Revista Eletrônica de Iniciação Científica em Computação, 2026, 24:1
ISSN: 3085-8461 • doi: 10.5753/reic.2026.7794

Artigo de Pesquisa

O Custo da Abstração no OpenCV: Análise Quantitativa de Desempenho em C++, Java e Python

The Cost of Abstraction in OpenCV: Quantitative Performance Analysis across C++, Java, and Python

Enzo Rocha Leite Diniz Ribas [Centro Universitário Dom Helder Câmara | enzorochaleitedinizribas@gmail.com]
Rafael de Andrade Alves [Centro Universitário Dom Helder Câmara | *rafaaalvescontato@gmail.com*]
Pedro Lucas Vieira De Souza Ramalho [Centro Universitário Dom Helder Câmara |*pedrolucasvramalho@gmail.com*]
João Augusto Vieira Ramalho [Centro Universitário Dom Helder Câmara | *joaoaugustovr@gmail.com*]
Eduardo do Amaral Melo Pereira [Centro Universitário Dom Helder Câmara | *eduamaralmp@gmail.com*]
Presleyson Plínio de Lima [Centro Universitário Dom Helder Câmara | *presleyson.lima@academico.domhelder.edu.br*]
Ricardo Luiz de Freitas [Centro Universitário Dom Helder Câmara | *ricardo.freitas@academico.domhelder.edu.br*]

Centro Universitário Dom Helder, R. Álvares Maciel, 628 — Santa Efigênia, Belo Horizonte — MG, 30150–250, Brasil.

Resumo. Este artigo investiga o "custo da abstração" no desenvolvimento de aplicações de visão computacional utilizando a biblioteca OpenCV, analisando o impacto da escolha da linguagem de programação no desempenho e na estabilidade do sistema. Por meio de experimentos quantitativos utilizando classificadores *Haar Cascade*, comparou-se a eficiência da implementação nativa em C++ contra ambientes gerenciados em Java e Python. Os resultados demonstram que a abstração impõe um custo mensurável, resultando em um acréscimo de latência entre 13% e 20% nas linguagens interpretadas e gerenciadas, além de maior variabilidade temporal. A análise estatística e os modelos de regressão evidenciam diferenças consistentes entre as linguagens avaliadas, tanto no crescimento da latência quanto na estabilidade temporal durante a execução. Os resultados reforçam a influência do nível de abstração sobre o desempenho de aplicações de visão computacional e contribuem com evidências quantitativas para avaliação comparativa do OpenCV em diferentes ambientes de execução. O estudo conclui fornecendo diretrizes para auxiliar na seleção da tecnologia levando em consideração o trade-off entre eficiência computacional e abstração de desenvolvimento.

Abstract. This article investigates the "cost of abstraction" in the development of computer vision applications using the OpenCV library, analyzing the impact of programming language choice on system performance and stability. Through quantitative experiments using *Haar Cascade* classifiers, the efficiency of native C++ implementation was compared against managed environments in Java and Python. The results demonstrate that abstraction imposes a measurable cost, resulting in a latency increase between 13% and 20% in interpreted and managed languages, as well as greater temporal variability. Statistical analysis and regression models reveal consistent differences between the evaluated languages, both in latency growth and temporal stability during execution. The findings reinforce the influence of abstraction level on the performance of computer vision applications and contribute with quantitative evidence for comparative evaluation of OpenCV across different execution environments. The study concludes by providing guidelines to assist in technology selection considering the trade-off between computational efficiency and development abstraction.

Palavras-chave: Visão Computacional, Linguagens de Programação, Java, C++, Cpp, C, Python, OpenCV

Keywords: *Computer Vision*, *Programming Languages*, *Java*, *C++*, *Cpp*, *C*, *Python*, *OpenCV*

Recebido/Received: 20 March 2026 • **Aceito/Accepted:** 08 July 2026 • **Publicado/Published:** 14 August 2026

1 Introdução

A visão computacional tornou-se um dos pilares fundamentais da tecnologia moderna, impulsionando inovações que variam desde sistemas de condução autônoma e diagnósticos médicos por imagem até autenticação biométrica em dispositivos móveis [Szeliski, 2022]. Nesse contexto, a biblioteca OpenCV (*Open Source Computer Vision Library*) tornou-se uma das ferramentas mais utilizadas tanto na indústria quanto na academia, devido à sua ampla coleção de algoritmos otimizados e suporte multiplataforma [Bradski, 2000; Kaehler and Bradski, 2017].

Embora o núcleo do OpenCV seja desenvolvido em C++ para garantir máxima eficiência computacional, a biblioteca oferece interfaces (*bindings*) para diversas outras linguagens, como Python e Java. A popularização dessas linguagens decorre de fatores distintos: Python expandiu-se fortemente em Ciência de Dados, Inteligência Artificial e prototipagem rápida devido à simplicidade sintática e ao ecossistema científico robusto [Lutz and Ascher, 1999], enquanto Java consolidou-se em aplicações corporativas e sistemas distribuídos devido à sua portabilidade e amplo suporte multiplataforma [Schildt, 2019].

Entretanto, a utilização de linguagens gerenciadas ou interpretadas adiciona camadas de abstração, incluindo mecanismos de interoperação, gerenciamento automático de memória e execução mediada por *runtime*. Na literatura, esse fenômeno é frequentemente associado ao chamado "custo da abstração", entendido como a sobrecarga computacional (*overhead*) decorrente de mecanismos que priorizam produ-

tividade e portabilidade em detrimento de eficiência de execução [Prechelt, 2000; Nanz and Furia, 2015; Pereira *et al.*, 2017].

Apesar da ampla adoção do OpenCV em múltiplas linguagens, ainda existem poucos estudos que investigam de forma controlada o impacto dessas camadas de abstração sobre tarefas clássicas de visão computacional utilizando implementações funcionalmente equivalentes. Grande parte dos benchmarks existentes concentra-se em comparações entre bibliotecas distintas, *workloads* heterogêneos ou cenários específicos de processamento [Iglovikov, 2025; Šrajer *et al.*, 2018].

Diante dessa lacuna, este trabalho tem como objetivo quantificar e analisar o impacto da escolha da linguagem de programação no desempenho de aplicações OpenCV implementadas em C++, Java e Python. Para isso, foi realizado um estudo comparativo entre implementações equivalentes utilizando classificadores *Haar Cascade* para detecção de placas veiculares.

De forma geral, tem-se como hipótese que os resultados confirmem a existência de um *trade-off* entre desempenho bruto e abstração, evidenciando que a escolha da linguagem pode impactar significativamente o desempenho de aplicações de visão computacional.

As contribuições deste artigo visam fornecer diretrizes empíricas para auxiliar arquitetos de *software* e desenvolvedores na escolha da pilha tecnológica mais adequada para aplicações de visão computacional sensíveis a desempenho.

Este trabalho está estruturado da seguinte forma: a seção 2 apresenta uma revisão bibliográfica sobre as linguagens de programação e biblioteca estudadas. A seção 3 detalha a metodologia adotada para a implementação e avaliação dos algoritmos. A seção 4 apresenta os resultados obtidos. A seção 5 discute os resultados obtidos, e a seção 6 conclui o artigo, destacando as principais descobertas e sugerindo direções para trabalhos futuros.

2 Revisão Bibliográfica

A revisão bibliográfica permitiu identificar as principais bibliotecas e linguagens de programação utilizadas em visão computacional, bem como suas características, vantagens e desvantagens. A seguir, são apresentadas as principais ferramentas analisadas:

2.1 Linguagens de Programação

Diversas linguagens de programação são utilizadas para desenvolver aplicações de visão computacional. A seguir, são apresentadas algumas características das principais linguagens analisadas, com base em referências da literatura e índices de popularidade como o TIOBE [TIOBE Software, 2026].

Em terceiro lugar no *ranking* de linguagem mais utilizada segundo o índice TIOBE, **Java** é uma linguagem de programação que se destaca por ser multiplataforma, segura e orientada a objetos [Schildt, 2019]. Apesar de não ser a principal escolha para visão computacional, possui um conjunto sólido, embora menos extenso, de bibliotecas para visão computacional, como JavaCV, BoofCV e OpenIMAJ, que permitem o desenvolvimento de aplicações nessa área.

Paralelamente, a linguagem **Python** figura entre as linguagens de programação mais populares do mundo segundo o índice TIOBE, sendo amplamente utilizada em áreas da matemática, ciência de dados, inteligência artificial e visão computacional, devido à sua simplicidade, facilidade de aprendizado e utilização, portabilidade e vasta quantidade de bibliotecas [Lutz and Ascher, 1999]. Ela possui um ecossistema robusto para processamento de imagens e visão computacional, com bibliotecas como OpenCV, TensorFlow e MediaPipe, que permitem o desenvolvimento de aplicações complexas de forma acessível e eficiente.

Por fim, segundo o índice TIOBE, **C++** é a quarta linguagem de programação mais utilizada dentre os programadores do mundo. Também é considerada uma linguagem de alto desempenho utilizada especialmente no desenvolvimento de aplicações que exigem eficiência computacional e controle detalhado dos recursos do sistema. De acordo com Meyers [2014], a linguagem apresenta características de programação de baixo nível permitindo suporte à programação orientada a objetos, possibilitando a criação de aplicações robustas e eficientes. No contexto da visão computacional, C++ é uma das linguagens mais utilizadas, principalmente em aplicações de tempo real, devido à sua alta velocidade de execução. Além disso, algumas das bibliotecas fundamentais para a área foram primeiramente desenvolvidas em C++, como o OpenCV e o Dlib, tornando-a uma escolha consolidada para sistemas de visão computacional de alto desempenho.

2.2 Visão computacional e OpenCv

Visão computacional é o campo de estudo que busca permitir aos computadores descrever e interpretar o mundo físico. Ela envolve o desenvolvimento de algoritmos e técnicas de processamento, análise e extração de informações de imagens e vídeos. A visão computacional é amplamente utilizada em diversas aplicações, como reconhecimento facial (autenticação visual), detecção de objetos, análise de imagens médicas, modelagem 3D, entre outras [Szeliski, 2022]. Originalmente lançada por Gary Bradski na Intel em 1999, a Biblioteca OpenCV (*Open Source Computer Vision Library*) foi concebida com o objetivo de acelerar a pesquisa em visão computacional e inteligência artificial, fornecendo uma infraestrutura sólida e acessível para desenvolvedores [Kaehler and Bradski, 2017].

Bradski [2000] define a ferramenta como uma biblioteca de código aberto voltada para a extração e processamento de dados significativos a partir de imagens. A biblioteca é multiplataforma e possui suporte para sistemas operacionais como Linux, Windows, Android e macOS [Kaehler and Bradski, 2017].

Embora sua estrutura principal seja escrita em C e C++ otimizado para eficiência computacional, a OpenCV oferece interfaces ativas para diversas linguagens, incluindo Python, Java e MATLAB [Kaehler and Bradski, 2017]. A literatura aponta que a biblioteca contém mais de 500 funções que abrangem diversas áreas da visão computacional [Kaehler and Bradski, 2017; Bradski, 2000].

Segundo Kaehler and Bradski [2017], essas ferramentas permitem a implementação de tarefas complexas, tais como: inspeção de produtos em fábricas, imagens médicas, segurança, interfaces de usuário, calibração de câmeras, visão estéreo e robótica.

Além disso, a biblioteca é capaz de rastrear movimen-

tos de objetos em 2D ou 3D, determinar formas de objetos a partir de imagens e associar dados de imagem a significados categóricos (como reconhecimento de gestos) [Bradski, 2000]. A ferramenta também integra um módulo completo de aprendizado de máquina (*Machine Learning*), focado em reconhecimento de padrões estatísticos e agrupamento (*clustering*) [Kaehler and Bradski, 2017].

2.3 Trabalhos Relacionados

O estudo de Iglovikov [2025] realiza um benchmark abrangente de bibliotecas de decodificação JPEG no ecossistema Python, incluindo OpenCV, Pillow e frameworks de aprendizado de máquina, com foco principal no impacto da implementação interna e do uso de bibliotecas nativas como *libjpeg-turbo* sobre o desempenho. Embora o trabalho discuta explicitamente o custo de camadas de abstração e apresente métricas quantitativas detalhadas, sua análise está restrita à linguagem Python e ao estágio de carregamento e decodificação de imagens, não abordando a comparação entre diferentes linguagens de programação nem a equivalência algorítmica de um mesmo método implementado em múltiplos ambientes.

Já o trabalho de Šrajer *et al*. [2018] investiga o impacto de diferentes ferramentas de diferenciação automática em problemas de visão computacional e aprendizado de máquina, comparando implementações em linguagens como C++, Python, Julia, MATLAB e F#. Apesar de envolver múltiplas linguagens e discutir variações significativas de desempenho associadas a abstrações e estratégias de implementação, o foco do estudo está na computação de derivadas e na eficiência de ferramentas de diferenciação automática, e não na execução de algoritmos clássicos de visão computacional nem no uso direto de bibliotecas amplamente adotadas como o OpenCV em tarefas de detecção de objetos.

Hasan and Sallow [2021] apresentam uma revisão abrangente do papel do OpenCV em detecção e reconhecimento facial, descrevendo seus principais algoritmos — como Haar Cascade, Local Binary Patterns (LBP), Local Binary Pattern Histogram (LBPH), EigenFaces e FisherFaces — e sua integração com Python. O trabalho demonstra que o Haar Cascade permanece amplamente empregado em sistemas de detecção em tempo real devido à sua eficiência e simplicidade. Contudo, apesar de mencionar adisponibilidade do OpenCV em C++, Python e Java, o estudo não realiza comparações de desempenho entre essas linguagens para um mesmo algoritmo, tampouco discute o impacto das camadas de abstração de cada *binding* sobre latência, estabilidade ou usabilidade em implantação.

Ahmed *et al*. [2026] conduzem uma avaliação de desempenho comparativa entre C++, Java e Python em tarefas computacionais variadas, incluindo algoritmos de ordenação, busca em profundidade (*Depth First Search*) e operações de leitura e escrita em arquivos, medindo velocidade de execução, consumo de memória e eficiência energética. O estudo destaca que C++ apresenta desempenho superior em tarefas computacionais intensivas, enquanto Python e Java oferecem maior facilidade de desenvolvimento e manutenção, embora com penalidades de desempenho associadas às camadas de abstração e gerenciamento automático de memória. No entanto, o trabalho não se concentra especificamente em algoritmos de visão computacional nem na utilização direta do OpenCV, limitando a aplicabilidade dos resultados para cenários práticos de implantação nessa área.

Diferentemente desses trabalhos, o presente estudo concentra-se em um algoritmo clássico e amplamente utilizado da biblioteca OpenCV, a detecção de placas veiculares por meio de *Haar Cascade*, permitindo uma análise direta e aplicada do impacto da abstração ao se utilizar a mesma biblioteca em diferentes linguagens de programação, reduzindo vieses introduzidos por diferenças de implementação. Além disso, ao contrário de benchmarks que variam implementações, ferramentas ou níveis de otimização, este trabalho busca manter equivalência algorítmica estrita entre C++, Java e Python, isolando o efeito da linguagem e de suas camadas de abstração sobre desempenho em cenários práticos de implantação.

3 Metodologia

A metodologia adotada neste trabalho envolve a análise comparativa de diferentes linguagens de programação e a biblioteca OpenCV. O processo metodológico foi dividido nas seguintes etapas:

3.1 Desenho do Estudo

O estudo foi estruturado como um experimento comparativo entre as linguagens de programação, utilizando a biblioteca OpenCV para implementar um algoritmo clássico de detecção de objetos (Haar Cascade) de maneira equivalente em cada linguagem, garantindo isonomia experimental. O objetivo é avaliar o impacto do nível de abstração de cada linguagem sobre o desempenho da biblioteca em tarefas de visão computacional.

A variável independente do estudo é a linguagem de programação utilizada (C++, Java, Python), enquanto as variáveis dependentes são as métricas de desempenho (tempo de execução), incluindo latência, média de tempo de execução, mediana e desvio padrão.

A biblioteca OpenCV foi escolhida devido à sua ampla adoção na comunidade de visão computacional, suporte a múltiplas linguagens e vasta gama de funcionalidades. As linguagens selecionadas para comparação foram C++, Java e Python, considerando ampla adoção industrial e acadêmica, diversidade de paradigmas de execução e relevância em aplicações de visão computacional, [SEBESTA, 2018]. A escolha dessas linguagens também se justifica pela disponibilidade de bindings oficiais da OpenCV, preservando maior proximidade com as implementações nativas da biblioteca e evitando camadas adicionais de abstração que poderiam distorcer os resultados de desempenho.

Os algoritmos de Haar Cascade foram escolhidos devido à sua baixa complexidade computacional relativa, ampla disponibilidade nas implementações do OpenCV e comportamento determinístico, tornando-os adequados para análises comparativas de desempenho entre linguagens.

A pesquisa caracteriza-se como experimental e aplicada, com abordagem predominantemente quantitativa, baseada na execução controlada de experimentos computacionais e análise comparativa de métricas de desempenho.

3.2 Critérios e Métricas

Foram estabelecidos critérios para a avaliação das linguagens e da biblioteca, com foco em medidas resumo de estatística descritiva do desempenho, incluindo.

A avaliação do desempenho foi realizada a partir da análise do **Tempo de Execução (Latência)**, correspondente ao tempo total gasto para processar um conjunto de imagens utilizando o método `detectMultiScale` da OpenCV. Para esse procedimento, a chamada de detecção foi isolada, com o objetivo de mensurar de forma mais precisa o impacto direto da linguagem de programação no processamento realizado.

A partir dos tempos obtidos, calculou-se a **Média Aritmética Simples de Tempo de Execução** para cada linguagem, após o tratamento dos dados e a remoção de outliers, a fim de reduzir possíveis distorções nos resultados.

Além disso, foi calculada a **Mediana (μ)** dos tempos de execução, por se tratar de uma medida estatística mais robusta diante de valores extremos e capaz de representar a tendência central do desempenho.

Por fim, o **Desvio Padrão (σ)** foi utilizado para avaliar a variabilidade dos tempos de execução em cada linguagem, permitindo identificar o grau de estabilidade e consistência do desempenho observado durante os experimentos.

3.3 Ambiente Experimental

3.3.1 Hardware e Software

Ambiente de *Hardware*: Todos os testes foram realizados em uma máquina com as seguintes especificações:

Tabela 1. Ambiente de Teste e Versões. Fonte: Próprio autor.

Linguagem	**Versão**
Python	3.13.5
Java	21
C++ (GCC)	10.3.0
Hardware	
Processador	AMD Ryzen 5 3600X 6 *Cores*
Memória RAM	32 GB
GPU	NVIDIA GeForce RTX 2060
S.O.	Windows 10 Pro 64 bits
Versão do OpenCV	**Método de Instalação**
OpenCV (Python)	4.11 (*bindings* oficiais cv2)
OpenCV (Java/C++)	4.12 (*Local build*)

3.3.2 Datasets

Para compor o corpus de teste, foram utilizados dois conjuntos de dados públicos de referência (*benchmarks*) obtidos via Kaggle, selecionados pela elevada variabilidade de cenários *in-the-wild*, evitando viés de dados sintéticos ou excessivamente controlados em laboratório.

O primeiro conjunto utilizado foi o *Diverse-LPD training ready*[1], caracterizado por elevada heterogeneidade geográfica e ambiental, contendo placas veiculares de diferentes países, ângulos oblíquos, variações de iluminação (sombras e superexposição) e diferentes distâncias de captura. Essas características impõem maior desafio de generalização aos classificadores Haar Cascade.

O segundo conjunto utilizado foi o *License Plates OCR*[2], composto por um grande volume de imagens com foco explícito em veículos e placas veiculares. Foi utilizada uma amostragem do conjunto para validar a consistência da detecção em imagens de maior resolução e com maior definição espacial do objeto de interesse.

3.3.3 Modelos e Hiperparâmetros

Algoritmos de *Haar Cascade* são compostos por uma série de estágios de classificação, cada um contendo um conjunto de features e um classificador fraco. O número de estágios e a densidade de features impactam diretamente o desempenho e a precisão do modelo. Para avaliar o impacto desses fatores, foram selecionados dois modelos, pré-treinados, para representar diferentes níveis de complexidade computacional:

- `haarcascade_license_plate_rus_16stages.xml`: Modelo com 16 estágios, priorizando velocidade.
- `haarcascade_russian_plate_number.xml`: Modelo focado em detalhes numéricos, com maior densidade de features.

Os testes foram executados utilizando parâmetros idênticos nas três implementações, garantindo equivalência experimental entre as linguagens. O método `detectMultiScale` foi configurado com *scaleFactor* igual a 1.1, realizando uma redução de 10% na pirâmide de imagem a cada escala analisada; *minNeighbors* igual a 3, definindo o número mínimo de vizinhos necessários para validar uma detecção; e *minSize* fixado em 30×30 *pixels*, limitando o tamanho mínimo dos objetos detectados.

3.3.4 Coleta e Tratamento dos Dados de Desempenho

Os dados brutos de tempo de execução (latência) foram coletados para aproximadamente 100 mil de iterações. Observou-se que processos em segundo plano do Sistema Operacional e, no caso de linguagens gerenciadas, pausas do *Garbage Collector*, introduziam *outliers* extremos (picos de latência) que não refletiam o desempenho real do algoritmo de visão computacional.

Para mitigar esse ruído e garantir uma comparação justa baseada na eficiência algorítmica, aplicou-se um método clássico de [Tukey, 1977], baseado no Intervalo Interquartil, o Método IQR. Foram removidos os registros que excederam o limite superior calculado como $Q3 + 1.5 \times IQR$. Este tratamento eliminou anomalias de sistema (latências > 100ms em processos que levam 5ms), permitindo o uso confiável da Média Aritmética (μ) e do Desvio Padrão (σ) como métricas centrais de avaliação, juntamente com a Mediana para análise de tendências centrais robustas. A aplicação do método IQR resultou na remoção de aproximadamente 7,6% dos registros, garantindo que as análises subsequentes refletissem o desempenho típico do algoritmo de detecção, sem distorções causadas por eventos externos ao processo de visão computacional.

[1] Disponível em: `https://www.kaggle.com/datasets/fxmikf/diverse-lpd-training-ready`.

[2] Disponível em: `https://www.kaggle.com/datasets/trainingdatapro/license-plates-1-209-438-ocr-plates`.

3.3.5 Implementação e Coleta de Dados

Para garantir a isonomia do experimento, foram implementados três algoritmos computacionalmente equivalentes em C++, Java e Python. Cada implementação seguiu as mesmas etapas de processamento, utilizando as APIs oficiais da OpenCV para cada linguagem, garantindo que as chamadas de função e os parâmetros fossem idênticos. O código foi estruturado para isolar a chamada de detecção (`detectMultiScale`) e medir exclusivamente o tempo gasto nessa etapa, eliminando a influência de operações de I/O ou pré-processamento. O processo de teste seguiu um pipeline padronizado, composto pelas seguintes etapas:

1. **Ingestão da Imagem**: carregamento direto da imagem do disco para a memória utilizando a função `imread` (padrão BGR), sem pré-processamento explícito de cor;
2. **Cronometragem de Precisão**: medição temporal isolada da função `detectMultiScale`, utilizando relógios monotônicos de alta resolução (nanossegundos) para eliminar a latência de I/O de disco;
3. **Pós-processamento**: renderização das *bounding boxes* na imagem original e salvamento em disco para verificação visual;
4. **Persistência**: exportação dos metadados (arquivo, modelo, contagem e tempo em nanossegundos) para arquivos CSV.

3.3.6 Pipeline de Teste de Desempenho

O pipeline completo utilizado durante os experimentos de desempenho pode ser descrito como segue. Inicialmente, o sistema realiza a criação das estruturas de saída e percorre iterativamente cada modelo Haar Cascade avaliado. Para cada modelo, é criado um arquivo CSV dedicado ao armazenamento das métricas coletadas.

Em seguida, o pipeline executa uma varredura recursiva nos diretórios dos conjuntos de dados, validando os formatos de imagem suportados (`.jpg`, `.png` e `.bmp`). Cada imagem válida é carregada na memória utilizando a função `cv::imread` no padrão BGR da OpenCV.

O núcleo do experimento consiste na execução temporizada da função `detectMultiScale`. A medição é realizada utilizando relógios de alta resolução em nanossegundos, isolando exclusivamente o tempo de execução da etapa de detecção. Durante essa fase, foram utilizados parâmetros padronizados (*scaleFactor*=1.1, *minNeighbors*=3 e *minSize*=30×30) para garantir equivalência experimental entre as implementações.

Após a detecção, o pipeline realiza o cálculo da duração da execução, desenha as *bounding boxes* nas imagens processadas, salva os resultados visuais em disco e registra os metadados experimentais nos arquivos CSV correspondentes. Esse processo é repetido para todos os modelos e conjuntos de dados utilizados no experimento.

O Fluxograma da Figura 1 ilustra o processo completo de cada iteração de teste.

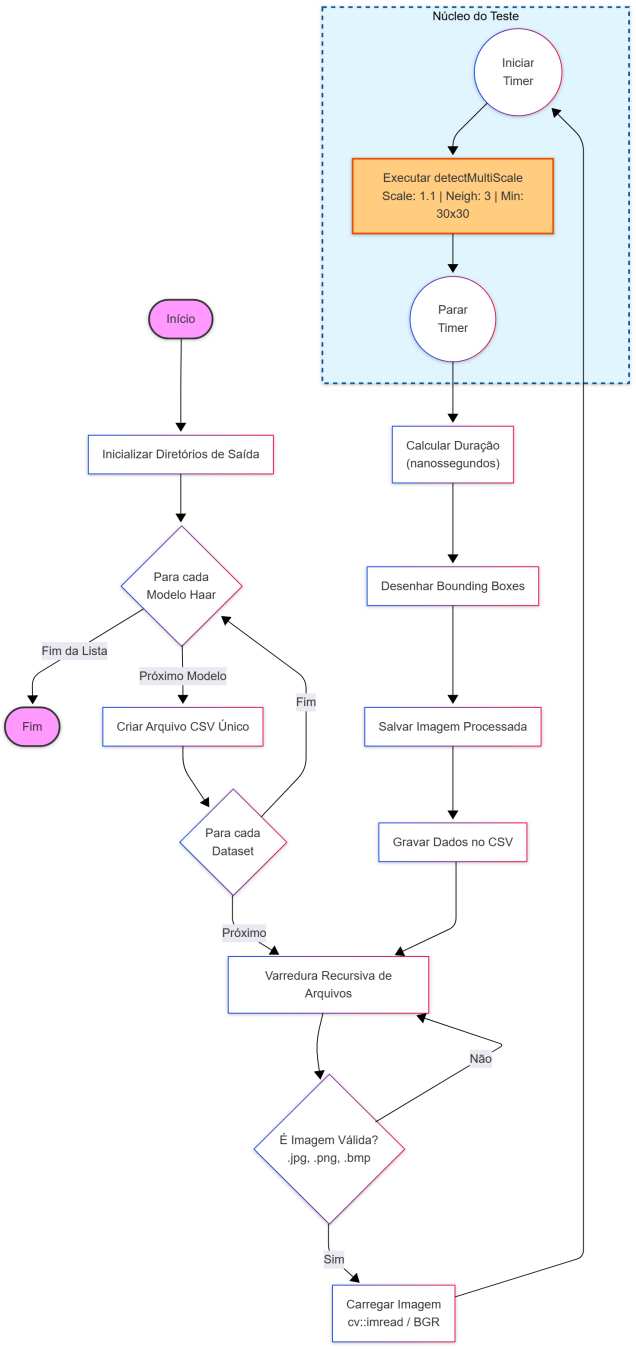

Figura 1. Fluxograma do Pipeline de Teste de Desempenho. Fonte: Próprio autor.

Diferentemente de *pipelines* tradicionais, optou-se por passar a matriz de imagem original diretamente ao método de detecção, delegando ao *framework* (OpenCV) qualquer tratamento interno necessário de canais de cor, visando mensurar o tempo total da chamada de API "como ela é" em uma aplicação final.

A Figura 2 apresenta exemplos de detecção realizados pelo algoritmo Haar Cascade utilizado durante os experimentos, ilustrando a saída visual produzida pelo pipeline de processamento. As imagens exemplificam a capacidade do classificador em identificar placas veiculares em cenários variados, evidenciando a aplicabilidade prática dos modelos selecionados e a consistência funcional das implementações em diferentes linguagens.

Figura 2. Exemplos de detecção de placas veiculares utilizando o método `detectMultiScale` com classificadores Haar Cascade sobre imagens dos conjuntos de dados utilizados nos experimentos. Fonte: Próprio autor e Conjunto de Dados.

4 Resultados

4.1 Visão Geral dos Dados Coletados

Foram analisadas 100 440 execuções do algoritmo de detecção Haar Cascade, distribuídas igualmente entre as linguagens C++, Java e Python. Cada linguagem processou aproximadamente 33 480 imagens, utilizando dois modelos Haar (16STAGES e RUS).

Antes do tratamento estatístico, os tempos médios globais de execução apresentaram valores de 21,12 ms (C++), 23,82 ms (Java) e 25,49 ms (Python), com ocorrência de picos extremos de latência superiores a 800 ms em todas as linguagens.

A aplicação do método IQR agrupado por linguagem e modelo resultou na remoção de 7 584 registros (7,6% do total), eliminando valores anômalos associados a interferências externas ao algoritmo de visão computacional. Os resultados apresentados a seguir consideram exclusivamente os dados limpos.

4.2 Resultados Quantitativos de Desempenho

A Tabela 2 sintetiza os principais resultados quantitativos obtidos após o tratamento estatístico dos dados de latência para cada linguagem e modelo Haar Cascade utilizado.

Em todas as linguagens, aproximadamente 91% das imagens não resultaram em detecções, refletindo a natureza heterogênea do conjunto de imagens junto às limitações inerentes do classificador Haar Cascade quando aplicado a cenários não controlados. A média de detecções por imagem manteve-se estável entre as linguagens, indicando consistência funcional do classificador independentemente da linguagem utilizada.

O gráfico ECDF da Figura 4 ilustra o comportamento da latência acumulada. Observa-se que as três linguagens apresentam tendências similares, com C++ exibindo ligeira vantagem em termos de latência média e menor dispersão na cauda da distribuição.

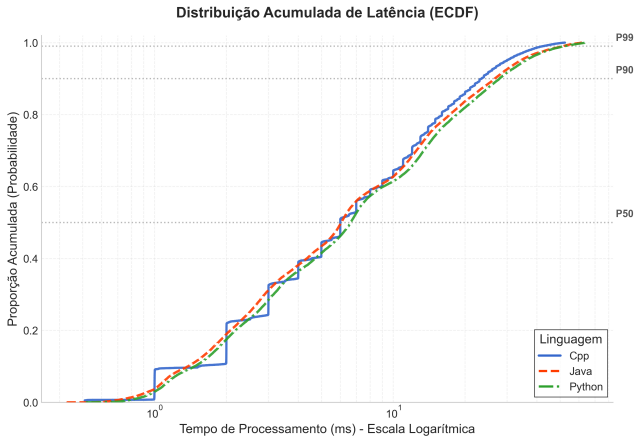

Figura 4. Gráfico ECDF mostrando a latência acumulada em função do número de detecções para os modelos 16STAGES e RUS nas linguagens C++, Java e Python. Fonte: Próprio autor.

A latência média cresce de forma aproximadamente linear à medida que o número de detecções aumenta. Esse comportamento reflete o custo computacional incremental exigido por cada objeto candidato processado, conforme demonstrado nos gráficos de tendência (*OLS Trendline*) da Figura 5. A regressão linear foi empregada com finalidade estritamente exploratória para identificar tendências de crescimento do custo computacional.

A linearidade observada sugere uma relação proporcional entre a densidade de objetos na cena e o tempo de execução, indicando previsibilidade e escalabilidade do sistema dentro do intervalo analisado. Esse padrão é consistente entre C++, Java e Python, embora as diferentes inclinações das retas evidenciem a disparidade na eficiência de execução entre as linguagens compiladas e interpretadas/gerenciadas.

4.3 Análise Estatística da Latência

O gráfico de Barras da Figura 6 apresenta a comparação visual das distribuições dos tempos de execução.

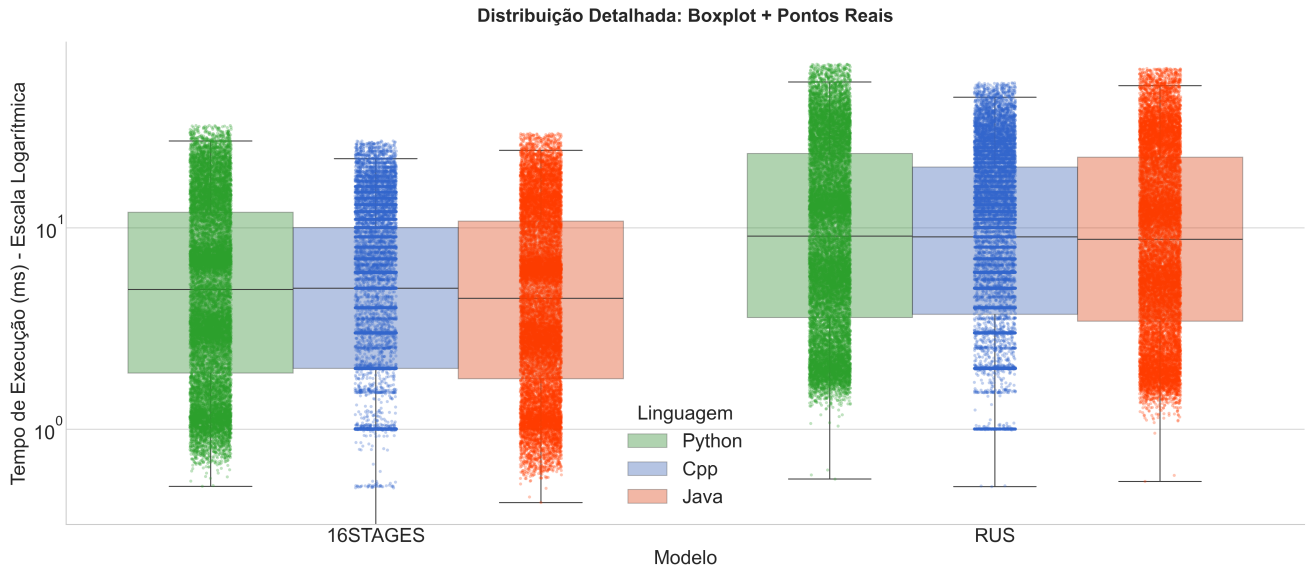

Figura 3. Box plot mostrando a distribuição dos tempos de execução para os modelos 16STAGES e RUS nas linguagens C++, Java e Python. Fonte: Próprio autor.

Tabela 2. Resumo estatístico dos experimentos. Fonte: Próprio autor.

Linguagem	Modelo	Imagens	Detecções	Vazias (%)	Média (ms)	Mediana (ms)	Desv. Padrão (ms)
C++	16STAGES	15521	1597	91.0	6.65	5.00	5.77
C++	RUS	15488	7589	69.1	12.79	9.00	11.29
Java	16STAGES	15471	1566	91.1	6.86	4.46	6.34
Java	RUS	15467	7612	69.1	14.07	8.75	13.49
Python	16STAGES	15516	1595	91.0	7.54	4.93	6.97
Python	RUS	15393	7492	69.3	14.69	9.07	13.97

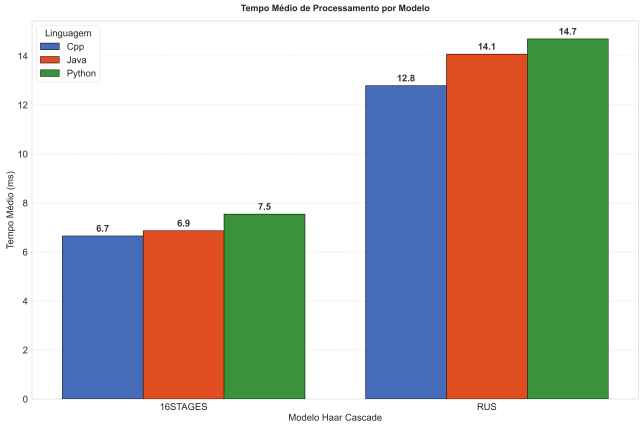

Figura 6. Gráfico de barras comparando os tempos médios de execução entre as linguagens C++, Java e Python. Fonte: Próprio autor.

A concentração dos valores em torno das medianas, evidenciada no *box plot* da Figura 3, sugere que as distribuições, embora apresentem certa assimetria positiva (com médias superiores às medianas devido a valores atípicos de alta latência), mantêm-se estáveis. A implementação em C++ apresentou os menores desvios padrão em ambos os modelos, confirmando sua maior estabilidade em comparação aos ambientes gerenciados (Java e Python).

As linguagens Java e Python exibiram maior dispersão, especialmente no modelo RUS, o que sugere maior sensibilidade a variações do ambiente de execução (*runtime overhead*).

Após a aplicação do método IQR para limpeza de *outliers*, os valores máximos observados reforçam a hierarquia de desempenho: 26,96 ms para C++, 29,42 ms para Java e 32,17 ms para Python (modelo 16STAGES). No modelo RUS, mais custoso computacionalmente, os máximos atingiram 52,42 ms, 61,85 ms e 64,86 ms, respectivamente.

4.4 Síntese Comparativa dos Resultados

De forma geral, os resultados indicam que a implementação em C++ apresentou os menores tempos médios de execução e menor variabilidade em ambos os modelos avaliados. Java e Python apresentaram desempenho intermediário e superior latência média, respectivamente, mantendo, contudo, consistência funcional equivalente no número de detecções realizadas. Os dados evidenciam diferenças mensuráveis de desempenho entre as linguagens analisadas, preservando a equivalência algorítmica e funcional das implementações. A análise quantitativa, sugere que a escolha da linguagem de programação tem impacto direto no desempenho do sistema de visão computacional, especialmente em cenários de alta densidade de objetos ou requisitos de tempo real. A implementação em C++ se destaca como a mais eficiente, enquanto Java e Python apresentam custos adicionais associados à abstração e gerenciamento de recursos, o que pode ser um fator crítico a ser considerado na seleção da tecnologia para aplicações específicas.

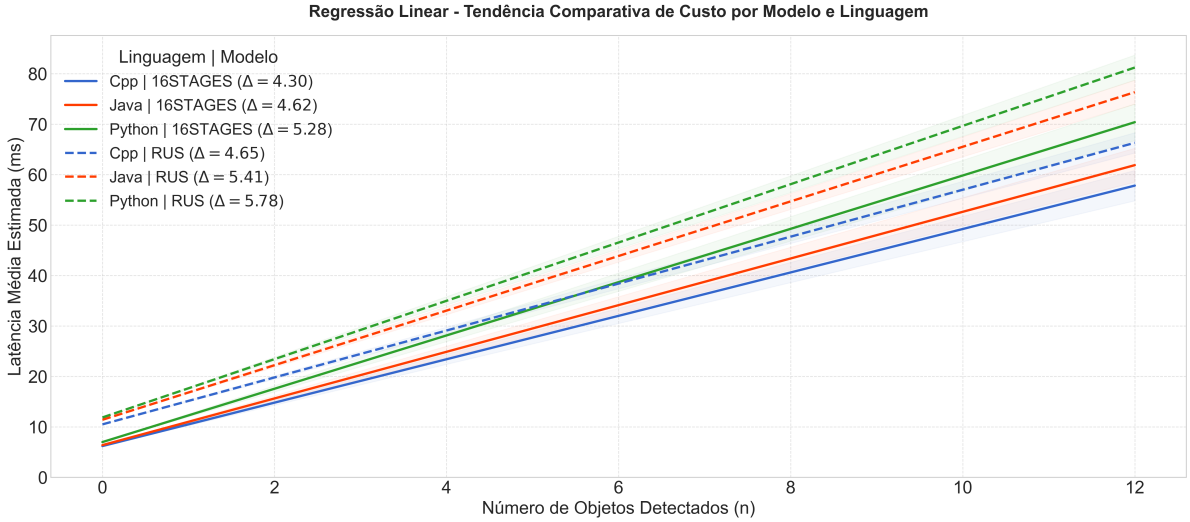

Figura 5. Gráficos de regressão linear para os dois modelos comparados. Fonte: Próprio autor.

5 Discussão

Os resultados obtidos confirmam a hipótese inicial de que existe um custo computacional tangível associado à abstração das linguagens de programação ao utilizar a biblioteca OpenCV. A supremacia da implementação em C++ em termos de latência (média de 6,65 ms no modelo 16STAGES) e estabilidade (menor desvio padrão) corrobora a literatura técnica, que atribui essa eficiência à ausência de camadas intermediárias de interpretação e ao gerenciamento manual de memória.

Por outro lado, Python apresentou as maiores médias de latência (7,54 ms e 14,69 ms para os modelos 16STAGES e RUS, respectivamente), comportamento esperado devido às camadas adicionais de abstração e ao modelo interpretado da linguagem. Ainda assim, a diferença observada permanece relativamente moderada em cenários não críticos de tempo real, sugerindo que Python continua sendo uma alternativa viável para aplicações de prototipagem rápida, processamento em lote e desenvolvimento acadêmico em visão computacional o que justifica sua popularidade em contextos acadêmicos ou exploratórios.

O comportamento do Java merece consideração particular. Embora a linguagem tenha apresentado desempenho médio competitivo, observou-se maior dispersão temporal e presença de outliers significativos, especialmente no modelo RUS, cujo desvio padrão atingiu 13,49 ms. Esse comportamento pode estar relacionado às características do ambiente gerenciado da *Java Virtual Machine* (JVM), incluindo mecanismos automáticos de gerenciamento de memória e pausas ocasionais do *Garbage Collector*, reduzindo a previsibilidade temporal em aplicações com requisitos rígidos de tempo real.

A Figura 5 apresenta a regressão linear da latência média em função do número de objetos detectados para os modelos 16STAGES e RUS nas linguagens C++, Java e Python. Observa-se crescimento aproximadamente linear em todos os cenários avaliados, indicando relação proporcional entre aumento da carga e tempo de processamento. Em ambos os modelos, C++ apresentou as menores inclinações ($\Delta = 4,30$ e $\Delta = 4,65$), seguido por Java e Python, respectivamente. Python registrou os maiores coeficientes angulares, evidenciando maior custo incremental por objeto detectado. Além disso, o modelo RUS manteve latências superiores ao 16STAGES em todas as linguagens, sugerindo maior custo computacional. As faixas sombreadas indicam a variabilidade associada à estimativa linear e demonstram baixa dispersão em torno das regressões ajustadas. Esse comportamento sugere que o custo computacional adicional introduzido pelas linguagens de maior abstração tende a permanecer aproximadamente proporcional ao crescimento da carga de processamento do algoritmo.

Uma barreira metodológica deste estudo é o uso de versões distintas do OpenCV entre os ambientes avaliados: a implementação em Python utilizou os bindings oficiais via cv2 (OpenCV 4.11), enquanto as implementações em C++ e Java utilizaram uma build local da versão 4.12. A análise do changelog oficial do [OpenCV Contributors, 2025] entre essas duas versões não evidencia alterações relevantes no módulo *objdetect* que é responsável pela função *detectMultiScale*, núcleo da análise do experimento ou outras funcionalidades diretamente relacionadas com *CascadeClassifiers*. As principais mudanças introduzidas na versão 4.12 concentram-se em otimizações de desempenho nos módulos Core e Imgproc, na reestruturação do *Hardware Abstraction Layer (HAL)*, em melhorias para inferência de redes neurais (DNN) e em correções pontuais para funcionalidades como ArUco e QRCode, sem impacto direto sobre o algoritmo de detecção *Haar Cascade* empregado neste trabalho. Dessa forma, entende-se que a diferença de versões não compromete a validade das comparações realizadas. Ainda assim, não se pode descartar a possibilidade de pequenas variações nos tempos absolutos de execução observados, decorrentes de otimizações internas da biblioteca. Além disso, considerando que a hierarquia de desempenho observada (C++ < Java < Python) é consistente com resultados amplamente reportados na literatura para comparações entre linguagens compiladas e gerenciadas, independentemente da biblioteca utilizada Ahmed *et al*. [2026], conclui-se que essa discrepância de versão não afeta a conclusão central do estudo, embora possa influenciar marginalmente as magnitudes percentuais observadas.

6 Análises Finais

Este estudo buscou quantificar o impacto da escolha da linguagem de programação no desempenho de algoritmos de visão computacional utilizando a biblioteca OpenCV. Os resultados obtidos indicam que o "custo da abstração" é mensurável e estatisticamente relevante, representando um acréscimo de latência observado entre 13% e 20% para linguagens gerenciadas (Java e Python) em comparação à implementação nativa em C++ nos cenários analisados. Esse comportamento mostra-se consistente com análises comparativas de desempenho entre linguagens nativas e gerenciadas reportadas na literatura [Ahmed *et al.*, 2026].

A análise comparativa dos experimentos permite destacar perfis de desempenho distintos para cada linguagem no contexto de aplicações de visão computacional:

- **C++:** apresentou os menores tempos médios de execução e menor variabilidade temporal, demonstrando elevada adequação para sistemas embarcados, aplicações críticas de segurança e cenários com requisitos rigorosos de desempenho e determinismo temporal. Características já documentadas em estudos de comparação entre linguagens de programação.
- **Python:** apresentou maior latência média entre as linguagens avaliadas, porém mantendo comportamento previsível e diferença moderada de desempenho em relação ao C++. Esses resultados sugerem viabilidade da linguagem em aplicações de prototipagem rápida, pesquisa acadêmica e processamento não crítico em tempo real.
- **Java:** apresentou desempenho intermediário, porém com maior dispersão temporal e ocorrência mais frequente de outliers, possivelmente associados às características do ambiente gerenciado da JVM e aos mecanismos automáticos de gerenciamento de memória.

Como trabalhos futuros, sugere-se a extensão da análise para outras bibliotecas populares de visão computacional, como TensorFlow, PyTorch e Keras, bem como a inclusão de métricas adicionais relacionadas ao consumo de memória, eficiência energética e escalabilidade em ambientes distribuídos e comparações qualitativas como usabilidade, facilidade de aprendizado e manutenibilidade, disponibilidade de suporte e documentação. Ademais, investigações sobre o impacto de diferentes paradigmas de programação (funcional, orientada a objetos, etc.) no desempenho de algoritmos de visão computacional podem oferecer insights valiosos para a comunidade.

Em suma, os resultados demonstram que não existe uma linguagem universalmente superior para o desenvolvimento de aplicações com OpenCV. A escolha da tecnologia mais adequada depende diretamente dos requisitos de desempenho, previsibilidade temporal, escalabilidade e complexidade do sistema em desenvolvimento.

Declarações complementares

Agradecimentos

Agradecemos ao Centro Universitário Dom Helder pela infraestrutura disponibilizada para a realização deste trabalho. Manifestamos também nosso reconhecimento ao Prof. Dr. Presleyson Plínio de Lima, pela orientação e pelo suporte prestado ao longo do desenvolvimento deste estudo; ao Prof. Me. Ricardo Luiz de Freitas, pela orientação e pelas valiosas contribuições acadêmicas; e ao Prof. Me. Marden Cicarelli, pelo auxílio durante a fase de experimentação. Estendemos, ainda, nossos agradecimentos aos demais colegas e colaboradores que contribuíram, direta ou indiretamente, para a concretização deste trabalho.

Contribuições dos autores

Enzo Rocha Leite Diniz Ribas atuou como principal autor e contribuidor deste manuscrito, participando das etapas de conceituação, definição metodológica, análise formal, redação do rascunho original, revisão, edição e administração do projeto. Ricardo Luiz de Freitas e Presleyson Plínio de Lima contribuíram com a administração, supervisão geral do projeto e revisão crítica do manuscrito. Eduardo do Amaral Melo participou das atividades de conceituação, investigação, experimentação, desenvolvimento de software e redação do rascunho original. João Augusto Vieira Ramalho, Pedro Lucas Vieira de Souza Ramalho e Rafael Alves contribuíram com a redação do rascunho original, investigação, experimentação e desenvolvimento de software.

Conflitos de interesse

Os autores declaram não haver conflitos de interesse.

Disponibilidade de dados e materiais

Todo o código de teste, *scripts* de medição e procedimentos de pré-processamento foram versionados e disponibilizados no repositório do projeto para garantir reprodutibilidade, em https://github.com/oEnzoRibas. e https://github.com/Atom-Laboratory. Ribas [2025].

Outras informações relevantes

Correspondência deve ser enviada para: Enzo Rocha L. Diniz Ribas. E-mail: enzorochaleitedinizribas@gmail.com ou para Ricardo Luiz de Freitas. E-mail: ricardo.freitas@academico.domhelder.edu.br.

Referências

Ahmed, I., Ahmed, N., Batool, K., Mahmood, J., Fatima, E., and Arshad, P. (2026). *Performance Comparison of C++, Java, and Python: A Comprehensive Analysis of Programming Languages*. Springer. DOI: 10.1007/978-981-95-2212-5_24.

Bradski, G. (2000). The opencv library. *Dr. Dobb's Journal of Software Tools*, 25(11):120–126. Disponível em: `https://www.proquest.com/trade-journals/opencv-library/docview/202684726/se-2`.

Hasan, R. T. H. and Sallow, A. B. (2021). Face detection and recognition using opencv. *Journal of Soft Computing and Data Mining*. DOI: 10.30880/jscdm.2021.02.02.008.

Iglovikov, V. (2025). Need for speed: A comprehensive benchmark of jpeg decoders in python. DOI: 10.48550/arXiv.2501.13131.

Kaehler, A. and Bradski, G. (2017). *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. O'Reilly Media, Sebastopol. Disponível em: `https://www.oreilly.com/library/view/learning-opencv-3/9781491937983/`.

Lutz, M. and Ascher, D. (1999). *Learning Python*. O'Reilly Media.

Meyers, S. (2014). *Effective Modern C++*. O'Reilly Media.

Nanz, S. and Furia, C. A. (2015). A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Soft-*

ware Engineering, volume 1, pages 778–788. IEEE. DOI: 10.1109/ICSE.2015.90.

OpenCV Contributors (2025). `https://github.com/opencv/opencv/wiki/OpenCV-Change-Logs`.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., and Saraiva, J. (2017). Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pages 256–267. DOI: 10.1145/3136014.3136031.

Prechelt, L. (2000). An empirical comparison of seven programming languages. *Computer*, 33(10):23–29. DOI: 10.1109/2.876288.

Ribas, E. R. L. D. (2025). Github - atom organization page. Disponível em: `https://github.com/Atom-Laboratory`.

Schildt, H. (2019). *Java: The Complete Reference, Eleventh Edition*. McGraw-Hill Education.

SEBESTA, R. W. (2018). *Conceitos de Linguagens de Programação*. Bookman, 11th edition.

Šrajer, F., Kukelova, Z., and Fitzgibbon, A. (2018). A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning. *Optimization Methods and Software*. DOI: 10.1080/10556788.2018.1435651.

Szeliski, R. (2022). *Computer Vision: Algorithms and Applications*. Springer. DOI: 10.1007/978-3-030-34372-9.

TIOBE Software (2026). Tiobe index. Disponível em: `https://www.tiobe.com/tiobe-index/`.

Tukey, J. W. (1977). *Exploratory Data Analysis*, volume 2. Addison-Wesley.