# Project Report: C-WildWater

Hydraulic Network Analysis and Visualization

**Pre-Eng 2 – 2025-2026**

**Project Team:**
Myriam Bensaid
Sheryne Ouarghi
Matthieu Vannereau

Date: December 19, 2025

# Contents

## 1 Introduction

This project, titled **C-WildWater**, immersed us in the heart of big data processing. Our mission was to design a software pipeline capable of analyzing a complex water distribution network in France. The challenge was not only to read a CSV file but to do so intelligently: extracting statistics, identifying leaks, and visualizing everything clearly, all using C and Shell.

We had to juggle memory management, the implementation of efficient data structures (AVL trees), and task automation, which allowed us to consolidate our technical skills.

## 2 Team Organization

For this project, we chose full collaboration. Rather than rigidly compartmentalizing tasks, we worked in constant synergy to ensure that every team member understood and mastered the entire codebase.

Although specific areas of responsibility naturally emerged to improve efficiency, mutual aid was the driving force behind our progress:

- **Sheryne** led the visualization part with Gnuplot, ensuring that our raw data became readable graphs. However, the interpretation of results and scale adjustments were validated collectively.
- **Myriam** structured the code architecture, managed utility functions, and supervised Git merges. She worked in pairs with other members to integrate their modules without breaking the existing base.
- **Matthieu** proposed the logic for complex calculations (leaks, bonus). The implementation of these algorithms and their debugging were carried out by multiple hands to ensure their robustness.

Debugging, particularly regarding pointer and memory management, was a true team effort, often gathering us around a single screen to solve the toughest problems.

## 3 Technical Architecture

Our solution rests on three pillars:

1. **The Conductor (Shell):** The ‘myScript.sh‘ script automates everything. It compiles, cleans the environment, launches the C program with the correct parameters, and triggers Gnuplot.
2. **The Engine (C):** This is where the magic happens. The program reads the CSV, builds an AVL tree in memory to store stations, and performs calculations (sums, differences, detections).
3. **The Showcase (Gnuplot):** Once the data is processed, Gnuplot generates the PNG images that we analyze.

## 4 Choice of Data Structures

Very quickly, we understood that a simple linked list would be too slow given the size of the file. We therefore opted for an **AVL Tree**.

```c
typedef struct Station {
    unsigned int id;            // Unique ID
    long capacity;              // Capacity
    long load;                  // Load
    struct Station *left;       // AVL Balancing
    struct Station *right;
    int height;
    // ...
} Station;
```

Listing 1: Our Station Structure

This choice guarantees that even in the worst-case scenario, retrieving a station to update its data is done very quickly (logarithmic complexity).

## 5 Challenges and Lessons Learned

This project was not without its hurdles. Here are the main obstacles we had to overcome together:

### 5.1 The Parsing "Nightmare"

At first, we underestimated the complexity of the CSV file. Between sometimes ambiguous separators and variable data formats, we encountered several "Segmentation Faults". It took a collective effort to test and reinforce the reading functions to make them tolerant of irregularities.

### 5.2 AVL Logic

Understanding AVL theory is one thing; implementing it is another. Managing rotations (simple and double) gave us a hard time. We spent a lot of time drawing trees on paper to understand why certain nodes were getting lost during rebalancing.

### 5.3 Bonus Integration (Gnuplot)

For the bonus, it wasn't enough to just output numbers. We had to format the data in a very specific way so that the Gnuplot script could interpret the columns as stacks (`rowstacked`). Making the C logic communicate correctly with the graphical script's expectations required several iterations of joint testing.

## 6 Optimization and Credits

In a performance perspective, we explored multi-threading. As shown in the `multiThreaded.c` files, we used `mutexes` to secure file writing during parallel processing.

**Credits:** The idea and technical basis for this multi-threaded implementation come from the work of **Tiago Charette**. We drew inspiration from his GitHub repository[1], which we adapted to our structure.

## 7 Visual Results

Here are the fruits of our labor, automatically generated by our processing pipeline.

### 7.1 Capacities (Max Mode)

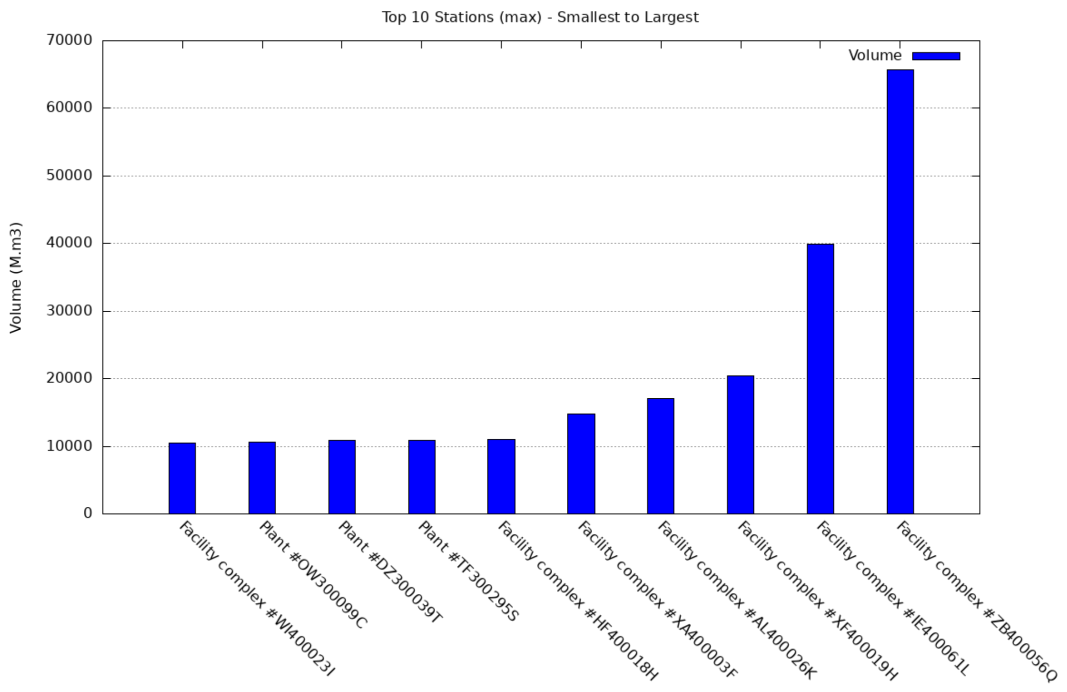

Figure 1: The top 10 stations with highest capacity

[1] `https://github.com/TogExe/MultiThreaded`

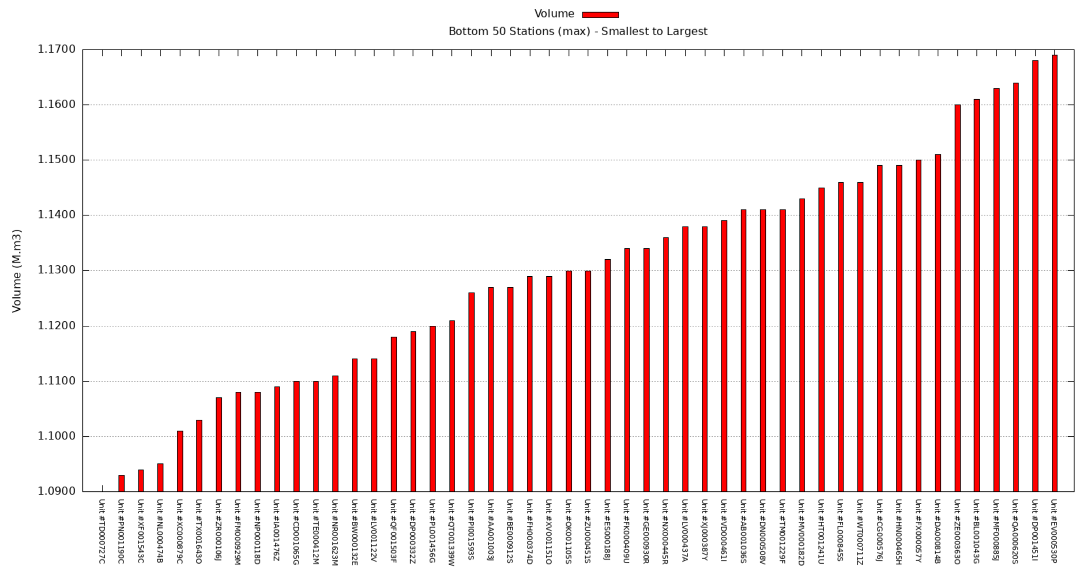

Figure 2: The 50 stations with the smallest capacities

## 7.2 Real Flows (Real Mode)

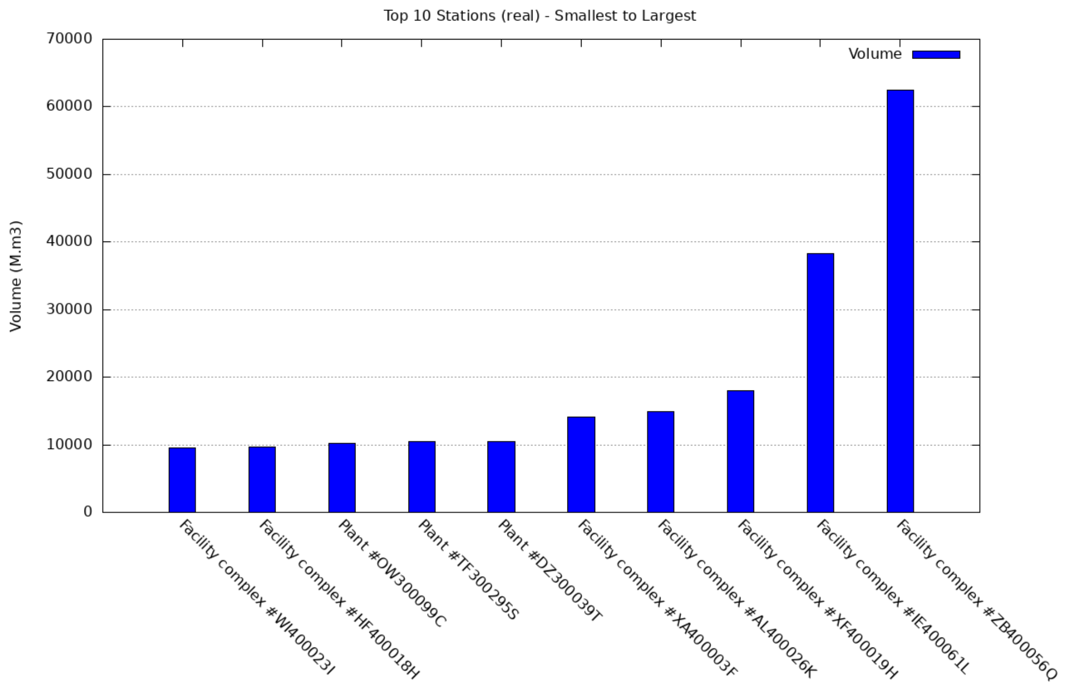

Figure 3: Top 10 real flow rates

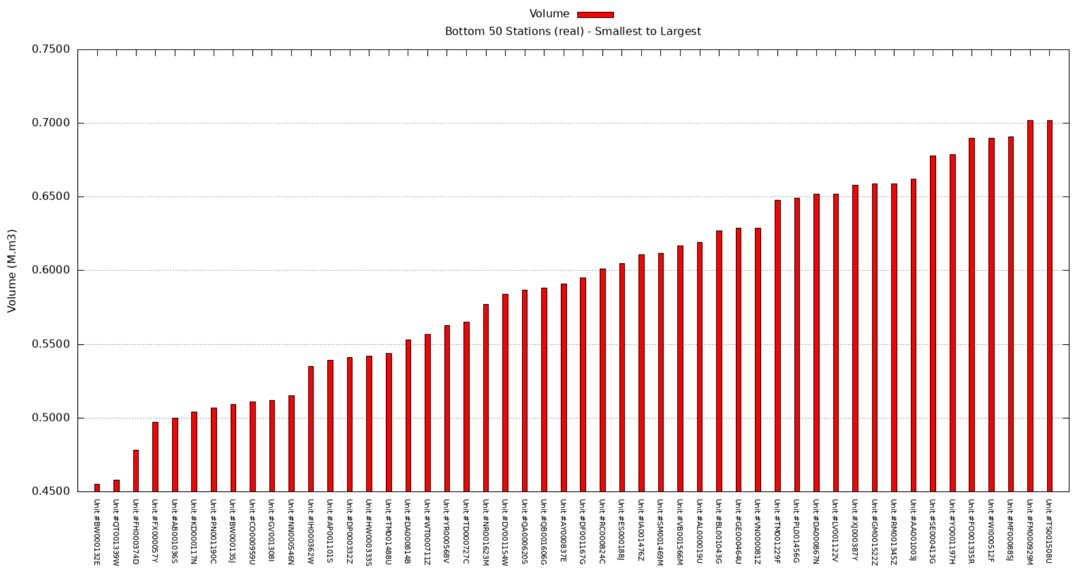

Figure 4: The 50 smallest flow rates

## 7.3 Sources (Src Mode)

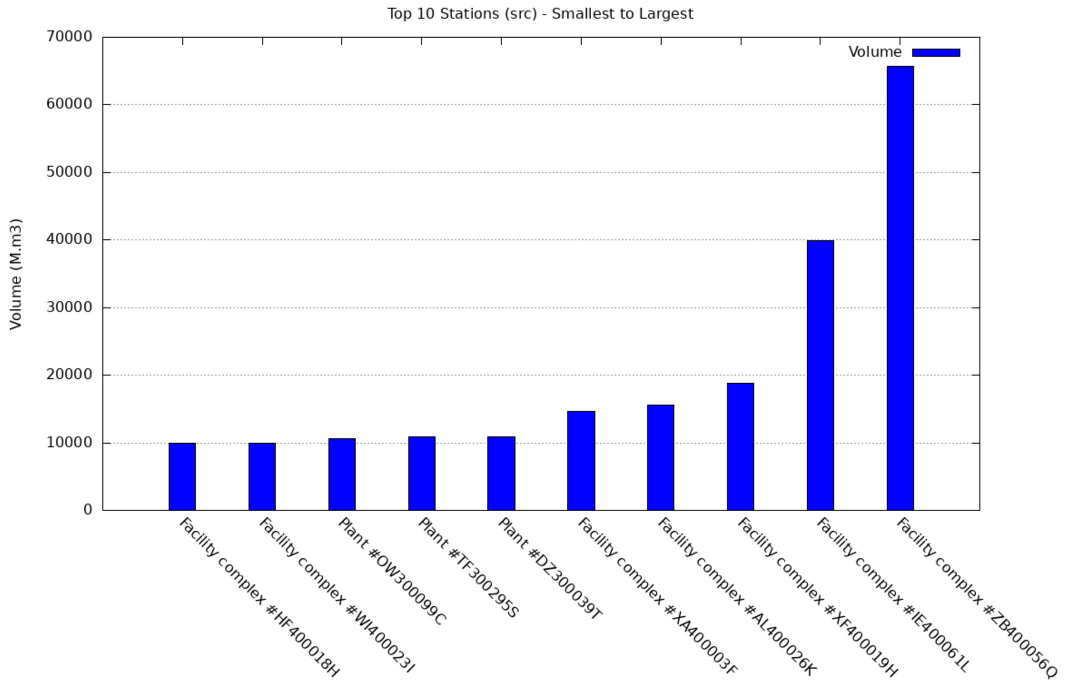

Figure 5: Top 10 water sources

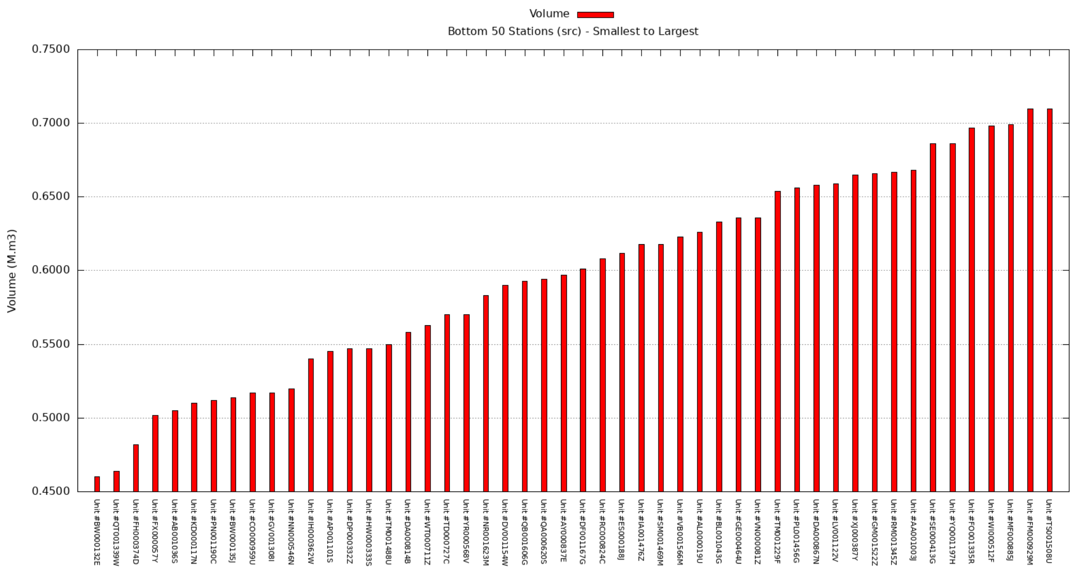

Figure 6: The 50 smallest sources

## 7.4 Advanced Analysis (Bonus)

These graphs are particularly interesting as they show the distribution: green for empty space, red for losses (leaks or overconsumption), and blue for normal flow.

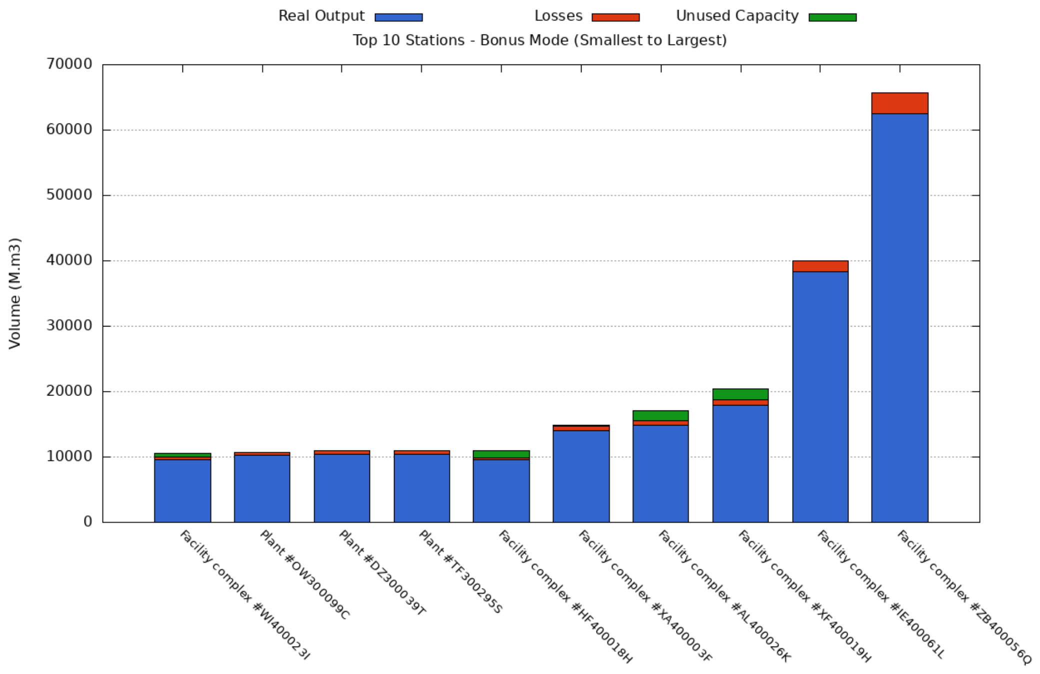

Figure 7: Detail of the Top 10 (with losses and free capacity)

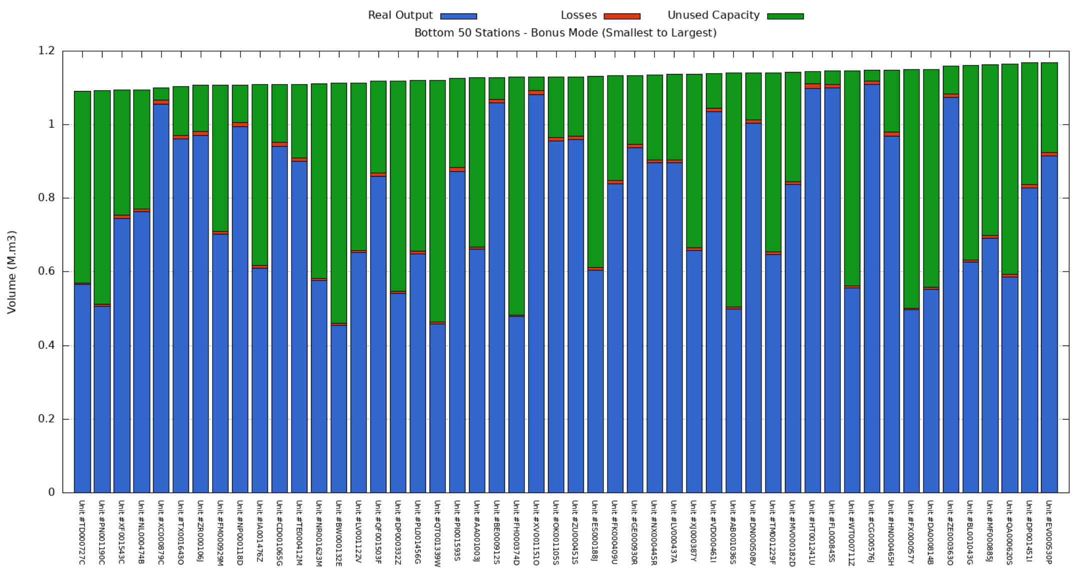

Figure 8: Detail of the 50 smallest stations

## 8 Conclusion

In the end, C-WildWater was a very educational project. We learned to work as a team on a common codebase, manage merge conflicts, and above all, debug complex memory issues. The satisfaction of seeing the graphs generate correctly at the end was well worth the hours spent fixing NULL pointers!