SOCIETY OF ACTUARIES®

Innovation and Technology

Discussion of Efficient Computational Structure of Nested Stochastic Modeling

July 2021

SOCIETY OF ACTUARIES

Discussion of Efficient Computational Structure of Nested Stochastic Modeling

WITH AN OPEN-SOURCED PYTHON CODE FOR VA VALUATION

AUTHOR Victoria Zhang, FSA, FCIA

SPONSOR Actuarial Innovation and Technology Steering Committee

CONTENTS

Discussion of Efficient Computational Structure of Nested Stochastic Modeling

WITH AN OPEN-SOURCED PYTHON CODE FOR VA VALUATION

Executive Summary

Nested stochastic (a.k.a. stochastic on stochastic or SOS) simulation is recognized as one of the best approaches to model equity linked variable annuity products (VA) as it is not only path dependent but adapts to both real-world and risk-neutral scenarios in the simulation.

There have been many studies on how to improve the efficiency of nested stochastic modeling. The research mainly comes from three different aspects, all of which are trying to reduce the actual number of computations in a nested stochastic model (NSM):

1. Policy clustering
2. Scenario reduction
3. Proxy model regression

In this paper, the author will first review recent year research on nested stochastic modeling efficiency improvement. Then, she will discuss and demonstrate the importance of efficient simulation structure and exhibit the proposed methods with Python programming language. The code will be released publicly in GitHub.

The main motivations can be summarized as follows:

1. All the previous work mentioned above is some kind of approximation of brute force nested stochastic modeling, which trades modeling accuracy in exchange for efficiency (runtime reduction). The author would like to take a step back to review whether there are ways to improve the model efficiency from a computational structure point view without the compromise of accuracy.

2. The scientific computing world has changed dramatically in the past few years due to technological innovation. The emerging technologies come from computer science, such as Python programming language, greatly lowering the barrier for actuaries to build a better model. Due to the great development of computer hardware, such as Graphics processing units (GPUs), many heavy computational tasks, such as larger machine and deep learning models, have become possible. Motivated by those exciting changes, the author intends to start building an open source project with Python so that actuaries can easily see how to use Python to simulate the NSM and build on top of it for their own purposes. This open sourced project is designed to mimic industry practice with large-sized portfolios and scenarios.

The author has proposed three different suggestions to improve NSM efficiency in this paper. These methods are not intended to try to approximate the brute force Monte Carlo method, but rather to improve the efficiency directly. The three ideas proposed are summarized below:

- Vectorization and Broadcasting of NumPy
  Vectorization is a function within NumPy (a scientific computing package within Python that helps speed up NSMs) to express operations as occurring on entire arrays rather than their individual elements. The array expression will replace explicit loops and often is much faster for the matrix operations. Broadcasting is another powerful feature of NumPy. The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations. Broadcasting accomplishes many mathematical tasks often

relegated to iteration. However, the underlying process of broadcasting is much more efficient. Iterations (e.g. loops) occur serially, acting one-by-one upon a collection; in contrast, broadcasting allows for parallel numerical calculations, which is more time efficient.

With the combination of vectorization and broadcasting, we could rearrange the calculation algorithm. Depending on the size of the portfolio versus the number of inner loop scenarios, there are two ways to rearrange a NumPy matrix to maximize the efficiency, which will be described in detail in a later section. In the case study, the runtime accelerates by about 50X with vectorization and broadcasting compared to the traditional looping structure of brute force nested stochastic modeling.

- Reverse Inner Switch Point (RISP)
  The journey of optimization did not stop with vectorization and broadcasting. The author noticed there was still redundancy for scenario generation. With one outer loop scenario, two inner loop scenarios and eight time-step points, a total of 14 different paths can be generated (1 * 2 * (8-1) as discussed in Section 4.2). If we rearrange the scenario matrix and group all the paths that switch from outer loop scenario to inner scenario by time step (i.e. the switch point), we will only need to recalculate one-time step projection for each cohort.

  Since the scenario projection is calculated backwards, from time step T to 1 for switch point, we call this method reverse inner switch point (RISP). The application of RISP further cut down the runtime by half.

- GPU acceleration for Nested Stochastic Models

Thanks to the recent advancement of the GPU, we are able to produce many large mathematical models or simulations with enormous numbers of parameters. There are many actuarial modeling software platforms, such as Aon PathWise, that harness GPUs to speed up the actuarial simulation. However, the modeling or simulation speed will not magically decrease significantly simply by moving the Central Processing Unit (CPU) version of modeling code to a computer with one or multiple GPUs. The programming methodology for GPUs is usually substantially different than CPUs and usually requires deep knowledge of both low-level programming languages[1], such as C and C++, and the GPU hardware. For example, to help the user better utilize NVIDIA GPUs, NVIDIA developed its own programming language, which is called CUDA. CuPy is the CUDA accelerated version of NumPy. The author will switch from NumPy to CuPy to show how GPUs reduce the nested stochastic simulation time. The speed goes up by about 9X to 18X with a portfolio size of 200K to 400K. Overall, we are seeing a runtime reduction of over 1000X with large-size portfolio (200K policies). An open source valuation project of VA policies in Python will exemplify and test the methods discussed.

[1] A low-level programming language is a programming language that provides little or no abstraction from a computer's instruction set architecture—commands or functions in the language map that are structurally similar to the processor's instructions. https://en.wikipedia.org/wiki/Low-level_programming_language

Section 1: Background

1.1 WHAT IS STOCHASTIC MODELING?

Financial solvency is one of the most critical areas for an insurance company to understand. The solvency of an insurance company depends on its assets and liabilities, neither of which are known exactly and depend on the projected cash flows of the policies, inflation, interest rates and the investment returns, etc. Insurers need to build models to best predict both their assets and liabilities to ensure solvency. Typically, there are two kinds of modeling approaches, deterministic and stochastic. With deterministic modeling, as long as the input is the same, the results will not change no matter how many times you run the model. This kind of modeling is usually not desirable for financial markets. Stochastic modeling is natively random and has uncertainty in its output so that it can capture the randomness of financial markets. By allowing this randomness, stochastic modeling can estimate the probability of different potential outcomes. In the finance world, the randomness usually comes from the investment return and market volatility, which makes stochastic models very useful in projecting future financial outcomes. Thus, when a stochastic model is used, the calculation usually is repeated many times to get an aggregate probability of different outcomes.

1.2 RISK-NEUTRAL VERSUS REAL-WORLD VALUATION

Risk-neutral (RN) valuation is a method developed to evaluate financial options. It assumes that there are never opportunities of arbitrage or any investments that continuously and reliably makes money with no upfront cost to the investor. Risk-neutral valuation means the options can be valued in terms of the expected payoffs, assuming that they grow on average at the risk-free rate. The expected payoff can be modeled by replicating a portfolio (composed of the underlying asset and a risk-free security), so the real rate at which the underlying grows on average does not affect the value of the option.

Risk-neutral valuations don't work well for valuing real assets in the real world because the assumption of a risk-free rate does not usually hold. Stocks must grow at a higher rate than risk-free assets given the higher risk it bears. Real-world (RW) valuation is viewed as a more realistic representation of stock returns and volatility.

In stochastic modeling, the concept of risk-neutral and real-world valuation is applied to scenario generation. An Economic Scenario Generator (ESG) uses the input parameters of two assumption settings to generate two sets of scenarios. It is worth mentioning that the difference between risk-neutral and real-world scenarios is not the individual scenario themselves; it is the probability of those scenarios occurring.[2]

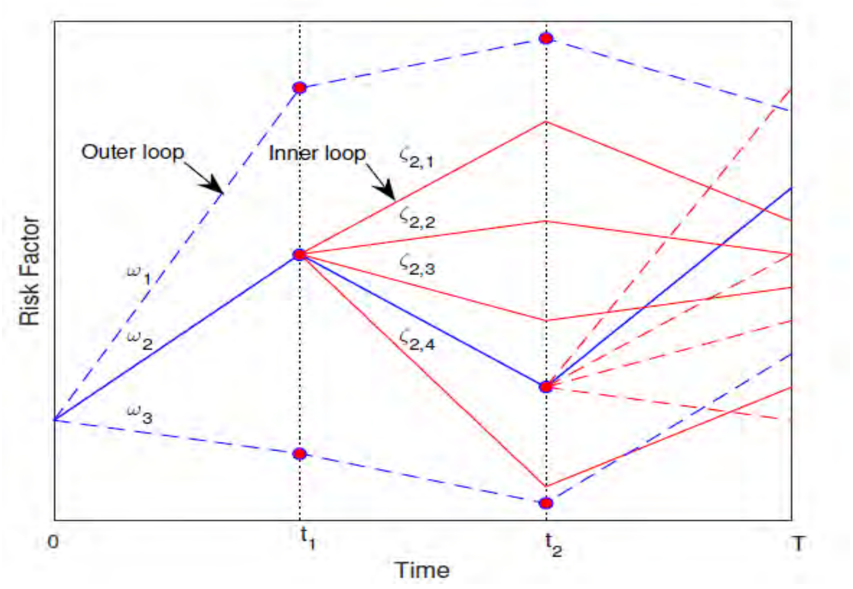
[2] Hatfield, G., "A Note Regarding Risk-Neutral and Real-World Scenarios – Dispelling a Common Misperception", SOA Article from "Product Matters!" Feb 2009- Issue 73. https://www.soa.org/globalassets/assets/library/newsletters/product-development-news/2009/february/pro-2009-iss73-hatfield.pdf

1.3 WHAT IS NESTED STOCHASTIC MODELING

Nested stochastic models are stochastic models inside of other stochastic models. They are often used when modeling components under each economic scenario which are themselves determined by stochastic scenarios in the future.

The graph from Cui, Feng, and Li [2016][3] provides a very clear illustration of a nested stochastic model:

Figure 1
ILLUSTRATION OF NESTED STOCHASTIC MODEL



Where ω_1, ω_2, ω_3 are the three outer loop scenarios

$\varsigma_{k,1}$, $\varsigma_{k,2}$, $\varsigma_{k,3}$, $\varsigma_{k,4}$ are the four inner loop scenarios derived from kth outer loop scenario.

At each node of an outer stochastic path, a set of inner stochastic paths is embedded. The branching process is repeated for the whole projection horizon. With three outer loop scenarios, four inner loop scenarios and two points of time (t1, t2), the total possible combination of movement paths would be $3*4*2 = 24$.

1.4 HOW NESTED STOCHASTIC MODELING IS USED FOR VA PRODUCTS

Variable Annuity (VA) products provide various guarantee features, including Guaranteed Minimum Maturity Benefit (GMMB), Guaranteed Minimum Death Benefit (GMDB) and Guaranteed Minimum Withdrawal Benefit (GMWB). These guarantees are essentially multi-dated, path-dependent put options where the options pay off when the policyholder's total fund value is below the guaranteed benefit.

The underlying assets of variable annuity products are the investment fund a policyholder invests in, which are mapped to different indexes. As mentioned earlier, risk-neutral scenarios are used to value financial instruments like

[3] Cui, Z., Feng, R., & Li, P., "Nested Stochastic Modeling for Insurance Companies", 2016. https://www.soa.org/globalassets/assets/files/static-pages/research/nested-stochastic-modeling-report.pdf

options, while real-world scenarios are usually suited for index prediction. The intercorrelation between underlying fund value and VA benefit payoff can be best modeled using the nested stochastic method.

Unlike the mortality risk associated with the traditional insurance product, the financial risk associated with the embedded guarantees cannot be mitigated by underwriting as much. Thus, the valuation and hedging of a VA block of business heavily relies upon a large number of scenarios of nested stochastic modeling. The outer loops are real-world stochastic paths to project the underlying market price of the total fund value. In contrast, the inner loops are risk-neutral scenarios, which calculate the payoff of VA guarantees under no-arbitrage assumptions.

The nested simulation is a two-step simulation. The outer loop simulation projects each policy's account value from time 0 to t under RW scenarios. The inner loop kicks in from time t until expiration (T) to project the future payoffs of embedded guarantees. The average present value of the payoff of all inner loops represents the expected cash flow at time t (t = 1…T) of the outer loop. On a block level, we calculate the reserve and capital based on a certain threshold of the total payoff.

Demanding computation time is one of the biggest drawbacks of nested stochastic modeling. For example, for a portfolio of 100K policies, with 2K real-world and 2K risk-neutral scenarios, a quarterly time-step projected over 30 years will require 100K*2K*2K*4*30 quarters = $4.8*10^{13}$ projections. Even with a computing power of 10^6 projections per second, we are still talking about 560 days to complete the run!

Section 2: Literature Review of Modern Research on Nested Stochastic Modeling

There has been a large amount of research done already to improve the efficiency of NSMs. After reviewing that work, we believe they can be grouped into the following three categories: policy clustering, scenario reduction, and proxy model regression to replace the inner loop. The first two focus on reducing the input data of NSM to decrease the computation time, whereas the last one uses a proxy fitting to replace the inner risk-neural loop computation. For the completeness of this report, the author will briefly introduce the core ideas behind those methods. Interested readers could refer to those papers to dive into the details of cited reference work.

2.1 POLICY CLUSTERING

Clustering, or cluster modeling, is a method that draws on the underlying theory of clustering analysis, which is widely used in many different fields. In a nutshell, it reduces the large number of samples to a much smaller amount of representative clusters. Then, we can assume all the samples within a single cluster will have the same properties and can perform the computation with one sample from each cluster, such as the cluster "center," and generalize the results to the whole group without repeating the computation. One of the most important concepts of clustering analysis is the distance measurement, which measures the similarities between any two samples. Typically, the Euclidean distance is used, and we can denote samples as multiple dimension points and compute the square root sum of two such points as the distance metric.

In the context of NSM, policy clustering will reduce the number of data points of the whole inforce portfolio, usually a very large number, to a smaller set and will only run the NSM simulation of these small selected samples.

In 2013, Gan[4] proposed a method to calculate the market value and volatilities of large portfolios of VA contracts. The method involves three steps:

I. Cluster the large VA portfolio to 1% of its original size,
II. Use the Monte Carlo method to calculate the market value and Greeks of the 1% representative contract, and
III. Use machine learning to perform the total portfolio regression based on the sample result to calculate reserve and capital.

[4] Gan, G., "Application of data clustering and machine learning in variable annuity valuation," 2013. https://www2.math.uconn.edu/~gan/ggpaper/gan2013va.pdf

With the original portfolio reduced by 99% from the clustering analysis, the proposed method is about 14 to 66X faster than the brute force Monte Carlo method. There are several barriers to using clustering analysis for large VA portfolios:

a. VA portfolios are highly non-homogeneous in nature. Risk factors, including mortality rate and benefit type, cause investment funds to vary a lot between policies. It is generally very hard to cluster VA portfolios to very small sizes.
b. From the initial determination of the number of clusters to the scaling of different risk factors, there are many manual tunings involved and the model results will be subject to the different decisions.
c. The centroid can be dragged by outliers, which will influence the estimation accuracy.

2.2 SCENARIO REDUCTION

A very comprehensive survey was done by Cui, Feng, & Li [2016][5], which discussed several methods of scenario reduction in NSM. The reduction can be achieved by reprocessing inner loop results under representative scenarios and inferring the results under the desired scenario from those under similar representative scenarios.

For example, if there are two risk factors to be considered (e.g. equity returns and interest rates), a pre-process grid matrix can be built based on the partition of the risk factors:

Table 1
ILLUSTRATION OF INNER LOOP REPROCESSING APPROACH

	$X^{(a)}_1$	$X^{(a)}_2$	$X^{(a)}_3$	$X^{(a)}_4$	$X^{(a)}_5$
$X^{(b)}_1$	$L_{1,1}$	$L_{1,2}$	$L_{1,3}$	$L_{1,4}$	$L_{1,5}$
$X^{(b)}_2$	$L_{2,1}$	$L_{2,2}$	$L_{2,3}$	$L_{2,4}$	$L_{2,5}$
$X^{(b)}_3$	$L_{3,1}$	$L_{3,2}$	$L_{3,3}$	$L_{3,4}$	$L_{3,5}$
$X^{(b)}_4$	$L_{4,1}$	$L_{4,2}$	$L_{4,3}$	$L_{4,4}$	$L_{4,5}$

where an inner loop calculation is carried out on each grid point to calculate the liability (L_{ij} : liability of i-th scenario of risk factor $X^{(a)}$ and j-th scenario of risk factor $X^{(b)}$). Then, interpolation functions can be used on the whole block level to approximate the inner loop scenario based on the pre-process grid result calculated.

The biggest drawback of this approach is it suffers from “the curse of dimensionality.” In the example above, the author partitioned the data by two risk factors to come up with the interpolation function. There are more than ten risk factors when evaluating the possible liability of a policy. The computation complexity will increase exponentially and the accuracy level of the approach will deteriorate a lot with more risk factors. In Cui, et al.’s work, the case they demonstrated is reasonably accurate when there are only two risk factors being considered, whereas its results are no longer credible after moving to four risk factors.

[5] Cui, Z., Feng, R., & Li, P., “Nested Stochastic Modeling for Insurance Companies,” 2016. https://www.soa.org/globalassets/assets/files/static-pages/research/nested-stochastic-modeling-report.pdf

Other challenges with this approach include:

- Difficult to determine the boundary points to cover all points for the interpolation
- Difficult to select grid points under higher dimensional calculations

2.3 PROXY MODEL REGRESSION TO REPLACE INNER LOOP

Although the previous two methods in theory could reduce the NSM computational burden significantly, in the current industry proxy model approximation is the more popular choice. The core idea of proxy fitting in NSM is generating an empirical approximation function to replace the inner simulation. The subsequent Monte Carlo simulation has become the new research trend in nested stochastic modeling. From least-squares regression to the generic partial differential equation approach (Cui, Feng, & Li [2016][6]) to the surrogate modeling approach (Lin & Yang, [2020][7]), each of these approaches involves building a proxy model, which is subsequently used to replace the full inner simulation of the nested stochastic modeling process.

Taking the Least Square Monte Carlo (LSMC) approach as an example, suppose the policy liability is determined by a set of risk factors $f(x) = a_0 + \sum_{i=1}^{k}(b_i \mathrm{x})$ where we are trying to calculate the weights $\vec{b}$ with a subset of the portfolio. The weights $\vec{b}$ are calculated such that the difference between the projected liability under full simulation and with the regression function is minimized for this subset.

Indeed, LSMC could significantly save the runtime of a nested stochastic model, but the training process of the regression model can be long given it is slow in convergence speed. Another practical issue faced by the LSMC approach is it is not robust enough for the change in risk factors. These changes could be economic-related market movements, an underlying assumption change, or a portfolio allocation change. The regression model needs to be re-calibrated every time a change occurs.

[6] Cui, Z., Feng, R., & Li, P., "Nested Stochastic Modeling for Insurance Companies," 2016. https://www.soa.org/globalassets/assets/files/static-pages/research/nested-stochastic-modeling-report.pdf

[7] Li, S. & Yang, S., "Fast and efficient nested simulation for large variable annuity portfolios: A surrogate modeling approach," 2020. https://www.sciencedirect.com/science/article/abs/pii/S0167668720300020

Section 3: Open Source Project for Nested Stochastic Modeling on VA Product

The primary goal of building an open source project is to demonstrate how the NSM works for large variable annuity portfolios. One of the main differences of this work from the work done by Guo Jun Gan is the choice of programming language. Compared to the Java programming language used in Gan's work, the author used Python in this project, which is higher level programming language and easier for people like actuaries without programming background to pick up. As the source code has been made available to the public, any user can test different inforce scenarios or even make changes to the assumptions.

The author used the synthetic dataset from Gan's work as the input inforce files and the SOA Economic Scenario Generator for the scenario inputs. As the first step, t a brute force nested stochastic model (NSM) was built and will be used to make the best use of NumPy. The author also shows the reader how to use CuPy, which is a GPU-accelerated version of NumPy, to further boost the computation speed. The combined acceleration of efficient NumPy code using CuPy can improve the NSM by around 1000X, which is very significant.

The contribution of this project can be summarized as follows:

- An open-source and license-free Python project for NSM in VA
- Use vectorization and broadcasting to speed NSM up by 50X (with sample of 200K policies portfolio)
- Propose a reverse inner switch point (RISP) algorithm to improve the NSM by 2X
- Show how to use CuPy to reduce the runtime of NSM by 10X (with sample of 200K policies portfolio)

In the following subsections, the author will first explain the input data and data processing architecture, then, illustrate how to use the above three speed up techniques to reduce the runtime of NSM.

3.1 NESTED STOCHASTIC MODELING SETUP

3.1.1 POLICY

One of the challenges of building an open source project for stochastic modeling for VA is the lack of a solid baseline dataset. As the inforce data of the real VA product contains the confidential information of customers, it is not easy to get that data from insurance companies. Thanks to the project of Gan and Valdez [2017][8], which generated 190,000 synthetic policies, we can use these synthetic policies as our inforce input.

Without the loss of generality, the author makes the below assumptions for the NSM:

- Three types of guarantees are offered: GMWB, GMMB, and GMDB.
- The benefit amount could roll up every quarter if there is no withdrawal yet. Once the policy is elected for withdrawal benefits, the benefit amount will decrease dollar by dollar of the withdrawal amount.
- No lapse or maintenance fees for simplicity.

[8] Gan, G. & Valdez, E., "*Valuation of Large Variable Annuity Portfolios: Monte Carlo Simulation and Synthetic Datasets*", Dependence Modeling. Vol. 5, pp. 354-374, 2017. https://www2.math.uconn.edu/~gan/software.html

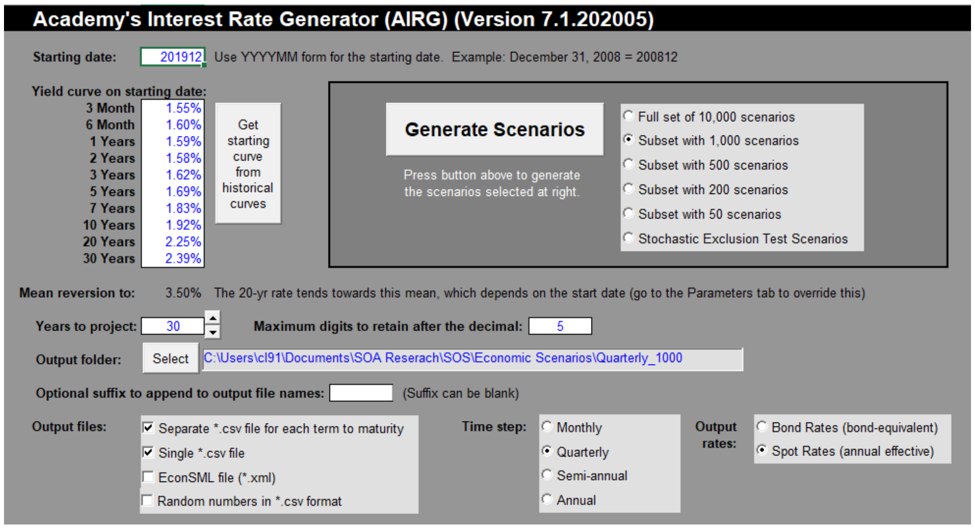
3.1.2 ECONOMIC SCENARIO GENERATOR (ESG)

Having a meaningful ESG is another challenge to building an open-sourced NSM model for VA. In this project, the ESG created by the SOA [May 2020][9] was used.

This is the specific ESG that was used and the below options were utilized to generate scenarios. The economic scenario will be in a quarterly format, for a 30-year projection with ten indexes.

Figure 2

USER INTERFACE OF SOA ECONOMIC SCENARIO GENERATOR



Academy's Interest Rate Generator (AIRG) (Version 7.1.202005)

Starting date: 201912 Use YYYYMM form for the starting date. Example: December 31, 2008 = 200812

Yield curve on starting date:

3 Month	1.55%
6 Month	1.60%
1 Years	1.59%
2 Years	1.58%
3 Years	1.62%
5 Years	1.69%
7 Years	1.83%
10 Years	1.92%
20 Years	2.25%
30 Years	2.39%

Get starting curve from historical curves

Generate Scenarios

Press button above to generate the scenarios selected at right.

( ) Full set of 10,000 scenarios
(●) Subset with 1,000 scenarios
( ) Subset with 500 scenarios
( ) Subset with 200 scenarios
( ) Subset with 50 scenarios
( ) Stochastic Exclusion Test Scenarios

Mean reversion to: 3.50% The 20-yr rate tends towards this mean, which depends on the start date (go to the Parameters tab to override this)

Years to project: 30 Maximum digits to retain after the decimal: 5

Output folder: Select C:\Users\cl91\Documents\SOA Reserach\SOS\Economic Scenarios\Quarterly_1000

Optional suffix to append to output file names: (Suffix can be blank)

Output files:
[✓] Separate *.csv file for each term to maturity
[✓] Single *.csv file
[ ] EconSML file (*.xml)
[ ] Random numbers in *.csv format

Time step:
( ) Monthly
(●) Quarterly
( ) Semi-annual
( ) Annual

Output rates:
( ) Bond Rates (bond-equivalent)
(●) Spot Rates (annual effective)

Also, for the purposes of this project, the author used the SOA ESG to generate two sets of ESGs with 1000/500 scenarios each, which were used as real-world and risk-neutral scenarios in the modeling. The modeling does not make any assumptions regarding the ESG; therefore, the users can replace the SOA ESG with their own ESG when using the code for their own purposes. Note that in the real NSM simulation for VA products, RW and RN scenarios are generally different. Those differences do not matter in this project as we are only interested in the efficiency aspect.

The important parameters used in this project:

- Quarterly projections, so projection frequency per year is 4
- Total of 30 projection years
- Total projection points T = 4 * 30 = 120

[9] SOA Economic Scenario Generator, https://www.soa.org/resources/tables-calcs-tools/research-scenario/ https://www2.math.uconn.edu/~gan/software.html

3.1.3 SYSTEM ARCHITECTURE

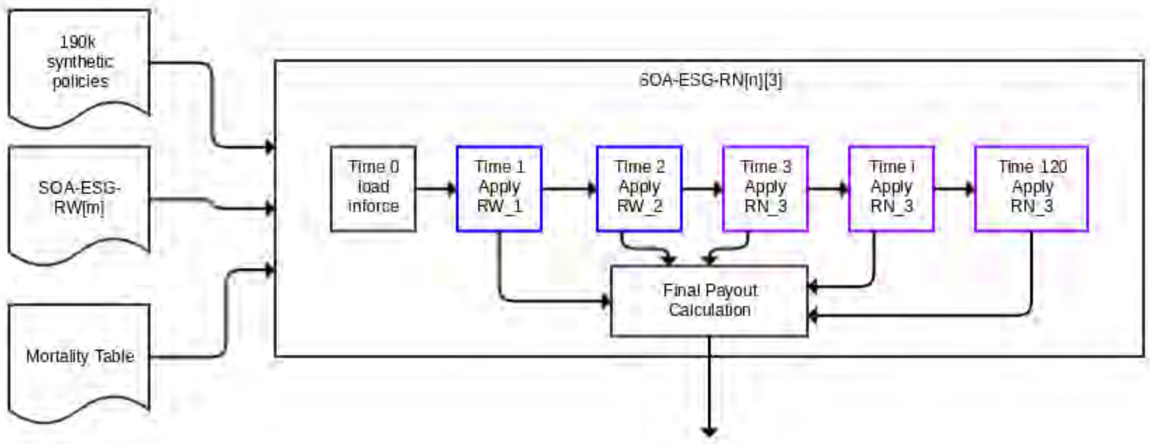
To facilitate the discussion and analysis of the computational runtime complexity, the below symbols were used to denote the total number of each input data point.

- T: total number of projection points. T = 120 (i.e. 30 years * 4 quarters)
- M: total number of RW scenarios
- N: total number of RN scenarios
- I: total number of policies. I = 190,000
- F: number of investment funds available. F is fixed at 10 in this project.

The below figure displays the computational structure of this project. The input data are the 190K synthetic policies and the mortality table is fixed and will not change while we are looping through the different RWs.

- RW[m] denotes the mth real-world scenario (m in the range of [1, M]), which is nothing but a T by F or (120 by 10) matrix
- RN[n][t] denotes the nth risk-neutral (n in the range of [1, N]) and t represents when we switch to RN[n] for a particular RW.
- As for a single RW and RN, we can have up to 120 switching points, therefore, the final payout calculator in the below figure will generate 120 outputs and will take the average of those 120 outputs as the average payout for a single RW.

Figure 3
VA PORTFOLIO VALUATION PROCESS FLOW



3.2 NAIVE BRUTE FORCE NESTED STOCHASTIC MODELING

For people who are new to programming, it is very tempting to write a naive NSM, which will be demonstrated later. This way of modeling is not even efficient by the "naïve brute force" standard.

Under the brute force valuation model, the number of simulations we are looking for is $M * N * I$

Where, M: number of RW scenarios

N: number of RN scenarios

I: number of policies in the portfolio

As mentioned earlier, for one RW scenario and one RN scenario, there are a possible T scenario paths generated depending on the switch time point where, for each path, there are T+1 time-steps (0, 1, 2, …, T-1, T).

For the 190,000 synthetic policy portfolios, with 1,000 RW scenarios and 500 RN scenarios, there are 190,000 * 1,000 $*$500 $*$ 120 = 1.14 $*$ 10^{13} simulations. Assuming a modern CPU could process 5,000 simulations per second, we are talking about 72 years to complete the valuation.

Algorithm 1: Naive Brute Force Nested Stochastic Valuation of a VA Contract

```
for real_world_scenario_id in range(0, M):

 for risk_neutral_scenario_id in range(0, N):

   for inforce_id in range(0, I):

     fund_value_10x1 =...

     for switch_points in range(1, T):

         fund_return_ratio = [rw_0, rw_1, ..., rn_switch_points,..., rn_T]

         for r in range(len(fund_return_ratio)):

           for t in range(1, T):

             ratio_10x1 = fund_return_ratio[r, t]

             benefit_calculation(ratio_10x1, fund_value_10x1)

 final_rw_payout_calculation()
```

Mathematically, there is nothing wrong with the naive approach and we would still get the correct results, but the it is too slow to have any real world use. From the testing done on a computer with Intel Core i7-8850H Hexa-core 2.6 GHz CPU, it takes about 0.2 seconds to process one single policy with one real-world and one risk-neutral scenario. With 190K policies, it will take a few years to complete the whole simulation. That's why it is very important to look for ways to optimize the computational efficiency of the NSM.

Most of the academic research and study in the past focused more on how to reduce the number of real-world M, N (scenario reduction, proxy model reduction) or I (policy clustering) to reduce the runtime of the NSM simulation. In the following discussion, the author will mostly look into the underlying computational structure of the NSM and will try to improve upon it from the coding efficiency point of view. It is important to note that the proposed modeling method used in this project can be combined with other efficiency improvement methods to further reduce the runtime complexity of NSM.

Section 4: Efficiency Improvement of Nested Stochastic Modeling

4.1 VECTORIZATION AND BROADCASTING OF NUMPY

The NSM process involves a lot of loops. As was shown in the above Algorithm 1, we would have six layers of loops for the naive approach and the runtime complexity for it is at least $O(M * N * I * T^2)$. We need to evaluate the payout at each time point and for each real-world path we could have T different switch points, thus we have T^2 in the complexity analysis.

The vectorization and broadcasting technique is to speed up the processing by simply reordering some of the matrix products, which can be explained by the below examples.

One of the most common computations of NSM is applying the fund return ratio to the funds from policies. In the below example, assume we have 10K policies and all the funds of each policy have been allocated into ten different funds. The scenario would generate the fund return ratio, which would be a vector with a size of 10.

Sample 2 Algorithms:

```
import numpy as np
num_policies = 10000
num_fund = 10
fund_return_ratio = np.random.uniform(0.8, 1.2, num_fund)
inforce_fund = np.random.uniform(0.8, 1.2, [num_policies, num_fund])

# Simply loop
for i in range(num_policies): # Loop the policies
    for in j in range(num_fund) # Loop the fund
        inforce_fund[i][j] = inforce_fund[i][j] * fund_return_ratio[j]

# Vectorization and Broadcasting
inforce_fund = inforce_fund * fund_return_ratio
```

In the code version 1 (simple loop), we would have to loop all the policies and loop over all the funds to update the projected fund values. In the code version 2 (vectorization and broadcasting), the vectorization and broadcasting feature of NumPy was used, which would call the NumPy matrix operation once and will apply the fund return ratio to all the policies in one shot. This turns out to be at least 30X faster than version 1.

Vectorization is a powerful function within NumPy to express operations as occurring on entire arrays rather than their individual elements. The array expression would replace the explicit loops, and often is much faster for matrix operations.

For example, the projection of returns of the ten investment funds and their dollar amounts can be transformed into element-wise product $A \odot B$ (Hadamard Product) where a fund return forms a scenario and $B_{10\times1}$ is the fund allocation of a policy.

The Hadamard product is a binary operation that takes two matrices of the same dimension and produces another matrix (same size as input), where each element is the product of the corresponding inputs.

$$A_{4\times1} \odot B_{4\times1} = [a_1, a_2, a_3, a_4] \odot [b_1, b_2, b_3, b_4] = [a_1 b_1, a_2 b_2, a_3 b_3, a_4 b_4]$$

With the help of vectorization, we can calculate the combined return of the ten funds in one shot as follows:

$$M \times N \times I \times T \times [rate_1, rate_2, rate_3, .., rate_{10}] \odot [Fv_1, Fv_2, Fv_3, ..., Fv_{10}]$$

The smallest computational unit is a product of two 10×1 vectors, which is still too granular for a large VA portfolio. That is where the term "broadcasting" comes in handy, which can further improve the computational speed of NSM.

Broadcasting is another powerful feature of NumPy. The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations. Broadcasting accomplishes many mathematical tasks often relegated to iteration. However, the underlying process of broadcasting is much more efficient. Iterations (e.g. loops) occur serially, acting one-by-one upon a collection; in contrast, broadcasting allows for parallel numerical calculations, which is more time efficient. The process of broadcasting can be conceptualized by imagining that, with one dimension fixed, a smaller array is "spread" or "stretched" to meet another larger-sized array.

"The Broadcasting Rule

To broadcast, the size of the trailing axes for both arrays in an operation must either be the same size or one of them must be one."[10]

The trailing axes is the fixed dimension, which is usually the smallest dimension of the calculation. In this case, it would be ten, the number of investment funds or fund returns. The discussion will be carried into two cases:

Case1: when the policy number is much more than the inner loop scenarios (I>> N)

In the example, the number of contracts in the portfolio is 190K, much larger than the possible inner loop scenario combination $500 \times 120 = 600K$; we want to replace the policy iteration with a vector calculation. To achieve that, we would build a $190K \times 10$ NumPy matrix where the investment fund allocations of all inforce are put into one vector. The fund returns of a particular period t will be applied to all policies together. Mathematically, the portfolio level calculation is:

$$M * N * T * \begin{bmatrix} rate\_1 \\ rate\_2 \\ \vdots \\ rate\_10 \end{bmatrix}_{10*1} \odot \begin{bmatrix} FV\_1^{1} & FV\_2^{1} & \cdots & FV\_9^{1} & FV\_10^{1} \\ FV\_1^{2} & FV\_2^{2} & \dots & FV\_9^{2} & FV\_10^{2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ FV\_1^{189,999} & FV\_2^{189,999} & \dots & FV\_9^{189,999} & FV\_10^{189,999} \\ FV\_1^{190,000} & FV\_2^{190,000} & \cdots & FV\_9^{190,000} & FV\_10^{190,000} \end{bmatrix}_{I*10}$$

The fund return ratio, $A_{10\times1}$, is being stretched to cover all 190K inforce policies. Intuitively, the portfolio can be viewed as a giant super policy with $190K \times 10$ investment funds. The giant investment mapping is applied by time step t, by scenarios like a single contract. We have successfully brought down the number of iterations to $M \times N \times T$ by putting all the inforce records into one giant vector.

[10] NumPy in Python |Set2 (Advanced) https://www.geeksforgeeks.org/numpy-python-set-2-advanced/

Case2: When the inner loop scenario combination is dominant over the policy number (N >> I)

Suppose we have a relatively small portfolio, only 100 policies, but need many RN scenarios for inner loop simulation (this can be the case for VA production pricing and development). We will try to broadcast the (10×1)investment fund value to all scenarios to replace the N loops of iteration calculations.

The algorithm efficiency we are looking for is:

$$M * I * T \begin{bmatrix} FV\_1 \\ FV\_2 \\ \vdots \\ FV\_10 \end{bmatrix}_{10*1} \odot \begin{bmatrix} Rate\_1^{1,1} & Rate\_2^{1,1} & \dots & Rate\_9^{1,1} & Rate\_10^{1,1} \\ Rate\_1^{1,2} & Rate\_2^{1,2} & \cdots & Rate\_9^{1,2} & Rate\_10^{1,2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Rate\_1^{1,T} & Rate\_2^{1,T} & \cdots & Rate\_9^{1,T} & Rate\_10^{1,T} \\ Rate\_1^{2,1} & Rate\_2^{2,1} & \cdots & Rate\_9^{2,1} & Rate\_10^{2,1} \\ Rate\_1^{2,2} & Rate\_2^{2,2} & \dots & Rate\_9^{2,2} & Rate\_10^{2,2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ Rate\_1^{N,T-1} & Rate\_2^{N,T-1} & \dots & Rate\_9^{N,T-1} & Rate\_10^{N,T-1} \\ Rate\_1^{N,T} & Rate\_2^{N,T} & \cdots & Rate\_9^{N,T} & Rate\_10^{N,T} \end{bmatrix}_{(NT)*10}$$

At time t with a particular real-world scenario, there are a possible $N \times T$ paths of the investment fund projection. The fund value $[Fv_1, Fv_2, \dots, Fv_{10}]$ is being mapped to all rates at the time t of the $N \times T$ paths to get the return of the next period.

In both cases, we use 10×1 as the trailing axes of broadcasting. Case1 stretches the fund rate return of each scenario to cover all inforce policies, while Case2 stretches the fund value of each policy to cover all possible scenario combinations at a particular time point. Depending on the size of the portfolio (I) versus the size of possible scenario combinations ($N \times T$), we can take full advantage of vectorization and broadcasting in a NSM to remove as many iterations as possible.

Insurance companies usually hold a VA portfolio with a very large number of policies, therefore, the Case1 from the previous discussion is more commonly seen. After vectorization and broadcasting of the nested stochastic calculation, we have an algorithm like below:

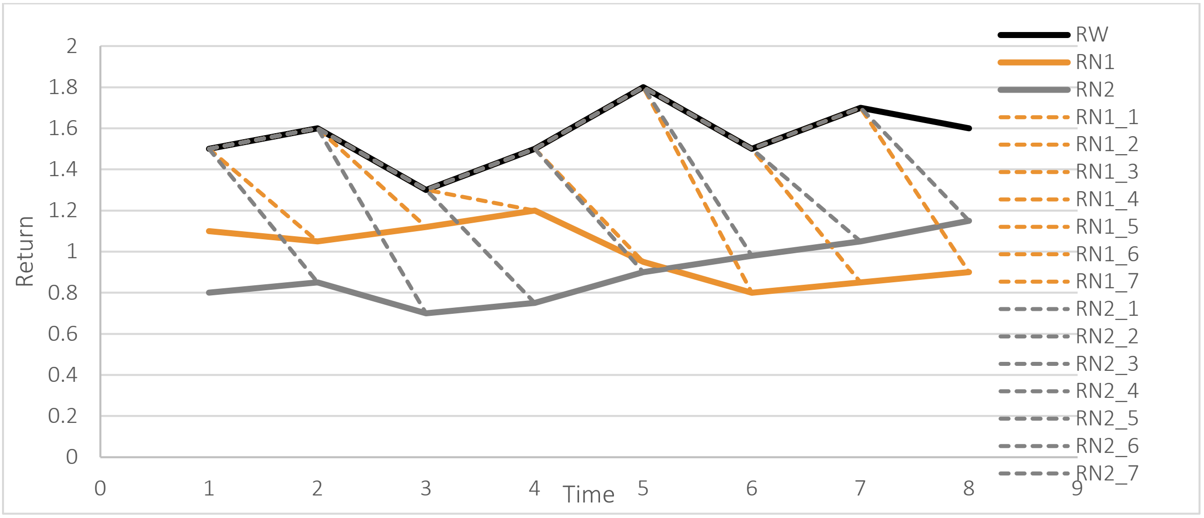
```
Loop M = 1000 # outer loop of RW scenarios
  Loop N = 500 # inner loop of RN scenarios
     Loop t = (0, 1, …. , 120)  #time step
                    Portfolio $Payout\ (t) = f([FV(t)]_{190,000*10})$, where
                        $[FV\ (t)]_{190,000*10} = [Rate\ (t)]_{10*1} \odot [FV\ (t-1)]_{190000*10}$
```

4.2 REVERSE INNER SWITCH POINTS (RISP)

The journey of optimization did not stop there. The author noticed there were still redundancies in the scenario generation (M ∗ N). In the graph below, she demonstrated that, with one outer loop scenario, two inner loop scenarios, and eight time-step points, a possible 14 different paths could be generated (1∗2∗ (8-1) =14).

Figure 4

ILLUSTRATION OF 1RW, 2RN SCENARIO COMBINATION ON 8 TIME STEPS



With a particular RW scenario m, all the possible paths we could have from time 0 to 120 would be:

$$
\left.\begin{bmatrix}
RW_0^m & RW_1^m & \dots & RW_{118}^m & RW_{119}^m & RN_{120}^1 \\
RW_0^m & RW_1^m & \cdots & RW_{118}^m & RN_{119}^1 & RN_{120}^1 \\
RW_0^m & RW_1^m & \cdots & RN_{118}^1 & RN_{119}^1 & RN_{120}^1 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
RW_0^m & RN_1^1 & \cdots & RN_{118}^1 & RN_{119}^1 & RN_{120}^1 \\
RW_0^m & RW_1^m & \cdots & RW_{118}^m & RW_{119}^m & RN_{120}^2 \\
RW_0^m & RW_1^m & \dots & RW_{118}^m & RN_{119}^2 & RN_{120}^2 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
RW_0^m & RW_1^m & \dots & RN_{118}^N & RN_{119}^N & RN_{120}^N \\
RW_0^k & RN_1^N & \cdots & RN_{118}^N & RN_{119}^N & RN_{120}^N
\end{bmatrix}\right._{120N*121}
$$

Cohort of paths with RN scenario ind1

Rearranging the matrix, we can transform all the paths that switch from RW scenario to RN scenarios by time step, i.e. the switch point.

$$\begin{bmatrix}
RW_0^m & RW_1^m & \cdots & \cdots & \cdots & \cdots & \cdots & RW_{118}^m & RW_{119}^m & RN_{120}^1 \\
RW_0^m & RW_1^m & \cdots & \cdots & \cdots & \cdots & \cdots & RW_{118}^m & RW_{119}^m & RN_{120}^2 \\
RW_0^m & RW_1^m & \cdots & \cdots & \cdots & \cdots & \cdots & RW_{118}^m & RW_{119}^m & RN_{120}^3 \\
RW_0^m & RW_1^m & \cdots & \cdots & \cdots & \cdots & \cdots & RW_{118}^m & RW_{119}^m & RN_{120}^4 \\
RW_0^m & RW_1^m & \cdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
RW_0^m & RW_1^m & \cdots & \cdots & \cdots & \cdots & \cdots & RW_{118}^m & RW_{119}^m & RN_{120}^N \\
RW_0^m & RW_1^m & \cdots & \cdots & \cdots & \cdots & \cdots & RW_{118}^m & RN_{119}^1 & RN_{120}^1 \\
RW_0^m & RW_1^m & \cdots & \cdots & \cdots & \cdots & \cdots & RW_{118}^m & RN_{119}^2 & RN_{120}^2 \\
RW_0^m & RW_1^m & \cdots & \cdots & \cdots & \cdots & \cdots & RW_{118}^m & RN_{119}^3 & RN_{120}^3 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
RW_0^m & RW_1^m & \cdots & \cdots & \cdots & \cdots & \cdots & RW_{118}^m & RN_{119}^N & RN_{120}^N \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
RW_0^m & RW_1^m & \cdots & RW_{t-1}^m & RW_{t-1}^m & RN_t^1 & RN_t^1 & \cdots & \cdots & \cdots \\
RW_0^m & RW_1^m & \cdots & RW_{t-1}^m & RW_{t-1}^m & RN_t^2 & RN_t^1 & \cdots & \cdots & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
RW_0^m & RN_1^1 & \cdots & \cdots & \cdots & \cdots & \cdots & RN_{118}^1 & RN_{119}^1 & RN_{120}^1 \\
RW_0^m & RN_1^2 & \cdots & \cdots & \cdots & \cdots & \cdots & RN_{118}^1 & RN_{118}^1 & RN_{120}^2 \\
RW_0^m & RN_1^3 & \cdots & \cdots & \cdots & \cdots & \cdots & RN_{118}^1 & RN_{118}^1 & RN_{120}^2 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
RW_0^m & RN_1^{N-1} & \cdots & \cdots & \cdots & \cdots & \cdots & RN_{118}^{N-1} & RN_{119}^{N-1} & RN_{120}^{N-1} \\
RW_0^m & RN_1^N & \cdots & \cdots & \cdots & \cdots & \cdots & RN_{118}^N & RN_{119}^N & RN_{120}^N
\end{bmatrix}_{120N*121}$$

Cohort1: paths switch to RN scenario at t=120

Cohort2: paths switch to RN scenario at t=119

Cohort: paths switch to RN scenario at t=1

We notice that the adjacent cohort of rows all have the same first t points and the divergence only happens starting from time t. If we read the scenarios backwards (Time T to 1), we would only need to recalculate one time step moving down to the next cohort.[11]

Taking a ten-point projection as an example, we start the algorithm by taking a real-world projection until time 10 and cash out the results. For the first path, the algorithm calculates backwards by replacing the time-10 rate only. The second path reads the results from path1 and only needs to re-evaluate time 9, time 10.

Table 2

ILLUSTRATION OF RISP 10 SAMPLE PATHS

TimeStep	0	1	2	3	4	5	6	7	8	9	10
Paths	RW(0)	RW(1)	RW(2)	RW(3)	RW(4)	RW(5)	RW(6)	RW(7)	RW(8)	RW(9)	RW(10)
1	RW(0)	RW(1)	RW(2)	RW(3)	RW(4)	RW(5)	RW(6)	RW(7)	RW(8)	RW(9)	RN(10)
2	RW(0)	RW(1)	RW(2)	RW(3)	RW(4)	RW(5)	RW(6)	RW(7)	RW(8)	RN(9)	RN(10)
3	RW(0)	RW(1)	RW(2)	RW(3)	RW(4)	RW(5)	RW(6)	RW(7)	RN(8)	RN(9)	RN(10)
4	RW(0)	RW(1)	RW(2)	RW(3)	RW(4)	RW(5)	RW(6)	RN(7)	RN(8)	RN(9)	RN(10)
5	RW(0)	RW(1)	RW(2)	RW(3)	RW(4)	RW(5)	RN(6)	RN(7)	RN(8)	RN(9)	RN(10)
6	RW(0)	RW(1)	RW(2)	RW(3)	RW(4)	RN(5)	RN(6)	RN(7)	RN(8)	RN(9)	RN(10)
7	RW(0)	RW(1)	RW(2)	RW(3)	RN(4)	RN(5)	RN(6)	RN(7)	RN(8)	RN(9)	RN(10)
8	RW(0)	RW(1)	RW(2)	RN(3)	RN(4)	RN(5)	RN(6)	RN(7)	RN(8)	RN(9)	RN(10)
9	RW(0)	RW(1)	RN(2)	RN(3)	RN(4)	RN(5)	RN(6)	RN(7)	RN(8)	RN(9)	RN(10)
10	RW(0)	RN(1)	RN(2)	RN(3)	RN(4)	RN(5)	RN(6)	RN(7)	RN(8)	RN(9)	RN(10)

[11]Note: Forward reading (from time 0 to T) would also work, however, backwards is more convenient for coding.

Assuming for each outer loop (RW) scenario that there are N possible inner loop (RN) scenarios, each time step (max T) is a possible switch point from outer loop to inner loop scenario. Let's evaluate the cost of the proposed method by number of simulations[12]:

Step1: Project the full path under RW scenario, cash out the results for the next step

Step2: When SP = T-1, read the results from step1 up to time T-1, project N scenarios from time step T.

Cost = N, project N scenarios for time step T.

When SP = T-2, read the results from step1 up to time T-2, project N scenarios from time step T-1, T

Cost = 2∗N…

When SP = 1, read the results from step1 for T=0, project N scenarios from time 2 … T

Cost = (T-1)*N

Total cost of one outer loop scenario valuation = $f(x) = \sum_{i=1}^{T-1} iN$ = $N * \frac{1}{2}T^2$

Recall that, in the original brute force approach, for every single outer loop scenario, we need to loop all the inner loop scenarios at each switch point for each projection period. The cost of the brute force approach is NT^2. The RISP method could reduce the runtime by half compared to the brute force approach.

4.3 GPU ACCELERATION FOR NESTED STOCHASTIC MODELING

Thanks to the recent advancements of the Graphics Processing Units (GPUs), we are able to perform large mathematical modeling or simulations with enormous amounts of parameters. For example, in the artificial intelligence (AI) domain, one of the most famous models is named Generative Pre-trained Transformer 3 (GPT-3), which is a language prediction model developed by the company, OpenAI. This model has more than 175 billion parameters that need to be trained; modern GPUs make this possible, although the training cost is over $4.6M USD. GPUs have been widely used in mathematical modeling and simulation in many different areas; however, they are not yet widely used in the actuarial field.

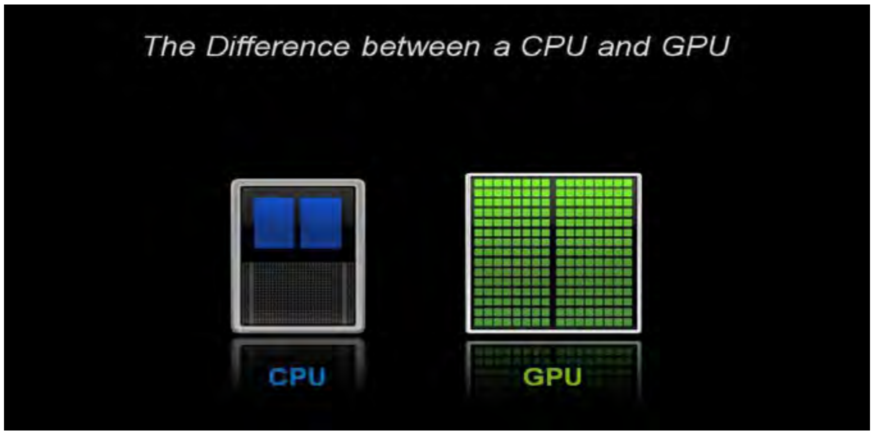
There are many actuarial modeling software platforms, such as Aon PathWise or GGY Axis, that harness GPUs to speed up actuarial simulations. However, the modeling or simulation speed will not magically decrease significantly simply by moving the Central Processing Unit (CPU) version of modeling code to a computer with one or multiple GPUs. The programming methodology for GPUs is usually significantly different compared to CPUs and usually requires deep knowledge of both low-level programming languages, such as C and C++, and the GPU hardware. For example, to help the user better utilize NVIDIA GPUs, NVIDIA developed its own programming language, which is called CuPy.

[12] Note: One time step under one scenario is counted as one simulation.

4.3.1 NSM MODELING DIFFERENCES BETWEEN GPU AND CPU

CPUs are often regarded as the brain of a computer. In the early 1980s, CPUs usually just had one core, which meant it could only run one task at a time. The Intel i7-10700K chip is widely recognized as one of the most powerful CPUs in 2021, which has eight cores and can run 16 threads or 16 tasks at the same time. Parallelization processing is the key to reducing the runtime for processing large amounts of data (in this case, large amounts of policies or large amounts of inner paths).

Figure 5
DIFFERENCE BETWEEN A CPU AND GPU[13]



We can see that, even with 20 years of technological advancement, the number of CPU cores did not improve that much, although CPU's became much more powerful and can process one task faster. As the above figure the author took from NVIDIA's website shows, GPUs usually have thousands of cores by design. Also, the memory and caches of GPUs are designed to make sure we can run multiple parallel processes efficiently. For example, NVIDIA's GeForce RTX 3060 Ti is a consumer-grade gaming GPU. It has 4,864 cores, which means that we can run 4,864 tasks in parallel. To make the best use of processors with multiple cores, we need to write our simulation code with a different mindset. The JAVA NSM simulation platform from Gan & Valdez [2018][14] demonstrates how to do it in a CPU, where the author breaks the policies and computational tasks into batches and dispatch those pitches across multiple CPU cores for parallel processing. The same concept can be applied to GPUs also, however, we need to get familiar with low-level CUDA-programming APIs to achieve paralleling, which is a bit challenging for actuaries without a computer science background. Therefore, we can use CuPy, which is the CUDA-accelerated version of NumPy, to show how GPUs reduce the nested stochastic simulation time.

[13] Source: blogs.nvidia.com

[14] Gan, G., & Valdez, E., *"Valuation of Large Variable Annuity Portfolios: Monte Carlo Simulation and Synthetic Datasets"*, Dependence Modeling. Vol. 5, pp. 354-374, 2017. https://www2.math.uconn.edu/~gan/software.html

4.3.2 CUPY: GPU VERSION OF NUMPY

Switching from NumPy to CuPy is quite straightforward as the interface of CuPy is highly compatible with NumPy; in most cases, it can be used as a drop-in replacement. Replacing NumPy with CuPy in Python could move computation to a GPU as it supports standard numerical data types, array indexing, slices, transposing, reshaping, and broadcasting.

Figure 6
CUPY VERSUS NUMPY SPEEDUP[15]

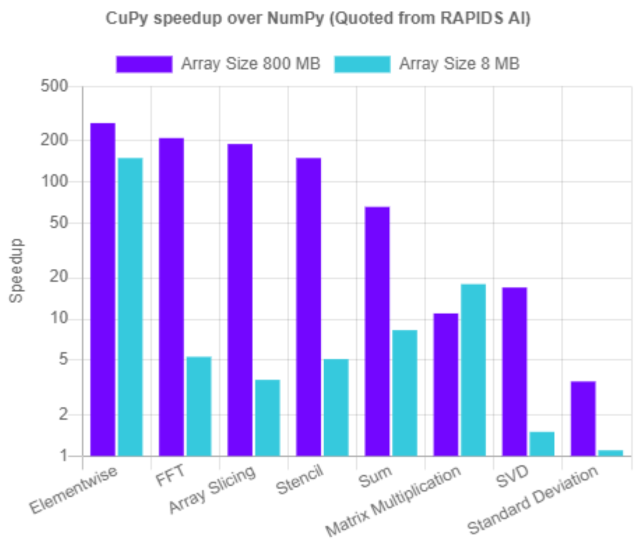


Figure 6 is a famous CuPy speedup comparison with NumPy. For element-wise calculations, the speedup ranges between 150X to 350X. However, it is important to note that the speedups you get are dependent on the size of the data you are working with. A simple rule of thumb is the speedup drastically increases when we have about ten million data points and gets faster still. Once we cross the 100 million-point mark[16]. In practice, a VA portfolio evaluated on a nested stochastic model could easily exceed the 100 million benchmarks, thus CuPy should be the go-to solution for most insurance companies.

A sample demo to estimate model efficiency is presented in Appendix A. In the demo, we see the efficiency of CuPy over NumPy when the inforce number is greater than 10,000 (with 100*100 scenario loops, ten investment funds and 121 projection points). The speedup goes to about 9X and 18X with portfolio sizes of 200K and 400K.

To show the runtime reduction, the processing time was summarized per policy with each efficiency improvement method on various portfolio sizes. Taking a portfolio of 200,000 policies as an example, the original naïve brute force method takes about 77 seconds to process each policy. With the broadcasting feature, we can rearrange the calculation algorithm and successfully bring down the average processing time to 1.45 seconds per policy. We can then replace NumPy with the CuPy package for calculation to maximize the benefit of the GPU's multi-core. The

[15] Source: https://cupy.dev/

[16] Seif, G., "Here's How to User CuPy to Make NumPy Over 10X Faster," 2019 https://towardsdatascience.com/heres-how-to-use-cupy-to-make-numpy-700x-faster-4b920dda1f56#:~:text=CuPy%20is%20a%20library%20that,leveraging%20the%20CUDA%20GPU%20library.&text=CuPy's%20interface%20is%20a%20mirror,boom%20you%20have%20GPU%20speedup.

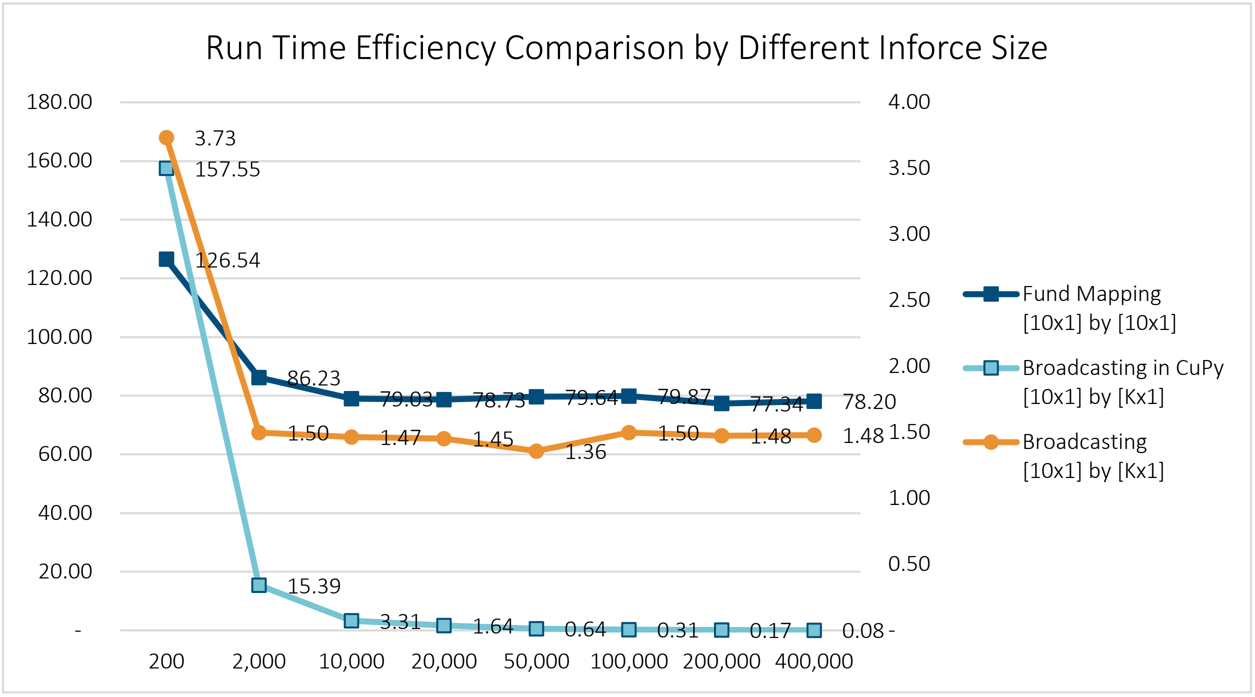
average processing time is further cut down to 0.17 seconds per policy. With both broadcasting and CuPy, we are looking at about 468X faster speed!

In general, the benefits of CuPy programming accelerates with larger-sized portfolios (i.e. 50K+) than smaller ones. As there are overhead costs to warm up the GPU, with a small-sized portfolio the average process time could be longer than the naïve brute force approach (e.g. when the inforce portfolio size is 200, the average processing time per policy is longer with CuPy optimization).

Table 3
TOTAL SPEED UP COMPARISON ON VARIOUS INFORCE SIZE (WITH 10 INVESTMENT FUNDS)

K (inforce size)	Fund Mapping [10x1] by [10x1]	Broadcasting [10x1] by [Kx1]	Broadcasting in CuPy [10x1] by [Kx1]	Total Speedup
200	126.54	3.73	157.55	0.80
2,000	86.23	1.50	15.39	5.60
10,000	79.03	1.47	3.31	23.87
20,000	78.73	1.45	1.64	48.03
50,000	79.64	1.36	0.64	125.00
100,000	79.87	1.50	0.31	260.80
200,000	77.34	1.48	0.17	468.45
400,000	78.20	1.48	0.08	986.56

Figure 7
RUNTIME EFFICIENCY COMPARISON BY DIFFERENT INFORCE SIZES



4.4 OPEN SOURCE OF EFFICIENT COMPUTATIONAL STRUCTURE OF NESTED STOCHASTIC MODELING

“Talk is cheap. Show me the code.” - Linus Torvalds

With all the methods proposed above, the valuation model was rebuilt using the nested stochastic approach. The full script is uploaded at: https://github.com/rranxxi/soa_nested_stochastic.

Section 5: Conclusions

Actuaries have a long history with stochastic modeling. The fact that stochastic modeling can capture the randomness of the financial market would allow a stochastic model to estimate the probability of different potential outcomes. These probabilities could further be translated into various hedging positions of ALM, financial results of valuations, and even business decisions with respect to the pricing of a new product. To actuaries, stochastic modeling is a powerful tool to gauge market movement.

NSM is often used when the modeling components under each economic scenario are themselves determined by stochastic scenarios in the future. The computational barrier of a normal stochastic model would be aggravated even more on a nested stochastic model. Thus, most of the research and testing focuses on how to improve the efficiency of a nested stochastic model.

The computational structure of a NSM has been the elephant in the room over the past few years. While we were all busy talking about the different approximation methods to avoid a true brute force NSM, it's easy to lose track of the more fundamental question: how to set up an efficient computational algorithm for an NSM?

In this paper, three methods have been suggested that can be added on top of existing enhancement methods to improve NSM efficiency. What's more important is that there is no compromise to the accuracy of a NSM model. The test results showed that we could achieve a 100% matching with the naïve brute force approach. The techniques discussed in this paper include vectorization plus broadcasting, RISP and CuPy programming. We achieved over 1000X runtime reduction by coming up with the most efficient modeling structure. All those methods are not to try to approximate the brute force Monte Carlo method, but rather to improve the efficiency directly.

The hope with this paper is to set a good foundation and platform for future research work, such as Machine Learning approximation to NSM.

Section 6: Acknowledgments

The researcher’s deepest gratitude goes to those without whose efforts this project could not have come to fruition: the Project Oversight Group and others for their diligent work overseeing the project and reviewing and editing this report for accuracy and relevance.

Project Oversight Group members:

Leandro Ao, FSA, FIAA

Erica Chan, FSA, CERA

Rick Hayes, ASA, CERA, MAAA

Stephen Krupa, FSA, MAAA

Gabriel Penagos, PhD, MSc

Alan (Xuefeng) Wang, MSc

At the Society of Actuaries:

Korrel Crawford, Senior Research Administrator

Mervyn Kopinsky, Senior Experience Studies Actuary, FSA, EA, MAAA

David Schraub, Senior Practice Research Actuary, FSA, AQ, CERA, MAAA

The author’s deepest gratitude also goes to the volunteers who generously shared their wisdom, insights, advice, guidance, and arm’s-length review of this study prior to publication. Any opinions expressed may not reflect their opinions nor those of their employers. Any errors belong to the author alone.

Appendix A: Demo of Efficiency Comparison

```
[5]: import cupy as cp # import cupy
import numpy as np # import numpy
from datetime import datetime
import time
[6]: num_rw_scenarios = 100
num_rn_scenarios = 100
num_funds = 10
num_proj_points = 121
num_inforces = 200000
[7]: inforce = np.random.uniform(0, 10000, [num_inforces, 10])
fund_return_ratio = np.random.uniform(0.8, 1.2, [num_proj_points, num_funds])
```

0.0.1 Speed of 10x1 x 10x1 elementwise product

```
[8]: s = time.time()
for t in range(num_proj_points): # loop the time
for inforce_id in range(num_inforces):
ret = fund_return_ratio[t] * inforce[inforce_id]
e = time.time()
delay0 = (e - s) / num_inforces * 1e6
print('Delay of ', delay0, 'us')
Delay of 77.69979596138 us
```

0.0.2 Speed of (10, 1) x (num_of_inforce, 10) elementwise product

```
[9]: s = time.time()
for t in range(num_proj_points): # loop the time
ret = fund_return_ratio[t] * inforce
e = time.time()
delay1 = (e - s) / num_inforces * 1e6
print('Delay of ', delay1, 'us')
Delay of 1.6042113304138184 us
1
[10]: speed_up0 = delay0 / delay1
print('with elementwise product broadcasting, speed up is ', speed_up0)
with elementwise product broadcasting, speed up is 48.43488790304003
```

0.1 Test with cupy

```
[11]: import cupy as cp # import cupy
inforce = cp.random.uniform(0, 10000, [190000, 10])
fund_return_ratio = cp.random.uniform(0.8, 1.2, [num_proj_points, num_funds])
[14]: s = time.time()
for t in range(num_proj_points): # loop over time
ret = fund_return_ratio[t] * inforce
# make sure all the gpu computation kernels have finished the processing
cp.cuda.Stream.null.synchronize()
e = time.time()
delay2 = (e - s) / num_inforces * 1e6
print('Delay of ', delay2, 'us')
Delay of 0.2076733112335205 us
[13]: speed_up1 = delay0 / delay2
```

```
print('with elementwise product broadcasting, speed up is ', speed_up1)
with elementwise product broadcasting, speed up is 448.4177456416748
[23]: print('Nmber of total inforce', num_inforces)
print('delay 10x1 and 10x1 elementwise product per inforce: ', delay0, ␣
↪'microseconds')
print('delay 10x1 and {}x1 elementwise product per inforce: '.
↪format(num_inforces), delay1, 'microseconds')
print('delay 10x1 and {}x1 elementwise product per inforce with cupy: '.
↪format(num_inforces), delay2, 'microseconds')
Nmber of total inforce 200000
delay 10x1 and 10x1 elementwise product per inforce: 77.69979596138
microseconds
delay 10x1 and 200000x1 elementwise product per inforce: 1.6042113304138184
microseconds
delay 10x1 and 200000x1 elementwise product per inforce with cupy:
0.2076733112335205 microseconds
2
```

About The Society of Actuaries

With roots dating back to 1889, the Society of Actuaries (SOA) is the world's largest actuarial professional organization with more than 31,000 members. Through research and education, the SOA's mission is to advance actuarial knowledge and to enhance the ability of actuaries to provide expert advice and relevant solutions for financial, business and societal challenges. The SOA's vision is for actuaries to be the leading professionals in the measurement and management of risk.

The SOA supports actuaries and advances knowledge through research and education. As part of its work, the SOA seeks to inform public policy development and public understanding through research. The SOA aspires to be a trusted source of objective, data-driven research and analysis with an actuarial perspective for its members, industry, policymakers and the public. This distinct perspective comes from the SOA as an association of actuaries, who have a rigorous formal education and direct experience as practitioners as they perform applied research. The SOA also welcomes the opportunity to partner with other organizations in our work where appropriate.

The SOA has a history of working with public policymakers and regulators in developing historical experience studies and projection techniques as well as individual reports on health care, retirement and other topics. The SOA's research is intended to aid the work of policymakers and regulators and follow certain core principles:

Objectivity: The SOA's research informs and provides analysis that can be relied upon by other individuals or organizations involved in public policy discussions. The SOA does not take advocacy positions or lobby specific policy proposals.

Quality: The SOA aspires to the highest ethical and quality standards in all of its research and analysis. Our research process is overseen by experienced actuaries and nonactuaries from a range of industry sectors and organizations. A rigorous peer-review process ensures the quality and integrity of our work.

Relevance: The SOA provides timely research on public policy issues. Our research advances actuarial knowledge while providing critical insights on key policy issues, and thereby provides value to stakeholders and decision makers.

Quantification: The SOA leverages the diverse skill sets of actuaries to provide research and findings that are driven by the best available data and methods. Actuaries use detailed modeling to analyze financial risk and provide distinct insight and quantification. Further, actuarial standards require transparency and the disclosure of the assumptions and analytic approach underlying the work.

Society of Actuaries
475 N. Martingale Road, Suite 600
Schaumburg, Illinois 60173
www.SOA.org