Basic Julia

David Evans

2026-11-13

Why Julia?

Why Julia?

- Julia is a high-level, high-performance language for technical computing
- Combines the **ease-of-use** of Python/R with the **speed** of C/Fortran
- Increasingly popular in economics, statistics, and computational science

> **The "Two Language Problem"**
>
> In many fields, researchers prototype in a slow language (Python, R, Matlab) then rewrite in a fast language (C, Fortran) for production. Julia solves this: one language for both.

What We'll Cover

1. **Primitive data types** — booleans, integers, floats, and strings
2. **Container types** — arrays, tuples, named tuples, and dictionaries
3. **Control flow** — `for` loops, comparisons, logical operators, `if` statements
4. **Functions** — defining reusable code with inputs and outputs
5. **Root solving** — finding CES market equilibria with the course's `EC410` package
6. **Optimization** — CES utility and profit maximization with Optim

By the end of this lecture you should be comfortable writing simple Julia programs that store data, loop, make decisions, organize code into functions, and use packages for root solving and optimization.

Primitive Data Types

Booleans

Booleans

- There are several basic types of variables in Julia, the most common are: boolean, integer, float, complex number.

- Booleans are very simple variables that can be in only one of 2 states: True and False

```
1 x = true #Boolean Type
2 y = 100 < 10 #evaluates the right hand side.  Assigns output to y
```

```
false
```

```
1 typeof(y) #Bool means Boolean
```

```
Bool
```

Integers

Integers

- Integer variables are variables x which can be any number $x = -\infty, \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots, \infty$
  - For example 0.5 and π are not integers

```
1 x = 3
2 y = 5
3 x+y
```

```
8
```

```
1 typeof(x) #Int64 means integer
```

```
Int64
```

Integer Division

- Default produces natural results

```
1 x/y
```

```
0.6
```

- Can also compute the remainder of integer division using %

```
1 5%3
```

```
2
```

```
1 51%17 #51 is a multiple of 17
```

```
0
```

Floating Point Numbers

Floats

- Real numbers are represented with floats in Julia

```
1 Pi = 3.14159265358979323846264
2 typeof(Pi)
```

```
Float64
```

- `pi` is also a prestored constant

```
1 pi
```

```
π = 3.1415926535897...
```

Float Arithmetic

- Multiplication and division work in the standard way

```
1 pi * 4
```

```
12.566370614359172
```

```
1 pi/3
```

```
1.0471975511965976
```

- Can use `a^b` to compute a^b

```
1 2^4
```

```
16
```

Ensuring Float Type

If you want to make sure a variable is a float, add a decimal point to the end

```
1 x = 3 #integer
2 y = 3. #float
```

```
3.0
```

Strings

Strings

- A **string** is a sequence of characters, enclosed in double quotes
- Single quotes are for individual characters (`Char`)

```
1 greeting = "Hello, world!"
2 typeof(greeting)
```

```
String
```

```
1 c = 'A'        # This is a Char, not a String
2 typeof(c)
```

```
Char
```

String Operations

- Concatenate strings with `*` (not `+`!)

```
first_name = "David"
last_name = "Evans"
first_name * " " * last_name
```

```
"David Evans"
```

- Repeat strings with `^`

```
"ha" ^ 3
```

```
"hahaha"
```

String Interpolation

- Embed variables in strings using $

```
name = "David"
age = 39
"My name is $name and I am $age years old."
```

```
"My name is David and I am 39 years old."
```

- Use parentheses for expressions

```
x = 10
"The square of $x is $(x^2)."
```

```
"The square of 10 is 100."
```

Useful String Functions

```
1 s = "Economics is great"
2 length(s)
```

```
18
```

```
1 uppercase(s)
```

```
"ECONOMICS IS GREAT"
```

```
1 occursin("great", s)    # Is substring present?
```

```
true
```

```
1 replace(s, "great" => "fascinating")
```

```
"Economics is fascinating"
```

```
1 split(s)    # Split on whitespace
```

```
3-element Vector{SubString{String}}:
 "Economics"
 "is"
 "great"
```

Converting Strings and Numbers

- `parse` converts a string to a number
- `string` converts a number to a string

```
1 parse(Int64, "42")
```

```
42
```

```
1 parse(Float64, "3.14")
```

```
3.14
```

```
1 string(2.718)
```

```
"2.718"
```

Container Types

Overview

Container Types Overview

Arrays `[]`

- Mutable, ordered by index
- `[1, 2, 3]`

Tuples `()`

- Immutable, ordered by index
- `(1, 2, 3)`

Named Tuples

- Immutable, access by name
- `(a=1, b=2)`

Dictionaries

- Mutable, named keys
- `Dict("a" => 1)`

Arrays

Arrays

- Sometimes a variable is just a number
- Other times we want more information associated with that variable
- In Julia it is possible to combine many elements of different types into an array

```
1 David = [40,"male",71.5] #square brackets indicate array
2 typeof(David)
```

```
Vector{Any} (alias for Array{Any, 1})
```

Note: `Any` means this array can hold any type of object.

Typed Arrays

We can also make arrays which are of only one type

```
1 Davidfloat = [32.1,16.,35]
2 typeof(Davidfloat)
```

```
Vector{Float64} (alias for Array{Float64, 1})
```

`Davidfloat` only holds floating point numbers. Julia is generally more efficient if it knows what type of objects variables are.

```
1 Davidfloat[2] = "test" # This will error!
```

```
MethodError: Cannot `convert` an object of type String to an object of type Float64
The function `convert` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  convert(::Type{T}, ::T) where T<:Number
   @ Base number.jl:6
  convert(::Type{T}, ::Number) where T<:Number
   @ Base number.jl:7
  convert(::Type{T}, ::T) where T
   @ Base Base_compiler.jl:136
  ...

Stacktrace:
 [1] setindex!(A::Vector{Float64}, x::String, i::Int64)
   @ Base ./array.jl:1019
 [2] top-level scope
   @ ~/work/EC410-Source/EC410-Source/Basic Julia/JuliaBasicsLecture.qmd:321
```

Accessing and Modifying Arrays

- You can access and change elements of a list using the `[]` operator

```
1 David[2]
```

```
"male"
```

- You can change elements of the list and append elements to it

```
1 David[3] = David[3]/12
2 push!(David,12) #push! adds an element to the end of an Array
3 # ! the exclamation point is because push changes David (adds to it)
4 David
```

```
4-element Vector{Any}:
 40
   "male"
  5.958333333333333
 12
```

Element Access and Indexes

Julia is indexing starts at 1!

Unlike Python (which starts at 0), Julia arrays start at index 1. This is a common source of bugs when switching languages.

- `L[1]` gives us the first element of array L
- `L[i]` gives us access to the element at index i of array `L`
- Element access also works in reverse: `L[end]` gives access to the last element, `L[end-1]` the second to last, etc.

Array Slicing

- You can access a slice of the list using `:`
- `L[1:4]` returns the list `[L[1],L[2],L[3],L[4]]`

```
1 print( David[1:3] ) #Could have also used David[1:end-1]
```

```
Any[40, "male", 5.958333333333333]
```

Tuples

Tuples

- Tuples are a list of elements (which are not necessarily of the same type) which cannot be changed

```
1 d = (39,"male",71.5) #use round brackets to denote tuple
2 d = 39,"male",71.5 #or no brackets at all
3 print( typeof(d) ) #note Julia knows what types these elements are
```

```
Tuple{Int64, String, Float64}
```

- One property of tuples is that they are **immutable**, meaning they can't be changed

```
1 d[1] = 5  # This will error!
```

```
MethodError: no method matching setindex!(::Tuple{Int64, String, Float64}, ::Int64, ::Int64)
The function `setindex!` exists, but no method is defined for this combination of argument types.
Stacktrace:
 [1] top-level scope
   @ ~/work/EC410-Source/EC410-Source/Basic Julia/JuliaBasicsLecture.qmd:385
```

Unpacking Tuples

- We can unpack tuples in the following way

```
1 integers = (4,5,6)
2 x,y,z = integers
3 x
```

```
4
```

```
1 z
```

```
6
```

This can also be done with arrays

```
1 x,y,z = David #Stores first 3 elements of David in variables x,y,z
2 x #first element of David
```

```
40
```

Named Tuples

- Named tuples let you access elements by **name** instead of just by index
- Combines the immutability of tuples with additional readability

```
person = (name="David", age=40, height=71.5)
typeof(person)
```

```
@NamedTuple{name::String, age::Int64, height::Float64}
```

```
person.name  # Access by name (preferred)
```

```
"David"
```

```
person[1]    # Can still access by index
```

```
"David"
```

> **When to Use Named Tuples**
>
> Use named tuples when you want immutable data with meaningful field names—–great for returning multiple values from functions!

Dictionaries

Dictionaries

- Dictionaries are like lists except that items are **named** instead of numbered

```
1 David = Dict("age" => 39, "gender" => "male", "height"=> 71.5)
2 print( typeof(David) )
```

```
Dict{String, Any}
```

```
1 David["age"]
```

```
39
```

Modifying Dictionaries

- Can also add to dictionary (unlike named tuples)

```
1 David["job"] = "Economist"
2 David["job"]
```

```
"Economist"
```

- `"age"`, `"gender"` and `"height"` are the **keys**
- The dictionary `David` maps them into the **values** 39, “male”, and 71.5
- Useful when dealing with many elements and don’t want to figure out what position a variable is in

Iterating

For Loops

For Loops

- The most basic loop in Julia is the for loop. The basic syntax is:

```
for x in iterable
    code
end
```

which means for each element in an iterable (array, tuple, etc.) repeat some lines of code, which can depend on the value of `x`

For Loop Examples

```
#1:5 is an iterable starting with 1 and ending in 5
for i in 1:5
    println( i ) #will print i for i running from 1 to 5
end
```

```
1
2
3
4
5
```

```
#This piece of code prints i^2 from 1:5
for i in 1:5
    x = i^2
    println( x )
end
```

```
1
4
9
16
25
```

Unrolling a For Loop

What does the loop actually do? It repeats the body once for each value of `i`:

```
# This for loop...
for i in 1:5
    x = i^2
    println(x)
end
```

```
1
4
9
16
25
```

is equivalent to:




```
x = 1^2; println(x)
x = 2^2; println(x)
x = 3^2; println(x)
x = 4^2; println(x)
x = 5^2; println(x)
```




```
1
4
9
16
25
```

The loop just saves us from writing the same code over and over!

Example: Printing Elements

```
1 David = [32,"male",71.5]; #the semicolon suppresses output
```

- How do you print out all the elements of `David`?

```
1 for element in David
2     println( element )
3 end
```

```
32
male
71.5
```

Looping Without Indices

- As we see, Julia by default loops through elements of a list

```
x_values = [4,2,3]
for x in x_values
    println( x * x )
end
```

```
16
4
9
```

- Can also do this, but it is not as easy to read

```
for i in 1:length(x_values)
    println( x_values[i]*x_values[i] )
end
```

```
16
4
9
```

Enumerate

> **Best Practice**
>
> Use `enumerate` when you need both the index and the value. It's cleaner than managing a counter variable manually.

```
letter_list = ['a', 'b', 'c']
for (i, letter) in enumerate(letter_list)
    println("Element $i of letter_list is $letter") #string interpolation
end
```

```
Element 1 of letter_list is a
Element 2 of letter_list is b
Element 3 of letter_list is c
```

Example: Inner Product

- How could I take the inner-product of two arrays $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_n]$?

```
x = [3,57,2,1,8,5]
y = [4.3,23,2,47,42,6]
ret = 0.
for i in 1:length(x)
    ret = ret + x[i]*y[i] #alternatively ret += x[i]*y[i]
end
ret
```

```
1740.9
```

Example: Geometric Sum

How would you compute the sum:

$$\sum_{j=0}^{50} \beta^j$$

```
β = 0.95 #can use Greek letters: type \beta then hit `tab`
ret = 0.
for j in 0:50
    ret += β^j
end
print( ret )
```

ret = ret + β^j

```
18.538045469742432
```

Julia Tip

You can use Greek letters in Julia! Type `\beta` then press `Tab` to get β.

Comparisons and Logical Operators

Comparisons

Comparisons

- Many different types of expressions evaluate to one of the Boolean values
- A common type is comparisons, such as

```
1 x,y = 1,2
2 x < y
```

```
true
```

```
1 x > y
```

```
false
```

Chaining Inequalities

- We can chain inequalities

```
1 1 < 2 < 3
```

```
true
```

```
1 1 <= 2 <= 3 > 4
```

```
false
```

Testing Equality

> ⚠ **Common Mistake**
>
> Don't confuse `=` (assignment) with `==` (equality test)!

```
1 x = 1 #assignment
2 x == 2
```

```
false
```

```
1 x != 2
```

```
true
```

Logical AND

- We can combine expressions using && for “and”

```
1 1 < 2 && 'f' in "foo" #true: 1 < 2 AND 'f' is in "foo"
```

```
true
```

```
1 1 < 2 && 'g' in "foo" #false: 1 < 2 but 'g' is NOT in "foo"
```

```
false
```

Logical OR

- We can combine expressions using `||` for “or”

```
1 1 < 2 || 'g' in "foo" #true: 1 < 2 (don't need both for OR)
```

```
true
```

Logical NOT

- We can negate expressions using `!` for “not”

```
1 !true
```

```
false
```

```
1 !!true
```

```
true
```

Logical Operators Summary

Expression	Result
`P && Q`	`True` if **both** are `True`, otherwise `False`
`P || Q`	`False` if **both** are `False`, otherwise `True`
`!P`	Negates `P`

If Statements

If Statement

- “If” is a special way of evaluating code only if a condition is met

```
if condition
    code
end
```

```
x = 5
if x > 4
    println( "Yay!" )
end
```

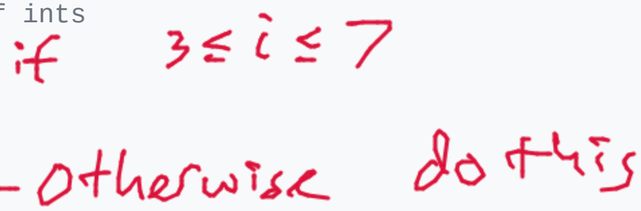
```
Yay!
```

```
if x < 4
    print( "We did not reach this code, it never ran, how existential" )
end
```

If-Else

- We can use `else` to evaluate code if the condition is not met

```
x = Int[] #x is an array of ints
for i in 1:10
    if 3<=i<=7
        push!(x,i^2)
    else
        push!(x,i)
    end
end
```



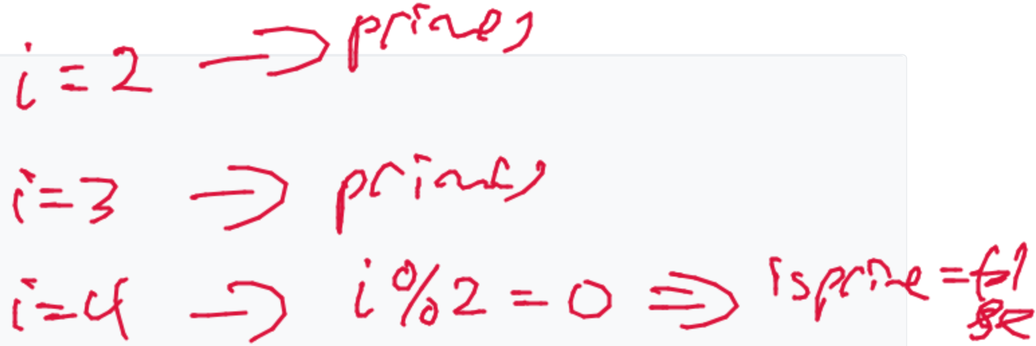

```
println(x)
```

```
[1, 2, 9, 16, 25, 36, 49, 8, 9, 10]
```

Example: Finding Primes

Find all the primes from 2 to 500

```
primes = Int[]                  # 1: Initialize empty array of integers
for i in 2:500                  # 2: Loop through candidates 2 to 500
    isprime = true              # 3: Assume i is prime until proven otherwise
    for p in primes             # 4: Check against all known primes
        if i%p == 0             # 5: Does p divide evenly into i?
            isprime = false     # 6: Yes, so i is not prime
        end                     # 7: End if
    end                         # 8: End inner loop
    if isprime                  # 9: If no prime divided i...
        push!(primes, i)        # 10: ...add i to our list of primes
    end                         # 11: End if
end                             # 12: End outer loop
```



Finding Primes: Result

```
 1 primes = Int[]
 2 for i in 2:500
 3     isprime = true
 4     for p in primes
 5         if i%p == 0
 6             isprime = false
 7         end
 8     end
 9     if isprime
10         push!(primes, i)
11     end
12 end
13 print( primes )
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29  …  443, 449, 457, 461, 463, 467, 479, 487, 491, 499]
```

Functions

Basics

What Are Functions?

- A **function** is a reusable piece of code that takes inputs and produces outputs
- For example: $f(x) = x^2$ takes input x and returns x^2

- We define functions to:
  - Avoid repeating code
  - Break complex tasks into simple, testable pieces
  - Make code easier to read and maintain

Why Functions Matter

- Functions in Julia are very flexible:
  - Any number of functions can be defined in a given file
  - Any object can be passed as an argument, including other functions
  - Functions can be defined inside other functions
  - A function can return any kind of object, including functions

Anonymous (Lambda) Functions

- The simplest way to define a function is the **arrow syntax**

```
1 f = x -> x^2
2 f(2)
```

```
4
```

```
1 f(3)
```

```
9
```

- Useful for short, one-off operations (e.g., passing to `map` or `sort`)

The Short-Form Syntax

- For simple one-line functions with a name, use the **assignment form**

```
1 g(x) = x^2
2 g(5)
```

```
25
```

- This is identical to the arrow syntax but gives the function a name directly

The `function` Keyword

- For multi-line functions, use the `function` keyword

```
function h(x)
    return x^2
end
h(pi)
```

```
9.869604401089358
```

- The `return` keyword specifies what the function gives back
- If omitted, Julia returns the value of the last expression

Default Arguments and Docstrings

Default Arguments

- You can give arguments **default values** so they don't have to be specified every time

```
1 function greet(name, greeting="Hello")
2     println("$greeting, $(name)!")
3 end
4 greet("David")
```

```
Hello, David!
```

```
1 greet("David", "Howdy")
```

```
Howdy, David!
```

Multiple Return Values

- Functions can return multiple values as a tuple

```
function divide(a, b)
    quotient = a ÷ b    # integer division
    remainder = a % b
    return quotient, remainder
end
q, r = divide(17, 5)
println("17 ÷ 5 = $q remainder $r")
```

```
17 ÷ 5 = 3 remainder 2
```

Docstrings

- You can document functions with a **docstring** (a string before the function definition)

```
"""
    square(x)

Return the square of `x`.
"""
function square(x)
    return x^2
end
```

```
square (generic function with 1 method)
```

- Access documentation with `?square` in the REPL

Application: Simulating an AR(1) Process

The AR(1) Process

- A key example from time series econometrics:

$$x_t = \rho x_{t-1} + \epsilon_t \quad \text{where} \quad \epsilon_t \sim N(0,1)$$

- Julia's `randn()` draws from the standard normal distribution

```
1 println(randn())
2 println(randn())
```

```
-0.7900737315665072
-0.5367340248554923
```

A Single Step

- If $x_{t-1} = 5$ and $\rho = 0.8$, one step looks like:

```
1 x = 5.0
2 ρ = 0.8
3 print(ρ * x + randn())
```

```
2.498776492070274
```

Multiple Periods with a For Loop

- Simulate $T = 10$ periods starting from $x_1 = 5$ with $\rho = 0.8$:

```
 1 T = 10
 2 x = zeros(T) # pre-allocate array of zeros
 3 x[1] = 5.0  # set initial condition
 4 ρ = 0.8
 5
 6 # simulate using a for loop
 7 for t in 2:T
 8     x[t] = ρ * x[t-1] + randn() #update using previous period's value
 9 end
10
11 @show x;
```

```
x = [5.0, 3.375023523586238, 0.8046191208040605, 0.8940126106929971, -0.4039020791001946, -1.2912781809989349, -0.787686853143605,
-0.19551540298772158, -0.5365316957129509, -1.2411202852661143]
```

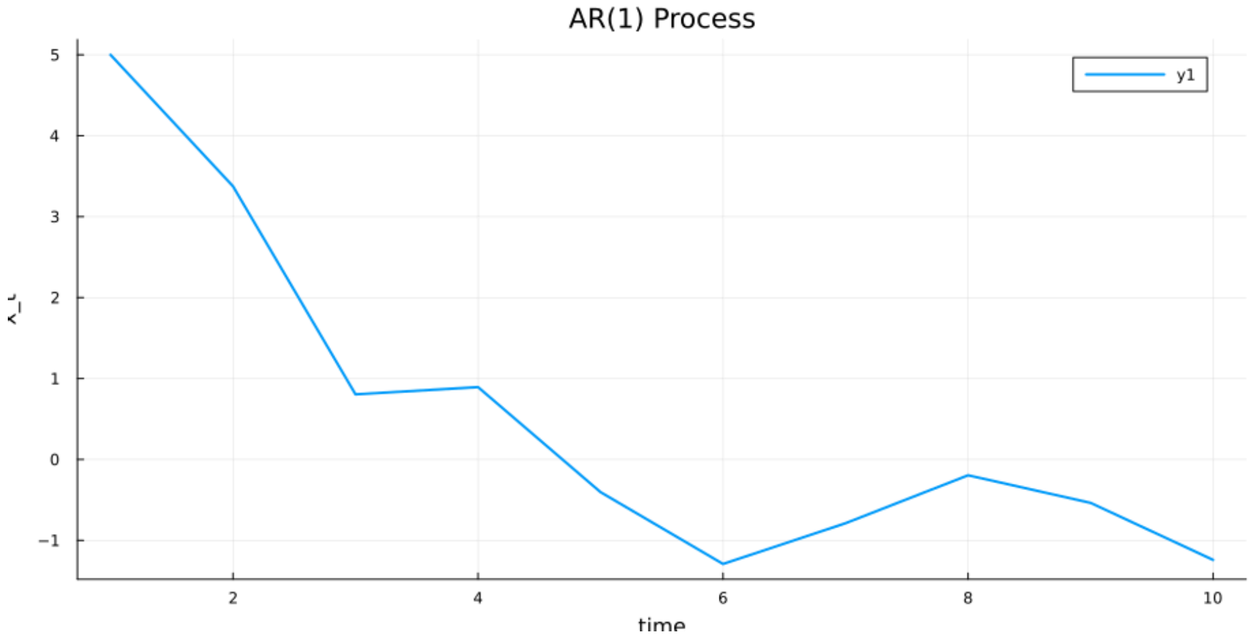
- Each iteration updates $x_t = \rho\, x_{t-1} + \varepsilon_t$
- The series **mean-reverts** toward zero as $|\rho| < 1$

Plotting the Results

```
1 using Plots
```

- Can use Plots.jl to plot the path of x

```
1 plot(x, linewidth=2)
2 xlabel!("time")
3 ylabel!("x_t")
4 title!("AR(1) Process")
```



- xlabel! and ylabel! are used to label the axes
- title! is used to set the title

Wrapping Up into a Function

```
"""
    simulateAR1(x0, T, ρ=0.8)

Simulate `T` periods of the AR(1) process xₜ = ρ*xₜ₋₁ + εₜ.
"""
function simulateAR1(x0, T, ρ=0.8)
   #pre-allocate array using T
   #set initial condition
   #for loop to update each period using previous period's value
   #return the array
end
```

Wrapping Up into a Function

```
"""
    simulateAR1(x0, T, ρ=0.8)

Simulate `T` periods of the AR(1) process xₜ = ρ*xₜ₋₁ + εₜ.
"""
function simulateAR1(x0, T, ρ=0.8)
    x = zeros(T) #pre-allocate array using T
    x[1] = x0 #set initial condition

    for t in 2:T #for loop to update
        x[t] = ρ * x[t-1] + randn()
    end
    return x #return the array
end
```

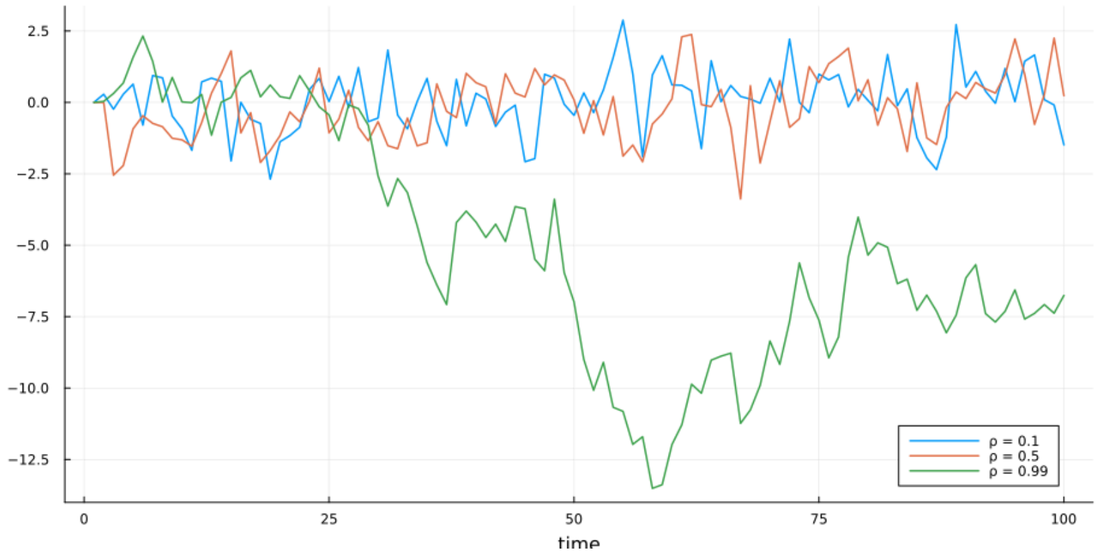
```
simulateAR1 (generic function with 2 methods)
```

- Uses a default argument for ρ
- Pre-allocates the array with `zeros(T)` for efficiency

Using the Function

- Can use Plots.jl to plot the results for different values of ρ

```
plot(simulateAR1(0., 100, 0.1), linewidth=1.5, label="ρ = 0.1")
plot!(simulateAR1(0., 100, 0.5), linewidth=1.5, label="ρ = 0.5")
plot!(simulateAR1(0., 100, 0.99), linewidth=1.5, label="ρ = 0.99")
xlabel!("time")
```



- `plot!` is used to add the second and third lines to the plot
- Higher ρ means more **persistence** — shocks die out more slowly

Numerical Methods

Root Solving

Why Root Solving?

- Many problems in economics reduce to finding x^* such that $f(x^*) = 0$
- Examples:
  - **Market equilibrium**: find price where excess demand equals zero
  - **Euler equations**: find optimal consumption satisfying first-order conditions
  - **Fixed points**: find x^* where $g(x^*) - x^* = 0$
- The course package `EC410` provides simple root solvers (it calls `NonlinearSolve.jl` under the hood)

The EC410 Root Solvers

```
1 using EC410
```

- `solve1D(f, a, b)` — root of a scalar function `f` between `a` and `b` (a **bracket**)
- `solve1D(f, x0)` — root of a scalar function `f` starting from an **initial guess** `x0`
- `solve(f, x0)` — root of a **system**: `x0` is a vector and `f` returns a vector

- You only write `f(x)` — each function returns the root itself (and warns you if it fails)

Example: CES Market Equilibrium

Constant-elasticity demand and supply:

$$D(p) = A\,p^{-\epsilon_d}, \quad S(p) = B\,p^{\epsilon_s}$$

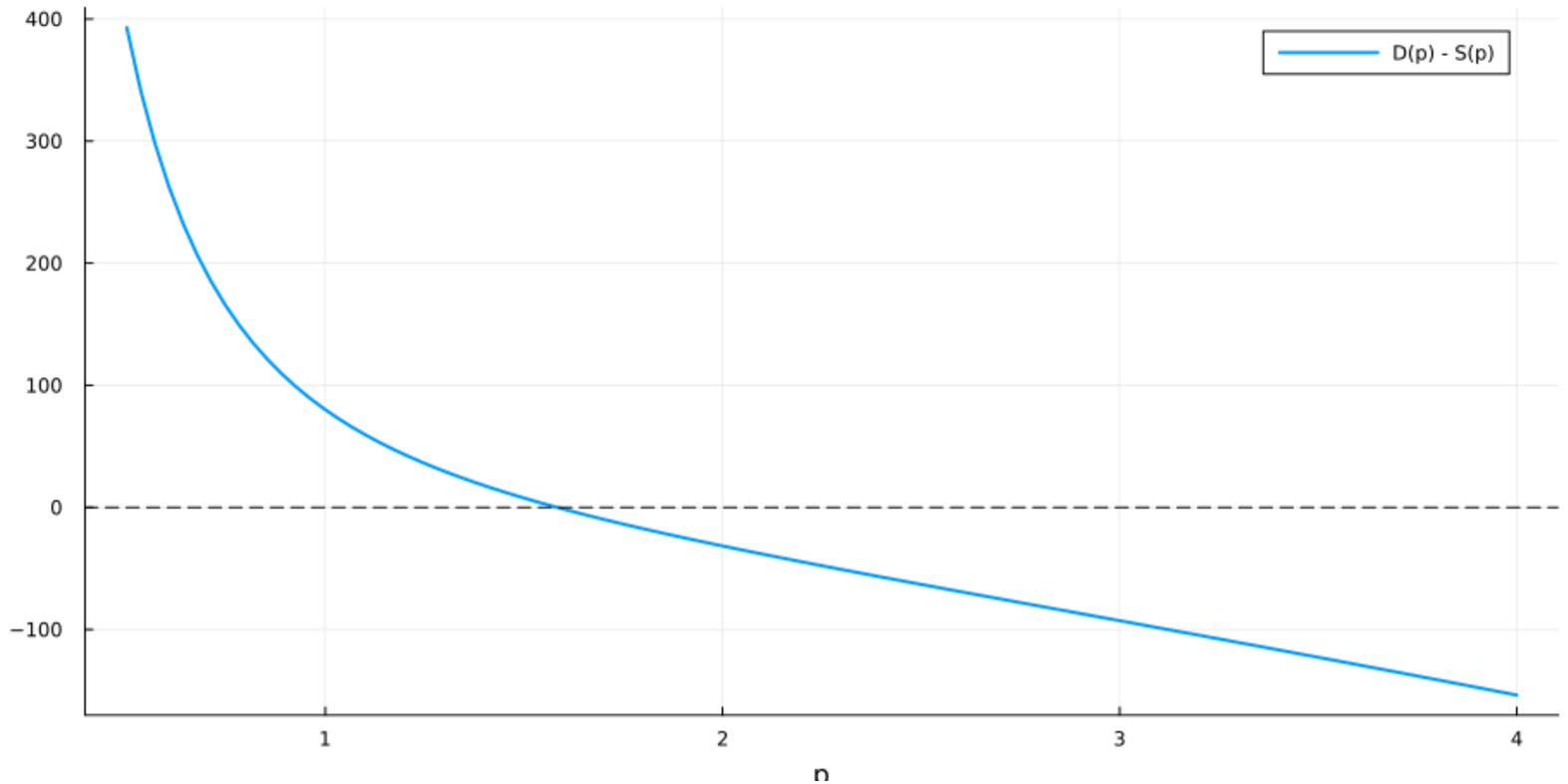
```
function excess_demand(p, A, εd, B, εs)
    D = A * p^(-εd)           # CES demand
    S = B * p^εs              # CES supply
    return D - S              # excess demand
end

A, εd = 100.0, 2.0            # demand parameters
B, εs = 20.0, 1.5             # supply parameters
```

- The equilibrium price p^* solves `excess_demand(p, A, εd, B, εs) = 0`

Bracketing the Root

```
p_grid = LinRange(0.5, 4.0, 100)
ED = [excess_demand(p, A, εd, B, εs) for p in p_grid]
plot(p_grid, ED, lw = 2, label = "D(p) - S(p)", xlabel = "p")
hline!([0], color = :black, ls = :dash, label = "")
```



- Excess demand is **positive** at $p = 0.5$ and **negative** at $p = 4$, so it must cross zero in between
- $[0.5, 4]$ is a **bracket**: a root is guaranteed inside it

Solving for p^*

```
p_star = solve1D(p -> excess_demand(p, A, εd, B, εs), 0.5, 4.0)
```

```
1.583819608766579
```

- `solve1D` needs a function of `p` alone — the anonymous function `p -> excess_demand(p, A, εd, B, εs)` plugs in the parameters
- No bracket? Start from an initial guess instead:

```
solve1D(p -> excess_demand(p, A, εd, B, εs), 1.0)
```

```
1.583819608766579
```

- Same answer here, but a guess comes with no guarantee — a bracket does

Example: Two-Good CES Equilibrium

Two goods with constant-elasticity demand and cross-price effects (substitutes):

```
function two_market(p)
    D1 = 100 * p[1]^(-2.0) * p[2]^(0.5)    # good 1 demand
    S1 = 20 * p[1]^(1.5)                     # good 1 supply
    D2 = 80 * p[2]^(-1.5) * p[1]^(0.3)      # good 2 demand
    S2 = 15 * p[2]^(2.0)                     # good 2 supply
    return [D1 - S1, D2 - S2]
end

solve(two_market, [2.0, 2.0])
```

```
2-element Vector{Float64}:
 1.7069573735223966
 1.6889575066262879
```

- `solve(f, x0)`: one unknown and one equation per entry of the vector
- Raising p_2 increases demand for good 1 (substitutes) — prices are jointly determined

Root Solving Tips

Working with `solve1D` and `solve`

- For one equation, **plot** `f` to find a bracket `[a, b]` where `f(a)` and `f(b)` have opposite signs
- With an initial guess, make it **reasonable** — bad guesses can lead to wrong roots or convergence failure
- Pass parameters with an **anonymous function**: `solve1D(x -> f(x, params), a, b)`
- Check your answer: `f(x*)` should be (approximately) zero

Optimization with Optim

Why Optimization?

- Economics is full of optimization problems:
  - **Consumers** maximize utility subject to a budget constraint
  - **Firms** maximize profit or minimize cost
  - **Econometricians** maximize likelihood functions
- The `Optim.jl` package provides algorithms for unconstrained optimization

The Optim Interface

```
1 using Optim
```

- `Optim.jl` **minimizes** by default
- To **maximize** $f(x)$, minimize $-f(x)$

- For scalar functions: `optimize(f, lower, upper)`
- For multivariate functions: `optimize(f, x0)`

Example: CES Utility Maximization

A consumer with CES utility over two goods and a budget constraint:

$$U(x_1, x_2) = \left(\alpha\, x_1^{\rho} + (1-\alpha)\, x_2^{\rho}\right)^{1/\rho}$$

Substituting $x_2 = (I - p_1 x_1)/p_2$ reduces this to a scalar problem:

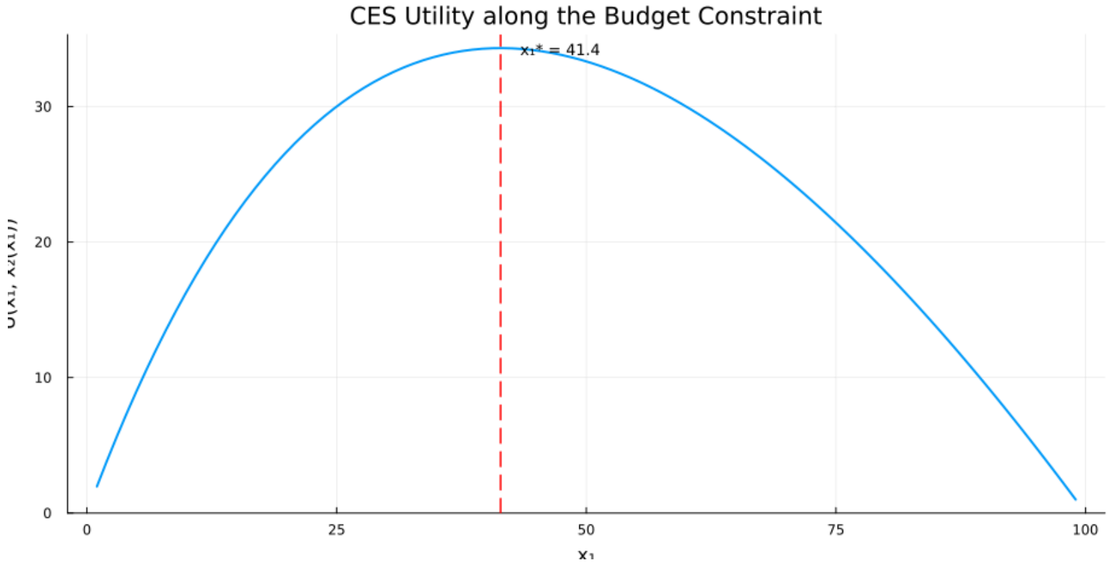
```julia
function neg_utility(x1; α=0.5, ρ=-1.0, p1=1.0, p2=2.0, I=100.0)
    x2 = (I - p1 * x1) / p2
    return -(α * x1^ρ + (1-α) * x2^ρ)^(1/ρ)
end

result = optimize(neg_utility, 1.0, 99.0)
x1_star = Optim.minimizer(result)
x2_star = (100.0 - x1_star) / 2.0
println("x₁ = $(round(x1_star, digits=2)), x₂ = $(round(x2_star, digits=2))")
```

```
x₁ = 41.42, x₂ = 29.29
```

Plotting Utility vs. x_1

```
 1 x1_grid = LinRange(1.0, 99.0, 100) # 100 points between 1 and 99
 2 utility = [-neg_utility(x1) for x1 in x1_grid] # compute utility for each x1
 3
 4 plot(x1_grid, utility,
 5      xlabel = "x₁", ylabel = "U(x₁, x₂(x₁))",
 6      title = "CES Utility along the Budget Constraint",
 7      lw = 2, legend = false)
 8 vline!([x1_star], ls = :dash, color = :red, lw = 1.5)
 9 annotate!(x1_star + 2, maximum(utility) - 0.1,
10           text("x₁* = $(round(x1_star, digits=1))", :left, 9))
```



- The budget constraint pins down $x_2 = (I - p_1 x_1)/p_2$, so utility is a function of x_1 alone
- The dashed red line marks the optimum found by `Optim`

Example: Short-Run Profit Maximization

A firm with Cobb-Douglas production chooses labor L given fixed capital $\bar{K}$:

$$\pi(L) = p \cdot A\,\bar{K}^{\alpha}\,L^{1-\alpha} - wL$$

```julia
function neg_profit(L; A=1.0, K̄=10.0, α=1/3, p=10.0, w=5.0)
    Y = A * K̄^α * L^(1-α)
    return -(p * Y - w * L)          # negate for minimization
end

result = optimize(neg_profit, 0.1, 1000.0)
L_star = Optim.minimizer(result)
println("Optimal labor: $(round(L_star, digits=2))")
println("Maximum profit: $(round(-Optim.minimum(result), digits=2))")
```

```
Optimal labor: 23.7
Maximum profit: 59.26
```

Optim Tips

Working with Optim

- `Optim.jl` **minimizes** — negate your function to maximize
- `Optim.minimizer(result)` returns the optimal x^*
- `Optim.minimum(result)` returns the optimal value $f(x^*)$
- For better performance on smooth problems, try `optimize(f, x0, LBFGS())`

Summary

Key Takeaways

Data Types

- Booleans: `true`/`false`
- Integers: `1, 2, 3`
- Floats: `3.14`
- Strings: `"hello"`
- Arrays: `[1, 2, 3]`
- Tuples: `(1, 2, 3)`
- Named Tuples: `(a=1, b=2)`
- Dicts: `Dict("a" => 1)`

Control Flow

- `for` loops
- `if`/`else` statements
- Comparisons: `<`, `>`, `==`
- Logical: `&&`, `||`, `!`

Functions

- Lambda: `x -> x^2`
- Short: `f(x) = x^2`
- Multi-line: `function ... end`
- Default arguments
- Multiple return values
- Docstrings

Numerical Methods

- Root solving: `using EC410`
- `solve1D(f, a, b)`, `solve(f, x0)`
- Optimization: `Optim.jl`
- Minimizes by default
- Negate to maximize