, 10, 10);

julia> @time x * x;
  0.120978 seconds (1.68 M allocations: 85.492 MiB, 5.48% gc time, 99.79% compilation time)
```

Julia as a…compiler?

- Recall: when you run code for the first time, Julia compiles it.
- Unlike Python, the Julia programming language allows you to analyze all the steps of compilation.
- Here, we're using macros @code_typed, @code_llvm, @code_native to analyze the way that Julia turns Python-like code into machine code.
- However, you don't need to know the details of this process to write efficient Julia code.

```julia
function my_add(x, y)
    return x + y
end
```

```
julia> @code_typed my_add(1, 2)
CodeInfo(
1 ─ %1 = Base.add_int(x, y)::Int64
└──      return %1
) => Int64
```

```
julia> @code_llvm my_add(1, 2)
; Function Signature: my_add(Int64, Int64)
;  @ /Users/jchan985/Desktop/cmor-420-520-f24/julia_codes/pisum.jl:12 within `my_add`
define i64 @julia_my_add_13226(i64 signext %"x::Int64", i64 signext %"y::Int64") #0 {
top:
;  @ /Users/jchan985/Desktop/cmor-420-520-f24/julia_codes/pisum.jl:13 within `my_add`
; ┌ @ int.jl:87 within `+`
  %0 = add i64 %"y::Int64", %"x::Int64"
; └
  ret i64 %0
}
```

```
julia> @code_native my_add(1, 2)
        .section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 14, 0
        .globl  _julia_my_add_12894             ; -- Begin function julia_my_add_12894
        .p2align        2
_julia_my_add_12894:                    ; @julia_my_add_12894
; Function Signature: my_add(Int64, Int64)
; ┌ @ /Users/jchan985/Desktop/cmor-420-520-f24/julia_codes/pisum.jl:12 within `my_add`
; %bb.0:                                ; %top
; │ @ /Users/jchan985/Desktop/cmor-420-520-f24/julia_codes/pisum.jl within `my_add`
        ;DEBUG_VALUE: my_add:x <- $x0
        ;DEBUG_VALUE: my_add:x <- $x0
        ;DEBUG_VALUE: my_add:y <- $x1
        ;DEBUG_VALUE: my_add:y <- $x1
; │ @ /Users/jchan985/Desktop/cmor-420-520-f24/julia_codes/pisum.jl:13 within `my_add`
; │ ┌ @ int.jl:87 within `+`
        add     x0, x1, x0
; │ └
        ret
; └
                                        ; -- End function
.subsections_via_symbols
```

Macros in Julia

- The “@“ character indicates a macro. These are similar to C macros; we will go into this more when we talk about metaprogramming in Julia.
- A lot of convenient features in Julia are implemented as macros:
  - @show: print out variables
  - @time: time a function call
  - @which: Julia uses a generalization of function overloading; this macro prints out which function actually gets called for specific arguments.

```julia
function my_add(x, y)
    @show x, y
    return x + y
end
```

```
julia> my_add(1, 2)
(x, y) = (1, 2)
3
```

```
julia> x = randn(1_000_000); y = -x;

julia> @time my_add(x, y);
  0.001990 seconds (3 allocations: 7.629 MiB, 32.38% gc time
```

```
julia> @which my_add(1, 2)
my_add(x, y)
     @ Main ~/Desktop/cmor-420-520-f24/julia_codes/test.jl:1

julia> x = randn(10);

julia> @which 2 * x
*(A::Number, B::AbstractArray)
     @ Base arraymath.jl:21
```

Exercise

- Make a function called 'LinearFunc' that:
  - has input 'x'
  - within the function, set m=2 and b=5
  - use @show to show m,b
  - outputs m*x+b
- use @time to time the compilation i.e. time the include("filename.jl") step
- use @time and run LinearFunc(2)

Splatting and slurping

- The "…" in Julia is used to either expand out (splat) arguments or collect (slurp) arguments.

```julia
my_norm(x, y) = sqrt(x^2 + y^2)
point = [1.0, 2.0]
my_norm(point...)
```

An example of "splatting" arguments.

```julia
function my_norm(coordinates...)
    @show coordinates
    val = 0.0
    for i in eachindex(coordinates)
        val += coordinates[i]^2
    end
    return sqrt(val)
end
```

```julia
my_norm(1.0, 2.0, 3.0, 4.0)
```

An example of "slurping" arguments.

Types in Julia

- All variables have a type in Julia; you can examine this using “typeof”
- Some types have parameters (similar to template parameters in C++).
  - Example: “Rational{Int64}”
  - Example: “UnitRange{Int64}”
- A very common type is an Array{T, N} for an N-dimensional array
  - For example, Vector{T} and Matrix{T} are aliases for Array{T, 1} and Array{T, 2}.

```
julia> typeof(1)
Int64

julia> typeof(1.0)
Float64

julia> typeof("hello")
String

julia> typeof(1 // 3)
Rational{Int64}

julia> typeof(1:4)
UnitRange{Int64}

julia> typeof(LinRange(0, 1, 10))
LinRange{Float64, Int64}
```

```
julia> typeof(zeros(10))
Vector{Float64} (alias for Array{Float64, 1})
```

Arrays in Julia

- Vectors and matrices (and tensors!) are “Array{T, N}” types in Julia.
  - Vector{T} is an alias for Array{T, 1}.
    Matrix{T} is an alias for Array{T, 2}.
- You can access array entries via “x[j]”, and slice vectors (e.g., “x[1:3]”) or matrices (e.g., “A[:, 2]”).
- Arrays behave like pointers; if you set one array equal to another and change its values, the other array changes as well.
- Some operations will create a copy of the array instead (e.g., “y = -x” will copy).

```
julia> x = [1, 2, 3, 4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> y = x; y[1] = 100;

julia> x
4-element Vector{Int64}:
 100
   2
   3
   4
```

Vectors and Arrays in julia

- Defining vectors is similar to Matlab and Fortran
- A ',' separates row entries and a ';' separates columns
- Many additional predefined functions such as

```julia
# Creating a 1D Vector
V = [1, 2, 3, 4]

# push 5 in vector
push!(V, 5)
```

```julia
# Creating a 1D Vector
V = [1, 2, 3, 4, 5]

# remove 5 from vector
pop!(V)
```

Vectors and Arrays in julia

- Defining matrices is similar to Matlab and Fortran
- A space separates row entries and a ';' separates columns
- Many additional predefined functions such as

```
# Defining a square matrix of size (2, 2)
A = [1 2; 3 4]

# Transpose of A
A'
```

```
reshape(A, (2, 3))
```

Broadcasting

- In Matlab/Python, broadcasting applies an operation to the elements of an array (e.g., broadcasting “+” over arrays)
- In Julia, you can broadcast *any* function or operator. Examples:
  - “x .= y” sets “x[j] = y[j]” for j = 1:length(x)
  - “foo.([1, 2], [3, 4])” returns “[foo(1,3), foo(2, 4)]”
  - “bar.([[1, 2], [3, 4]])” returns “[bar([1, 2]), bar([3, 4])]”
- Can use the “@.” macro to broadcast everything in an expression.

```
x = [[1, 2], [3, 4], [5, 6]]
y = [[1, 2], [3, 4], [5, 6]]
```

```
x .- y
```

```
3-element Vector{Vector{Int64}}:
 [0, 0]
 [0, 0]
 [0, 0]
```

```
((x, y) -> [x, y]).(x, y)
```

```
4-element Vector{Vector{Int64}}:
 [1, 5]
 [2, 6]
 [3, 7]
 [4, 8]
```

Incorrect!

```
@. x += 2 * y + 3
```

```
x = @. sin(2 * y + 3)
```

```
@. my_norm(x, y)
```

Exercise

- Make a new 'LinearFunc' called 'LinarFunc_v2'to take:
  - input of any length using splatting `and/or slurping`
  - use @. to broadcast the operation m*x+b to all entries, within the function definition
- Run LinearFunc_v2(2,3,4)

- Alternatively you can do `@. LinearFunc([2,3,4])` to broadcast
  Try both!

Modules

- To include a module into code: 'using Module_name'
  - This load Module_name' and adds it to the global namespace
  - Anything in 'Module_name' is acessed through Module_name.function_name'
- Can include specific functions belonging to a module using 'import': 'import Module_name: function_name'

```julia
using LinearAlgebra

x=[1;2;3;4]
y=[5;6;7;8]

@show LinearAlgebra.dot(x,y)
```

```julia
import LinearAlgebra: dot

x=[1;2;3;4]
y=[5;6;7;8]

@show dot(x,y)
```

Multiple dispatch

- Similar to function overloading in C++, but with a few key differences:
  - Arguments are checked at run time instead of at compile time.
  - You can dispatch on partially specialized methods (similar to partial specialization for C++ template parameters).
- Julia multiple dispatch is *fast.*
- Enables extending functionality to custom user-defined types.

```julia
my_norm(x, y, z) = sqrt(x^2 + y^2 + z^2)

function my_norm(x::Vector, y::Vector, z::Vector)
    return @. sqrt(x^2 + y^2 + z^2)
end

function my_norm(message::String, x, y, z)
    println(message)
    return my_norm(x, y, z)
end
```

```
julia> methods(my_norm)
# 5 methods for generic function "my_norm" from Main:
 [1] my_norm(message::String, x, y, z)
     @ ~/Desktop/cmor-420-520-f24/julia_codes/test.jl:44
 [2] my_norm(x::Vector, y::Vector, z::Vector)
     @ ~/Desktop/cmor-420-520-f24/julia_codes/test.jl:40
 [3] my_norm(x, y)
     @ ~/Desktop/cmor-420-520-f24/julia_codes/test.jl:10
 [4] my_norm(x, y, z)
     @ ~/Desktop/cmor-420-520-f24/julia_codes/test.jl:38
 [5] my_norm(coordinates...)
     @ ~/Desktop/cmor-420-520-f24/julia_codes/test.jl:14
```

Exercise

- Use multiple dispatch on a new function called 'LinearComb':
  - First version should take scalar arguments x, y, z and return x+2*y+3*z
  - Second version should be specialized to take vector arguments and retunr the same as the first version
  - Third version should take an additional vector argument called c which stores the coefficients of the linear combination i.e. returns c[1]*x+c[2]*y+c[3]*z

Custom types

- Custom (or composite) types in Julia are just structs (the same as C).
- They can include fields of any type.
- If you do not specify a type, the field is assumed to be of type “Any” (the most general Julia type).
  - Note: computing with types “Any” can be bad for performance. We’ll go over this in more detail when we discuss “type stability”.
- Multiple dispatch works with custom types the same way as built-in types.

```julia
struct Point2D
    x::Float64
    y::Float64
end

struct Point3D
    x
    y
    z
end

my_norm(point::Point2D) =
    my_norm(point.x, point.y)
```

Custom parametric types

- If you don't want to specify the type of an argument, you can instead let the type of each field become a type *parameter*.
- Types with parameters are *parametric types.*
- You can also include extra parameters that are not the type of a struct field.
- What is the difference between this and not specifying the type of a field?
  - If you don't specify the type of a field, it is assumed to be of abstract type "Any". If any field is an abstract type, you can run into efficiency issues like *type instabilities.*

```julia
struct Point2D{T}
    x::T
    y::T
end
```

```
julia> Point2D(1.0, 2.0)
Point2D{Float64}(1.0, 2.0)
```

```julia
struct Point3D{T, ExtraParameter}
    x::T
    y::T
    z::T
end
```

Constructors for custom types

- Every type has a default constructor (one argument for each field)
- You can use multiple dispatch to create new constructors.
- You can also overwrite the default constructor with an “inner constructor” defined inside the struct.
  - This is the only function you can define inside a Julia structure.

Should be y>x

```julia
struct OrderedPair
    x::Real
    y::Real
    function OrderedPair(x, y)
        x > y && error("Arguments out of order.")
        return new(x, y)
    end
end

OrderedPair(xy::Pair) =
    OrderedPair(xy.first, xy.second)
```

Exercise

- Make a custom type called ’3DVector’ that has 3 arguments
  - Make it a parametric type (a general type T)
  - make a cosutom constructor that just prints the message ’Made a new 3D vector’

The type hierarchy

- All Julia types are part of a type hierarchy; if no super type is specified, the supertype is “Any” by default.
- You can determine if a type is a subtype via, e.g., “supertype(Float64)

```
julia> supertype(Float64)
AbstractFloat

julia> supertype(supertype(Float64))
Real

julia> supertype(supertype(supertype(Float64)))
Number

julia> supertype(supertype(supertype(supertype(Float64))))
Any
```

Custom Constructors for Parametric types

- Declare the parametric datatype *T* in the custom constructors name and using 'where *T*'
- Create an instance of the parametric type using 'Name{datatype}(variables)'
- Overload the constructor for each concrete datatype so that 'Name(variable)' is the constructor for any datatype

```julia
struct Vector3D{T}
    x::T
    y::T
    z::T
    function Vector3D{T}(x::T,y::T,z::T) where T
        println("New 3d vector")
        return new(x,y,z)
    end
end
```

```
Vector3D{Float64}(1.0, 2.0, 3.0)

Vector3D(x::Float64, y::Float64, z::Float64)=Vector3D{Float64}(x, y, z)
Vector3D(x::Int64, y::Int64, z::Int64)=Vector3D{Int64}(x, y, z)

Vector3D(2.0, 4.0, 6.0)
Vector3D(4, 8, 12)
```

Specifying Supertype for a Parametric type *T*

- For a parametric type *T*, you can specify the supertype using $<:$
- Improves the speed of the code because *T* is no longer the 'Any' datatype and is restricted to a more specific range

```
struct Vector3D{T<:Real}
    Click to collapse the range.
    y::T
    z::T
    function Vector3D{T}(x::T,y::T,z::T) where T<: Real
        println("New 3d vector")
        return new(x,y,z)
    end
end
```

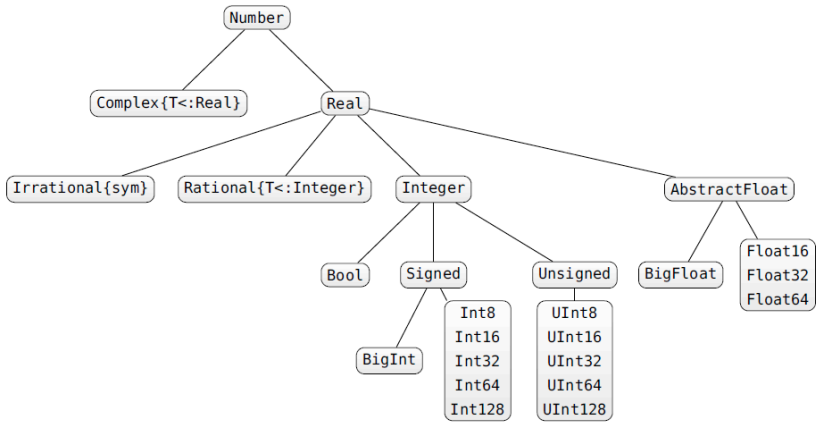
Abstract types

- All supertypes in Julia are *abstract types* (in contrast to *concrete types*). You cannot create an instance of an abstract type (e.g., “AbstractFloat”) in Julia; you can only instantiate concrete types (e.g., “Float64”).
- You can create your own type hierarchy by defining custom abstract types, or by plugging into existing abstract types (like “AbstractVector”)
- The “<:” operator in Julia checks if one type is a subtype of another, or can be used to declare that a new type is a subtype of an abstract type.

```julia
abstract type AbstractPoint end
abstract type AbstractPoint2D <: AbstractPoint end
struct Point2D <: AbstractPoint2D
    x::Float64
    y::Float64
end
```

An example of a type hierarchy

- A type hierarchy is just used to apply multiple dispatch at different levels.
- Abstract types just serve as nodes in the tree; they cannot be instantiated. Only concrete types (leaves of the type tree) can be instantiated.



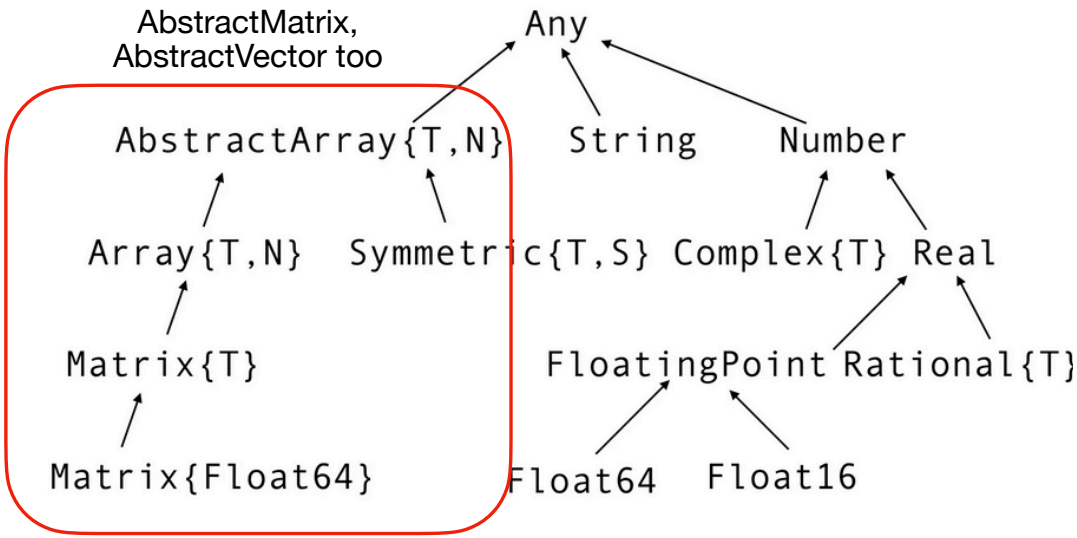
You can check if something is an abstract or concrete type via `isconcretetype` or `isabstracttype`.

https://github.com/bkamins/The-Julia-Express

Alternatives to classes in Julia?

- Julia **does not support classes and object-oriented programming**.
- Instead, Julia relies on multiple dispatch, interfaces, and “duck typing”.
  - Duck typing: “the code itself doesn't care about whether an object is a duck, just about whether it quacks.”
  - Downside: duck typing can lead to run time errors if certain methods are missing. Sometimes it’s also unclear what specific method is missing.
- *Interfaces* in Julia enable complex functionality if you extend a few functions for a custom type.
  - Examples: Iterators, indexing, AbstractArray, customized broadcasting.

Type hierarchy around AbstractArray



https://www.slideshare.net/acidflask/designing-linear-algebra-into-julia#1

AbstractArray interface in Julia

- https://docs.julialang.org/en/v1/manual/interfaces/#man-interface-array
- To create an AbstractArray, just do the following:
  - Create a custom array type which is a subtype of AbstractArray (e.g., MyArrayType <: AbstractArray)
  - Specialize “size(x::MyArrayType)”, which returns the dimensions of the array
  - Specialize “getindex(x::MyArrayType, index)”
    - Alternatively, specialize “getindex(x::MyArrayType, index_1, index_2, …)”
- That’s it! You should be able to treat this vector as a standard one now.

* size and getindex belong to the Base module for AbstractArrays, so these must be specialized for the custom type

Exercise: the AbstractArray interface

- First, get Julia and VSCode set up and running.
- Create a type for a canonical basis vector with all zero entries except the “i”th entry
- This lends itself to efficient implementations of linear algebra operations.
  - For example: computing $x^T A y$ when x or y (or both) are canonical basis vectors.
  - It should also (mostly) behave like a standard Vector.

```julia
# define CanonicalBasisVector type as a new AbstractVector
struct CanonicalBasisVector <: AbstractVector{Bool}
    length::Int64
    index::Int64
end

# using "import" to extend Julia's built-in functions
import Base: size, getindex
Base.size(x::CanonicalBasisVector) = (x.length, )
Base.getindex(x::CanonicalBasisVector, i) =(i == x.index )
```

```julia
import LinearAlgebra: dot

# Specialize the following methods:
dot(x::CanonicalBasisVector, A, y)
dot(x, A, y::CanonicalBasisVector)
dot(x::CanonicalBasisVector, A,
    y::CanonicalBasisVector)
```

When is Julia *not* efficient? Example 1

- We will install the package BenchmarkTools.jl via “using Pkg; Pkg.add("BenchmarkTools")”
  - the `@btime` macro times a bunch of samples and takes the minimum timing.

```julia
struct Point2D
    x
    y
end

mynorm(x, y) = sqrt(x^2 + y^2)
```

```julia
function calc_mynorm()
    mynorms = zeros(100)
    x = randn(100)
    y = randn(100)
    for i in 1:100
        mynorms[i] = mynorm(x[i], y[i])
    end
    return mynorms
end
```

```julia
function calc_mynorm2()
    points = [Point2D(randn(), randn()) for i in 1:100]
    mynorms = zeros(100)
    for i in 1:100
        mynorms[i] = mynorm(points[i].x, points[i].y)
    end
    return mynorms
end
```

When is Julia *not* efficient? Example 1

```julia
struct Point2D
    x
    y
end
```

```julia
function calc_mynorm2()
    points = [Point2D(randn(), randn()) for i in 1:100]
    mynorms = zeros(100)
    for i in 1:100
        mynorms[i] = mynorm(points[i].x, points[i].y)
    end
    return mynorms
end
```

- What went wrong here? Two hints:
  - The issue is in the “Point2D” type and is related to why Julia is fast.
  - Julia cannot compile fast code unless it knows the type of each variable involved. Because the “Point2D” fields do not specify a type, they are assumed to be of the most general (and slow) “Any” type.

When is Julia *not* efficient? Example 2

```julia
function mysum()
    val = 0.0
    for j = 1:1000
        val += sin(j)
    end
    return val
end
```

```julia
scaling = 1.0
function mysum2()
    val = 0.0
    for j = 1:1000
        val += scaling * sin(j)
    end
    return val
end
```

- Which one do you think is faster?
- Why do you think the slower code is slower?

Type instabilities

- These two examples are both instances of *type instability*
- Recall how Julia works for a given function: it compiles a C function for a specific set of inputs during the first run, and then reuses that compiled code.
- This runtime compilation step involves two things:
  - Type-agnostic Julia code
  - The types of the input variables
- If Julia can *infer* all variable types inside a function from the Julia code and input variables, this is referred to as *type stable* code.
- A *type instability* is when Julia cannot infer all variable types inside a function.

Type stable code, illustrated

Inputs

```
julia> x = 1; typeof(x)
Int64
```

Generic (no type) function

```
foo(x) = 2 * x
```

Julia creates a statically typed function

```
julia> @code_typed foo(1)
CodeInfo(
1 ─ %1 = Base.mul_int(2, x)::Int64
└──      return %1
) => Int64
```

Fast compiled code

Type unstable code, illustrated

Inputs

```
julia> x = 1; typeof(x)
Int64
```

Generic (no type) function

```
foo(x) = 2 * x + y
```

Some variable that the function doesn't know about

```
y = 4
```

Julia creates a statically typed function, but the type of "y" is unknown

```
julia> @code_warntype foo(1)
MethodInstance for foo(::Int64)
  from foo(x) @ Main ~/Desktop/c
t_examples.jl:80
Arguments
  #self#::Core.Const(foo)
  x::Int64
Body::Any
1 ─ %1 = Main.:+::Core.Const(+)
    %2 = (2 * x)::Int64
    %3 = Main.y::Any
    %4 = (%1)(%2, %3)::Any
└──      return %4
```

Slow compiled code to deal with a variable of unknown type

Examples (and analysis) of type instabilities

- Consider two versions of the “pos” function; which is type stable?
- Using untyped global variables or untyped struct fields also introduces a type instability.
- Using non-concrete (e.g., abstract) types in a container is more general, but also leads to a type instability and slower code (see Example 3 in “inefficient_examples.jl”)

```julia
pos(x) = x < 0 ? 0 : x
```

```julia
pos(x) = x < 0 ? zero(x) : x
```

```julia
x = randn(10)
@btime sum(x)
y = Real[]
for i in 1:10
    push!(y, randn())
end
@btime sum(y)
```

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

Detecting a type instability

- How to detect a type instability? You don't have to memorize all ways a type instability can be triggered.
- Julia provides the macro "@code_warntype" to check for type instabilities. Running "@code_warntype some_function(some_inputs)" will print out the inferred types of all input, output, and intermediate variables.
- Variables in red indicate type instability.

```julia
scaling = 1.0
function mysum2()
    val = 0.0
    for j = 1:1000
        val += scaling * sin(j)
    end
    return val
end
```

```
julia> @code_warntype mysum2()
MethodInstance for mysum2()
  from mysum2() @ Main ~/Desktop/cmor-420-520-f24/ju
Arguments
  #self#::Core.Const(mysum2)
Locals
  @_2::Union{Nothing, Tuple{Int64, Int64}}
  val::Any
  j::Int64
Body::Any
1 ─       (val = 0.0)
    %2  = (1:1000)::Core.Const(1:1000)
          (@_2 = Base.iterate(%2))
    %4  = @_2::Core.Const((1, 1))
    %5  = (%4 === nothing)::Core.Const(false)
    %6  = Base.not_int(%5)::Core.Const(true)
└──       goto #4 if not %6
2 ┄ %8  = @_2::Tuple{Int64, Int64}
          (j = Core.getfield(%8, 1))
    %10 = Core.getfield(%8, 2)::Int64
    %11 = Main.:+::Core.Const(+)
    %12 = val::Any
    %13 = Main.:*::Core.Const(*)
    %14 = Main.scaling::Any
    %15 = j::Int64
    %16 = Main.sin(%15)::Float64
    %17 = (%13)(%14, %16)::Any
          (val = (%11)(%12, %17))
          (@_2 = Base.iterate(%2, %10))
    %20 = @_2::Union{Nothing, Tuple{Int64, Int64}}
    %21 = (%20 === nothing)::Bool
    %22 = Base.not_int(%21)::Bool
└──       goto #4 if not %22
3 ─       goto #2
4 ┄ %25 = val::Any
└──       return %25
```

When is Julia *not* efficient? Example 4

```julia
function calc_mynorm()
    mynorms = zeros(100)
    x = randn(100)
    y = randn(100)
    for i in 1:100
        mynorms[i] = mynorm(x[i], y[i])
    end
    return mynorms
end
```

```julia
function calc_mynorm3()
    mynorms = zeros(100)
    xy = randn(2, 100)
    for i in 1:100
        pt = xy[:, i]
        mynorms[i] = mynorm(pt[1], pt[2])
    end
    return mynorms
end
```

- Which one do you think is faster? Why do you think this is the case?
- The second one is slower, not due to type instability, but due to *slicing*, which creates a new vector (and produces allocations) inside of a loop.

How to figure out if your Julia code is efficient

- Check for type instabilities using “@code_warntype” or Cthulhu.jl
- Check for allocations using “@time” or “@btime” from BenchmarkTools.jl
- Profile your code using “Profile.@profile” or “@profview” in VSCode.
  - Note: Julia uses a “sampling” profiler and flame graph visualization by default. I prefer VSCode’s visualization, which highlights time-consuming lines of code.
  - There is also an allocation profiler via “Profile.Allocs.@profile” or “@profview_allocs” in VSCode.
- Apply C/C++ optimizations (e.g., accessing memory in a contiguous fashion)
- Apply shared memory parallelism (via Base.Threads); not covered in this class.

Exercise

- Download unstable.jl from canvas
- Resolve the type instabilities. (Do not delete the struct. I know it does essentially nothing but this is for practice)

- The fastest it can be is approximatly 100 ns

External libraries in Julia

- We've talked about how to write efficient Julia code, but less about how to manage Julia projects - which typically involve external libraries.
- External libraries can be great (you avoid re-inventing the wheel) but managing external libraries can be annoying!
  - In C and C++, you (or a system administrator) often install packages manually. Installation may need to be customized for your specific machine.
  - In Python, you can install packages via package manager (conda, pip, etc). However, packages change, and your code may work under one version and break under another.
- Julia tries (not always successfully) to deal with both of these issues via their package manager "Pkg".

The Julia package ecosystem

- The Julia REPL provides a Pkg mode (similar to the “shell” mode)
- You can query “status” to see what packages are currently installed.
- You can install a package via “add SomePackageName” or remove a package via “rm SomePackageName”
- Note: anything you do in Pkg mode can also be done directly in the REPL by loading the Pkg package via “using Pkg”.

```
julia> # this is Julia REPL

(@v1.10) pkg> # pressing "]" gives Pkg mode
```

```
(@v1.11) pkg> status
Status `~/.julia/environments/v1.11/Project.toml`
  [295af30f] Revise v3.6.2
  [37e2e46d] LinearAlgebra v1.11.0
```

```
(@v1.11) pkg> add SparseArrays
   Resolving package versions...
    Updating `~/.julia/environments/v1.11/Project.toml`
  [2f01184e] + SparseArrays v1.11.0
    Updating `~/.julia/environments/v1.11/Manifest.toml`
  [2f01184e] + SparseArrays v1.11.0
  [bea87d4a] + SuiteSparse_jll v7.7.0+0
```

Local vs global package environments

- If you're working on multiple projects, you may not want to install the same version of a package for every single project.
- You can create a local package environment tied to

```
(@v1.11) pkg> activate .
  Activating project at `~/Desktop/cmor-420-520-f24/julia_codes`

(julia_codes) pkg>
```

- Executing "activate" (no dot ".") in Pkg mode returns you to the *global* environment. Installing packages here means they are always loaded (this can lead to slow start-up since Julia tries to precompile installed packages).

```
(julia_codes) pkg> activate
  Activating project at `~/.julia/environments/v1.11`

(@v1.11) pkg>
```

The Project.toml and Manifest.toml files

- If you initialize a local package, you generate two files: a Project.toml file and a Manifest.toml file.
- **The Project.toml file** is automatically edited when you install a package. It is also used to impose compatibility restrictions (i.e., enforce the use of a specific version of a package)
- **The Manifest.toml file** is not intended to be user-edited; Julia will create this automatically from the Project.toml file. It keeps track of all information about the currently installed packages.

TOML

Article Talk

From Wikipedia, the free encyclopedia

Tom's Obvious, Minimal Language (**TOML**, originally *Tom's Own Markup Language*[2]) is a file format for configuration files.[3] It is intended to be easy

Project.toml

```
[deps]
BenchmarkTools = "6e4b80f9-dd63-53aa-95a3-0cdb28fa8baf"
Cthulhu = "f68482b8-f384-11e8-15f7-abe071a5a75f"
```

A simple Project.toml file

An example Project.toml file

Project.toml

```
[deps]
BenchmarkTools = "6e4b80f9-dd63-53aa-95a3-0cdb28fa8baf"
Cthulhu = "f68482b8-f384-11e8-15f7-abe071a5a75f"

[compat]
BenchmarkTools = "<1.4"
Cthulhu = "=2.15.0"
```

- Project.toml files contain package hash identifiers, but can also be used to impose version restrictions on packages.
- See https://pkgdocs.julialang.org/v1/compatibility/#Version-specifier-format for a full list of version specifier operators.

What do we mean by a package “version”?

- Julia uses semantic versioning (SemVer), which specifies version numbers in MAJOR.MINOR.PATCH format.
  - Example: Julia is currently version 1.11.1,
- The idea is that changes in the each version indicate the level of change you make. You should increment the
  - MAJOR version when you make incompatible API changes
  - MINOR version when you add functionality that is backward compatible
  - PATCH version when you make backward compatible bug fixes

Exercise

- Make a directory called 'Package_Exercise'
- Make a local package environment
- Install BenchmarkTools and SparseArrays
- Enforce BenchmarkTools version 1.5.0

- What happens if you try to enforce SparseArrays version 1.9.0

What packages are available to install?

- Registered Julia packages via “add PackageNameHere”
  - Julia keeps a official registry of registered packages. The process of registering a package is remarkably easy (several undergraduates have done so) and basically just requires filling out a form.
- Unregistered Julia packages via “add https://github.com/PathToThePackage.jl”.
  - If you have a package which hasn’t been registered, you can install it by providing a Github link, so long as the repository has the appropriate directory structure (Julia also provides “Pkg.generate” to help initialize).
  - For example, “add https://github.com/giordano/StarWarsArrays.jl" installs the un-registered StarWarsArrays.jl package.

What else can you do with Pkg mode?

- Pkg mode is pretty powerful; you can install:
  - a specific version of a package via e.g., “add PackageName@2.1”
  - a specific branch of a package via e.g., “add PackageName#branch_name”. This can also be used to add a specific commit of a package (see https://pkgdocs.julialang.org/v1/managing-packages/#Adding-packages)
- You can run “update” to update all packages to the most recent versions (while preserving version compatibility). “resolve” can similarly be used to resolve incompatible versions.
- You can also use Pkg mode to run automated tests for a package using Julia’s automatic continuous integration (CI) framework “Test.jl”.

Integration with Github

- Most Julia packages are hosted on Github, with nicely-formatted documentation, CI testing status, and other badge links.

- Julia is designed to allow users to easily contribute and make pull requests.
  - The REPL's Pkg mode lets you "dev" a package, which clones a copy of the package onto your machine. The "dev" feature lets you modify this package and have the changes show up when you load the package.
  - You can run a package's automated testing in the REPL's Pkg mode by running "test PkgName"

Misc. notes on Julia's package manager

- If you add several packages, the order in which you add them matters.
  - Julia will add the latest version of the first package, then use that package's compatibility requirements to determine which version of the next package to install, and so on.
  - This means the version of packages that you will get may be order-dependent.

Note: it is possible to run into version conflicts when adding multiple packages!

```
(@v1.6) pkg> add DataFrames@0.21 CSV@0.5 Tables@1
   Resolving package versions...
ERROR: Unsatisfiable requirements detected for package CSV [336ed68f]:
 CSV [336ed68f] log:
 ├possible versions are: 0.3.0-0.7.7 or uninstalled
 ├restricted to versions 0.5 by an explicit requirement, leaving only versions 0.5.0-0.5.26
 ├restricted by compatibility requirements with Tables [bd369af6] to versions: [0.3.0-0.3.1, 0.5.24
-0.7.7] or uninstalled, leaving only versions: 0.5.24-0.5.26
 │ └Tables [bd369af6] log:
 │   ├possible versions are: 0.1.0-1.1.0 or uninstalled
 │   └restricted to versions 1 by an explicit requirement, leaving only versions 1.0.0-1.1.0
 └restricted by compatibility requirements with DataFrames [a93c6f00] to versions: 0.6.2-0.7.7 or u
ninstalled - no versions left
   └DataFrames [a93c6f00] log:
     ├possible versions are: 0.11.7-0.21.8 or uninstalled
     └restricted to versions 0.21 by an explicit requirement, leaving only versions 0.21.0-0.21.8
```

A brief note on Julia's style guide

- Julia has a style guide as part of the manual https://docs.julialang.org/en/v1/manual/style-guide/. You don't have to follow it, but it is helpful to know.
- Some are just generally good suggestions (e.g., avoid abbreviations in function names when possible).
- The Julia-specific stylistic points we will follow:
  - Module names and type names should use CamlCase
  - Function and variable names should use snake_case
  - Functions which modify one of their arguments should end in "!" (e.g., "sort!")

Base vs external packages

- Julia has several categories of packages:
  - Base: there is a lot of functionality in “Base” packages, which are available with any installation of Julia (e.g., LinearAlgebra, SparseArrays).
    - These packages are maintained by Julia Computing, Inc (e.g, Julia itself), and are generally the most reliable and stable.
  - External packages: packages which are not maintained by Julia Computing, Incorporated. Julia tends to leave a lot up to open source packages.
    - There are packages just maintained by individuals.
    - There are official Julia communities (JuliaStats, JuliaPlots, etc). When a package is matured it is often moved into one of these communities.

Some useful packages to know

- Linear algebra: for those coming from Matlab and Python, LinearAlgebra.jl is IMO the most important Julia library. SparseArrays.jl provides sparse matrix support, though it is less full-featured than Matlab's sparse arrays.
- Plots.jl is not an official Julia package, but it is the main visualization package people tend to use.
  - Plotting is less mature in Julia. Matlab > Python >> Julia when it comes to ease and polishedness of plotting. Makie.jl is a promising alternative.
- We will go over several other packages later (ForwardDiff.jl, StaticArrays.jl, JuMP.jl, OrdinaryDiffEq.jl) which IMO are representative of good Julia libraries.

Linear algebra

- LinearAlgebra.jl supports dense matrix operations, including:
  - Standard matrix operations: matrix multiplication, “\” back solve, linear least squares problems, etc.
  - Factorizations: LU, QR, Cholesky, LDLt, Schur, etc
  - Computing eigenvalues, SVD, etc.
    - Some special matrix types (e.g., Symmetric, Hermitian) are used to trigger specialized eigenvalue or SVD algorithms.
- Special types like the Adjoint type (lazy transposes) or Diagonal.
- Most things you can do in Matlab you can do in Julia, but some are not in Base.

Plots.jl

- Not an official Julia package, but pretty important to the Julia ecosystem.
- Plots.jl does not implement plotting, but provides a unified *interface* to multiple plotting *backends* (GR, PyPlot, Plotly, etc).
  - Can use “gr()” or “plotly()” to switch backends and specify settings.
  - Some backends like Plotly support interactivity.
- Plotting on top of another plot (e.g., “hold on” in Matlab) is achieved by calling a plot command with “!” after it (e.g., “scatter!” or “title!”)
- Making movies is incredibly simple; see the “@gif” macro in Plots.jl.

How do you create a Julia package?

- It turns out that it is remarkably easy to create your own Julia package!
- Use “generate” in the REPL’s Pkg mode to initialize the minimum structure necessary for a package.
  - Note: PkgTemplates.jl can be used to perform a much more in-depth initialization (e.g., setting up CodeCov, CI, documentation, and more).
- The Project.toml generated for a package has more information than for a standard local Project.toml file.

```
(@v1.8) pkg> generate HelloWorld
```

```
shell> tree HelloWorld/
HelloWorld/
├── Project.toml
└── src
    └── HelloWorld.jl
```

```
name = "HelloWorld"
uuid = "b4cd1eb8-1e24-11e8-3319-93036a3eb9f3"
version = "0.1.0"
authors = ["Some One <someone@email.com>"]

[deps]
```

The content of `src/HelloWorld.jl` is:

```
module HelloWorld

greet() = print("Hello World!")

end # module
```

Modules in Julia

- Modules are used to organize types, code, and create packages in Julia.
- They create a new namespace, and you can specify which variables are exported if you load a module.
- You can create submodules within a module, though these are not often necessary in Julia.

```julia
module ExampleModule

export ExampleStruct
export example_function

struct ExampleStruct{T}
    x::T
end

function example_function(x::ExampleStruct)
    return x
end

end
```

```
julia> include("module_example.jl")
Main.ExampleModule

julia> using .ExampleModule

julia> example_function(ExampleStruct(1))
Int64
```

Example: creating a Julia package

- Lets create a Julia package “CanonicalBasisVectors.jl”.
- Main steps:
  - Create a module to hold and export the structure CanonicalBasisVector, as well as as the function “inner”.
  - Create a Github repository to host the package.
- We’ll test to see if we can then then load the package from Github.
- Demo…

What can open source Julia packages do for you?

- While Plots.jl is technically not a base Julia package, plotting is pretty general.
- We'll cover some assorted scientific computing packages and topics in the last 1.5 weeks of class:
  - Useful array types (for speed or convenience)
  - Automatic differentiation (AD); this is IMO the key enabling tech of ML.
  - Tabular data (e.g., CSV and Excel files)
  - Ordinary differential equations (ODEs)
  - Linear programming and optimization

Before we start…BenchmarkTools.jl

- We've used "@btime" with BenchmarkTools.jl previously.
- You can see more detail with the "@benchmark" macro.
- An important detail: for accurate timings, we should <u>interpolate</u> function inputs when benchmarking.
  - Example: "@btime foo(x + 2 * y)" vs "@btime foo($(x + 2 * y))"
  - Avoids timing both intermediate computations and the cost of <u>accessing global variables</u>.

```
julia> using BenchmarkTools

# The `setup` expression is run once per sample, and is not included in the
# timing results. Note that each sample can require multiple evaluations
# benchmark kernel evaluations. See the BenchmarkTools manual for details.
julia> @benchmark sort(data) setup=(data=rand(10))
BenchmarkTools.Trial: 10000 samples with 972 evaluations.
 Range (min … max):  69.399 ns …  1.066 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     83.850 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   89.471 ns ± 53.666 ns  ┊ GC (mean ± σ):  3.25% ± 5.16%

  69.4 ns           Histogram: frequency by time            145 ns (top 1%)

 Memory estimate: 160 bytes, allocs estimate: 1.
```

An example of the @benchmark macro

Useful array types: StaticArrays.jl

- From https://github.com/JuliaArrays/StaticArrays.jl: “StaticArrays provides a framework for implementing statically sized arrays in Julia…”
- Basic premises:
  - Statically sized arrays are generally faster to access.
  - You can optimize several routines for smaller matrices (e.g., explicit formulas for determinants for 2x2 or 3x3 matrices).

```
=========================================
    Benchmarks for 3×3 Float64 matrices
=========================================
Matrix multiplication                -> 5.9x speedup
Matrix multiplication (mutating)     -> 1.8x speedup
Matrix addition                      -> 33.1x speedup
Matrix addition (mutating)           -> 2.5x speedup
Matrix determinant                   -> 112.9x speedup
Matrix inverse                       -> 67.8x speedup
Matrix symmetric eigendecomposition -> 25.0x speedup
Matrix Cholesky decomposition        -> 8.8x speedup
Matrix LU decomposition              -> 6.1x speedup
Matrix QR decomposition              -> 65.0x speedup
```

Useful array types: StaticArrays.jl, cont.

- Consider the following example:
  - Given 1 million points with (x, y) coordinates, apply a rotation matrix to transform each point.
- Option 1: store x and y coordinates in separate vectors, build StaticVector and StaticMatrix instances in the loop.
- Option 2: store a vector of StaticVector of points.
- Option 3: store a vector of Point2D using a FieldVector abstract type.

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

```julia
x = randn(1_000_000)
y = randn(1_000_000)
for i in eachindex(x)
    R = [cos(theta) -sin(theta);
         sin(theta) cos(theta)]
    rotated = R * [x[i], y[i]]
    x[i] = rotated[1]
    y[i] = rotated[2]
end
```

rotate_points.jl

Useful array types: StructArrays.jl

- Consider “Option 1: store x and y coordinates in their own separate vectors” from the previous example.
- What if we wanted the convenience of accessing a vector of Point2D, but want to keep access to “x” and “y”?
  - Can be useful if you need to perform operations on the vector (e.g., mean, sum, mat-vec, etc)
- Solution: use “StructArrays.jl” to lazily create instances of “Point2D” from entries of “x” and “y”

```julia
using StaticArrays
using StructArrays
struct Point2D{T} <: FieldVector{2, T}
    x::T
    y::T
end
xy = StructArray{Point2D}((x, y))
```

Exercise

- Run and time arrays_ex.jl
- Edit arrays_ex.jl to use StaticArrays
- Run and time arrays_ex.jl again, and compare the timings

- Change the size of the matrices to 20 and repeat. What do you observe?

Other useful array types

- FillArrays.jl: arrays with only one entry. I've found these useful for cases when an array is sometimes constant and sometimes not. I can implement the general non-constant case and then swap in a FillArray for the constant case.
- MappedArrays.jl: lazy evaluation of $f(A)$, where A is a matrix over which the function is evaluated entrywise.
- LinearOperators.jl: when it's easier to evaluate Ax instead of directly evaluating A. You can reconstruct an entire matrix without needing to explicitly define its entries.
- There are plenty more arrays out there, but note that there are usually operations for which these arrays have not been optimized. These arrays can be useful, but only use them if you need.

Automatic differentiation (AD).

- What is AD? An almost too-good-to-be-true way to compute derivatives of any computationally implemented function.
  - Similar in concept to the complex step method, but relies on “dual numbers” which are a symbolic generalization of imaginary numbers.
  - AD tends to be more convenient and actually cheaper than complex step.
- Works for *any function*, not just analytical expressions!

```julia
using ForwardDiff

f(x) = sin(pi * x)
df = ForwardDiff.derivative(f, .2)
```

AD_examples.jl

```julia
function herons_method(x0)
    @assert x0 > 0
    x = x0
    for _ in 1:25
        x = 0.5 * (x + x0 / x)
    end
    return x
end

x = 0.1
@show ForwardDiff.derivative(herons_method, x)
```

Automatic differentiation, cont.

- Can create functions which represent function derivatives using ForwardDiff.jl
- These behave like any other function in Julia; you can broadcast, pass them as arguments, analyze type stability, etc.

```
julia> @code_warntype deriv_heron(.1)
MethodInstance for deriv_heron(::Float64)
  from deriv_heron(x) @ Main Untitled-1:12
Arguments
  #self#::Core.Const(deriv_heron)
  x::Float64
Body::Float64
1 ─ %1 = ForwardDiff.derivative::Core.Const(ForwardDiff.derivative)
│   %2 = Main.heron::Core.Const(heron)
│   %3 = (%1)(%2, x)::Float64
└──      return %3
```

```julia
using ForwardDiff
using Plots

function heron(x0)
    @assert x0 > 0
    x = x0
    for _ in 1:2
        x = 0.5 * (x + x0 / x)
    end
    return x
end

deriv_heron(x) =
    ForwardDiff.derivative(heron, x)

x = LinRange(.1, 1, 100)
plot!(x, deriv_heron.(x))
```

Higher, partial, and multi-dimensional derivatives

- AD computes arbitrarily high derivatives accurately and stably.
- Partial derivatives can be computed by using “closures”
  - Closure: an anonymous function which encloses an external variable into the function’s scope.
- Note: closures can be slow in Julia, but <u>they can usually be made efficient</u>.
- If you have a vector-valued input or output, you can compute the gradient or Jacobian of the function.

```julia
function newton_step(f, x0)
    J = ForwardDiff.jacobian(f, x0)
    δ = J \ f(x0)

    return x0 - δ
end

function newton(f, x0)
    x = x0

    for i in 1:10
        x = newton_step(f, x)
        @show x
    end

    return x
end
```

What about efficiency?

- Surprisingly, AD usually takes about the same amount of time that it takes to evaluate a function.
- Can be used within hot loops without significant overhead (at least when using JAX in Python or Julia).
- Caveat: for higher dimensional functions (e.g., many inputs or many outputs) AD becomes more expensive.
  - The cost scales with the dimension; this can be reduced using specific AD algorithms (e.g., “reverse mode” AD).

```julia
using BenchmarkTools
@btime heron($.1)
@btime deriv_heron($.1)
```

```
3.000 ns (0 allocations: 0 bytes)
4.208 ns (0 allocations: 0 bytes)
```

```julia
f(x) = sum(x)
@btime f($(randn(10)))
@btime ForwardDiff.gradient($f, $(randn(10)))
```

```
4.958 ns (0 allocations: 0 bytes)
346.259 ns (3 allocations: 1.92 KiB)
```

Exercise

- Compute the critical points ($\frac{df}{dx} = 0$) and inflection points ($\frac{d^2f}{dx^2} = 0$) of the given function using automatic differentiation and the provided Bisection function

* Make the function accept vector input, hint: x^2 -> x'*x

* Compute the gradient and jacobian

Tabular data: DataFrames.jl

- Similar to the “pandas” Python library for handling tables and time-series.
- Helpful for manipulating data from CSV (comma-separated value) files (e.g., spreadsheets).
  - CSV.jl is a separate library for reading CSV files into Julia data types such as a DataFrame.
- Can slice, join, sub-select, compute statistics, etc.

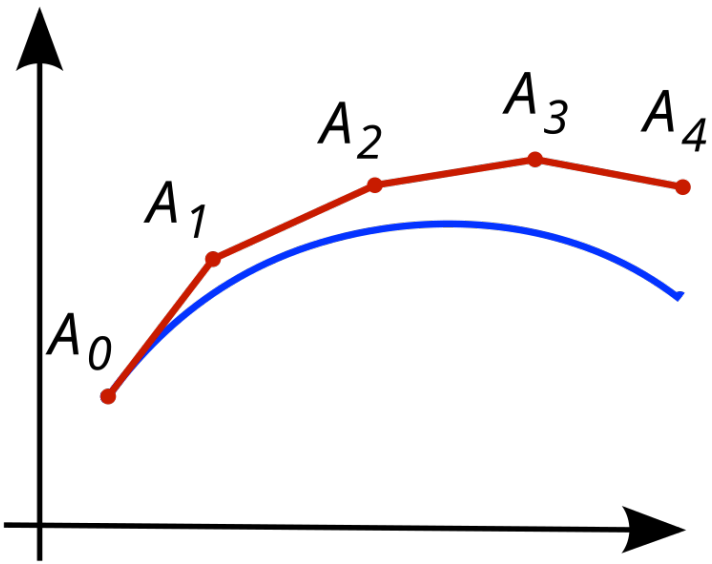
```julia
using CSV
using DataFrames

path = joinpath(pkgdir(DataFrames), "docs",
                "src", "assets", "german.csv");
german = CSV.read(path, DataFrame)
```

```
julia> german
1000×10 DataFrame
```

Row	id	Age	Sex	Job	Housing
	Int64	Int64	String7	Int64	String7
1	0	67	male	2	own
2	1	22	female	2	own
3	2	49	male	1	own
4	3	45	male	2	free
5	4	53	male	2	free
6	5	35	male	1	free
7	6	53	male	2	own
8	7	35	male	3	rent
⋮	⋮	⋮	⋮	⋮	⋮
993	992	23	male	1	rent
994	993	30	male	3	own
995	994	50	male	2	own
996	995	31	female	1	own
997	996	40	male	3	own
998	997	38	male	2	own
999	998	23	male	2	free
1000	999	27	male	2	own

Ordinary differential equations (ODEs)

- "Rate of change" phenomena are typically modeled as systems of ODEs.
  - Examples: infectious disease modeling, predatory-prey population models, mechanical oscillators, neuron models, simplified climate models, etc…
  - Time-dependent partial differential equations (PDEs) are turned into very large systems of ODEs.
- Most ordinary differential equations don't have an explicit solution, so we use time-stepping (Explicit Euler, Runge-Kutta, etc)

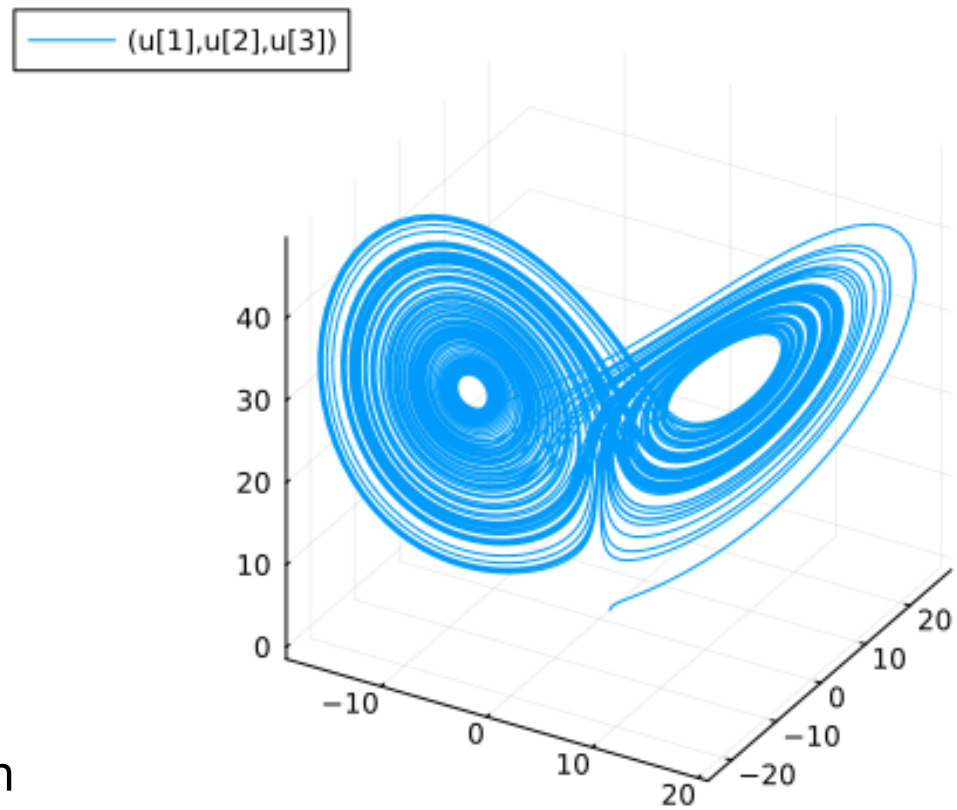


$$\frac{dx}{dt} = f(x, t)$$

$$x_{k+1} = x_k + \Delta t f(x_k, t_k)$$

Why use an ODE package?

- Take advantage of efficient but cumbersome implementations.
- Time-adaptivity is great for a lot of problems, but the algorithms tend to be more technical.
- Convenience: lots of functions are inconvenient to implement:
  - Saving intermediate solutions at arbitrary times.
  - Activating behavior at specific times which may not line up with a uniform time-stepping grid.

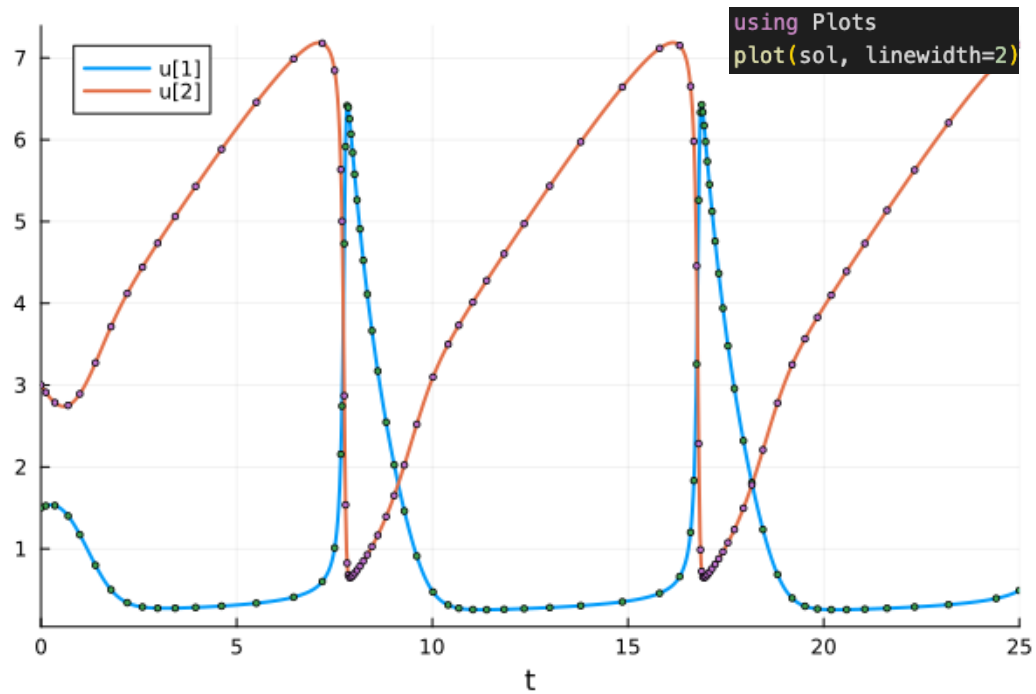


OrdinaryDiffEq.jl

- Part of the SciML and “DifferentialEquations.jl” ecosystem, which includes neural ODEs, stochastic ODEs, uncertainty quantification, etc…

$$\frac{du}{dt} = f(x, t)$$

```julia
using OrdinaryDiffEq # or DifferentialEquations

function brusselator(du, u, p, t)
    B = 4
    y1, y2 = u
    du[1] = 1 + y1^2 * y2 - (B+1) * y1
    du[2] = B * y1 - y1^2 * y2
end
u0 = [1.5, 3.0]
tspan = (0.0, 25.0)
prob = ODEProblem(brusselator, u0, tspan)
sol = solve(prob, Tsit5())
```



Adaptive time-stepping

- Handles <u>adaptive time-stepping</u> automatically.
  - Adaptivity is easy conceptually but tends to complicate implementation significantly.
  - Good parameter choices for adaptive time-stepping vary depending on the method as well.
- When applicable, takes advantage of *embedded* time-stepping methods which make it cheaper to compute adaptive step sizes.

Algorithm 2 High-level overview of a time integration loop using explicit RK methods with embedded error estimator and PID controllers in OrdinaryDiffEq.jl and deviations in PETSc/TS

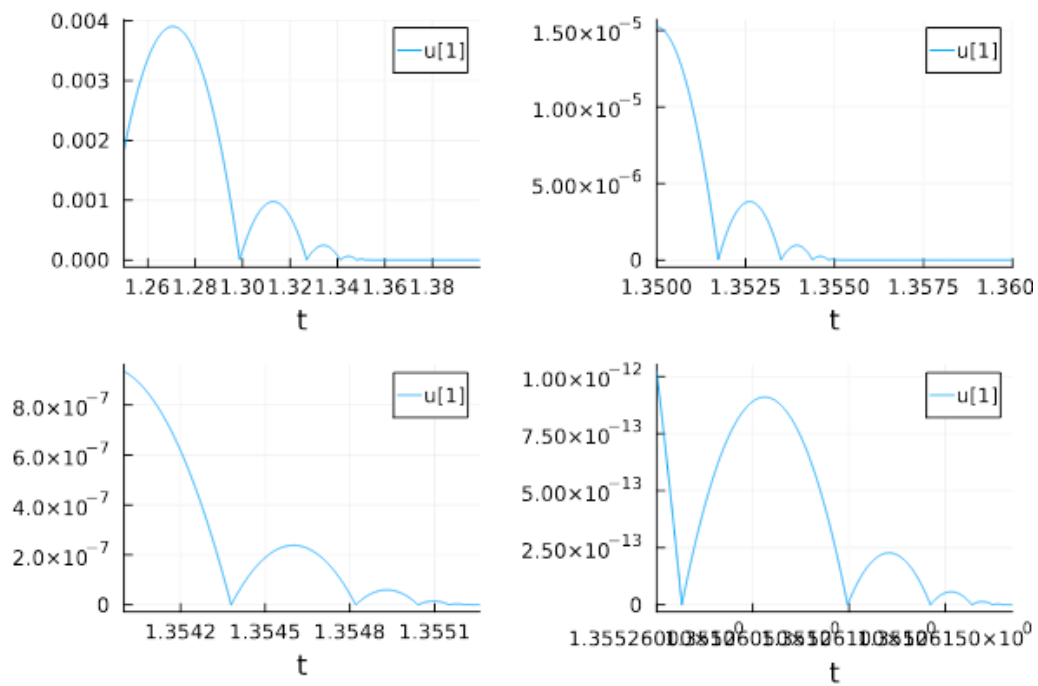
Initialize time $t \leftarrow 0$, time step number $n \leftarrow 0$, and initial state u^0.
Initialize PID controller with $\varepsilon_0 \leftarrow 1$, $\varepsilon_{-1} \leftarrow 1$.
Initialize $\Delta t_0 = \widetilde{\Delta t}$ with a given value or the algorithm of [38, p. 169].
Initialize `accept_step` $\leftarrow$ `false`
while $t < T$ **do**
  if `accept_step` **then** ▷ callbacks can change `accept_step`
    `accept_step` $\leftarrow$ `false` ▷ prepare for the next step
    $t \leftarrow \widetilde{t}$
    $\Delta t_{n+1} \leftarrow \widetilde{\Delta t}$
    $n \leftarrow n + 1$
  else
    $\Delta t_n \leftarrow \widetilde{\Delta t}$
  end if
  if $t + \Delta t_n > T$ **then**
    $\Delta t_n \leftarrow T - t$
  end if
  Compute u^{n+1} and $\widehat{u}^{n+1}$ with time step Δt_n
  $w_{n+1} \leftarrow \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\frac{u_i^{n+1}-\widehat{u}_i^{n+1}}{\tau_a+\tau_r \max\left\{\left|u_i^{n+1}\right|,\left|\widetilde{u}_i\right|\right\}}\right)^2}$
  ▷ $\widetilde{u} = u^n$ in OrdinaryDiffEq.jl, $\widetilde{u} = \widehat{u}^{n+1}$ in PETSc
  $w_{n+1} \leftarrow \max\{w_{n+1}, w_{\min}\}$
  ▷ $w_{\min} \approx 2.2 \times 10^{-16}$ in OrdinaryDiffEq.jl, $= 1.0 \times 10^{-10}$ in PETSc
  $\varepsilon_{n+1} \leftarrow 1/w_{n+1}$
  `dt_factor` $\leftarrow \kappa\left(\varepsilon_{n+1}^{\beta_1/k}\varepsilon_n^{\beta_2/k}\varepsilon_{n-1}^{\beta_3/k}\right)$ ▷ default $\kappa(a) = 1 + \arctan(a-1)$
  $\widetilde{\Delta t} \leftarrow$ `dt_factor` $\cdot \Delta t_n$
  if `dt_factor` $\geqslant$ `accept_safety` **then** ▷ default `accept_safety` $= 0.81$
    `accept_step` $\leftarrow$ `true`
  else
    `accept_step` $\leftarrow$ `false`
  end if
  if `accept_step` **then**
    $\widetilde{t} \leftarrow t + \Delta t_n$
    if $\widetilde{t} \approx T$ **then** ▷ within 100 units in last place in OrdinaryDiffEq.jl
      $\widetilde{t} \leftarrow T$
    end if
    Apply callbacks ▷ AMR, CFL-based control in OrdinaryDiffEq.jl
  end if
end while

Solution interpolation and callbacks

- Handles solution interpolation, which is non-trivial to implement since the accuracy of the interpolation depends on the time-stepper!
  - Can use “saveat” or just “sol(t)” to evaluate the solution at arbitrary times.
- Includes at least 300 unique named time-stepping methods, over 1000 solvers (explicit, implicit, multi-step, stabilized, stochastic, etc)
- Can insert event data or extract observable data using *callbacks*. Algorithms for handling event data are quite nuanced!
  - Events are triggered if a condition (which is a function of the solution) is met. OrdinaryDiffEq.jl solves a root finding problem (using bisection, Newton’s method, etc) to determine when *exactly* the condition is met so that there is a time-step precisely on this condition.

Event handling example: Zeno's paradox

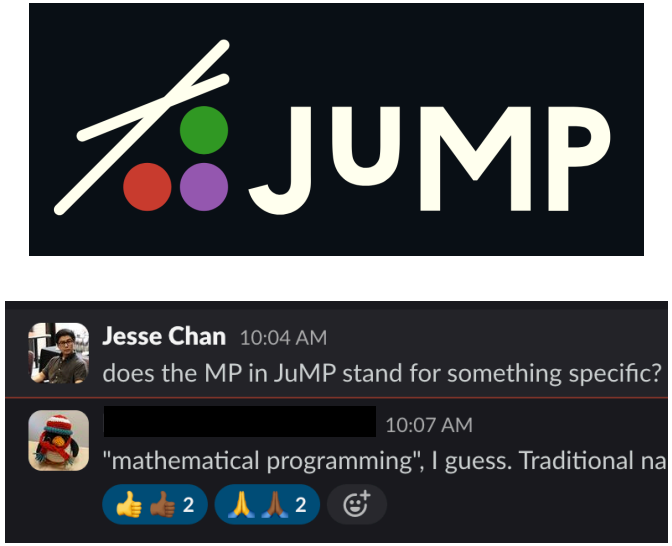
- The ODE is the equation for acceleration under gravity $x''(t) = -9.8x(t)$.
- If the height of a ball goes below zero, the ball bounces.
- A bounce is assumed to set the velocity to -1/2 * (original velocity).



https://docs.sciml.ai/DiffEqDocs/stable/features/callback_functions/

Optimization and modeling

- JuMP.jl is an "algebraic modeling language for mathematical optimization” (good to know if you’re an operations research (OR) major!)
- Like Plots.jl, JuMP interfaces with several different solver *backends*, but focuses on providing an *algebraic modeling language*.
  - Modeling language: simplifies complex problem formulations, provide a unified solver interface, enables pre-solve simplification and reformulation.
- Focused on differentiable optimization with constraints; can handle continuous, integer and binary variable types.



Note: I am very much not an expert in this area.

Some examples

- Linear programs in standard form; can also use individual variable syntax instead of linear algebra syntax if you prefer.
- Knapsack example: easy way to specify which variables are constrained to be {0, 1}

```
@variable(model, x[1:n], Bin)
```

- More complex examples, e.g., quadratic constraints, bilevel/nested optimization

$$
\begin{aligned}
\min_{x,z} \quad & x_1^2 + x_2^2 + z \\
s.t. \quad & z = \max_{y} \quad x_1^2 y_1 + x_2^2 y_2 - x_1 y_1^4 - 2 x_2 y_2^4 \\
& \qquad s.t. \quad (y_1 - 10)^2 + (y_2 - 10)^2 \leq 25 \\
& x \geq 0.
\end{aligned}
$$

```
using JuMP
using HiGHS, Ipopt

# max c' * x
# s.t. A * x = b, x ≥ 0
model = Model(HiGHS.Optimizer)
# model = Model(Ipopt.Optimizer)
A = [1 1 9 5; 3 5 0 8; 2 0 6 13]
b = [7, 3, 5]
c = [1, 3, 5, 2]

@variable(model, x[1:4] >= 0)
@constraint(model, A * x .== b)
@objective(model, Min, c' * x)
print(model)
```

```
Min x[1] + 3 x[2] + 5 x[3] + 2 x[4]
Subject to
 x[1] + x[2] + 9 x[3] + 5 x[4] = 7
 3 x[1] + 5 x[2] + 8 x[4] = 3
 2 x[1] + 6 x[3] + 13 x[4] = 5
 x[1] ≥ 0
 x[2] ≥ 0
 x[3] ≥ 0
 x[4] ≥ 0
```

JuMP_examples.jl