SOCIETY OF ACTUARIES

TECHNOLOGY SECTION

ACTUARIAL TECHNOLOGY TODAY

OCTOBER 2020

What's in a Number? Considerations for Mapping the COVID-19 Pandemic

By Phil Ellenberg and Kelsie Gosser

Author's note: Using COVID-19 visualizations as a case study, this article highlights the considerations involved with preparing data to tell a story. As actuaries we are often challenged with how to tell a complex story using nuanced data. This article provides a helpful framework.

As COVID-19 remains a mainstay in 2020, it seems like the only certainty we can count on is that we remain uncertain about the global pandemic. In an effort to mitigate this uncertainty, we have collectively turned to data and, by extension, data visualizations. Despite numerous issues with the underlying data, COVID-19 dashboards and other visualizations have become ubiquitous in our daily lives, each with their own recipe for showing the impact of the virus. Using confirmed case count data for the United States from the Center for Systems Science and Engineering (CSSE) at Johns Hopkins University[1], we intend to explore the interaction between selecting a metric and then visualizing it. We chose the Johns Hopkins' data over other data sources such as The New York Times or The COVID-19 Tracking Project due to its prominence in the media, credibility, and its ease of access.

When we first started writing this article, we went into it with a plan to write strictly about data visualization techniques and how to pick which graph to best illustrate the data it represents—the twist being that we were going to use COVID-19 data. We quickly found ourselves uninterested in the topic, especially after referencing countless other articles on the same topic (although, to be fair, none of them use COVID-19 data). We were, however, intrigued by the variety of metrics used across the endless patchwork of COVID-19 dashboards. If case counts, hospitalizations, fatalities, and tests were the bricks for our house, figuring out how best to report them was like deciding what color to paint the walls. Do we look at cumulative counts going back to March? Do we look at newly reported numbers on a daily basis? Do we adjust the data for population? What about trend? Should we look at day-over-day changes? Week-over-week? Moving averages?

These are questions we failed to thoroughly think through before diving in. This was evident when we first started drafting this article and dropped cumulative case counts by State onto a bubble map (formally known as a proportional symbol map) in Microsoft's Power BI. As shown in Figure 1 (page 2), looking solely at case counts causes New York to eclipse the rest of the country. An argument can be made that this is due primarily to the population density of New York City. States like California, Florida, and Texas also jump out as being hard hit, but these are also big states to begin with. To be sure, these states have all made headlines during the pandemic, but other headline-grabbing states like Louisiana, Illinois, and Arizona are lost in the shuffle.

Copyright © 2020 Society of Actuaries. All rights reserved.

Figure 1
Cumulative Confirmed Case Counts by State

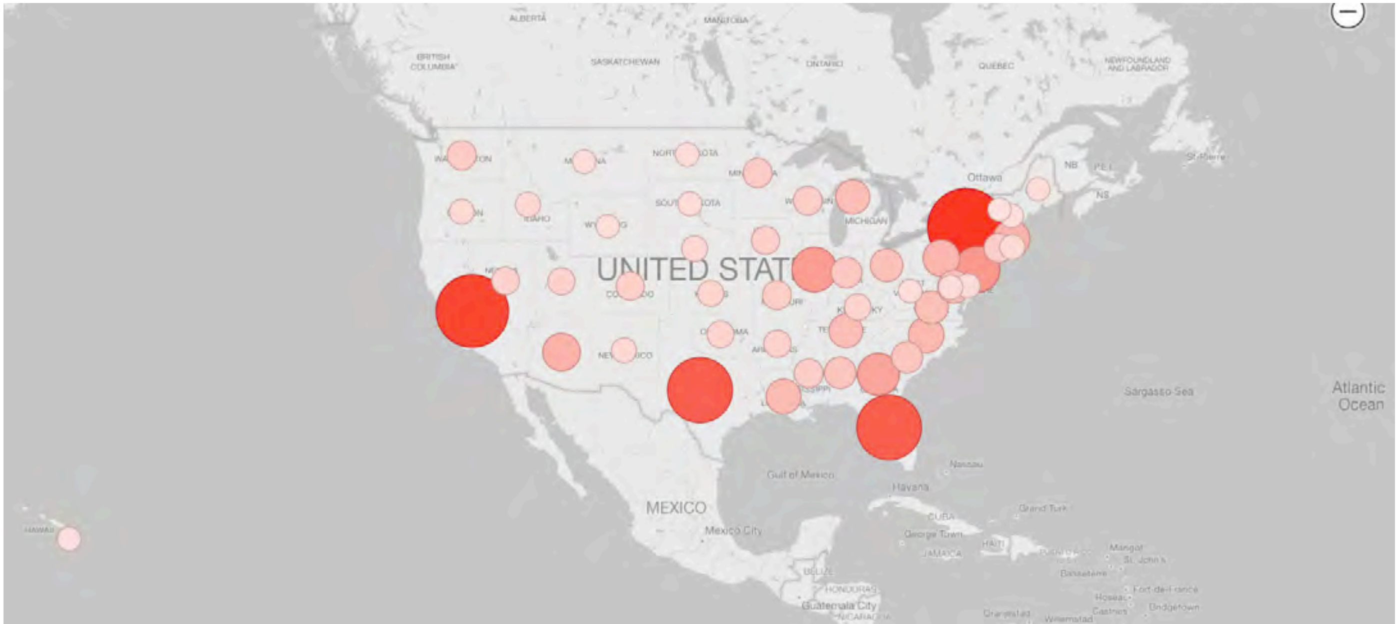
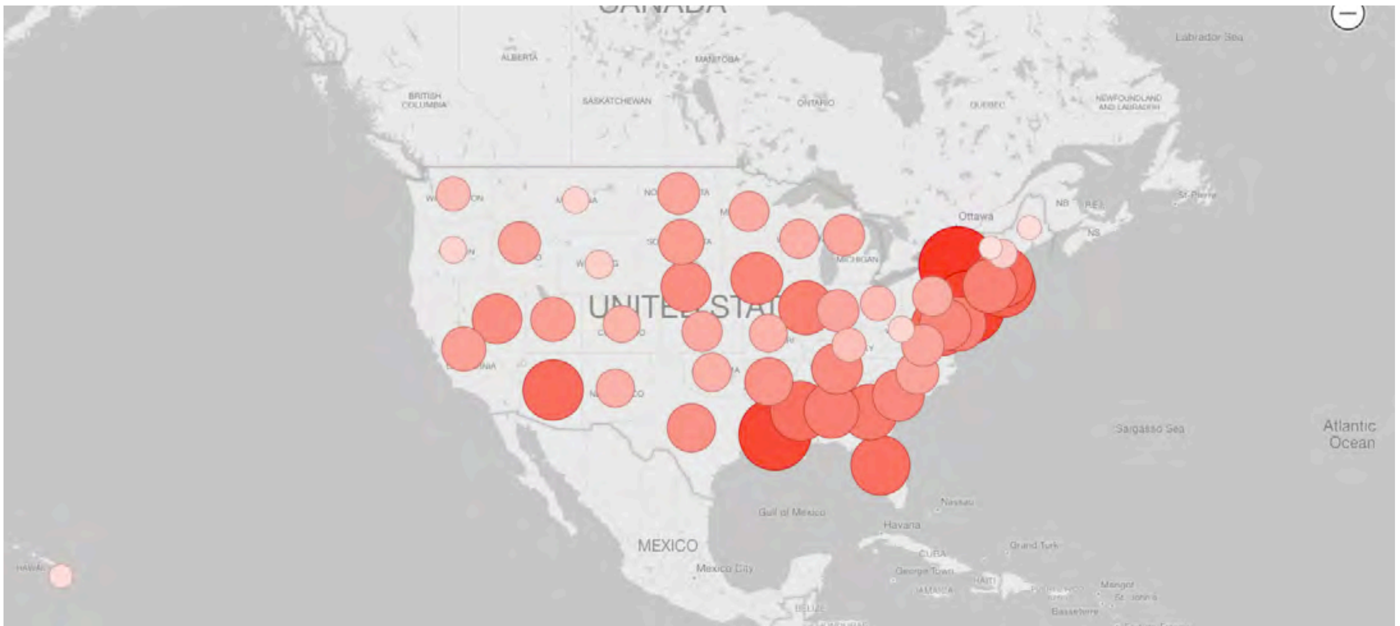


Figure 2
Cumulative Confirmed Case Counts per 100k by State



Simply adjusting the cumulative case count data for population shows a slightly different story. While New York still dominates when showing cumulative case counts per 100k residents in a state, the impacts on other New England and Mid-Atlantic states such as New Jersey, Massachusetts, Rhode Island, Connecticut, and Delaware are immediately more noticeable. Adjusting for population also highlights states like Louisiana, Illinois, and Arizona as being more heavily impacted. (See Figure 2.)

What cumulative case counts miss, however, is how case counts are changing over time. By nearly all accounts, New York's cases peaked in early-April. To somehow capture the trend, we were faced with two decisions: what type of map to use, and what metric to use. Generally speaking, proportional maps should be used to show totals rather than rates. Filled maps, or choropleths, are a standard way of displaying rates or ratios between well-defined geographies, so this was an obvious choice. The less obvious choice was the metric. While some dashboards

Copyright © 2020 Society of Actuaries. All rights reserved.

Figure 3
Seven-Day Moving Average of New Cases

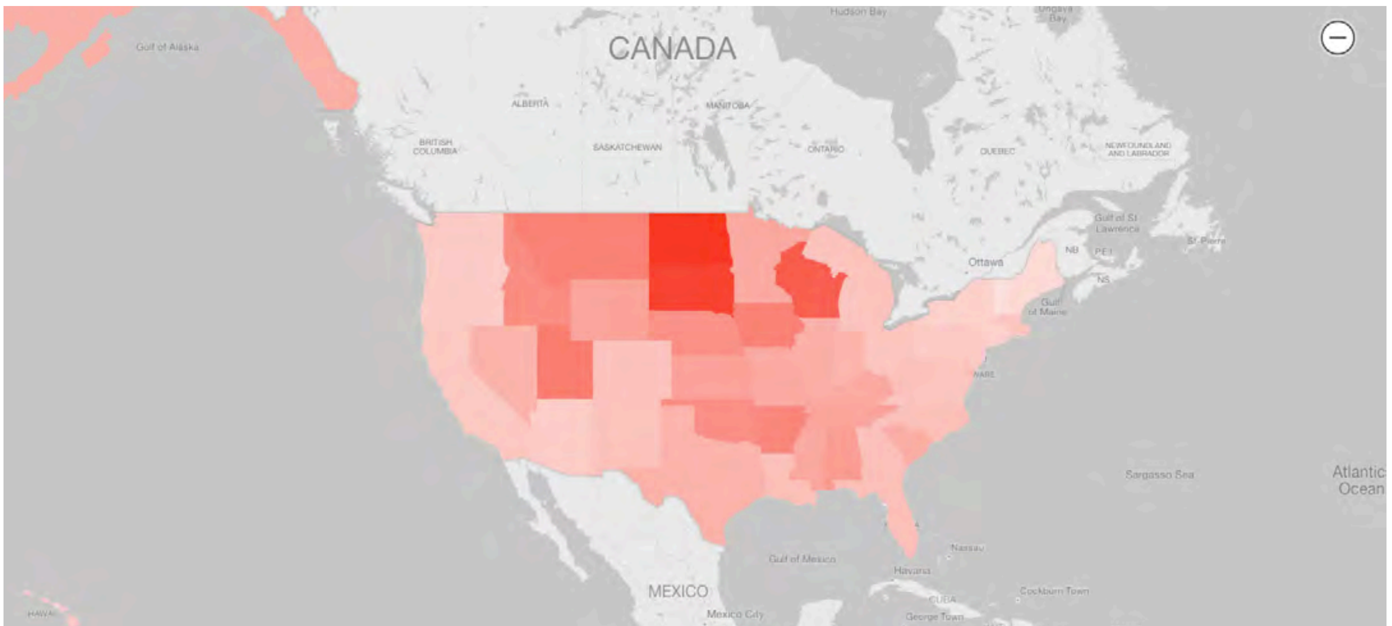
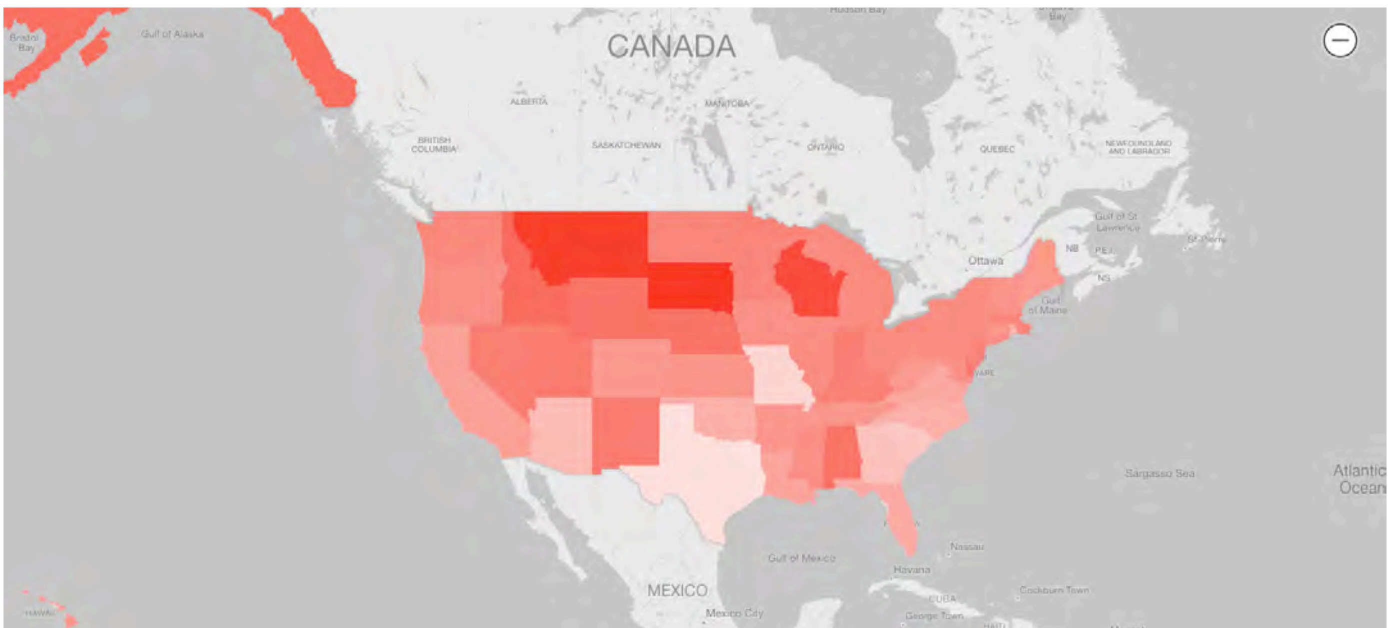


Figure 4
Seven-Day Moving Average of New Cases Week-Over-Week

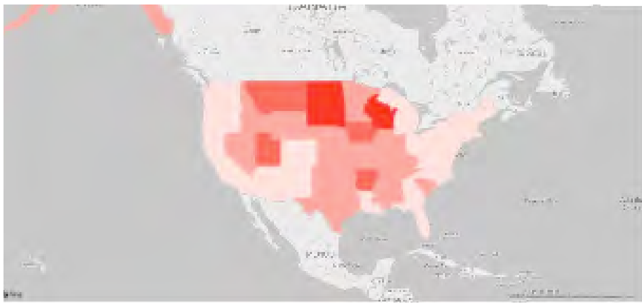


show the daily change in new cases, we felt that this metric was too sensitive to change and susceptible to large swings following weekends when test volume is typically down. Similarly, weekly change was prone to the same pitfalls in many states where case counts experience weekly seasonality with lower volume during weekends. We eventually found that using a seven-day moving average of new cases per 100k managed to smooth over these types of swings without distorting the direction or strength of new case counts. (See Figure 3.)

By definition, a simple moving average takes the mean of a given set of data over a specific number of time periods in the past. In effect, this minimizes the effect of random fluctuations in the short-term. To truly understand which states should be concerned about becoming a hot spot, we want to be aware of short-term fluctuations. Comparing the most recent seven-day moving average of new cases to the previous seven-day period captures the momentum of new case counts, which we concluded was a more meaningful way of looking at these data. (See Figure 4.)

Copyright © 2020 Society of Actuaries. All rights reserved.

Figure 5a
7-Day Moving Average of New Cases



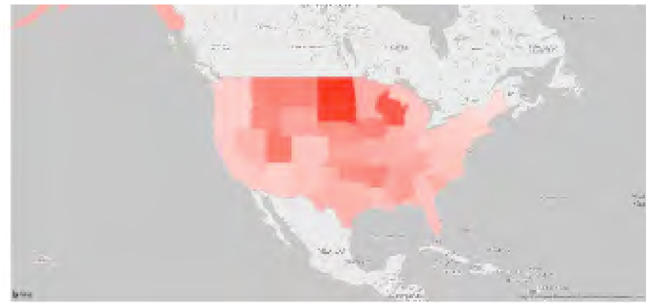
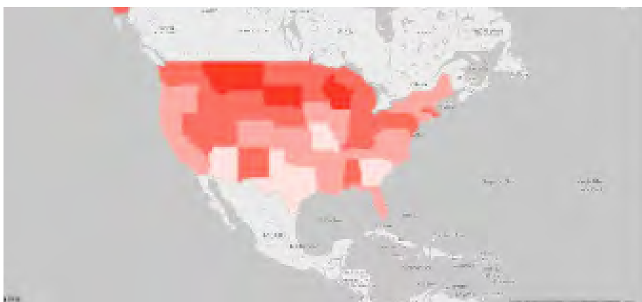
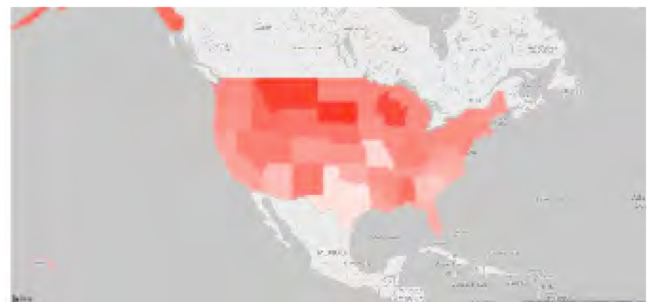


Figure 5b
7-Day Moving Average of New Cases Week-Over-Week





Once we settled on a more useful metric, we wanted to ensure that we were using the best color scheme for the graphic. We started by looking for common heat map color palettes, which led us to "magma," a popular palette in the even more popular Matplotlib Python library. Originally, we simply used Power BI's out-of-the-box color gradient to represent the lowest and highest values with darker colors representing a higher week-over-week change. This, unfortunately, resulted in a muddy graph that failed to highlight the hot spots. After some testing, we found that using a simple gradient with a soft, light pink color as the low value and an intense bright red as the high value clearly highlighted the states that were being impacted the most based on our metric. Even though magma is an awesome color palette, and one that finds its way into many Jupyter notebooks, it felt too intense for the state-level data we are displaying here.

Next, we had to decide if we wanted to continue with the continuous gradient effect, or if we wanted to create color steps by binning the data. Determining the number of steps requires a balancing act between showing too many steps, which can cause the change to seem more drastic than it really is, and showing too few steps, which can have the opposite effect. We landed on using four steps based on both the distribution of the week-over-week change in the seven-day moving average as well as the readability of the map itself. To be fair, both the gradient and the steps for these maps make sense. If we were to show the data at the county level, bins of color would likely be more appropriate as you can more easily discern each small piece of the larger puzzle and where it stands with cases. When it comes to 50 states, the gradient works just fine. (See Figure 5a and 5b.)

It goes without saying that there are many ways to display data. Similar to data analysis, data visualization requires practitioners to journey through a garden of forking paths. From determining what metric to display to deciding on how best to display it, there is no perfect way to design a data visualization. Exploring metrics, graphics, and colors will get you closer to your intended goal, but oftentimes the goal itself can be a moving target. Moreover, the availability and quality of your underlying data may lead you down a rabbit hole of developing metrics or drill downs. For example, the visualizations in this article could benefit from the ability to filter by date, or the ability to drill down to county-level detail. That said, it is imperative to recognize when your visualization has met its intended goal.

We started out with the goal of simply developing a visualization related to COVID-19. Instead, we found ourselves thinking more critically about the entire process. We ultimately presented our journey of using different metrics, their effect on developing an appropriate map, and how to think about making the map as readable as possible without losing critical information in the data. We hope to not only inspire readers to thoroughly consider both their data and purpose when developing any type of visualization, but also to recognize the need to be flexible throughout the process. Being conscious of your underlying data, the potential for errata, and how your data is prepared are

Copyright © 2020 Society of Actuaries. All rights reserved.

important steps in telling the data's story. An even more important step is to tell the story.

LIMITATIONS

This article is written based on data available as of the time of writing. The underlying data and circumstances related to COVID-19 are subject to uncertainty, given the emerging experience of the pandemic. These data are dependent on many factors at the state and local levels such as shelter in place orders, availability of testing, and reporting agency capabilities. The visualizations presented herein are based on past data and are not projections of the future. ■

This article is intended only for the purpose of approaching a data visualization task and should not be used for other purposes.

Phil Ellenberg is a healthcare consultant with Milliman. He can be reached at *phil.ellenberg@milliman.com.*

Kelsie Gosser is a user experience (UX) engineer with Milliman. She can be reached at *kelsie.gosser@milliman.com.*

ENDNOTES

1 *https://github.com/CSSEGISandData/COVID-19/blob/master/csse_covid_19_data/csse_covid_19_time_series/time_series_covid19_confirmed_US.csv*

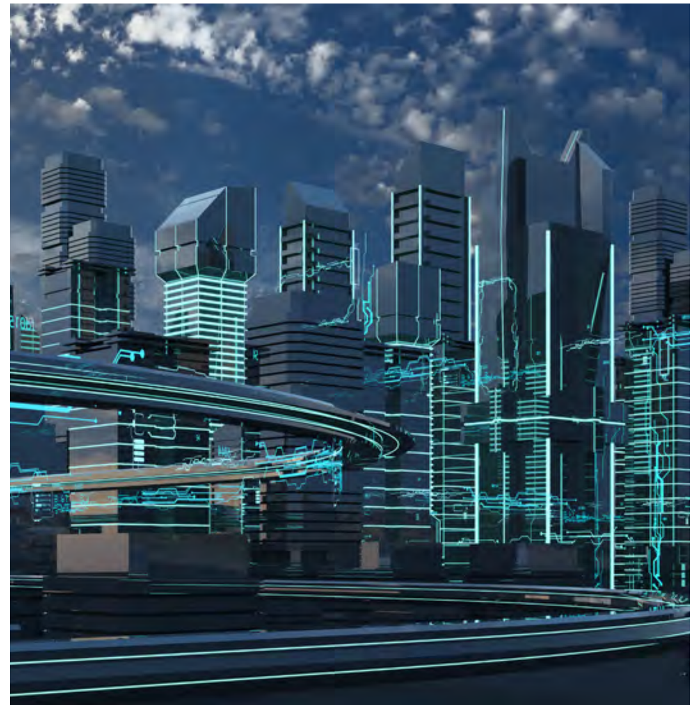
Copyright © 2020 Society of Actuaries. All rights reserved.

SOCIETY OF ACTUARIES

TECHNOLOGY SECTION

Coding the Future

By Alec Loudenback



"… the insurance business is perhaps the purest example of an 'information-based' industry—that is, an industry whose sole activity consists of gathering, processing, and distributing information."—Martin Campbell-Kelly, writing about the Prudential in the Victorian Era.[1]

Insurance and financial services are fundamentally information services. To thrive, technology needs to be central to company strategy and actuaries should view coding as a core competency—equal to or more important than other "traditional" skills.

THE INSURANCE INDUSTRY: YESTERDAY, TODAY, AND TOMORROW

It might be odd to say that technology and its use in insurance is on a one-hundred-year cycle, but that seems to be the case.

One hundred thirty years ago, actuaries crowded into a room at a meeting of the Actuarial Society of America to watch a demonstration that would revolutionize the industry: Herman Hollerith's tabulating punch card machine.[1]

For the next half-century, the increasing automation—from tabulating machines to early-adopting mainframes and computers—was a critical competitive differentiator. Companies like Prudential, MetLife, and others partnered with technology companies in the development of hardware and software.[2]

The dramatic embodiment of this information-driven cycle was portrayed in the infamous Billion Dollar Bubble movie, which showcased the power and abstraction of the computer to commit millions of dollars of fraud by creating and maintaining fake insurance policies.

The movie also starts to hint at the oscillation away from the technological-competitive focus of insurance companies. I argue that the focus on technology was lost over the last 50 years with the rise of Wall Street finance, investment-oriented life insurance, industry consolidation, and the explosion of financial structuring like derivatives, reserve financing, or other advanced forms of reinsurance.

Value-add came from the C-Suite, not from the underlying business processes, operations, and analysis. The result is, e.g., ever-more complicated reinsurance treaties layered into mainframes and admin systems older than most of the actuaries interfacing with them.

The pace of **strategic value-add** isn't slowing, though it must stretch farther (in complexity and risk) to find comparable opportunities as the past. Having more agile, data-oriented operations enables companies to be able to react to and implement those opportunities. **Technological value-add** can improve a company's bottom line through lower expenses and higher topline growth, but often with a more favorable risk profile than some of the "strategic" opportunities.

Today, there is a trend reverting back to technological value-creation and is evident across many traditional sectors. Tesla claims that it's a technology company; Amazon is the #1 product retailer because of its vehement focus on internal information sharing[3]; airlines are so dependent on their systems that the skies become quieter on the rare occasion that their computers give way.

Copyright © 2020 Society of Actuaries. All rights reserved.

Why is it, that companies that are so involved in **things** (cars, shopping) and **physical services** (flights) are so much more focused on improving their technological operations than insurance companies **whose very focus is "information-based"? The market has rewarded those who have prioritized their internal technological solutions.**

Commoditized investing services and low yield environments have reduced insurance companies' comparative advantage to "manage money." Yield compression and the explosion of consumer-oriented investment services means a more competitive focus on the ability to manage the entire policy lifecycle efficiently (digitally), perform more real-time analysis of experience and risk management, and handle the growing product and regulatory complexity.

These are problems that have technological solutions and are waiting for insurance company adoption.

Companies that treat data like coordinates on a grid (spreadsheets) **will get left behind**. Two main hurdles have prevented technology companies from breaking into insurance:

1. High regulatory barriers to entry, and
2. difficulty in selling complex insurance products without traditional distribution.

Once those two walls are breached, traditional insurance companies without a strong technology core will struggle to keep up. The key to thriving is not just adding "developers" to an organization; it's going to be **getting domain experts like actuaries to be an integral part of the technology transformation**.

WHAT'S CODING GOT TO DO WITH THIS?

Everything. Programming is the optimal way to interact between the computer and actuary—and importantly between computer and computer. Programming is the actionable expression of ideas, math, analysis, and information. Think of programming as the 21st-century leap in the actuary's toolkit, just as spreadsheets were in the preceding 40 years, versus a spreadsheet-oriented workflow:

- More natural automation of, and between processes,
- better reproducibility,
- scaling to fit any size dataset and workload,
- statistics and machine learning capabilities, and
- advanced visualizations to garner new views into your data.

This list isn't comprehensive, and some benefits are subtle—when you are code-oriented instead of spreadsheet-oriented, you tend to want to structure your data in a portable and shareable way. For example, relying more on data warehouses instead of email attachments. This, in turn, enables data discovery and insights that otherwise wouldn't be there. Investing in a code-oriented workflow is playing the long-game.

The actuary of the future needs to have coding as one of their core skills. Already today, the advances of business processes, insurance products, and financial ingenuity are written with lines of code—not spreadsheets. Not being able to code **necessarily** means that you are **following** what others are doing today.

It's commonly accepted now that to gather insights from your data, you need to know how to code. Similar to your data, your business architecture, modeling needs, and product peculiarities are often better suited to customized solutions. Why stop at data science when learning how to solve problems with a computer?

THE 10X ACTUARY

As we swing back to a technological focus, we do not leave the finance-driven complexity behind. The increasingly complex business needs will highlight a large productivity difference between an actuary who can code and one who can't—simply because the former can react, create, synthesize, and model faster than the latter. From the efficiency of transforming administration extracts, summarizing, and aggregating valuation output, to analyzing claims data in ways that spreadsheets simply can't handle, you can become a **10x Actuary**.[4]

Flipping switches in a graphical user interface versus being able to build models is the difference between having a surface-level familiarity and having full command over the analysis and the concepts involved—with the flexibility to do what your software can't.

Your current software might be able to perform the first layer of analysis but be at a loss when you want to visualize, perform sensitivity analysis, statistics, stochastic analysis, or process automation. Things that, when done programmatically, are often just a few lines of additional code.

Do I advocate dropping the license for your software vendor? No, not yet anyway. But the ability to supplement and break out of the modeling box has been an increasingly important part of most actuaries' work.

Additionally, code-based solutions can leverage the entire technology sector's progress to solve problems that are **hard** otherwise: scalability, data workflows, integration across functional areas, version control and versioning, model change governance, reproducibility, and more.

Thirty to forty years ago, there were no vendor-supplied modeling solutions, and so you had no choice but to build models internally. This shifted with the advent of vendor-supplied modeling solutions. Today, it's never been better for companies to leverage open source to support their custom modeling, risk analysis/monitoring, and reporting workflows.

Copyright © 2020 Society of Actuaries. All rights reserved.

RISK GOVERNANCE

Code-based workflows are highly conducive to risk governance frameworks as well. If a modern software project has all of the following benefits, then why not a modern insurance product and associated processes?

- Benefits of Modern Risk Governance
- Access control and approval processes.
- Version control, version management, and reproducibility.
- Continuous testing and validation of results.
- Open and transparent design.
- Minimization of manual overrides, intervention, and opportunity for user error.
- Automated trending analysis, system metrics, and summary statistics.
- Continuously updated, integrated, and self-generating documentation.
- Integration with other business processes through a formal boundary (e.g., via an API).
- Tools to manage collaboration in parallel and in sequence.

MANAGING AND LEADING THE TRANSFORMATION

The ability to understand the concepts, capabilities, challenges, and lingo is not a dichotomy, it's a spectrum. Most actuaries, even at fairly high levels, are still often involved in analytical work. Still above that, it's difficult to lead something that you don't understand.

Conversely, the skill and practice of coding enhances managerial capabilities. When you are really skilled at pulling apart a problem or process into its constituent parts and designing optimal solutions, that's a core attribute of leadership: Having the vision of where the organization should be instead of thinking about where it is now.

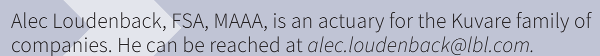
Nor is the skillset described here limiting in any other aspect of career development any more than mathematical ability, project collaboration, or financial acumen—just to name a few.

OUTLOOK

It will increasingly be essential for companies to modernize to remain competitive. That modernization isn't built with big black-box software packages; it will be with domain experts who can translate the expertise into new forms of analysis—doing it faster and more robustly than the competition.

SpaceX doesn't just hire rocket scientists—they hire rocket scientists who code.

Be an actuary who codes. ■


Alec Loudenback, FSA, MAAA, is an actuary for the Kuvare family of companies. He can be reached at *alec.loudenback@lbl.com.*

ENDNOTES

1 Yates, J. (1993). *Co-evolution of Information Processing Technology and Use: Interaction Between the Life Insurance and Tabulating Industries* [Scholarly project]. *In Sloan School of Management.*

2 Yates, J. (1993). *From Tabulators to Early Computers in the U.S. Life Insurance Industry: Co-evolution and Continuities* [Scholarly project]. *In Sloan School of Management*.

3 Mason, R. (2017, August 25). Have you had your Bezos moment? What you can learn from Amazon. Retrieved 2020, from *https://www.cio.com/article/3218667/have-you-had-your-bezos-moment-what-you-can-learn-from-amazon.html*

4 Brikman, Y. (2013, September 29). The 10x developer is NOT a myth. Retrieved 2020, from *https://www.ybrikman.com/writing/2013/09/29/the-10x-developer-is-not-myth/*

Copyright © 2020 Society of Actuaries. All rights reserved.

Julia for Actuaries

By Alec Loudenback

I have suggested that actuaries who are competent coders will differentiate both themselves and the companies they work for. Coding ability will be useful no matter what tools you utilize every day (e.g., Python/R/C++/etc. and associated packages) and all of those tools and communities contribute to moving actuarial processes out of the "Spreadsheet Age."

There's a newer programming language called Julia, and in this article, I'd like to state why Julia is worth considering for actuarial work.

JULIA OVERVIEW

Julia is relatively new[1], and **it shows**. It is evident in its pragmatic, productivity-focused design choices, pleasant syntax, rich ecosystem, thriving communities, and its ability to be both very general purpose and power cutting edge computing.

With Julia: math-heavy code **looks like math**; it's easy to pick up, and quick-to-prototype. Packages are well-integrated, with excellent visualization libraries and pragmatic design choices.

Julia's popularity continues to grow across many fields and there's a growing body of online references and tutorials, videos, and print media to learn from.

Large financial services companies have already started realizing gains: BlackRock's Aladdin portfolio modeling, the Federal Reserve's economic simulations, and Aviva's Solvency II-compliant modeling. The last of these has a great talk on YouTube by Aviva's Tim Thornham, which showcases an on-the-ground view of what difference the right choice of technology and programming language can make. Moving from their vendor-supplied modeling solution was **1000x faster, took 1/10 the amount of code, and was implemented 10x faster**.[2]

The language is not just great for data science—but also modeling, ETL, visualizations, package control/version management, machine learning, string manipulation, web-backends, and many other use cases.

FOR THE ACTUARY

Julia is well suited for actuarial work: easy to read and write and very performant for large amounts of data/modeling.

Expressiveness and Syntax

Expressiveness is the **manner in which** and **scope of** ideas and concepts that can be represented in a programming language. **Syntax** refers to how the code **looks** on the screen and its readability.

In a language with high expressiveness and pleasant syntax, you:

- Go from idea in your head to final product faster.
- Encapsulate concepts naturally and write concise functions.
- Compose functions and data naturally.
- Focus on the end-goal instead of fighting the tools.

Expressiveness can be hard to explain, but perhaps two short examples will illustrate.

Copyright © 2020 Society of Actuaries. All rights reserved.

Example: Retention Analysis

This is a really simple example relating Cessions, Policys, and Lives to do simple retention analysis.

First, let's define our data:

```
# Define our data structures
struct Life
  policies
end

struct Policy
  face
  cessions
 end

struct Cession
  ceded
end
```

Now to calculate amounts retained. First, let's say what retention means for a `Policy`:

```
# define retention
function retained(pol::Policy)
  pol.face - sum(cession.ceded for
  cession in pol.cessions)
end
```

And then what retention means for a `Life`:

```
function retained(l::Life)
  sum(retained(policy) for policy in
life.policies)
end
```

It's almost exactly how you'd specify it in English. No joins, no boilerplate, no fiddling with complicated syntax. You can express ideas and concepts the way that you think of them, not, for example, as a series of dataframe joins or as row/column coordinates on a spreadsheet.

We defined `retained` and adapted it to mean related, but different things depending on the specific context. That is, we didn't have to define `retained_life(...)` and `retained_pol(...)` because Julia can be **dispatched** based on what you give it. This is, as some would call it, unreasonably effective.

Let's use the above code in practice then.

The `julia>` syntax indicates that we've moved into Julia's interactive mode (REPL mode):

```
# create two policies with two and one
cessions respectively
julia> pol_1 = Policy( 1000, [
Cession(100), Cession(500)] )

julia> pol_2 = Policy( 2500, [
Cession(1000) ] )

# create a life, which has the two
policies
julia> life = Life([pol_1, pol_2])
```

```
julia> retained(pol_1)
400
```

```
julia> retained(life)
1900
```

And for the last trick, something called "broadcasting," which automatically vectorizes any function you write, no need to write loops or create if statements to handle a single versus repeated case:

```
julia> retained.(life.policies) # re-
tained amount for each policy
[400 ,  1500]
```

Example: Random Sampling

As another motivating example showcasing multiple dispatch, here's random sampling in Julia, R, and Python.

We generate 100:

- Uniform random numbers
- standard normal random numbers
- Bernoulli random number
- Random samples with a given set

Copyright © 2020 Society of Actuaries. All rights reserved.

Julia	R	Python

`using Distributions`
`rand(100)`
`rand(Normal(), 100)`
`rand(Bernoulli(0.5), 100)`
`rand(["Preferred", "Standard"],100)`
	`runif(100)`
	`rnorm(100)`
	`rbern(100,0.5)`
	`sample(c("Preferred", "Standard"), 100, replace=TRUE)`
		`import scipy.stats as sps`
		`import numpy as np`
		
		`sps.uniform.rvs(size=100)`
		`sps.norm.rvs(size=100)`
		`sps.bernoulli.rvs(p=0.5,size=100)`
		`np.random.choice(["Preferred","Standard"],size=100)`

By understanding the different types of things passed to `rand()`, it maintains the same syntax across a variety of different scenarios. We could define `rand(Cession)` and have it generate a random `Cession` like we used above.

The Speed

Julia is also fast. As the journal *Nature* said, "Come for the Syntax, Stay for the Speed."

Recall the Solvency II compliance that ran 1000x faster than the prior vendor solution mentioned earlier: what does it mean to be 1000x faster at something? It's the difference between something taking **10 seconds** instead of **three hours**—or **one hour** instead of **42 days**.

What analysis would you like to do if it took less time? A stochastic analysis of life-level claims? Machine learning with your experience data? Daily valuation instead of quarterly?

Speaking from experience, speed is not just great for production time improvements. During development, it's really helpful too. When building something, I can see that I messed something up in a couple of seconds instead of 20 minutes. The build, test, fix, iteration cycle goes faster this way.

Admittedly, most workflows don't see a 1000x speedup, but 10x to 1000x is a very common range of speed differences versus R or Python or MATLAB.

Sometimes you will see less of a speed difference; R and Python have already circumvented this and written much core code in low-level languages. This is an example of what's called the "two-language" problem where the language productive to write in isn't very fast. For example, more than half of R packages use C/C++/Fortran and core packages in Python like Pandas, PyTorch, NumPy, SciPy, etc., do this too.

Within the bounds of the optimized R/Python libraries, you can leverage this work. Extending it can be difficult: what if you have a custom retention management system running on millions of policies every night?

Julia packages you are using are almost always written in pure Julia: you can see what's going on, learn from them, or even contribute a package of your own!

More of Julia's Benefits

Julia is easy to write, learn, and be productive in:

- It's free and open-source
  - Very permissive licenses, facilitating the use in commercial environments (same with most packages)
- Large and growing set of available packages
- Write how you like because it's multi-paradigm: vectorizable (R), object-oriented (Python), functional (Lisp), or detail-oriented (C)
- Built-in package manager, documentation, and testing-library
- Jupyter Notebook support (it's in the name! **Ju**lia-**Pyt**hon-**R**)
- Many small, nice things that add up:
  - Unicode characters like α or β
  - Nice display of arrays
  - Simple anonymous function syntax
  - Wide range of text editor support
  - First-class support for missing values across the entire language
  - Literate programming support (like R-Markdown)
- Built-in Dates package that makes working with dates pleasant
- Ability to directly call and use R and Python code/packages with the `PyCall` and `RCall` packages
- Error messages are helpful and tell you what line the error came from, not just the type of error
- Debugger functionality so you can step through your code line by line

For power-users, advanced features are easily accessible: parallel programming, broadcasting, types, interfaces, metaprogramming, and more.

These are some of the things that make Julia one of the world's most loved languages on the StackOverflow Developer Survey.

Copyright © 2020 Society of Actuaries. All rights reserved.

For those who are enterprise-minded: in addition to the liberal licensing mentioned above, there are professional products from organizations like Julia Computing that provide hands-on support, training, IT governance solutions, behind-the-firewall package management, and deployment/scaling assistance.

The Tradeoff

Julia is fast because it's compiled, unlike R and Python where (loosely speaking) the computer just reads one line at a time. Julia compiles code "just-in-time": right before you use a function for the first time, it will take a moment to pre-process the code section for the machine. Subsequent calls don't need to be re-compiled and are very fast.

A hypothetical example: running 10,000 stochastic projections where Julia needs to precompile but then runs each 10x faster:

- Julia runs in two minutes: the first projection takes one second to compile and run, but each 9,999 remaining projections only take 10ms.
- Python runs in 17 minutes: 100ms of a second for each computation.

Typically, the compilation is very fast (milliseconds), but in the most complicated cases it can be several seconds. One of these is the "time-to-first-plot" issue because it's the most common one users encounter: super-flexible plotting libraries have a lot of things to pre-compile. So, in the case of plotting, it can take several seconds to display the first plot after starting Julia, but then it's remarkably quick and easy to create an animation of your model results. The time-to-first plot is a solvable problem that's receiving a lot of attention from the core developers and will get better with future Julia releases.

For users working with a lot of data or complex calculations (like actuaries!), the runtime speedup is worth a few seconds at the start.

Package Ecosystem

Using packages as dependencies in your project is assisted by Julia's bundled package manager.

For each project, you can track the **exact** set of dependencies and replicate the code/process on another machine or another time. In R or Python, dependency management is notoriously difficult and it's one of the things that the Julia creators wanted to fix from the start.

Packages can be one of the thousands of publicly available, or private packages hosted internally behind a firewall.

Another powerful aspect of the package ecosystem is that due to the language design, packages can be combined/extended in ways that are difficult for other common languages. This means that Julia packages often interop without any additional coordination.

For example, packages that operate on data tables work without issue together in Julia. In R/Python, many features tend to come bundled in a giant singular package like Python's Pandas which has Input/Output, date manipulation, plotting, resampling, and more. There's a new Consortium for Python Data API Standards that seeks to harmonize the different packages in Python to make them more consistent (R's Tidyverse plays a similar role in coordinating their subset of the package ecosystem).

In Julia, packages tend to be more plug-and-play. For example, every time you want to load a CSV you might not want to transform the data into a dataframe (maybe you want a matrix or a plot instead). To load data into a dataframe, in Julia the practice is to use both the CSV and DataFrames packages, which help separate concerns. Some users may prefer the Python/R approach of less modular but more all-inclusive packages.

Some highlighted/recommended packages:

- Actuarial Specific (part of the *JuliaActuary.org* umbrella[3])
  - **MortalityTables**—Common tables and parametric models with survivorship calculations
  - **ActuaryUtilities**—Robust and fast calculations for common functions
  - **LifeContingencies**—Insurance, annuity, premium, and reserve maths.
- Data Science and Statistics
  - **DataFrames**—Work with datasets; similar to R's data.table and able to handle much larger datasets than Python's Pandas[4]
  - **Distributions**—Common and exotic statistical distributions
  - **GLM**—Generalized Linear Models
  - **Turing**—Bayesian statistics like STAN
  - **Gen**—Probabilistic programming
  - **OnlineStats**—Single-pass algorithms for real-time/large data
  - **CSV**—The fastest CSV reader
  - **ODBC**—Database Connections
  - **Dates**—Robust date types and functions
- Machine Learning
  - **Flux**—Elegant, GPU-powered ML
  - **Knet**—Deep learning framework
  - **MLJ**—ML models
- Notebooks
  - **IJulia**—the Julia kernel for Jupyter notebooks
  - **Pluto**—reactive/interactive notebooks that address some of the biggest complaints with Jupyter
- Visualization
  - **Plots**—Powerful but user-friendly plots and animations
  - **Queryverse**—Tidyverse-like data manipulation and plotting

Copyright © 2020 Society of Actuaries. All rights reserved.

- Dashboards
  - **Plot.ly Dash**
- Miscellaneous
  - **Optim**—Uni/Multivariate function optimization
  - **LinearAlgebra**—Built-in library for working with arrays/matrices
  - **JuMP**—Linear, Nonlinear, and other advanced optimization
  - **CUDA**—GPU programming made easier
  - **Revise**—Edit code while you work on it
- Interoperability
  - **PyCall**—use existing Python code/libraries inside Julia
  - **RCall**—use existing R code/libraries inside Julia
- Web
  - **HTTP**—Core web utilities
  - **Genie**—Full application framework
  - **Franklin**—Flexible Static Site Generator
- Documentation
  - **Weave/Literate**—Literate programming like RMarkdown
  - **Documenter**—Write your documentation as comments to your code and produce full docpages

And finally, some general resources to get started:

- JuliaLang.org, the home site with the downloads to get started and links to learning resources.
- JuliaHub indexes open-source Julia packages and makes the entire ecosystem and documentation searchable from one place.
- JuliaAcademy, which has free short courses in Data Science, Introduction to Julia, DataFrames.jl, Machine Learning, and more.
- Data Wrangling with DataFrames Cheat Sheet
- Learn Julia in Y minutes, a great quick-start if you are already comfortable with coding.
- Think Julia, a free e-book (or paid print edition) that introduces programming from the start and teaches you valuable ways of thinking.
- Design Patterns and Best Practices, a book that will help you as you transition from smaller, one-off scripts to designing larger packages and projects.

CONCLUSION

Looking at other great tools like R and Python, it can be difficult to summarize a single reason to motivate a switch to Julia, but hopefully this article piqued an interest to try it for your next project.

In an earlier article, I talked about becoming a **10x Actuary** which meant being proficient in the language of computers so that you could build and implement great things. In a large way, the choice of tools and paradigms shape your focus. Productivity is one aspect, expressiveness is another, speed one more. There are many reasons to think about what tools you use and trying out different ones is probably the best way to find what works best for you.

It is said that you cannot fully conceptualize something unless your language has a word for it. Similar to spoken language, you may find that breaking out of spreadsheet coordinates (and even a dataframe-centric view of the world) reveals different questions to ask and enables innovated ways to solve problems. In this way, you reward your intellect while building more meaningful and relevant models and analysis. ■

Alec Loudenback, FSA, MAAA, is an actuary for the Kuvare family of companies. He can be reached at *alec.loudenback@lbl.com*.

ENDNOTES

1 Python first appeared in 1990. R is an implementation of S, which was created in 1976, though depending on when you want to place the start of an independent R project varies (1993, 1995, and 2000 are alternate dates). The history of these languages is long and substantial changes have occurred since these dates.

2 Julia Computing. (2017). Aviva Solvency II Compliance. Retrieved June 04, 2020, from *https://juliacomputing.com/case-studies/aviva.html*

3 The author of this article contributes to JuliaActuary.

4 H20 AI. (2020). Database-like ops benchmark. Retrieved October 04, 2020, from *https://h2oai.github.io/db-benchmark/*

Copyright © 2020 Society of Actuaries. All rights reserved.