Avaliação de Desempenho da Equação do Calor: Uma Análise Comparativa entre Python e Julia

ENZO LISBÔA PEIXOTO, Universidade Federal do Rio Grande do Sul, Brasil
NATHAN MATTES, Universidade Federal do Rio Grande do Sul, Brasil
PEDRO SCHOLZ SOARES, Universidade Federal do Rio Grande do Sul, Brasil

Este trabalho apresenta uma análise de desempenho comparativa entre as linguagens Python e Julia na resolução numérica da Equação do Calor utilizando o Método das Diferenças Finitas em 1D, 2D e 3D. O estudo foi conduzido em um ambiente controlado utilizando contêineres Docker e isolamento de recursos de hardware (CPU Pinning) para garantir a reprodutibilidade. Os resultados demonstram a superioridade de desempenho da linguagem Julia em tarefas intensivas de CPU, com speedups significativos em relação à implementação pura em Python, embora com um consumo de memória superior. A análise estatística, incluindo regressão linear e gráficos de estabilidade, valida o comportamento assintótico esperado dos algoritmos e destaca a importância do controle de ruído do sistema operacional em benchmarks de computação.

CCS Concepts: • **General and reference** → **Measurement**; **Empirical studies**.

Additional Key Words and Phrases: Equação do Calor, Diferenças Finitas, Python, Julia, Benchmarking, Docker

ACM Reference Format:
Enzo Lisbôa Peixoto, Nathan Mattes, and Pedro Scholz Soares. 2025. Avaliação de Desempenho da Equação do Calor: Uma Análise Comparativa entre Python e Julia. *J. ACM* 1, 1, Article 1 (November 2025), 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introdução

A simulação numérica de fenômenos físicos, como a difusão de calor, é uma tarefa computacionalmente intensiva que exige eficiência tanto do hardware quanto do software. Historicamente, linguagens compiladas como C e Fortran dominam este cenário. No entanto, linguagens de alto nível como Python e Julia ganharam popularidade pela facilidade de desenvolvimento. Este trabalho investiga o trade-off entre facilidade de uso e desempenho, comparando implementações iterativas da Equação do Calor nas duas linguagens.

2 Metodologia

2.1 O Problema Computacional

A Equação do Calor foi discretizada utilizando o Método das Diferenças Finitas (FDM)[3] com esquema explícito no tempo. Foram implementados três dimensões:

- **1D:** Complexidade $O(L)$, onde L é o tamanho do domínio linear.

Authors' Contact Information: Enzo Lisbôa Peixoto, elpeixoto@inf.ufrgs.br, Universidade Federal do Rio Grande do Sul, Porto Alegre, Rio Grande do Sul, Brasil; Nathan Mattes, nmattes@inf.ufrgs.br, Universidade Federal do Rio Grande do Sul, Porto Alegre, Rio Grande do Sul, Brasil; Pedro Scholz Soares, pssoares@inf.ufrgs.br, Universidade Federal do Rio Grande do Sul, Porto Alegre, Rio Grande do Sul, Brasil.

Manuscript submitted to ACM

- **2D:** Complexidade $O(L^2)$, representando uma malha quadrada.
- **3D:** Complexidade $O(L^3)$, representando um volume cúbico.

2.2 Design Experimental

Foi utilizado um projeto fatorial completo considerando os fatores:

- **Linguagem:** Python vs. Julia.
- **Dimensão:** 1D, 2D, 3D.
- **Tamanho do Problema (L):** Três níveis (Low, Mid, High) ajustados para cada dimensão para manter tempos de execução mensuráveis.

Cada configuração foi executada 30 vezes (‘REPLICACOES = 30‘) para garantir significância estatística e permitir a análise de variabilidade[2][1].

3 Ambiente e Ferramentas

3.1 Hardware e Controle de Ruído

Para minimizar a interferência do Sistema Operacional e garantir resultados determinísticos, foram aplicadas as seguintes técnicas de controle de ambiente:

- **CPU Pinning:** Os experimentos foram isolados em um núcleo físico específico (‘–cpuset-cpus="0"‘) via Docker.
- **Frequency Governor:** O governador da CPU foi fixado em `performance` para evitar oscilações de *clock*.
- **Turbo Boost:** Desativado para mitigar variações térmicas (*thermal throttling*).
- **Inibição de Sistema:** Utilização de `systemd-inhibit` para prevenir suspensão ou hibernação durante a execução.

3.2 Software e Automação

O fluxo de trabalho foi totalmente automatizado para garantir reprodutibilidade ("*One script to rule them all*"):

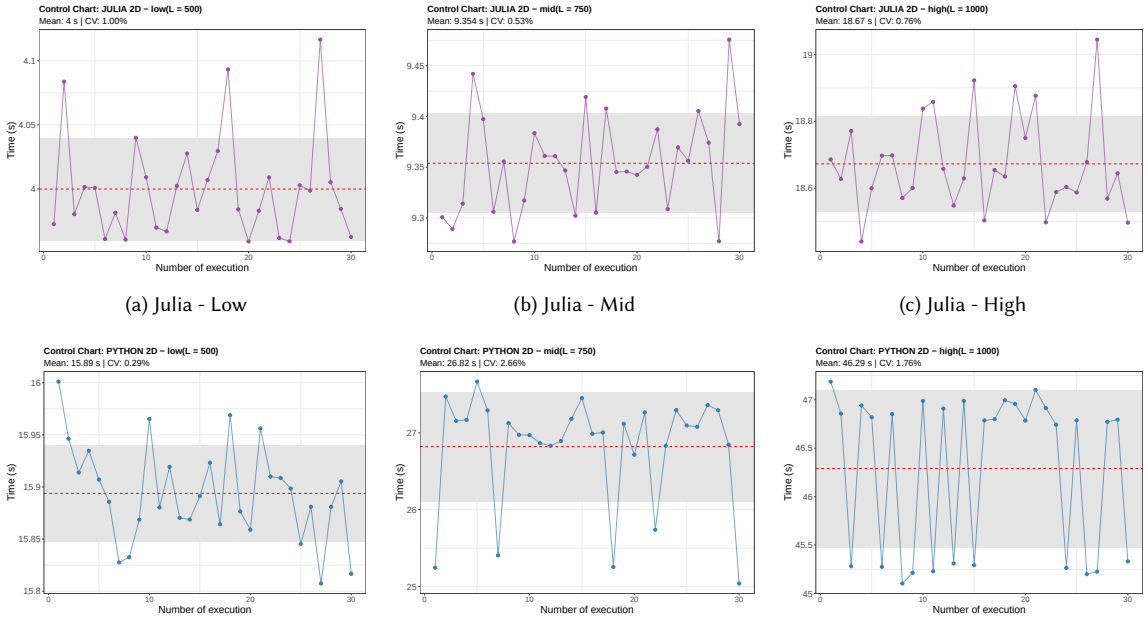
- **Docker:** Contêineres isolados garantiram que as versões das linguagens e bibliotecas fossem idênticas em todas as execuções.
- **Orquestração:** Um makefile que gerenciou a construção da imagem, a geração do plano de execução aleatorizado e a coleta de dados.
- **Análise:** Scripts em R foram utilizados para processar os logs brutos e gerar visualizações estatísticas.

4 Resultados Finais

4.1 Tempo de Execução

Os resultados indicam uma diferença expressiva de desempenho. A Figura 1 apresenta a distribuição do tempo de execução para as três dimensões.

Os experimentos demonstram que o Julia oferece tempos de resposta substancialmente menores que o Python para a mesma lógica algorítmica. A validação estatística apresentada na Tabela 1 confirma a consistência dos dados: o alto valor de R^2 em todos os casos indica que a variação do tempo é explicada quase integralmente pelo aumento do tamanho do problema (L), seguindo a complexidade polinomial teórica.



(a) Julia - Low (b) Julia - Mid (c) Julia - High

(d) Python - Low (e) Python - Mid (f) Python - High

Fig. 1. Gráficos de Estabilidade para a dimensão 2D. A linha superior apresenta os resultados para Julia e a inferior para Python. Embora apresentados apenas os dados de 2D, o comportamento de estabilidade mostram o controle do experimento e a diferença da ordem de grandeza do tempo entre Julia e python. Graficos similares para 1d e 2d podem ser encontrados nos anexos.

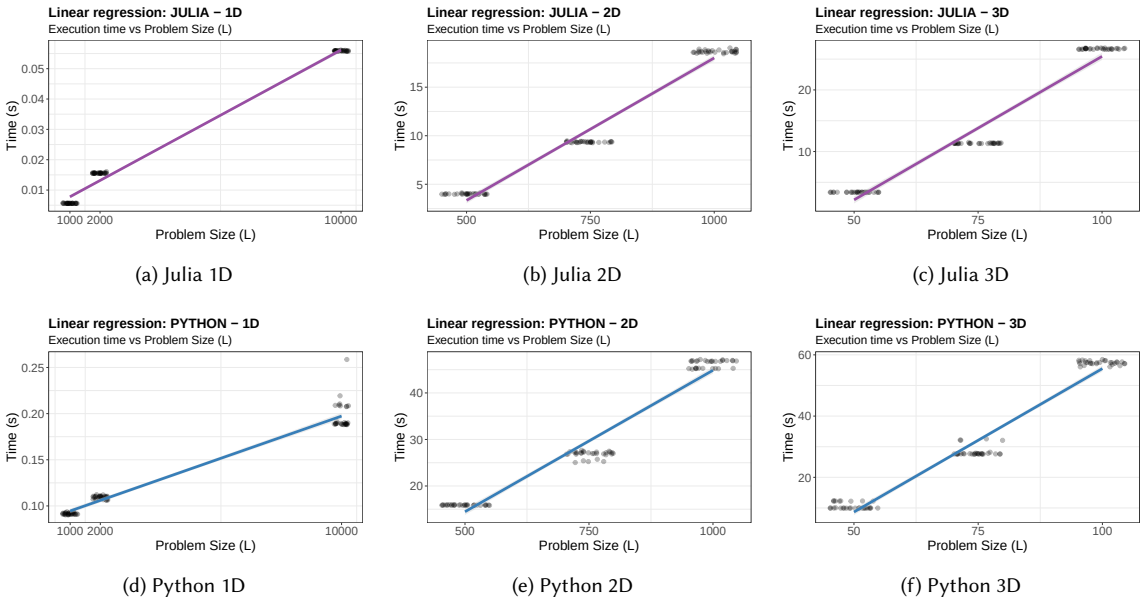
Tabela 1. Coeficiente de Determinação (R^2) para Regressão Linear do Tempo de Execução.

Linguagem	**Dimensão**	R^2	R^2_{adj}
Python	1D	0.9631	0.9627
Python	2D	0.9720	0.9717
Python	3D	0.9773	0.9771
Julia	1D	0.9924	0.9923
Julia	2D	0.9760	0.9758
Julia	3D	0.9675	0.9671

4.2 Consumo de Memória

A análise de memória revela o custo do desempenho. A Figura 3 e a Tabela 2 mostram que o Julia consome consistentemente mais memória RAM que o Python.

A análise do consumo de memória revela um claro trade-off entre tempo e espaço. Enquanto o Julia oferece um desempenho temporal (velocidade) muito superior, ele exige uma alocação de memória RAM significativamente maior que o Python em todas as dimensões testadas.



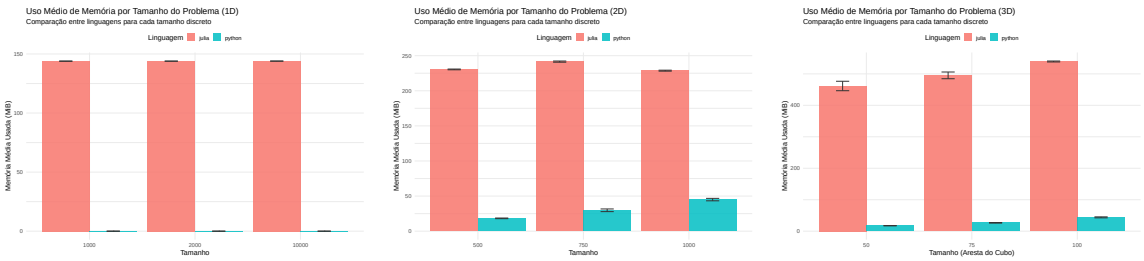
(a) Julia 1D (b) Julia 2D (c) Julia 3D

(d) Python 1D (e) Python 2D (f) Python 3D

Fig. 2. Regressão Linear do Tempo de Execução (t_{exec}) versus Tamanho do Problema (L).%.

Tabela 2. Regressão Linear: Pico de Memória

Linguagem	**Dimensão**	R^2	R^2_{adj}
Python	1D	**1.0000**	1.0000
Python	2D	**0.9908**	0.9907
Python	3D	**0.9661**	0.9657
Julia	1D	**1.0000**	1.0000
Julia	2D	**0.9908**	0.9907
Julia	3D	**0.9667**	0.9663



(a) Dimensão 1D (b) Dimensão 2D (c) Dimensão 3D

Fig. 3. Comparação do consumo de memória entre Python e Julia. Observa-se que, em todas as dimensões, a alocação de memória do Julia é superior, refletindo o custo do JIT e da gestão de tipos.

5 Discussão

5.1 Compilação JIT vs Interpretação

A vantagem massiva do Julia deve-se à sua compilação *Just-In-Time* (JIT) baseada em LLVM. Ao encontrar um laço `for` tipado, o Julia gera instruções de máquina otimizadas, similares a C/C++. O Python, por sua vez, realiza a verificação de tipos (‘type checking‘) e o dispatch de métodos dinamicamente a cada iteração do laço, o que introduz um overhead massivo em algoritmos numéricos iterativos ($O(N)$ verificações).

5.2 Trade-off Memória-Velocidade

Confirmou-se que não existe opção perfeita. O desempenho superior do Julia exige uma alocação de memória mais agressiva para manter as estruturas de dados e o código compilado. Para sistemas com restrição severa de memória, uma implementação Python otimizada (via C-extensions) poderia ser preferível, apesar da complexidade de desenvolvimento.

6 Conclusão

Este estudo demonstrou que, para computação científica baseada em simulações iterativas, a linguagem Julia oferece uma combinação superior de produtividade (sintaxe de alto nível) e desempenho (velocidade de execução), superando Python.

Do ponto de vista metodológico, o trabalho evidenciou que a medição precisa de desempenho em sistemas modernos exige um controle rigoroso do ambiente. O isolamento de processos via Docker e o controle de frequência da CPU foram fundamentais para reduzir o ruído experimental e obter dados consistentes e homescedásticos.

Recomenda-se para trabalhos futuros a exploração de paralelismo (multithreading e GPU) para avaliar se a lacuna de desempenho pode ser reduzida.

Referências

[1] Perf Schnorr 2025. Perf. Analysis: Análise de Desempenho (CMP223 / INF01146). https://github.com/schnorr/perf.
[2] SMPE Schnorr 2025. SMPE: Scientific Methodology and Performance Evaluation for Computer Scientists. https://github.com/schnorr/SMPE.
[3] REAMAT UFRGS. 2025. *Cálculo Numérico: Um Livro Colaborativo.* UFRGS. https://github.com/reamat/CalculoNumerico