Università di Catania

Department of Mathematics and Computer Science
Master's Degree in Computer Science (MSc)

Dott. Andrew R. Darnall

Statistical Region Merging Benchmark Report

Course - Multimedia e Laboratorio

Academic Year 2024 - 2025

Contents

Chapter 1

Overview

In this report, I present the work undertaken throughout the development of a study conducted as part of the 'Multimedia e Laboratorio' course. This course, aimed at graduate students, delves into advanced concepts in image processing and traditional (pre-deep learning) computer vision techniques.
The primary goal of this study is to evaluate the performance of different implementations of the Statistical Region Merging (SRM) algorithm under various constraints, including computational resources and execution time.
To facilitate this evaluation, the SRM algorithm was implemented in three programming languages—Python, C++, and Rust—utilizing the OpenCV library in each case for image input/output and manipulation tasks. The implementations, along with the rest of the study materials, are available in the associated GitHub repository.
A grid search was performed on several parameter values to assess the performance of each implementation. Performance metrics were primarily based on the time utility provided by the Linux operating system. The results were collected for each implementation, and a preliminary yet insightful exploratory data analysis was conducted to further interpret the findings.

Chapter 2

The Statistical Region Merging Algorithm

Image segmentation is a critical task in computer vision and image processing, aiming to partition an image into multiple segments or regions. Each region corresponds to a set of pixels that share certain characteristics, such as color, intensity, or texture. The objective of segmentation is to simplify the representation of an image or make it more meaningful and easier to analyze. It is a fundamental step in many image processing applications, including object detection, image recognition, medical imaging, and scene understanding.

Segmentation serves as a bridge between low-level image processing techniques, such as pixel-wise operations, and high-level understanding tasks, such as object classification. The idea is to divide an image into areas that are homogeneous according to specific criteria, making subsequent analysis more manageable and more efficient.

Before the advent of deep learning techniques, numerous segmentation algorithms were developed. These classical image segmentation algorithms can be broadly categorized into the following types:

Thresholding is one of the simplest and most widely used techniques for image segmentation. The basic idea is to classify pixels into two or more classes based on their intensity values. A global or local threshold is applied to separate the foreground from the background. In binary thresholding, pixels with intensity values above a certain threshold are classified as foreground, while pixels below the threshold are considered background.

- **Global Thresholding:** A single threshold value is used for the entire image. Otsu's method is a well-known approach that automatically determines the optimal threshold by minimizing intra-class variance.
- **Local Thresholding:** In this approach, threshold values are computed for each pixel or neighborhood based on local intensity distributions, making it suitable for images with varying lighting conditions.

Edge detection algorithms aim to identify boundaries between different regions of an image by detecting significant changes in pixel intensity. The goal is to locate discontinuities or edges within the image, which can then be used to segment the image. Common edge detection methods include the Canny edge detector, the Sobel operator, and the Laplacian of Gaussian (LoG). While edge-based segmentation is effective in detecting boundaries, it is often sensitive to noise and may miss finer details in the image.

Region-based segmentation methods are designed to group pixels into regions based on certain properties, such as similarity in color, texture, or intensity. These methods aim to find large, contiguous regions in the image that exhibit homogeneity in terms of the chosen feature.

- **Region Growing:** Starts with seed points and adds neighboring pixels that are similar to the seed.

- **Region Splitting and Merging:** Divides the image into smaller regions and then merges those that are homogeneous.
- **Watershed Segmentation:** Treats the image as a topographic surface and segments it into basins based on pixel intensity.

Clustering algorithms aim to group pixels based on similarity without any prior knowledge of the image structure. The most common clustering algorithm used in segmentation is K-means clustering, which assigns pixels to K distinct clusters based on their feature similarity.

- **Fuzzy C-means Clustering:** A variant of K-means that allows a pixel to belong to multiple clusters with varying degrees of membership. These algorithms are effective for segmenting images with diverse textures or color patterns.

Graph-based algorithms treat image segmentation as a graph partitioning problem. In these methods, the image is represented as a graph, where each pixel is a node, and edges between nodes represent the similarity between neighboring pixels.

- **Normalized Cut, Graph Cuts, Min-Cut/Max-Flow:** These are powerful graph-based algorithms that can handle complex segmentation tasks, such as segmenting images with weak boundaries.

One of the region-based segmentation techniques is Statistical Region Merging (SRM), a sophisticated algorithm designed to address the limitations of earlier region-based algorithms. SRM is a powerful approach to image segmentation that combines statistical information about image regions with a

merging strategy to partition the image into meaningful regions.

The SRM algorithm was developed by Adrian B. Gibbons, Andrew D. King, and Peter J. Rousseeuw in 1999. The key idea behind SRM is to merge small regions based on statistical tests to form larger homogeneous regions. Unlike earlier methods, which only relied on pixel intensity values or local features, SRM also incorporates statistical tests to ensure that the merged regions have a strong similarity in their statistical properties.

The SRM algorithm was initially designed for multispectral remote sensing images but was soon found to be applicable to a wide range of image segmentation tasks, including medical imaging, object recognition, and natural scene segmentation.

The SRM algorithm works by first over-segmenting the image into a large number of small regions. This over-segmentation provides a fine-level division of the image and helps preserve detailed information that may be lost in a global segmentation process. The core of SRM is the statistical test used to decide whether two adjacent regions should be merged.

The merging process follows these steps:

1. **Initial Region Creation:** The first step in SRM is to divide the image into small, homogeneous regions based on pixel intensity or other local features (such as color, texture, or edge information). This can be done using techniques like k-means clustering or superpixel segmentation.

2. **Similarity Measurement:** For each pair of neighboring regions, a similarity measure is computed. Typically, this measure is based on the statistical difference between the mean intensities or features of the two regions.

3. **Statistical Test:** The algorithm uses a statistical test (such as a t-test) to determine whether the two regions are statistically similar. If the regions are deemed similar, they are merged into a single region. If they are different, they remain separate.

4. **Region Merging:** The process of merging continues iteratively, with small regions being merged to form larger, more homogeneous regions based on the statistical test.

5. **Stopping Criteria:** The merging process stops when no further merging is possible, meaning that all the remaining regions are statistically distinct from each other.

The strength of SRM lies in its ability to adapt to the structure of the image and its reliance on statistical principles to ensure that merged regions are truly homogeneous. This makes SRM particularly effective in segmenting images with complex textures and varying intensity levels.
The SRM algorithm offers several advantages over other segmentation techniques:

- **Robustness:** SRM is robust to noise and can handle variations in lighting and color. The statistical approach ensures that only regions with statistically significant similarities are merged, making the algorithm less susceptible to random variations.

- **Flexibility:** SRM can be applied to a wide range of image types, including grayscale, color, and multispectral images. It can also be adapted to different feature sets (e.g., color, texture, or edge information).

- **Adaptive Region Formation:** SRM produces an adaptive segmentation that accurately captures the inherent structures in the image, resulting in regions that are meaningful and relevant to the application at hand.
- **Better Preservation of Boundaries:** The over-segmentation step preserves fine boundaries in the image, which can be beneficial for downstream tasks such as object detection or boundary-based classification.

Despite its many advantages, the SRM algorithm does have some limitations:

- **Computational Complexity:** SRM can be computationally expensive, particularly for large images. The need to calculate statistical tests for every pair of neighboring regions in the image can lead to high computational costs, especially when dealing with high-resolution images.
- **Sensitivity to Initial Segmentation:** The quality of the final segmentation depends heavily on the initial over-segmentation step. If the initial regions are poorly defined, the merging process may fail to capture the correct boundaries.
- **Parameter Sensitivity:** SRM's performance can be sensitive to the choice of parameters, such as the initial region size, the similarity measure, and the statistical test threshold. Choosing the right parameters often requires experimentation or domain-specific knowledge.
- **Limitations in Handling Complex Textures:** While SRM is robust to noise and variations in intensity, it may struggle with images that have complex or highly textured regions, where the statistical differences

between neighboring regions may not be pronounced enough for the merging process to be effective.

The Statistical Region Merging (SRM) algorithm represents a powerful and sophisticated approach to image segmentation, particularly for images with complex and varied characteristics. While it is computationally demanding and requires careful parameter tuning, SRM excels in producing high-quality segmentations that are both robust and adaptive. Its reliance on statistical tests to guide region merging makes it particularly effective in scenarios where regions have significant statistical differences, which is often the case in real-world image processing tasks.

Despite the emergence of deep learning-based segmentation techniques in recent years, SRM and other traditional methods continue to be valuable tools in many applications, offering interpretable results and efficient performance for specific tasks. As image segmentation remains a cornerstone of computer vision, algorithms like SRM contribute significantly to the continued advancement of the field.

Chapter 3

The Implementations of the SRM Algorithm

The algorithm has been implemented in three widely-used programming languages: Python, C++, and Rust. The choice of these languages is driven by both practical and performance considerations. Each language offers distinct advantages that make them well-suited for specific aspects of the algorithm's design, particularly in the context of Computer Vision (CV) and Image Processing.

Python has established itself as a versatile, high-level programming language and is considered a 'jack of all trades' in the software development world. It is extensively used in fields ranging from web development to machine learning, data science, and, of course, Computer Vision and Image Processing. Python's popularity can be attributed to its easy syntax, extensive standard library, and vast ecosystem of third-party libraries and frameworks. These features make Python an ideal choice for rapid prototyping and experimentation.

In the domain of Computer Vision, Python's role cannot be overstated. It is the preferred language for many researchers and professionals due to its wide range of powerful libraries, such as OpenCV, Scikit-Image, and Pillow, among others. These libraries provide optimized implementations of complex image processing algorithms and make it easier to integrate machine learning models for vision-based tasks. Python's ability to interface seamlessly with low-level languages like C and C++ further extends its capabilities, enabling efficient execution of computationally intensive tasks while maintaining ease of use. However, it is important to note that Python, being an interpreted language, may not always provide the best performance for certain use cases. This is particularly true for resource-intensive applications that require low-latency operations or high-performance computing. Performance bottlenecks can arise unless certain underlying modules are implemented in highly optimized languages like C or C++, which is the case with libraries such as OpenCV.

Despite these potential performance concerns, Python remains a dominant force in the field due to its accessibility, rich ecosystem, and the growing popularity of high-performance computing techniques.

C++ remains the gold standard when it comes to writing highly efficient software, particularly for systems that require low-level memory management and real-time processing. As the language in which OpenCV—arguably the most widely-used library in Computer Vision—is written, C++ excels in providing the necessary performance for handling large-scale, computationally demanding tasks. OpenCV's C++ core allows for seamless integration with hardware and offers fine-grained control over system resources, making it a preferred choice for time-sensitive applications such as robotics, autonomous

vehicles, and real-time image processing.
In addition to its performance benefits, C++ enables developers to write elegant, object-oriented code, making it easier to scale and maintain complex systems. The language provides direct access to memory, allowing for manual optimization and fine-tuning of computational processes. This is crucial in fields like Computer Vision, where algorithms often involve intensive matrix operations, image transformations, and pixel manipulations, all of which benefit from the low-level performance optimizations C++ offers.

Rust is a relatively newer player in the programming language landscape but has quickly gained recognition for its focus on performance, safety, and concurrency. While Rust has not yet achieved the same level of maturity in the domain of Image Processing as Python or C++, its design principles make it an excellent candidate for systems that require both high performance and memory safety.
Rust's ability to handle concurrent operations with minimal risk of race conditions, combined with its zero-cost abstractions, positions it as a promising alternative for parallel image processing tasks. The growing interest in Rust for systems programming, combined with its strong support for modern software development practices, has led to an increasing adoption of the language in performance-critical fields, including AI and Machine Learning.

A crucial consideration in the implementation of the algorithm across Python, C++, and Rust was maintaining consistency in the version of OpenCV used across all three languages. OpenCV is the backbone of many image processing applications, providing an extensive set of tools and functions for tasks such as object detection, image segmentation, and feature extraction. The consis-

tency of OpenCV versions ensured that the algorithm's behavior was uniform across different implementations, which was critical for accurate benchmarking and performance comparisons.

In both Python and Rust, the OpenCV library is accessed through wrappers of the installed native OpenCV library (libopencv-dev). This allowed for seamless integration of OpenCV's functionalities into the respective language environments, ensuring that the same set of image processing tools was available regardless of the implementation language.

Chapter 4

The Benchmark of the Implementations

The benchmark was conducted using a grid search encompassing a total of 1,225 values per implementation. Each input parameter corresponds to a specific mechanism of the implementation, with the Q parameter controlling the complexity of the distribution used for the statistical test. Additionally, the parameters for maximum regions and minimum size account for all possible cases of over- and under-merging in the segmentation algorithm.

Following the execution, which is documented in the Jupyter Notebook available in the previously mentioned GitHub repository, I obtained three distinct DataFrames, each containing the data from individual runs. These DataFrames include the parameters, overall user time, system time, and CPU usage percentage associated with each specific parameter set.

Subsequently, I described the contents of the DataFrames to analyze the statistical moments present within the dataset.

In [15]:
```python
# Running the Python implementation's Grid Search
PYTHON_ITERATION = 0

for bsd_image in bsd_images:
    bsd_image = "./images/" + bsd_image
    for q in q_params:
        for max_reg in max_reg_params:
            for min_size in min_size_params:
                try:
                    # 1) run the program on the speific parameter set
                    result_str = py_benchmark_program(bsd_image, str(q), str(max_reg), str(min_size))
                    # 2) create a record from the obtained information
                    new_record = generate_record("python", q, max_reg, min_size, result_str)
                    # 3) append the newly found record to the reference DataFrame
                    py_df = pd.concat([py_df, pd.DataFrame([new_record])], ignore_index=True)
                    print(f"({PYTHON_ITERATION} / {tot_runs}) Processed [{bsd_image}] - [{q}, {max_reg}, {min_size}]")
                    PYTHON_ITERATION += 1
                except Exception as e:
                    print(f"{e}")
```

```
(0 / 1225) Processed [./images/seg_img_346.jpg] - [28, 8, 0.001]
(1 / 1225) Processed [./images/seg_img_346.jpg] - [28, 8, 0.0032500000000000003]
(2 / 1225) Processed [./images/seg_img_346.jpg] - [28, 8, 0.0055000000000000005]
(3 / 1225) Processed [./images/seg_img_346.jpg] - [28, 8, 0.007750000000000001]
(4 / 1225) Processed [./images/seg_img_346.jpg] - [28, 8, 0.01]
(5 / 1225) Processed [./images/seg_img_346.jpg] - [28, 9, 0.001]
(6 / 1225) Processed [./images/seg_img_346.jpg] - [28, 9, 0.0032500000000000003]
(7 / 1225) Processed [./images/seg_img_346.jpg] - [28, 9, 0.0055000000000000005]
(8 / 1225) Processed [./images/seg_img_346.jpg] - [28, 9, 0.007750000000000001]
(9 / 1225) Processed [./images/seg_img_346.jpg] - [28, 9, 0.01]
(10 / 1225) Processed [./images/seg_img_346.jpg] - [28, 10, 0.001]
(11 / 1225) Processed [./images/seg_img_346.jpg] - [28, 10, 0.0032500000000000003]
(12 / 1225) Processed [./images/seg_img_346.jpg] - [28, 10, 0.0055000000000000005]
(13 / 1225) Processed [./images/seg_img_346.jpg] - [28, 10, 0.007750000000000001]
(14 / 1225) Processed [./images/seg_img_346.jpg] - [28, 10, 0.01]
(15 / 1225) Processed [./images/seg_img_346.jpg] - [28, 11, 0.001]
(16 / 1225) Processed [./images/seg_img_346.jpg] - [28, 11, 0.0032500000000000003]
(17 / 1225) Processed [./images/seg_img_346.jpg] - [28, 11, 0.0055000000000000005]
(18 / 1225) Processed [./images/seg_img_346.jpg] - [28, 11, 0.007750000000000001]
```

Figure 4.1: Code for the gridsearch on the Python implementation

To gain initial insights, I conducted a simple Exploratory Data Analysis (EDA) to identify potential patterns within the data. This preliminary analysis informed subsequent queries for deeper exploration.

To begin the analysis, I visualized the data using boxplots for user time, system time, and CPU percentage usage. This visualization revealed several noteworthy insights:

- **User Time**: The Python implementation exhibited the highest runtime, whereas the Rust implementation achieved the lowest runtime.
- **System Time**: While the C++ implementation had the lowest system time, as expected, the Rust implementation recorded the highest system time. However, all three implementations displayed overall similar system runtimes.
- **CPU Percentage Usage**: Despite the distribution being right-skewed, the Python implementation demonstrated the lowest CPU consumption, with the C++ implementation following closely behind. Surprisingly, the Rust implementation showed the highest CPU usage, contrary to expectations given Rust's reputation for simplicity and efficiency.

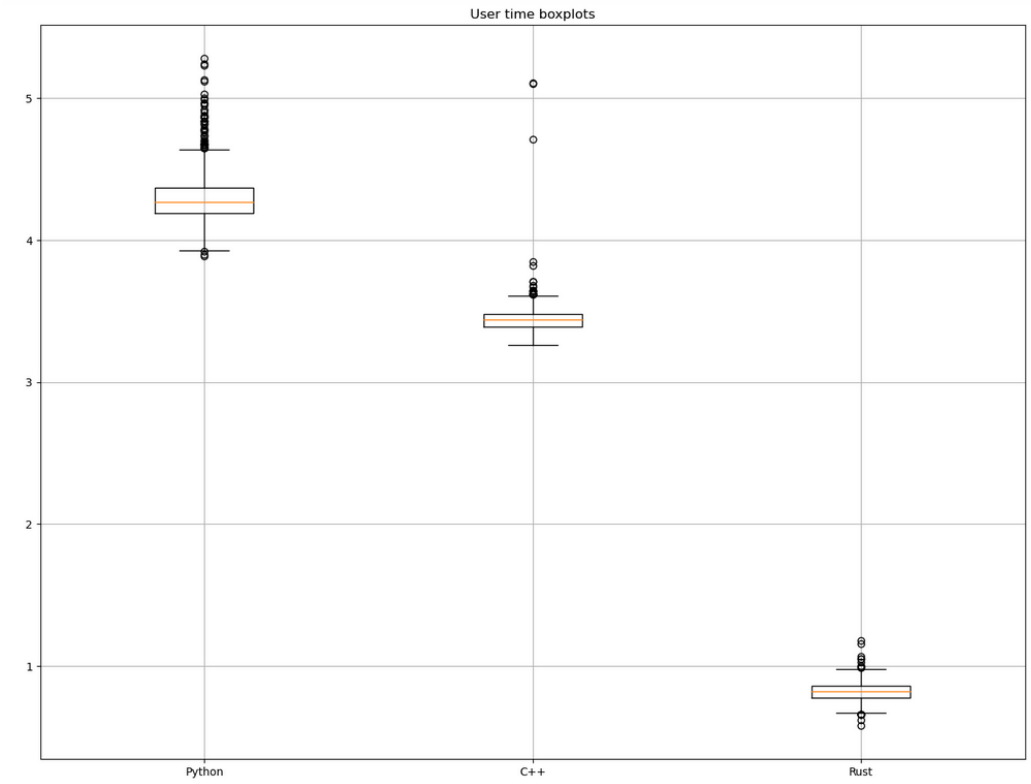

Figure 4.2: Boxplots of the User time of the implementations

The benchmarking process utilized the `time` utility provided in Linux operating systems, specifically in Debian-based distributions. The tests were conducted on five randomly selected images from the BSDS500 segmentation dataset segmentation dataset.

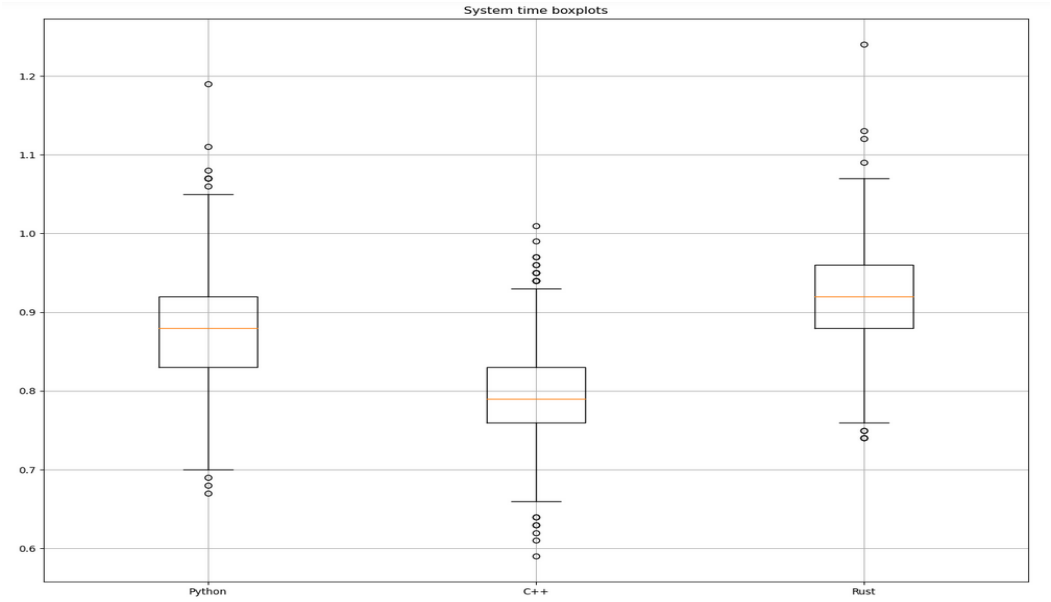

Figure 4.3: Boxplots of the System time of the implementations

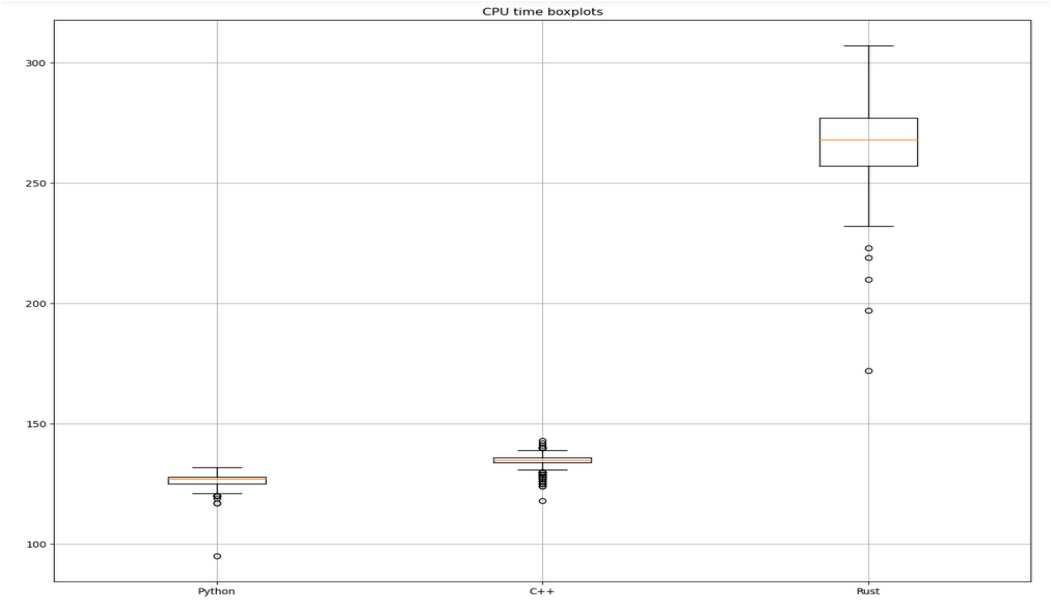

Figure 4.4: Boxplots of the CPU % time of the implementations

I subsequently performed statistical tests, specifically t-tests, to validate the observed differences between the distributions. The results of all tests yielded very low p-values, indicating that the differences identified in the boxplots are statistically significant.

Chapter 5

Conclusions

In conclusion, based on the data analysis performed on the results from benchmarking the various implementations, it can be concluded that, if resource consumption is of paramount importance, the Python implementation emerges as the most suitable option. However, it should be noted that Python is not highly optimizable without resorting to the creation of a highly optimized Python-wrapper C-module, potentially incorporating assembly code for enhanced speed and performance.

Conversely, if speed is the primary consideration, the Rust programming language proves to be the most favorable choice, contrary to initial expectations.

Thank you for your attention,
Dott. Andrew R. Darnall