High Integrity Systems Project Time Series Analysis

Assignment 5

Mehjabeen Jahangeer Khan

1 Introduction to Juliya

- Work began in 2009 but first appeared in 2012
- Originally developed by computer scientists and mathematicians at MIT [1]
- Juliya combines three key features for highly intensive computing tasks which maybe currently no programming language does
  - fast
  - easy to learn and use
  - open source
- Julia is high-level, dynamic programming language
- It is designed to give users the speed of C/C++ but remains as easy as Python.
- Now programmers can solve problems faster and more effectively
- Juliya is great for computational complex problems
- Juliya is general purpose language but can be used for other tasks such as:
  - Data science
  - Artificial intelligence
  - Machine learning
  - Modeling and simulation
  - Numerical analysis
  - Web development
  - Game development
  - Many more
- Juliya programs can reuse libraries from other languages by calling them e.g calling C
- Juliya can also be called from other languages e.g. Python and R

2 Installing Juliya

- Go to the Microsoft store and search for Juliya and download it
- Open Juliya

- To understand the complete tutorial you can visit `chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.sas.upenn.edu/~jesusfv/Chapter_HPC_8_Julia.pdf`
- There are two ways to use Juliya
  - Launch a Juliya terminal window (the easiest way to learn and experiment is through this interactive session). This interactive session is also known as REPL (read-eval-print-loop). To open the interactive session you just need to click on the Juliya executable/command line. To exit the interactive session tye *CTRL + D* or exit().

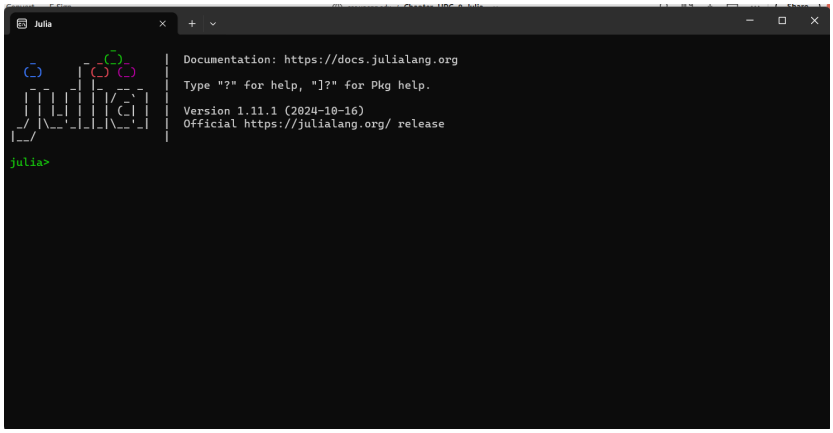

Figure 1: Juliya terminal process

  - Use Juliya through Jupyter. The first time you need to install the package follow the steps and later on you can use the package. Steps: write in Julia prompt >import Pkg >Pkg.add("IJulia") >using IJulia >notebook()
- First command to run is *versioninfo()*
- Juliya has auto-completion
- *?command* will provide you more information on command
-] is used to switch to the package manager mode
- *CTRL + L* for clearing the console

2.1 Juliya commands

2.2 Useful Packages

To use any package in your code you only need to include [2].
Some useful packages in Julia are:

We also saw in Section 1.2, that one of the first things you may want to do after installing `Julia` is to add some useful packages. Recall that the first thing you need is to switch to the package manager mode with `]`. After doing so, and to check the status of your packages, and to add, update, and remove additional packages, you can use:

```
st                      # checks status
add IJulia              # add package IJulia
up IJulia               # update IJulia
rm IJulia               # remove package
```

The first command, `st`, checks the status of all the packages installed in your computer. The second command, `add IJulia`, will add the package `IJulia` that we will need below. Be patient: each command might take some time to complete, but this only needs to be done when you first install `Julia`. The third command, `up IJulia`, will updatate the package to the most recent version. The last command, `rm IJulia`, will remove the package, but hopefully you will be convinced that `IJulia` is a good package to keep.

Figure 2: Switching to package manager mode

```
?                       # help
up arrow key            # previous command
down arrow key          # next command
3+2;                    # ; suppresses output if working on the REPL
;                       # activates shell model
clearconsole()          # clearconsole; also ctrl+L
```

The result from the last computation performed by `Julia` will always be stored in `ans`:

```
ans                     # previous answer
```

You can use `ans` as any other variable or even redefine it:

```
ans+1.0                 # adds 1.0 to the previous answer
ans = 3.0               # makes ans equal to 3.0
println(ans)            # prints ans in screen
```

If there was no previous computation, the value of `ans` will be `nothing`.

Figure 3: Some of Juliya commands

```
using Pandas            # Using package Pandas
```

The first time you use a package, it will require some compilation time. You will not need to wait the second time you use the package, even if you are working in a different session days later. T

Figure 4: Using any package

- QuantEcon : Quantitative Economics functions for Julia.
- Plots: easy plots.

- PyPlot : plotting for Julia based on matplotlib.pyplot.
- Gadfly : another plotting package; it follows Hadley Wickhams's ggplot2 for R and the ideas of Wilkinson (2005).
- Distributions : probability distributions and associated functions.
- DataFrames : to work with tabular data.
- Pandas : a front-end to work with Python's Pandas.
- TensorFlow : a Julia wrapper for TensorFlow.

Several packages facilitate the interaction of Julia with other common programming languages. Among those, we can highlight:

- Pycall : call Python functions.
- JavaCall : call Java from Julia.
- RCall: embedded R within Julia.

Julia can directly call C++ and Python's functions. Note that most of these packages already come with the JuliaPro distribution. There are additional commands to develop and distribute packages.

2.3 Types

Juliya has variables, values, and types. A variable is always bound to value. Juliya is case-sensitive. A value is a content (1,2,3,'abc', etc.). Technically Julia considers that all values are objects which is why Julia is closer to Objected Oriented Programming. Finally, values have types (i.e. integer, boolean, float, string). A variable does not have a type, but its value has. Types specify the attributes of content.

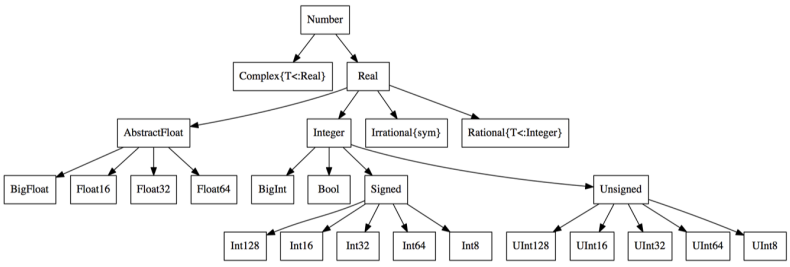

Figure 5: Types in Julia

2.4 Variables

Here are some basic examples of how to declare a variable and assign it a value with different types in Fig 6. Julia has a style guide (`https://docs.julialang.`

```
a = 3                   # integer
a = 0x3                 # unsigned integer, hexadecimal base
a = 0b11                # unsigned integer, binary base
a = 3.0                 # Float64
a = 4 + 3im             # imaginary
a = complex(4,3)        # same as above
a = true                # boolean
a = "String"            # string
```

Figure 6: Declaring a variable

`org/en/latest/manual/style-guide/`) for variables, functions, and types of naming conventions that we will (mostly) follow in the next pages. By default, integer values will be Int64 and floating point values will be Float64, but we also have shorter and longer types. Some common manipulations with variables are included in Fig 7. We also have many rounding, truncation, and module

```
eval(a)         # evaluates expression a in a global scope
real(a)         # real part of a
imag(a)         # imaginary part of a
reim(a)         # real and imaginary part of a (a tuple)
conj(a)         # complex conjugate of a
angle(a)        # phase angle of a in radians
cis(a)          # exp(i*a)
sign(a)         # sign of a
```

Figure 7: Manipulation with variables

functions in Fig 8. The rounding and truncation functions have detailed options to accomplish various numerical goals(including changes in the default of ties, which is rounding down).

2.5 Functions

In the tradition of programming languages in the functional approach, Julia considers functions "first-class citizens" (i.e., an entity that can implement all the operations-which are themselves functions- available to other entities). This means, among other things, that Julia likes to work with functions without side effects and that you can follow the recent boom in functional programming without jumping into purely functional language. Recall that functions in Julia use

```
round(a)        # rounding a to closest floating point natural
ceil(a)         # round up
floor(a)        # round down
trunc(a)        # truncate toward zero
clamp(a,low,high) # returns a clamped to [a,b]
mod2pi(a)       # module after division by 2\pi
modf(a)         # tuple with the fractional and integral part of a
```

Figure 8: Rounding, truncation, and module functions

methods with multiple dispatch: each function can be associated with hundreds of different methods. Furthermore, you can add methods to an already existing function. There are two ways to create a function shown in Fig 9

```
# One-line
myfunction1(var) = var+1
# Several lines
function myfunction2(var1, var2="Float64", var3=1)
        output1 = var1+2
        output2 = var2+4
        output3 = var3+3 # var3 is optional, by default var3=1
        return [output1 output2 output3]
end
```

Figure 9: Functions

3 Juliya Scientific Libraries

Julia is a high-performance programming language that has gained significant traction in the scientific computing, machine learning, and modeling and simulation communities due to its speed, ease of use, and growing ecosystem of libraries. Below, I'll provide an overview of Julia's libraries in key areas such as scientific computing, machine learning, and modeling and simulation.

3.1 Scientific Computing with Julia

Julia excels in scientific computing, providing a rich ecosystem of libraries for various scientific domains.
Here are some prominent libraries:

- SciML (Scientific Types and Models): SciML is an umbrella organization for scientific machine learning and simulation libraries in Julia. It includes several key packages for solving problems in simulation, modeling, optimization, and analysis.

  - DifferentialEquations.jl: One of the most widely used libraries for solving differential equations, including ODEs (Ordinary Differential Equations), DDEs (Delay Differential Equations), SDEs (Stochastic Differential Equations), and PDEs (Partial Differential Equations).
  - Optimization.jl: A library for solving mathematical optimization problems, including linear programming, nonlinear programming, and mixed-integer optimization.
  - Turing.jl: A probabilistic programming library that supports Bayesian inference using Markov Chain Monte Carlo (MCMC) and other methods.
  - ProbabilisticProgramming.jl: Another package for probabilistic modeling and inference that integrates with SciML.
  - DynamicalSystems.jl: A library for studying dynamical systems, including chaos, bifurcation, and attractor analysis.

- Plots.jl: While not strictly scientific computing, Plots.jl is an important visualization tool for scientific computing in Julia. It supports multiple backends like GR, Plotly, and PyPlot and is easy to integrate into scientific workflows.

- DataFrames.jl: A powerful library for handling and manipulating tabular data, akin to pandas in Python.

- LinearAlgebra and SparseArrays: Julia has native support for high-performance linear algebra, including support for sparse matrices, making it a go-to tool for scientific applications that require large-scale matrix computations.

3.2 Machine Learning in Julia

Julia has several libraries dedicated to machine learning, providing both flexibility and speed:

- Flux.jl: One of the most popular deep learning frameworks in Julia. It is lightweight, flexible, and provides a clean interface for building neural networks and other machine learning models. It supports GPU acceleration with CUDA.jl.

- Knet.jl: Another deep learning library, similar to Flux, but designed to be more performance-oriented and for research in machine learning.

- MLJ.jl: A comprehensive machine learning framework in Julia. It offers tools for data preprocessing, model training, evaluation, and deployment. MLJ is highly extensible and integrates well with SciML for scientific machine learning applications.

- Turing.jl: In addition to its use in scientific modeling, Turing is also used in machine learning for probabilistic modeling, providing a powerful tool for Bayesian machine learning.

- ScikitLearn.jl: A Julia wrapper around the Python scikit-learn library, providing access to a wide range of classical machine learning algorithms.
- CUDA.jl: While not specifically a machine learning library, CUDA.jl enables GPU acceleration, which is crucial for training large machine learning models efficiently.
- Zygote.jl: A high-performance automatic differentiation library in Julia, used for backpropagation in neural networks, often paired with Flux.jl or Knet.jl for deep learning.

3.3 Modeling and Simulation in Julia

Julia provides powerful tools for modeling complex systems, including simulations of physical systems, agent-based modeling, and optimization of real-world systems:

- DifferentialEquations.jl: As mentioned earlier, this package is essential for simulation, particularly for solving differential equations in areas such as physics, biology, and economics. It supports a wide range of solvers and allows for high customization of models.
- Agents.jl: A library for agent-based modeling, which is useful for simulating interactions in systems where entities (agents) act based on a set of rules, such as in sociology, ecology, and economics.
- JuMP.jl: This library is used for mathematical optimization. It provides a high-level interface for formulating optimization problems, especially linear and mixed-integer optimization problems. It's widely used in operations research, economics, and engineering design.
- QuantumOptics.jl: A package for simulating quantum optics and quantum systems, which is important for researchers in quantum computing and quantum information science.
- SimJulia.jl: A simulation library for discrete-event simulation (DES), often used for systems modeling in engineering, logistics, and manufacturing.
- SystemIdentification.jl: For creating models based on observed data (e.g., identifying transfer functions, state-space models) and used in control systems, process engineering, and signal processing.
- ModelingToolkit.jl: A library for symbolic modeling and analysis, making it easier to define, manipulate, and solve differential equations symbolically. It integrates well with other packages such as DifferentialEquations.jl

3.4 Interfacing with Other Languages

- PyCall.jl: This library allows Julia to call Python functions directly, enabling users to leverage the extensive Python ecosystem (e.g., TensorFlow, scikit-learn, pandas) when needed.
- RCall.jl: Similarly, RCall.jl allows the integration of R libraries into Julia code.
- Cxx.jl: Julia can also interface with C++ libraries, which is useful for performance-critical tasks or to reuse existing C++ code.

4 The state of Juliya for scientific machine learning (Summary)

4.1 Introduction — A Tale of Two Languages

The introduction discusses the historical dominance of Python in machine learning and the natural sciences due to its intuitiveness and rich ecosystem, despite its limitations such as being slow, a scripting language, and difficult to maintain. Julia, introduced in 2012, was designed to address these issues, offering features for intuitive and fast numerical computations. Despite its steady growth and advantages for scientific machine learning (SML), Julia has not overtaken Python in popularity, partly due to Python's established momentum within the community [3].

The work examines Julia's readiness as a primary tool for SML, focusing on its libraries, performance, design philosophy, and user ergonomics. It highlights how Julia's abstractions differ from those of other ecosystems, influencing how users approach SML problems. However, the authors argue that Julia's ecosystem has significant limitations that hinder its broader adoption. The paper concludes by urging the community to address these limitations to enhance Julia's utility in scientific machine learning.

4.2 The scientific computing ecosystem

This section compares Julia's ecosystem to other programming languages, particularly Python, while also referencing C, C++, Fortran, and CUDA. The analysis focuses on Julia's strengths in scientific machine learning problems (SMLPs) and highlights its capabilities across various domains:

- **Linear Algebra:** Julia's Just-in-Time (JIT) compiler makes it highly efficient for numerical computations, achieving speeds comparable to C for tasks like random matrix operations. Its robust support for sparse matrix computations makes it especially suited for graph-related SMLPs, such as SLAM (Simultaneous Localization and Mapping). While Python benefits from tools like Jax for JIT compilation, Julia's functionality and speed offer a distinct edge.

- **Constrained Optimization:** Julia excels in optimization tasks, with advanced tools for working on manifolds (e.g., Manopt) and Physics-Informed Neural Networks (PINNs). Julia's libraries like Flux.jl, Lux.jl, and NeuralPDE.jl provide seamless support for neural and differential equation-based approaches, surpassing Python in breadth and integration.

- **Automatic Differentiation (AD):** Julia offers forward and reverse mode AD through packages like ForwardDiff.jl and Zygote.jl, but the ecosystem is fragmented, requiring more domain knowledge compared to Python's more streamlined tools (e.g., Jax). Custom differentiation rules in Julia are less user-friendly than Python's intuitive macros.

- **Probabilistic Programming:** Julia provides robust tools like SOSS.jl and Turing.jl for Bayesian statistics, which simplify SMLPs and support advanced functionalities like measure-theoretic tools. While Python has alternatives like PyAutoFit and NumPyro, Julia's PPL ecosystem is more comprehensive and versatile.

- **Symbolic Regression:** Julia's SymbolicRegression.jl leads in this area, with Python's PySR relying on it as a wrapper. Julia's native support for symbolic regression is unmatched in other ecosystems.

  While Julia excels in performance, integration, and specialized libraries, it faces challenges in user ergonomics and ecosystem fragmentation, particularly in areas like automatic differentiation. This makes Julia highly suitable for certain SMLPs but less universally adopted than Python.

4.3 Design Philosophy and Ergonomic Machine Learning

This section explores Julia's design philosophy and how its features enhance usability and efficiency in solving scientific machine learning problems (SMLPs):

- **Multiple Dispatch:** Julia leverages multiple dispatch to choose function behavior based on input types, making code more user-friendly and eliminating the need for complex subclass hierarchies. For instance, unlike Python's TreeCorr library, which requires users to manage 18 different subclasses for correlation functions, Julia's CosmoCorr.jl achieves the same functionality more intuitively by redefining the same function for different input types. Multiple dispatch is also integral to libraries like JuMP.jl and ForwardDiff.jl, promoting simplicity and flexibility.

- **Composition vs. Inheritance:** Julia emphasizes composable interfaces over inheritance. Functions in Julia are globally declared and adaptable for various types, enhancing modularity. For example, Flux.jl represents neural networks as chains of composable functions, avoiding rigid input formatting and providing greater flexibility compared to PyTorch. This approach minimizes package conflicts and enhances usability, further supported by Julia's efficient package manager, Pkg.jl, which outperforms Python's Pip and Conda in handling dependencies.

- **The Two-Language Problem:** Julia addresses the "two-language problem," where other ecosystems require a mix of a high-level language (e.g., Python) for ease of use and a low-level language (e.g., C++) for performance. This separation complicates project maintenance, reproducibility, and contributions. Julia eliminates this barrier by allowing both rapid prototyping and performance optimization within the same language. Libraries like Flux.jl and CosmoCorr.jl exemplify Julia's ability to seamlessly integrate high-level functionality with performance-critical tasks.

  By integrating multiple dispatch, composition, and a unified high-performance language, Julia provides an ergonomic and efficient development environment, uniquely suited for SMLPs.

4.4 Limitations of Julia for Scientific Machine Learning

This section outlines key limitations hindering Julia's adoption for scientific machine learning problems (SMLPs) despite its strengths, followed by a call for revitalizing the language's development:

Limitations of Julia Lack of Software Engineering Features: Julia's testing infrastructure is underdeveloped compared to Python's robust libraries like unittest and pytest. It also lacks advanced testing methodologies (e.g., property-based, symbolic execution, and contract-based testing), which are critical for ensuring the precision of scientific algorithms. Furthermore, Julia has no actively maintained static type checker, compromising rigorous code verification.

Complex Debugging Messages: Error messages in Julia are notoriously difficult to interpret due to long stack traces and the complexities of multiple dispatch, leading to frustration among users.

Limited Industry Adoption: While Julia is growing in academic circles, its lack of widespread support from industry giants (unlike Python's Jax, backed by Google DeepMind) limits its reach. Industry-standard tools like HuggingFace's model-sharing API are designed exclusively for Python, forcing Julia users to rely on Python wrappers for compatibility.

Poor Interoperability: Although Julia can call functions from Python, R, C, and C++ easily, the reverse is complex and unintuitive, requiring explicit management of Julia contexts and data types, complicating integration into multi-language workflows.

4.5 Conclusion and Call to Action

Despite Julia's advanced tools for constrained optimization and its ability to address Python's weaknesses, its adoption remains limited. The paper suggests that Julia's community has shifted focus from improving the language to building domain-specific libraries, leading to stagnation in addressing core language issues.

The authors argue that Julia needs a renewed vision and a formal roadmap to tackle critical limitations at the language level, mirroring Python's goal-oriented development. Without such efforts, Julia risks repeating Python's shortcomings

and missing its opportunity to become the de-facto language for scientific machine learning.

5 Practice

- To start notebook (> using IJulia > notebook())
- Check your current path (currentPath = pwd())
- to change directory use "cd("assignment5 juliya")

References

[1] *Julia (programming language)*.

[2] P. A. . Sciences, *Juliya Tutorial*.

[3] E. Berman and J. Ginesin, "The state of julia for scientific machine learning," *arXiv preprint arXiv:2410.10908*, 2024.