Computational Economics Lecture 2:
Language Choices, Mathematical Tools, and Environments

Min Fang

University of Florida

Spring 2025

Outline

1. **Language Choices**
2. **Mathematical Tools**
3. **Environments**
4. Covers Chapter 1 to Chapter 13 (https://julia.quantecon.org/intro.html)
5. You should finish Chapter 13 by the end of next week
6. Certainly, feel free to skip some materials!

Language Choices

- First Principal: Whatever that works!
  (I have papers in Python/MATLAB because I can borrow codes from my coauthors)
- Second Principal: The need for speed of running the code
- Third Principal: The need for speed of writing the code
- Run this chapter for details: Chapter 13. The Need for Speed

The Need for Speed: High-level vs Low-level

- High-level languages (Python/MATLAB) aim to maximize productivity by
  - being easy to read, write, and debug
  - automating standard tasks (e.g., memory management)
  - being interactive, etc.
- Low-level languages (C++/FORTRAN) aim for speed and control by
  - being closer to the metal (direct access to CPU, memory, etc.)
  - requiring relatively more information from the user (e.g., all data types must be specified)
- The usual trade-off
  - Python/MATLAB is extremely flexible to write but runs slowly
  - C++ is quite annoying to write but runs quickly
  - Julia is a bit in between and borrowed benefits from both!

The Need for Speed: Multiple Dispatch

- Definition: a function can be dynamically dispatched based on the run-time type or, in the more general case, some other attribute of more than one of its arguments.

- Example: + is a function; 1+1 or 1.0+1.0 are different things on a CPU

This operator + is itself a function with multiple methods.

We can investigate them using the @which macro, which shows the method to which a given call is dispatched

```
x, y = 1.0, 1.0
@which +(x, y)
```

+(x::**T**, y::**T**) *where T<:Union{Float16, Float32, Float64}* in Base at float.jl:491

We see that the operation is sent to the `+` method that specializes in adding floating point numbers.

Here's the integer case

```
x, y = 1, 1
@which +(x, y)
```

+(x::**T**, y::**T**) *where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128, UInt16, UInt32, UInt64, UInt8}* in Base at int.jl:87

- Clearly defining object type could speed up your program and create fewer bugs

The Need for Speed: Foundations

- Example: A function that takes objects a and b and returns 2a+8b
- The assembly language (a symbolic representation of machine code) is

```
pushq   %rbp
movq    %rsp, %rbp
addq    %rdi, %rdi
leaq    (%rdi,%rsi,8), %rax
popq    %rbp
retq
nopl    (%rax)
```

- But you will never write the above, but the below:

```julia
function f(a, b)
    y = 2a + 8b
    return y
end
```

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

or Python

```python
def f(a, b):
    y = 2 * a + 8 * b
    return y
```

or even C

```c
int f(int a, int b) {
    int y = 2 * a + 8 * b;
    return y;
}
```

The Need for Speed: Foundations & In Practice

- The speed of your codes depends on how fast you generate machine codes
  - AOT Compiled Languages: C++/FORTRAN (ahead of time)
  - Interpreted Languages: Python or Terrible-written Julia (during program execution)
  - Just-in-time compilation: Well-written Python or Julia (just-in-time)
- To write efficient code with relatively little effort:
  - JIT compilation (not your effort)
  - Multiple dispatches (not your effort)
  - Type declarations for variables and hence compile efficient code (your effort!)
- Additional tips to write fast codes:
  - Avoid global variables (which people often write badly in MATLAB!)
  - Composite types with abstract field types (which cannot be done in Python!)

Mathematical Tools: Types

- You understand math in equations, but your computer only knows objects and operators
- Object types: Chapter 4. Arrays, Tuples, Ranges, and Other Fundamental Types
  - Often used types: arrays (vector, matrix, etc), tuples and named tuples, ranges
  - Less used types: nothing, missing, unions, and even Markov Chains (QuantEcon.jl package)
- Operating types: Chapter 5. Introduction to Types and Generic Programming
  - Generic Programming: Never manually declare variable types unless necessary
  - High Performance: Learning parametric types (advanced)
  - My Preference: I like manually declaring types to find bugs more easily! (often type error)
- You don't need to be a master of types, but you need to get it correct

Mathematical Tools: General Purpose Packages

- A language is easy to use because of many pre-packaged general-purpose functions
- Chapter 7. General Purpose Packages provides a very good demo of them

```julia
using LinearAlgebra, Statistics
using QuadGK, FastGaussQuadrature, Distributions, Expectations
using Interpolations, Plots,  ProgressMeter
```

- I will introduce Interpolations.jl today and demo you how to go deeper if you need more
  - Basic Interpolation Methods
  - Other Interpolation Packages
  - Read Into Docs and Source Codes
- The same applies to all other packages if needed by you

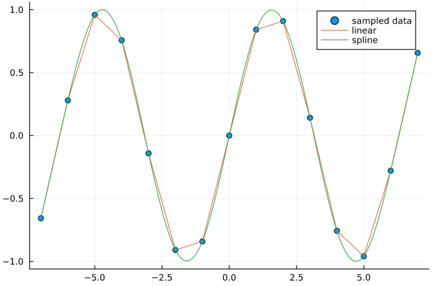
Basic Interpolation Methods

```julia
li = LinearInterpolation(x, y)
li_spline = CubicSplineInterpolation(x, y)

@show li(0.3) # evaluate at a single point

scatter(x, y, label = "sampled data", markersize = 4)
plot!(xf, li.(xf), label = "linear")
plot!(xf, li_spline.(xf), label = "spline")
```

```
li(0.3) = 0.25244129544236954
```



Other Interpolation Packages

Other Interpolation Packages

Other interpolation packages for Julia include:

- ApproXD.jl implements B-spline and linear interpolation in Julia.
- BarycentricInterpolation.jl implements the Barycentric formula for polynomial interpolation on equispaced points and Chebyshev points of the first and second kind.
- BasicInterpolators.jl provides a collection of common interpolation recipes for basic applications.
- BSplineKit.jl offers tools for B-spline based Galerkin and collocation methods, including for interpolation and approximation.
- Curves.jl supports log-interpolation via immutable `Curve` objects.
- DataInterpolations.jl is a library for performing interpolations of one-dimensional data.
- Dierckx.jl is a wrapper for the dierckx Fortran library, which also underlies `scipy.interpolate`.
- DIVAnd.jl for N-dimensional smoothing interpolation.
- FastChebInterp.jl does fast multidimensional Chebyshev interpolation on a hypercube using separable grid of interpolation points.
- FEMBasis.jl contains interpolation routines for standard finite element function spaces.
- FineShift.jl does fast sub-sample shifting of multidimensional arrays.
- FourierTools.jl includes sinc interpolation for up and down sampling.
- GeoStats.jl provides interpolation and simulation methods over complex 2D and 3D meshes.
- GridInterpolations.jl performs multivariate interpolation on a rectilinear grid.
- InterpolationKernels.jl provides a library of interpolation kernels.
- KernelInterpolation.jl implements scattered data interpolations in arbitrary dimensions by radial basis functions with support for solving linear partial differential equations.
- KissSmoothing.jl implements denoising and a Radial Basis Function estimation procedure.
- LinearInterpolations.jl allows for interpolation using weighted averages allowing probability distributions, rotations, and other Lie groups to be interpolated.
- LinearInterpolators.jl provides linear interpolation methods for Julia based on InterpolationKernels.jl, above.
- LocalFunctionApproximation.jl provides local function approximators that interpolates a scalar-valued function across a vector space.
- NaturalNeighbours.jl provides natural neighbour interpolation methods for scattered two-dimensional point sets, with support for derivative generation.
- PCHIPInterpolation.jl for monotonic interpolation.
- PiecewiseLinearApprox.jl performs piecewise linear interpolation over an arbitrary number of dimensions.
- ScatteredInterpolation.jl interpolates scattered data in arbitrary dimensions.

Read Into Docs and Source Codes

- Docs: https://juliamath.github.io/Interpolations.jl/stable/ (More details for your needs)

Interpolations.jl

Search docs

When you have some one-dimensional data that is monotonic, many standard interpolation methods may give an interpolating function that it is not monotonic. Monotonic interpolation ensures that the interpolating function is also monotonic.

Here is an example of making a cumulative distribution function for some data:

```
percentile_values = [0.0, 0.01, 0.1, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 20.0, 3(

y = sort(randn(length(percentile_values))); # some random data

itp_cdf = extrapolate(interpolate(y, percentile_values, SteffenMonotonicInterpolation()), Flat()

t = -3.0:0.01:3.0 # just a range for calculating values of the interpolating function

interpolated_cdf = map(itp_cdf, t) # interpolating the CDF
```

There are a few different monotonic interpolation algorithms. Some guarantee that for non-monotonic data the interpolating function does not exceed the range of values between two successive points while other do not (this is called overshooting in the list below).

- `LinearMonotonicInterpolation` – simple linear interpolation. Does not overshoot.
- `FiniteDifferenceMonotonicInterpolation` – it may overshoot.
- `CardinalMonotonicInterpolation` – it may overshoot.
- `FritschCarlsonMonotonicInterpolation` – it may overshoot.
- `FritschButlandMonotonicInterpolation` – it does not overshoot.
- `SteffenMonotonicInterpolation` – it does not overshoot.

i.e., the one I like the most

You can read about monotonic interpolation in the following sources:

- Fritsch & Carlson (1980), "Monotone Piecewise Cubic Interpolation", doi:10.1137/0717021.
- Fritsch & Butland (1984), "A Method for Constructing Local Monotone Piecewise Cubic Interpolants", doi:10.1137/0905021.
- Steffen (1990), "A Simple Method for Monotonic Interpolation in One Dimension", URL

Mathematical Tools: Data and Statistics Packages

- Not Julia's strength, but necessary (Same applies to MATLAB)
- Chapter 8. Data and Statistics Packages provides a very good demo of them

```julia
using LinearAlgebra, Statistics
using DataFrames, RDatasets, DataFramesMeta, CategoricalArrays, Query
using GLM
```

- Sometimes you need to export statistics, simulated samples, or run regressions
- I will introduce DataFrames.jl today and demo you how to go deeper if you need more
  - Create DataFrame
  - Push Data into DataFrame
  - Generate Statistics from DataFrame
- The same applies to all other packages if needed by you

Create DataFrame

The first is to set up columns and construct a dataframe by assigning names

```
using DataFrames, RDatasets  # RDatasets provides good standard data examples from R

# note use of missing
commodities = ["crude", "gas", "gold", "silver"]
last_price = [4.2, 11.3, 12.1, missing]
df = DataFrame(commod = commodities, price = last_price)
```

4×2 DataFrame

Row	commod	price
	String	Float64?
1	crude	4.2
2	gas	11.3
3	gold	12.1
4	silver	*missing*

Columns of the `DataFrame` can be accessed by name using `df.col`, as below

```
df.price
```

```
4-element Vector{Union{Missing, Float64}}:
  4.2
 11.3
 12.1
   missing
```

Generate Statistics from DataFrame

```julia
using GLM

x = randn(100)
y = 0.9 .* x + 0.5 * rand(100)
df = DataFrame(x = x, y = y)
ols = lm(@formula(y~x), df) # R-style notation
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePred

y ~ 1 + x

Coefficients:
──────────────────────────────────────────────────────────────────────────
                Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%
──────────────────────────────────────────────────────────────────────────
(Intercept)  0.248789   0.014957   16.63    <1e-29   0.219107   0.278471
x            0.903574   0.0146723  61.58    <1e-79   0.874458   0.932691
──────────────────────────────────────────────────────────────────────────
```

To display the results in a useful tables for LaTeX and the REPL, use RegressionTables for output similar to the Stata package esttab and the R package stargazer.

```julia
using RegressionTables
regtable(ols)
# regtable(ols,  renderSettings = latexOutput()) # for LaTex output
```

```
----------------------
                   y
----------------------
(Intercept)   0.249***
              (0.015)
x             0.904***
              (0.015)
----------------------
N                  100
R2               0.975
----------------------
```

Mathematical Tools: Solvers/Optimizers

- Chapter 9. Solvers, Optimizers, and Automatic Differentiation

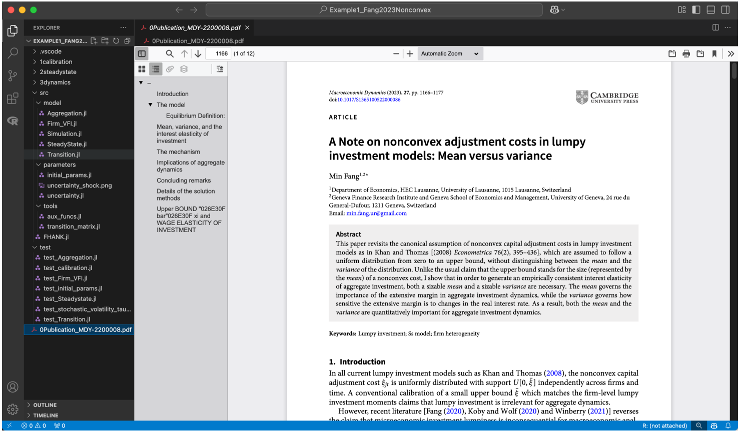
```
using LinearAlgebra, Statistics
using ForwardDiff, Optim, JuMP, Ipopt, Roots, NLsolve
using Optim: converged, maximum, maximizer, minimizer, iterations #some extra functions
```

- Optimizer is at the core of any optimization problem (therefore, every econ problem!)
- We will only touch bases today and do much more later (explain details)
  - Optim.jl: (Unconstrained or box-bounded) optimization of univariate and multivariate function
  - JuMP.jl + IPOPT (Interior Point Optimizer): A much more general idea!
  - BlackBoxOptim.jl, NLsolve.jl, LeastSquaresOptim.jl: Mainly specific cases
- Personal Experience: Optim.jl for simple cases; IPOPT for complicated cases
- We will talk about details and auto-diff in other algorithm lectures later
- *Disclaimer: IPOPT is not Julia-specific; it is a universal package across many languages!

Environment

- Finally, about the environment! Please give a read of the three chapters:
- Chapter 10. Visual Studio Code and Other Tools
- Chapter 11. GitHub, Version Control and Collaboration
- Chapter 12. Packages, Testing, and Continuous Integration
- I do not expect you to do exactly the same, which is quite serious.
- But you may do to some extent, like one of my papers here: (GitHub)
- Nonconvex Adjustment Costs in Lumpy Investment Models: Mean versus Variance

Environment: Demo



- Pros: Source files, test files, folder management
- Cons: No version controls, no configure, no collaboration (joke!)