ApproxSciMate: A Python library for approximating SciPy functions

Eleni Papadopoulou, Lucian Negru, Yang Li

March 28, 2024

1 Problem

Statistical computations often require complex mathematical operations that can be computationally expensive, especially when large datasets are used. Typical statistical libraries like SciPy contain several functions that operate in high precision, sometimes at the expense of unnecessary resource consumption. High-precision computations require more memory and computational resources, potentially leading to scalability issues, especially in memory-constrained environments. Another problem with high-precision computations is that they can result in longer execution times, reducing the responsiveness and efficiency of software applications, especially in time-sensitive or real-time systems. Regarding sustainability, high-precision computations can significantly impact energy consumption, even when high precision is not needed.

Many real-world use cases can benefit from the paradigm of approximate computing. Machine Learning and AI training can integrate approximate computing when adapting existing models, designed to run locally on performant systems, to lower-powered devices such as the Rabbit R1[1] or a smartphone. Approximations are also an integral part of signal processing, where the incoming data streams implement compression, can suffer from transmission-related losses regardless and where processing speed is prioritised over accuracy, such as in autonomous driving sensors. Video games – the largest entertainment industry in the world – benefit immensely from computational approximations in their rendering pipelines, where frame generation rate is often more important than graphics quality.

All of the aforementioned fields are some of the largest in the tech industry, having a large impact on environmental and economic sustainability. In this report, we want to focus on the environmental aspect of approximate computing and explore how it can be used to reduce our negative impact. We explore one of the most popular Python libraries, SciPy[2], and apply the paradigm of approximate computing in a new library called *ApproxSciMate*[3].

2 Why we choose SciPy functions

SciPy, a fundamental library for scientific computing in Python, encompasses a wide range of functions that are crucial for mathematics, science, and engineering. These functions cover areas such as linear algebra, optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations, and statistics. The importance of these functions lies in their ability to provide accurate methods for numerical computation, which is essential for modelling, simulating, and solving complex problems across various domains of science and engineering. SciPy contains many high-precision functions that may consume unnecessary resources when performing high-precision computations.

In particular, SciPy functions are widely used in areas such as machine learning. Machine learning algorithms usually require a large number of statistical computations, such as probability distribution fitting, feature engineering, etc. These statistical functions are usually very computationally intensive. Therefore, accelerating the computation of these functions through approximation methods can greatly improve the training and inference efficiency of machine learning algorithms while reducing resource consumption, which is more in line with the goal of environmentally sustainable development.

[1]https://www.rabbit.tech/
[2]https://scipy.org/
[3]https://pypi.org/project/approxscimate/

3 Solution

To address the challenges above, we propose the development of a Python library containing proof-of-concept functions that approximate statistical computations with the goal of making them more energy efficient. *ApproxSciMate* would focus on providing efficient approximations to popular statistical functions currently found in libraries like SciPy. By offering a tunable level of approximation, *ApproxSciMate* aims to balance computational efficiency with acceptable accuracy, catering to a wide range of application requirements, and empowering the users to think about and choose their level of impact on the environment.

3.1 Target Functions

The library will initially focus on approximating the following functions chosen for their widespread use and computational intensity:

3.1.1 Cube Root Function

While not as used as the square root or other basic arithmetic operations, the cube root function is widely used in the areas of computer graphics, signal compression algorithms, error detection and correction, and cryptography. In our library, we aim to provide 2 additional levels of approximation to this function based on Halley's and Newton's methods of root approximation.

3.1.2 Combination Function

The combination function, or nCk (read as "n choose k"), calculates the number of ways to arrange k items out of a set of n distinct items, where the order of selection **does not** matter. The function curve increases exponentially with n and follows a quadratic curve for the k parameter. It is widely used in the areas of bioinformatics, economics, and data analysis. Our implementation will include 3 additional levels of approximation: a lower-bound guaranteed approximation, an upper-bound guaranteed approximation, and an approximation using Stirling's method of approximating factorials.

3.1.3 Permutation Function

The permutation function calculates the number of ways to arrange k items out of a set of n distinct items, where the order of selection **does** matter. It is a rapidly exponential function on both n and k that has many use cases in network routing, cryptography, and algorithm design and analysis. Our implementation of the permutation function includes 3 additional levels of approximation: a lower-bound guaranteed approximation, an upper-bound guaranteed approximation, and an approximation using Stirling's method of approximating factorials.

4 Implementation details

For each of the functions mentioned in the Solutions section, we have implemented additional levels of approximation on top of the existing SciPy ones. Each of the functions includes a level 0 which represents the default, accurate SciPy version. We have opted to keep the SciPy version as default to not mislead users who don't read the documentation and are simply looking for accurate result. The full implementation and documentation can be found here.

4.1 cbrt(n, level=0)

Level 1: Halley's Method

Halley's method is an iterative root-finding algorithm that improves upon Newton's method by incorporating the second derivative of the function. The iterative formula for Halley's method to approximate the cube root of a number x_n is given by:

$$x_{n+1} = \frac{x_n * {x_n}^3 + 2 * x}{2 * {x_n}^3 + x}$$

, where x_n is the current approximation

Halley's method offers faster convergence compared to Newton's method by taking into account the second derivative of the function. This additional information accelerates the rate of convergence towards the root, resulting in fewer iterations required to achieve the desired level of accuracy. The rationale behind using Halley's method lies in its superior convergence properties, which make it suitable for moderately accurate cube root approximation.

Level 2: Newton's Method

Newton's method is a classic iterative root-finding algorithm commonly used for approximating roots of equations. The iterative formula for Newton's method to approximate the cube root of a number x_n is given by:

$$x_{n+1} = x_n - \frac{{x_n}^3 - x}{3 * {x_n}^2}$$

, where x_n is the current approximation

Newton's method provides a straightforward approach to root approximation by iteratively refining an initial guess. While it may converge more slowly compared to Halley's method, it still offers a reliable means of approximation, particularly when computational resources are limited or when only a low level of accuracy is required. The rationale for using Newton's method lies in its simplicity and ease of implementation, making it suitable for basic cube root approximation tasks.

4.2 comb(n, k, level=0)

The combination function has the following formula: $\frac{n!}{k!(n-k)!}$

Level 1: Lower Bound Approximation

For the lower bound formula, we implement Farhis [Farhi, 2007] equality that:

$$\frac{n^k}{k^k} \leq \frac{n!}{k!(n-k)!}$$

The guaranteed lower bound approximation can be very useful in use cases where we are dealing with very large values of n and k, for example in cryptography. Knowing that a certain encryption length will have **at least** a certain number x of variations can help the user decide quickly whether its viable to consider in the first place or not.

Level 2: Upper Bound Approximation

For the upper bound approximation, we have implemented Farhi's equality:

$$\frac{n!}{k!(n-k)!} \leq \frac{n^k}{k!}$$

The upper bound calculation can be useful in fields such as algorithm design when determining if the worst-case complexity of a sub-set generation or graph traversal is more than what the domain requirements state. It can help remove options which have the potential of being too large for the use case.

Level 3: Stirling's method

Stirling's [Robbins, 1955] method is used to approximate the factorials used in the combination function. It proves that $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ as n tends to infinity. We implement it in the combination function alongside mathematical constants rounded to integers as such:

$$\frac{\sqrt{6n}\left(\frac{n}{3}\right)^n}{\sqrt{6k}\left(\frac{k}{3}\right)^n * max(1, \sqrt{6(n-k)}\left(\frac{n-k}{3}\right)^{n-k})}$$

This level is intended to provide a fairly high level of accuracy while reducing the complexity through the elimination of factorials.

4.3 perm(n, k, level=0)

The permutation function has the following formula: $\frac{n!}{(n-k)!}$

Level 1: Lower Bound Approximation

$k!$ provides a gross underestimation, essentially calculating the permutations of k items within themselves, ignoring the larger pool of n. This approximation might be used to quickly assess a very conservative lower bound of permutations but lacks practical accuracy since it doesn't consider the total number of items n.

Level 2: Upper Bound Approximation

We use n^k to estimate the permutations by considering each of the k selections to have all n possibilities, which over-counts since it allows repetitions. It is useful as an upper bound, especially when calculating probabilities where overestimations might be tolerable, but it's not accurate for exact counts due to the repetition issue.

Level 3: Stirling's method

This uses an approximation of $n!$ for large n based on Stirling's approximation, $\sqrt{2\pi n \left(\frac{n}{e}\right)^n}$, and rounds the mathematical constants to the nearest integer. Stirling's approximation is effective for large n because it closely estimates the logarithmic growth of factorial functions. It might not be accurate for smaller values of n or when n and k are close. The final formula becomes:

$$\frac{\sqrt{6n\left(\frac{n}{3}\right)^n}}{max(1, \sqrt{6(n-k)\left(\frac{n-k}{3}\right)^{n-k}})}$$

5 Validation

In this section, we will validate the implementation by using the function, and their varying levels, in stress tests designed to mimic real-world use cases. We will be looking at result accuracy and energy usage in particular, as these represent the potential use cases and the impact on environmental sustainability.

The experiments were automated and run on an M1 Pro CPU with minimal background processes and the same external variables.

5.1 Accuracy validation

We validate accuracy so we know that the functions can replace the original SciPy versions within certain degrees of error according to the varying use cases.

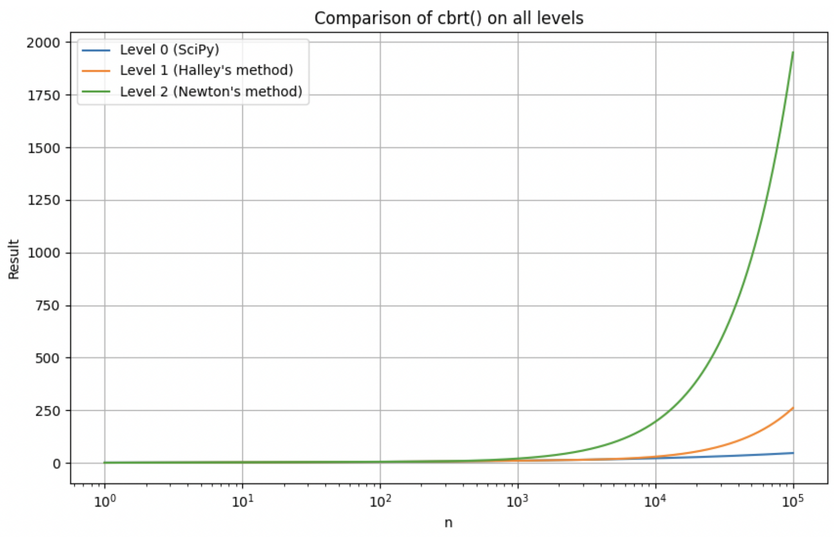

Figure 1: Comparison of the cube root function on all levels of approximation. We can observe that level 2 diverges from the SciPy version at a factor of 10 earlier than level 1.

cbrt()

The cube root approximation functions follow the SciPy level of accuracy until certain thresholds, where the rounding of constants and number of iterations make them eventually diverge. As seen in Figure 1, Newton's method (level 2) starts diverging from 10^3, while Halley's method (level 1) maintains accuracy for one more order of magnitude. Even though they both have the same number of iterations, Halley's method converges to a more accurate result faster.

comb()

The results for the combination function, seen in Figure 2, show that the upper bound approximation remains accurate for much longer than the lower bound. For values of $k < \frac{n}{2}$ it is a more suitable approximation, but for anything greater it deviates greatly. This is due to the logarithmic nature of the approximation and is something which does not affect the lower bound function, which is more suitable for the regions where $k > \frac{n}{2}$.

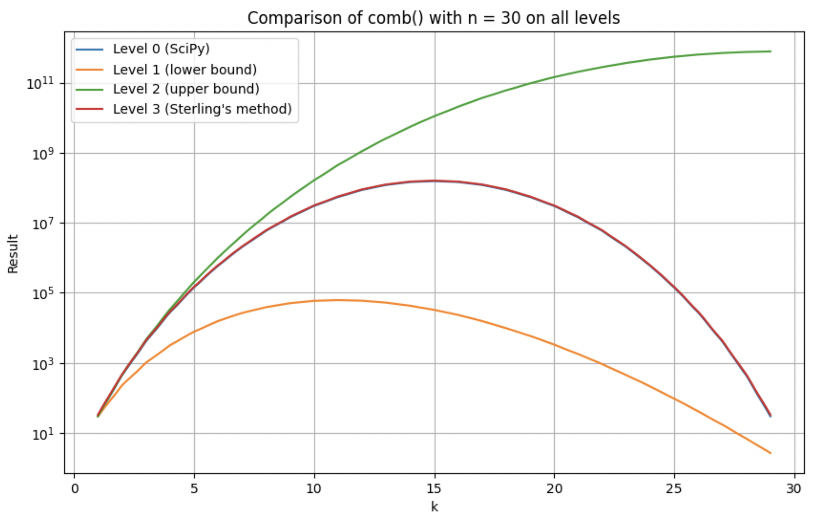

Figure 2: Comparison of the combination function on all levels of approximation. Note: on the logarithmic scale, levels 0 and 3 seem to overlap.

Though not visible in Figure 2, Sterling's method proves to be extremely accurate, as it seemingly overlaps the SciPy method. Only when looking closer, in Figure 3, can we see that it only slightly over-shoots the value – albeit by approx. $2.5 * 10^6$.

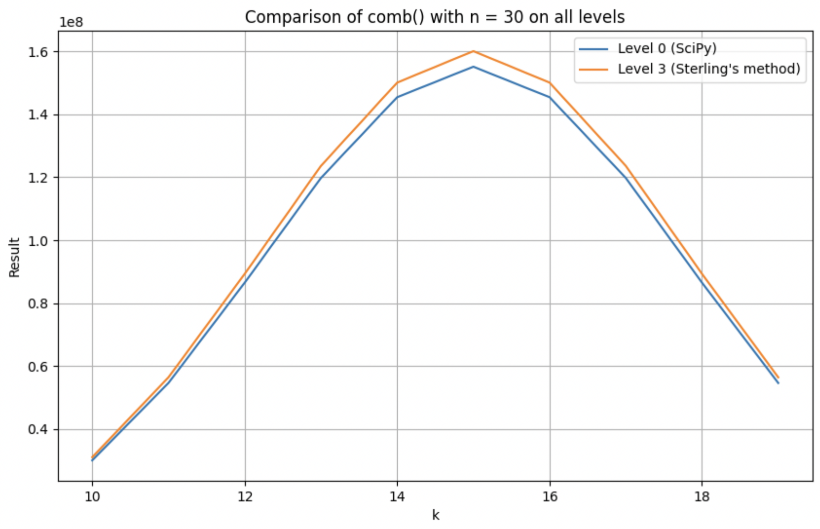

Figure 3: Comparison between levels 0 and 3 of the combination function. We can see that Stirling's approximation deviates the most as the function reaches its peak; there, it overshoots by approximately $2.5 * 10^6$.

perm()

The results for the permutation factor, seen in Figure 4 share similarities with the combination function. The lower bound diverges much quicker than the upper bound in the region $k < \frac{n}{2}$, making the latter more suitable for computations with that use case. Similarly to *comb*(), the lower bound tends towards the actual value. The lower bound is a doubly symmetrical (or 180° rotated) curve with respect to the SciPy variant.

As for the implementation of Sterling's method, it maintains accuracy to the actual value as well as the *comb*() approximation. However, that is only until the results reach $\sim 10^8$ (the maximum value from the *comb*() result). Afterwards, it diverges until reaching $k!$, where it intersects the lower bound approximation.

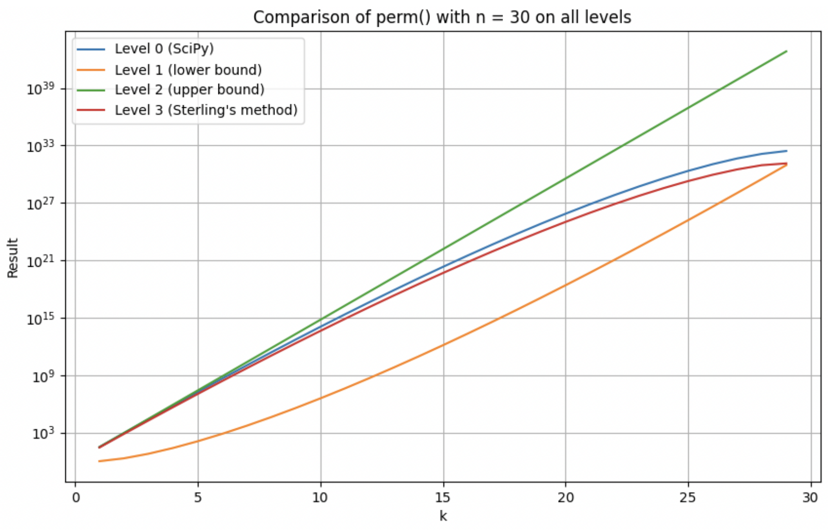

Figure 4: Comparison of the permutation function on all levels of approximation. We can observe Stirling's method (level 3) diverges from the accurate SciPy function rather greatly, as opposed to the combination function.

5.2 Energy validation

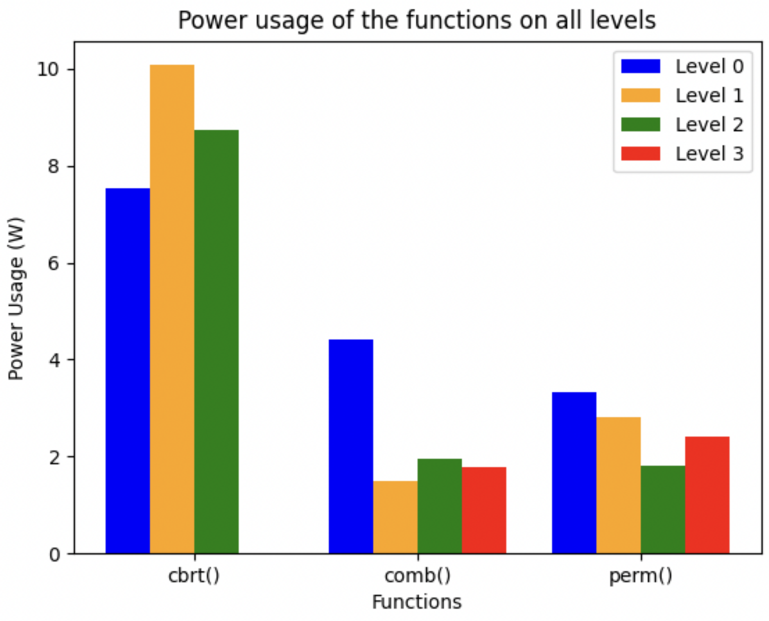

Figure 5: The power usages of all functions on all their levels. The *cbrt*() function was on $n = range(1, 10, 000)$. *comb*() and *perm* were both run on $n = range(1, 175)$ and $k = range(1, n + 1)$. The plot consists of an average of 50 such runs.

Our solution to the identified problem in Section 1 states that the approximated functions should be more energy efficient than their SciPy counterparts. To test this we have used the tool EnergiBridge[4] to compare the consumption of each level for each function.

For all functions, we have setup scripts to run them over multiple ranges and asses the execution times and the energy consumption, used to calculate the power consumption. $cbrt()$ was run with input ranging from 1 to 10,000, while both $comb()$ and $perm()$ were run on n ranging from 1 to 175 and k ranging from 1 to $n+1$. Each run was repeated 50 times and their results were averaged in Figure 5.

The results for the cube root function are unexpected. SciPy proves to be more energy efficient than both of our approximation functions. It seems that iterative operations, such as the ones in Halley's and Newton's methods are more demanding on the system than the implementation which SciPy uses.

On the other hand, the combination approximations show a much better energy efficiency than the original version. As expected the lower bound is the most efficient as it is the most simple, not using any factorials. These results imply that by far the largest contributing factor to these functions' energy consumption is the factorial. It would be interesting to delve deeper into further approximations for $n!$.

Lastly, permutation approximations also display a better efficiency than the SciPy counterpart, for many of the same reasons as the combination approximations. An unexpected result is that the efficiency of the approximations is not as good as for $comb()$; especially Sterling's method, which is the same except one less $k!$ approximation. This most likely indicates an error in test setup or execution of either the $comb()$ or the $perm()$ tests (likely $comb()$), and is an obvious limitation.

6 Discussion

We have discovered that approximate computing can result in energy-efficient alternatives for specific use cases. Although the cube root function approximation proved to consume more energy than the default, more accurate variant, combinations and permutations can be approximated efficiently.

For both combinations and permutations, upper-bound approximations tend to be more accurate when k is smaller. Conversely, as k reaches n, lower bound approximations can be a useful tool for getting moderately accurate results. Furthermore, approximating the factorial using Sterling's method showed the most promise for a wider range of applications, as it was relatively accurate and significantly more energy efficient than the SciPy defaults.

It is important to note that even though our permutation approximations, for example, only reduced power usage by 1W, and most uses of this computation only last a few fractions of a second, the total energy usage can amount to a very large number in the long term and when considering the number of computations happening. In the technology industry, every small percentile improvement amounts to a very large outcome given the sheer size of the industry.

7 Limitations and future work

The largest limitation of the project, we believe, is the lack of testing in a real-world environment such as the ones described in Section 1. This is a limitation due mostly to time and the complexity of projects that use these functions.

Another limitation is the fact that we could only conduct tests on the combination and permutation functions up to $n = 175$. This is due to limits in the Python implementation of Sterling's method which uses 32-bit numbers.

Lastly, Pyplot introduced a limitation for the visualisation of the combination and permutation results. Initially, we intended to visualise the results on 3-dimensional mesh grids of the functions, such that the influence of both the n and k parameters can be observed. However, Pyplot does not work with logarithmic scaling in 3d graphs and the results would be much harder to interpret. Therefore, we compromised by only showcasing the results for $n = 30$ (although more values of n were tested).

For future work, we would be interested in exploring better approximations of the cube root function, to hopefully improve upon the efficiency. Additionally, the library can be extended beyond its proof-of-concept intention to include more functions from SciPy (and other computing libraries).

[4]https://github.com/tdurieux/energibridge

8 Conclusion

To conclude, the results have demonstrated that approximate computing can be used for mathematical computation to reduce energy consumption when certain levels of accuracy are suitable. The provided implementations of the SciPy functions act as a proof of concept for what is a much larger endeavour: to educate the technology industry on the impacts of computation on their energy consumption, and in turn the environment; and to enable programmers to be aware of and choose their required level of computational complexity. We hope that more developers implement the paradigm of approximate computing within their libraries and projects going forward.

References

[Farhi, 2007] Farhi, B. (2007). Nontrivial lower bounds for the least common multiple of some finite sequences of integers. *Journal of Number Theory*, 125(2):393–411.

[Robbins, 1955] Robbins, H. (1955). A remark on stirling's formula. *The American Mathematical Monthly*, 62(1):26–29.