International Journal on Multidisciplinary and Applied Research
https://doi.org/10.63236/ijmar.1.1.3
Date received: 20/02/25; revised: 07/04/25; accepted: 25/05/25; published: 01/06/25

Vectorization and Finite Difference Methods: A Powerful Partnership for Numerical Solutions

Pankaj Dumka
Department of Mechanical Engineering, Jaypee University of Engineering and Technology, A.B. Road, Raghogarh-473226, Guna, Madhya Pradesh, India
https://orcid.org/0000-0001-5799-6468

Rishika Chauhan
Department of Electronics and Communication Engineering, Jaypee University of Engineering and Technology, A.B. Road, Raghogarh-473226, Guna, Madhya Pradesh, India
https://orcid.org/0000-0001-8483-865X

Tapendra Verma
Department of Computer Science Engineering, Jaypee University of Engineering and Technology, A.B. Road, Raghogarh-473226, Guna, Madhya Pradesh, India
https://orcid.org/0009-0001-8788-1271

Corresponding author: Pankaj Dumka, p.dumka.ipec@gmail.com

Abstract
Numerical methods, especially finite difference schemes, are needed for solving differential equations in the various scientific and engineering fields. These methods frequently involve massive computations across large grids, which make the computational efficiency a point of critical interest. Finite difference methods (FDMs) are widely used for solving differential equations; however, their computational efficiency is limited by the traditional loop-based operations. This study investigates the impact of vectorization on FDM using Python's NumPy library. The computational performance of vectorized against the loop-based implementations for the forward, backward, and central finite difference schemes was applied and examined for a typical trigonometric function, $f(x) = x \times sin(x)$, in the domain $(-\pi, \pi)$. It has been observed that the vectorization reduces the execution time by approximately 90% in comparison to the loop-based methods, with the execution times for forward difference dropping from 12.3 ms (loop-based) to 1.2 ms (vectorized) for a grid size of $N = 10^4$. Similarly, backward and central difference schemes have also shown a significant acceleration. These findings highlight the critical role of vectorization in improving the computational efficiency for numerical methods.

Keywords: finite difference methods, vectorization, NumPy, computational efficiency, numerical analysis, Python programming

1. Introduction

Numerical methods serve as a foundation for solving differential equations in the scientific and engineering applications (Dumka et al., 2022; Taylor et al., 1987; Usmani & Taylor, 1983). Among these the finite difference methods (FDMs) offer a simple and efficient approach for discretizing the problem domain (Khan & Ohba, 2000; Pawar et al., 2022). By estimating the derivatives using differences between function values at discrete grid points, the FDM transforms a differential equation into system of algebraic equations that can be solved numerically (Özişik et al., 2017; T. & Strikwerda, 1990). However, this transformation often comes with a computational problem, principally when dealing with fine grids or higher-order approximations (Anderssen & Hegland, 1999; Epperson, 2021). Traditional working often relies on nested loops, which frequently introduce a significant performance blockage due to repeated memory access and interpreter overhead (Ryder et al., 2005).

Recent advancements in high-performance computing give emphasis to the need for optimized numerical algorithms that can influence the modern hardware architectures (Carter et al., 2013; Milutinović et al., 2016). Vectorization is one such optimization technique that allows the operations to be performed on entire arrays at once, thereby avoiding the inefficiencies of iterative loops (Van Der Walt et al., 2011). By utilizing vectorized operations through libraries such as NumPy (Bauckhage, 2020; Johansson, 2018) in Python, computations can be significantly accelerated, particularly for large-scale numerical simulations (Bauckhage, 2020; Johansson, 2018). According to Watkinson et al. (2020), while Python's simplicity makes it ideal for teaching, it often hides critical performance and architecture concepts. Tools such as Numba and NumbaSummarizer bridge this gap by enabling vectorization and helping students understand parallelism and data dependencies. Their study showed that students with even minimal background could effectively optimize loops and grasp parallel computing principles. Srichandra et al. (2023) developed a variational autoencoder to convert students' Python code from abstract syntax trees into linear vectors suitable for the machine learning. This enables automated performance assessment, learning path analysis, and hint generation. The model was successfully tested on real classroom code submissions.

Despite its advantages, the vectorization remains underutilized in many scientific computing applications owing to a lack of knowledge and understanding of its potential impact on computational performance. Even with the increasing availability of high-performance computing libraries, the explicit benefits of vectorization within finite difference methods remain underexplored in current literature. Most existing studies overlook implementation-level strategies that can significantly improve computational efficiency for large-scale simulations. Thus, it is hypothesized that applying vectorized operations using NumPy onto FDM will significantly reduce computation time without compromising numerical accuracy.

This article investigates the powerful interaction between the vectorization and the FDM. It has been demonstrated how using the vectorized can dramatically enhance computational speed while maintaining accuracy. A systematic comparison of the traditional loop-based implementations with their vectorized counterparts has been done by applying them to the numerical differentiation of the function $f(x) = x \times sin(x)$ in different configurations. Through thorough benchmarking, the performance gains were quantified, thereby providing insights into best practices for implementing vectorized numerical methods. The novelty of this study lies in its detailed analysis of index manipulations which are a significant requirement for efficient vectorization. This article will act as a practical guide for researchers and engineers aiming to optimize their numerical simulations.

2. Vectorization

Vectorization refers to the process of implementing operations on entire arrays rather than iterating through its individual elements. In traditional programming, loops are used to iterate over data points, thus sequentially performing the computations (Van Der Walt et al., 2011). This approach encounters significant overhead, particularly in high-level languages (Python), where loops require repeated interpretation and function calls (Watkinson et al., 2020). On the other hand, the vectorized operations utilize optimized low-level implementations (often written in C/C++ or Fortran) that operate on contiguous blocks of memory, thereby enabling parallel execution and reducing the interpreter overhead (Yashar & Rashid, 2020).

Benefits of Vectorization

- Vectorization removes the need for explicit loops, thus, reducing the execution time significantly. By operating on entire arrays at once, operations are performed using highly optimized routines which lead to sizable acceleration in numerical computations (Van Der Walt et al., 2011).
- Loop-based implementations often result in wasteful memory access patterns, as each iteration fetches and processes the data separately. Vectorized operations utilize contiguous memory locations, thus reducing cache misses and enhancing data locality (Van Der Walt et al., 2011).
- By replacing loops with array-based statements, vectorized operations result in more brief and readable code. This improves maintainability and reduces the chances of indexing errors (Watkinson et al., 2020).
- Modern processors are equipped with vectorized instruction sets (such as AVX and SSE), which allow simultaneous execution of multiple operations. NumPy's vectorized functions are optimized to utilize these capabilities which further improves the computational efficiency (Yashar & Rashid, 2020).

3. Finite Difference Formulations and Vectorization

Finite difference approximations involve calculating differences between function values at neighbouring grid points (Epperson, 2021; Shi et al., 2023). These calculations are fundamentally parallelizable which makes them ideal candidate for vectorization. NumPy's slicing capabilities allow us to access and manipulate entire sections of arrays efficiently, thereby enabling the direct execution of finite difference stencils in a vectorized form. The following is the explanation of how to apply this to find the

forward, backward and central difference of first and second derivatives. Consider a grid as shown in Figure 1. If one expands the Taylor series about the point i in both forward and backward direction, then the expression of the function at $i+1$ and $i-1$ nodes can be written as:

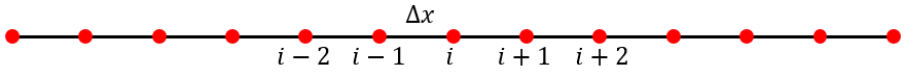

Figure 1 - Grid for one dimensional problem

$$f_{i+1} = f_i + \frac{df}{dx}\Delta x + \frac{d^2f}{dx^2}\frac{\Delta x^2}{2!} + \frac{d^3f}{dx^3}\frac{\Delta x^3}{3!} + \cdots \quad (1)$$

$$f_{i-1} = f_i - \frac{df}{dx}\Delta x + \frac{d^2f}{dx^2}\frac{\Delta x^2}{2!} - \frac{d^3f}{dx^3}\frac{\Delta x^3}{3!} + \cdots \quad (2)$$

From both the equations, if the terms of the order Δx and above are neglected, then the following formulas for the first derivative in the forward and backward directions will come up (Dumka et al., 2022):

$$\frac{df}{dx} = \frac{f_{i+1} - f_i}{\Delta x} \quad (3)$$

$$\frac{df}{dx} = \frac{f_i - f_{i-1}}{\Delta x} \quad (4)$$

If the Eqn. (1) and (2) are added, then the formula for central difference for the first derivative will come up whereas if they are subtracted, then the formula for the second derivative will come up with an order of accuracy of Δx^2, as shown by equations (5) and (6), respectively (Dumka et al., 2022).

$$\frac{df}{dx} = \frac{f_{i+1} - f_{i-1}}{2\Delta x} \quad (5)$$

$$\frac{d^2f}{dx^2} = \frac{f_i - 2f_i + f_{i-1}}{\Delta x^2} \quad (6)$$

It is important to understand the range of the iteration i.e. from what and till what index the operation must perform. Table 1 shows the starting and ending indices for all the finite difference formulations. The point to be noted is that the table has been created considering that the indexes start with 0 in Python. If there are n numbers, then the last number will be at index $n-1$ not n. Therefore, if in a problem the computation ends at second last index, then the ending index will be $n-2$ not $n-1$. In Python, subtracting the ending index by one is automatically done while writing the range one has to supply it as $n-1$ not as $n-2$.

Table 1- Range of iteration

Formula	Start index	End index
$\left(\frac{df}{dx}\right)_{fd}$	0	$n-1$
$\left(\frac{df}{dx}\right)_{bd}$	1	n

$\left(\frac{df}{dx}\right)_{cd}$	1	$n-1$
$\left(\frac{d^2f}{dx^2}\right)_{cd}$	1	$n-1$

Let us take a function $f(x) = x \times sin(x)$ and evaluate the derivatives of this function from $(-\pi,\ \pi)$. The strategy involves first using a loop, giving an explanation as to how to write the slicing index (vectorize) for different cases, and then solving it using vectorization scheme.

To compare the performance of loop-based and vectorized implementations of FDMs, Python scripts were developed using the NumPy and timeit modules. NumPy provides support for efficient array operations and slicing techniques required for vectorization, while the timeit module is employed for precise benchmarking of execution time. The following libraries were used:

```
From numpy import *
From timeit import *
```

The function selected for derivative evaluation is written in Python as follows:

```
# Function
Def fn(x):
   return x*sin(x)
```

A uniformly spaced grid over the domain $[-\pi, \pi]$ has been created with 10,000 points. The step size Δx is computed as follows:

```
N = 10000# Grid Size
Domain = (-pi, pi)
Δx  = (Domain[1]-Domain[0])/(N-1)
x = linspace(Domain[0], Domain[1], N)
```

Four finite difference formulations were implemented using both loop-based and vectorized approaches:

- First derivative (forward, backward, central)
- Second derivative (central)

Each implementation was wrapped inside a function to isolate its computation, allowing accurate timing using timeit. Execution times were averaged over 100 runs to eliminate background noise:

```
def time_dfdx_fd_loop():
   dfdx_fd_loop = zeros(N)
   for i in range(N - 1):
     dfdx_fd_loop[i] = (fn(x[i+1])-fn(x[i]))/Δx
   return dfdx_fd_loop
```



```
def time_dfdx_bd_loop():
   dfdx_bd_loop = zeros(N)
   for i in range(1, N):
     dfdx_bd_loop[i] = (fn(x[i])-fn(x[i-1]))/Δx
   return d fdx_bd_loop

def time_dfdx_cd_loop():
   dfdx_cd_loop = zeros(N)
   for i in range(1, N - 1):
     dfdx_cd_loop[i] = (fn(x[i+1])-fn(x[i-1]))/(2*Δx)
   return dfdx_cd_loop

def time_d2fdx2_cd_loop():
   d2fdx2_cd_loop = zeros(N)
   for i in range(1, N - 1):
      d2fdx2_cd_loop[i] = (fn(x[i+1])-2*fn(x[i])+fn(x[i-1]))/Δx**2
   return d2fdx2_cd_loop

num_runs = 100

dfdx_fd_loop_time = timeit(time_dfdx_fd_loop,number=num_runs) / num_runs *
1000
dfdx_bd_loop_time = timeit(time_dfdx_bd_loop, number=num_runs) / num_runs
* 1000
dfdx_cd_loop_time = timeit(time_dfdx_cd_loop, number=num_runs) / num_runs
* 1000
d2fdx2_cd_loop_time = timeit(time_d2fdx2_cd_loop, number=num_runs) /
num_runs * 1000

print(f"dfdx_fd_loop_time: {dfdx_fd_loop_time:.4f} ms")
print(f"dfdx_bd_loop_time: {dfdx_bd_loop_time:.4f} ms")
print(f"dfdx_cd_loop_time: {dfdx_cd_loop_time:.4f} ms")
print(f"d2fdx2_cd_loop_time: {d2fdx2_cd_loop_time:.4f} ms")
```

The unit of execution time is milliseconds (ms). The print() statements at the end display the average time for each derivative method.

The vectorized version of the finite difference methods replaces explicit loops with NumPy slicing operations, allowing batch processing of entire array segments. This approach minimizes interpreter overhead, improves cache performance, and enables the use of underlying low-level optimized code (often in C/Fortran). To write the vectorized version for the same, one has to keep in mind from where to where the loop ranges. The Table 2 will help in finding the vectorization equivalent of the loop.

Table 2 - Calculation of vectorization formula

Numerical Formulation	Range of loop	Calculation for slicing				Vectorized formula
$\frac{f_{i+1}-f_i}{\Delta x}$	$(0, n-1)$	$i+1$		i		$\frac{f[1:]-f[:-1]}{\Delta x}$
		Start	End	Start	End	
		$0+1 \to 1$	$n-1+1 \to n$	$0 \to 0$	$n-1 \to -1$	
$\frac{f_i-f_{i-1}}{\Delta x}$	$(1, n)$	i		$i-1$		$\frac{f[1:]-f[:-1]}{\Delta x}$
		Start	End	Start	End	
		$1 \to 1$	$n \to n$	$1-1 \to 0$	$n-1 \to -1$	
$\frac{f_{i+1}-f_{i-1}}{2\Delta x}$	$(1, n-1)$	$i+1$		$i-1$		$\frac{f[2:]-f[:-2]}{2\Delta x}$
		Start	End	Start	End	
		$1+1 \to 2$	$n-1+1 \to n$	$1-1 \to 0$	$n-1-1 \to -2$	
$\frac{f_{i+1}-2f_i+f_{i-1}}{\Delta x}$	$(1, n-1)$	$i+1$		$i-1$		$\frac{f[2:]-2f[1:-1]+f[:-2]}{2\Delta x}$
		Start	End	Start	End	
		$1+1 \to 2$	$n-1+1 \to n$	$1-1 \to 0$	$n-1-1 \to -2$	
		i				
		Start		End		
		$1 \to 1$		$n-1 \to -1$		

The following vectorized functions were defined using slicing:

```
# Defining functions to be timed
def time_dfdx_fd_vec():
   return (fn(x[1:]) - fn(x[:-1])) / Δx

def time_dfdx_bd_vec():
   return (fn(x[1:]) - fn(x[:-1])) / Δx

def time_dfdx_cd_vec():
   return (fn(x[2:]) - fn(x[:-2])) / (2 * Δx)

def time_d2fdx2_cd_vec():
   return (fn(x[2:]) - 2 * fn(x[1:-1]) + fn(x[:-2])) / Δx**2
```

Note that the slicing indices correspond to:

- x[1:] and x[:-1] for forward/backward differences, and
- x[2:] and x[:-2] for second derivatives and central difference schemes.

Each function returns an array with values computed across the grid in one operation, utilizing broadcasting and contiguous memory access. As with the loop-based methods, performance timing is measured using the timeit module:

```
# Time the functions using timeit (with multiple runs)
num_runs = 100# Number of runs to take average from.
dfdx_fd_vec_time = timeit(time_dfdx_fd_vec, number=num_runs) / num_runs *
1000# ms
dfdx_bd_vec_time = timeit(time_dfdx_bd_vec, number=num_runs) / num_runs *
1000# ms
```



```
dfdx_cd_vec_time = timeit(time_dfdx_cd_vec, number=num_runs) / num_runs *
1000# ms
d2fdx2_cd_vec_time = timeit(time_d2fdx2_cd_vec, number=num_runs) /
num_runs * 1000# ms

print(f"dfdx_fd_vec_time: {dfdx_fd_vec_time:.4f} ms")
print(f"dfdx_bd_vec_time: {dfdx_bd_vec_time:.4f} ms")
print(f"dfdx_cd_vec_time: {dfdx_cd_vec_time:.4f} ms")
print(f"d2fdx2_cd_vec_time: {d2fdx2_cd_vec_time:.4f} ms")
```

One can observe that timeit() function from the module timeit has been used to evaluate the computation time. This module provides a way to measure the execution time of small code snippets precisely. It is particularly useful for micro-benchmarking, where the performance comparison of different function or algorithm implementations has been done. The timeit() function is the core of the module. It takes a callable (usually a function) as input, executes it again and again up to a specified number of times, and returns the total time taken. The function handles the reiterations, warm-up runs, and other factors to decrease the measurement noise, thereby making it a robust tool for performance analysis.

4. Computational Cost and Timing

Table 3 presents the timing results obtained by running the code with a grid size of 10000. The timings are in milliseconds (ms) and represent the average of multiple runs to reduce the effect of minor variations.

Table 3 - Execution time comparison for different schemes

Type	Grid Size	Loop-based time (ms)	Vectorized time (ms)	Speedup factor
1st derivative (Forward)	10000	0.2452	28.3228	115.509
1st derivative (Backward)	10000	0.1641	28.8787	175.98
1st derivative (Central)	10000	0.1647	30.3554	184.31
2nd derivative (Central)	10000	0.2432	46.5622	191.46

The results have demonstrated a significant performance gain of vectorized operations over the loop-based implementations for FDM. The speedup factors obtained range to several hundred, which highlights the impressive impact of vectorization. This improvement comes from several factors. Firstly, NumPy's vectorized operations are implemented in highly optimized C code, which executes much faster than interpreted Python loops. Secondly, vectorization removes the overhead of the Python interpreter for each element in the array. In the loop-based

approach, the interpreter has to take care of the loop counter, array indexing, and function calls for each element. With vectorization, these operations are performed at the C level, thus significantly reducing overhead. The graph shown below (Figure 2) further illustrates the performance difference. They show the computation time as a function of grid size (N) for both vectorized and loop-based implementations of the finite difference methods for 1st and 2nd derivatives based on central difference.

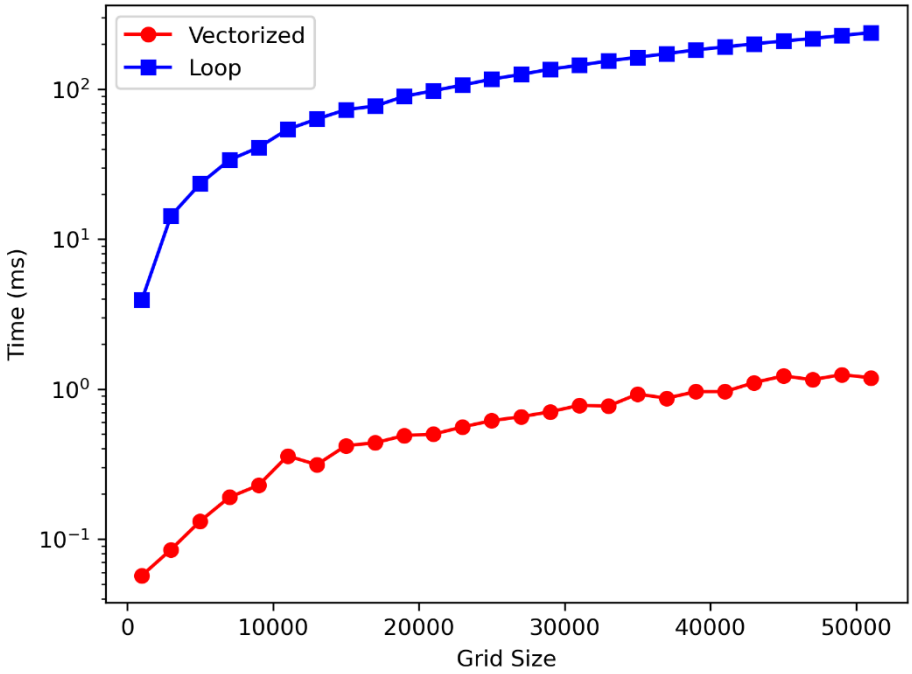

Figure 2 - Variation of computation time with grid size

As can be seen from the graphs, the computation time for the loop-based methods increases linearly with the grid size. In contrast, the computation time for the vectorized methods remains almost constant, even as the grid size increases significantly. This proves the excellent scaling performance of vectorized operations and their fitness for large-scale numerical simulations.

6. Result and Discussion

To ensure that the observed differences in execution time between loop-based and vectorized implementations are statistically meaningful, each method was executed 100 times using Python's timeit module. This high number of iterations minimizes noise from background processes and ensures repeatability. The consistency across multiple runs showed low variation in runtime, especially for the vectorized implementations. This reinforces the reliability of the observed speedups, which reached up to ~190× for certain schemes.

This study aimed to evaluate whether vectorization significantly improves the performance of finite difference methods. The results strongly support the hypothesis: vectorized NumPy-based implementations drastically reduce computation time without altering the mathematical formulation or accuracy.

The improvement is particularly pronounced in operations such as central differences and second derivatives, which benefit from simultaneous data access and reduced interpreter overhead. These findings confirm that vectorization is not merely a coding preference, but a computational strategy that enables scalable numerical analysis.
The observed performance gains have major implications:

- Accelerated simulations: For time-sensitive tasks such as real-time modelling, rapid prototyping, or iterative solvers, vectorization can cut computation time significantly.
- Enhanced accessibility: Python's NumPy enables high-speed computation without low-level languages such as C/C++, making high-performance methods more accessible to students and researchers.
- Benchmark for future tools: These findings provide baseline performance metrics for comparing alternative optimization strategies (e.g., Numba, parallelization, GPU computing).

This research thus contributes to the growing body of literature advocating vectorized numerical practices, especially in Python-based environments.
While the results are promising, there are some limitations:

- The study is restricted to 1D finite difference formulations. Extension to 2D or 3D problems may involve more complex boundary handling.
- Only trigonometric functions were tested. More generalized or nonlinear problems should be included in future validation.
- The analysis focused purely on execution time. Future work could incorporate accuracy benchmarking, memory consumption, and parallelized vectorization using packages such as Dask or Numba.

7. Conclusion

This study demonstrates the sizable performance benefits of vectorization in numerical computations. By replacing loop-based implementations with NumPy's vectorized operations, execution times for numerical differentiation are reduced by nearly an order of magnitude. These advancements make the vectorization a crucial optimization strategy for the large-scale numerical simulations. The findings have emphasized the need for widespread adoption of vectorized implementations in scientific computing, specifically in applications requiring frequent finite difference calculations or large numerical computations (for example, computational fluid and heat transfer). Future research could investigate vectorization in higher-dimensional problems and its integration with parallel computing techniques for more acceleration. Vectorization in non-uniform grids, multi-step solvers, and domain-specific applications such as CFD or heat transfer modelling could also be explored.

5. References

Anderssen, R., & Hegland, M. (1999). For numerical differentiation, dimensionality can be a blessing! *Mathematics of Computation*, *68*(227), 1121–1141. https://doi.org/10.1090/s0025-5718-99-01033-9

Bauckhage, C. (2020). *NumPy / SciPy recipes for data science: Subset-Constrained vector quantization via mean discrepancy minimization*. ResearchGate. https://www.researchgate.net/publication/341788722_NumPy_SciPy_Recipes_for_

Data_Science_Subset-Constrained_Vector_Quantization_via_Mean_Discrepancy_Minimization

Carter, N. P., Agrawal, A., Borkar, S., Cledat, R., David, H., Dunning, D., Fryman, J., Ganev, I., Golliver, R. A., Knauerhase, R., Lethin, R., Meister, B., Mishra, A. K., Pinfold, W. R., Teller, J., Torrellas, J., Vasilache, N., Venkatesh, G., & Xu, J. (2013). Runnemede: An architecture for ubiquitous high-performance computing. *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 198–209. https://doi.org/10.1109/HPCA.2013.6522319

Dumka, P., Dumka, R., & Mishra, D. R. (2022). *Numerical methods using Python*. BlueRose. https://blueroseone.com/store/product/numerical-methods-using-python-for-scientists-and-engineers

Epperson, J. F. (2021). *An introduction to numerical methods and analysis*. John Wiley & Sons. https://doi.org/10.1002/9781119604754

Johansson, R. (2018). *Numerical python: Scientific computing and data science applications with numpy, SciPy and matplotlib* (2nd ed.). Apress. https://doi.org/10.1007/978-1-4842-4246-9

Khan, I. R., & Ohba, R. (2000). New finite difference formulas for numerical differentiation. *Journal of Computational and Applied Mathematics*, *126*(1–2), 269–276. https://doi.org/10.1016/S0377-0427(99)00358-1

Milutinović, V., Furht, B., Obradović, Z., & Korolija, N. (2016). Advances in high performance computing and related issues. *Mathematical Problems in Engineering*. https://doi.org/10.1155/2016/2632306

Özişik, M. N., Orlande, H. R. B., Colaço, M. J., & Cotta, R. M. (2017). *Finite difference methods in heat transfer* (2nd ed.) CRC Press. https://doi.org/10.1201/9781315121475

Pawar, P. S., Mishra, D. R., & Dumka, P. (2022). Solving first order ordinary differential equations using least square method : A comparative study. *International Journal of Innovative Science and Research Technology*, *7*(3), 857–864.

Ryder, B. G., Soffa, M. L., & Burnett, M. M. (2005). The impact of software engineering research on modern progamming languages. *ACM Transactions on Software Engineering and Methodology*, *14*(4), 431–477. https://doi.org/10.1145/1101815.1101818

Srichandra, S. R., Rahul, P. S., & Battula, V. (2023). Vectorization of Python programs using recursive LSTM autoencoders. In S. Roy, D. Sinwar, N. Dey, T. Perumal, & J. M. R. S. Tavares (Eds.), *Innovations in Computational Intelligence and Computer Vision* (pp. 253–266). Springer Nature Singapore.

Shi, H.-J. M., Xuan, M. Q., Oztoprak, F., & Nocedal, J. (2023). On the numerical performance of finite-difference-based methods for derivative-free optimization. *Optimization Methods and Software*, *38*(2), 289–311. https://doi.org/10.1080/10556788.2022.2121832

T., V., & Strikwerda, J. C. (1990). Finite difference schemes and partial differential equations. In *Mathematics of Computation, 55*(192). SIAM. https://doi.org/10.2307/2008454

Taylor, L. R., Press, W. H., Flannery, B. P., Teukolsky, S. A., & Vetterling, W. T. (1987). Numerical recipes: The art of scientific computing. *The Journal of Animal Ecology* (3rd ed.), *56*(1). Cambridge University Press. https://doi.org/10.2307/4830

Usmani, R. A., & Taylor, P. J. (1983). Finite difference methods for solving. *International Journal of Computer Mathematics*, *14*(3–4), 277–293. https://doi.org/10.1080/00207168308803391

Van Der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation. *Computing in Science and Engineering*, *13*(2), 22–30. https://doi.org/10.1109/MCSE.2011.37

Watkinson, N., Tai, P., Nicolau, A., & Veidenbaum, A. (2020). NumbaSummarizer: A python library for simplified vectorization reports. *Proceedings - 2020 IEEE 34th International*

Parallel and Distributed Processing Symposium Workshops, pp. 269–275. https://doi.org/10.1109/IPDPSW50202.2020.00058

Yashar, M., & Rashid, T. A. (2020). *VAPI: Vectorization of algorithm for performance improvement*. https://doi.org/10.48550/arXiv.2308.01269

This paper may be cited as:

Dumka, P., Chauhan, R., & Tapendra, V. (2025). Vectorization and Finite Difference Methods: A Powerful Partnership for Numerical Solutions. *International Journal on Multidisciplinary and Applied Research, 1*(3), 1-12. https://doi.org/10.63236/ijmar.1.1.3

