MILLIMAN WHITE PAPER

An actuary's guide to Julia: Use cases and performance benchmarking in insurance

Yun-Tien Lee, FSA, MAAA, Senior Data Scientist
Victor Morales, Software Engineer
Jim Brackett, Principal and Senior Director of Financial Technology
Joe Long, ASA, MAAA, Consulting Actuary and Data Scientist
Tom Peplow, Principal and Senior Director of Technology Strategy

Milliman

WHAT IS JULIA?

Julia is a general-purpose open-source programming language that debuted in 2012, prioritizing ease of use and execution efficiency. It blends a variety of innovative programming language design concepts and is licensed under the Massachusetts Institute of Technology (MIT) license. The journal Nature fittingly introduces it as, "Come for the Syntax, Stay for the Speed." Over the years, there has been considerable growth in Julia's library ecosystem and community. For further information, visit https://julialang.org.

In this paper, we will explore Julia's potential in the insurance industry by contrasting its capabilities with other popular languages, including Python, Rust, .NET (C#), and C++, focusing on development efficiency and processing performance. We will also introduce Mojo, a new programming language released in 2023, designed to combine the usability of Python with the performance of C.

It is important to note that our discussion does not encompass the statistical programming language R. This omission should not be interpreted as a diminishment of R's value or the utility of other programming languages for actuaries. Our focus is predominantly on high-performance numerical computing and general-purpose programming languages. While R is indeed a powerful tool used across the actuarial and other professions to rapidly prototype new ideas, it is predominantly recognized for its prowess in statistical analysis and data visualization, rather than as a high-performance numerical computing and general-purpose programming language.

Throughout, we will highlight challenges associated with these mentioned languages, offering insights into their optimal applications. The primary objective of this paper is to present Julia as a potential language that actuaries and other professionals can leverage to help balance the trade-offs between development and run-time costs when creating new products and tools. Additionally, we highlight how Julia can alleviate the common frustrations users face associated with managing multiple third-party package dependencies, as seen in other languages like Python. A significant portion of Julia's packages are written in native Julia, facilitating seamless integration with the language's features. This native development not only enhances performance but also simplifies package management and dependency resolution, making it an efficient choice for a variety of projects.

WHAT ARE JULIA'S CORE CONCEPTS AND DATA ANALYTICS PACKAGES AND TOOLS?

One of the original design concepts of Julia is to tackle the two-language problem, where many data analytics applications are prototyped in a slow yet easy-to-use language in order to quickly test an idea and later on moved to a faster language to boost performance. Julia comes with simple syntax and handy tools to allow users to implement ideas or concepts in a quick manner but it can still optimize code execution using the most up-to-date compiler mechanism. It supports Unicode, which makes it straightforward to translate formula into code if necessary, and the core design idea of multiple dispatch allows effortless extension and collaboration. In fact, users are even allowed to inspect lower-level compiler-generated representations by invoking simple macro calls. It also offers simple internal-environment management of packages.

In terms of data analytics tools, Julia offers a wide variety of notebook utilities and data manipulation and visualization packages. It also supports reading and putting data in Excel format and even doing data analyses in an Excel environment, as does Python.[i] Here are some common data manipulation procedures in both Julia and Python. Specifically, the first exhibit shows procedures to:

- Compute number of occurrences of specific values in a certain data table
- Get unique values from a certain data table
- Calculate a specific quantile value from a data table
- Extract a specific range of values from a data table
- Filter out specific values from a data table

JULIA (DATAFRAMES)

```
using DataFrames, Statistics, Query

n = 100_000_000
df = DataFrame(x = rand(1:10000, n))


count(==(100), df.x) # number of 100's
unique!(df.x) # get all unique numbers
quantile!(df.x, 0.25) # the first quantile
df.x[101:200] # get 101st to 200th elements
filter(x -> x > 100, df.x) # get all
values > 100
```

PYTHON (PANDAS)

```
import numpy as np, pandas as pd

n = 100_000_000
df = pd.DataFrame({'x':
np.random.choice(range(1, 10000), size = n)})

df.x.value_counts()[100] # number of 100's
df.x.drop_duplicates() # get all unique numbers
np.quantile(df.x, 0.25) # the first quantile
df.x[100:200] # get 101st to 200th elements
list(filter(lambda x: x > 100, df.x)) # get all
values > 100
```

Both Julia and Python offer a diverse array of data manipulation packages, each tailored to specific applications. For the comparisons above and below we have chosen to focus on DataFrames in Julia and Pandas in Python, as they share similar applications and are both known for their user-friendly interfaces. The DataFrames package in Julia, developed natively, capitalizes on Julia's strengths for seamless integration and optimized performance. It is particularly acclaimed for its efficient columnar data storage, which enhances cache locality. Additionally, DataFrames can be effectively used in conjunction with native Julia modules like Threads and Distributed, enabling robust parallel and multithreaded data processing capabilities. In contrast, Pandas in Python is primarily designed for single-threaded operations by default. While parallelism can be achieved with Pandas through additional packages such as Dask, multiprocessing, or by opting for alternative libraries like Polars – which is inherently designed for multi-threaded data processing – these approaches typically require more setup and configuration in contrast to the more inherent parallel capabilities within the Julia ecosystem.

The second exhibit shows procedures to group by on a certain key value of a data table, as well as to join two data tables on a specific key value in common ways, including inner, outer, left, and anti.

JULIA (DATAFRAMES)

```
n = 1_000_000

df_1 = DataFrame(x = rand(1:10000, n), y = rand(n))

df_2 = DataFrame(x = rand(5000:15000, n), z = rand(n))


df_3 = innerjoin(df_1, df_2, on = :x) # inner join of df_1 and df_2
df_4 = outerjoin(df_1, df_2, on = :x) # outer join of df_1 and df_2
df_5 = leftjoin(df_1, df_2, on = :x) # left join of df_1 and df_2
df_6 = antijoin(df_1, df_2, on = :x) # anti join of df_1 and df_2



combine(groupby(df_1, :x), :y => mean) # get average of y within all groups of x
```

PYTHON (PANDAS)

```
n = 1_000_000

df_1 = pd.DataFrame({'x': np.random.choice(range(1, 10000), size = n), 'y': np.random.rand(n)})
df_2 = pd.DataFrame({'x': np.random.choice(range(5000, 15000), size = n), 'z': np.random.rand(n)})

df_3 = df_1.merge(df_2, on = 'x', how = 'inner') # inner join of df_1 and df_2
df_4 = df_1.merge(df_2, on = 'x', how = 'outer') # outer join of df_1 and df_2
df_5 = df_1.merge(df_2, on = 'x', how = 'left') # left join of df_1 and df_2
df_4 = df_1.merge(df_2, on = 'x', how = 'outer', indicator = True)
df_6 = df_4[(df_4._merge == 'left_only')].drop('_merge', axis = 1) # anti join of df_1 and df_2

df_1.groupby('x').y.mean() # get average of y within all groups of x
```

In terms of common machine learning libraries, Julia also offers a wide range of them. Consider the usage of some common machine learning libraries in both Julia and Python.

JULIA

```julia
using GLM, DecisionTree, Flux



# generalized linear regression, given a
data frame (df) with columns ["x", "y"]
model = lm(@formula(y ~ x), df)

# random forest, given a data frame (df)
with columns ["target", "fea1", "fea2"]
model = build_forest(Vector(df[:,
"target"]), Matrix(df[:, ["fea1",
"fea2"]]))

# neural network
model = Chain(
 Dense(13 => 3),
 BatchNorm(3),
 Dense(3 => 1),
 relu) |> gpu
```

PYTHON

```python
import statsmodels.api as sm,
statsmodels.formula.api as smf
import sklearn.ensemble, torch.nn

# generalized linear regression, given a data
frame (df) with columns ["x", "y"]
model = smf.glm(formula = 'y ~ x', data =
df).fit()
# random forest, given a data frame (df) with
columns ["target", "fea1", "fea2"]
model =
ensemble.RandomForestRegressor().fit(df.drop("t
arget"), df[["target"]])

# neural network
model = nn.Sequential(
 nn.Linear(13, 3),
 nn.BatchNorm2d(3),
 nn.Linear(3, 1),
 nn.ReLU).to(device('cuda:0'))
```

The above comparisons show a high degree of similarity between the two languages and how easy it is to do data processing and modeling with them.

USE CASES FOR JULIA IN INSURANCE RELATED FIELDS

The insurance field is a complicated world with so many diverse factors intertwined when it comes to modeling in data science-related approaches. However, the whole process can generally be classified into three marketing phases—before, during, and after sale.[ii] Based on our experience and past client projects, we have identified several relevant models below to focus on for comparisons between various languages.

At the time of writing this paper in 2023, Mojo as a new language has gained traction because it combines the usability of Python with the performance of C, unlocking the performance of Python by adding features including parallelization and vectorization. We have made an initial attempt to implement the use cases in Mojo and will also benchmark its performance on one of the virtual machines. However, at the time of writing, it only provides a subset of the syntax included in Python. Therefore, we only implemented two use cases in Mojo to illustrate its syntax and capabilities.

SIMILARITY CALCULATION

Applications like risk segmentation, customer classification, and fraud detection are typically modeled using unsupervised learning approaches on structured data, where the distance between a pair of records is determined from a similarity measure. Data fields can generally be categorized into two types, categorical and numerical. In the case of categorical data fields, they are usually transformed into one-hot encoded format, where all field values are binary, i.e., either 0 or 1. After the transformation, one record becomes essentially a series of bits, or bit arrays. It is much more efficient, in terms of both time and space efficiency, if those bits can be arranged together into a series of contiguous bytes, or byte arrays. Julia offers very handy primitive data types like BitVector, which allows easy generation or conversion of bit arrays into byte arrays, and users can still do bitwise operations on converted arrays, which not only results in greater usability but it also hugely boosts performance.

We highlight this below, in both Julia and Python, by randomly generating two bit arrays, which can also be converted from categorical fields of two records in a real dataset.

JULIA

```julia
n = 100_000_000
x = BitVector(rand(Bool, n))
y = BitVector(rand(Bool, n))
```

PYTHON

```python
n = 100_000_000
x = np.random.choice([0, 1], size = n)
y = np.random.choice([0, 1], size = n)
```

In most applications we are interested in finding out the degree of overlap between two records, or similarity measures, which in the case of one-hot encoded arrays can be directly translated into the total number of 1s after bitwise-and operations. Besides, Julia offers handy benchmarking tools that allow users to find out execution time of code sections in a very intuitive and straightforward manner, shown in the graphic below. From the statistics on a certain virtual machine we can see the median time of doing several runs is around 3.6 milliseconds (ms). On the other hand, we can do similar operations in NumPy, one of the most powerful computational packages, or a special-purposed BitVector library in Python. It is obvious that the performance of similar operations in Python is clearly not impressive. Windows 11 Enterprise is installed on a virtual machine with 12th Gen Intel Core i7-1270P 2.2 megahertz (MHz) with 12 physical and 16 virtual cores and 32 gigabytes (GB) of memory, and the same virtual machine applies through all benchmarking results in this paper (excluding the appendix).

JULIA

```
begin
    n = 100_000_000
    x = BitVector(rand(Bool, n))
    y = BitVector(rand(Bool, n))
end

BenchmarkTools.Trial: 1159 samples with 1 evaluation.
 Range (min … max):  3.180 ms … 8.036 ms  ┊ GC (min … max):  0.00% … 49.44%
 Time  (median):     3.575 ms             ┊ GC (median):     0.00%
 Time  (mean ± σ):   4.299 ms ± 1.339 ms  ┊ GC (mean ± σ):  15.59% ± 17.99%

 3.18 ms         Histogram: frequency by time        7.24 ms <

Memory estimate: 11.92 MiB, allocs estimate: 6.
@benchmark sum(x .& y)
```

PYTHON

```
from datetime import datetime
from time import time
import numpy as np
n = 100_000_000; x = np.random.choice([0, 1], n); y = np.random.choice([0, 1], n)
begin_time = datetime.now()
np.dot(x, y)
end_time = datetime.now()
print((end_time - begin_time).total_seconds() * 1000, "ms")
✓ 1.4s

61.148 ms
```

```
from datetime import datetime
from BitVector import BitVector
n = 100_000_000; x = util.urandom(n); y =util.urandom(n)
begin_time = datetime.utcnow()
BitVector.count_bits(x & y)
end_time = datetime.utcnow()
print((end_time - begin_time).total_seconds() * 1000, "ms")
✓ 0.7s

754.638 ms
```

We can do even better by using functions in built-in libraries in Julia, for example LinearAlgebra. From the statistics we can see that the median time of doing several runs of the same operation, by taking advantage of parallel instruction processing in the processors, is around 0.9 ms. On the other hand, a specially purposed library, bit array, must be used to seek higher performance in Python. Note two major differences here:

- The built-in library has not taken parallelism into account. Users must find ways to do it in parallel, which can be a burdensome process.
- The random number generator interface is different in each library, cf, a unified name and structure in Julia, which would lead to better code readability and maintenance.

If we compare the performance between Julia LinearAlgebra and Python bit array, the boost multiplier in performance becomes approximately nine times. The boost multiplier is defined as the run-time ratio between two languages, which in the current case is approximately $9 / 1 \approx 9$.

JULIA

```
begin
    n = 100_000_000
    x = BitVector(rand(Bool, n))
    y = BitVector(rand(Bool, n))
end

BenchmarkTools.Trial: 4977 samples with 1 evaluation.
 Range (min … max):  802.800 µs …   2.069 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     914.800 µs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   994.872 µs ± 199.731 µs  ┊ GC (mean ± σ):  0.00% ± 0.00%

 803 µs           Histogram: frequency by time         1.57 ms <

Memory estimate: 16 bytes, allocs estimate: 1.
@benchmark x • y # equivalent to LinearAlgebra.dot(x, y)
```

PYTHON

```
from datetime import datetime
from bitarray import bitarray, util
n = 100_000_000; x = util.urandom(n); y =util.urandom(n)
begin_time = datetime.utcnow()
bitarray.count(x & y)
end_time = datetime.utcnow()
print((end_time - begin_time).total_seconds() * 1000, "ms")
✓ 0.0s

9.015 ms
```

On the other hand, vectorization functionalities available in Mojo can be used to speed up the calculation while maintaining the simple syntax like Python. After compilation and execution on the same machine, the boost multiplier over Python is approximately 9 / 2 ≈ 4.5.

MOJO

```
    alias simd_width_u8 = simdwidthof[DType.uint8]()
    var sum = 0
    let begin_time = now()
    @parameter
    fn cnt[simd_width: Int](offset: Int):
        sum += (a.simd_load[simd_width](offset) & b.simd_load[simd_width](offset)).reduce_add().to_int()
    vectorize[simd_width_u8, cnt](n)
    let end_time = now()
    let execution_time_sequential = Float64((end_time - begin_time))
    print(execution_time_sequential / 1000000, "ms")
```

```
root@MPLSH-T14S-YL:/# ./similarity
2.0502720000000001 ms
```

We can go a step further in customizing calculations by taking advantage of the parallel processing available on almost all modern central processing unit (CPU) architectures, which we have used to implement the calculation procedure in C#, C++, and Rust. Although it has been actively advised not to use C++ anymore because of security risks,[iii] we chose to include it here because it is still considered the fastest language. It is useful to compare the complexity incurred of using C++ directly over the other languages. Many of the challenges in the C++ solution come from complexity of the developer ecosystem, which are purposefully solved for in the other languages. It is important to highlight that much of the optimized Python libraries wrap C++ native code, whereas Python could be viewed as a domain-specific language (DSL) for invoking C++. Although our C++ implementations do not excel, typically teams are more productive using other languages without needing the reach for C++ as a tool. Please refer to the appendix for all benchmarking results.

On the other hand, in terms of ease of use and development time, it often takes a significant amount of time, compared to Julia or Python, to get working codebases in these general-purpose languages, even for experienced programmers. Take the C# codebase, for example. We would typically need to build up a program structure available in the Parallel library to do parallel processing. A similar structure has to be set up in Rust. Moreover, because pure C++ does not provide utilities to count the number of 1s in a memory unit (PopCount in C# codebase), we would need to either implement customized functions or import proper libraries to achieve that, which would translate to higher development costs in general.

C#

```
[Benchmark]
[MethodImpl(MethodImplOptions.AggressiveInlining)]
0 references
public int cnt_ParForEach()
{
    int countOnes = 0;
    Partitioner<Tuple<int, int>> partitioner = Partitioner.Create(0, n);
    Parallel.ForEach(partitioner, range =>
    {
        int localCountOnes = 0;
        var xs = x.AsSpan();
        var ys = y.AsSpan();
        for (int i = range.Item1; i < range.Item2; i++)
        {
            localCountOnes += int.PopCount(xs[i] & ys[i]);
        }

        Interlocked.Add(ref countOnes, localCountOnes);
    });
    return countOnes;
}
```

```
// AfterAll
// Benchmark Process 26592 has exited with code 0.

Mean = 629.037 us, StdErr = 3.161 us (0.50%), N = 19, StdDev = 13.780 us
Min = 613.447 us, Q1 = 619.771 us, Median = 625.106 us, Q3 = 634.403 us, Max = 661.460 us
IQR = 14.632 us, LowerFence = 597.824 us, UpperFence = 656.350 us
ConfidenceInterval = [616.640 us; 641.434 us] (CI 99.9%), Margin = 12.397 us (1.97% of Mean)
Skewness = 1.05, Kurtosis = 3.13, MValue = 2
```

We can do similar benchmarking for numerical data fields. Here we randomly generate two numerical arrays, which can also be converted from numerical fields of two records in a real dataset. Because both use the optimized Basic Linear Algebra Subprograms (BLAS) library,[iv] the performance boost is not material in this case.

JULIA

```
n = 100_000_000
g = rand(n)
h = rand(n)
@btime g ⋅ h # equivalent to LinearAlgebra.dot(g, h)
```

✓ 11.5s

```
  31.055 ms (1 allocation: 16 bytes)
```

PYTHON

```
from datetime import datetime
import numpy as np
n = 100_000_000; g = np.random.rand(n); h = np.random.rand(n)
begin_time = datetime.utcnow()
np.dot(g, h)
end_time = datetime.utcnow()
print((end_time - begin_time).total_seconds() * 1000, "ms")
```

✓ 1.4s

```
36.309 ms
```

There are several ways to speed up the Python codebase. For example, using additional settings in Cython it is straightforward to convert Python to C implementations and speed up the codebase. However, because it involves an additional layer of translation between codebases, the performance boost is not as high as if the codebase were implemented in a more efficient language, as expected. Please refer to the appendix for benchmarking results from different languages on certain virtual machines and standardized machines using workflow functions in GitHub Actions.

Using graphic processing units (GPUs) is another common way to speed up a variety of codebases by running simple calculations in parallel across hundreds to thousands of GPU units. The Julia CUDA[v] package allows users to write generic Julia code that works across multiple GPU platforms and offers a very similar syntax compared to codebases run on CPU. However, because results from GPU operations are not end results and users may have to transfer data from GPU memory to CPU memory back and forth, GPU operations may slow down the whole process in the end due to additional data transfer overhead. Related topics may be discussed in papers to follow.

PARAMETER ESTIMATION

Applications like cross-selling and recommendation systems typically involve matrix decomposition or similar approaches to estimate optimal parameters. There are several parameter estimation algorithms, but one of the most popular is stochastic gradient descent, where the direction and size of descent are generally determined from a loss function and historical values of the parameters. One of the related algorithms, Factorization Machines, decomposes user-feature matrices into different degrees of interaction variables.[vi] There have been several implementations on dense matrices but, due to the infrequent nature of insurance transactions, they would typically result in sparse matrices, where the decomposition algorithm must be adapted.

Consider one possible implementation to estimate all interaction terms when the degree is 2. Let X denote the user-feature matrix to decompose, V and ΔV the interaction variables to estimate in matrix form, total losses the difference between predicted and actual values, κ the degree of freedom of the interaction variables, α the learning rate, and γ and λ_v the momentums. The following is a simplified procedure to update V and ΔV with respect to the total losses and the sparse matrix X. Because Julia offers rich manipulation options for sparse matrices, the whole process can be made straightforward to implement and run fast at the same time. The boost multiplier between Julia and Python is around six times on the same virtual machine. Benchmarking results among different languages can also be found in the appendix. This use case is currently not implemented in Mojo because a proper library support on sparse matrices has not been identified but, as it continues to grow, we plan to have it implemented in future researches.

JULIA

```
function sgd_V(X, total_losses, cross_terms, V, ΔV, κ = 10, α = 0.99, γ = 0.1, λ_v = 0.1)
    x_loss = copy(X)
    xxll = copy(X)
    currV = copy(V)
    x_loss.nzval .*= total_losses[X.rowval]; x_loss.nzval ./= X.m
    xxll .*= xxll; xxll.nzval .*= total_losses[X.rowval]; xxll.nzval ./= X.m
    xxl = sum(xxll, dims=1)
    xvxl = x_loss' * cross_terms
    @inbounds for f in 1:κ
        @inbounds for i in 1:X.n
            V[i, f] -= α * ((xvxl[i, f] - xxl[i] * V[i, f]) + γ * ΔV[i, f] + λ_v * V[i, f])
        end
    end
    ΔV .= currV .- V
end

V = rand(X.n, κ)
ΔV = rand(X.n, κ)
@btime sgd_V(X, total_losses, cross_terms, V, ΔV)
```

✓ 13.2s

```
250.002 ms (37 allocations: 594.03 MiB)
```

PYTHON

```
def sgd_V(X, total_losses, cross_terms, V, dv,
          k=10, learning_rate=0.99, m_r=0.1, m_l=0.1):
    V = v.copy()
    delta_V = dv.copy()
    x_loss = scipy.sparse.csr_matrix.copy(X)
    currV = np.matrix.copy(V)
    x_loss = X.multiply(total_losses) / X.shape[0]
    xxl = X.multiply(X).multiply(total_losses).sum(axis=0) / X.shape[0]
    xvxl = x_loss.T * cross_terms
    for f in range(k):
        for i in range(X.shape[1]):
            V[i, f] -= learning_rate * \
                ((xvxl[i, f] - xxl[0, i] * V[i, f]) +
                 m_r * delta_V[i, f] + m_l * V[i, f])
    delta_V = currV - V
    return delta_V

V = np.random.rand(num_attributes, k)
delta_V = np.random.rand(num_attributes, k)

begin_time = datetime.utcnow()
result = sgd_V(X, total_losses, cross_terms, V, delta_V)
end_time = datetime.utcnow()
print((end_time - begin_time).total_seconds(), "s")
```

✓ 1.6s

```
1.582906 s
```

On the other hand, we take C++ codebases as an example. The core calculation procedures look a lot more similar to what Julia or Python provides, but in fact separate procedures were created to do sparse matrix manipulation. C++ represents a language choice somewhat different from the others, with a lot more flexibility and complexity in the developer experience. For example:

- It is not paired with a package system like PIP or NuGet.
- It is sensitive to architecture and toolset, e.g., Linux versus Windows, Intel versus Advanced Micro Devices (AMD), GNU Compiler Collection (GCC) versus Microsoft Visual C++ (MSVC), etc.

- Integrating libraries can require complex build steps that vary by application program interface (API) or application binary interface (ABI), compiler, and operating system. Consequently, there can be countless ways to solve any given problem, both in terms of possible library dependencies and in how selective one is in choice of architecture.

For the purposes of this paper, the authors chose to use one and only one external library for the examples covered here: Boost. Boost is readily accessible in the public domain, compatible with multiple C++ compilers, comparatively straightforward to build, and portable across multiple architectures. The benefits of Boost include developer conveniences that approximate those inherent in the other languages (and related package management solutions), as well as applicability to a broad array of operating systems and chipsets, like how those sensitivities are hidden by interpreted and just-in-time (JIT) compiled code. The resulting C++ examples are also of similar source code size and complexity to the other languages.

However, a consequence of this decision is that maximum run-time performance is potentially compromised. Theoretically, because all the other languages and libraries were built on top of C or C++, it should always be possible to equal or exceed their performance starting with C++. This would, however, require relaxing the other factors considered above, including build complexity, portability, and source code footprint. Accordingly, the C++ examples here are more accurately described as "C++ with Boost only." The authors believe this context is important when evaluating relative performance of the various solutions covered here.

C++ (WITH BOOST ONLY)

```
static void sgd_V(dense_double_matrix_t& result, sparse_double_matrix_t& x, dense_double_matrix_t& total_losses
{
    double x_row_count_reciprocal = 1.0 / static_cast<double>(x.size1());

    coordinate_matrix<sparse_double_matrix_t::value_type> x_loss(x.size2(), x.size1(), x.nnz());
    vector<sparse_double_matrix_t::value_type> xxl(x.size2());

    for (sparse_double_matrix_t::iterator1 major = x.begin1(); major != x.end1(); ++major)
    {
        for (sparse_double_matrix_t::iterator2 minor = major.begin(); minor != major.end(); ++minor)
        {
            sparse_double_matrix_t::value_type xelement = *minor;
            sparse_double_matrix_t::value_type loss = xelement * total_losses(minor.index1(), 0);
            x_loss.append_element(minor.index2(), minor.index1(), loss * x_row_count_reciprocal);
            xxl[minor.index2()] += loss * xelement;
        }
    }

    xxl *= x_row_count_reciprocal;

    dense_double_matrix_t xvxl;
    matrix_multiply(x_loss, cross_terms, xvxl);

    dense_double_matrix_t v_modified(v);

    for (size_t f = 0; f < static_cast<size_t>(k); ++f)
    {
        for (size_t i = 0; i < x.size2(); ++i)
        {
            dense_double_matrix_t::value_type& vref(v_modified.at_element(i, f));
            vref -= alpha * ((xvxl(i, f) - xxl[i] * vref) + gamma * dv(i, f) + lambda * vref);
        }
    }
```

```
[RESULTS] trial(s):    4
[RESULTS] total time:  12852538000 nanoseconds
[RESULTS] min time:    3142318700 nanoseconds
[RESULTS] max time:    3300581400 nanoseconds
{"trial_count": 4, "total_seconds": 12.852538000000, "min_seconds": 3.142318700000, "max_seconds": 3.300581400000, "mean_seconds": 3.213
134500000 }
```

SIMULATION

Applications like customer behavior analyses, agency force analyses, 1-in-100 risk analysis, and pricing and reserving of certain complex insurance products, including interest-sensitive life and variable annuities, typically require modeling through simulation to get the average or values at certain percentiles. The number of scenarios typically ranges from 1,000 to 2,000, and the main loop of the simulation will usually need to reference an existing series of values, for example mortality tables or yields from pre-generated interest rate scenarios, or customer or agency information.

Let "MORT" denote mortality tables and "yield" denote interest rate scenarios. We would like to find reserves in terms of maximum deficiency of a certain variable annuity contract. Also assume there are 10,000 policies and 1,000 scenarios. It is shown in the following comparison that Julia is again the better by a boost multiplier of around 1.5 times. Benchmarking results among different other languages can also be found in the appendix.

JULIA

```
gc = zeros(n) # policy charge
b = zeros(n) # benefit
res = zeros(m) # reserve
function sima(gc, b, res, survival, YIELD, POL, MORT, m, n, τ)
    @inbounds for k = 1:m # number of scenarios
        r = zeros(n) # accumulated deficiency
        av = ones(n) # account value
        @inbounds for j = 1:τ # number of timepoints
            gc .= av .* POL[1]
            av .-= gc
            av .*= YIELD[k]
            b .= (POL[2] .- av) .* (survival[j] * MORT[j])
            r .= max.((b .- gc) ./ YIELD[k] ^ j, r)
        end
        res[k] = sum(r) # total reserve is the sum of deficiencies across all policies
    end
end
@btime sima(gc, b, res, survival, YIELD, POL, MORT, m, n, τ)
```

✓ 1m 55.9s

```
  28.863 s (7204000 allocations: 328.46 MiB)
```

PYTHON

```
gc = np.zeros(n) # policy charge
b = np.zeros(n) # benefit
res = np.zeros(m) # reserve
def sima(gc, b, res, survival, YIELD, POL, MORT, m, n, t):
    for k in range(m): # number of scenarios
        r = np.zeros(n) # accumulated deficiency
        av = np.ones(n) # account value
        for j in range(t): # number of timepoints
            gc = av * POL[0]
            av -= gc
            av *= YIELD[k]
            b = (POL[1] - av) * (survival[j] * MORT[j])
            r = np.maximum((b - gc) / (YIELD[k] ** (j + 1)), r)
        res[k] = sum(r) # total reserve is the sum of deficiencies across all policies
begin_time = datetime.utcnow()
sima(gc, b, res, survival, YIELD, POL, MORT, m, n, t)
end_time = datetime.utcnow()
print((end_time - begin_time).total_seconds(), "s")
```

✓ 39.6s

```
39.656924 s
```

In the meantime, Julia can be made even faster fairly easily by flattening vector notations into plain loops also parallelizing on the scenario loop due to their independent nature (the number of workers is 16 in the example below).

JULIA

```
res = zeros((m ÷ nworkers() + 1) * nworkers())
res = distribute(res; dist = nworkers())
@everywhere function simt(survival, YIELD, POL, n, τ)
    av = ones(n) # account value
    gc = zeros(n)
    r = zeros(n) # accumulated deficiency
    y = YIELD
    @inbounds @fastmath for j = 1:τ
        @inbounds @fastmath for k = 1:n
            gc[k] = av[k] * POL[1][k]
            av[k] -= gc[k]
            av[k] *= YIELD
            r[k] = max.(((POL[2][k] - av[k]) * survival[j] - gc[k]) / y, r[k])
        end
        y *= YIELD
    end
    return sum(r) # total reserve is the sum of deficiencies across all policies
end
function sima(res, survival, YIELD, POL, m, n, τ)
    @sync @distributed for i = 1:m
        res_local = localpart(res)
        res_local[(i - 1) % (m ÷ nworkers() + 1) + 1] = simt(survival, YIELD[i], POL, n, τ)
    end
end
@btime sima(res, survival .* MORT, YIELD, POL, m, n, τ)
```

25.6s

```
3.448 s (6456 allocations: 582.84 KiB)
```

The run-time of Mojo after parallelizing the loop for scenarios and vectorizing the loop for different policies is around 4.3 seconds on the same machine. In this example, Mojo is a lot faster compared to our plain Python implementation that had a run-time of about 39.7 seconds, but the syntax may suggest a deeper learning curve. Based on our initial exploration of Mojo in this paper, we recommend keeping an eye on it as it continues to grow.

MOJO

```
    let begin_time = now()
    @parameter
    fn scen(m: Int):
        let r = DTypePointer[DType.float64].alloc(n)
        let av = DTypePointer[DType.float64].alloc(n)
        for i in range(n):
            av.simd_store[1](i, 1.0)
        var y = YIELD[m]
        for j in range(t):
            @parameter
            fn val[nelts: Int](n: Int):
                gc.simd_store[nelts](n, av.simd_load[nelts](n) * POL0.simd_load[nelts](n))
                av.simd_store[nelts](n, (av.simd_load[nelts](n) - gc.simd_load[nelts](n)) * YIELD[m])
                let p = ((POL1.simd_load[nelts](n) - av.simd_load[nelts](n)) * survival[j]
                    - gc.simd_load[nelts](n)) / y
                if p > r.simd_load[nelts](n):
                    r.simd_store[nelts](n, p)
            vectorize[nelts, val](n)
            y *= YIELD[m]
        var s = r.simd_load[1](0)
        for j in range(n):
            s += r.simd_load[1](j)
        res.simd_store[1](m, s)
    let rt = Runtime()
    parallelize[scen](rt, m)
    let end_time = now()
    let execution_time_sequential = Float64((end_time - begin_time))
    print(execution_time_sequential / 1000000, "ms")
```

```
root@MPLSH-T14S-YL:/# ./simulation_vec_par
4323.7056970000003 ms
```

On the other hand, we can implement it in C++ with similar core structures. However, separate class utilities also must be built up to invoke parallel processing routines, which again increases development costs.

C++ (WITH BOOST ONLY)

```cpp
/* Constructor assigns task count and references to synchronization helper, read-only input and muable output */
simulation_tasks(size_t count, latch* synchronizer, simulation_input const* input, simulation_output* output) :
    count_(count),
    synchronizer_(synchronizer),
    input_(input),
    output_(output)
{
    assert(input != NULL);
    assert(output != NULL);
}

/* Required of function object (bound to task number in scheduling loop) */
result_type operator ()(size_t task_number)
{
    assert(task_number < count_);

    double reserve = 0.0;
    double yield = input().yield[task_number];

    /* Loop over policies */
    for (policy_vector_type::const_iterator policy_iter = input().inforce.begin(); policy_iter != input().inforce
    {
        double compounded_yield = 1.0;
        double deficiency = 0.0;
        double av = 1.0;
```

```cpp
        /* Loop over timesteps */
        for (size_t timestep = 0; timestep < TIMESTEP_COUNT; ++timestep)
        {
            double charge = av * policy_iter->av;

            av -= charge;
            av *= yield;

            double benefit = (policy_iter->benefit - av) * input().survival[timestep];

            compounded_yield *= yield;
            double exposure = (benefit - charge) / compounded_yield;

            if (exposure > deficiency)
            {
                deficiency = exposure;
            }
        }

        /* Accumulate total reserves over all policies */
        reserve += deficiency;
    }

    /* Commit reserve for given scenario */
    output().reserves[task_number] = reserve;

    /* Deduct number of outstanding scenario-specific parallel calculations */
    synchronizer_->count_down();

    return task_number;
}
```

```
[RESULTS] trial(s):    4
[RESULTS] total time:  19609857200 nanoseconds
[RESULTS] min time:    3163678400 nanoseconds
[RESULTS] max time:    5694107900 nanoseconds
{"trial_count": 4, "total_seconds": 19.609857200000, "min_seconds": 3.163678400000, "max_seconds": 5.694107900000, "mean
_seconds": 4.902464300000 }
```

The authors even asked ChatGPT[vii] to convert Python code in this use case to C++ with a carefully crafted prompt that included specific requests such as parallelizing all possible loops to take advantage of the high-performance computing capabilities of C++. For this use case, ChatGPT correctly identifies the requests and utilizes the Open Multi-Processing (omp) package to parallelize loops, but in terms of efficiency it does not outperform the above code, which was crafted by a professional C++ programmer. This ChatGPT migration example is a great illustration of the trade-offs between development and run-time costs when creating new products and tools, and how code can likely always be improved to increase run-time efficiency. As technology evolves, we will still have to resort to experts for properly maximizing performance wherever possible.

C++ (CONVERTED FROM PYTHON BY CHATGPT)

```
void sima(ArrayXd &gc, ArrayXd &b, ArrayXd &res, const ArrayXd &survival,
          const ArrayXd &YIELD, const pair<ArrayXd, ArrayXd> &POL, const ArrayXd &MORT) {

    #pragma omp parallel for
    for (int k = 0; k < m; ++k) {
        ArrayXd r = ArrayXd::Zero(n);
        ArrayXd av = ArrayXd::Ones(n);
        for (int j = 0; j < t; ++j) {
            gc = av * POL.first;
            av -= gc;
            av *= YIELD[k];
            b = (POL.second - av) * (survival[j] * MORT[j]);
            r = ((b - gc) / pow(YIELD[k], j + 1)).max(r);
        }
        res[k] = r.sum();
    }
}
```

```
root@MPLSH-T14S-YL:/mnt/c/Users/yuntien.lee/Downloads/gpt# ./simulation.exe
705020 ms
```

COMMENTARY ON JULIA'S PAIN POINTS

Day-to-day big data analysis for the insurance industry typically involves researching and quickly prototyping various ideas by actuaries and other professionals. Julia's ecosystem comes with rich libraries in many aspects, and the easy-to-use syntax system allows users to implement algorithms in a straightforward manner. Its support to Unicode symbols also allows easier transliteration of formulas to code, which also creates easy-to-consume documentation. Moreover, the ease of use does not sacrifice performance. If there are two equally expressive languages but one is faster than the other, it is recommended to use the faster one. The speed improvement would not only benefit in production but also everywhere in the development, test, and release cycles.

Having said the above, Julia as a general-purpose language is not a panacea under all circumstances. When it comes to implementing performance-critical tasks on specific platforms, it would be better to do them in lower-level languages such as C or assembly. As we've shown here, the language's ecosystem is an important factor. Oftentimes out-of-the-box packages are available to solve narrow computational problems, which can be combined to solve domain-specific problems. In some cases, actuarial science finds itself in a corner where those narrow computational problems are not widely solved, or solutions using a combination of packages are complex to achieve. There is merit in collaboration around general-purpose high-performance actuarial libraries to ease the development burden for all, but they should be used to solve the finer-grained problems rather than the coarse-grained, and forever changing, regulatory problems. On the other hand, for extensions to existing libraries on certain systems, it would be better to stick to the original language if the main purpose is to maintain the system, and the choice of language may be better based on the wealth of the ecosystem, rather than specific language features favorable to niche problems.

In the rapidly evolving landscape of programming technologies, only time will tell which language will become the most robust ecosystem for actuaries and data scientists alike. As demonstrated in this paper, there are already several promising and user-friendly languages available to tackle a variety of tasks. With the increasing computational complexity of the problems we face, the economic and environmental impacts of resource-intensive code are becoming more significant. Therefore, it is crucial to be cognizant not only of

the speed and ease of development but also of the costs associated with the code we create. Our efficiency and effectiveness as problem solvers are directly influenced by these factors.

Furthermore, before embarking down the path of developing new code bases to solve a task, it is crucial to assess whether existing software platforms, which can be licensed, already address the problem at hand. For common challenges, existing software platforms available for licensing might offer a more cost-effective and time-efficient solution than developing and maintaining an entirely new code base. The potential savings from license fees, when compared to the resources required for developing new software, can be substantial. Thus, a thorough assessment of available options is essential for making informed decisions in our increasingly complex and technology-driven landscape.

CONTACT

Yun-Tien Lee
yuntien.lee@milliman.com

Victor Morales
victor.morales@milliman.com

Jim Brackett
jim.brackett@milliman.com

Joe Long
joe.long@milliman.com

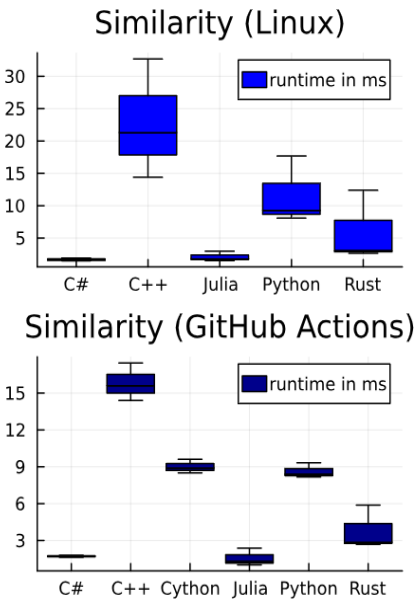
Tom Peplow
thomas.peplow@milliman.com

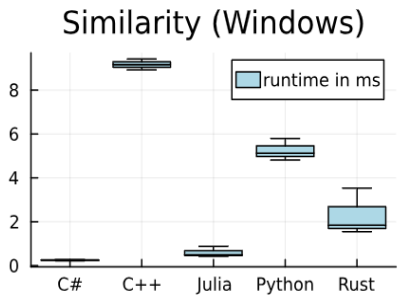
APPENDIX

GitHub Actions[viii] is a continuous integration and continuous delivery (CI/CD) platform that allows users to automate build, test, and deployment pipeline. Users can create workflows that build and test every pull request to the repository, or deploy merged pull requests to production. In addition, it is expedient for the user to test the running time of different code bases on standardized machine settings. We have extended the performance benchmarking of Julia and Python on the three insurance-related use cases discussed in the article to Cython, Rust, C#, and C++[ix] using the latest ubuntu build from GitHub Action. However, with GitHub Action it is not guaranteed that you will get the same class of hardware every time and, if you do, it is still possible that the hardware is being shared to run another action. Therefore, that is important to keep in mind as it may cause variations in the effective performance available. Because of this, we have also included benchmarking results on two other virtual machines: (1) Windows 10 Enterprise, with 16 physical cores and 24 virtual on 12th Gen Intel® Core™ i9-12900@2.4 gigahertz (GHz) and 64GB memory, and (2) Linux Ubuntu 22.04.2 LTS, with 1 physical core and 2 virtual on Intel® Xeon® Platinum 8168@2.7GHz and 4GB memory. All benchmark results are summarized with 20 runs (except Rust in parameter estimation where only one run was issued).

Note, while reviewing these results, that with every language it is likely possible to improve the run-times of these examples if more time is allowed. To create these examples, we focused on having competent developers spend similar time coding each example to highlight the trade-off between development efficiency and processing performance. It's likely that the best developer in the world could spend enough time to make C++ or other equivalent ones run the fastest of all of these.

- Similarity calculation. From the graph on GitHub Actions it shows a boost multiplier of about 8, that is, Julia implementation is almost eight times faster than either Python or Cython.





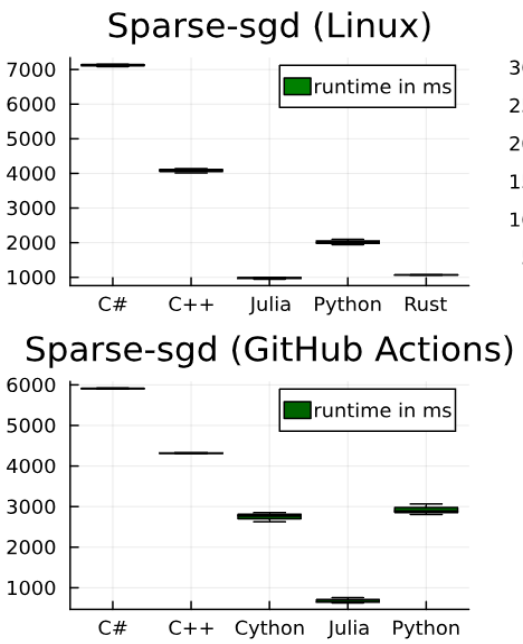
Linux	C#	C++	Julia	Python	Rust
Min	1.49	14.39	1.51	8.10	2.66
Mean	1.62	21.31	1.81	9.29	3.09
Max	1.93	32.68	2.96	17.67	12.39

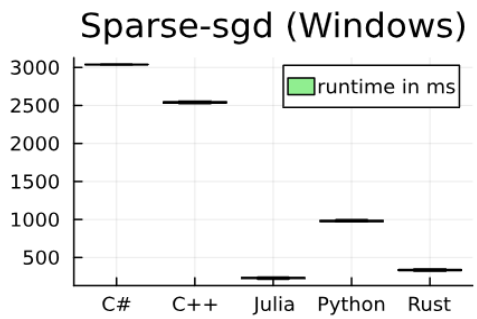
Windows	C#	C++	Julia	Python	Rust
Min	0.22	8.92	0.42	4.81	1.55
Mean	0.25	9.17	0.49	5.13	1.84
Max	0.28	9.42	0.88	5.80	3.53

Windows	C#	C++	Julia	Python	Rust
Min	0.22	8.92	0.42	4.81	1.55
Mean	0.25	9.17	0.49	5.13	1.84
Max	0.28	9.42	0.88	5.80	3.53

Source: GitHub Actions

- Parameter estimation. From the graph on GitHub Actions it shows a boost multiplier of about 4, that is, Julia implementation is about four times faster than either Python or Cython.





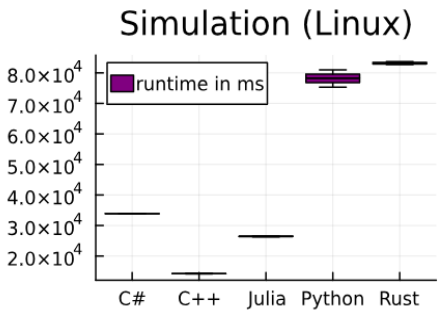
Linux	C#	C++	Julia	Python	Rust
Min	7,083	4,014	946	1,944	1,066
Mean	7,131	4,087	983	2,011	1,066
Max	7,152	4,140	1,003	2,095	1,066

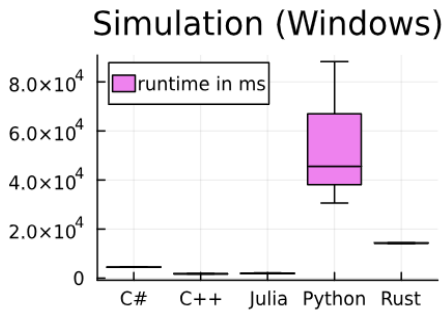
Windows	C#	C++	Julia	Python	Rust
Min	3,034	2,526	215	971	334
Mean	3,038	2,538	232	977	334
Max	3,042	2,558	242	997	334

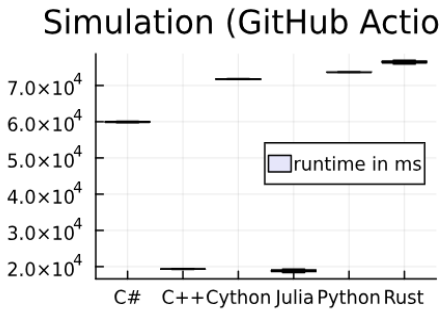
GitHub	C#	C++	Cython	Julia	Python
Min	5,895	4,308	2,626	627	2,809
Mean	5,910	4,317	2,774	658	2,894
Max	5,923	4,331	2,851	759	3,063

Source: GitHub Actions

- Simulation. From the graph on GitHub Actions it shows a boost multiplier of 4, that is, Julia implementation is about four times faster than either (un-parallelized version of) Python or Cython.







Linux	C#	C++	Julia	Python	Rust
Min	33,861	14,163	26,284	75,331	82,766
Mean	33,888	14,234	26,431	78,274	83,100
Max	33,917	14,282	26,568	80,962	83,679

Windows	C#	C++	Julia	Python	Rust
Min	4,405	1,754	1,838	30,661	14,153
Mean	4,491	1,775	1,910	45,637	14,301
Max	4,565	1,800	2,132	88,259	14,425

GitHub	C#	C++	Cython	Julia	Python	Rust
Min	59,720	19,270	71,710	18,370	73,630	75,920
Mean	59,940	19,300	71,760	18,860	73,660	76,520
Max	60,100	19,420	71,850	19,310	73,710	76,950

ENDNOTES

[i] Refer to https://docs.juliahub.com/JuliaInXL/AZenl/1.2.0/ and https://www.pyxll.com/docs/introduction.html.
[ii] Refer to the published paper at https://www.milliman.com/en/insight/potential-data-sources-for-life-insurance-ai-modelling.
[iii] Refer to some discussions at https://www.zdnet.com/article/programming-languages-its-time-to-stop-using-c-and-c-for-new-projects-says-microsoft-azure-cto/.
[iv] Refer to the documentation at https://numpy.org/devdocs/reference/generated/numpy.dot.html.
[v] Refer to the introduction at https://github.com/JuliaGPU/CUDA.jl.
[vi] Refer to the notebook at https://www.kaggle.com/code/chenhaokun/factorization-machines/notebook.
[vii] Refer to the website at https://chat.openai.com/.
[viii] Refer to the documentation at https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions.
[ix] Refer to all code bases at Julia_research_2023 (github.com).