MILLIMAN WHITE PAPER

Actuarial use cases, model development, and governance in Julia

Yun-Tien Lee, FSA, MAAA, Senior Data Scientist
Pranav Rupireddy, MS, Data Scientist
Mehdi Echchelh, Senior Consultant
Victor Morales, Software Engineer
Pierre Miehe, IA, CERA, Principal

Introduction

In an increasingly margin-aware and tightly regulated environment, (re)insurers, pension funds, and other financial institutions are re-evaluating the economics of their modeling infrastructure. Many firms, wary of rising public-cloud fees, data-sovereignty constraints, and the need to safeguard intellectual property, are gravitating toward high-performance systems that can run on local or private hardware. Others, however, are fully cloud-ready and prefer a no-code, turnkey platform. For these organizations, Milliman Mind delivers audited, production-grade models without writing a single line of code. Meanwhile, those prioritizing on-premise control must still escape the limits of the traditional toolchains—Excel workbooks, VBA macros, and high-level scripting languages—that compel an uncomfortable trade-off between developer productivity and the raw speed required for asset-liability monitoring or daily profit testing. Decision makers therefore face a pivotal question: Which technology can deliver C-level performance, transparent governance, and a gentle learning curve—without resorting to cloud clusters?

This paper positions Julia as a possible solution. In the initial installment of the Julia series, "An actuary's guide to Julia: Use cases and performance benchmarking in insurance," we identified some relevant and convincing use cases in the insurance field to compare productivity and efficiency between several languages, including Julia, Python, Mojo, and C#. We further demonstrate, via three additional representative use cases, how Julia's just-in-time (JIT) compilation, multiple dispatch, and deterministic package management translate into tangible in-house advantages. Collectively, these results empower financial institutions to build self-sufficient modeling infrastructures.

How Julia helps in the model development process

Julia is a modern programming language designed with scientific computing and data-intensive applications in mind, making it an excellent choice for model development. Its core design philosophy emphasizes high performance without sacrificing ease of use, allowing developers to move seamlessly from rapid prototyping to production deployment. Julia achieves this through JIT compilation via Low Level Virtual Machine (LLVM), a powerful open-source compiler infrastructure project designed to optimize code during both compilation and runtime. As a result, Julia can deliver performance that is competitive with C or Fortran for many numerically intensive workloads[1]. This means that complex models, such as those used in machine learning or numerical simulations, can often be implemented directly in Julia without the need to rewrite performance-critical sections in another language. Additionally, Julia's syntax is expressive and approachable, reducing the learning curve for researchers and scientists transitioning from other high-level languages.

One of Julia's most powerful core design ideas is multiple dispatch, which allows functions to be defined for different combinations of argument types. This is particularly valuable in model development, where different models or data structures may require specialized processing. Using multiple dispatch, developers can write generic algorithms that automatically adapt their behavior based on the types of inputs provided—whether they are matrices, sparse arrays, or custom model objects—without resorting to complex conditional logic. This results in cleaner, more maintainable code and makes it easy to extend models to new scenarios simply by defining additional method signatures.

1. Julia. (n.d.). Julia micro-benchmarks. Julia.org. Retrieved February 16, 2026, from https://julialang.org/benchmarks/

Beyond multiple dispatch, Julia incorporates other core design principles that enhance model development workflows. Its unified type system supports both concrete and abstract types, enabling polymorphism and generic programming while still allowing fine-grained performance tuning. Julia also embraces composability, meaning that packages and functions can be combined in flexible ways without compatibility issues, which is crucial when integrating different modeling tools and libraries. Furthermore, Julia's ability to handle metaprogramming and macros allows developers to generate optimized code dynamically, which can be invaluable for automating repetitive modeling tasks or adapting algorithms to specific hardware architectures. Together, these design elements make Julia a powerful and versatile tool for building, experimenting with, and refining models across a wide range of scientific and research domains.

Expanded use cases in insurance-related fields

In the initial installment of this series, we explored several practical use cases, specifically focusing on similarity calculations, reserve simulations, and sparse matrix decomposition. We examined these use cases within relevant contexts and applications where the Julia language can provide significant value, such as improving computational efficiency, enhancing accuracy in actuarial estimations, and enabling more effective handling of complex datasets. Each of these techniques has demonstrated Julia's ability to address challenges in data-driven decision making, particularly in actuarial modeling and analysis.

Building on that foundation, we now extend our analysis to another critical area for most insurance institutions—asset and liability models. These models play a pivotal role in ensuring that an organization can meet its future obligations while optimizing the performance of its assets under various economic conditions. It involves balancing risks, returns, and liquidity, often requiring sophisticated modeling techniques to account for uncertainties in interest rates, market fluctuations, and policyholder behaviors. This expansion will allow us to investigate how Julia and other popular languages can enhance resilience and performance in a dynamic model development process.

LANGUAGES USED FOR COMPARISON

We chose to benchmark performance results in Julia versus Python, Mojo, and Milliman Mind.

Python is a high-level, interpreted programming language known for its simplicity, readability, and versatility.[2] It has become one of the most widely used languages in fields ranging from web development and automation to data science, artificial intelligence, and scientific computing.[3] Python's extensive standard library, combined with a vast ecosystem of third-party packages, allows developers to quickly build and deploy robust applications. Its syntax emphasizes clarity, making it ideal for both beginners and experienced programmers. Additionally, Python's strong community support and integration with other languages and platforms make it a go-to choice for rapid prototyping and production-ready solutions alike. Despite these advantages

Python's interpreted nature can lead to slower execution times compared with compiled languages.[4] One practical approach to mitigating this performance gap is Numba, an open-source JIT compiler that translates a subset of Python and NumPy code into optimized machine code at runtime.[5] Developers can achieve substantial speedups—often approaching those of compiled languages—without rewriting their codebase. That said, Numba optimization is not compatible with common object-oriented programming techniques.[6] Numba's absence of memory-safety features can also lead to more complex debugging and could, in some cases, contribute to model inaccuracies.[7]

Mojo is a relatively new programming language designed to combine the usability of Python with the performance of low-level languages like C and C++.[8] Built for AI and high-performance computing workloads, Mojo aims to bridge the gap between research and production by offering Python-compatible syntax while enabling fine-grained control over hardware resources. It supports advanced features like compile-time metaprogramming, explicit memory management, and integration with modern accelerators such as GPUs and TPUs. This makes

2. Python. (n.d.). The Python language reference. Retrieved February 16, 2026, from https://docs.python.org/3/reference/index.html.
3. Stack Overflow. (n.d.). 2025 Developer Survey. Retrieved February 16, 2026, from https://survey.stackoverflow.co/2025/.
4. Dannenberg, R.B. (January 10, 2024). Making Python run fast. Retrieved February 16, 2026, from https://www.cs.cmu.edu/~rbd/blog/fast/fast-blog10jan2024.html.
5. Lam, S., Pitrou, A., & Seibert, S. (November 15, 2015). Numba: A LLVM-based Python JIT compiler. ACM Digital Library. Retrieved February 16, 2026, from https://dl.acm.org/doi/pdf/10.1145/2833157.2833162
6. Brackett J., Schuster, C., McGinley, D., Grigg, C., & Srivastava, K. (April 15, 2025). "Python and Numba: Techniques for maximizing acceleration. Milliman. Retrieved February 16, 2026, from https://www.milliman.com/en/insight/python-and-numba-techniques-maximizing-acceleration.
7. Deviations from Python semantics. (n.d.). Numba documentation. Retrieved February 16, 2026, from https://numba.readthedocs.io/en/stable/reference/pysemantics.html cf. https://juliahub.com/blog/us-national-security-standards-and-julias-memory-safe-capabilities
8. Modular. (n.d.). Mojo Manual. Retrieved February 16, 2026, from https://docs.modular.com/mojo/manual/.

Mojo particularly appealing for developers working on machine learning models, numerical simulations, and other computationally intensive tasks that require both developer productivity and execution speed. As the language evolves, it promises to become a powerful tool for next-generation AI and scientific computing applications.

Milliman Mind is a cloud-based modeling platform that enables organizations to develop, manage, and deploy complex financial and actuarial models.[9] One of its capabilities is the ability to upload existing Excel-based or Python-based models to a production environment with a full audit trail and a workflow, allowing businesses to transition from spreadsheet-driven or Python-driven processes to more robust, scalable, and maintainable applications. In the case of the Excel language, this conversion not only improves performance and reliability of existing spreadsheets but also enhances governance by reducing spreadsheet-related risks such as versioning issues and calculation errors.

USE CASE 1: OPTIMAL ASSET ALLOCATION UNDER CONSTRAINTS

The first use case builds upon the concept of optimal asset allocation, where the goal is to determine the best combination of assets in a portfolio to minimize duration mismatches while adhering to a set of market value constraints. In addition, this scenario introduces binary inclusion constraints, meaning each asset can only be either fully included in the portfolio (represented by 1) or completely excluded (represented by 0), with no partial allocations allowed. This transforms the problem into a binary or integer optimization challenge, which is often more complex than continuous allocation problems, as it requires finding optimal solutions from a combinatorial search space. Such constraints are particularly relevant in situations where assets represent indivisible investment opportunities, regulatory restrictions, or strategic mandates.

For all implementations in this use case, the HiGHS[10] library was used for the integer optimization task. HiGHS is a highly optimized C++ library that is widely used for integer optimization, so it is logical to leverage this library rather than re-implementing similar functionality in other languages.

Because all implementations relied on the same underlying library, the execution times across the different versions showed minimal variation. While additional general computations are performed outside of HiGHS, the shared dependency resulted in the core computational workload being handled identically, regardless of the programming language used, resulting in performance differences that were negligible. This highlights how, in certain scenarios, the choice of language may have little impact on speed when the heavy lifting is delegated to a common optimized library.

In this use case and others that follow, we ran through the Julia and Python codebases on the same virtual machine, which is configured with Windows 11, an Intel Core i9-9900 processor running at 3.6 GHz with eight physical cores, and 128 GB of RAM.

We also included results from Milliman Mind for this use case. However, it should be noted that the comparison is not strictly equivalent, as an actuary developed an Excel spreadsheet that replicates the logic implemented in the Julia and Python source code, and Milliman Mind subsequently converts and executes the model based on this spreadsheet.

We also noted that the optimized objective value of the asset allocation achieved with Python (Scipy[11]) is slightly lower. This discrepancy can be attributed to the fact that the version of the HiGHS library bundled with Scipy is older than the version utilized in Julia and Milliman Mind, which leads to the application of different heuristics

FIGURE 1: RUNTIME RESULTS (30 RUNS, NUMBERS IN SECONDS) FOR USE CASE 1 ON THE VIRTUAL MACHINE AND ON THE MIND ENVIRONMENT

ON THE VIRTUAL MACHINE (MIN/MEAN/MAX)	
Julia	(6.295/6.448/7.781)
Python	(7.546/7.624/7.917)

ON THE MILLIMAN MIND ENVIRONMENT	
Milliman Mind converted excel	8.500

9. Milliman. (n.d.). Milliman Mind. Retrieved February 16, 2026, from https://www.milliman.com/en/products/milliman-mind.

10. HiGHS. (n.d.). HiGHS: High-performance software for mathematical optimization. Retrieved February 16, 2026, from https://highs.dev/.

11. SciPy. (n.d.). Fundamental algorithms for scientific computing in Python. Retrieved February 16, 2026, from https://scipy.org/.

USE CASE 2: PROFIT MARGIN AND RESERVE INCREASE FOR A CERTAIN AMOUNT OF POLICIES

The second use case focuses on performing detailed calculations of the monthly increase in mathematical reserves and the gross profits for a portfolio containing varying numbers of life insurance policies. Mathematical reserves represent the amount an insurer must hold to meet future policyholder obligations, and accurately tracking their monthly changes is essential for ensuring financial stability and regulatory compliance. In this scenario, the process involves applying actuarial formulas and assumptions—such as mortality rates, interest rates, and expense loadings—to each policy and then aggregating the results to determine overall reserve movements. At the same time, gross profit calculations are carried out, factoring in premium income, claims, expenses, and reserve changes to assess the financial performance of the portfolio over time.

Upon completion of each run, the model exports its outputs into a two-dimensional, pure floating-point matrix. One dimension enumerates the model points, uniquely representing an individual policyholder or contract, while the other dimension enumerates the state variables captured at the conclusion of the projection horizon.

For illustrative purposes, to name a few, the seventh index corresponds to the model point's attained age, the 44th index records the change in mathematical reserves, the 53rd index reports the gross profit, and the remaining indices correspond to additional policy-specific attributes. While the matrix maintains homogenous data types in floating points, enabling direct handoff to high-performance numerical libraries without preprocessing, certain index-related attributes (e.g., policy year) must be stored as floating-point values and then manually converted to integers.

For the Julia and Python versions of the model, we evaluate the execution speed and resource utilization when the calculations are performed seriatim (processing policies one by one in sequence) or in parallel (distributing the workload across multiple CPU cores or threads). These comparisons not only highlight the computational efficiency of each language under different execution strategies but also provide insights into how well each environment handles large-scale data processing and numerical computation

Since by default Julia stores arrays in column-major order and Python stores them in row-major order, we aligned the benchmarking by iterating over the first index in the inner loop of the Julia model and over the second index in the inner loop of the Python model. The following tables present seriatim and parallel results (in seconds) for different programming languages when executed on the virtual machine with policy counts of 1 million, 5 million, and 10 million, respectively.

FIGURE 2: RUNTIME RESULTS (30 RUNS, NUMBERS IN SECONDS (30 RUNS, NUMBERS IN SECONDS) FOR USE CASE 2 WITH DIFFERENT POLICY COUNTS ON THE VIRTUAL MACHINE

SERIATIM (MIN/MEAN/MAX)		
Julia	1M	(76.109/76.828/79.044)
	5M	(377.159/379.942/385.486)
	10M	(758.452/761.729/777.030)
Python (NUMBA)	1M	(76.923/77.387/79.731)
	5M	(385.377/386.977/389.826)
	10M	(777.999/781.269/788.460)
Python	1M	(10,339.019/10,393.485/10,756.279)
	5M	(51,582.830/51,682.969/52,138.539)
	10M	(103,509.211/103,813.915/104,417.246)

PARALLEL (MIN/MEAN/MAX)		
Julia	1M	(11.346/11.677/13.087)
	5M	(56.546/58.190/59.767)
	10M	(116.593/117.741/119.954)
Python (NUMBA)	1M	(11.688/11.874/12.143)
	5M	(58.445/60.219/67.279)
	10M	(117.090/119.090/127.735)

The results clearly demonstrate that a well-designed Julia program can achieve state-of-the-art performance.

We again use the same conversion process as in use case 1 to obtain the results for Milliman Mind. The tables below present the results (in seconds) for running VBA on the model on a virtual machine, and running the converted model on Milliman Mind environment, respectively.

FIGURE 3: RUNTIME RESULTS IN SECONDS FOR THE EXCEL MODEL WITH DIFFERENT POLICY COUNS USING VBA ON THE VIRTUAL MACHINE AND ON THE MIND ENVIRONMENT (WITH HPC)

EXCEL MODEL	100K	250K	1M
VBA	20,694.220	51,355.164	205,302.500

EXCEL MODEL	100K	250K	1M
Milliman Mind Converted Model	34.000	85.000	340.000

If Excel language is a must, Excel/VBA shows too poor performance to be usable. However, Milliman Mind enables compensating for the speed issue without coding/architecture skills (yet involving cloud costs).

If actuaries are at ease with (re)coding the model in Julia and have enough IT skills, they would benefit from the increased computational speed without too much hardware/cloud cost, and still with full memory safety compared to Numba.

We also include a comparison among the Julia, Python, and Mojo versions of the model on a GPU (leveraging massively parallel processing capabilities of graphics processors). The comparison results were from a Runpod server[12] (Ubuntu 24, 16 virtual AMD EPYC 9354 processors running at 3.8 GHz, 140 GB of RAM, and NVIDIA GeForce RTX 5090 32GB of memory). The Julia and Python codebases were converted in a straightforward and direct manner, whereas the adaptation of the Mojo codebase required a more complex approach due to its more involved and evolving syntax. This exercise also highlights potential opportunities for further enhancements in GPU codebase.

FIGURE 4: RUNTIME RESULTS IN SECONDS FOR USE CASE 2 ON THE RUNPOD SERVER

GPU	100K	250K	1M
Julia	0.429	2.074	4.127
Python (NUMBA CUDA)	0.783	3.915	7.825
Mojo	0.221	1.103	2.207

CPU VERSUS GPU

Actuarial computations have traditionally favored CPUs over GPUs, primarily because the structure of insurance valuation and projection tasks often conflicts with the parallel, SIMD-based architecture of GPUs. Many actuarial models exhibit strong sequential dependencies, wherein calculations for each period rely on the outcomes of preceding periods. This characteristic inherently limits the degree of parallelism achievable on GPU platforms, while multicore CPUs with advanced pipelines and substantial cache memory are well suited to these workloads. Nonetheless, for certain well-structured actuarial models and problem types, the application of GPU technology can yield a substantial reduction in runtime compared to conventional CPU-based approaches. Although we generated the results on various machine configurations, these findings underscore the significant potential for accelerating select actuarial computations through judicious use of GPU resources. It is important to note, however, that such performance gains are contingent upon the suitability of the model to GPU architectures and are not universally attainable across all actuarial applications.[13]

USE CASE 3: A BASIC ASSET-LIABILITY MANAGEMENT EXERCISE

The third use case focuses on setting up a basic asset-liability management (ALM) exercise, a core practice in financial risk management for insurers, pension funds, and other institutions with long-term obligations. In this scenario, the institution's future payment obligations are modeled as aggregate liability cash flows represented by a single representative policy, whose lapse rates adjust dynamically in response to asset returns. The task then entails a continuous assessment of both the market value and projected cash flows for each asset class, such as fixed-income securities, equities, and money market instruments, and the development of a rebalancing framework that aligns those assets with the timing and magnitude of future liability payments. The rebalancing process follows a predefined asset allocation strategy, which may be based on risk tolerance, regulatory requirements, or strategic investment guidelines. This setup allows for the simulation of portfolio adjustments in response to liability schedules, market movements, and changes in asset values, providing insights into liquidity needs, funding ratios, and potential shortfalls. For the Julia and Python versions of the model, we also conduct a comparative performance analysis either seriatim or parallel looping through various interest rate scenarios.

MODELING ASSET TYPES

By leveraging multiple dispatch in Julia, each asset type—such as cash, bonds, or equities—can have its own specialized methods for operations like cash flow calculation, valuation, or risk assessment. This approach eliminates the need for cumbersome conditional statements and type checks, making the codebase cleaner and more maintainable. As new asset types or behaviors are introduced, they can be seamlessly integrated by simply defining new structs and extending existing functions without altering or risking the integrity of established logic. This ensures the model remains robust and adaptable to evolving financial instruments, while also benefiting from Julia’s performance optimizations through type-stable, method-specific code paths. On the other hand, it is important to note that multiple dispatch can introduce performance trade-offs, particularly when the number of method combinations grows large, potentially increasing method lookup times and compilation overhead. Careful design and benchmarking may be necessary to balance code flexibility with optimal performance. We include an example of multiple dispatch in the appendix (Figure 7).

12. Runpod. (n.d.). Retrieved February 16, 2026, from https://console.runpod.io/.

13. Maciejewski, K., Schuster, C., Brackett, J., & Grigg, C. (February 4, 2026). GPU: Exploring use cases in actuarial modeling. Milliman. Retrieved February 16, 2026, from https://www.milliman.com/en/insight/gpu-use-cases-actuarial-modeling.

A NOTE ON MIXED DATA TYPES

Before passing any data into an @njit function (a Numba decorator that compiles Python functions to fast native machine code using LLVM, removing almost all Python overhead) in Numba, developers must carefully ensure that all arrays use concrete numeric data types—such as float64 or int64—because Numba is unable to compile functions that receive lists of lists, object arrays, or heterogeneous data structures. These limitations often require substantial preprocessing and manual type sanitation, especially in asset-modeling workflows where portfolios may contain nested structures or mixed asset types. In fact, arrays of asset classes are converted to a pure float64 array in the Python codebase in the runtime result comparison shown in Figure 5. In contrast, Julia's type system and array semantics provide far greater resilience in these situations: The language's parametric, compile-time types enable heterogeneous but strongly typed containers. As a result, practitioners can pass vectors of custom structs (e.g., equity, bond, or even custom policy-info struct[14]) directly into actuarial valuation kernels without an intermediate homogenization phase. Julia's compiler generates efficient, type-specialized machine code for each variant, yielding performance comparable to hand-tuned C while preserving the expressiveness of high-level abstractions. Consequently, the end-to-end workflow is streamlined—data ingestion, transformation, and simulation occur with minimal boilerplate—reducing both development effort and the risk of type-related runtime errors.

FIGURE 5: RUNTIME RESULTS 30 RUNS, NUMBERS IN SECONDS) FOR USE CASE 3 ON 1 MILLION SCENARIOS ON THE VIRTUAL MACHINE

SERIATIM	
Julia	(7.584/7.719/8.411)
Python (NUMBA)	(10.136/10.429/14.110)
Python	(664.529/667.002/672.784)
PARALLEL (MIN/MEAN/MAX)	
Julia	(2.219/2.370/2.532)
Python (NUMBA)	(6.107/6.495/11.382)

FIGURE 6: RUNTIME RESULTS IN SECONDS FOR USE CASE 3 ON 1 MILLION SCENARIOS ON THE VIRTUAL MACHINE AND ON THE MIND ENVIRONMENT (WITHOUT HPC)

EXCEL MODEL	
Excel VBA	2,860.938
ON THE MILLIMAN MIND ENVIRONMENT	
Milliman Mind Converted Excel	450.000

We again use the same conversion process as in use case 1 to obtain the results for Milliman Mind. It outperforms Excel VBA as expected.

Once again, the results clearly demonstrate that a well-designed Julia program can achieve state-of-the-art performance.

Model governance, package management, and testing

Julia provides strong capabilities that can support model governance, which is the process of managing, monitoring, and controlling models throughout their life cycle to ensure compliance, reliability, and transparency. It offers various built-in tools to make model governance easier. One key feature is its environment management system, which allows developers to create project-specific environments with clearly defined dependencies. This ensures that models can be reproduced exactly, even years later, by capturing the exact versions of packages and libraries used during development. Combined with Julia's emphasis on clear and expressive code, this makes it easier to audit, document, and validate models for regulatory or organizational compliance.

In terms of package management—the process of installing, upgrading, configuring, and removing software components—Julia uses the built-in Pkg system, which is both powerful and user-friendly. Pkg allows developers to easily add, update, and manage packages while maintaining isolated environments for different projects. This isolation prevents dependency conflicts and ensures that changes in one project do not inadvertently affect another. For model governance, this means that a model's dependencies are explicitly tracked in the default Project.toml and Manifest.toml files, enabling precise control over the

14. Loudenback, A. (February 11, 2026). Stochastic mortality projections. Modern Financial Modeling. Retrieved February 16, 2026, from https://modernfinancialmodeling.com/stochastic-mortality.

A reference implementation of Use Case 2, constructed in accordance with these recommended practices, is also available in the accompanying GitHub repository.

software stack used for training, testing, and deployment. This level of transparency supports version control, reproducibility, and auditing, which are essential for regulated industries such as finance and healthcare. In contrast, Python typically relies on external tools such as virtualenv, conda, or uv to create isolated environments. Environment activation and management require additional commands and tooling outside the core Python language.

Furthermore, Julia’s package ecosystem encourages modularity and composability, making it straightforward to integrate governance-related tools—such as logging systems, monitoring frameworks, and compliance checkers—directly into the model development workflow. Because Julia's package installation and environment setup are deterministic, organizations can deploy models confidently across different environments without worrying about dependency drift. This combination of reproducibility, transparency, and flexibility makes Julia an excellent choice for teams that need to maintain rigorous model governance while benefiting from modern package management practices.

With regard to testing, Julia distinguishes itself among programming languages through its advanced metaprogramming and performance testing capabilities, which are particularly advantageous for scientific and technical computing. Julia’s metaprogramming facilities allow developers to generate and manipulate code dynamically, enabling the creation of highly flexible and reusable testing frameworks. This approach facilitates the automated generation of test cases, especially in domains where exhaustive or parameterized testing is required, thus enhancing test coverage and reliability. Furthermore, Julia’s focus on high performance is complemented by robust benchmarking tools, such as the Benchmark Tools package, which enable precise measurement and analysis of code execution times. This empowers researchers and developers to identify performance bottlenecks and optimize algorithms effectively, ensuring that tested solutions not only produce correct results but also meet the stringent performance demands of modern computational applications. Collectively, these features position Julia as a powerful environment for both functional and performance-oriented testing in research and industry settings.

Conclusion

This research demonstrates that Julia is a powerful and efficient language for actuarial and financial modeling, offering significant advantages in speed, maintainability, and flexibility. Through benchmarking across key insurance use cases, Julia consistently delivered state-of-the-art performance, often surpassing Python/Numba. Conversely, Milliman Mind offers very attractive no-code performance for organizations that are not building in-house coding capability. In addition, Julia’s modern package management, environment isolation, and advanced testing tools further support strong model governance and reproducibility—key needs in regulated industries.

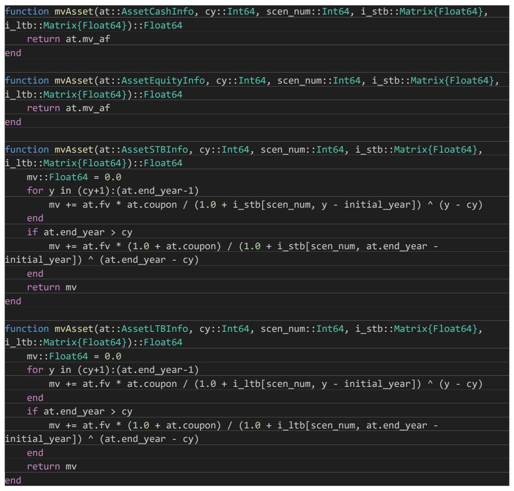
In summary, Julia’s blend of usability, speed, and modern development practices makes it a compelling choice for advanced actuarial and financial modeling in insurance and financial industry. (Re)insurers and pension funds should consider piloting Julia for critical actuarial, reserving,and ALM models.

Appendix

Following is a comparison of Julia and Python code implementing market value calculation. The Julia code follows the multiple dispatch approach. Using multiple dispatch, Julia can express complex conditional logic instead of using separate functions for each data type. This results in a cleaner and more robust codebase.

FIGURE 7: MULTIPLE DISPATCH JULIA AND PYTHON COUNTERPART

JULIA



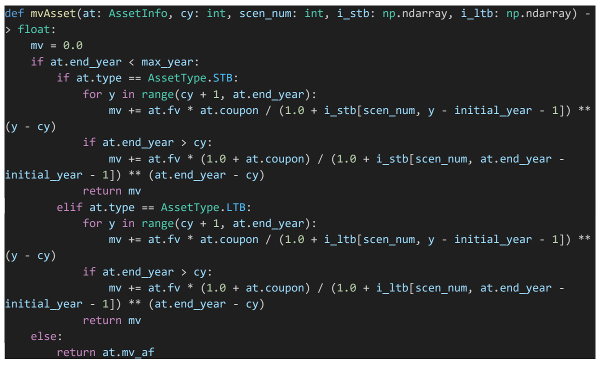
```
function mvAsset(at::AssetCashInfo, cy::Int64, scen_num::Int64, i_stb::Matrix{Float64},
i_ltb::Matrix{Float64})::Float64
    return at.mv_af
end

function mvAsset(at::AssetEquityInfo, cy::Int64, scen_num::Int64, i_stb::Matrix{Float64},
i_ltb::Matrix{Float64})::Float64
    return at.mv_af
end

function mvAsset(at::AssetSTBInfo, cy::Int64, scen_num::Int64, i_stb::Matrix{Float64},
i_ltb::Matrix{Float64})::Float64
    mv::Float64 = 0.0
    for y in (cy+1):(at.end_year-1)
        mv += at.fv * at.coupon / (1.0 + i_stb[scen_num, y - initial_year]) ^ (y - cy)
    end
    if at.end_year > cy
        mv += at.fv * (1.0 + at.coupon) / (1.0 + i_stb[scen_num, at.end_year -
initial_year]) ^ (at.end_year - cy)
    end
    return mv
end

function mvAsset(at::AssetLTBInfo, cy::Int64, scen_num::Int64, i_stb::Matrix{Float64},
i_ltb::Matrix{Float64})::Float64
    mv::Float64 = 0.0
    for y in (cy+1):(at.end_year-1)
        mv += at.fv * at.coupon / (1.0 + i_ltb[scen_num, y - initial_year]) ^ (y - cy)
    end
    if at.end_year > cy
        mv += at.fv * (1.0 + at.coupon) / (1.0 + i_ltb[scen_num, at.end_year -
initial_year]) ^ (at.end_year - cy)
    end
    return mv
end
```

PYTHON



```
def mvAsset(at: AssetInfo, cy: int, scen_num: int, i_stb: np.ndarray, i_ltb: np.ndarray) -
> float:
    mv = 0.0
    if at.end_year < max_year:
        if at.type == AssetType.STB:
            for y in range(cy + 1, at.end_year):
                mv += at.fv * at.coupon / (1.0 + i_stb[scen_num, y - initial_year - 1]) **
(y - cy)
            if at.end_year > cy:
                mv += at.fv * (1.0 + at.coupon) / (1.0 + i_stb[scen_num, at.end_year -
initial_year - 1]) ** (at.end_year - cy)
            return mv
        elif at.type == AssetType.LTB:
            for y in range(cy + 1, at.end_year):
                mv += at.fv * at.coupon / (1.0 + i_ltb[scen_num, y - initial_year - 1]) **
(y - cy)
            if at.end_year > cy:
                mv += at.fv * (1.0 + at.coupon) / (1.0 + i_ltb[scen_num, at.end_year -
initial_year - 1]) ** (at.end_year - cy)
            return mv
    else:
        return at.mv_af
```

If you are interested in obtaining access to the code, programs, or datasets used in this research, please contact the authors directly. Contact information is provided in the following contact section. Subject to applicable data-sharing policies, licensing restrictions, and ethical considerations, the authors will make reasonable efforts to provide the requested materials to support transparency, reproducibility, and further research ideas.

CONTACT

Yun-Tien Lee
yuntien.lee@milliman.com

Pranav Rupireddy
pranav.rupireddy@milliman.com

Mehdi Echchelh
mehdi.echchelh@milliman.com

Victor Morales
victor.morales@milliman.com

Pierre Miehe
pierre.miehe@milliman.com