ACTA SCIENTIFIC COMPUTER SCIENCES

Volume 7 Issue 8 November 2025

Review Article

SiPy 0.8.0 on the Three Legs of Python, R, and Julia

Maurice HT Ling[1,2]*

[1]*HOHY PTE LTD, Singapore*

[2]*Newcastle Australia Institute of Higher Education, University of Newcastle, Australia*

***Corresponding Author:** Maurice HT Ling, HOHY PTE LTD, Singapore and Newcastle Australia Institute of Higher Education, University of Newcastle, Australia.

DOI: 10.31080/ASCS.2025.07.0596

Received: January 16, 2026

Published: February 13, 2026

Abstract

Python and R dominate contemporary scientific data analysis due to their mature ecosystems and extensive methodological support, yet performance limitations often arise in computation-intensive scenarios, leading to fragmented multi-language workflows. SiPy is a lightweight statistical interface written in Python that addresses this challenge by explicitly coordinating multiple languages while preserving clear execution boundaries. Building on earlier versions that integrated R as a statistical backend, this article reports SiPy 0.8.0 (released on 09 January 2026), which extends the framework to incorporate Julia as a high-performance computational engine and formalises script-level execution across Python, R, and Julia using a uniform subprocess-based model. In this three-legged architecture, Python functions both as the primary orchestration layer and as a scriptable computational backend, R provides rigorously validated statistical methods, and Julia supports performance-critical numerical computation and simulation. The system architecture underlying this design is described, including environment isolation, script-based execution, and conservative data exchange strategies that prioritise reproducibility and portability.

Keywords: Scientific Computing; Statistical Analysis; Python; R; Julia; Multi-Language Workflows; Reproducible Research; High-Performance Computing; Data Analysis Frameworks; Interoperability; External Script Execution

Introduction

Python and R have become the dominant languages in contemporary scientific workflows [1] due to their mature ecosystems, extensive libraries, and strong community support. Python excels as a general-purpose scientific programming language, offering broad interoperability, flexible orchestration, and a rich ecosystem for data manipulation, visualization, and automation [2]. R, in contrast, has established itself as the *de facto* standard for statistical analysis, inference, and methodological development [3], with a depth of rigor and breadth of statistical techniques that remain unmatched in many domains. Together, Python and R underpin a large proportion of modern computational research and reproducible data analysis pipelines [1]. SiPy [4] is a lightweight statistical interface written in Python, and has been demonstrated as a potential platform for incorporating R methods while reducing the learning curve needed to learn R.

Despite these strengths, scientific workflows frequently encounter performance limitations [5] when computational

demands increase, particularly in scenarios involving large-scale simulations, complex numerical solvers, or algorithmically intensive models. In such cases, researchers often face the so-called two-language problem [6]; where high-level languages such as Python or R are used for prototyping and analysis, while performance-critical components must be re-implemented in lower-level languages such as C, C++, or Fortran [7,8]. This separation introduces additional complexity, increases development and maintenance costs, and raises barriers to reproducibility and extensibility.

Julia was explicitly designed to address this challenge by reconciling high-level expressiveness with low-level performance, thereby collapsing the traditional divide between prototyping and execution. As articulated by Bezanson., *et al*. [6], Julia's language design enables developers to write code that is both readable and performant, allowing algorithmic descriptions to remain close to their mathematical formulations without sacrificing computational efficiency. This design philosophy directly targets the two-language problem and offers an alternative paradigm for performance-critical scientific computing. Empirical evidence supports Julia's effectiveness in domains where numerical performance and composability are central. High-performance scientific areas; most notably differential equation solving, simulation frameworks, and large-scale numerical modelling; have reported substantial success with Julia, demonstrating both competitive or superior performance relative to established tools and a high degree of composability across scientific abstractions [9,10]. These reports further highlight Julia's suitability as a computational engine within broader scientific workflows, rather than as a wholesale replacement for existing languages [11]. Importantly, Julia's strengths do not diminish the continued relevance of Python and R; instead, they suggest a complementary relationship in which each language occupies a distinct and well-defined role [12,13]. Python remains well suited for orchestration, workflow control, and integration across heterogeneous tools. R continues to serve as the authoritative environment for statistical modelling, inference, and methodological validation. Julia, meanwhile, provides a natural home for performance-critical numerical computation, algorithm development, and simulation-heavy workloads [13].

Motivated by this perspective, this article reports SiPy 0.8.0 (released on 09 January 2026) as adopting a three-legged architecture that leverages the respective strengths of Python, R, and Julia. In SiPy, Python functions as the primary orchestration layer, coordinating data flow, execution, and user interaction. R is employed for statistical analysis and inference, drawing on its extensive and rigorously validated ecosystem. Julia is incorporated as a high-performance computational backend, enabling efficient execution of numerically intensive tasks while maintaining clarity and expressiveness in algorithmic implementation. By explicitly delineating the roles of each language and integrating them within a coherent framework, SiPy seeks to provide a practical and extensible approach to multi-language scientific computing.

Incorporating Julia into Python/R-based SiPy Version 0.7.0

R has been an important statistical engine in SiPy since version 0.6.0 [4] where clear execution boundaries between languages have been preserved, which is also the basis of improvement from SiPy since version 0.6.0 [4] to SiPy since version 0.7.0 [14]. Rather than embedding interpreters or relying on in-process foreign function interfaces, SiPy adopts a subprocess-based execution model [4]; in which Python serves as the primary orchestration layer, coordinating the invocation of R as external computational backends. This architectural choice prioritises reproducibility, portability, and transparency over tight coupling, and reflects the intended role of SiPy as a coordination and integration layer rather than a monolithic runtime. Incorporation of Julia into SiPy will use the same mode. The main reason for choosing this subprocess-based method over language bridge; such as rpy2 (Python package to call R), and PyJulia (Python package to call Julia) is that subprocess package is native in Python Standard Library and does not require additional installations for subprocess-based method to work. This reduces future maintenance load. Moreover, using language bridge is likely to result in a blurring of boundaries between SiPy, R, and Julia.

At the core of SiPy's architecture is a unidirectional execution flow initiated from Python (Figure 1). Analytical tasks are expressed in Python and, where appropriate, translated into language-specific scripts that are executed in isolated R or Julia subprocesses. Python is responsible for preparing inputs, invoking the appropriate runtime, monitoring execution, and collecting outputs. This execution model provides several advantages. First,

it avoids the complexity and fragility associated with embedding language runtimes or maintaining long-lived cross-language bindings. Second, it ensures that each language executes within its native runtime environment, thereby preserving expected semantics and reducing unexpected side effects. Third, subprocess-based execution makes failure modes explicit and recoverable, as errors and warnings are captured directly from standard output and error streams. The same execution strategy is applied consistently across R and Julia backends, allowing both languages to be treated as interchangeable computational engines from the perspective of the Python orchestration layer.

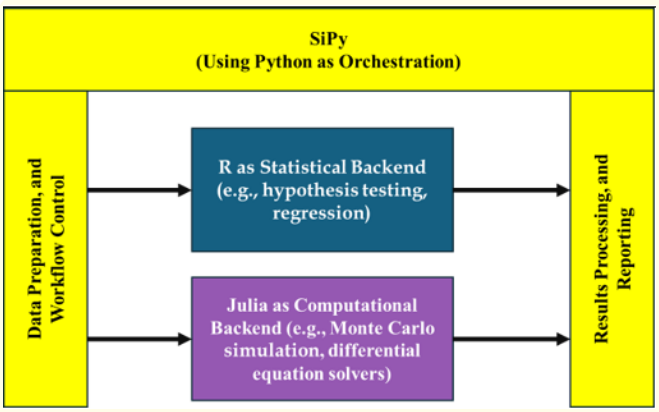

Figure 1: Interoperability Between Python, R, and Julia in SiPy.

To support reproducibility and ease of deployment, SiPy emphasises explicit control over execution environments. Each backend language is invoked using a known executable path and a controlled runtime configuration, avoiding reliance on user-level global installations wherever possible. For R, SiPy supports the use of a portable R distribution bundled alongside the framework, enabling analyses to be executed in a self-contained environment with a known set of packages. For Julia, SiPy adopts a similar approach by invoking a locally bundled Julia binary in conjunction with an isolated project environment and package depot. In both cases, environment variables and runtime flags are used to prevent leakage into user-specific configuration files or global package directories. This approach allows SiPy workflows to be distributed, reproduced, and executed across systems with minimal external dependencies, while remaining agnostic to operating system–specific package managers.

SiPy communicates with R and Julia through dynamically generated scripts rather than direct function calls. These scripts encode the analytical intent expressed in Python and are executed as standalone units within their respective runtimes. Inputs are passed through explicitly defined files or arguments, and outputs are written to structured data formats such as plain text, CSV, or JSON. By enforcing explicit language boundaries, SiPy avoids implicit state sharing across languages and reduces the cognitive load associated with debugging multi-language workflows. This design choice also aligns with established best practices in reproducible research, where computational steps are expected to be inspectable, re-executable, and auditable.

Data exchange between Python, R, and Julia in SiPy is intentionally conservative. Rather than attempting to share in-memory objects or complex language-specific data structures, SiPy relies on serialised representations that are widely supported across languages. Tabular data, model outputs, and summary statistics are exchanged using standard formats that prioritise transparency and long-term stability. While this approach incurs modest practical overhead relative to in-process communication (see Appendix A, B), it offers clear benefits in terms of robustness, reproducibility, and ease of inspection. Moreover, because the most computationally intensive operations are delegated to the backend languages, the overhead associated with data serialisation does not typically dominate overall execution time.

The architectural decisions underlying SiPy reflect a deliberate trade-off between tight integration and long-term sustainability. By avoiding deep interlanguage coupling (such as the use of language bridge), SiPy reduces maintenance burden and minimises sensitivity to changes in any individual language's internal APIs. At the same time, by assigning well-defined roles to Python, R, and Julia, the framework leverages the strengths of each language without forcing convergence onto a single programming model. In this sense, SiPy's architecture is not intended to abstract away language differences entirely, but rather to make them explicit and manageable. This design supports both methodological development and pedagogical clarity, allowing users to reason about where and why a particular language is employed within a given workflow.

Appendix A: Comparison Execution Time Between SiPy and R

A preliminary benchmark was performed using the same set of data to execute linear regression (command = rregress lm data=df y=yN x=all) within SiPy and from the generated R script. The operations for SiPy is as follows:

```
let yN be clist 1.2, 2.3, 3.1, 4.8, 5.6, 6.2, 7.9, 8.4, 9.7, 10.5

let yB be dlist 1, 0, 1, 0, 1, 0, 1, 1, 0, 1

let yC be slist A, B, C, A, B, C, A, B, C, A

let x1 be clist 2, 3, 5, 7, 11, 13, 17, 19, 23, 29

let x2 be clist 1, 4, 9, 16, 25, 36, 49, 64, 81, 100

let x3 be clist 5, 8, 6, 10, 12, 14, 18, 20, 24, 30

let x4 be clist 3.1, 5.2, 2.7, 8.6, 9.1, 4.4, 7.8, 6.5, 10.2, 11.3

let x5 be clist 100, 90, 80, 70, 60, 50, 40, 30, 20, 10

let df be dataframe yN:yN yB:yB yC:yC x1:x1 x2:x2 x3:x3 x4:x4 x5:x5

rregress lm data=df y=yN x=all
```

And this is the generated R script:

```
data <- read.csv("data_ffce36e7.csv")

model <- lm(yN ~ yB + yC + x1 + x2 + x3 + x4 + x5, data=data)

summary(model)
```

The linear regression was performed 10 times and the summary results (Table A1) shows that SiPy averaged 27 times slower than the native R code (p-value of 8.32E-14 for 2-samples t-test with unequal variance). The most likely reason is the overhead in code generation and disk operations. Despite so, execution time of about 1 second in SiPy is still practically acceptable.

Table A1. Execution Time Between SiPy and R.

Statistic in Seconds	SiPy	R
Minimum	0.7850	0.0287
Maximum	0.8803	0.0323
Median	0.8274	0.0306
Mean	0.8304	0.0303
Standard Deviation	0.03468	0.00102

Appendix B: Executing Shell-Based Commands

SiPy supports the direct execution of arbitrary shell-based commands using a string-based interface that preserves native shell semantics, including pipes, redirection, and command chaining (Figure A1). Any command that can be executed from the system command line can therefore be invoked unchanged from within SiPy. Functionally, this mechanism generalises SiPy's execution model beyond language runtimes. The operating system shell itself becomes a backend, allowing external programs, utilities, and workflows to be integrated without additional wrappers or plugins. This includes file system operations, data preprocessing steps, environment inspection, and the invocation of third-party tools that fall outside Python, R, or Julia.

The primary implication of this design is that extensibility in SiPy is no longer constrained by supported languages or APIs. Methodological completeness is achieved through execution rather than abstraction: if a tool can be run from the command line, it can participate in a SiPy workflow. This further reduces the need for bespoke integrations while keeping execution boundaries explicit, inspectable, and reproducible. Shell-based execution is intended for advanced usage and aligns with SiPy's broader philosophy of treating scripts and commands as first-class computational artefacts rather than embedding functionality within the framework itself.

```
SiPy: 1 >>> execute shell command="dir /p | findstr "sipy*""
Command to run: dir /p | findstr "sipy*"
Directory of C:\Dropbox\MyProjects\sipy
18/09/2025  10:32 pm             8,131 conda_sipy_environment.txt
16/12/2025  07:27 pm    <DIR>          libsipy
18/09/2025  10:32 pm             1,405 pip_sipy_environment.txt
18/12/2025  11:03 pm           221,880 sipy.py
08/10/2022  12:23 am             1,395 sipy_CLI.py
12/12/2025  01:44 pm             2,369 sipy_info.py
04/04/2025  04:17 pm    <DIR>          sipy_plugins
04/05/2025  12:47 pm             3,837 sipy_pm.py

SiPy: 2 >>> .dir /p | findstr "sipy*"
Command to run: dir /p | findstr "sipy*"
Directory of C:\Dropbox\MyProjects\sipy
18/09/2025  10:32 pm             8,131 conda_sipy_environment.txt
16/12/2025  07:27 pm    <DIR>          libsipy
18/09/2025  10:32 pm             1,405 pip_sipy_environment.txt
18/12/2025  11:03 pm           221,880 sipy.py
08/10/2022  12:23 am             1,395 sipy_CLI.py
12/12/2025  01:44 pm             2,369 sipy_info.py
04/04/2025  04:17 pm    <DIR>          sipy_plugins
04/05/2025  12:47 pm             3,837 sipy_pm.py

SiPy: 3 >>>
```

Figure A1: Executing Shell-Based Commands.

Testing Julia and R in SiPy

Two tests were performed to demonstrate that Julia is callable from SiPy. The first test is on linear regression. Given that the dependent variable, yN, is {1.2, 2.3, 3.1, 4.8, 5.6, 6.2, 7.9, 8.4, 9.7, 10.5}; and the independent variables, x1 and x2, to be {2.0, 3.0, 5.0, 7.0, 11.0, 13.0, 17.0, 19.0, 23.0, 29.0} and {1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0, 100.0} respectively; both R and Julia were asked to run a linear regression with the model, yN = b1(x1) + b2(x2) + b0. The test shows that both R and Julia gave the same results (Figure 2) as yN = 0.608(x1) – 0.070(x2) + 0.828 with

R-square of 0.968; thereby, demonstrating that Julia is callable from SiPy. Similarly, R-based and Julia-based linear regression can also be called if data is in a file (Figure 3).

The second test is to execute external R and Julia scripts from within SiPy (Figure 4). In this test, a data file in the form of comma-delimited file was used to perform linear regression using both R and Julia. Differing from the first test where the codes for linear regression were implemented within SiPy, this test assumes that the R or Julia operations were provided as a script – r_lm.R and julia_lm.jl respectively; where the scripts will be called from within SiPy, and parameters / arguments are routed from SiPy into the scripts. The test shows that the script execution is successful. This suggests that users can link up R and Julia scripts for execution within SiPy.

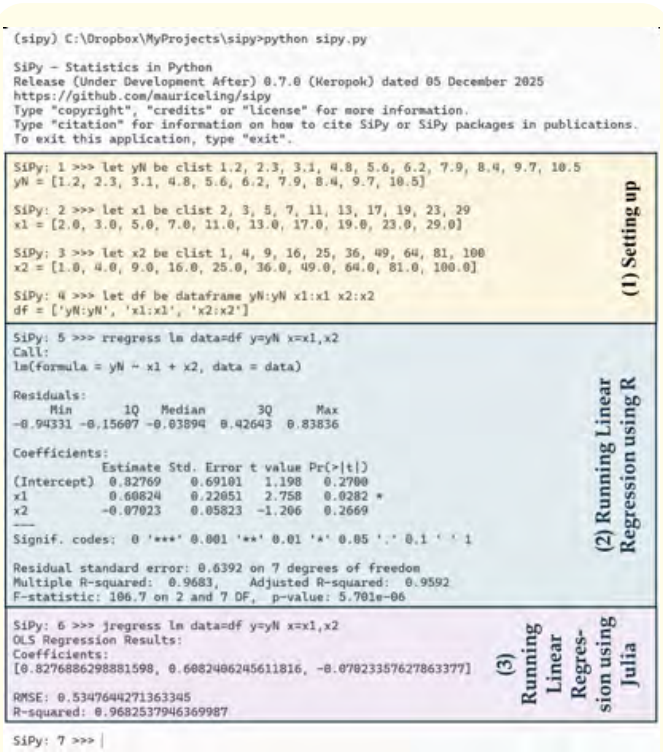

Figure 2: Running Linear Regression on R and Julia Using Identical Data Values.

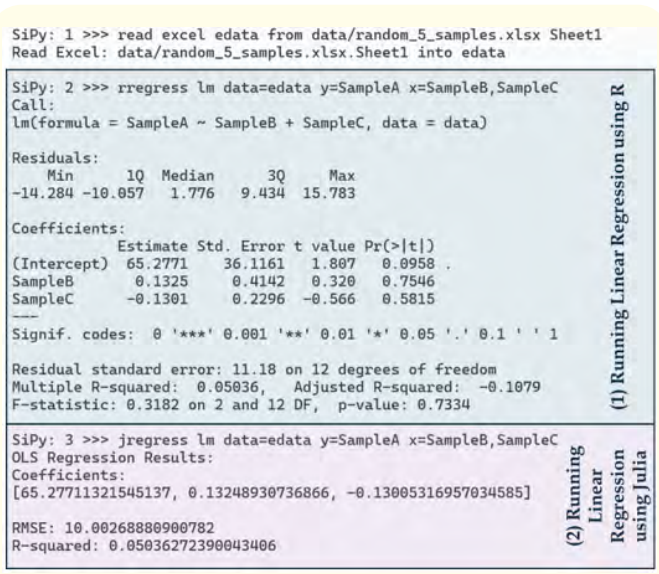

Figure 3: Running Linear Regression on R and Julia Using Identical Data File.

Figure 4: Running External R and Julia Scripts on Identical Data File.

Script-Level Extensibility and User-Defined Backends

A central design principle of SiPy is that extensibility is achieved at the level of executable scripts rather than through exhaustive API wrapping or language-level bindings. SiPy does not attempt to re-expose the full functionality of R or Julia within Python, nor does it seek to mirror the evolving APIs of these languages. Instead, SiPy provides a controlled execution framework in which user-authored Python, R, or Julia scripts are treated as first-class computational backends. This approach explicitly shifts the extension mechanism from the SiPy source code to the user's analytical scripts, allowing the framework to remain lightweight while retaining access to the full expressive power of each backend language.

Under this model; any valid Python (Figure 5), R, or Julia script that can be executed from the command line can also be executed from within SiPy. Users are free to write scripts that implement bespoke statistical models, numerical solvers, simulation studies, or experimental methods using native language idioms and ecosystem-specific packages. SiPy's responsibility is limited to preparing inputs, invoking the appropriate runtime, and collecting outputs in a structured and reproducible manner. As a result, new methods can be incorporated into SiPy workflows without requiring changes to the SiPy codebase, avoiding the maintenance burden and conceptual complexity associated with function-by-function wrapping.

This script-level extensibility has several practical advantages. First, it eliminates the need for SiPy to anticipate or support every conceivable statistical or numerical method. As the backend languages remain fully accessible, methodological completeness is achieved through execution rather than abstraction. Second, it preserves transparency and auditability: backend scripts are explicit, inspectable artefacts that can be version-controlled, shared, and executed independently of SiPy. Third, it respects disciplinary expertise, allowing statisticians, data analysts, and numerical modellers to work directly in Python, R, or Julia without adapting their workflows to a single, unified programming interface.

User-defined backends also provide a natural pathway for experimentation and method development. Prototype implementations can be written rapidly as standalone scripts and integrated into larger workflows through SiPy's orchestration layer. Performance-critical components can be iterated on independently in Julia, while statistical validation and benchmarking can be carried out in R. Python scripts can likewise be executed as external backends, enabling symmetry between orchestration and computation and allowing complex preprocessing or auxiliary analyses to be isolated as reproducible execution units. SiPy coordinates these components without imposing a unified programming model, making language boundaries explicit rather than implicit.

By elevating script execution across Python, R, and Julia to a first-class capability, SiPy avoids the common pitfalls of tightly coupled multi-language systems while remaining extensible, maintainable, and pedagogically clear. This design choice reinforces SiPy's role as a coordination framework for statistical and data analysis rather than as a monolithic analytical environment, and it underpins the scalability of the three-legged architecture as additional scriptable computational backends are incorporated in the future.

```
SiPy: 1 >>> execute python example_scripts\\python_lm.py inputfile=example_script
s\\lm_data.csv response=yN predictors="x1,x2"
Command to run: C:\Users\mauri\anaconda3\envs\sipy\python.exe C:\Dropbox\MyProjec
ts\sipy\example_scripts\python_lm.py --inputfile example_scripts\\lm_data.csv --r
esponse yN --predictors "x1,x2"
OLS Regression Results:
Response: yN
Predictors: x1, x2
Coefficients:
[ 0.82768863  0.60824062 -0.07023358]

RMSE: 0.5347644271363345
R-squared: 0.9682537946369987
```

Figure 5: Running External Python Script.

Using SiPy with R and Julia as Backend

SiPy's three-legged architecture enables practical workflows in which Python orchestrates data processing and visualization while R and Julia provide specialized computational support (Figure 1). The following five examples illustrate common patterns relevant to statistical and data analysis.

Firstly, in clinical survival analysis with simulation-based inference [15] where patient datasets can be pre-processed and cleaned in Python, with missing values imputed and covariates transformed. R is then used to fit survival models or Cox proportional hazards regressions, taking advantage of its extensive statistical packages. For uncertainty estimation or bootstrap resampling, computationally intensive simulations can be executed in Julia, allowing rapid assessment of model stability using simulations and prediction intervals. Results are reintegrated into Python for visualization and reporting, producing a fully reproducible

analytical pipeline. Secondly, in ecological population modelling [16,17] where sensor data from field studies can be aggregated and filtered in Python. R performs species distribution modelling or generalized linear mixed models to evaluate environmental and ecological covariates. Julia executes large-scale simulations, such as stochastic population models or differential equation–based ecosystem projections, which would be computationally prohibitive in pure Python or R. SiPy manages the seamless transfer of intermediate results and ensures traceable, scriptable execution. Thirdly, in economic survey data analysis [18–20] where survey responses are imported, cleaned, and weighted in Python. R is used for survey-weighted regressions, hypothesis testing, and inferential statistics. Julia performs Monte Carlo simulations to quantify uncertainty in policy impact estimates or to explore alternative economic scenarios. By centralizing control in Python, analysts can iterate on preprocessing or model parameters without manually coordinating multiple tools. Fourthly, in high-dimensional genomic data processing [21] where genomic datasets with millions of features can be pre-processed and filtered in Python using efficient array operations. R executes statistical analyses such as differential expression or pathway enrichment testing. Julia carries out large-scale permutation tests, principal component analyses, or other computationally intensive multivariate operations, returning results to Python for integration into interactive notebooks and reports. And finally, in method development and prototyping [8,12]; researchers developing new statistical methods can implement prototypes entirely in Python for flexibility. Once an algorithm is performance-critical, it can be ported to Julia for intensive simulations, resampling, or numerical optimization. R is then used to benchmark the new method against established statistical procedures. SiPy orchestrates this cycle, maintaining reproducibility and reducing the friction of switching languages.

Across these examples, SiPy functions as the central orchestrator, coordinating workflow steps, managing data exchange, and integrating results into a coherent, reproducible pipeline. By using R and Julia as backends, users benefit from both methodological rigor and high-performance computation while maintaining the accessibility and flexibility of Python-driven analysis [22,23]. Hence, future work can be both porting commonly used operations from both R and Julia (such as shown in SiPy 0.7.0 [14], or building a library of R and Julia scripts for use in SiPy.

Conclusion

This work presents SiPy 0.8.0 as a three-legged statistical and data analysis framework that explicitly integrates Python, R, and Julia through a transparent, subprocess-based architecture. By treating R and Julia as complementary computational backends rather than competing frontends, SiPy offers a sustainable and reproducible approach to combining statistical rigor with performance-aware computation within a single, coherent workflow.

Supplementary Materials

Source codes of SiPy can be found at https://github.com/mauriceling/sipy while documentation can be found at https://github.com/mauriceling/sipy/wiki. The release page for SiPy 0.8.0 (codenamed as Mango Yoghurt Cake, and released on 09 January 2026) can be found at https://bit.ly/SiPy-080.

Conflict of Interest

The author declares no conflict of interest.

Bibliography

1. Lortie CJ. "Python and R for the Modern Data Scientist". *Journal of Statistical Software* 103 (2022): Book Review 2.

2. Perez F., *et al*. "Python: An Ecosystem for Scientific Computing". *Computing in Science and Engineering* 13.2 (2011): 13–21.

3. Racine J and Hyndman R. "Using R to Teach Econometrics". *Journal of Applied Econometrics* 17.2 (2021): 175–189.

4. Tan NT., *et al*. "SiPy - Bringing Python and R to the End-User in a Plugin-Extensible System". *Medicon Medical Sciences* 8.6 (2025): 32–41.

5. Perera N., *et al*. "In-Depth Analysis on Parallel Processing Patterns for High-Performance Dataframes". *Future Generation Computer Systems* 149 (2023): 250–264.

6. Bezanson J., *et al*. "Julia: A Fresh Approach to Numerical Computing". *SIAM Review* 59.1 (2017): 65–98.

7. Rassokhin D. "The C++ Programming Language in Cheminformatics and Computational Chemistry". *Journal of Cheminformatics* 12.1 (2020): 10.

8. Gmys J., *et al*. "A Comparative Study of High-Productivity High-Performance Programming Languages for Parallel Metaheuristics". *Swarm and Evolutionary Computation* 57 (2020): 100720.

9. Perkel JM. "Julia: Come for the Syntax, Stay for the Speed". *Nature* 572.7767 (2019): 141–142.

10. Rackauckas C and Nie Q. "Differential Equations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia". *Journal of Open Research Software* 5.1 (2017): 15.

11. Bezanson J., *et al*. "Julia: Dynamism and Performance Reconciled by Design". *Proceedings of the ACM on Programming Languages* 2 (2018): 1–23.

12. Hodson TO., *et al*. "Reproducibility Starts at the Source: R, Python, and Julia Packages for Retrieving USGS Hydrologic Data". *Water* 15.24 (2023): 4236.

13. Maheshwari BR. "Comparative Analysis of Program Execution Time Required by Python, R and Julia Compiler". *Indian Journal of Computer Science and Technology* 3.1 (2024): 25–27.

14. Ambel WB., *et al*. "SiPy 0.7.0 – R-Based ANOVA and Survival Analyses". *Open Access Journal of Science* 9 (2026): 1–5.

15. Micoli C., *et al*. "Simulation-Based Bayesian Predictive Probability of Success for Interim Monitoring of Clinical Trials With Competing Event Data: Two Case Studies". *Pharmaceutical Statistics* 25.1 (2026): e70050.

16. Hesselbarth MHK., *et al*. "Computational Methods in Landscape Ecology". *Current Landscape Ecology Reports* 10.1 (2024): 2.

17. McCrea R., *et al*. "Realising the Promise of Large Data and Complex Models. *Methods in Ecology and Evolution* 14.1 (2023): 4–11.

18. Heiss F. "Using R for Introductory Econometrics". 2nd Ed (2020).

19. Heiss F and Brunner D. "Using Python for Introductory Econometrics". 2nd Ed (2020).

20. Heiss F and Brunner D. "Using Julia for Introductory Econometrics" (2023).

21. Chu BB., *et al*. "Second-Order Group Knockoffs with Applications to Genome-Wide Association Studies". *Bioinformatics (Oxford, England)* 40.10 (2024): btae580.

22. Bishnu S., *et al*. "Comparing the Performance of Julia on CPUs versus GPUs and Julia-MPI versus Fortran-MPI: a case study with MPAS-Ocean (Version 7.1)". *Geoscientific Model Development* 16.19 (2023): 5539–5559.

23. Eschle J., *et al*. "Potential of the Julia Programming Language for High Energy Physics Computing". *Computing and Software for Big Science* 7.1 (2023): 10.