Precision-Aware Numerical Computing in `modelestimator`

Oscar Foley

January 4, 2026

Abstract

Understanding how protein sequences evolve over time is a central problem in computational biology. Models of amino-acid substitution describe this process using continuous-time Markov chains, where the accuracy and efficiency of numerical computations are critical for estimating evolutionary parameters from large sequence alignments. The modelestimator framework estimates amino-acid substitution rate matrices by aggregating empirical observations from aligned protein sequences. The original modelestimator program was written in 2006 [1]. An updated python implementation, modelestimator-v2 was created in 2019[2].

This project uses the `modelestimator` framework as a case study to investigate the energy and performance implications of using single-precision (float32) arithmetic in numerical computation. The program was originally implemented with default NumPy double-precision (float64) arrays; we compare this to a version where the computation is refactored to use float32 representations throughout.

By instrumenting the execution with Linux `perf` and its energy-related performance counters, I observe an approximate $\approx$ 3–5% reduction in energy consumption with minimal loss of numerical fidelity for the target workload. This essay provides the theoretical motivation for mixed/lowered-precision numerical computation, analyzes the behavior of NumPy's type system, and explains both when and when not precision changes affect energy characteristics. The results demonstrate when carefully chosen numerical precision can serve as a practical optimization strategy for scientific computing pipelines.

Sammanfattning

Förståelsen av hur proteinsekvenser utvecklas över tid är ett centralt problem inom beräkningsbiologi. Modeller för aminosyrasubstitution beskriver denna process med hjälp av kontinuerliga Markovkedjor, där noggrannheten och effektiviteten hos de numeriska beräkningarna är avgörande för att kunna uppskatta evolutionära parametrar från stora linjeringar.

Ramverket modelestimator används för att estimera substitutionshastighetsmatriser för aminosyror genom att aggregera empiriska observationer från proteinsekvenslinjeringar. Det ursprungliga modelestimator-programmet utvecklades år 2006 [1]. En uppdaterad Python-implementation, modelestimator-v2, skapades år 2019 [2].

Detta projekt använder ramverket modelestimator som en fallstudie för att undersöka energi- och prestandakonsekvenserna av att använda enkelprecision (float32) i numeriska beräkningar. Programmet var ursprungligen implementerat med NumPys dubbelprecision (float64), och jämförs här med en version där beräkningarna konsekvent har omstrukturerats till att använda float32.

Genom att instrumentera exekveringen med Linux-verktyget `perf` och dess energirelaterade prestandaräknare observeras en minskning av energiförbrukningen på cirka 3–5%, med minimal försämring av den numeriska noggrannheten för den studerade arbetsbelastningen. Arbetet presenterar den teoretiska grunden för beräkningar med blandad och reducerad precision, analyserar NumPys typsystem samt förklarar både när och när ändringar i numerisk precision påverkar energiegenskaper. Resultaten visar att ett medvetet val av numerisk precision kan utgöra en praktisk optimeringsstrategi för vetenskapliga beräkningar.

Acknowledgements

I would like to thank Lars "Lasse" Arvestad for providing the thesis subject and for his guidance throughout the project.

Contents

1 Introduction

1.1 Mathematical Background on Floating-Point Precision

Numerical computing has undergone a significant transformation in recent decades. As data volumes increase and computational models become more complex, the demands placed on commodity hardware – laptops, workstations, and cloud instances – continue to grow. While algorithmic efficiency remains paramount, attention has increasingly turned toward numerical precision as an underexplored performance axis. Many numerical workloads default to 64-bit floating-point arithmetic, even when such precision exceeds the stability needs of the underlying mathematical operations. As the energy expenditure of large-scale computations grows, there emerges a compelling argument to weight the cost of precision against the cost of electricity.

Floating-point numbers follow the IEEE-754 standard , which defines formats such as single precision (float32) and double precision (float64). A float64 number provides approximately 15-16 decimal digits of precision, whereas float32 provides about 6-7 digits[3].

The concept of machine epsilon expresses the smallest representable difference between 1 and the next floating-point number. For float32 this value is roughly 1.19×10^{-7}, and for float64 it is about 2.22×10^{-16} [13]. A matrix has property called its conditioning number which roughly states how much error is introduced through mathematical operations. To properly motivate the correctness of the float32 version of the algorithm, beyond the fact that all testing data producing identical results using both float32 and float64 we will also attempt to prove that a reduction in machine epsilon will not impact the calculation by showing that the algorithm uses well-conditioned matrices.

The original `modelestimator` paper presents a Markov-model framework for estimating how related protein sequences evolve over time[1]. Aligned sequences of common ancestry are treated as a continuous-time substitution process, and the software infers the rate matrix that best explains the observed amino-acid changes. There are a lot of different observed rate matrices and in the literature they are divided into different models, for example the WAG [7] and JTT [6] model. This thesis preserves the biological model and focuses on analyzing how the same computation behaves when implemented using different floating-point precisions.

This project examines energy efficiency in practice by studying the relative energy usage of a refactored version of the Python-based `modelestimator` project [2] and the original. The original implementation uses 64 bit floats, and the refactored fork replaces these with 32 bit floats whenever possible. By benchmarking both versions using Linux `perf` and then quantifying the energy we see the impact of halving the storage, memory and arithmetic width. The research questions are as follows

- Does reduced precision affect energy consumption?
- Dose it affect runtime?
- Does it preserve numerical correctness?

Previous research on this topic has shown that reduced precision does not harm the correctness of many scientific and machine-learning workloads. Studies on mixed precision, particularly in GPU computing, demonstrate that reduced precision often yields substantial performance benefits without compromising results. In a 2022 paper Micikevicius, P et al. showed techniques for using mixed precision when training and running a neural network. The paper discusses that this approach could almost halve the memory requirements and that on modern GPUs it could lead to a 2x to 8x speed up without significantly affecting the accuracy of the model[8].
Similarly Baboulin, M et al. showed in 2009 that many scientific computations could be accelerated using mixed precision algorithms without significant loss of accuracy[12]. The paper presents the speed differences, both in memory access and computation time, between single, double and quadruple precision.

2 Methods

2.1 Background: Markov Models and the BW-Method

The mathematical foundations of the `modelestimator` software are grounded in Markov models of amino-acid evolution. As described in the original Modelestimator-V2 thesis [2], proteins of common ancestry are modeled as sequences evolving under a continuous-time Markov process governed by a rate matrix Q. The BW-method decomposes the problem of estimating Q into two subproblems:

1. Estimating its eigenvectors using aggregated frequency matrices derived from aligned protein pairs.
2. Estimating its eigenvalues by clustering divergence-weighted sequence pairs.

A rate matrix Q satisfies the standard properties of a generator of a continuous-time Markov chain: off-diagonal entries are non-negative, diagonal entries are negative row-sums, and $P(t) = e^{Qt}$ defines transition probabilities. The BW-method exploits the fact that Q and all associated transition matrices share eigenvectors, enabling stable eigenvector estimation through aggregated empirical counts. This of course allows one to generate an estimation of Q from an estimated $P(t)$

This project does not modify the mathematical model but evaluates how its computational realization behaves under reduced precision.

2.2 Mathematical Formulation of Core Operations

The computations in the `modelestimator` project rely heavily on linear algebra operations, including matrix multiplication, vector transformations, and least squares style updates.

For linear systems of the form $Ax = b$, classical numerical analysis provides an upper bound on the forward error: the relative error in the computed solution $\hat{x}$ is bounded by the condition number of A multiplied by machine epsilon[13]. So if our matrix A has a conditional number that when multiplied by machine epsilon is lower than 10^{-3} (i.e., the chosen threshold for acceptable precision), then even using float32 arithmetic will not significantly degrade the accuracy of the solution. Because many operations in the project take the form $y = Ax + b$ and $A \cdot B$ [2], small rounding errors will not meaningfully influence the output so long as input matrices are reasonably well scaled.

Stage	Time (s)
Building matrix from input (integer-only)	25
Floating-point operations	3
Output handling	1

Table 1: Approximate runtime breakdown isolating the floating point work from I/O and preprocessing.

2.3 Code Refactoring

To complement the theoretical expectations, I conducted a detailed analysis of hardware performance counters for two versions of the `modelestimator` codebase.

- The original `modelestimator` project, implemented using NumPy float64 arrays.
- A modified fork that replaces all instances of NumPy float64 with float32.

This study compares the two codebases, from now on referred to V2 and V3 respectively.

I refactored modelestimator-v2 [2] with a few things in mind.

1. **Type Assertions:** Through detailed code review and debug statements ensure that all initialized numbers and arrays are float32.
2. **Type Consistency:** Ensuring that intermediate computations do not implicitly cast float32 arrays back to float64. Built in NumPy functions often upcast inputs unless explicitly given a type.
3. **Algorithmic Preservation:** Maintaining identical algorithmic structure ensures that performance differences arise only from type changes.
4. **Program Reduction:** Trimming both versions to isolate the core mathematical operations from I/O overhead.

The first three points are trivial but to conduct the measurements certain changes to the program had to be done. The naive approach of running the benchmarks of the whole program would not properly answer the question of the change caused by the floating point precision change. This is because the majority of the program's run time is spent on creating the matrices representing the transitions between the various protein alignments in the sequence, see table 1. These are integer matrices, and are therefore not affected by our floating point change . To circumvent this I precomputed those matrices for all of the testing data and simply loaded the variable from file during testing. Cutting away the output handling was also crucial in getting only the raw mathematical operations on display. But to ensure that both programs gave the same result, before cutting the output step I ran the two versions on the input data and compared the given results. Once they were shown to be identical, only then could I remove the output parsing and only run the calculations.

2.4 Performance and Energy Measurement

Energy data was collected using `perf` with Running Average Power Limit (RAPL) counters on a Linux machine[4][5]. All extraneous processes were shut down during the testing to ensure as little noise as possible. This includes all graphical processes as well as background services such as the Network Manager. Because of the limits in the `perf` program you can only measure the energy usage of the whole CPU, not a singular process. So depending on how heavy the program is computationally even the smallest background noise such as the screen and OS being turned on could dominate. To account for this, a regular system benchmark was conducted, to see the energy usage per unit time for the already stripped Linux machine. When all these precautions were taken, I could in good confidence run the benchmarks of the two programs. Specifically, the following command was used to gather performance and energy metrics:

```
$ sudo perf stat -a  -e cycles,instructions, cache-references,
cache-misses,branches,branch-misses, cpu-clock,task-clock,
stalled-cycles-frontend, stalled-cycles-backend,
uops_issued.any, uops_executed.core,l1d.replacement,
l2_rqsts.references, l2_rqsts.miss,
mem_load_uops_retired.l1_miss, mem_load_uops_retired.l2_miss,
mem_load_uops_retired.l3_miss,power/energy-pkg/,
power/energy-cores/,power/energy-ram/
```

The measurements include a lot of different performance metrics, but we are mostly interested in the following:

- Execution time
- Energy consumed by CPU cores
- Energy consumed by DRAM
- Cache metrics

Measurements were taken over repeated runs to ensure statistical stability. To gather the benchmark data, I used the phylogenetic simulation software iqtree3 [10] to generate amino acid sequence alignments[11]. I generated ten sequences of length 500, ten of 510, all the way up to ten sequences of length 1000, for a total of 500 sequences. CPU frequency scaling was disabled to avoid confounding results.

2.5 Numerical Stability Evaluation

To evaluate numerical differences between float32 and float64 results, I compared outputs elementwise using relative error metrics and assessed whether discrepancies meaningfully affected downstream model behavior. The threshold for acceptable numerical deviation was set at 0.001, consistent with the original modelestimator's tolerance.

3 Numerical Conditioning Analysis

To be able to properly motivate that the change from float64 to float32 is acceptable in this case we need to look at the mathematical properties of the operations being done in the program. The main operations being done are matrix multiplications, dot-products, eigenvalue decompositions and least squares fitting[2]. All of these operations have been studied extensively in numerical analysis literature and it has been shown that they can be performed in lower precision without significant loss of accuracy, provided that the underlying matrices are well-conditioned.[13]. To properly motivate this we can look at the condition number of a matrix, which is defined as the ratio of the largest singular value to the smallest singular value[13]. A matrix with a low condition number is said to be well-conditioned, meaning that small changes in the input will result in small changes in the output. Conversely, a matrix with a high condition number is said to be ill-conditioned, meaning that small changes in the input can result in large changes in the output. All the matrices the program works with do not contain floating point numbers [2], so we will only be discussing the conditional numbers for certain matrices.

To be able to determine if the matrices are well-conditioned or not we will work backwards from a theoretical output. We start by taking a given rate matrix, let's say WAG[10][7]. Then by the assumption of the original modelestimator paper [2] we know that the calculated eigenvectors from the observed frequency matrix are the same as its rate matrix. So the eigenvector matrices, which are the matrices we are primarily interested in, can for any given rate matrix be shown to be well conditioned. This by simply calculating the eigenvectors of that rate matrix. So working from the theoretical result matrix, we can construct the matrices we use in the modelestimator program and show that they are well conditioned. The used float based matrices used in matrix multiplication operations are as follows

- The right eigenvector of Q
- The left eigenvector of Q
- The matrix created by putting the frequency list on the diagonal (np.diag(eq))
- The matrices created by raising Q to different powers between 0 and 400.

So by reconstructing Q from the rate matrix and frequency list we can calculate the conditional numbers of all of these matrices.

The error is relative in size to the product of the condition number and the machine epsilon of the used floating point representation[13]. In our

Matrix	Condition Number
Right Eigenvector of Q	2.24
Left Eigenvector of Q	2.24
Diagonal Frequency Matrix	6.02
Powers of Q (biggest)	6.18

Table 2: Condition numbers of key matrices used in `modelestimator` for the WAG model.

Matrix	Condition Number
Right Eigenvector of Q	1.96
Left Eigenvector of Q	1.96
Diagonal Frequency Matrix	6.44
Powers of Q (biggest)	6.81

Table 3: Condition numbers of key matrices used in `modelestimator` for the JTT model.

case this means that the conditional number far below 10^5 will not cause any significant error when using float32, as the machine epsilon for float32 is roughly 10^{-7}. All of the calculated conditional numbers are below this threshold, meaning that we can be confident that using float32 will not cause any significant numerical errors in the calculations. This has also been shown to be true for JTT, vt, mtArt and flu.

4 Results and Discussion

4.1 Perf-Based Quantitative Comparison of V2 (float64) and V3 (float32)

The V2 and V3 `perf` outputs reveal several trends consistent with precision-driven performance improvements.

4.1.1 Execution Time

When comparing the average time the two programs took to run we see the following results:

- V2 total task-clock: ~ 5665 ms.
- V3 total task-clock: ~ 5531 ms.

A modest $\sim 2.4\%$ speedup was observed. This is a lot smaller than the halving of memory access would imply. But this is however expected for a few reasons. To begin with, even after stripping the code, `modelestimator` still contains substantial Python-level overhead not accelerated by narrower types. This could be remediated by once again rewriting the program into a lower level language such as C++ or Rust, but that is outside the scope of this project (See appendix A for a Rust based implementation of the first part of the algorithm). The expectation of improved performance when using float32 arises from two main considerations. First, single-precision numbers occupy half the memory of double-precision numbers, which theoretically reduces memory bandwidth requirements and improves cache efficiency. Second, on many hardware architectures, single-precision operations can be executed more rapidly than double-precision operations because vectorized instructions can process more elements per cycle [9].

The assumption of smaller precision being faster holds on modern GPUs, which are heavily optimized for single-precision workloads[8]. However, on CPUs, the situation is more nuanced. Furthermore, many NumPy operations are limited by memory bandwidth rather than arithmetic speed. Reducing the precision of stored values from 64 to 32 bits only marginally affects performance unless the working dataset is large enough to exceed the CPU cache and saturate memory throughput. The size of the floating point matrices that were worked on are also only 20×20. This is a relatively small size and does not leave a significant memory footprint. The theorized energy savings would have come from needing to read half as much from memory. In the case of the hardware the tests were run on, an Intel(R) Core(M) i3-4030u CPU. This particular CPU, despite being over a decade old at this point, has an L1 data cache of 32 KBs and an L3 of 3 MB. Which is several orders of magnitudes bigger than the 1.6 KBs required to store the 400 double precision floating point numbers in the matrix that the program uses. So in the case of 'modelestimator' all of the floating point numbers can fit in the CPU cache, regardless of if they are four or eight bytes. This largely eliminates the need for memory access and thus diminishes the potential decrease of energy expenditure from accessing memory.

4.1.2 Instruction and Cycle Behavior

The change in instruction and cycle counts between V2 and V3 is slight:

- Total CPU instructions decreased minimally ($\sim 10.74\text{B} \rightarrow \sim 10.38\text{B}$).
- Instructions per cycle decreased by a similar amount ($\sim 1.03 \rightarrow\sim 1.01$).

This suggests that the control-flow structure of the program is similar, and any gains originate from memory and vectorization effects rather than algorithmic changes. Wich alignes with the fact that no changes to the algorithm were made.

4.1.3 Cache Behavior

The change in cache statistics are as follows:

- L1 miss rates unaffected.
- L2 miss rates decreased by about 8.4 %
- L3 cache miss rates unaffected.

This indicates some improvement stemming from the cache usage of the program decreasing. But, as the L3 cache miss rate is unaffected it shows that the working set still fits in the cache regardless of precision. This means that the both programs read about as much data from memory. So the hypothesised energy saved from reducing memory access, is naught. And while the float64 version more often has to go to the L3 cache this is not a significant decrease and thus didn't have a large effect the observed energy usage.

4.1.4 Energy Consumption (System-Wide)

The energy consumption as measured by the RAPL energy-cores is as follows:

- V2 energy-cores: 11.25 J.
- V3 energy-cores: 10.82 J.

This is an $\sim$ 3.8% reduction for the full testing data. For individual runs in I observed energy decreases around $\sim$ 3–5%. It is worth noting however that short `perf stat` windows often underestimate energy deltas because baseline OS activity dominates [9], but V3 consumed less energy than V2 in all the testing data.

4.2 Numerical Error Analysis

Numerical deviations between float32 and float64 outputs were assessed using relative error metrics. Across all tested workloads, errors remained within the expected range for float32 arithmetic, generally around one part in a million. These deviations did not accumulate excessively across iterations, indicating numerical stability. For all the testing data and using the default threshold of 0.001 V2 and V3 produced identical results.

4.3 Summary and Interpretation of Findings

The refactoring yielded an approximate $\sim$ 3–5% reduction in energy consumption as measured by perf. Execution time also improved modestly, although the primary benefit was energy efficiency. Although modern GPUs can handle float32 operations more efficiently, due to SIMD packing density and increased vectorization [9], this was not the case for the V2 and V3 implementations, due to using NumPys CPU native calculations. Energy savings slightly exceeded the speedup because energy correlates not only with elapsed time but also with how energy intensive the instruction are each cycle. Float64 operations tend to demand more from execution units and memory controllers, increasing power even if runtime differences appear small.

Overall, the findings support a precision-aware computing strategy for scientific workloads that can tolerate single-precision arithmetic without sacrificing correctness.

5 Conclusion

Using the `modelestimator` project as a case study, I show that the transition reduced energy usage by approximately $\approx$ 3–5%, primarily due to marginally lower memory-bandwidth. For this specific program however the particular floating point matrices that are being worked with are not big enough for the memory saved to have any real effect. The findings do however hint that for more memory intensive programs as well as programs written in languages with a higher degree of type control would see higher speed ups. See appendix A for a Rust based implementation of the first part of the algorithm.

Through calculating the conditional numbers of the matrices used in the program for different rate matrices, we can show that the numerical accuracy will not be noticably affected by the change in precision. These mathematical results also proved correct in theory when the change in precision didn't change any of the result the program produced, calculated with a threshold of 0.001.

Beyond the specific codebase, this work highlights the broader importance of aligning numerical precision with algorithmic needs. Tools such as NumPy provide flexible control over types, and system-level utilities like `perf` enable quantitative evaluation of their impact. As energy efficiency becomes an increasingly important concern in scientific computing, precision-aware numerical methods offer a practical and effective optimization strategy.

6 References

References

[1] Arvestad L. Efficient methods for estimating amino acid replacement rates. J Mol Evol. 2006 Jun;62(6):663-73. doi: 10.1007/s00239-004-0113-9. Epub 2006 Apr 28.

[2] Ridderström, R. *Approximating amino acid replacement rates efficiently through weighted data aggregation*. Bachelor's thesis, Stockholm University, 2019.

[3] IEEE Standards Association. *IEEE Standard for Floating-Point Arithmetic (IEEE 754-2019).*

[4] Linux `perf` documentation. `https://perf.wiki.kernel.org/`.

[5] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna and C. Le, RAPL: Memory power estimation and capping, 2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED), Austin, TX, USA, 2010, pp. 189-194, doi: 10.1145/1840845.1840883.

[6] Jones, D. T., Taylor, W. R., & Thornton, J. M. (1992). The rapid generation of mutation data matrices from protein sequences. Computer Applications in the Biosciences, 8(3), 275–282. `https://doi.org/10.1093/bioinformatics/8.3.275`

[7] Whelan, S., & Goldman, N. (2001). A general empirical model of protein evolution derived from multiple protein families using a maximum-likelihood approach. Molecular Biology and Evolution, 18(5), 691–699. https://doi.org/10.1093/oxfordjournals.molbev.a003851

[8] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaev, O., Venkatesh, G., Wu, Hao. (2017). Mixed Precision Training. 10.48550/arXiv.1710.03740.

[9] Hennessy, J. L., Patterson, D. A. *Computer Architecture: A Quantitative Approach*. 6th ed., Morgan Kaufmann, 2017.

[10] T.K.F. Wong, N. Ly-Trong, H. Ren, H. Banos, A.J. Roger, E. Susko, C. Bielow, N. De Maio, N. Goldman, M.W. Hahn, G. Huttley, R. Lanfear, B.Q. Minh (2025) IQ-TREE 3: Phylogenomic Inference Software using Complex Evolutionary Models. Submitted, https://doi.org/10.32942/X2P62N.

[11] N. Ly-Trong, G.M.J. Barca, B.Q. Minh (2023) AliSim-HPC: parallel sequence simulator for phylogenetics. Bioinformatics, 39:btad540. https://doi.org/10.1093/bioinformatics/btad540

[12] Baboulin, M., Buttari, A., Dongarra, J., et al. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2009.

[13] Higham, N. J. *Accuracy and Stability of Numerical Algorithms.* SIAM, 2002.

7 Appendix A. Rust Implementation of Modelestimator

In modelestimator-v2 and 3, all float based calculations are done using highly optimized cpython NumPy code. This, together with the fact that 90% of the runtime was spent on creating the count matrices, made it obvious that the room for improvement was in the calculating of the count matrices. To do so I wrote a Rust based implementation of the core algorithm. Rewriting it in Rust meant eliminating slow python overhead, and the possibility of using enums instead of strings for the protein types. Also allowing for a SIMD implementation of the matching_letters function, among other algorithmic speed ups. Local non rigorous testing showed that this implementation was around 100x faster than the python version for the same input data, V2 took 2.6 seconds and the rust version took 0.018 seconds.

The code can be found at:
`https://github.com/Poacatat/modelestimator-v4`.