# Project Summary and Background

In this project I solved 3 simple macroeconomic models in 6 (relatively) common programming languages: Python, R, Matlab, Julia, C++ and Fortran. The purpose was to practice these languages and to serve as a template for future work. The models are a basic growth model, a stochastic growth model, and a model with idiosyncratic endowments. I show some basic performance benchmarking and document my subjective experience across the languages. My main conclusions are as follows:

- As expected, C++ and Fortran provided the best performance. Python, R, and Matlab provided poor performance when using for-loops but had considerably better performance when using a vectorized approach instead.
- Python punched above it's weight with Numba, performing better than Fortran with minimal additional development time.
- Julia stood out as having the cleanest syntax, speed that rivalled C++ and Fortran, and easy parallelization without any additional packages or development time.
- Python, R, Matlab, and C++ have excellent IDEs, most importantly with built-in debugging and performance profiling. Julia and especially Fotran suffer from worse IDE and community support.
- My Fortran experience was disappointing; it was challenging to set up the environment, and even then, the debugging and profiling tools fell short of other languages. This, combined by the poor documentation and support, has made me view C++ as the superior high-performance language.
- I believe Julia is the best choice for this type of problem when prototyping or when performance is not critical thanks to its clean and simple syntax. Its performance may still be adequate for high-performance applications, and in the event that it's not, I would choose C++.

# Description of Models

I solved three basic macroeconomic models with increasing complexity. Table 1 shows the details of each model. At the heart of each model is an infinitely lived household that has income (from wages, investments, or endowments) that they must allocate between current period consumption and savings for next period. The Base Model uses a single firm that takes capital and labor to produce more of the capital good. The Stochastic Growth Model adds a productivity shock that follow a Markov process. Lastly, the Idiosyncratic Endowment Model replaces the firm with an endowment that follows a Markov process and households can borrow and lend amongst themselves. They are therefore heterogeneous in both their starting period wealth and their current endowment.

The models are solved by using value function iteration to find a solution to the sequential social planner's problem. The corresponding value function and policy functions are the main result for each model. For the Idiosyncratic Endowment Model, the distribution of wealth and endowment levels is also found.

Table 1. Description of macroeconomic models. A description of each variable is shown in the Additional Technical Notes.

| | **Base Model** | **Stochastic Growth** | **Idiosyncratic Endowment** |
|---|---|---|---|
| **Environment** | • Households<br>  ○ Infinitely lived<br>  ○ Identical<br>  ○ Discount future periods<br>  ○ 1 unit of labor endowment<br>• Firms<br>  ○ Identical<br>  ○ Cobb-Douglas production function<br>• Other restrictions/assumptions<br>  ○ Choice variables must be non-negative<br>  ○ Labor supply is inelastic and equal to 1 | • Same environment as Base Model<br>• Firm's productivity follows a stochastic process that is represented by a Markov process with 11 states. | • Households<br>  ○ Heterogeneous in starting capital and endowment level<br>  ○ CRRA utility with risk aversion parameter $\sigma$<br>  ○ Endowment follows a Markov process<br>  ○ Households can lend amongst themselves subject to exogenous borrowing limit and net zero supply of capital.<br>• No production |
| **Household's Problem** | $\max_{\{c_t,k_t\}_{t=0}^{\infty}} \left\{ \sum_{t=0}^{\infty} \beta^t \log\ c_t \right\}$<br>$s.t:$<br>$c_t + k_{t+1} \leq\ 1-\delta\ k_t + r_t k_t + w_t$ | Same as Base Model | $\max_{\{c_t,k_t\}_{t=0}^{\infty}} \left\{ \sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\sigma}}{1-\sigma} \right\}$<br>$s.t:$<br>$c_t + p_t a_{t+1} \leq a_t + e_t$<br>$c_t \geq 0$<br>$a_{t+1} \in [-2,4]$ |
| **Firm's Problem** | $\max_{K,N}\{K^\alpha N^{1-\alpha} - rK - wN\}$ | $\max_{K,N}\{z_t K^\alpha N^{1-\alpha} - rK - wN\}$ | • Uses stochastic endowment and inter-household borrowing instead of a firm and capital accumulation. |
| **Sequential Problem** | $V\ k\ = \max_{k'}\{\log\ k^\alpha +\ 1-\delta\ k-k'$<br>$+\beta V\ k\ \}$<br>$g\ k\ = k'$ | $V\ k,z\ = \max_{k'}\{\log\ zk^\alpha +\ 1-\delta\ k-k'$<br>$+\beta \cdot E[V\ k',z'\ \|z]\}$<br>$where\ E[z'\|z] = \sum_{z' \in Z} \pi\ z'\|z\ \cdot z$ | $V\ a,e\ = \max_{a'}\{u\ c\ +\beta \cdot E[V\ a',e'\ \|e]\}$ |

| | | | |
|---|---|---|---|
| **Solution Approach** | • Solve the sequential Social Planner's Problem for a representative household by value function iteration.<br>• Output<br>  ○ Value Function<br>  ○ Policy Function<br>  ○ Capital and Labor each period<br>  ○ Interest rate and wage each period | • Same as Base Model while also incorporating the expectation of future productivity into sequential Social Planner's Problem. | • Guess bond price.<br>• Solve recursive competitive equilibrium by value function iteration.<br>• Get corresponding policy function.<br>• Calculate net demand for bonds.<br>• Perform binary search over bond prices (positive net demand, try higher price) until net demand is sufficiently close to 0. |

# Programming

## Approach

All three models are solved with value function iteration according to the follow steps:

1. Solve the household's problem for a given starting state (starting wealth, productivity, or endowment).
2. Apply this function to each possible starting state to get the Value Function and Policy Function.
3. Repeat steps 1. and 2. using the newly found Value Function and Policy Function until the functions converge.

In the Stochastic Growth and Idiosyncratic Endowment Models, Step 1. is contained within its own function. This approach lends itself well to implementing the function in parallel over all starting states and makes it easy to modify the algorithm used to solve the household's optimization problem, which is the computation bottleneck. To illustrate, the Stochastic Growth solutions feature 2 different versions of this function. Version 1 uses a simple for-loop to check all possible values of the choice variable, while version 2 eschews the for-loop for a vectorized approach. The Python implementation includes 2 additional versions that have been further optimized with the Numba package. I also included a parallel/multi-threaded implementation in Python and Julia to illustrate that the process of solving the household's problem can be run in parallel over all possible starting conditions.

I used the exact same solution approach in each language. Most lines of code perform the same operation and only differ in the language syntax. I also chose to keep the solution approach simplistic; at the heart of each solution is value function iteration and solving the household's problem by checking all possible values of the choice variables. The solution approach could be made more efficient though techniques such as policy function iteration, exploiting monotonicity, incorporating optimality conditions, and using non-linear grids. It therefore performs more loops and stores more information in memory that is strictly necessary but does not introduce features or bottlenecks that would not be present in larger or more complicated models.

I tried to find and employ the best practices in each language. For example, whether to loop over indices or use the enumerate function in Julia, whether to use C-style arrays or std::vector arrays in C++, or whether to loop over columns versus rows according to how the language stores multi-dimensional arrays.

## Languages

I solved the models in 6 different programming languages. I chose Python, R, Matlab, and Julia as they are commonly used in economics for data processing, statistical models, or numerical computations. I chose C++ and Fortran since they are high-performance languages known for superior computational speeds at the cost of more challenging development. Fortran maintains a niche in scientific computing and C++ is ubiquitous in applications like video games.

## Benchmarking

Table 2 shows performance benchmarks for each of the 6 languages and the different versions of the solution approach. Python's base implementation with for-loops was far behind other languages in terms of speed. Python, Matlab, and R all benefited significantly from using vectorized functions instead of a for-loop. Python had notable additional improvements using the package Numba which creates precompiled and optimized code. This brought the performance of Python for-loops ahead of even the vectorized implementations in other languages, and when Numba was combined with vectorized Numpy functions, lead to code that was faster than the Fortran implementations. As expected, C++ and Fortran had solid performance, although C++ is consistently faster. Julia stood out as having "out-of-the-box" for-loops with speed that rivalled Fortran and C++, and that were easily parallelized. Parallelization was implemented with a single line of code and with the extra computing power, Julia bested C++ for the stochastic Growth Model. In the other 2 models, Julia's for-loop implementation was between that of C++ and Fortran. I must note, however, that a decent knowledge of the language is still needed to avoid common mistakes such as needlessly copying arrays.

## Development Environment

I would also like to note my subjective experiences as a user with a moderate amount of experience in scientific computing. Prior to this project, I have executed at least 2 major projects in each of Python, R, Matlab and Julia and had no prior experience in C++ and Fortran. Example projects include reduced-form papers in R, machine learning projects in Python, replicating the original BLP paper in Julia, and solving the Aiyagari model in Matlab.

Python, R, Matlab and C++ all have at least one excellent IDE. R and Matlab have their default IDEs dedicated to scientific computing which brings to focus the most relevant features: variable explorer, plotting, debugging, and profiling. I found the Spyder IDE for Python to have a similarly good level of functionality, and Python's popularity means that there are many great choices of a suitably focused IDE.

My experience learning C++ with Visual Studio 2022 was excellent. While not focused on scientific computing, the built-in debugging and performance profiling tools were second to none. Further, Visual Studio handles compiler commands and git versioning. It was much more welcoming to have all these tools available out-of-the-box and with the click of a button.

In contrast, programming in Fortran was the opposite experience. I struggled to find a suitable development environment and ultimately went with VS Code. Setting up the build, debug, and profiling process was challenging. Despite this functionality being essential to scientific computing, I could not find self-contained step-by-step setup guide that worked on my system, instead piecing together this information from scant sources. I'm not ashamed to say that it took me over 10 hours before I could even run a "hello world" script. Even after setting up all these features, the functionality is still inferior to, for example, the built-in performance profiling of R Studio, Matlab, and Visual Studio.

My experience with Julia sat somewhere in between. It was easier to get started with Julia in VS Code as it is now the only supported development platform. Still, VS Code is just a text editor with various extensions attached, and not built specifically for scientific computing. Therefore, I found the functionality and usability of key features to be somewhat worse than the other development environments.

Lastly, I consider the "development environment" beyond the four corners of the program window. For example, Python is one of the most popular programming languages and is ubiquitous in data science applications. There is an abundance of documentation (i.e. questions and answers on the Stack Overflow website) on any conceivable question related to numerical computations. There is also an excellent collection of mature packages for extracting and processing numerical data. This doesn't show up on the benchmarks but does have a big impact on development time and reduces the likelihood of achieving poor performance due to mistakes. On the other extreme is Fortran. It's much harder to find relevant documentation or sample code even for common tasks like reading a CSV file. In between are R and Matlab, which also have a large user base, and below that, Julia and C++ as Julia is a relatively new language and C++ is less commonly used in scientific computing.

Table 2. Performance benchmarking results. All times are in seconds.

| Language | Approach | Base Model | Stochastic Growth | | Idiosyncratic Endowment |
|---|---|---|---|---|---|
| | | | Single Core | Multithreaded | |
| **Python** | Base for-loop | 50 | 1117 | 244 | 726 |
| | Numpy vectorized calculation | | 26 | | 100 |
| | Numba precompiled for-loop | | 17 | | 19 |
| | Numba precompiled + Numpy vectorized calculation | | 7.3 | | 14 |
| **Julia** | Base for-loop | 0.8 | 33 | 4.2 | 16 |
| | Vectorized calculation | | 9.1 | 8.1 | 16 |
| **Matlab** | Base for-loop | 5.1 | 174 | | 73 |
| | Vectorized calculation | | 22 | | 60 |
| **R** | Base for-loop | 12 | 602 | | 440 |
| | Vectorized calculation | | 42 | | 75 |
| **C++** | Base for-loop | 0.8 | 5.0 | | 4.5 |
| **Fortran** | Base do-loop | 1.3 | 14 | | 20 |

# Additional Technical Notes

## Description of variables used in Table 1.

| | |
|---|---|
| $c_t$ | Consumption in period t. |
| $k_t$ | Capital allocation for period t. |
| $r_t$ | Interest rate in period t. |
| $w_t$ | Wage in period t. |
| $p_t$ | Price of 1 unit of consumption in period t+1. |
| $a_t$ | Amount of the consumption good loaned in period t-1. |
| $e_t$ | Endowment in period t. Random variable that follows a Markov process. |
| $\delta$ | Depreciation rate. |
| $\sigma$ | Utility parameter; coefficient of risk aversion. |
| $\beta$ | Discount factor. |
| $\alpha$ | Cobb-Douglas productivity parameter. |
| $K$ | Firm's capital demand. |
| $N$ | Firm's labor demand. |
| $z_t$ | Firm productivity in period t. Random variable that follows a Markov process. |
| $V(.,.)$ | Household's value function. |

## Language-specific design and technical notes

| Language | Notes |
|---|---|
| **Python** | • Python benchmark scripts for the 4 versions of the Stochastic Growth model are found in the "Extra Content" folder. |
| **R** | • Plotting is easier and cleaner using ggplot rather than the base functions, but I opted to use base plotting functions to avoid having to install packages.<br>• I encountered an issue with poor performance of the dot product function for small vectors. |
| **Julia** | • Julia benchmarks were run with the scrips found in the "Extra Content" folder. The main script is put in a main() function (which is then compiled) and passed to the benchmarking and performance profiling tools.<br>• Julia passes function arguments by reference rather than creating a copy which would be costly in this application. Array slicing, however, does create a copy. In critical lines of code, the @views macro is used to access a slice of an array without creating a copy.<br>• Julia allows for the enumerate() iterator in for-loops like Python, but this slowed down performance significantly. The issue is discussed here: https://github.com/JuliaLang/julia/issues/16190 |
| **Fortran** | • It was difficult to initialized variables that were unpacked from the parameter struct so I opted to just use them directly in expressions.<br>• Regular vs. allocatable arrays brought up a similar issue to static vs. dynamic arrays in C++. I opted for regular arrays except for the Value Function and Policy Function in the second model due to their large size. |
| **C++** | • This type of computation relied heavily on arrays and I am now confident that the std::vector array type is the correct choice for this application.<br>  ○ Std::vector arrays are dynamic (heap-allocated) arrays that are accessed in the same way as standard arrays, carry information such as their size, can have their size changed, and take care of allocating and de-allocating memory automatically.<br>  ○ They can be easily passed by reference (no costly copying) to structs and to functions.<br>  ○ They work with standard functions to have functionality like iterating over the array directly.<br>  ○ With the "using" and "template" functionality, you can create simple Array2D<double> and Array3D<double> classes. |

| | |
|---|---|
| | ◦ They carry a small amount of additional overhead due to the additional information stored in the object and the fact that the array is heap-allocated. This overhead should be negligible relative to bottlenecks like mathematical calculations.<br>• I also provide an implementation using 1-dimensional base arrays (which access multiple dimensions through array arithmetic). I don't think that this should provide a performance advantage over properly arrays build of std::vector.<br>◦ Slightly smaller overhead than std::vector.<br>◦ Automatically passed by reference, so stack-allocated arrays can still be used by structs and functions without templates.<br>◦ More suitable for adapting the code to run in C.<br>• C++ functions cannot return multiple values. Instead, outputs should be passed by reference and overwritten by the function as output. The implementation may be a bit clumsy, but is done to mirror the syntax in the other languages.<br>• Templates were used to provide a clean implementation of function overloading. |