Capstone FMS 2025

Team D

Yuankai Tao, Chen He, Xile Wang, Haoyan Li, Yuhang Li

December, 2025

Code Repository: All weekly updates, experimental notes, and performance summaries of this project can be found in the GitHub repository:

`https://github.com/YuankaiTao-super/Capstone_FMS_2025`

Abstract

This project improves the performance of yield-to-worst (YTW) computation in the MuniBond library and analyzes which factors affect the runtime. First, we address numerical inconsistencies and redundant computation in the original scripts. By refining the initial yield guess, stabilizing the Newton solver, and applying Numba-based optimization, we achieve more than a threefold improvement in runtime and over a 200× acceleration in the solver component. In the second part of the study, we examine the determinants of computation time. The results show that, after optimization, initialization and Newton iterations contribute minimally to runtime variance. Instead, cashflow-generation cost scales approximately linearly with the number of remaining coupon payments. These findings clarify the computational structure of YTW evaluation and provide a practical basis for predicting runtime for MuniBond library.

[0]We thank Dr. Wildman of FMS Company for his weekly guidance and insightful feedback throughout the development of this project. We are also grateful to Prof. Melvin and Prof. Girand for their direction and support, particularly in helping us formulate the research question on the determinants of computation time. Any remaining errors are our own.

1 Introduction

1.1 Background

The U.S. municipal bond market, valued at approximately $3.9 trillion, is a large and complex part of the fixed-income universe. With roughly 1 million active securities, it is much broader than the markets for U.S. government and corporate bonds. The market is now undergoing a shift toward electronic trading, so the speed of computation has become increasingly important. A core task for fixed-income dealers is the price-to-yield calculation, which determines the internal rate of return (Yield to Maturity, or YTM) for a bond based on its market price, along with several related calculations.

1.2 Overall Problems

Currently, the analytics library measuring risks related to municipal bonds written in Python, the `munibond.py` library[1], takes too long to complete the computation of yield-to-worst [2] for a single municipal bond. In a market where a large number of securities must be evaluated in real time, this delay remains a clear bottleneck. The main questions for this project are as follows:

- Are there issues in the current library that can be identified and resolved?
- How can the existing scripts be modified and optimized so that the computation time can be reduced?

1.3 Importance of Project

As the market continues to move toward electronic trading, speed has become closely linked to profitability. Slow computations can lead to missed trading opportunities and other business costs. Although this can usually not be applied to high-frequency trading, it is still important as it can save the trader valuable time. By reducing this latency, the project will strengthen the sponsor's competitive position and support a more timely response to market changes.

1.4 Empirical Summary

We focused on identifying and addressing the main performance bottlenecks in the ytw (yield-to-worst) computation process. We collected detailed timing information for the functions in the scripts so that we could determine where the computation time was concentrated. Through this profiling work, several sources of slowdown were identified and tested with different optimization methods. We conducted a series of experiments. First, we introduced a new argument, `initial_guess`, in the `scipy.newton_solver` so that the number of iterations could be reduced. Second, we removed unnecessary

[1] Originally developed by Chad Wildman for municipal bond pricing and risk analytics. Hereafter, we refer to this module as the MuniBond. A full variable dictionary is provided in Appendix D.

[2] Hereafter, we refer to this as YTW

cash-flow generation in the root-solving function, which resulted from an inefficient loop structure. Third, we applied Numba to two hotspots in the municipal bond pricing pipeline. These included the core numerical calculation extracted from the original pricing routine and the Newton solver used to compute yields from prices. A small helper function for discount factors was also accelerated. For future work, batch-level parallelism may be considered. Parallel computing remains difficult for this type of task because Python has operating system–level constraints such as the Global Interpreter Lock and other locking mechanisms. Even so, it may still provide gains if these limitations can be addressed.

2 Core Formulas

We note that in this section we present only a simplified version of the formulas, which is meant to be more conceptual. The realistic calculation can be more complex because it involves a larger number of branches and loops.

2.1 Bond Pricing

Case 1: Short-term bond ($N \leq 1$)

$$P = 100 \left(\frac{\frac{RV}{100} + \frac{R}{M}}{1 + \frac{E-A}{E} \cdot \frac{y}{M}} - \frac{R\,A}{B} \right)$$

Case 2: Coupon-paying bond ($N \geq 1$)

$$P = \frac{RV}{\left(1 + \frac{y}{M}\right)^{N-1+\frac{E-A}{E}}} + \frac{100R}{M} \sum_{k=1}^{N} \frac{1}{\left(1 + \frac{y}{M}\right)^{k-1+\frac{E-A}{E}}} - \frac{100RA}{B} \tag{1}$$

Table 1: Variables Definition

Symbol	Description
P	Bond price
y	Nominal annual yield
R	Annual coupon rate
M	Num of coupon payments per year
RV	Redemption value
N	Remaining num of full coupon periods
E	Num of days in the current coupon period
A	Num of days accrued since last coupon payment
B	Day count convention base (360)

2.2 Newton Solver

We wish to find a root of a differentiable function $f(x)$, i.e., a value $x^\star$ such that $f(x^\star) = 0$. Starting from an initial guess x_0, Newton's method updates the iterate by

$$x_{k+1} \;=\; x_k \;-\; \frac{f(x_k)}{f'(x_k)} \tag{2}$$

where $f'(x_k)$ denotes the derivative of f at x_k. The iteration proceeds until a stopping criterion is met (e.g., $|f(x_k)|$ is below a tolerance or a maximum number of steps is reached).

3 Other Calculations(Rounding Conventions)

These formulas mainly explained how the counts of some variables are rounded.

3.1 MSRB Day Count Convention

The MSRB 30/360 day count convention is fundamental to municipal bond calculations

$$\text{Days} = (Y_2 - Y_1) \times 360 + (M_2 - M_1) \times 30 + (D_2 - D_1) \tag{3}$$

with adjustments:

- If $D_1 = 31$, then $D_1 = 30$
- If $D_2 = 31$ and ($D_1 = 30$ or $D_1 = 31$), then $D_2 = 30$

[Insert code block from Appendix C.1 here]

3.2 MSRB Price Rounding

Prices are rounded:

$$\text{Rounded Price} = \frac{\lfloor \text{Price} \times 1000 \rfloor}{1000} \tag{4}$$

[Insert code block from Appendix C.1 here]

3.3 MSRB Yield Rounding

Yields are rounded to the nearest 0.001%:

$$\text{Rounded Yield} = \begin{cases} \frac{\lfloor y \times 10000 \rfloor + 1}{1000} & \text{if last digit} \geq 5 \\ \frac{\lfloor y \times 10000 \rfloor}{1000} & \text{otherwise} \end{cases} \tag{5}$$

[Insert code block from Appendix C.1 here]

4 Dataset and Preprocessing

Introduction of Data Source: We used the dataset provided by Chad, which contains recent real trading data. From this dataset, we randomly extracted a sample of 10,000 CUSIPs so that we could evaluate the script performance under realistic conditions.

4.1 Dataset

There are mainly two kind of datasets:

- **Main Dataset:** muniSecMaster.csv, including: cusip, Dates (e.g., maturityDate, nextCouponDate, RefundDate, etc.), Prices (e.g., nextCallPrice, parCallPrice, etc.). This dataset contains a large amount of variables that are used in the main library constructed, if you want to gain a better understanding about some of the detailed code blocks, we strongly recommend searching the unclear variables from the appendix: Municipal Bond Computation Library Variables Codebook (Appendix D), in which we explained all pricing and risk variables included in MuniBond in detail.
- **Black List Datasets:** issuerBlackList, obligorBlackList, stateBlackList

4.2 Data Preprocessing

We carried out several preprocessing steps:

- Randomly extracted a performance test sample of 10,000 CUSIPs.
- Excluded CUSIPs that appeared in any of the blacklist datasets.
- Converted variables into appropriate data types, primarily primitive numeric types, so that they are compatible with Numba and support efficient computation.

5 Key Issues Identified and Resolved

5.1 Issue 1: Lack of Detailed Timer

Originally, the script provided only total runtime measurements, which made it difficult to determine which parts of the code consumed the most resources. Without detailed timers, we could not quantify how much time was spent in cashflow generation, yield solving, or intermediate data preparation.

To address this, we first used third-party profilers like `scalene` and `cProfile` to identify hotpots. The profiling results confirmed that functions `generate_cashflows` and the bond pricing solver accounted for the majority of the total runtime. Based

on this finding, high-resolution timers (`perf_counter_ns`) were integrated into the key computation paths within MuniBond. This allows each component (cashflow generation, solver iteration, and overall yield calculation) to be measured separately.

Table 2: Timing comparison across computation Parts

Metric	**generate cash flow**	**newton solver**	**stabilized ytw calc**
N	9778	9778	9778
Avg time (ms)	1.17	1.47	2.15
Max time (ms)	40.84	45.12	45.81
Min time (ms)	0.00	0.05	0.67

The results in the Table 2 clearly show that both the cashflow generation and the Newton solver dominate the runtime. Together they account for nearly the entire computational cost of the YTW evaluation. This finding provided the foundation for the next stage of targeted optimization.

5.2 Issue 2: Lack of Automated Monitoring

Each time the script was modified, there was no workflow that automatically verified its runtime and correctness after the change of scripts.

Therefore, we setup a continuous integration (CI) workflow on GitHub Actions to automatically test both accuracy and runtime performance each time the main code is updated.

5.3 Issue 3: Outlier Behavior in YTW Calculation

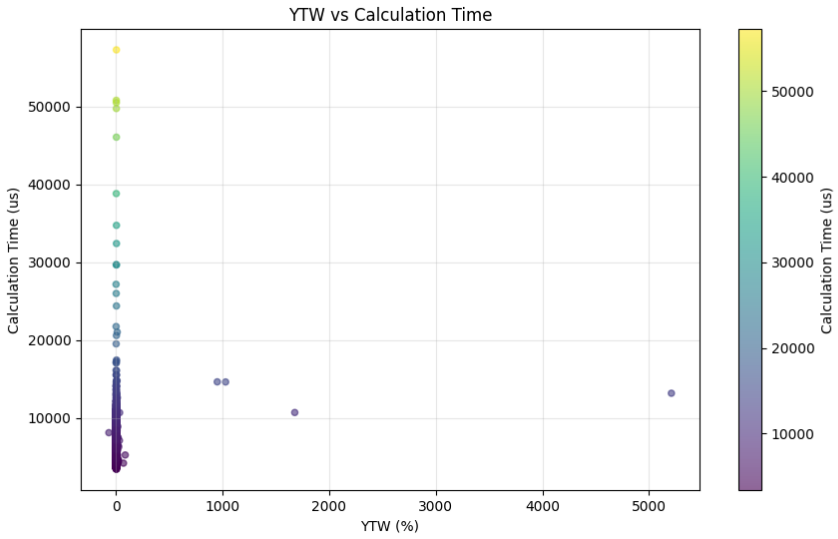


Figure 1: Outlier Behavior

We plotted the YTW solved versus the total calculation time at first and found several outliers. The outliers observed in Figure 1 arise from numerical instability in the YTW solver when the initial yield guess is poorly aligned with the true yield. For bonds trading far from par value, the coupon rate can deviate from the actual yield a lot, leading the Newton iteration to start from an inaccurate initial guess. Originally, if the initial guesses were not set for the solver, the default branches would use $coupon\_rate + 0.0001$ as the initial guess if market price of the bond overtake 100 or otherwise use that $rate - 0.0001$ as the guess:

[Insert code block from Appendix C here]

In these cases, the price-yield equation may have more than one solution, or the derivative may be very flat, leading to slow convergence or divergence of the solver, thus, the solver cannot converge to the actual yield rate within the max iterations (set as 100). These numerical irregularities and instabilities explain the why there are a small number of abnormal data point with extremely large yield.

We solved this issue by modifying the core root-solving function, which we will explain in Experiment 4 6.2.1.

5.4 Issue 4: Redundant Cashflow Generation for Multi-Workout Bonds

A more severe inefficiency was discovered in bonds with multiple workout scenarios (for example, a bond with three possible workouts). In such cases, the `ytw(price)` function calls `yieldWorkout` once per workout. However, during the Newton iteration inside each `yieldWorkout`, the solver repeatedly calls `bond_price_periodic`, which in turn calls `generate_cashflows`. This results in the same cashflow being regenerated many times unnecessarily.

For instance, with 3 workouts and 10 Newton iterations, the function `generate_cashflows` was executed 30 times, even though only 3 unique cashflow sets were actually needed. This redundant computation wasted significant CPU time and became a key performance bottleneck.

To eliminate this issue, a caching mechanism (`cash_flow_cache{}`) was implemented during CUSIP initialization. Before generating new cashflows, the system checks whether the requested combination of settlement date and workout date already exists in the cache. If found, the previously computed cashflows are immediately restored instead of being recalculated. Code excerpt from the revised function:

[Insert Code Block C here]

After introducing this caching system, `generate_cashflows()` is now executed once per workout, not once per iteration. The improvement nearly halved total runtime and drastically reduced the time spent on cashflow generation.

Overall, as shown in Table 3 total computation time improved from 2.15 ms to 1.10 ms (a 1.95× speedup), and the cashflow generation phase alone achieved nearly 9× faster

Table 3: Performance comparison before and after caching

Metric	Before	After	Speedup
Total avg calc time (ms)	2.15	1.10	$\times$ 1.95
Max total time (ms)	45.81	7.40	$\times$ 6.2
Min total time (ms)	0.67	0.66	$\approx$
avg gen_cf time (ms)	1.17	0.13	$\times$ 9

performance at the first stage. The change also made performance far more predictable across different bonds.

6 Experiments Conducted

We had conducted four main experiments in this project with the first three experiments trying to make improvement on the calculation speed and the last one mainly focusing on the correlation between the bond characteristics and the calculation time of ytw.

6.1 Experiment 1: Introduction of Better Initial Guesses

We introduced an `initial_guess` parameter to the `ytw` and `yieldWorkout` functions, allowing the solver to start from a more informed yield estimate (its historical yield from a .psv file). The purpose was to reduce the number of Newton iterations needed for convergence.

When tested on the sample, the optimized initialization provided only a small improvement—about 2–3% faster on average. While this helps slightly, it does not significantly change the overall performance profile. Therefore, this optimization was documented but not prioritized in later stages.

6.2 Experiment 2: Numba Acceleration

We introduced Numba, a just-in-time (JIT) compiler for Python that translates numerical Python code into fast, machine-level instructions. Unlike the standard Python interpreter, which executes code line by line, Numba compiles selected functions into native code the first time they are run, then reuses that optimized version on subsequent calls. This eliminates much of Python's overhead, especially in numerical loops.

We rewrote the core computation parts for the bond pricing and Newton solver's numerical kernel using Numba's `@njit` decorator in "nopython" mode.

6.2.1 Numba-decorated Changes

Below are the detailed code in which we placed under Numba control. We transferred the objects of main parameters into numeric types, no Python objects in the hot loop,

decorated the function with `@njit`. The function then contained only numeric operations and simple loops so it compiled well in Numba's nopython mode.

Numba-optimized function of periodic bond pricing

[Insert code block from Appendix C here]

Numba-optimized Newton solver

We overwrite the previous Python Newton solver with Numba that can call the numeric core above directly, making the whole calculation with solver more efficiently.

[Insert code block from Appendix C here]

6.2.2 Observed anomaly and Explanation

Problems we found

During experiments a single CUSIP repeatedly produced a very large computation time relative to the rest of the sample. As Table 4 showed, this high-latency outlier leaded to an abnormal maximum computation time and further skewed distribution statistics and made the overall speedup look inconsistent.

Table 4: Performance Comparison Before and After Numbalization

Metric	Last version	After Numba	Speedup
Total avg calc time (ms)	1.10	0.79	40%× faster
Max avg calc time (ms)	7.40	361.51	much slower
Min avg calc time (ms)	0.66	0.61	≈

We investigated and found the cause: Numba's first-call compilation penalty.

Explanation

- When a Numba-decorated function is invoked for the first time with a new signature, Numba compiles machine code — this can take hundreds of ms for non-trivial functions.
- If the timing measurement includes this first-call compilation time, the sample distribution will show an extreme outlier (a single call taking much longer), which misrepresents steady-state per-call cost.

Remedy (warm-up section added): We added a warm-up step that triggers compilation before we start latency-sensitive measurement or production use. **Warm-up function**

[Insert code block from Appendix C here]

This function constructs a representative bond instance and calls the path that will execute the Numba-compiled functions, ensuring compilation occurs once during controlled startup rather than in a live measurement or low-latency flow.

6.2.3 Performance results

As we had already conducted several experiments on the initial sample, the system had already retain some implicit familiarity with the original sample set, which meaned that the operating system somehow had "memorized" the dataset, thus, we selected a new random sample of 10,000 bonds to evaluate the performance after optimization. This procedure ensured potential memory effects eliminated and prevented bias in assessment of performance. To simulate real trading conditions, we recorded only the timing measurements from the first execution run.

Below we report corrected speedups after selecting a new sample and excluding the first-call compilation. The bond core numbers (i.e. computation/calculation time) come from our recorded measurements. Comparison between the distributions of renewed time data displayed here:

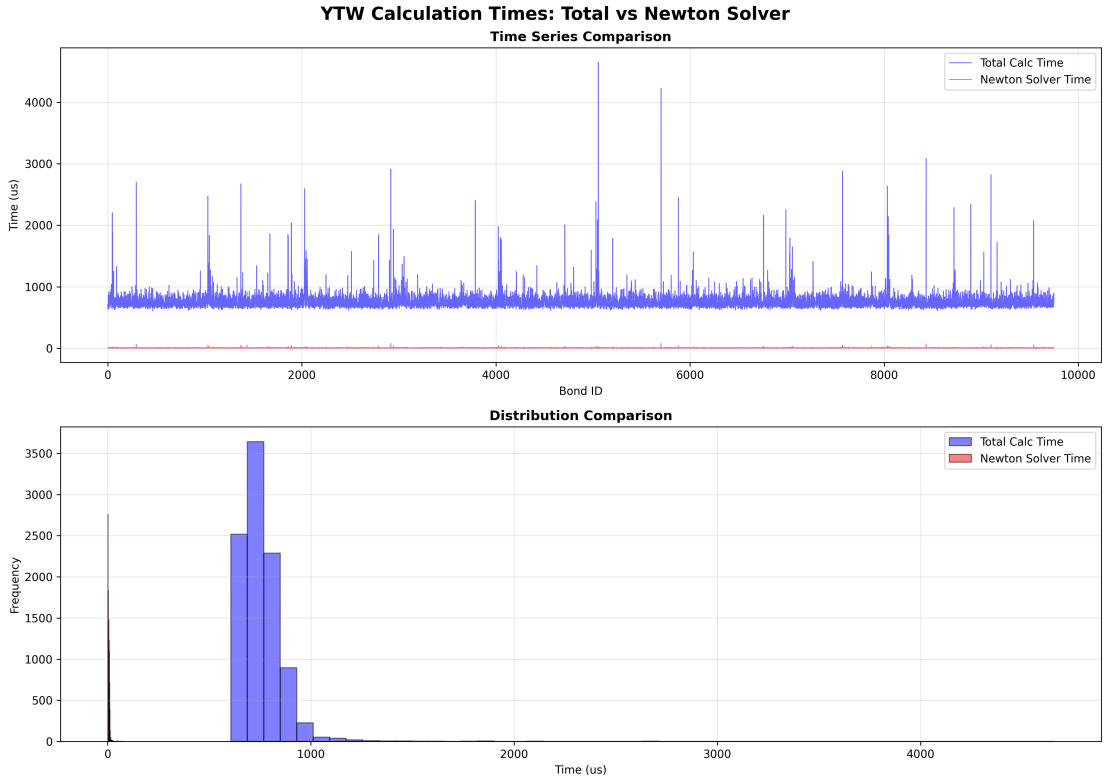


Figure 2: Distribution of time consumed

Performance of Newton solver

We moved the Newton iterations into Numba, and Table 5 showed the new result with time overhead for newton solver

The optimized solver demonstrates a dramatic improvement with Numba, reducing average solve time from 1.5678 ms to 0.0075 ms, which is 200× faster

Table 5: Newton Solver Performance Comparison (1st Run)

Metric	Optimized	Baseline	Notes
Mean (ms)	0.0075	1.5678	200× faster
Median (ms)	0.0060	1.1764	–
Max (ms)	0.1221	15.1850	–
Min (ms)	0.0007	0.0518	–

Whole calculation

Table 6: Bond Pricing Performance Comparison (1st Run)

Metric	Optimized	Baseline	Notes
Bonds processed	9,872	9,854	–
Mean calc. time (ms)	0.7551	2.3666	3.13× faster
Median calc. time (ms)	0.7784	1.9616	–
Max calc. time (ms)	2.9774	14.8902	–
Min calc. time (ms)	0.5929	0.6864	–

After we excluded the time overhead of first-call compilation by applying the additional warm-up section, the average time has been reduced from 2.366 ms to 0.855 ms, indicated in Table 6 demonstrating Numba's effectiveness in real trading world on the whole sample computation.

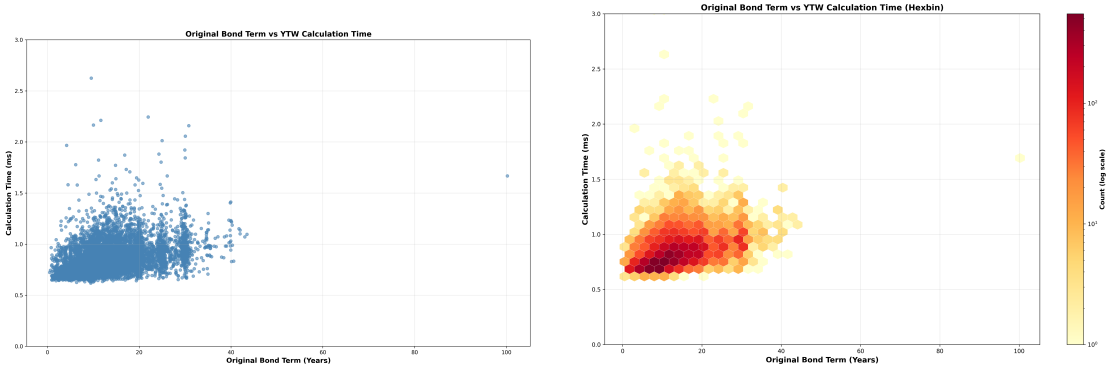
6.3 Experiment 3: Determinants of YTW Calculation Time

6.3.1 Exploratory Data Analysis (EDA)

This analysis was first motivated by a suggestion to examine whether bond term affects computation time. We used scatter plot, shown in Figure 3ato reveal if a bond's term has virtually effect for YTW calculation time. Additionally, because individual observations are densely concentrated, the raw scatter plot suffers from substantial overplotting. Therefore, we plotted Figure 3b to shows a density of count of bond terms. Both of the two plots shows that there is no virtually effect as the data points are almost randomly distributed.

We then transferred our attention into the remaining number of coupon payments as each additional coupon might require a cash-flow evaluation and root-solution in MuniBond, making it potentially a primary driver of computational time overhead. To understand the structure of our dataset, we began by examining the distribution of coupon count.

Figure 4 presents the distribution of coupon count across the sample. The majority of bonds fall within the 1–40 range, with very few observations having higher coupon count. This is consistent with the pattern of U.S. municipal bonds. Since most municipal bonds have maturities less than 30 years and pay coupons semiannually, the majority of



(a) Scatter Plot (b) Hexbin Plot

Figure 3: Bond Term vs Calculation Time

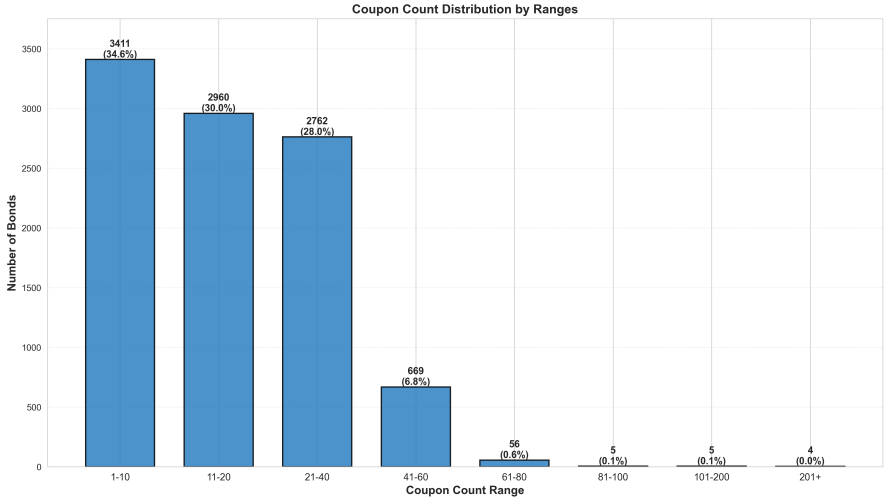


Figure 4: Distribution of Coupon Payments

bonds usually fall within the 1–40 coupon range, while long-duration bonds are relatively rare.

6.3.2 Empirical Specification

To quantify the determinants of YTW computation time, we estimate the following linear model:

$$\text{CalcTime}_i = \beta_0 + \beta_1 \,\text{CouponCounts}_i + \beta_2 \,\text{InterestFreq}_i + \beta_3 \,\text{PriceDeviation}_i + \varepsilon_i, \quad (6)$$

6.3.3 Variables Definition

where CalcTime_i denotes the computation time (in ms) for a single YTW call on bond i, and ε_i is an idiosyncratic error term. We defined the meaning of variables in Table 7 and the reasons why we choose to take these variables into consideration.

Table 7: Variable Definitions for Analysis

Variable	Definition and Rationale
CouponCounts$_i$	Remaining number of coupon payments (N). This is the dominant determinant of computation time because each additional coupon requires a cash-flow evaluation. The YTW routine scales almost linearly with N, making it a primary structural driver of computational load.
InterestFreq$_i$	Coupon payment frequency (Monthly, Quarterly, Semiannual, Annual). More frequent schedules require evaluating more coupon dates, increasing the cash-flow discounting operations and Newton-iteration complexity.
PriceDeviation$_i$	Absolute deviation of the bond's clean price from par. Bonds farther from par require more Newton iterations to converge due to greater nonlinearity, raising computational burden even with identical cash-flow structures.

Table 8: Regression Analysis

	Model 1 (Baseline)	**Model 2** (w/ Controls)	**Model 6** (Gen CF)
Dependent Variable	calc_time	calc_time	gen_cashflows
Coupon Count	0.005697***	0.005950***	0.005639***
	(0.0000926)	(0.0000968)	(0.0000344)
Interest Frequency		0.0957***	-0.031635***
		(0.0051)	(0.001894)
Price Deviation		0.0004***	0.000184***
		(0.0002)	(0.000085)
Constant	0.7401***	0.6923***	0.0808***
	(0.0081)	(0.0114)	(0.0037)
R-squared	0.2774	0.2816	0.7236
Adjusted R^2	0.2772	0.2813	0.7235
F-statistic	3,789.6***	1,278.3***	8610.0***
Observations	9,872	9,872	9,872

Notes: Standard errors in parentheses.

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$.

Additional specifications (Models 3, 4 and 5) are reported in Appendix Table 10.

6.3.4 Main Results

Table 8 reports the baseline regression analysis on the determinants of YTW computation time. Additional specifications (Models 3, 4, and 6) are provided in Appendix Table 10.

According to result of baseline models in the first two columns, the number of remaining coupon payments is the primary driver of runtime. The estimated coefficients range from 0.00542 to 0.00595, implying that each additional coupon increases computation

time by approximately 5 to 6 μs. In contrast, the effects of *Interest Frequency* and *Price Deviation* are negligible. A one-unit increase in interest frequency raises computation time by only 0.096 ms, and a one-unit increase in price deviation changes computation time by merely 0.0004 ms.Overall, the analysis shows that YTW runtime is driven almost exclusively by the number of coupon count, whereas other bond characteristics exert only minimal influence.

Although coupon count explained only about 29% of the variation in total computation time in Models 1 and 2, we further examined how the runtime of each component of YTW computation changes as coupon count increases.

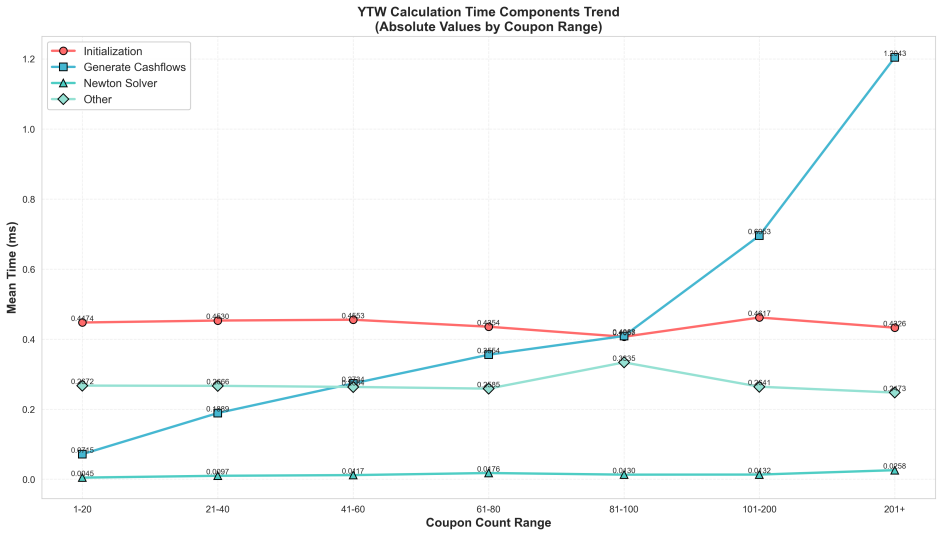


Figure 5: Breakdown Trends

The Figure 5 shows that only the cashflow generation stage varies significantly with coupon count, while initialization, Newton solving, and other components remain nearly constant. We had also plotted a stacked composition plot in Appendix B, These confirms that the computational costs grows almost entirely through the expansion of the cashflow schedule. (We also plotted the distribution of cashflow-generation time, and the corresponding figure is provided in the appendix B for reference.)

Thus, in our renewed specification 7, we changed the dependent variable from total computation time into cash flow generation time and conducted the regression based on the new specification:

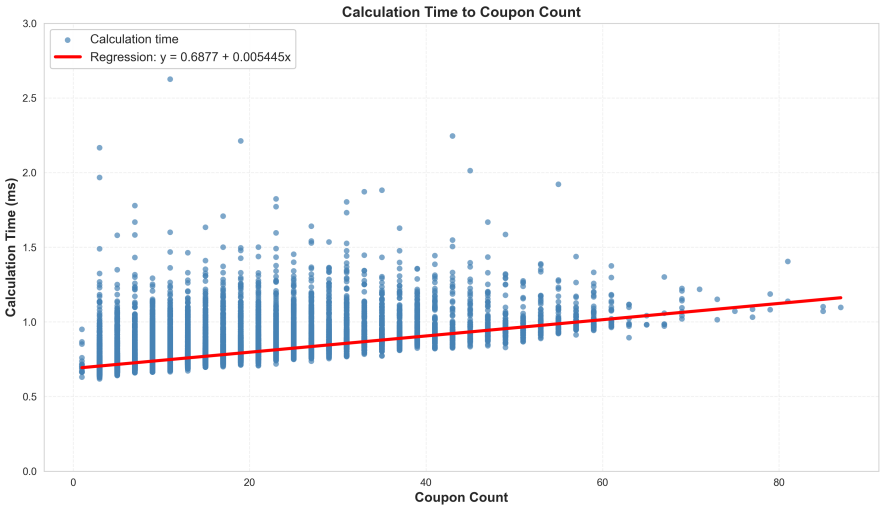
$$\text{CFGeneration}_i = \beta_0 + \beta_1\,\text{CouponCounts}_i + \beta_2\,\text{InterestFreq}_i + \beta_3\,\text{PriceDeviation}_i + \varepsilon_i, \quad (7)$$

According to the result of Model 6 from Table 8, we found that when runtime for cash flows generation is treated as the dependent variable, coupon count explains 71.52% of variance, which is a dramatic improvement from 27.74%. This can be a dominant variable cost in YTW calculation, especially when the coupon count is large for a bond. Thus, it further convinced us that, as the time overhead for newton solver and initialization are almost constant, cash flows generation can be very large when the

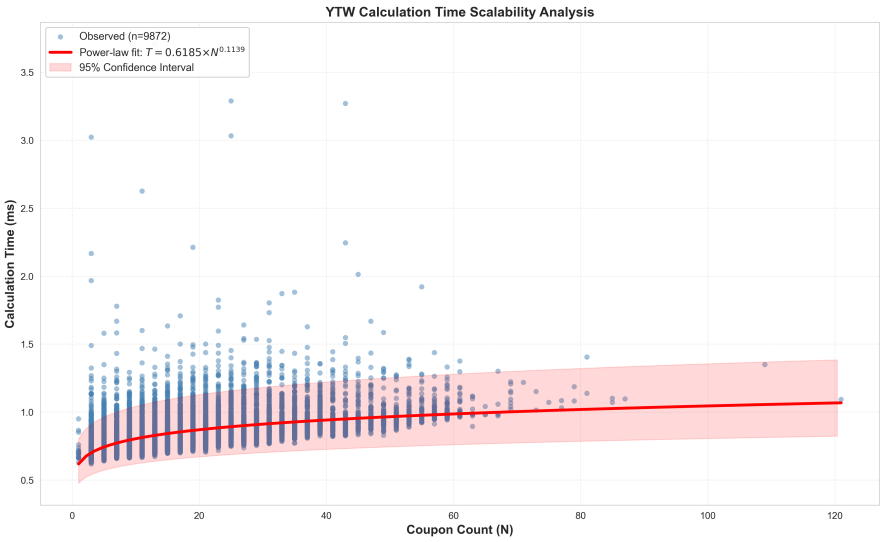
coupon count increases.

6.3.5 Visual Evidence

For the baseline model, we visualized the result with a new scatter plot showing the trend between the coupon count and the total calculation times, while it showed noticeable scatter, particularly among bonds with low coupon count, suggesting some unexplained variance with the baseline model (Model 1).



(a) Coupon Count vs Calculation Time



(b) Scalability Analysis

Figure 6: Linear and Scalability Analysis

We had also conducted the scalability analysis of total YTW calculation time in Figure 6b, which revealed a sub-linear power-law relationship ($T = 0.5875 \times N^{0.1076}$, $R^2 = 0.3651$) with exponent $b = 0.1076$. This sub-linear pattern, significantly lower than the $O(N)$, reflects the dominance of some fixed overheads in the overall computation.

Therefore, we removed the initialization time from the total calculation time, and

we ran the regression of coupon count and the residual time (total calculation time - initialization time), and it turned out that the data points cluster much more tightly around the regression line.

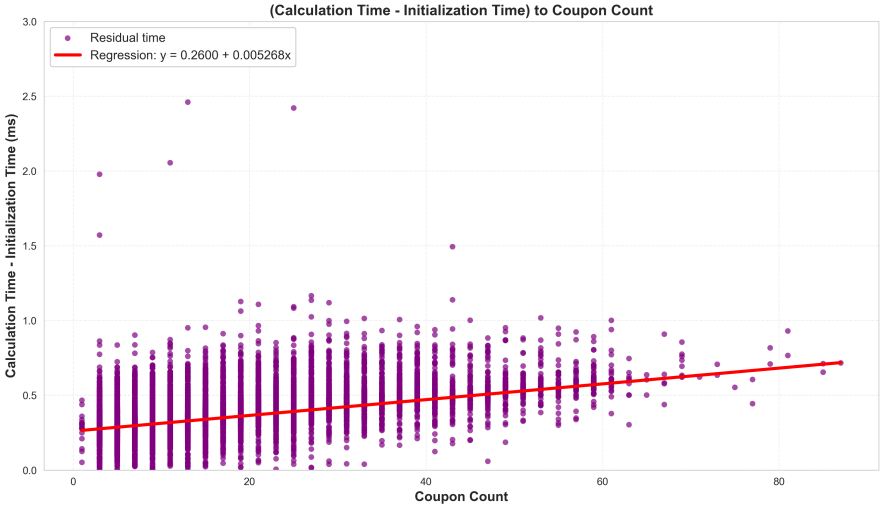


Figure 7: Coupon Count vs Residual Time

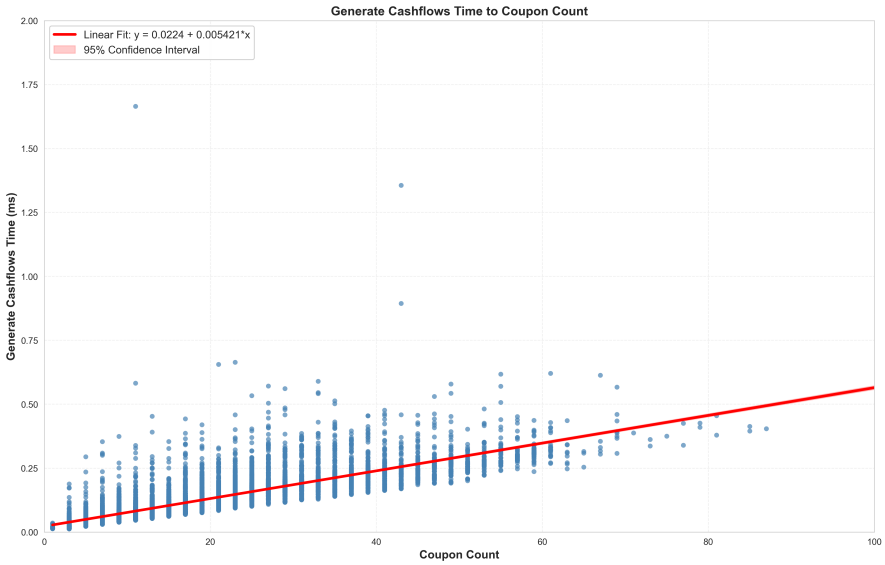
This improvement indicates that initialization overhead, driven by some stochastic system-level processes such as memory allocation and caching system, is the main reason of scatter in the previous plot.

For the valuable finding that cash flows generation time varies with the coupon count, we visualized the pattern with scatter plot in Figure 8a.

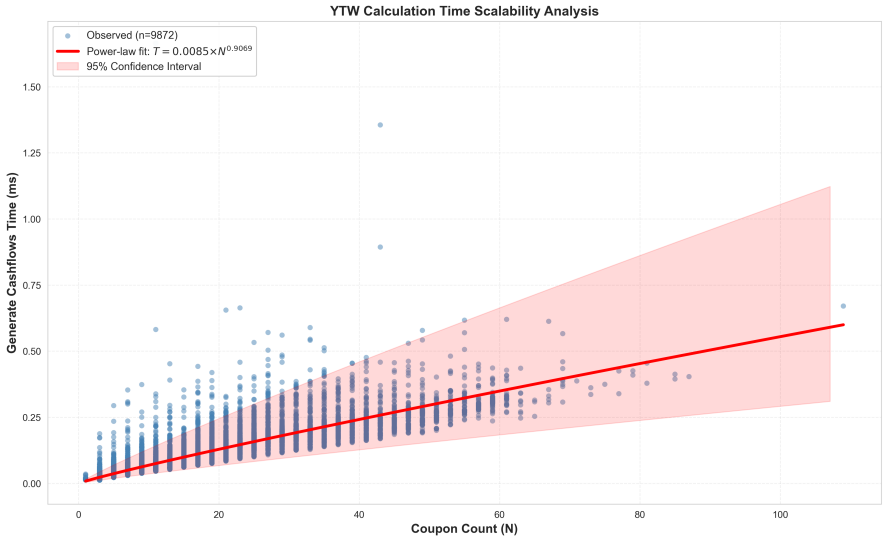
In contrast, the isolated generate_cashflows component shows a near-linear scaling ($T = 0.0085 \cdot N^{0.9069}$, $b \approx 0.91$). The exponent ($b \approx 0.907$) confirms $O(N)$ computational complexity, with the deviation from unity attributable to fixed-cost initialization overhead. Shaded confidence regions (pink, 95% CI) quantify the prediction uncertainty. It further confirms that the weakness of the total-time regression is due to the multi-component structure of YTW calculation

Additionally, because individual observations are densely concentrated, the raw scatter plot suffers from substantial over-plotting. Therefore, Figure 9 shows a hexbin density plot of generate_cashflows computation time versus estimated coupon count, which more clearly reveals the hidden pattern with the regression results. In this figure, the color intensity represents observation density on a logarithmic scale. The blue dashed line shows the fitted linear relationship. The clear pattern confirms the strong linear dependence of cashflow generation time on coupon count, with most observations clustering tightly around the regression line.

Figure 12 in Appendix B provides a hexbin-based visualization comparing how different components (cf generation, newton and initialization parts) of the YTW calculation scale with coupon count. The figure illustrates the density structure of the data.



(a) Coupon Count vs CFs Generation Time



(b) Scalability Analysis

Figure 8: Coupon Count vs CF Generation Time

6.4 Experiment 4: Parallel Computing

To further reduce end-to-end time, we combined the Numba improvements with parallel computing. Batch-level parallelism: partition the 10,000 CUSIP list into worker chunks and run workers in parallel processes (multiprocessing). Because Numba-compiled functions produce native machine code per-process, using multiple processes avoids Python GIL(Global Interpreter Lock) limits and scales across CPU cores. Vectorized / SIMD options: evaluate whether some inner loops can be expressed with NumPy or Numba-parallel constructs (e.g., `prange`) for additional gains; careful attention to numerical determinism is required.

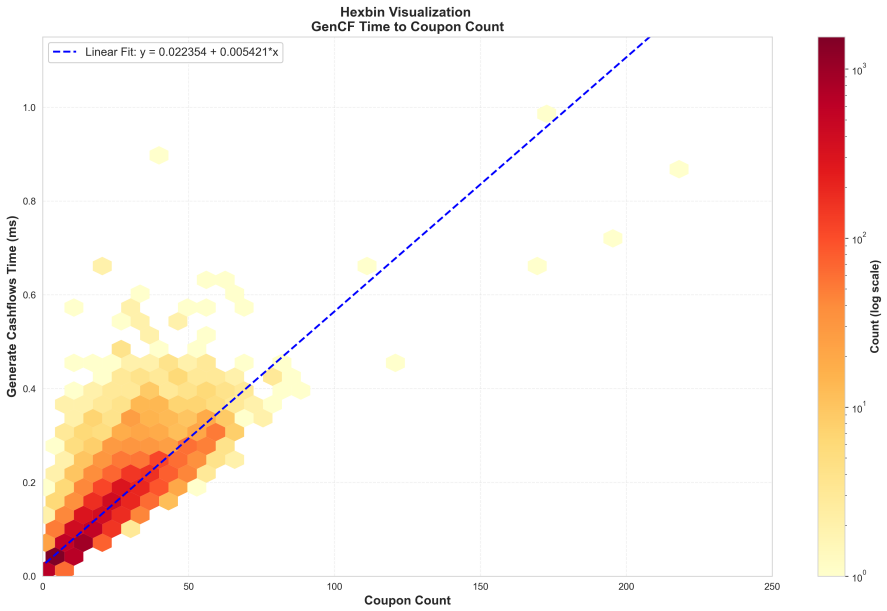


Figure 9: Hexbin: Coupon Count vs CF Generation Time

6.4.1 Pseudocode

Here, we attached the pseudocode to demonstrate how the parallel computing works in the real situation, and we also inserted our detailed code which you can find in Appendix C.

[Insert code block from Appendix C here]

Algorithm 1: Pseudocode for Parallelization

Input : List of CUSIPs
Output: Computed bond yields for all CUSIPs

Worker(CUSIP_List) Warm up Numba in the subprocess;
Initialize empty list *Results*;
foreach *CUSIP c in CUSIP_List* **do**
  Create bond object $b \leftarrow CreateBond(c)$;
  Compute $y \leftarrow b.YieldToWorst(Price)$;
  Append y to *Results*;
end
return *Results*;

Split total CUSIP list into N chunks;
Create a process pool with $N = 8$ workers;
foreach *chunk in split CUSIP list* **do**
  Assign **Worker**(chunk) to one process;
end
Collect and merge all results;

6.4.2 Performance Result

Intuitively, we expected that computational work across multiple processors should accelerate the whole process, however, our result of analysis in Table 9 shows that multi-process parallelization of YTW calculation demonstrates almost 3× slower performance compared to single-threaded computation.

Table 9: Single Thread vs Parallel Computing

Metric	Optimized	Parallel Run	Notes
Bonds processed	9,872	9,872	–
Mean calc. time (ms)	0.7551	1.1752	1.57× slower
Median calc. time (ms)	0.7784	0.8129	1.04× slower
Max calc. time (ms)	2.9774	775.7800	261× slower
Min calc. time (ms)	0.5929	0.6387	1.08× slower
Total processing time (s)	14.88	41.74	2.81× slower

Python's multiprocessing module creates completely independent processes, each requiring: 1. Separate Python interpreter instance: 50 MB memory per process. 2. Complete module reloading: All imports re-executed in each process. 3. Numba JIT recompilation: Bond pricing kernels compiled independently in each process. 4. secMaster data (Main dataset) duplication: 9,872 bond reference records loaded 4 times. As a whole, this is the extra Process creation cost, adding unnecessary time overhead per bond.

So let's do a simple calculation to gain intuition for why the parallel version incurs higher runtime. Consider the following rough breakdown: Suppose the sequential version processes 10,000 bonds with an average computation time of 0.8 ms per bond:

$$\text{Sequential time} = 10{,}000 \times 0.8\,\text{ms} = 8{,}000\,\text{ms} \approx 8\,\text{s}.$$

In the parallel version with four worker processes, the pure computation time remains

$$\text{Parallel compute time} = \frac{10{,}000}{4} \times 0.8\,\text{ms} = 2{,}000\,\text{ms}.$$

We additionally incur (i) process start-up overhead of approximately $4 \times 80\,\text{ms} = 320\,\text{ms}$ and (ii) per-bond inter-process communication overhead of roughly 0.6 ms:

$$\text{IPC overhead} \approx 10{,}000 \times 0.6\,\text{ms} = 6{,}000\,\text{ms}.$$

Including modest operating-system scheduling and synchronization costs (about 4–5 s in practice) gives a total parallel time of

$$\text{Total parallel time} \approx 2{,}000 + 320 + 6{,}000 + 4{,}000 = 12{,}300\,\text{ms} \approx 12\text{–}13\,\text{s}.$$

The resulting slowdown factor is therefore

$$\frac{8{,}000}{12{,}300} \approx 0.65\times,$$

meaning that the parallel computation is about 1.55–1.60× slower, consistent with the empirical results in Table 9. This calculation indicates that the per-bond IPC and scheduling overhead can outweigh the computational benefits of parallelizing the task.

7 Conclusions

This project provides both methodological and empirical insights into improving and understanding the computational performance. We resolve sources of numerical error and repeated work in the original script, introduce a more reliable initial guess for the solver, and apply Numba optimization to accelerate critical routines. These modifications produce substantial performance gains, removing nearly all Newton-solver overhead and yielding more than a threefold speedup in overall YTW computation.

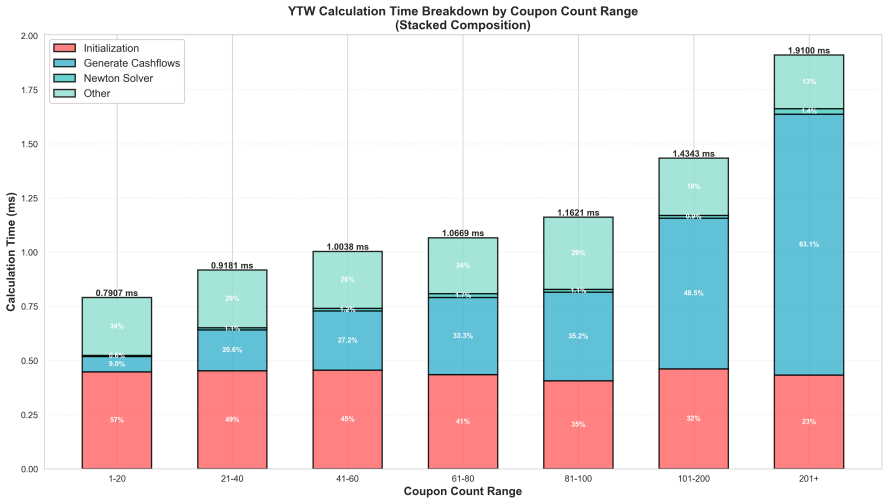
Beyond optimization, our analysis offers a clear characterization of the factors that determine runtime. We demonstrate that, once the solver is stabilized and optimized, initialization and Newton iterations contribute little to computation time. We find that computation time does not scale with bond term, as initially assumed, but instead with the number of remaining coupon payments. The near-linear scaling of the cashflow-generation component provides a practical basis for estimating computational cost and predicting performance under different bond structures. Additionally, this project also provided broader insights into performance engineering, which is a pefect training for all of our teammates. Through diagnosing numerical instability, profiling runtimes, and evaluating trade-offs between JIT optimization and multiprocessing overhead, we gained practical experience of system design, including workload decomposition, bottleneck identification, and the limits of parallelism under communication costs. These lessons are directly applicable to the development of scalable computational pipelines in fixed-income analytics and beyond.

A Full Regression Specifications

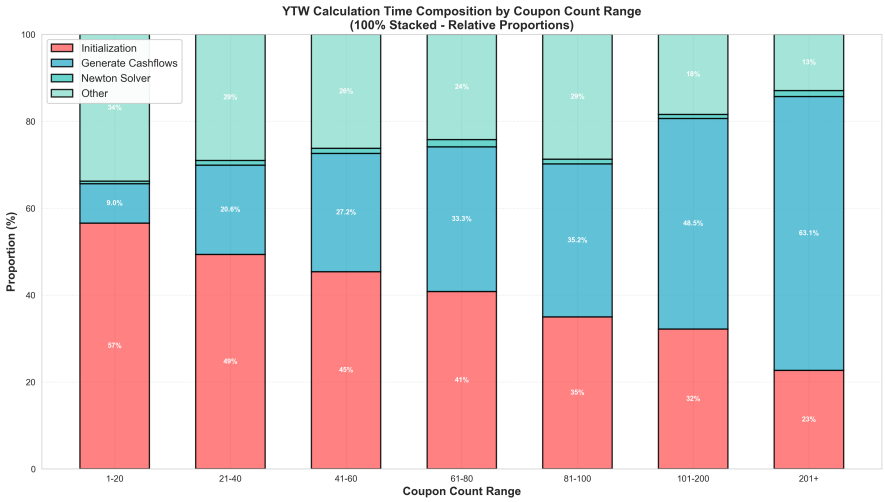
This appendix reports the full set of regression specifications of the regression analysis in Section 6.3.4.

B Additional Visualizations

B.1 Time Breakdown



(a) Stacked Time Breakdown



(b) Time Breakdown Decomposition

Figure 10: Time Breakdown Analysis

B.2 Distribution of Cash Flows Generation Time

The distribution of cashflow-generation time is highly concentrated near 0.5ms, with a sharp peak and a long right tail. Most of the bond do not require too much time to generate cashflows, though still a considerable proportion, and a small number of observations exhibit extremely high runtimes.

Table 10: Regression Analysis – Full Specifications

	Model 1 (Baseline)	**Model 2** (w/ Controls)	**Model 3** (Offset Init)	**Model 4** (Newton Only)	**Model 5** (Gen CF Only)	**Model 6** (Multivariate CF)
Dependent Variable	calc_time	calc_time	residual_time	newton_solver	gen_cashflows	gen_cashflows
Coupon Count	0.005697***	0.005950***	0.005607***	0.000194***	0.005421***	0.005639***
	(0.0000926)	(0.0000968)	(0.0000914)	(0.0000058)	(0.0000344)	(0.00000xx)
Interest Frequency		0.0957***				-0.001635***
		(0.0051)				(0.000xxx)
Price Deviation		0.0004***				0.000184***
		(0.0002)				(0.0000xx)
Constant	0.7401***	0.6923***	0.2923***	0.0031***	0.0224***	0.080793***
	(0.0081)	(0.0114)	(0.0077)	(0.0001)	(0.0010)	(0.000xxx)
R-squared	0.2774	0.2816	0.2027	0.1031	0.7152	0.7236
Adjusted R^2	0.2772	0.2813	0.2025	0.1029	0.7150	0.7234
F-statistic	3,789.6***	1,278.3***	3,758.0***	1,134.8***	24,787.5***	?
Observations	9,872	9,872	9,872	9,872	9,872	9,872

Notes: Standard errors in parentheses. t-statistics in brackets.
$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$.

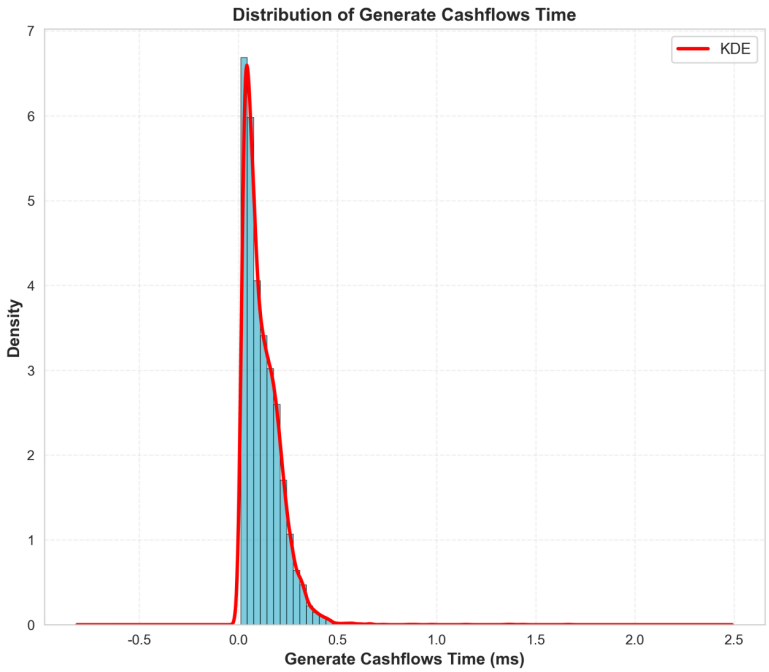


Figure 11: Hexbin: Distribution of Cash Flows Generation Time

B.3 Hexbin Plots of Three Components

C Detailed Code for Reference

C.1 Other Calculations

C.1.1 MSRB Day Count Convention

```
    def msrbDayCount(startDate,endDate):
        D1 = 30 if (startDate.day==31) else startDate.day
        D2 = 30 if ((endDate.day==31) and ((D1==30) or (D1==31))) else endDate.
            day
        M1 = startDate.month
        M2 = endDate.month
        Y1 = startDate.year
        Y2 = endDate.year
        days = (Y2 - Y1)*360 + (M2-M1)*30 + (D2-D1)
        return days
```

C.1.2 MSRB Price Rounding

```
def msrbRoundPrice(prc):
    rprc = None
    if prc is not None:
        rprc = floor(prc*1000.0)/1000.0
    else:
        pass
```

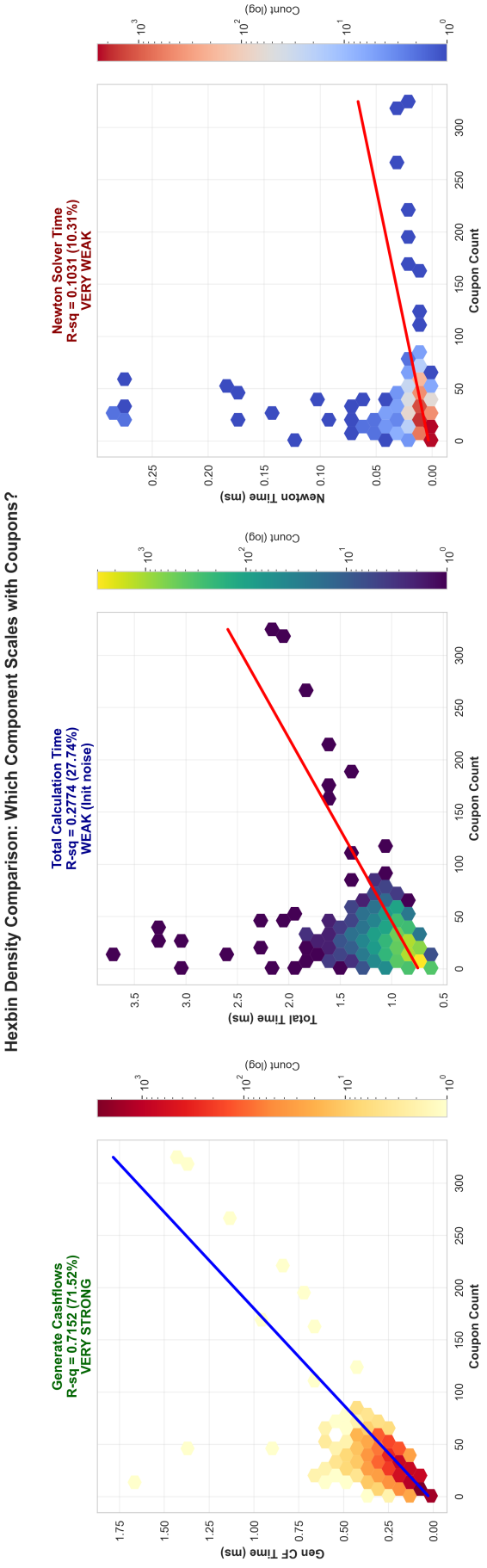


Figure 12: Hexbin Plots of Three Components

```
    return rprc
```

C.1.3 MSRB Yield Rounding

```
def msrbRoundYield(yld):
    trunc = str(floor(yld*10000.0))
    if int(trunc[-1])>=5:
        result = (int(trunc[:-1])+1)/1000.0
    else:
        result = int(trunc[:-1])/1000.0
    return result
```

C.2 Issue 3: Redundant Cashflow Generation for Multi-Workout Bonds

```
cache_key = (settleDate, workout['date'], workout.get('price', None))
if cache_key in self._cashflow_cache:
    cached = self._cashflow_cache[cache_key]
    self.currentCalcSettleDate = cached['settleDate']
    self.cashflows = cached['cashflows']
    self.calcPrevCouponDate = cached['prevCoupon']
    self.calcNextCouponDate = cached['nextCoupon']
    return
```

C.3 Experiment 1: Introduction of Better Initial Guesses for the Solver

```
defaultGuess = self.coupon - 0.0001 if p>100 else self.coupon + 0.0001
guess = overrideGuess if overrideGuess is not None else defaultGuess
```

C.4 Experiment 2: Numba Acceleration

C.4.1 Periodic Bond Pricing Core

```
@njit
def bond_price_periodic_core(y, RV, N, R, M, E, A, B):
    if N <= 1:
        P = ((RV/100.0)+(R/M))/(1+((E-A)/E)*(y/M)) - (R*A/B)
        P = P*100.0
    else:
        PV = 0.0
        coupon_amount = 100.0*R/M
        for k in range(1, N+1):
            t = k-1+((E-A)/E)
            df = 1.0/(1+y/M)**t
            PV = PV + df*coupon_amount
```

```
        t_principal = N-1+((E-A)/E)
        principal_df = 1.0/(1+y/M)**t_principal
        P = RV*principal_df + PV - (100.0*R*A/B)
    return P
```

C.4.2 Newton Solver Core

```
@njit(fastmath=True)
def newton_solver_optimized(
    target_price, initial_guess, RV, N, R, M, E, A, B,
    tol=1e-4, maxiter=100
):
    # Numba-optimized Newton solver for bond yield
    y = initial_guess

    for k in range(maxiter):
        current_price = bond_price_periodic_core(y, RV, N, R, M, E, A, B)
        f_val = current_price - target_price

        if abs(f_val) < tol:
            return y

        # Use a relative step to improve numerical stability
        eps = 1e-8 * (1.0 + abs(y))
        price_plus = bond_price_periodic_core(y + eps, RV, N, R, M, E, A, B)
        derivative = (price_plus - current_price) / eps

        # Guard against near-zero derivative to avoid huge steps
        if abs(derivative) < 1e-15:
            return y

        y = y - f_val / derivative

    return y
```

C.4.3 Warm-up Function

```
def warm_up_numba():
    """Pre-compil_warm-up"""
    warm_cusip = '797299KR4'
    warm_bond = muniBond.muniBond(warm_cusip)
    _ = warm_bond.ytw(100.0)
```

C.5 Experiment 4: Parallel Computing

```
# Preparation for parallel processing
args_list = [(row['securityId'], row['bidPx']) for _, row in df_sample.iterrows
    ()]
n_processes = 4

# Main calculation
with Pool(processes=n_processes) as pool:
results = pool.map(calculate_ytw_for_bond, args_list, chunksize=100)
```

D MuniBond Variables Codebook

This appendix provides a comprehensive mapping of all variables used in the Municipal Bond pricing and risk management system.

D.1 Core Identifiers

Variable	Type	Description
cusip	string	Committee on Uniform Securities Identification Procedures code – unique 9-character identifier for US securities
isin	string	International Securities Identification Number – global unique identifier
ticker	string	Trading symbol or abbreviated identifier
bref	DataFrame row	Bond reference data row from security master

D.2 Date Variables

D.2.1 Maturity & Lifecycle Dates

Variable	Type	Description
maturityDate	date	Final maturity date of the bond
accrualDate	date	Date from which interest begins to accrue
firstCouponDate	date	Date of the first coupon payment
effectMaturityDate	date	Actual maturity considering refundings (refundDate if applicable, else maturityDate)

D.2.2 Coupon Payment Dates

Variable	Type	Description
priorCouponDate	date	Previous coupon payment date (alias: prevCouponDate)
nextCouponDate	date	Next scheduled coupon payment date
penultCouponDate	date	Penultimate (second to last) coupon date (alias: lastPeriodAccrualDate)
calcPrevCouponDate	date	Calculated previous coupon date for pricing calculations
calcNextCouponDate	date	Calculated next coupon date for pricing calculations

D.2.3 Call & Put Dates

Variable	Type	Description
nextCallDate	date	Next available call date
parCallDate	date	Date when bond becomes callable at par (100)
nextPutDate	date	Next available put date
refundDate	date	Date when bond is scheduled to be refunded
refundAnnounceDate	date	Date when refunding was announced

D.3 Financial Variables

D.3.1 Coupon & Interest

Variable	Type	Description
coupon	float	Annual coupon rate (as decimal, e.g., 0.05 for 5%)
couponType	string	Type of coupon (Fixed rate, Zero coupon, Variable, etc.)
intFreqDesc	string	Interest frequency description (Semiannually, Annually, etc.)
intFreq	float	Interest payment frequency per year (2.0 for semiannual)
intPeriodMonths	int	Months between interest payments
intPeriodDays	float	Days in each interest period

D.3.2 Pricing

Variable	Type	Description
effectiveMaturityPrice	float	Price at effective maturity (100.0 or refundPrice)
nextCallPrice	float	Price if called at next call date
parCallPrice	float	Price if called at par call date (usually 100.0)
nextPutPrice	float	Price if put at next put date
refundPrice	float	Price at which bond will be refunded

D.3.3 Yield & Risk

Variable	Type	Description
y	float	Yield to maturity (as decimal)
yld	float	Calculated yield result
p	float	Bond price
prc	float	Calculated price result

D.4 Rating Variables

D.4.1 S&P Ratings

Variable	Type	Description
spLongRating	string	Standard & Poor's long-term rating (AAA, AA+, AA, etc.)
spUnderlyingRating	string	S&P underlying credit rating (without insurance enhancement)
spShortRating	string	S&P short-term rating (A-1+, A-1, etc.)
spIssuerRating	string	S&P issuer credit rating
spLongOutlook	string	S&P rating outlook (Stable, Positive, Negative, Developing)

D.5 Technical Calculation Variables

D.5.1 Day Count & Calendar

Variable	Type	Description
dayCount	string	Day count convention (30/360, Actual/365, etc.)
daysInYear	int	Number of days in year for calculations (360 or 365)
A	int	Days from accrual date to settlement date
B	int	Days in year (same as daysInYear)

Variable	Type	Description
DIR	int	Days from accrual date to effective maturity
E	int	Days between previous and next coupon dates

D.5.2 Mathematical Variables

Variable	Type	Description
M	float	Payment frequency per year (same as intFreq)
N	int	Number of remaining coupon payments
R	float	Annual coupon rate (same as coupon)
RV	float	Redemption value (usually 1.0 or workout price/100)
PV	float	Present value calculation intermediate
df	float	Discount factor
DF(t)	float	Discount factor function at time t

D.6 Feature Flags & Options

D.6.1 Call/Put Features

Variable	Type	Description
callable	boolean	Whether bond is callable before maturity
putType	string	Type of put feature (if any)
refundType	string	Type of refunding (Called, Pre-refunded, ETM, etc.)

D.6.2 Special Categories

Variable	Type	Description
specialRefundTypes	list	List of special refund types that modify workout logic

D.7 Data Structures

D.7.1 Workout Scenarios

Variable	Type	Description
workouts	list of dict	All possible maturity scenarios with date, price, and type
workout	dict	Single workout scenario {date, price, type}

D.7.2 Cash Flows

Variable	Type	Description
cashflows	dict	Dictionary containing coupon and principal cash flows
cashflows['coupon']	list	List of coupon payment cash flows
cashflows['principal']	dict	Principal repayment cash flow

D.7.3 Black Lists (Compliance)

Variable	Type	Description
blockedCusips	list	List of blocked CUSIP identifiers
blockedIssuerTokens	list	List of blocked issuer name keywords
blockedObligorTokens	list	List of blocked obligor name keywords
blockedStates	list	List of blocked state codes

D.8 Classification Variables

D.8.1 Rating Buckets

Variable	Type	Description
ratingBucket	string	Credit rating classification (AAA, AA, A, BBB, etc.)
compositeRating	string	Combined S&P and Moody's rating
compositeRatingRank	int	Numeric rank of composite rating

D.8.2 Tenor Buckets

Variable	Type	Description
tenorBucket	string	Maturity classification (short, front, intermediate, long)
callBucket	string	Call feature classification
tenor	int	Years to maturity
callTenor	int	Years to next call

D.8.3 Size & Coupon Buckets

Variable	Type	Description
maturitySizeBucket	string	Issuance size classification (micro, small, medium, round, large)
cpnBucket	string	Coupon rate classification (3, 4, 5, 6)

D.9 Mathematical Functions & Constants

D.9.1 MSRB Functions

Function	Description
msrbDayCount(start, end)	MSRB 30/360 day count calculation
msrbRoundPrice(prc)	MSRB price rounding to nearest 1/8 of 1/32
msrbRoundYield(yld)	MSRB yield rounding to nearest 0.001%

D.9.2 Frequency Mapping

Variable	Value	Description
regSettleDays	1	Regular settlement days
Semiannually	2.0	Payment frequency
Annually	1.0	Payment frequency
Monthly	12.0	Payment frequency
Quarterly	4.0	Payment frequency

D.10 Reference Data Variables

D.10.1 Bond Characteristics

Variable	Type	Description
issuerName	string	Name of bond issuer
obligor	string	Ultimate obligor/guarantor
stateCode	string	State of issuance
useOfProceeds	string	Purpose of bond proceeds
fedTaxStatus	string	Federal tax treatment
stateTaxStatus	string	State tax treatment
assetStatus	string	Current status (Live, Matured, Called, etc.)
isDefaulted	boolean	Whether bond is in default
isBQ	boolean	Whether bond is bank qualified

D.10.2 Trading Parameters

Variable	Type	Description
minPiece	int	Minimum trading increment
minIncrement	int	Minimum size increment
principalFactor	float	Outstanding principal factor (1.0 = full principal)

D.11 Eligibility & Compliance Codes

Code	Meaning
ELIGIBLE	Bond passes all eligibility checks
MISSING_REF_DATA	Required reference data not available
MAKE_WHOLE_CALL	Bond has make-whole call provision
BANK_QUALIFIED	Bond is bank qualified (restricted)
IRREGULAR_INTEREST_FREQUENCY	Non-standard payment frequency
IMMINENT_CALL	Call date within 1 year
BELOW_RATING_CUTOFF	Rating below minimum requirement
OUTSIDE_COUPON_RANGE	Coupon outside acceptablerange
BLOCKED_CUSIP	CUSIP on blocked list
BLOCKED_ISSUER	Issuer on blocked list
BLOCKED_STATE	State on blocked list

This codebook serves as a comprehensive reference for understanding all variables used in the Municipal Bond pricing and risk management system.