Environmetrics

WILEY

RESEARCH ARTICLE OPEN ACCESS

Computational Benchmark Study in Spatio-Temporal Statistics With a Hands-On Guide to Optimise R

Lorenzo Tedesco[1] | Jacopo Rodeschini[2] | Philipp Otto[3]

[1]Department of Economics, University of Bergamo, Bergamo, Italy | [2]Department of Engineering and Applied Sciences, University of Bergamo, Dalmine, Italy | [3]School of Mathematics and Statistics, University of Glasgow, Glasgow, UK

Correspondence: Lorenzo Tedesco (lorenzo.tedesco@unibg.it)

Received: 24 December 2024 | **Revised:** 11 April 2025 | **Accepted:** 13 April 2025

Funding: This research was supported by the National Recovery and Resilience Plan (NRRP), Mission 4 Component 2 Investment 1.3, under Call for tender No. 341 of 15/03/2022, funded by the Italian Ministry of University and Research and the European Union's Next-Generation EU initiative. Award Number: PE 00000018, Concession Decree No. 1558 of 11/10/2022, CUP F83C22001720001, under the project titled Growing Resilient Inclusive and Sustainable (GRINS).

Keywords: benchmarking | computational performance | geospatial modeling | matrix operations | spatio-temporal analysis

ABSTRACT

This study provides a comprehensive evaluation of the computational performance of `R`, `MATLAB`, `Python`, and `Julia` for spatial and spatio-temporal modelling, focusing on high-dimensional datasets typical in geospatial statistical analysis. We benchmark each language across key tasks, including matrix manipulations and transformations, iterative programming routines, and Input/Output processes, all of which are critical in environmetrics. The results demonstrate that `MATLAB` excels in matrix-based computations, while `Julia` consistently delivers competitive performance across a wide range of tasks, establishing itself as a robust, open-source alternative. `Python`, when combined with libraries like `NumPy`, shows strength in specific numerical operations, offering versatility for general-purpose programming. `R`, despite its slower default performance in raw computations, proves to be highly adaptable; by linking to optimized libraries like `OpenBLAS` or `MKL` and integrating `C++` with packages like `Rcpp`, `R` achieves significant performance gains, becoming competitive with the other languages. This study also provides practical guidance for researchers to optimize `R` for geospatial data processing, offering insights to support the selection of the most suitable language for specific modelling requirements.

1 | Introduction

Spatio-temporal statistics has become essential for understanding processes that evolve over space and time, as many pressing scientific questions-ranging from climate change and biodiversity loss to epidemic forecasting and urban development-require models that account for both spatial and temporal variation (Cressie and Wikle 2011). Meeting these demands poses methodological challenges, including the need to model non-stationarity, complex dependencies, and potential causal mechanisms in dynamic systems (Gneiting 2002; Hufnagel et al. 2004). These challenges are further complicated by data characteristics such as missing values, measurement error, multifidelity, and high dimensionality. Recent methodological developments increasingly integrate modern machine-learning techniques to enhance predictive accuracy (Reichstein et al. 2019; Bonas et al. 2024) and extend the inferential capabilities of traditional statistical models (Otto et al. 2024).

In environmental science, we typically deal with univariate or multivariate data, leading to three main sources of dependence. First, due to the temporal proximity between observations,

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2025 The Author(s). *Environmetrics* published by John Wiley & Sons Ltd.

environmental processes are likely to be dependent across time. This temporal dependence is one-sided and causal, meaning that past observations influence future ones. Similarly, proximity in space induces spatial dependence, which is, by contrast, usually two-sided and contemporaneous. Observations at a given location are instantaneously influenced by their surroundings and vice-versa, with the strength of this influence typically diminishing with distance. Lastly, when analyzing multivariate processes, we usually face correlated features, a third source of dependence.

Accounting for these dependencies is particularly critical in the ÃćâĆňÅŞbig dataÃćâĆňÂİ era, where data sets such as those from the GOES-16 geostationary satellite capture high-resolution spatial and temporal details, recording radiance across 16 wavelength bands with spatial resolutions as fine as 1 km and temporal intervals as short as 1 minute. For example, such data has been utilized by Guinness (2022) to estimate stationary multivariate spatial and spatio-temporal spectra. Similarly, Datta et al. (2016) analyzed over 114,000 georeferenced forest biomass plots from the USDA FIA program, encountering memory constraints due to the large distance matrices required for spatial modelling. This challenge spurred the development of scalable methodologies to efficiently process and analyze large spatial data sets. For instance, Heaton et al. (2019) directly address the high-dimensional data issue in the spatial context, evaluating the limitations of traditional models and exploring novel approaches to improve the tractability and efficiency of handling high-dimensional spatial data sets.

Scalability plays a critical role in this context; see also Jordan (2013). For instance, one could distinguish between strong and weak scalability. The former is crucial when dealing with fixed-size datasets, where increasing the computational efficiency is expected to decrease the runtime. For instance, matrix operations such as transposition or exponentiation on a fixed-size matrix should ideally scale well across multiple processors, with each processor working on smaller portions of the matrix in parallel. Weak scalability becomes relevant when the problem size grows proportionally with the number of processors. For example, as larger spatial data sets or finer temporal resolutions are analyzed, the computational workload per processor remains constant. Operations like matrix creation and deformation, which involve reshaping or partitioning data for independent processing, are well-suited for weak scalability.

In spatio-temporal statistics, we can identify certain operations that are frequent and common across the models used. Matrix creation, transposition, deformation, and exponentiation facilitate reshaping spatial data, modelling temporal or spatial relationships, and analyzing higher-order dependencies. This approach is exemplified in Li et al. (2024), where higher-order spatial autoregressive models are effectively analyzed. Sorting large arrays is also a frequent pre-processing step necessary for different types of operations like spatial indexing (essential in spatial databases), nearest-neighbor searches, which is common in geostatistics (Altman 1992), optimising sparse matrix operations needed, for instance, for sparse covariance matrices (Hsieh et al. 2011), and organizing results for further analysis or visualization (e.g., mapping and trend analysis in environmetrics). Matrix operations such as cross-products, matrix inversion (e.g., for OLS estimation of linear regression models), eigenvalue decomposition and Cholesky decomposition are fundamental to statistical analysis, forming the backbone of many computational methods. Additionally, Fast Fourier Transforms (FFT) are vital for decomposing time series or spatial data into frequency components, see for instance Guinness (2019), aiding in the analysis of periodic or cyclic behaviors in spatio-temporal data sets. Likewise recursive algorithms play a significant role in spatial statistics, especially within MCMC and EM estimation procedures, with the former being widely utilised in Bayesian spatial statistics, as demonstrated by Otto et al. (2023), and the latter frequently employed in frequentist approaches, such as in Wang et al. (2021). Finally, robust Input/Output operations, as reading and writing large comma-separated value (CSV) files, are crucial for efficiently handling and processing high-dimensional data sets, ensuring seamless integration between data storage and analysis workflows.

Efficient computational methods are critical for spatio-temporal data analysis. Equally important, however, is the choice of a suitable software environment, as algorithm performance can vary widely based on the programming language and its implementation. This paper has two main goals. First, it offers a detailed performance comparison of various programming languages, focusing on execution times for matrix operations and programming tasks. Second, it provides practical advice for spatial statisticians who may be less familiar with technical aspects of R optimisation. By following these recommendations and leveraging the overview provided, researchers can enhance their programming environments, deepen their understanding of each tool's strengths, and make informed decisions to optimise both time efficiency and computational cost. This work builds upon and expands the analysis in Klamer (2024), broadening the comparison across multiple programming languages to reach a wider audience.

For that, we examine the most commonly used software environments in spatial statistics and organise the discussion as follows. Section 2 provides an overview of the selected programming languages, including their core features and relevance for spatial statistics and spatio-temporal data analysis. Section 3 presents the setup for our Monte Carlo comparison study, detailing the computational tasks, benchmarking methodology, performance metrics, and the comparison results, which highlight the computational efficiency of the languages across various tasks. Detailed specifications of the hardware used for benchmarking are also provided. Section 4 focuses on practical strategies for optimising R's performance, including leveraging optimised libraries and integrating C++, specifically within the Ubuntu environment on Linux systems. Finally, Section 5 concludes the paper with key insights and recommendations for researchers in spatial statistics.

2 | Overview of Selected Programming Languages

The main programming environments for spatio-temporal data analysis include R, MATLAB, Python, and Julia. Each offers distinct strengths in terms of statistical modelling, data

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

handling, and computational efficiency, particularly for spatial and spatio-temporal tasks, as summarized in Table 1.

Known for its extensive package ecosystem tailored to data analysis, R supports a vast array of statistical and graphical techniques and offers strong support for spatial and spatio-temporal analysis. R provides robust native support for matrix operations, seamlessly integrating high-level syntax with efficient performance. By default, R leverages optimized low-level linear algebra libraries (BLAS and LAPACK) to further enhance speed and scalability. The central role of R is also highlighted by prominent books on spatial statistics, such as Wikle et al. (2019); Bivand et al. (2013); Pebesma (2018), which present material directly through R implementations. A comprehensive list of R packages for analysis of spatial data is available at https://cran.r-project.org/web/views/Spatial.html, from which we underline the packages sp (Pebesma and Bivand 2005) and sf (Pebesma and Bivand 2023) which provide general classes and methods for handling and analysing spatial data, the package spdep/spatialreg (Bivand and Wong 2018; Bivand 2022) which focuses on spatial dependence for areal data and spatial econometrics, providing tools for analyzing spatial autocorrelation, constructing spatial weights, and implementing spatial regression models, the package gstat (Gräler et al. 2016; Pebesma 2004) which provides tools for geostatistical modelling, spatial interpolation (e.g., Kriging), and spatio-temporal prediction, supporting multivariable and variogram-based analyses; spmodel (Dumelle et al. 2023), which fits statistical models to geostatistical and areal data with support for various covariance structures, CARBayes (Lee 2013), which provides Bayesian hierarchical spatial models, and spBayes (Finley and Banerjee 2024; Finley et al. 2015; Finley et al. 2007), which enables Bayesian geostatistical modelling. The spatstat (Baddeley et al. 2015; Baddeley et al. 2013; Baddeley and Turner 2005) package offers extensive capabilities for spatial point pattern analysis, including model fitting, simulation, and statistical inference. For visualisation, packages, such as tmap (Tennekes 2018), leaflet (Cheng et al. 2019), and ggplot2 (Wickham 2016) provide advanced tools for creating static, interactive, and thematic maps, supporting a wide range of spatial data formats and visualization techniques. The central role of R in spatial statistics is also underlined by the fact that while many popular models are available across platforms, like those in Stan (Carpenter et al. 2017), certain others are exclusive to R, such as the popular INLA approach (Rue et al. 2009) and the Fixed Rank Kriging model FRK (Zammit-Mangion and Cressie 2021; Sainsbury-Dale et al. 2024). R supports parallel computing via built-in packages like parallel and third-party packages such as foreach (see also Schmidberger et al. 2009), enabling users to efficiently distribute computations across multiple cores or clusters.

Second, MATLAB is a high-level programming language and interactive environment optimized for numerical computing, visualization, and application development. Designed primarily for matrix and linear algebra operations, MATLAB is built upon efficient versions of the BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage) libraries, making it particularly popular for scientific and engineering applications. Its intuitive interface and a wide array of toolboxes have made it a staple in academia and research, though it requires a commercial license. In the context of spatial analysis, there are mainly three toolboxes available: (a) the Spatial Statistics Toolbox (Pace and Barry 2003) which provides efficient tools for spatial statistics, including SAR, CAR, and MRSA models, sparse spatial weight matrices, log-determinants for maximum likelihood estimation, and computational optimizations for large data sets; (b) the Spatial Econometrics Toolbox (LeSage and Pace 2009; LeSage 1998) which offers over 350 functions, including tools for maximum likelihood and Bayesian spatial econometrics, regression diagnostics, time-series modelling, and simulations, accompanied by extensive documentation and demonstrations; and (c) the Mapping Toolbox which provides tools for importing, processing, and visualizing geographic data, coordinate transformations, and support for formats like shapefile, GeoTIFF, and KML. A limited number of spatial statistical models have been uniquely implemented in MATLAB, with Wang et al. (2021) being a notable example. MATLAB provides parallel computing capabilities through the Parallel Computing Toolbox (The MathWorks Inc. 2024), which facilitates execution across multicore processors, GPUs, and clusters, significantly improving computational efficiency for intensive tasks.

Third, Python is a versatile and widely used programming language that provides powerful tools for spatial data analysis through its extensive library ecosystem. Python supports spatial and spatio-temporal analysis with libraries, such as NumPy (Harris et al. 2020), which enables efficient handling of large multidimensional arrays and matrices—by default, leveraging the highly optimized OpenBLAS backend for linear algebra operations—or matplotlib (Hunter 2007), which provides robust visualization capabilities, including the ability to create static and interactive map plots. Built on these, SciPy (Virtanen et al. 2020) adds advanced mathematical and scientific computation tools, while pandas (pandas development team 2020; McKinney 2010) facilitates the manipulation of tabular data, offering functionality similar to R's data frames. For geospatial analysis, Shapely (Gillies et al. 2007) provides planar geometry operations, and GeoPandas (Jordahl et al. 2020) extends pandas to handle geospatial vector data, including geoprocessing, spatial joins, and projection support. Moreover, PySAL (Rey and Anselin 2009) is a comprehensive library for spatial econometrics and spatial data science, offering tools for spatial regression, spatial autocorrelation, and clustering. GDAL/OGR (GDAL/OGR contributors 2024) enables Python users to work with raster and vector data formats from numerous Geographic Information System (GIS) platforms. It is worth noting that machine learning and deep-learning methods in environmental contexts is gaining popularity. Many related packages leverage the accessibility and ease of implementation provided by Python to develop tools within this framework (Wikle and Zammit-Mangion 2023). Notable examples include Chen et al. (2020); Zeroual et al. (2020). Python offers extensive parallel computing support through libraries such as multiprocessing (McKerns et al. 2012), which efficiently distribute workloads over multiple CPUs or clusters, optimizing performance for large-scale computations.

Fourth, Julia is a high-level, high-performance programming language designed for technical computing, renowned for its speed and user-friendliness in scientific applications. Combining the ease of use of high-level languages with the performance of low-level languages like C and Fortran, Julia is particularly well-suited for linear algebra and matrix computations, utilizing

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 1 | Comparative summary of `R`, `MATLAB`, `Python`, and `Julia` for spatial-statistical workflows, including their general focus, cost, community size, available spatial packages, parallel computing features, user-friendliness, numeric speed, and visualization support.

Aspect	R	MATLAB	Python	Julia
General Overview	Statistical computing environment designed specifically for data analysis, visualisation, and academic research.	Numerical computing environment widely used in engineering and academia for simulations and algorithm development.	Versatile general-purpose programming environment popular for statistics, data science, and machine learning.	High-performance language optimized for numerical and scientific computing with a focus on speed.
License	Open-source and free.	Commercial (expensive if no institutional access).	Open-source and free.	Open-source and free.
Community Support	Large community with a wealth of spatial statistical packages and long-term academic usage.	Large academic community with significant support in engineering.	Extensive community, a large number of libraries (112k+ packages).	Small but rapidly growing community with active contributions.
Spatial Statistics Libraries	Extensive libraries for manipulation of spatial data, as `sp`, `sf`, `spatstat`, several popular statistical methodologies as `spatpat` and `spmodel` and exclusive framework for popular model as `FRK` and `INLA`.	Limited to some toolbox, as `Spatial Econometrics Toolbox`, `Spatial Statistics Toolbox`, `Mapping Toolbox` and few contributions such as `D-STEM`.	Rich ecosystem for spatial analysis, such as `GeoPandas`, `Shapely`, `Scipy` and popular packages for fast numerical computing as `NumPy` and `pandas`. Several popular methods in spatial statistics are missing.	No mature native libraries, but performance advantages for custom implementations.
Parallelization	Possible using popular libraries such as `parallel` and `foreach`.	Native support using `parfor`.	Possible using popular libraries such as `multiprocessing`.	Strong built-in support for parallelization (shared and distributed) with `Threads`.
Ease of Use	Generally user-friendly, but occasionally outdated and unintuitive.	User-friendly, especially with built-in routines and plotting.	Designed to be concise and easy to read, making it more accessible to first-time coders.	Similar to `MATLAB` syntax and in-situ/ex-situ matrix operations can easily be handled.
Performance	Generally slow due to its interpreted nature, but supports integration with `C` and several optimization packages are available for performance enhancement.	Slower in certain operations due to its interpreted nature, but excellent in matrix operations.	Generally slow due to its interpreted nature, but several optimization packages are available for performance enhancement.	Fastest for numerical operations, similar to `C` in speed.
Visualisation	Excellent plotting capabilities using popular packages such as `ggplot2`.	Excellent built-in plotting methods.	Several visualization libraries are competitive with other frameworks, such as `matplotlib`.	Built-in plotting capabilities with `Plots.jl`, but less mature than other languages.

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

LAPACK and BLAS libraries. Its just-in-time (JIT) compilation ensures efficient execution of complex computations, making it ideal for high-performance tasks in scientific and technical fields. In the realm of spatial analysis, libraries like GeoStats.jl (Hoffimann 2018) provide tools for geospatial data science and geostatistical modelling, though the ecosystem is still developing. Books and resources focusing on spatial analysis with Julia have recently started to emerge, such as Hoffimann (2018), highlighting the language's growing potential in this domain. Julia has built-in parallel computing functionality that allows easy scaling from multicore processors to distributed computing environments, providing efficient performance through simple and intuitive syntax.

Choosing an environment depends on various factors, including ethical and technical considerations, such as a commitment to open-source software and available packages. However, user familiarity with specific environments, particularly R and Python for statistics, and the performance capabilities of these languages are primary determinants. In practice, many researchers may not realize that properly configuring their software environment can considerably impact package performance. For instance, while R is generally considered slower than MATLAB or Julia, optimizing R's settings can yield comparable performance. This knowledge could benefit spatial statisticians, who predominantly use the R environment.

Although several performance comparisons are available in the literature, such as in Coleman et al. (2021), a comprehensive analysis that guides spatial statisticians in choosing a language is lacking.

3 | Monte-Carlo Comparison Study

3.1 | Setup and Choice of Operations

The benchmarking suite is structured around a comprehensive set of tasks organised into four clearly defined categories: **Matrix Manipulations**, **Matrix Transformations**, **Programming Tasks**, and **I/O Operations**. These categories are designed to capture distinct aspects of computational performance, from basic and advanced matrix operations to algorithmic tasks and file input/output handling. Each category includes several tasks that collectively aim to evaluate performance within their respective computational domains. Due to inconsistencies in how different programming languages allocate and manage memory, we focus our analysis on execution time, while memory usage is not directly compared. Details of the algorithm for each task are provided in the Appendix A. Performance for each task is gauged by calculating the average computation time from 100 replications. We compute the average for each category and present the mean of these averages as the overall score. To vary the dimensionality of the data in each task, we introduce a parameter $k \in \{1, 2, 4\}$, which scales the task dimension relative to a baseline size. Specifically, $k = 1$ corresponds to the baseline dimensionality, while $k = 2$ and $k = 4$ represent twofold and fourfold increases, respectively. In Appendix B, we detail the functions used to implement the benchmark tasks across different programming languages. Specifically, Table B2 shows the implementation in R, Table B3 in C++, Table B4 in MATLAB, Table B5 in Python, and Table B6 in Julia. Our goal was to ensure that the comparison across languages was as fair and consistent as possible.

In the **Matrix Manipulations** category, the suite evaluates operations involving large matrix manipulations. The first task involves the creation of a $(2500 \cdot k) \times (2500 \cdot k)$ matrix filled with random numbers following a standard normal distribution, followed by transposing and reshaping it to assess basic matrix manipulation efficiency. We note here that all the framework uses a default 64-bit float. The second task raises a $(2500 \cdot k) \times (2500 \cdot k)$ matrix, filled with normally distributed random values, to the 10th power, testing computational intensity and memory handling during power operations. The third task involves sorting a vector containing k million random values, which tests the sorting performance for large datasets. For that, we used for each language the default sort function with the relative default setting, and the NumPy sort function for Python based on the Quick Sort algorithm. The fourth task calculates a cross-product for a $(2500 \cdot k) \times (2500 \cdot k)$ matrix, focusing on matrix multiplication performance. Finally, the fifth task computes the determinant of a $(2500 \cdot k) \times (2500 \cdot k)$ matrix, evaluating the library's performance in calculating matrix properties. See Algorithms 1–5 in Appendix A for the detailed specifications.

The **Matrix Transformations** category assesses performance in a set of matrix and vector transformation operations. The first task applies a Fast Fourier Transform (FFT) on a vector of k million elements to test the efficiency of spectral transformations. The second task calculates the eigenvalues of a $(500 \cdot k) \times (500 \cdot k)$ matrix, which involves finding the roots of the characteristic polynomial and is a measure of eigen-decomposition speed. Third, linear regression is performed by generating a random $(2500 \cdot k) \times (2500 \cdot k)$ matrix with standard normal entries and a target vector $b = (1, 2, \ldots, 2500 \cdot k)^\top$, benchmarking matrix-solving techniques within linear algebra using the LU decomposition method. The fourth task applies Cholesky decomposition on a $(2500 \cdot k) \times (2500 \cdot k)$ positive-definite matrix, which benchmarks the efficiency of decomposing large matrices. The final matrix function test involves inverting a $(1000 \cdot k) \times (1000 \cdot k)$ matrix, which is again based on solving a linear system. See algorithms 6–10 in Appendix A for the detailed specifications.

The **Programming Tasks** category focuses on core algorithmic and computational patterns, emphasizing performance aspects such as memory usage, recursion, and numerical computation. To ensure consistency and comparability, all tasks are implemented using only standard built-in functions, without relying on specialized libraries or environment-specific features. In the first task, temporal autocorrelation for a time series of k million points is calculated for various lags, focusing on statistical computation performance over large data sets. The second task calculates the distance matrix for $1000 \cdot k$ points in a two-dimensional space, which involves computing Euclidean distances between all pairs of points and tests nested looping efficiency and memory usage. The third task implements hierarchical clustering for 100 points in two-dimensional space, testing recursive data structure manipulation and cluster merging algorithms. The fourth task performs Kriging interpolation on $1000 \cdot k$ spatial data points over a grid, which involves using a variogram model to predict values at unobserved locations and benchmarks geostatistical

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

interpolation efficiency. Finally, kernel density estimation is conducted for $1000 \cdot k$ data points, focusing on density estimation using a Gaussian kernel and benchmarking spatial analysis performance. See Algorithms 11–15 in Appendix A for the detailed specifications.

In the **I/O Operations** category, the suite assesses file manipulation efficiency. The first task was to write a CSV file with one million rows of random data and two columns, benchmarking the time required to generate and write large datasets. The second task reads the previously generated large CSV file, testing reading performance for large text-based data sets. The third task writes 1 million random values to a binary file, assessing efficiency in handling binary data storage. The fourth task reads the binary file to test rapid access to and manipulation of large binary datasets. Finally, the fifth task performs random access reads from a text file with one million lines, simulating random data access scenarios in large files by repeatedly accessing specific lines. In this last category, we keep the dimensionality of the tasks fixed, as it does not play an essential role. See algorithms 16–20 in Appendix A for the detailed specifications.

We emphasize that the fairness of the comparison is ensured by implementing the benchmark algorithms using commonly accepted and efficient practices in each language, relying on idiomatic approaches and well-established libraries. In particular, we employed built-in or widely used standard libraries: `rnorm()` and `crossprod()` in `R`, `Eigen` and `FFTW` in `C++`, built-in functions and matrix notation in `MATLAB`, `NumPy`, `SciPy`, and `pandas` in `Python`, and `LinearAlgebra` and `FFTW` in `Julia`. While some implementations utilize widely adopted external libraries-such as `Eigen` and `FFTW` in `C++`, or `SciPy` and `pandas` in `Python`-these libraries are considered standard in their respective ecosystems and were selected to reflect typical usage in scientific computing. In all cases, we ensured algorithmic equivalence by adhering to identical computational procedures, numerical operations, and data structures across languages. Therefore, any performance differences observed in the benchmarks can be attributed to the inherent design and execution characteristics of the languages, including compiler optimisations, memory management, and the efficiency of underlying numerical libraries, rather than to algorithmic formulation or implementation fidelity differences.

3.2 | Hardware and Software Specification

All computational experiments reported in this paper were conducted on a high-performance computing node running Ubuntu 24.04 LTS on a `6.8.0-40-generic` Linux kernel (x86_64 architecture). The system featured two Intel Xeon Platinum 8460Y+ processors, each with 40 cores (totaling 80 physical cores and 160 threads via multi-threading), and was equipped with 1024 GiB of RAM. The software stack used for benchmarking included: `MATLAB` Version 24.2.0.2863752 (R2024b Update 5), `R` version 4.4.3 (2025-02-28), `Julia` v1.10.5 with the `LinearAlgebra` and `FFTW.jl` (v1.8.0) standard libraries, and `Python` 3.12.3 with `NumPy` 2.1.3, `SciPy` 1.15.2, and `pandas` 2.2.3. `C++` implementations were compiled using `g++` 13.3.0 and employed the `Eigen` 3.4.0 and `FFTW` 3.3.10 libraries for matrix and Fourier operations, respectively. We note that while the benchmark results presented here are representative, performance outcomes may vary across different high-performance computing systems due to variations in hardware configurations, such as CPU architecture, manufacturer-specific optimisations, and memory hierarchy. Nonetheless, the relative trends observed across programming languages are generally expected to hold across comparable computational environments.

3.3 | Results of Time Complexity Comparison

Monte-Carlo simulation results are presented in Tables 2 and 3, which report mean execution times and relative performance metrics, respectively. The relative values are benchmarked against `R` using the default `BLAS` libraries, which serve as the baseline (a value of 1 indicates baseline-level performance). These results correspond to the highest scaling parameter considered, $k = 4$; results for lower scaling parameters $k = 1, 2$ are available in Appendix Table B1, see Table C1–C4 in Appendix C. The relative values indicate how much faster or slower a programming environment performs compared to `R BLAS` for given tasks, with higher values indicating better performance.

Figures 1–4 illustrate mean execution times across all tasks and programming environments for all scaling parameter values $k \in \{1, 2, 4\}$. Each figure corresponds to one of the four benchmarking categories: Matrix Manipulations, Matrix Transformations, Statistical and Programming Tasks, and File I/O Operations. These visualizations comprehensively depict performance scaling relative to task complexity across different computational frameworks. Since increasing the value of k magnifies performance differences without altering the relative ranking of programming environments, we focus our analysis on the largest problem size ($k = 4$). This setting is particularly relevant for computationally intensive applications in environmetrics.

Significant differences in computational efficiency were observed both across programming languages and among configurations within the same environment. Using the default configuration of `R` (with standard `BLAS` libraries) as a baseline, nearly all other environments demonstrate substantial speed improvements. A notable exception is `Rcpp` in matrix manipulation and transformation tasks, where performance is only marginally different—sometimes slightly better, sometimes worse. This is primarily because standard `Rcpp` compilations do not employ optimized mathematical libraries, limiting potential gains. In contrast, the other environments—most of which rely on some form of optimized `BLAS`-exhibit significantly faster execution. Thus, `R BLAS` emerges as the slowest setup overall, making it a suitable baseline for benchmarking relative gains. `MATLAB` shows consistently strong performance, particularly in matrix-centric tasks. It ranks among the top in matrix exponentiation (Task 1.2) and Cholesky decomposition (Task 2.4), underscoring its strength in numerical linear algebra. `Python`, used with `NumPy` (defaulting to `OpenBLAS`), delivers competitive performance in matrix operations and often rivals `MATLAB` and `Julia`. However, its performance in computation-heavy tasks is generally lower due to interpreter overhead. Although tools like `numba` (Lam et al. 2015) enable just-in-time (JIT)

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 2 | Mean execution times (in seconds) for each task across different programming environments, computed over 100 trials. Tasks are grouped into four categories: Matrix Manipulations, Matrix Transformations, Programming Tasks, and I/O Operations. Category averages are calculated as the arithmetic mean of the tasks within each category, and the overall average is the mean of these category-level averages. Results shown correspond to scaling parameter $k = 4$.

Task	R BLAS	R MKL	R OpenBLAS	Rcpp	MATLAB	Python	Julia
1.1 Creation, transpose, and deformation of a (2500Â·k) × (2500Â·k) matrix	5,053	5,037	5,073	3,991	1,069	1,974	0,728
1.2 Exponentiation of a (2500Â·k) × (2500Â·k) matrix to the power of 10	1,465	1,465	1,462	3,096	0,105	0,300	0,889
1.3 Sorting of kÂ·1,000,000 random values	0,338	0,336	0,336	0,297	0,103	0,032	0,118
1.4 Cross-product of a (2500Â·k) × (2500Â·k) matrix	393,385	0,836	0,960	118,660	0,733	0,792	9,888
1.5 Determinant of a (2500Â·k) × (2500Â·k) random matrix	166,076	1,524	0,749	42,078	0,724	1,693	1,691
2.1 FFT over kÂ·1,000,000 random values	0,205	0,207	0,216	0,135	0,032	0,072	0,109
2.2 Eigenvalues of a (500Â·k) × (500Â·k) random matrix	10,821	1,462	4,844	8,152	1,251	0,879	3,469
2.3 Linear regression over a (2500Â·k) × (2500Â·k) matrix	484,767	1,752	1,792	197,252	2,252	3,545	1,805
2.4 Cholesky decomposition of a (2500Â·k) × (2500Â·k) matrix	91,498	0,669	0,586	21,567	0,147	0,798	0,870
2.5 Inverse of a (1000Â·k) × (1000Â·k) random matrix	32,906	0,584	0,336	13,770	0,355	0,543	0,561
3.1 Temporal autocorrelation calculation for kÂ·1,000,000 data points	0,631	0,624	0,627	0,291	0,191	0,175	0,402
3.2 Distance matrix calculation for 1000Â·k points	1,833	1,821	1,881	0,159	0,116	16,780	0,062
3.3 Hierarchical clustering on 100Â·k points	99,843	98,821	99,834	10,964	11,546	169,381	7,916
3.4 Kriging interpolation for 1000Â·k points	0,061	0,060	0,060	0,202	0,385	0,107	0,025
3.5 Kernel density estimation for 1000Â·k points	0,213	0,212	0,214	0,140	0,193	0,109	0,162
4.1 Writing a 1,000,000-row CSV file	0,887	0,888	0,891	0,443	0,673	1,369	0,456
4.2 Reading a 1,000,000-row CSV file	0,896	0,890	0,891	0,254	0,650	0,265	0,047
4.3 Writing a large binary file	0,065	0,066	0,074	0,004	0,004	0,004	0,006
4.4 Reading a large binary file	0,014	0,014	0,014	0,001	0,001	0,001	0,001
4.5 Random access of 10 lines from a large text file	0,010	0,010	0,010	0,015	0,017	0,001	0,000

compilation for numerical routines, we excluded such enhancements to preserve comparability across base configurations. Julia offers robust performance across all categories, particularly excelling in programming-intensive tasks. It also performs competitively in linear algebra benchmarks, often surpassing both Python and MATLAB.

While the default R BLAS configuration consistently ranks lowest, significant performance improvements are achievable using optimized libraries such as OpenBLAS and MKL. These libraries enable R to achieve competitive execution speeds in key linear algebra operations, including cross-products, determinants, eigenvalue decomposition, Cholesky factorization, and matrix inversion. Moreover, employing Rcpp illustrates that linear algebra operations largely depend on the underlying numerical libraries rather than the inherent speed of the programming language itself. Consequently, the performance gain from using C++ directly for these operations is minimal. Nonetheless, Rcpp provides substantial speed improvements in both Programming Tasks and I/O operations, effectively competing with Julia and MATLAB in these categories. Although both changing the default mathematical libraries (e.g., from BLAS to OpenBLAS) and using Rcpp lead to performance improvements in R, they operate at different levels: the change of libraries accelerates low-level linear algebra computations, whereas Rcpp provides a bridge to C++ for implementing efficient algorithms within R packages. As such, they are complementary rather than interchangeable tools.

In summary, the benchmarking results highlight distinct strengths across programming environments. MATLAB consistently delivers top-tier performance in numerically intensive tasks, reaffirming its role as a leading platform for scientific computing. Julia, with its high-level syntax and performance approaching that of low-level languages, proves to be a strong open-source contender-especially excelling in programming-heavy tasks. Python, while somewhat hindered by its interpreted nature, remains a well-rounded and widely adopted tool, particularly effective when paired with optimized libraries like NumPy.

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

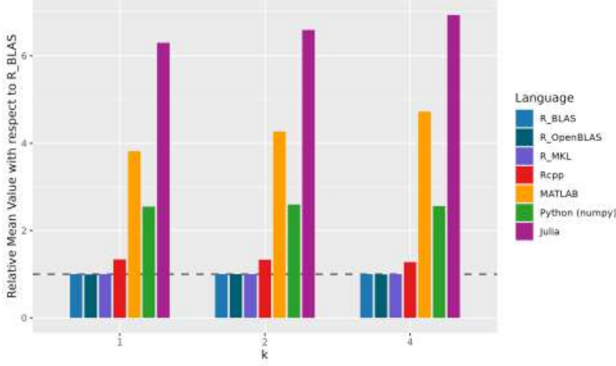
TABLE 3 | Relative efficiency across programming languages and frameworks for various computational tasks, benchmarked against R with BLAS (baseline = 1). The table presents performance comparisons for matrix manipulations, matrix transformations, general programming tasks, and I/O operations using six environments: R with MKL, R with OpenBLAS, Rcpp (C++ integration with R), MATLAB, Python (with NumPy), and Julia. Each value reflects how many times faster a given method is relative to R BLAS for the specified task. Higher values indicate faster performance. Arithmetic means are used to compute category and overall averages. All benchmarks were conducted with a scaling parameter $k = 4$.

Task	R MKL	R OpenBLAS	Rcpp	MATLAB	Python	Julia
1.1 Creation, transpose, and deformation of a (2500·k) × (2500·k) matrix	1	1	1	5	3	7
1.2 Exponentiation of a (2500·k) × (2500·k) matrix to the power of 10	1	1	0	14	5	2
1.3 Sorting of k·1,000,000 random values	1	1	1	3	10	3
1.4 Cross-product of a (2500·k) × (2500·k) matrix	471	410	3	537	497	40
1.5 Determinant of a (2500·k) × (2500·k) random matrix	109	222	4	229	98	98
2.1 FFT over k·1,000,000 random values	1	1	2	6	3	2
2.2 Eigenvalues of a (500·k) × (500·k) random matrix	7	2	1	9	12	3
2.3 Linear regression over a (2500·k) × (2500·k) matrix	277	271	2	215	137	269
2.4 Cholesky decomposition of a (2500·k) × (2500·k) matrix	137	156	4	624	115	105
2.5 Inverse of a (1000·k) × (1000·k) random matrix	56	98	2	93	61	59
3.1 Temporal autocorrelation calculation for k·1,000,000 data points	1	1	2	3	4	2
3.2 Distance matrix calculation for 1000·k points	1	1	11	16	0	29
3.3 Hierarchical clustering on 100·k points	1	1	9	9	1	13
3.4 Kriging interpolation for 1000·k points	1	1	0	0	1	2
3.5 Kernel density estimation for 1000·k points	1	1	2	1	2	1
4.1 Writing a 1,000,000-row CSV file	1	1	2	1	1	2
4.2 Reading a 1,000,000-row CSV file	1	1	4	1	3	19
4.3 Writing a large binary file	1	1	17	17	18	11
4.4 Reading a large binary file	1	1	20	11	21	19
4.5 Random access of 10 lines from a large text file	1	1	1	1	11	21

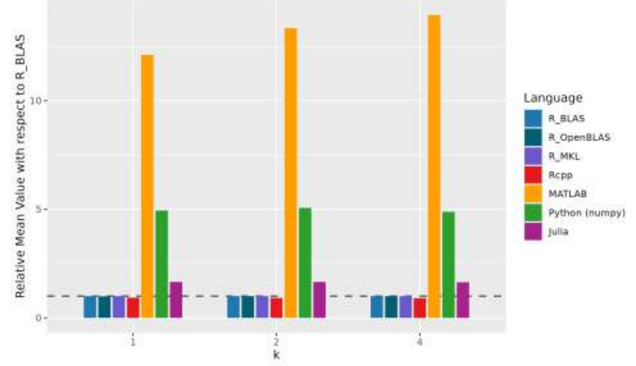
R, often critiqued for its performance limitations, demonstrates substantial potential when enhanced with targeted optimizations. Replacing the default BLAS with optimised libraries such as OpenBLAS or MKL markedly improves execution speed in core linear algebra operations. Meanwhile, leveraging Rcpp enables seamless integration with C++, offering significant speedups in algorithmic and I/O-intensive tasks. These enhancements are not mutually exclusive; they complement each other, accelerating both low-level computations and high-level algorithmic workflows. These findings suggest that while R may not lead in raw performance out of the box, it remains a flexible and highly adaptable environment. With modest effort, it can be transformed into a competitive option for demanding computational workloads, bridging the gap between user-friendly syntax and high-performance execution. We note that while all the programming languages discussed can utilize various linear algebra libraries and interface with C++ functions, our optimization analysis specifically focused on the R framework. This choice aligns with our primary objective: to assist statisticians working with geospatial data who predominantly use R in improving the computational efficiency of their programs through these optimization techniques.

In addition, we have shown that when high performance is needed in R, packages like Rcpp are recommended for iterative tasks. However, for matrix operations in particular, users

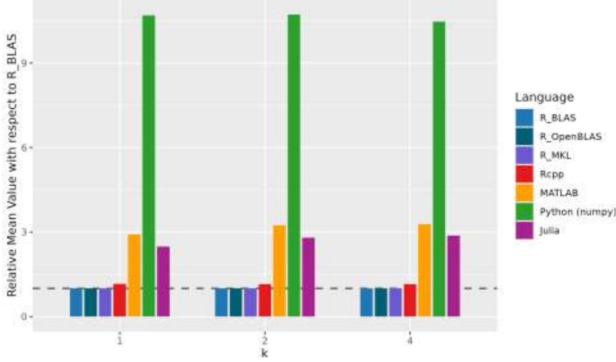
1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



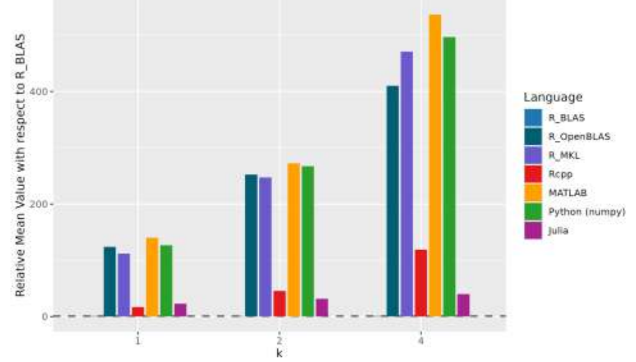
(1.1) Creation, transpose, and deformation of a (2500·k)x(2500·k) matrix



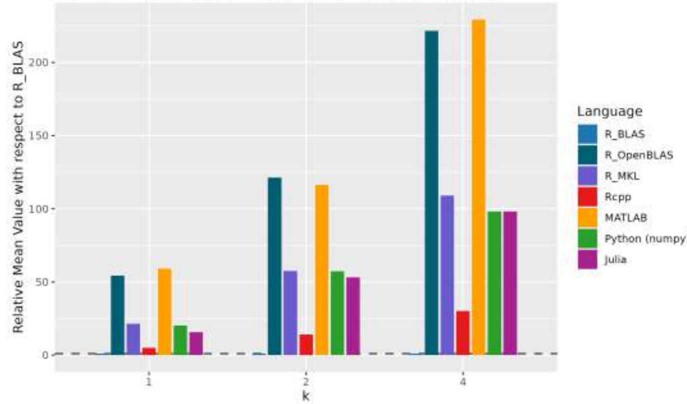
(1.2) Exponentiation of a (2500·k)x(2500·k) matrix to the power of 10



(1.3) Sorting of k·1,000,000 random values



(1.4) Cross-product of a (2500·k)x(2500·k) matrix

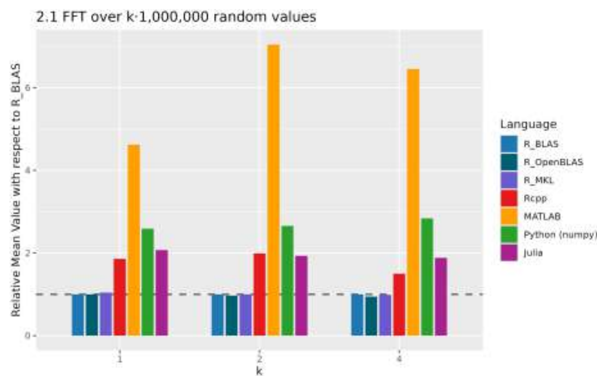


(1.5) Determinant of a (2500·k)x(2500·k) random matrix

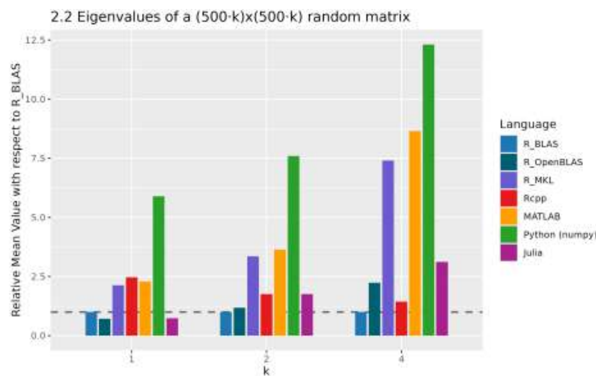
FIGURE 1 | Performance plots for Group 1 tasks (Matrix Manipulations), showing relative efficiency across programming environments for $k = 1$. Tasks include: (1.1) Creation, transpose, and deformation of a $(2500 \cdot k) \times (2500 \cdot k)$ matrix; (1.2) Matrix exponentiation to the power of 10; (1.3) Sorting of $k \cdot 1,000,000$ random values; (1.4) Cross-product of a $(2500 \cdot k) \times (2500 \cdot k)$ matrix; (1.5) Determinant of a random $(2500 \cdot k) \times (2500 \cdot k)$ matrix.

may prefer solutions that do not require switching languages. Several `R` packages, including `Rfast` (Papadakis et al. 2018), offer linear algebra functions implemented in `C++`. Alternatively, the recent `MPCR` package (Salvana et al. 2024) supports multi- and mixed-precision (16-, 32-, and 64-bit), optimizing memory usage and computational speed. `MPCR` utilizes CUDA for GPU acceleration and streamlines CPU-GPU memory management. It is compatible with Intel's `MKL` (or `OpenBLAS` if unavailable) and automatically installs dependencies like `Blaspp` and `Lapackpp`, making setup seamless. We chose to exclude a comparison using these packages, as they are based on the same principle we presented-leveraging optimized libraries and `C++` code to enhance computational speed.

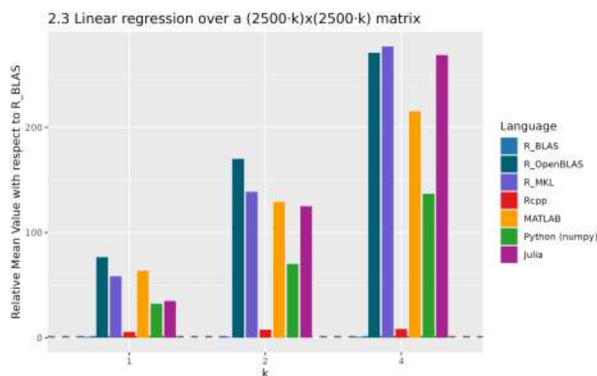
1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



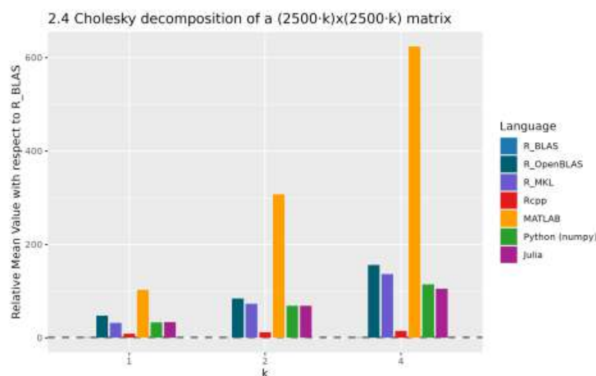
(2.1) FFT over k·1,000,000 random values



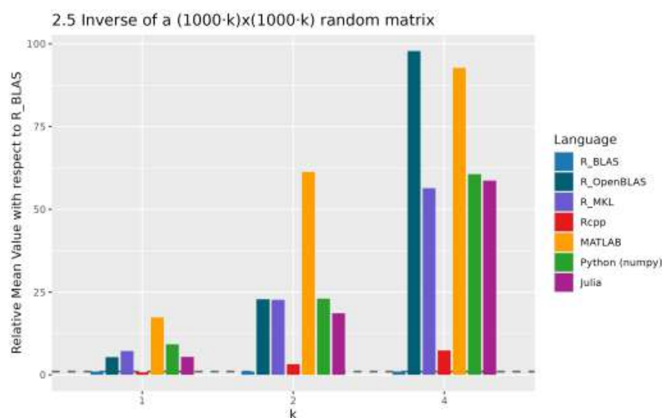
(2.2) Eigenvalues of a (500·k)x(500·k) random matrix



(2.3) Linear regression over a (2500·k)x(2500·k) matrix



(2.4) Cholesky decomposition of a (2500·k)x(2500·k) matrix



(2.5) Inverse of a (1000·k)x(1000·k) random matrix

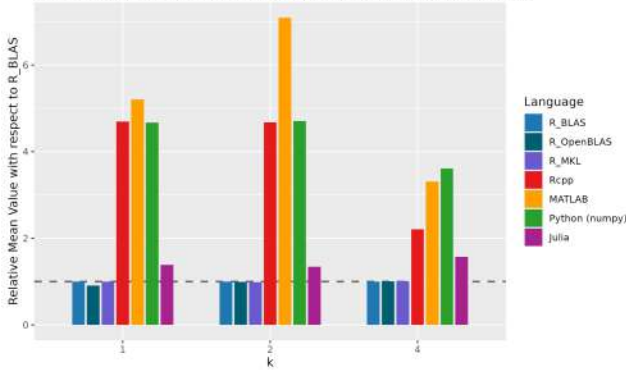
FIGURE 2 | Performance plots for Group 2 tasks (Matrix Transformations), showing relative efficiency across programming environments for $k = 1$. Tasks include: (2.1) Fast Fourier Transform over $k \cdot 1,000,000$ values; (2.2) Eigenvalue computation for a $(500 \cdot k) \times (500 \cdot k)$ matrix; (2.3) Linear regression over a $(2500 \cdot k) \times (2500 \cdot k)$ matrix; (2.4) Cholesky decomposition of a $(2500 \cdot k) \times (2500 \cdot k)$ matrix; (2.5) Inversion of a $(1000 \cdot k) \times (1000 \cdot k)$ random matrix.

4 | Practical Guide to Improve `R` Performances

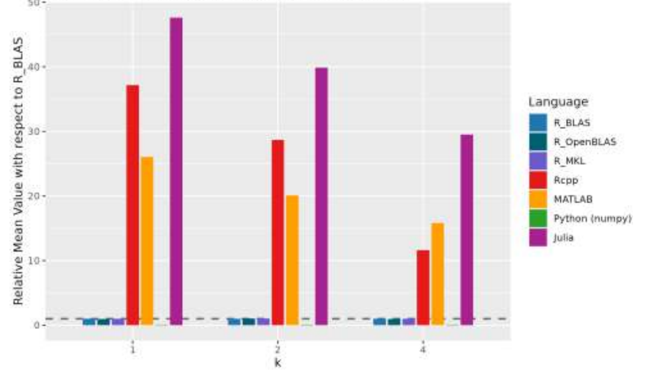
4.1 | Change Default Libraries

As we showed, the default `R` libraries are not fully optimized, so you may see significant performance improvements for linear algebra computations by switching to a different `BLAS` library. Traditional open-source solutions for this include `ATLAS`, `GotoBLAS` (no longer developed), and more recently `OpenBLAS` (a fork of `GotoBLAS2`). The Intel `MKL` (Math Kernel Library) provides a closed-source alternative. While optimized for Intel processors, it has specific license terms and is known to perform suboptimally on AMD processors. MKL is also not natively compatible with Apple M-series chips based on ARM architecture, limiting its applicability in those environments.

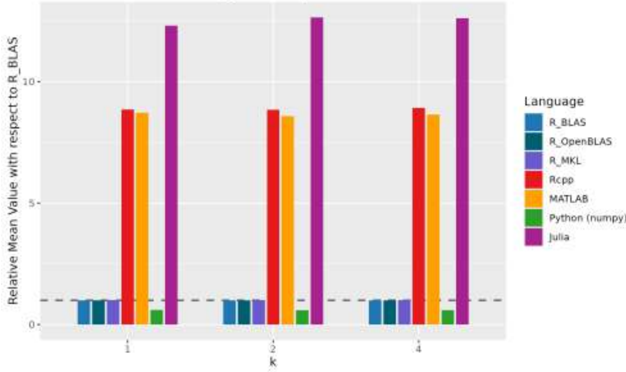
1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



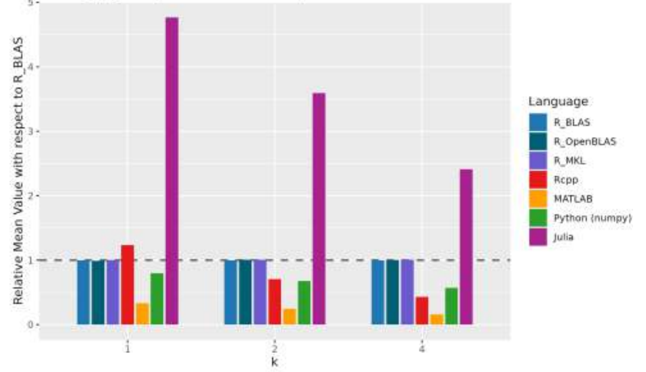
(3.1) Temporal autocorrelation calculation for k·1,000,000 data points



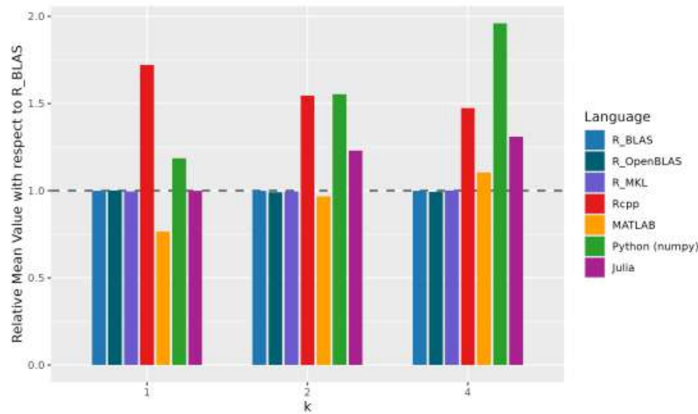
(3.2) Distance matrix calculation for 1000·k points



(3.3) Hierarchical clustering on 100·k points



(3.4) Kriging interpolation for 1000·k points

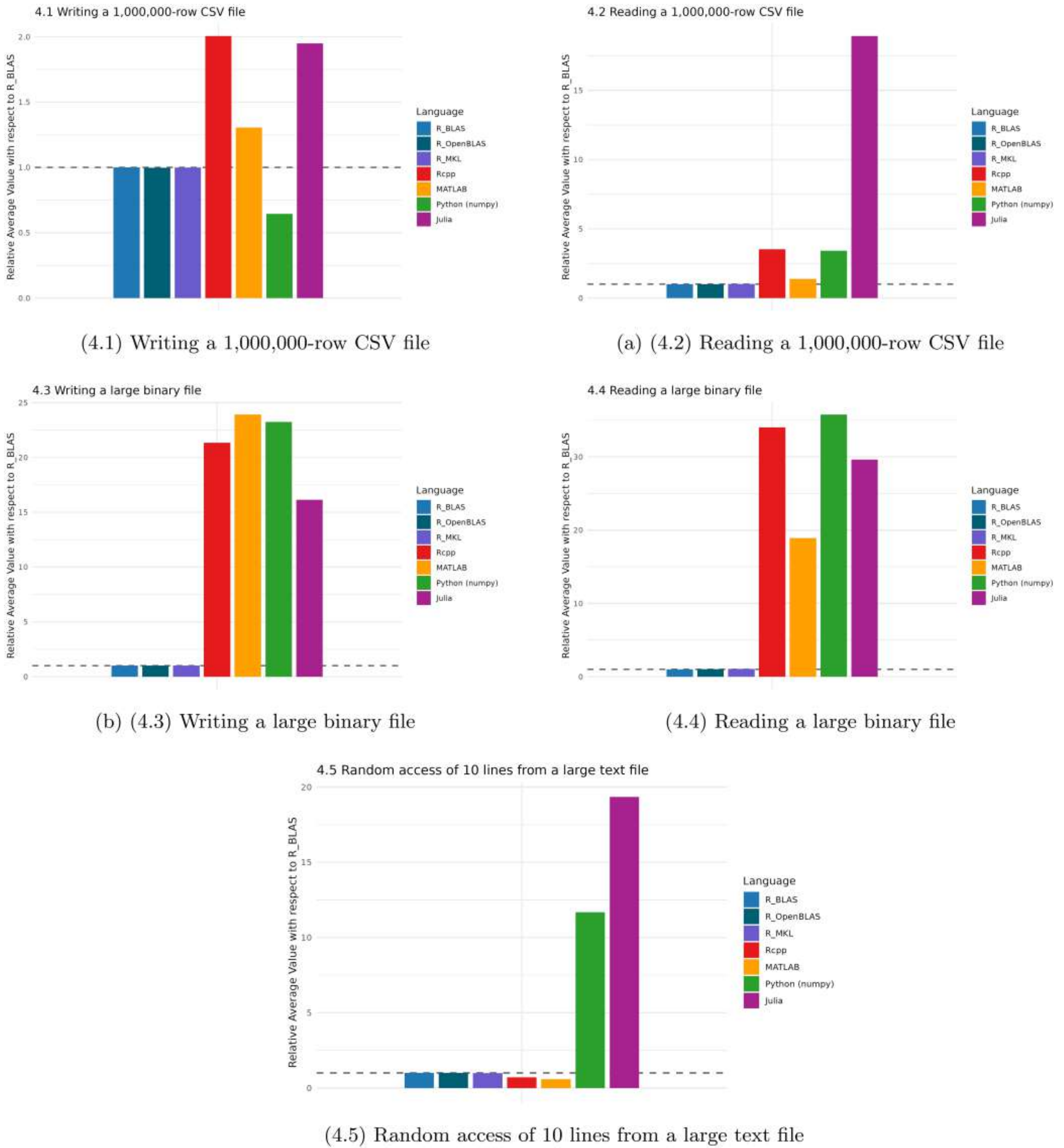


(3.5) Kernel density estimation for 1000·k points

FIGURE 3 | Performance plots for Group 3 tasks (Statistical and Programming Tasks), showing relative efficiency across programming environments for $k = 1$. Tasks include: (3.1) Temporal autocorrelation calculation over $k \cdot 1,000,000$ data points; (3.2) Distance matrix computation for $1000 \cdot k$ points; (3.3) Hierarchical clustering on $100 \cdot k$ points; (3.4) Kriging interpolation for $1000 \cdot k$ points; (3.5) Kernel density estimation over $1000 \cdot k$ points.

Compiling `R` and `BLAS` libraries from source is recommended for optimal performance, though this process can be intricate and susceptible to unexpected issues. Employing non-standard `BLAS` libraries with `R` may also lead to suboptimal outcomes (for further details, see Appendix in "Programming Tasks" of R Core Team (2024)). Our guide specifically targets the `Ubuntu` environment within Linux. This choice is strategic, considering that these guidelines might become quickly outdated. We chose `Ubuntu` because it exemplifies the feasibility and simplicity of switching libraries and is the predominant operating system in many university-owned personal high-performance computing (HPC) setups.

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(4.1) Writing a 1,000,000-row CSV file

(a) (4.2) Reading a 1,000,000-row CSV file

(b) (4.3) Writing a large binary file

(4.4) Reading a large binary file

(4.5) Random access of 10 lines from a large text file

FIGURE 4 | Performance plots for Group 4 tasks (File Input/Output Operations), showing relative efficiency across programming environments. Tasks include: (4.1) Writing a CSV file with 1,000,000 rows; (4.2) Reading a CSV file with 1,000,000 rows; (4.3) Writing a large binary file; (4.4) Reading a large binary file; (4.5) Random access of 10 lines from a large text file.

In `Ubuntu`, there are multiple free and open-source solutions, like `ATLAS` and `OpenBLAS`, or proprietary libraries like Intel' `MKL`. There are three different versions of `OpenBLAS`: serial, `OpenMP`, and `pthread` (`POSIX` threads). `OpenMP` and pthreads allow for parallel processing. It appears that `libopenblas-base` installs `pthreads` by default. To install `ATLAS`, `OpenBLAS` or Intel' `MKL` one can use the following commands:

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Bash Code

```
# install OpenBLAS
sudo apt-get install libopenblas-base

# install ATLAS
sudo apt-get install libatlas3-base liblapack3

# install MKL
sudo apt install intel-mkl
```

To prevent potential numerical errors with Intel's `MKL`, it is important to set the environment variable `MKL_THREADING_LAYER=GNU` when using `MKL` libraries.

Terminal Command

```
export MKL_THREADING_LAYER=GNU
```

It is possible to install multiple libraries and easily switch between them using:

Bash Code

```
sudo update-alternatives --config liblapack.so.3-x86_64-linux-gnu
```

This command should display the following output (with potentially different paths, installations, and selections):

Terminal Output

```
@ubuntu:~$ sudo update-alternatives --config liblapack.so.3-x86_64-linux-gnu
Choices for liblapack.so.3-x86_64-linux-gnu:

Sel   Path                                                    Priority  Status
------------------------------------------------------------------------------------
* 0   /usr/lib/x86_64-linux-gnu/openblas-pthread/liblapack.so.3   100   auto
  1   /usr/lib/x86_64-linux-gnu/atlas/liblapack.so.3              35    manual
  2   /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3             10    manual
  3   /usr/lib/x86_64-linux-gnu/openblas-pthread/liblapack.so.3   100   manual

Press <enter> to keep current choice[*], or type selection number:
```

Then, make sure to choose the corresponding LAPACK. Run:

Bash Code

```
sudo update-alternatives --config liblapack.so.3-x86_64-linux-gnu
```


1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

This command should display:

Terminal Output

```
@ubuntu:~$ sudo update-alternatives --config liblapack.so.3-x86_64-linux-gnu
Choices for liblapack.so.3-x86_64-linux-gnu:

Sel   Path                                                     Priority  Status
------------------------------------------------------------------------------
* 0   /usr/lib/x86_64-linux-gnu/openblas-pthread/liblapack.so.3   100   auto
  1   /usr/lib/x86_64-linux-gnu/atlas/liblapack.so.3              35    manual
  2   /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3             10    manual
  3   /usr/lib/x86_64-linux-gnu/openblas-pthread/liblapack.so.3   100   manual

Press <enter> to keep current choice[*], or type selection number:
```

To check the currently used LAPACK and BLAS within an R session, run:

R Code

```
sessionInfo()$LAPACK
sessionInfo()$BLAS
```

4.2 | Integrating R With C++

Another method to decrease computational time in R is integrating C++ functions. This can be accomplished using several packages such as RcppEigen (Bates and Eddelbuettel 2013), Rcpp (Eddelbuettel and François 2011), or RcppArmadillo (Eddelbuettel 2013). The Rcpp package offers a versatile interface to C++ without a specific focus on linear algebra, whereas RcppEigen and RcppArmadillo provide specialized, optimized linear algebra operations by interfacing with the Eigen and Armadillo C++ libraries, respectively. Detailed guides for these packages are readily available online. This section demonstrates the simplicity of integrating C++ code using only Rcpp through an iterative loop example.

To illustrate the use of Rcpp, we provide a minimal example that performs an iterative summation. The first code block defines a C++ function using Rcpp to compute the cumulative sum of a numeric vector. The second block contains an R script that loads the Rcpp package, compiles the C++ function via sourceCpp, and verifies its correctness.

C++ Code

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector cumulativeSum(NumericVector x) {
  int n = x.size();
  NumericVector result(n);
  double total = 0;
  for(int i = 0; i < n; ++i) {
    total += x[i];
    result[i] = total;
  }
  return result;
}
```


1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

R Code

```r
# Load the Rcpp library
library(Rcpp)
# Compile and load the C++ functions
sourceCpp('path/to/file.cpp')

# Define a numeric vector
x <- c(1, 2, 3, 4, 5)

# Check if the cumulative sum using R's cumsum function matches
# the result from cumulativeSum function in C++
all.equal(cumsum(x), cumulativeSum(x))
```

5 | Conclusions

In this study, we evaluated the performance of R MATLAB, Python, and Julia in handling geospatial modelling tasks, focusing on computationally intensive operations essential for spatial and spatio-temporal analysis. In the final section, we derive conclusions from our study, aiming to provide guidance for researchers developing or optimizing software for these applications.

Considering its competitive performance in matrix operations, MATLAB remains a strong contender for geospatial modelling tasks. However, its commercial licensing can hinder accessibility and reproducibility, two fundamental pillars of scientific progress. Therefore, researchers may benefit from exploring open-source alternatives. The results highlight Julia's strengths in numerical computing and matrix manipulation, positioning it as a robust option for complex spatial statistical tasks. This could be a valuable choice for developers familiar with MATLAB who do not require extensive pre-built solutions for spatial statistics, as Julia's package ecosystem in this domain is still maturing. Python demonstrated competitive performance in several numerical tasks but lagged in iterative tasks, for which exist workaround methods depending on the specific task. Given its popularity across numerous scientific fields, we believe Python is an excellent choice for developing spatial statistics software intended for a general audience. Moreover, the growing popularity of Machine Learning and Deep-Learning models in spatial statistics highlights the importance of using a framework capable of accessing the most widely adopted deep-learning libraries. As of 2025, these libraries include PyTorch (Paszke et al. 2019) and TensorFlow/Keras Team (2015), which positions Python as a strong candidate for such tasks, alongside R. Finally, R, while benefiting from a rich ecosystem of geospatial packages, generally lagged in speed compared to the other languages. However, by implementing proposed optimisation strategies-such as linking R to optimized libraries and integrating C++ for performance-critical tasks-our findings suggest that R's computational efficiency can be significantly improved. By adopting these techniques, spatial statisticians and data scientists can maximize the performance of their R environment, making it more competitive for high-demand geospatial analysis.

Acknowledgments

Lorenzo Tedesco's research was supported by the National Recovery and Resilience Plan (NRRP), Mission 4 Component 2 Investment 1.3, under Call for tender No. 341 of 15/03/2022, funded by the Italian Ministry of University and Research and the European Union's NextGenerationEU initiative. Award Number: PE 00000018, Concession Decree No. 1558 of 11/10/2022, CUP F83C22001720001, under the project titled *Growing Resilient Inclusive and Sustainable*(GRINS). Philipp Otto gratefully acknowledges partial funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) project number 501539976. Lorenzo Tedesco is also a member of the GNCS group of INdAM, whose support is gratefully acknowledged. Open access publishing facilitated by Universita degli Studi di Bergamo, as part of the Wiley - CRUI-CARE agreement.

Data Availability Statement

The authors have nothing to report.

References

Altman, N. S. 1992. "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression." *American Statistician* 46, no. 3: 175–185.

Baddeley, A., E. Rubak, and R. Turner. 2015. *Spatial Point Patterns: Methodology and Applications With R*. Chapman and Hall/CRC Press.

Baddeley, A., and R. Turner. 2005. "Spatstat: An R Package for Analyzing Spatial Point Patterns." *Journal of Statistical Software* 12, no. 6: 1–42. https://doi.org/10.18637/jss.v012.i06.

Baddeley, A., R. Turner, J. Mateu, and A. Bevan. 2013. "Hybrids of Gibbs Point Process Models and Their Implementation." *Journal of Statistical Software* 55, no. 11: 1–43.

Bates, D., and D. Eddelbuettel. 2013. "Fast and Elegant Numerical Linear Algebra Using the RcppEigen Package." *Journal of Statistical Software* 52: 1–24.

Bivand, R., and D. W. S. Wong. 2018. "Comparing Implementations of Global and Local Indicators of Spatial Association." *Test* 27, no. 3: 716–748.

Bivand, R. S., E. Pebesma, and V. Gomez-Rubio. 2013. *Applied Spatial Data Analysis With R*. 2nd ed. Springer.

Bivand, R. 2022. "R Packages for Analyzing Spatial Data: A Comparative Case Study With Areal Data." *Geographical Analysis* 54, no. 3: 488–518.

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Bonas, M., A. Datta, C. K. Wikle, et al. 2024. "Assessing Predictability of Environmental Time Series With Statistical and Machine Learning Models." *Environmetrics* 36, no. 1: e2864.

Carpenter, B., A. Gelman, M. D. Hoffman, et al. 2017. "Stan: A Probabilistic Programming Language." *Journal of Statistical Software* 76: 1–32.

Chen, W., Y. Li, B. J. Reich, and Y. Sun. 2020. "Deepkriging: Spatially Dependent Deep Neural Networks for Spatial Prediction." arXiv Preprint arXiv:2007.11972.

Cheng, J., B. Karambelkar, Y. Xie, et al. 2019. "Package 'leaflet'." R Package Version, 2(1).

Coleman, C., S. Lyon, L. Maliar, and S. Maliar. 2021. "Matlab, Python, Julia: What to Choose in Economics?" *Computational Economics* 58: 1263–1288.

Cressie, N., and C. K. Wikle. 2011. *Statistics for Spatio-Temporal Data*. John Wiley & Sons.

Datta, A., S. Banerjee, A. O. Finley, and A. E. Gelfand. 2016. "Hierarchical Nearest-Neighbor Gaussian Process Models for Large Geostatistical Datasets." *Journal of the American Statistical Association* 111, no. 514: 800–812.

Dumelle, M., M. Higham, and J. M. Ver Hoef. 2023. "Spmodel: Spatial Statistical Modeling and Prediction in R." *PLoS One* 18, no. 3: 1–32.

Eddelbuettel, D. 2013. "Rcpparmadillo." In *Seamless R and C++ Integration With Rcpp*, 139–153. Springer.

Eddelbuettel, D., and R. François. 2011. "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software* 40: 1–18.

Finley, A., and S. Banerjee. 2024. spBayes: Univariate and Multivariate Spatial-Temporal Modeling. R Package Version 0.4–8.

Finley, A. O., S. Banerjee, and B. P. Carlin. 2007. "spBayes: An R Package for Univariate and Multivariate Hierarchical Point-Referenced Spatial Models." *Journal of Statistical Software* 19, no. 4: 1–24.

Finley, A. O., S. Banerjee, and A. E. Gelfand. 2015. "spBayes for Large Univariate and Multivariate Point-Referenced Spatio-Temporal Data Models." *Journal of Statistical Software* 63, no. 13: 1–28.

GDAL/OGR contributors. 2024. *GDAL/OGR Geospatial Data Abstraction Software Library*. Open Source Geospatial Foundation.

Gillies, S., A. Bierbaum, K. Lautaportti, and O. Tonnhofer. 2007. "Shapely: Manipulation and Analysis of Geometric Objects".

Gneiting, T. 2002. "Nonseparable, Stationary Covariance Functions for Space–Time Data." *Journal of the American Statistical Association* 97, no. 458: 590–600.

Gräler, B., E. Pebesma, and G. Heuvelink. 2016. "Spatio-Temporal Interpolation Using Gstat." *R Journal* 8: 204–218.

Guinness, J. 2019. "Spectral Density Estimation for Random Fields via Periodic Embeddings." *Biometrika* 106, no. 2: 267–286.

Guinness, J. 2022. "Nonparametric Spectral Methods for Multivariate Spatial and Spatial–Temporal Data." *Journal of Multivariate Analysis* 187: 104823.

Harris, C. R., K. J. Millman, S. J. van der Walt, et al. 2020. "Array Programming With NumPy." *Nature* 585, no. 7825: 357–362.

Heaton, M. J., A. Datta, A. O. Finley, et al. 2019. "A Case Study Competition Among Methods for Analyzing Large Spatial Data." *Journal of Agricultural, Biological and Environmental Statistics* 24: 398–425.

Hoffimann, J. 2018. "Geostats.Jl - High-Performance Geostatistics in Julia." *Journal of Open Source Software* 3, no. 24: 692.

Hsieh, C.-j., I. Dhillon, P. Ravikumar, and M. Sustik. 2011. "Sparse Inverse Covariance Matrix Estimation Using Quadratic Approximation." In *Advances in Neural Information Processing Systems*, edited by J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, vol. 24. Curran Associates, Inc.

Hufnagel, L., D. Brockmann, and T. Geisel. 2004. "Forecast and Control of Epidemics in a Globalized World." *Proceedings of the National Academy of Sciences* 101, no. 42: 15124–15129.

Hunter, J. D. 2007. "Matplotlib: A 2d Graphics Environment." *Computing in Science & Engineering* 9, no. 3: 90–95.

Jordahl, K., J. V. den Bossche, M. Fleischmann, et al. 2020. "geopandas/geopandas: v0.8.1".

Jordan, M. I. 2013. "On Statistics, Computation and Scalability." *Bernoulli* 19, no. 4: 1378–1390.

Klamer, B. 2024. "Faster BLAS in R." https://brettklamer.com/diversions/statistical/faster-blas-in-r/.

Lam, S. K., A. Pitrou, and S. Seibert. 2015. "Numba: A Llvm-Based Python Jit Compiler." In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 1–6. Association for Computing Machinery.

Lee, D. 2013. "CARBayes: An R Package for Bayesian Spatial Modeling With Conditional Autoregressive Priors." *Journal of Statistical Software* 55, no. 13: 1–24.

LeSage, J., and R. K. Pace. 2009. *Introduction to Spatial Econometrics*. Chapman and Hall/CRC.

LeSage, J. P. 1998. *Applied Econometrics Using MATLAB*. University of Toledo.

Li, T., Y. Wang, and K. Fang. 2024. "A Semiparametric Dynamic Higher-Order Spatial Autoregressive Model." *Statistical Papers* 65, no. 2: 1085–1123.

McKerns, M. M., L. Strand, T. Sullivan, A. Fang, and M. A. Aivazis. 2012. "Building a Framework for Predictive Science." arXiv Preprint arXiv:1202.1056.

McKinney, W. 2010. "Data Structures for Statistical Computing in Python." In *Proceedings of the 9th Python in Science Conference*, edited by S. van der Walt and J. Millman, 56–61. SciPy Proceedings.

Otto, P., O. Doğan, and S. Taşpinar. 2023. "A Dynamic Spatiotemporal Stochastic Volatility Model With an Application to Environmental Risks." *Econometrics and Statistics*.

Otto, P., A. Fusta Moro, J. Rodeschini, et al. 2024. "Spatiotemporal Modelling of PM 2.5 Concentrations in Lombardy (Italy): A Comparative Study." *Environmental and Ecological Statistics* 31, no. 2: 1–28.

Pace, R. K., and R. Barry. 2003. "Spatial Statistics Toolbox 2.0. FDELW2. m. February, 15."

pandas development team, T. 2020. "Pandas-Dev/Pandas: Pandas".

Papadakis, M., M. Tsagris, M. Dimitriadis, et al. 2018. "Package 'rfast'".

Paszke, A., S. Gross, F. Massa, et al. 2019. "Pytorch: An Imperative Style, High-Performance Deep Learning Library." *Advances in Neural Information Processing Systems* 33: 8026–8037.

Pebesma, E. 2018. "Simple Features for R: Standardized Support for Spatial Vector Data." *R Journal* 10, no. 1: 439–446.

Pebesma, E., and R. Bivand. 2023. *Spatial Data Science: With Applications in R*. Chapman and Hall/CRC.

Pebesma, E. J. 2004. "Multivariable Geostatistics in S: The Gstat Package." *Computers & Geosciences* 30: 683–691.

Pebesma, E. J., and R. Bivand. 2005. "Classes and Methods for Spatial Data in R." *R News* 5, no. 2: 9–13.

R Core Team. 2024. *R Installation and Administration*. R Foundation for Statistical Computing.

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Reichstein, M., G. Camps-Valls, B. Stevens, et al. 2019. "Deep Learning and Process Understanding for Data-Driven Earth System Science." *Nature* 566, no. 7743: 195–204.

Rey, S. J., and L. Anselin. 2009. "Pysal: A Python Library of Spatial Analytical Methods." In *Handbook of Applied Spatial Analysis: Software Tools, Methods and Applications*, 175–193. Springer.

Rue, H., S. Martino, and N. Chopin. 2009. "Approximate Bayesian Inference for Latent Gaussian Models by Using Integrated Nested Laplace Approximations." *Journal of the Royal Statistical Society, Series B: Statistical Methodology* 71, no. 2: 319–392.

Sainsbury-Dale, M., A. Zammit-Mangion, and N. Cressie. 2024. "Modeling Big, Heterogeneous, Non-Gaussian Spatial and Spatio-Temporal Data Using FRK." *Journal of Statistical Software* 108, no. 10: 1–39.

Salvana, M. L. O., S. Abdulah, M. Kim, D. Helmy, Y. Sun, and M. G. Genton. 2024. "Mpcr: Multi-and Mixed-Precision Computations Package in R." *arXiv preprint arXiv:2406.02701*.

Schmidberger, M., M. Morgan, D. Eddelbuettel, H. Yu, L. Tierney, and U. Mansmann. 2009. "State of the Art in Parallel Computing With R." *Journal of Statistical Software* 31: 1–27.

Team, T. 2015. "Tensorflow: Large-Scale Machine Learning on Heterogeneous Systems, 2015." http://tensorflow.org/Softwareavailablefromtensorflow.org.

Tennekes, M. 2018. "Tmap: Thematic Maps in R." *Journal of Statistical Software* 84, no. 6: 1–39.

The MathWorks, Inc. 2024. *Parallel Computing Toolbox User's Guide*. MathWorks, Inc. https://www.mathworks.com/products/parallel-computing.html.

Virtanen, P., R. Gommers, T. E. Oliphant, et al. 2020. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." *Nature Methods* 17, no. 3: 261–272. https://doi.org/10.1038/s41592-019-0686-2.

Wang, Y., F. Finazzi, and A. Fassò. 2021. "D-Stem v2: A Software for Modelling Functional Spatio-Temporal Data." arXiv Preprint arXiv:2101.11370.

Wickham, H. 2016. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

Wikle, C. K., and A. Zammit-Mangion. 2023. "Statistical Deep Learning for Spatial and Spatiotemporal Data." *Annual Review of Statistics and Its Application* 10, no. 1: 247–270.

Wikle, C. K., A. Zammit-Mangion, and N. Cressie. 2019. *Spatio-Temporal Statistics With R*. Chapman and Hall/CRC.

Zammit-Mangion, A., and N. Cressie. 2021. "FRK: An R Package for Spatial and Spatio-Temporal Prediction With Large Datasets." *Journal of Statistical Software* 98, no. 4: 1–48.

Zeroual, A., F. Harrou, A. Dairi, and Y. Sun. 2020. "Deep Learning Methods for Forecasting COVID-19 Time-Series Data: A Comparative Study." *Chaos, Solitons & Fractals* 140: 110121.

Appendix A

Algorithms

This section outlines the algorithms employed for the tasks detailed in Table 2. Specifically, the algorithms used for tasks related to Matrix Manipulations, Matrix Transformations, Programming Tasks, and I/O Operations are described in Sections in "Algorithm: Matrix Manipulations, Matrix Transformations, Programming Tasks, and I/O Operations", respectively. In each algorithm, states denoted by an asterisk (*) indicate operations that are excluded from computation time measurements.

Algorithm: Matrix Manipulations

ALGORITHM 1 | Creation, transpose, and deformation of a $(2500 \cdot k) \times (2500 \cdot k)$ matrix.

1: Generate a $(2500 \cdot k) \times (2500 \cdot k)$ random matrix A with independent standard normal components
2: Transpose A
3: Reshape A as a $(1500 \cdot k) \times (5000 \cdot k)$ matrix
4: Transpose A again

ALGORITHM 2 | Exponentiation of a $(2500 \cdot k) \times (2500 \cdot k)$ to power 10.

1: *Generate a random $(2500 \cdot k) \times (2500 \cdot k)$ matrix A with independent standard normal components
2: *Take the absolute value of A and divide by 2: $A \leftarrow \frac{|A|}{2}$
3: Raise A element-wise to the power of 10: $A \leftarrow A^{10}$

ALGORITHM 3 | Sorting of k million random values.

1: *Generate a $(k \cdot 1000000)$–dimensional random vector a with independent standard normal components
2: Sort the elements of a in ascending order

ALGORITHM 4 | Cross-product of a $(2500 \cdot k) \times (2500 \cdot k)$ matrix.

1: *Generate a $(2500 \cdot k) \times (2500 \cdot k)$ random matrix A with independent standard normal components
2: Compute the cross-product: $B \leftarrow A^\top A$

ALGORITHM 5 | Determinant of a $(2500 \cdot k) \times (2500 \cdot k)$ random matrix.

1: *Generate a $(2500 \cdot k) \times (2500 \cdot k)$ random matrix A with independent standard normal components
2: Compute the determinant of A: det_val $\leftarrow$ det(A)

Matrix Transformations

ALGORITHM 6 | FFT over k million random values.

1: *Generate a $(k \cdot 1000000)$–dimensional random vector a with independent standard normal components
2: Compute the Fast Fourier Transform (FFT): $b \leftarrow$ FFT(a)

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

ALGORITHM 7 | Eigenvalues of a $(500 \cdot k) \times (500 \cdot k)$ random matrix.

1: *Generate a $(500 \cdot k) \times (500 \cdot k)$ random matrix A with independent standard normal components
2: Compute the eigenvalues of A: $\lambda \leftarrow \text{eigenvalues}(A)$

ALGORITHM 8 | Linear regression over a $(2500 \cdot k) \times (2500 \cdot k)$ matrix.

1: *Generate a $(2500 \cdot k) \times (2500 \cdot k)$ random matrix A with independent standard normal components
2: *Generate a vector $b = [1, 2, \dots, 2500 \cdot k]^\top$
3: **Form** the normal equations: $M \leftarrow A^\top A, \quad d \leftarrow A^\top b$
4: **Solve** the linear system $Mc = d$ using LU decomposition

ALGORITHM 9 | Cholesky decomposition of a $(2500 \cdot k) \times (2500 \cdot k)$ matrix.

1: *Generate a $(2500 \cdot k) \times (2500 \cdot k)$ random matrix A with independent standard normal components
2: *Form a positive definite matrix: $A \leftarrow AA^\top$
3: Compute the Cholesky decomposition: $L \leftarrow \text{Cholesky}(A)$, where L is a lower triangular matrix

ALGORITHM 10 | Inverse of a $(1000 \cdot k) \times (1000 \cdot k)$ random matrix.

1: *Generate a random matrix A with independent standard normal components
2: Compute the inverse of A by solving the linear system: $A^{-1} \leftarrow \text{solve}(A, I)$, where I is the identity matrix of size $(1000 \cdot k)$

Programming Tasks

ALGORITHM 11 | Temporal autocorrelation calculation for $k \cdot 1000000$ data points.

1: *Generate a $(k \cdot 1000000)$−dimensional random time series S with independent uniform components
2: **for** $l \leftarrow 1$ to 10 **do**
3: Compute the temporal autocorrelation for lag l:

$$\text{Autocorr}(S, l) = \frac{\sum_{i=1}^{n-l} \left(S_i - \bar{S}\right)\left(S_{i+l} - \bar{S}\right)}{\sum_{i=1}^{n} \left(S_i - \bar{S}\right)^2}$$

where $\bar{S}$ is the mean of S and n is the length of the series.
4: **end for**

ALGORITHM 12 | Distance matrix calculation for $1000 \cdot k$ points.

1: *Generate $n = 1000$ random points in 2D space: $P = \{(x_i, y_i) \mid i = 1, 2, \dots, n\}$, where $x_i, y_i \sim \text{Uniform}(0, 1)$
2: Compute the distance matrix $D \in \mathbb{R}^{n \times n}$, where each element is given by:

$$D_{j,k} = \sqrt{(x_j - x_k)^2 + (y_j - y_k)^2} \quad \text{for } j, k = 1, 2, \dots, n$$

ALGORITHM 13 | Hierarchical clustering on $100 \cdot k$ points.

1: *Generate $n = 100k$ random points in 2D space: $P = \{(x_i, y_i) \mid i = 1, 2, \dots, n\}$, where $x_i, y_i \sim \text{Uniform}(0, 1)$
2: *Initialise each point as its own cluster: $\text{Clusters} \leftarrow \{\{1\}, \{2\}, \dots, \{n\}\}$
3: **while** the number of clusters >1 **do**
4: Set min_dist $\leftarrow \infty$ and merge_indices $\leftarrow (0, 1)$
5: **for** each pair of clusters (C_j, C_k) **do**
6: Compute the centroids:

$$\text{Centroid}(C_j) = \frac{1}{|C_j|} \sum_{i \in C_j} P_i, \quad \text{Centroid}(C_k) = \frac{1}{|C_k|} \sum_{i \in C_k} P_i$$

7: Compute the Euclidean distance between centroids:

$$d_{j,k} = \sqrt{\sum_{l=1}^{2} \left(\text{Centroid}_j^l - \text{Centroid}_k^l\right)^2}$$

8: **if** $d_{j,k}$<min_dist **then**
9: Update min_dist $\leftarrow d_{j,k}$ and merge_indices $\leftarrow (j, k)$
10: **end if**
11: **end for**
12: Merge the closest clusters:

$$C_{\text{merge_indices[0]}} \leftarrow C_{\text{merge_indices[0]}} \cup C_{\text{merge_indices[1]}}$$

13: Remove the second cluster:

$$\text{Clusters} \leftarrow \text{Clusters} \setminus C_{\text{merge_indices[1]}}$$

14: **end while**

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

ALGORITHM 14 | Kriging interpolation for $1000 \cdot k$ points.

1: *Generate $n = 1000 \cdot k$ random spatial coordinates: Coords $= \{(x_i, y_i) \mid i = 1, 2, \dots, n\}$, where $x_i, y_i \sim \text{Uniform}(0, 1)$
2: *Generate random values $v_i \sim \text{Uniform}(0, 1)$ at each coordinate
3: *Define a 50×50 grid: $G = \{(g_x, g_y) \mid g_x, g_y \in [0, 1], g_x = \frac{i}{49}, g_y = \frac{j}{49}, i, j \in \{0, 1, \dots, 49\}\}$
4: *Define (spherical) variogram model:

$$\gamma(h; \text{sill}, \text{range}) = \begin{cases} \text{sill} \cdot \left(1.5 \cdot \frac{h}{\text{range}} - 0.5 \cdot \left(\frac{h}{\text{range}}\right)^3\right), & \text{if } 0 < h \leq \text{range} \\ \text{sill}, & \text{if } h > \text{range} \end{cases}$$

5: **for** each grid point $g \in G$ **do**
6: Compute distances: $d_i = \sqrt{(x_i - g_x)^2 + (y_i - g_y)^2}$ for $i = 1, 2, \dots, n$
7: Apply variogram model: $w_i = \text{variogram_model}(d_i, \text{sill} = 1.0, \text{range} = 0.5)$
8: Predict value at g:

$$v_g = \frac{\sum_{i=1}^n w_i v_i}{\sum_{i=1}^n w_i}$$

9: **end for**

ALGORITHM 15 | Kernel density estimation for $1000 \cdot k$ points.

1: *Generate $n = 1000 \cdot k$ random spatial coordinates: Data $= \{(x_i, y_i) \mid i = 1, 2, \dots, n\}$, where $x_i, y_i \sim \text{Uniform}(0, 1)$
2: *Set bandwidth parameter $h = 0.1$
3: **for** each point $(x_j, y_j) \in$ Data **do**
4: Compute distances to all other points:

$$d_i = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \quad \text{for } i = 1, 2, \dots, n$$

5: Apply kernel density function:

$$K(d_i, h) = \frac{1}{\sqrt{2\pi}h} \exp\left(-\frac{1}{2}\left(\frac{d_i}{h}\right)^2\right)$$

6: Compute density estimate at (x_j, y_j):

$$f_j = \sum_{i=1}^n K(d_i, h)$$

7: **end for**

I/O Operations

ALGORITHM 16 | Writing a 1,000,000-row CSV file.

1: *Generate a dataset $D \in \mathbb{R}^{1,000,000 \times 2}$ with independent random values uniformly distributed in $[0, 1]$
2: Write the dataset to a CSV file:

Save D as `large_test_file.csv` with comma delimiters

ALGORITHM 17 | Reading a 1,000,000-row CSV file.

1: Open and Read the CSV file `large_test_file.csv` containing 1,000,000 rows and 2 columns into memory as a dataset D

ALGORITHM 18 | Writing a large binary file.

1: *Generate a dataset $D \in \mathbb{R}^{1,000,000}$ with independent random values uniformly distributed in $[0, 1]$
2: Write the dataset to a binary file:

Save D as `large_test_file.bin`

ALGORITHM 19 | Reading a large binary file.

1: Read the binary file `large_test_file.bin` containing 1,000,000 floating-point numbers into a dataset D assuming the data type is `float64`

ALGORITHM 20 | Random access of 10 lines from a large text file.

1: *Create a text file `large_text_file.txt` with 1,000,000 identical lines: `"This is a line of text"`
2: Open the text file `large_text_file.txt`
3: **for** each specified line index in $\{1, 10, 100, 1000, 10000\}$ **do**
4: Reset the file pointer to the beginning: `seek(0)`
5: Read lines sequentially up to the desired line index
6: Store the content of the desired line
7: **end for**
8: Close the text file

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Appendix B

Specification of Functions Used in Benchmark Tasks

TABLE B1 | Main R functions used to implement benchmark tasks involving matrix operations, statistical methods, and file handling.

Task	R
1.1 Creation, transpose, deformation of a (2500·k) × (2500·k) matrix	rnorm(), t(), dim()
1.2 Matrix exponentiation (element-wise)	a ^ 10
1.3 Sorting values	sort(. . . , method="quick")
1.4 Cross-product	crossprod()
1.5 Determinant	det()
2.1 FFT	fft()
2.2 Eigenvalues	eigen(. . .)
2.3 Linear regression	solve(crossprod(a), crossprod(a, b))
2.4 Cholesky decomposition	chol()
2.5 Matrix inverse	solve()
3.1 Temporal autocorrelation	—
3.2 Distance matrix	—
3.3 Hierarchical clustering	—
3.4 Kriging interpolation	—
3.5 Kernel density estimation	—
4.1 Write CSV	write.csv()
4.2 Read CSV	read.csv()
4.3 Write binary file	writeBin()
4.4 Read binary file	readBin()
4.5 Random access in text file	file(), seek(), readLines()

Note: A dash (—) denotes tasks that do not rely on specific built-in R functions within the tested framework.

TABLE B2 | Main C++ functions used to implement benchmark tasks involving matrix operations, statistical methods, and file handling.

Task	Rcpp (C++)
1.1 Creation, transpose, deformation of a (2500·k) × (2500·k) matrix	Eigen::MatrixXd::Random(), .transpose(), Eigen::Map()
1.2 Matrix exponentiation (element-wise)	.array().pow(10)
1.3 Sorting values	std::sort()
1.4 Cross-product	A.transpose() * A
1.5 Determinant	mat.determinant()
2.1 FFT	fftw_plan_dft_1d(), fftw_execute()
2.2 Eigenvalues	SelfAdjointEigenSolver::eigenvalues()
2.3 Linear regression	LDLT().solve(A.transpose() * b)
2.4 Cholesky decomposition	LLT().matrixL()
2.5 Matrix inverse	mat.inverse()
3.1 Temporal autocorrelation	—
3.2 Distance matrix	—
3.3 Hierarchical clustering	—
3.4 Kriging interpolation	—
3.5 Kernel density estimation	—
4.1 Write CSV	ofstream
4.2 Read CSV	ifstream, getline(), istringstream
4.3 Write binary file	ofstream.write()
4.4 Read binary file	ifstream.read()
4.5 Random access in text file	ifstream, getline(), vector<string>

Note: A dash (—) denotes tasks that do not rely on specific built-in C++ functions within the tested framework.

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE B3 | Main MATLAB functions used to implement benchmark tasks involving matrix operations, statistical methods, and file handling.

Task	MATLAB
1.1 Creation, transpose, deformation of a (2500·k) × (2500·k) matrix	randn(), reshape(), transpose
1.2 Matrix exponentiation (element-wise)	A.^{10}
1.3 Sorting values	sort()
1.4 Cross-product	a'*a
1.5 Determinant	det()
2.1 FFT	fft()
2.2 Eigenvalues	eig()
2.3 Linear regression	(A'*A)\(A'*b)
2.4 Cholesky decomposition	chol(A*A')
2.5 Matrix inverse	A\eye()
3.1 Temporal autocorrelation	—
3.2 Distance matrix	—
3.3 Hierarchical clustering	—
3.4 Kriging interpolation	—
3.5 Kernel density estimation	—
4.1 Write CSV	writematrix()
4.2 Read CSV	readmatrix()
4.3 Write binary file	fwrite()
4.4 Read binary file	fread()
4.5 Random access in text file	frewind(), fgetl()

Note: A dash (—) denotes tasks that do not rely on specific built-in MATLAB functions within the tested framework.

TABLE B4 | Main Python NumPy, SciPy and pandas functions used to implement benchmark tasks involving matrix operations, statistical methods, and file handling.

Task	Python
1.1 Creation, transpose, deformation of a (2500·k) × (2500 · k) matrix	numpy.random.randn(), numpy.reshape(), .T
1.2 Matrix exponentiation (element-wise)	numpy.power()
1.3 Sorting values	numpy.sort()
1.4 Cross-product	numpy.dot(a.T, a)
1.5 Determinant	numpy.linalg.det()
2.1 FFT	scipy.fft.fft()
2.2 Eigenvalues	numpy.linalg.eigh()
2.3 Linear regression	scipy.linalg.solve()
2.4 Cholesky decomposition	numpy.linalg.cholesky()
2.5 Matrix inverse	numpy.linalg.solve()
3.1 Temporal autocorrelation	—
3.2 Distance matrix	—
3.3 Hierarchical clustering	—
3.4 Kriging interpolation	—
3.5 Kernel density estimation	—
4.1 Write CSV	numpy.savetxt()
4.2 Read CSV	pandas.read_csv()
4.3 Write binary file	array.tofile()
4.4 Read binary file	numpy.fromfile()
4.5 Random access in text file	file.readline() loop after seek()

Note: A dash (—) denotes tasks that do not rely on specific built-in Python functions within the tested framework.

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE B5 | Main `Julia` functions used to implement benchmark tasks involving matrix operations, statistical methods, and file handling.

Task	Julia
1.1 Creation, transpose, deformation of a (2500·k) × (2500·k) matrix	randn(), reshape(), transpose()
1.2 Matrix exponentiation (element-wise)	$A.^{10}$
1.3 Sorting values	sort!()
1.4 Cross-product	A' * A
1.5 Determinant	det()
2.1 FFT	FFTW.fft()
2.2 Eigenvalues	eigen().values
2.3 Linear regression	A \ b
2.4 Cholesky decomposition	cholesky(Symmetric(A*A'))
2.5 Matrix inverse	inv()
3.1 Temporal autocorrelation	—
3.2 Distance matrix	—
3.3 Hierarchical clustering	—
3.4 Kriging interpolation	—
3.5 Kernel density estimation	—
4.1 Write CSV	open(. . . , "w") + println()
4.2 Read CSV	open(. . . , "r") + readline()
4.3 Write binary file	write()
4.4 Read binary file	read!()
4.5 Random access in text file	seekstart(), readline() loop

Note: A dash (—) denotes tasks that do not rely on specific built-in `Julia` functions within the tested framework.

Appendix C

Simulation Results

TABLE C1 | Mean execution times (in seconds) for each task across different programming environments, computed over 100 trials. Tasks are grouped into four categories: Matrix Manipulations, Matrix Transformations, Programming Tasks, and I/O Operations. Category averages are calculated as the arithmetic mean of the tasks within each category, and the overall average is the mean of these category-level averages. The results shown correspond to scaling parameter $k = 1$.

Task	R BLAS	R MKL	R OpenBLAS	Rcpp	MATLAB	Python	Julia
1.1 Creation, transpose, and deformation of a (2500·k) × (2500·k) matrix	0.269	0.268	0.269	0.202	0.070	0.106	0.043
1.2 Exponentiation of a (2500·k) × (2500·k) matrix to the power of 10	0.091	0.091	0.092	0.191	0.008	0.018	0.055
1.3 Sorting of k·1,000,000 random values	0.077	0.077	0.077	0.067	0.027	0.007	0.031
1.4 Cross-product of a (2500·k) × (2500·k) matrix	4.206	0.038	0.034	1.896	0.030	0.033	0.182
1.5 Determinant of a (2500·k) × (2500·k) random matrix	2.390	0.111	0.044	0.708	0.040	0.118	0.152
2.1 FFT over k·1,000,000 random values	0.035	0.034	0.035	0.019	0.008	0.014	0.017
2.2 Eigenvalues of a (500·k) × (500·k) random matrix	0.234	0.110	0.325	0.126	0.102	0.040	0.319
2.3 Linear regression over a (2500·k) × (2500·k) matrix	5.577	0.096	0.073	2.673	0.088	0.173	0.159
2.4 Cholesky decomposition of a (2500·k) × (2500·k) matrix	1.359	0.043	0.028	0.396	0.013	0.041	0.040
2.5 Inverse of a (1000·k) × (1000·k) random matrix	0.479	0.066	0.089	0.228	0.028	0.052	0.088
3.1 Temporal autocorrelation calculation for k·1,000,000 data points	0.146	0.146	0.161	0.026	0.028	0.031	0.106
3.2 Distance matrix calculation for 1000·k points	0.106	0.106	0.108	0.003	0.004	1.028	0.002
3.3 Hierarchical clustering on 100·k points	1.567	1.562	1.563	0.171	0.180	2.601	0.127
3.4 Kriging interpolation for 1000·k points	0.040	0.040	0.040	0.047	0.120	0.050	0.008
3.5 Kernel density estimation for 1000·k points	0.015	0.015	0.015	0.009	0.020	0.013	0.015
4.1 Writing a 1,000,000-row CSV file	0.888	0.889	0.891	0.443	0.680	1.379	0.455
4.2 Reading a 1,000,000-row CSV file	0.896	0.892	0.894	0.254	0.651	0.262	0.047
4.3 Writing a large binary file	0.085	0.084	0.084	0.004	0.004	0.004	0.005
4.4 Reading a large binary file	0.024	0.023	0.023	0.001	0.001	0.001	0.001
4.5 Random access of 10 lines from a large text file	0.010	0.010	0.010	0.015	0.016	0.001	0.000

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE C2 | Relative efficiency for each programming environment with respect to R BLAS, serving as the baseline (i.e., a value of 1 corresponds to R BLAS performance). Each entry indicates how many times faster (or slower) a method is compared to R BLAS for the corresponding task. Category and overall averages are calculated using arithmetic means. The results shown correspond to scaling parameter $k = 1$.

Task	R MKL	R OpenBLAS	Rcpp	MATLAB	Python	Julia
1.1 Creation, transpose, and deformation of a (2500·k) × (2500·k) matrix	1	1	1	4	3	6
1.2 Exponentiation of a (2500·k) × (2500·k) matrix to the power of 10	1	1	0	12	5	2
1.3 Sorting of k·1,000,000 random values	1	1	1	3	11	2
1.4 Cross-product of a (2500·k) × (2500·k) matrix	112	124	2	140	127	23
1.5 Determinant of a (2500·k) × (2500·k) random matrix	22	54	3	59	20	16
2.1 FFT over k·1,000,000 random values	1	1	2	5	3	2
2.2 Eigenvalues of a (500·k) × (500·k) random matrix	2	1	2	2	6	1
2.3 Linear regression over a (2500·k) × (2500·k) matrix	58	77	2	64	32	35
2.4 Cholesky decomposition of a (2500·k) × (2500·k) matrix	32	48	3	103	33	34
2.5 Inverse of a (1000·k) × (1000·k) random matrix	7	5	2	17	9	5
3.1 Temporal autocorrelation calculation for k·1,000,000 data points	1	1	6	5	5	1
3.2 Distance matrix calculation for 1000·k points	1	1	38	26	0	48
3.3 Hierarchical clustering on 100·k points	1	1	9	9	1	12
3.4 Kriging interpolation for 1000·k points	1	1	1	0	1	5
3.5 Kernel density estimation for 1000·k points	1	1	2	1	1	1
4.1 Writing a 1,000,000-row CSV file	1	1	2	1	1	2
4.2 Reading a 1,000,000-row CSV file	1	1	4	1	3	19
4.3 Writing a large binary file	1	1	23	24	23	16
4.4 Reading a large binary file	1	1	33	19	36	30
4.5 Random access of 10 lines from a large text file	1	1	1	1	12	19

TABLE C3 | Mean execution times (in seconds) for each task across different programming environments, computed over 100 trials. Tasks are grouped into four categories: Matrix Manipulations, Matrix Transformations, Programming Tasks, and I/O Operations. Category averages are calculated as the arithmetic mean of the tasks within each category, and the overall average is the mean of these category-level averages. Results shown correspond to scaling parameter $k = 2$.

Task	R BLAS	R MKL	R OpenBLAS	Rcpp	MATLAB	Python	Julia
1.1 Creation, transpose, and deformation of a (2500·k) × (2500·k) matrix	1,182	1,182	1,182	0,888	0,277	0,456	0,179
1.2 Exponentiation of a (2500·k) × (2500·k) matrix to the power of 10	0,366	0,366	0,366	0,764	0,027	0,072	0,220
1.3 Sorting of k·1,000,000 random values	0,162	0,162	0,162	0,141	0,050	0,015	0,058
1.4 Cross-product of a (2500·k) × (2500·k) matrix	41,377	0,167	0,164	14,946	0,152	0,155	1,306
1.5 Determinant of a (2500·k) × (2500·k) random matrix	20,261	0,352	0,167	5,393	0,174	0,354	0,380
2.1 FFT over k·1,000,000 random values	0,082	0,082	0,084	0,041	0,012	0,031	0,042
2.2 Eigenvalues of a (500·k) × (500·k) random matrix	1,562	0,466	1,322	1,038	0,429	0,206	0,888
2.3 Linear regression over a (2500·k) × (2500·k) matrix	52,383	0,377	0,308	21,603	0,405	0,747	0,419
2.4 Cholesky decomposition of a (2500·k) × (2500·k) matrix	11,148	0,153	0,132	2,863	0,036	0,163	0,163
2.5 Inverse of a (1000·k) × (1000·k) random matrix	3,730	0,164	0,163	1,779	0,061	0,162	0,200
3.1 Temporal autocorrelation calculation for k·1,000,000 data points	0,308	0,312	0,310	0,060	0,043	0,066	0,230
3.2 Distance matrix calculation for 1000·k points	0,453	0,444	0,438	0,023	0,023	4,174	0,011
3.3 Hierarchical clustering on 100·k points	12,421	12,301	12,426	1,364	1,447	21,019	0,982
3.4 Kriging interpolation for 1000·k points	0,048	0,048	0,048	0,098	0,196	0,071	0,013
3.5 Kernel density estimation for 1000·k points	0,055	0,056	0,056	0,035	0,057	0,036	0,045
4.1 Writing a 1,000,000-row CSV file	0,890	0,886	0,890	0,443	0,675	1,383	0,479
4.2 Reading a 1,000,000-row CSV file	0,892	0,896	0,899	0,254	0,653	0,269	0,047
4.3 Writing a large binary file	0,086	0,082	0,084	0,004	0,004	0,004	0,005
4.4 Reading a large binary file	0,023	0,025	0,022	0,001	0,001	0,001	0,001
4.5 Random access of 10 lines from a large text file	0,010	0,010	0,010	0,015	0,017	0,001	0,000

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE C4 | Relative efficiency for each programming environment with respect to R BLAS, serving as the baseline (i.e., a value of 1 corresponds to R BLAS performance). Each entry indicates how many times faster (or slower) a method is compared to R BLAS for the corresponding task. Category and overall averages are calculated using arithmetic means. Results shown correspond to scaling parameter $k = 2$.

Task	R MKL	R OpenBLAS	Rcpp	MATLAB	Python	Julia
1.1 Creation, transpose, and deformation of a (2500·k) × (2500·k) matrix	1	1	1	4	3	7
1.2 Exponentiation of a (2500·k) × (2500·k) matrix to the power of 10	1	1	0	13	5	2
1.3 Sorting of k·1,000,000 random values	1	1	1	3	11	3
1.4 Cross-product of a (2500·k) × (2500·k) matrix	247	252	3	273	267	32
1.5 Determinant of a (2500·k) × (2500·k) random matrix	58	121	4	116	57	53
2.1 FFT over k·1,000,000 random values	1	1	2	7	3	2
2.2 Eigenvalues of a (500·k) × (500·k) random matrix	3	1	2	4	8	2
2.3 Linear regression over a (2500·k) × (2500·k) matrix	139	170	2	129	70	125
2.4 Cholesky decomposition of a (2500·k) × (2500·k) matrix	73	84	4	307	68	68
2.5 Inverse of a (1000·k) × (1000·k) random matrix	23	23	2	61	23	19
3.1 Temporal autocorrelation calculation for k·1,000,000 data points	1	1	5	7	5	1
3.2 Distance matrix calculation for 1000·k points	1	1	19	20	0	40
3.3 Hierarchical clustering on 100·k points	1	1	9	9	1	13
3.4 Kriging interpolation for 1000·k points	1	1	0	0	1	4
3.5 Kernel density estimation for 1000·k points	1	1	2	1	2	1
4.1 Writing a 1,000,000-row CSV file	1	1	2	1	1	2
4.2 Reading a 1,000,000-row CSV file	1	1	4	1	3	19
4.3 Writing a large binary file	1	1	23	22	23	16
4.4 Reading a large binary file	1	1	32	19	32	30
4.5 Random access of 10 lines from a large text file	1	1	1	1	11	20

1099095x, 2025, 5, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/env.70017 by Universita Di Bergamo Servizi, Wiley Online Library on [06/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License