CPP Programming tutorial report

Zine el Abidine Belkadi

July 4, 2024

1 Technical Background

1.1 Linux

Linux, the operating system you use natively, particularly Ubuntu 22.04 which is the distribution I am mainly using at work and in my personal life, operates on foundational principles of open-source collaboration and modularity. Created by Linus Torvalds in 1991, Linux is built around a powerful kernel that manages hardware resources, memory, and facilitates communication between software and hardware components.
Ubuntu, a widely-used Linux distribution, builds upon this kernel with a user-friendly interface and a comprehensive suite of pre-installed applications and utilities. Linus Torvalds' initial vision of Linux as a freely available and customizable operating system has thrived, thanks to its open-source nature. This allows global developers and enthusiasts to contribute improvements and innovations continuously.
Ubuntu 22.04, with its Long Term Support (LTS) designation, ensures stability and reliability for desktop and server environments alike. Its package management system, powered by APT (Advanced Package Tool), simplifies software installation and updates by managing dependencies effectively.
Linux's security model, emphasizing file permissions, user accounts, and robust networking capabilities, enhances system integrity and resilience against vulnerabilities. This design ethos not only prioritizes security but also empowers users to tailor their computing experience extensively.
In essence, by using Ubuntu 22.04, you're embracing the strengths of Linux—stability, security, flexibility, and community-driven innovation—founded on Linus Torvalds' original vision of an open-source operating system that continues to evolve and meet diverse computing needs worldwide. [3] [2]

1.2 Git

Another invention of Linus Torvalds is Git not to confuse with Github or Gitlab was created in 2005, revolutionized version control systems by offering a distributed approach that enhances collaboration and manages software development projects efficiently. The key features of Git are that most operations in Git are performed locally, which makes it fast and efficient. It also rovides robust support for branching and merging, enabling developers to work on different versions of a project simultaneously and merge changes back into the main codebase. and finally it maintains a complete history of all changes made to the codebase, facilitating easy navigation and rollback to previous versions if needed.
Github and Gitlab in the other hand are web-based hosting service for Git repositories. It provides a platform for developers to store, manage, and collaborate on projects using Git. the main features of these web UIs are that they host remote repositories where developers can push their local Git repositories, making it accessible to others. It is also sotred in the cloud so there is no need for an on Prem solution. They also offers features like issue tracking, pull requests(or merge request in Gitlab), code review tools, and project management tools, enhancing collaboration among team members.
Gitlab in the otherhand offers various features that Github does not has at the first glance. like an integrated DevOps platform and a Built-in CI/CD which is the direct concurent of github actions.

1.2.1 How does git work?

here is a small overview on how GIT works. Git operates with a repository (repo) model, where each project has its own repository containing all project files and their complete history of changes. To

initilize project using Git following command is required:

```
git init
```

Before Git tracks changes, files need to be added to the staging area. This is done using git add. For example, to add all files in the current directory to the staging area:

```
git add .
```

the dot signalizes git that it has to add everything that has changed will get into the staging envirenement.
Once files are added to the staging area, Git records their current state with a commit. Each commit captures a snapshot of the project at a particular point in time, along with a commit message describing the changes. To commit changes:

```
git commit -m "Detailed commit message here"
```

and finally once you are sure, that your code is ready to be reviewed by your piers, you can commit you can push your changes.

```
git push origin main
```

however when working with multiple developers, it is good to not interfer with each other therefore we do not push everything to main/master. Because you might create bugs unwanted issues or just ruin the day for all your colleagues. Therefore git allows allows developers to work on multiple versions of a project simultaneously through branches. To create and switch to a new branch:

```
git checkout -b new-feature
```

Here, new-feature is the name of the new branch. You can now make changes specific to this feature without affecting the main branch (main or master). To switch between branches:

```
git checkout main
```

if you modify your branch anything you can push to your branch

```
git push origin your-branch
```

after your code gets reviewed by other colleagues and senior devlopers and are happy with it and all the tests and checks will pass, you can then merge your branch when your Pull request/merge request will get approved. [10]

1.3 C++ and Cmake

C++ is a powerful and versatile and also scary programming language used widely for system software, game development, embedded systems, and more. It combines procedural, object-oriented, and generic programming features, making it suitable for a wide range of applications.

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Explanation of the C++ Code Snippet: `#include <iostream>`: This line includes the input/output stream library, allowing us to use `std::cout`.`int main() { ... }`: Every C++ program starts execution from the `main` function. `std::cout << "Hello, World!" << std::endl;`: This line outputs "Hello, World!" to the console. `std::cout` is used for output, and `<<` is the stream insertion operator. `return 0;`: Indicates successful termination of the `main` function.

C++ supports various data types such as integers (int), floating-point numbers (float, double), characters (char), and more complex types like arrays, structs, and classes.

```cpp
#include <iostream>

int main() {
    int age = 30;
    double height = 175.5;
    char grade = 'A';

    std::cout << "Age: " << age << std::endl;
    std::cout << "Height: " << height << " cm" << std::endl;
    std::cout << "Grade: " << grade << std::endl;

    return 0;
}
```

C++ uses control structures like if, else, for, while, and switch to control the flow of execution based on conditions.

```cpp
#include <iostream>

int main() {
    int number = 10;

    if (number > 0) {
        std::cout << "Number is positive." << std::endl;
    } else if (number < 0) {
        std::cout << "Number is negative." << std::endl;
    } else {
        std::cout << "Number is zero." << std::endl;
    }

    return 0;
}
```

C++ is highly regarded for its efficiency, versatility, and speed, making it a preferred choice across various domains. The examples provided in this report highlight fundamental aspects such as displaying text, defining variables, and employing control structures for decision-making. C++ has further capabilities including object-oriented programming, template usage, and efficient memory management. These features enable developers to craft sophisticated and scalable software solutions, meeting diverse application requirements effectively. [1, 11]

While writing code in c++ it is most likely the case that you will have multiple components that has to be linked together and also built together. For this we use a tool called CMake CMake is a versatile tool widely used in software development to streamline the build process across different platforms. It simplifies the management of build configurations by generating native build files (like Makefiles or IDE project files) from a unified set of configuration files called CMakeLists.txt. This approach ensures consistency in building software projects on various operating systems, including Windows, macOS, and Linux.

One of CMake's strengths lies in its cross-platform compatibility and flexibility. Developers can write CMakeLists.txt files that specify project requirements, dependencies, and build instructions in a clear and concise manner. This allows for efficient handling of complex project structures and facilitates modularity by organizing components into libraries and executables.

Moreover, CMake supports integration with a wide range of IDEs and build systems, enabling developers to work seamlessly across different development environments. It can generate project files tailored for IDEs like Visual Studio, Xcode, and Eclipse, as well as build systems such as Make, Ninja, and MSBuild. This adaptability makes CMake suitable for both small-scale projects and large-scale enterprise applications.

In practice, developers interact with CMake primarily through command-line tools (cmake and cmake-

gui). The cmake command processes the CMakeLists.txt files to configure and generate the necessary build files according to the specified settings. This promotes the practice of out-of-source builds, where build artifacts are kept separate from the source code, ensuring a clean and organized project structure. Overall, CMake enhances productivity and scalability in software development by providing a robust framework for managing build configurations and dependencies. Its ability to handle complex project requirements and support diverse development environments makes it an indispensable tool for modern software engineering practices. [8, 9]

1.4 CI/CD

In software devlopement one of the key concepts is to write code efficitently in a team and also respond quickly, to new features bugs and further demand from your users or your customers. DevOps is a combination of software devlopement and operations and it is defined a software engineering methodology that makes the processes of devloeping softwares more efficient. While CI/CD is not equal DevOps however CI/CD is an important componenent of DevOps software devloepement CI/CD, which stands for Continuous Integration and Continuous Deployment (or Continuous Delivery), is a set of best practices and tools used in modern software development to automate and streamline the process of building, testing, and deploying applications. Continuous Integration involves automating the integration of code changes from multiple contributors into a shared repository. This is typically achieved through automated build and test processes triggered whenever code changes are committed to version control. CI ensures that all changes are regularly tested and validated, helping to detect integration issues early in the development cycle. Continuous Deployment (or Delivery) extends CI by automating the deployment of applications to production environments once they pass through the CI pipeline. This includes automating tasks such as packaging, deploying, and configuring applications in staging and production environments. Continuous Deployment aims to deliver new features and updates to users quickly and reliably, reducing the time and effort required for manual deployment processes. [4] [7]

1.5 Docker

Docker is a popular platform for developing, shipping, and running applications inside containers. Containers are lightweight, portable, and self-sufficient environments that encapsulate everything needed to run an application, including dependencies, libraries, and configuration settings.

Docker enables containerization by using Docker Engine to create and manage containers. Each container runs as an isolated process, making it easy to package and deploy applications consistently across different environments.

A Dockerfile is a text document that contains all the instructions needed to build a Docker image. It specifies the base image to use, commands to run during the build process, and configuration settings for the container environment.

```
# Use an official Ubuntu runtime as a base image
FROM ubuntu:latest

# Set the working directory inside the container
WORKDIR /app

# Copy the local hello.cpp file to the working directory
COPY hello.cpp .

# Install g++ compiler and necessary build tools
RUN apt-get update && apt-get install -y \
    g++ \
    && rm -rf /var/lib/apt/lists/*

# Compile the C++ program
```

```
RUN g++ -o hello hello.cpp

# Command to run the executable when the container starts
CMD ["./hello"]
```

to build a docker file into a docker image you can use the following

```
docker build -t hello .
```

to run the docker container

```
docker run hello
```

and to finally push it to a registry you have to login to your registry and then push it with

```
docker login myregistry.com  \\
docker push hello myregistry.com/repo/hello:tag
```

Docker simplifies application development and deployment by providing a standardized way to package, distribute, and run software in isolated containers. Its ease of use, portability, and support for automation through Dockerfiles and CLI commands make it a valuable tool for modern software development practices and DevOps workflows.

[5],[6]

2 Project Execution

In this section I will provide an explanation of my Code and the decisions I made throught out the project.

2.1 Code Architecture

The code architecture and file structure

```
/
├── CMakeLists.txt
├── Dockerfile
├── runSolver.sh
└── src/
    └── main/
        ├── config/
        │   ├── param.cpp
        │   └── param.h
        ├── derivatives/
        │   ├── derivative.cpp
        │   └── derivative.h
        ├── fields/
        │   ├── fields.cpp
        │   └── fields.h
        ├── main.cpp
        ├── newtonImpl/
        │   ├── newton.cpp
        │   └── newton.h
        ├── solver/
        │   ├── Algorithm.cpp
        │   └── Algorithm.h
        ├── streamfile/
        │   ├── vtkWriter.cpp
        │   └── vtkWriter.h
        ├── third_party/
        │   └── eigen-3.4.0/..
        └── utils/
            └── utils.hpp
```

in the root of the project we have the cmake file that has the following structure:

```
cmake_minimum_required(VERSION 3.22.1)
project(CPP_tutorial_programming)

set(CMAKE_CXX_STANDARD 20)

# Include Eigen library
include_directories(src/main/third_party/eigen-3.4.0)

# Add VTK directories
find_package(VTK REQUIRED PATHS "~/vtk/build" NO_DEFAULT_PATH)
include(${VTK_USE_FILE})

#find_package(VTK REQUIRED)
#include(${VTK_USE_FILE})

# Add the executable
```

```
add_executable(CPP_tutorial_programming
        src/main/main.cpp
        src/main/config/param.cpp
        src/main/fields/fields.cpp
        src/main/derivatives/derivative.cpp
        src/main/solver/Algorithm.cpp
        src/main/newtonImpl/newton.cpp
        src/main/streamfile/vtkWriter.cpp
)

target_link_libraries(CPP_tutorial_programming ${VTK_LIBRARIES})
```

where I include my executables and libraries that I use. to run my programm I use a bashscript that has defualt arguments to run if you do not specify any. The Solver runs in the main.cpp file where the solver for each time step will run in a for loop. The parameters are setup in the Parameters class which is written as following

```
class Parameters {
public:
    double Lx;
    double Ly;
    static int Nx;
    static int Ny;
    double dt;
    double alpha;
    double beta;
    double gamma;
    int timeStep;
    double dx;
    double dy;

    // Constructor to initialize non-static parameters
    Parameters();

    // Method to set parameters
    void setParams(double Lx, double Ly, int nx, int ny, double dt,
        double alpha, double beta, double gamma,int timeStep);


    // Static member initialization
    static void initializeStaticMembers(int nx, int ny);
};
```

This class will be inherited in all other classes in order to be able to access to the parameters of the simulation such as the number of grid points N_x, N_y and dx etc. another important class is my field class. where I setup the fields that will be used for the calculations like the u^k and u^{k+1}, Δq and the analytical solution. In the field class Methods to initialize the initial conditions and source term are also defined.

```
    void setInitialCond( std::vector<std::vector<double>>& field);
```

All the other class needed for the calculations inherits from these two classes and are then executed in the main.cpp

2.2 CI/CD and containerization

In my project, I set up a Docker environment to facilitate development and deployment. Initially, I created a Dockerfile defining a container image based on Ubuntu, including all necessary dependencies such as external libraries. This Dockerfile was stored within my GitLab repository. After configuring the Docker environment, I proceeded to push the container image to GitLab's container registry, ensuring it was readily accessible for deployment and CI/CD processes.

```
FROM ubuntu:latest

# Set non-interactive mode for apt-get
ENV DEBIAN_FRONTEND=noninteractive

# Update and install dependencies
RUN apt-get update -y \
    && apt-get install -y --no-install-recommends \
    wget \
    build-essential \
    cmake \
    libeigen3-dev \
    libgl1-mesa-dev \
```

```
    && rm -rf /var/lib/apt/lists/*

WORKDIR /tmp
# Download and install VTK (if required)
RUN wget --no-check-certificate https://www.vtk.org/files/release/9.3/
   VTK-9.3.0.tar.gz && \
    tar xf VTK-9.3.0.tar.gz && \
    cd VTK-9.3.0 && \
    mkdir build && \
    cd build && \
    cmake .. && \
    make -j$(nproc) && \
    make install && \
    cd /tmp && \
    rm -rf VTK-9.3.0 VTK-9.3.0.tar.gz

#ENV VTK_DIR=/usr/local/lib/cmake/vtk-9.3

# Set working directory for the application
WORKDIR /app

# Set VTK_DIR environment variable (adjust path if needed)

# Copy application source code, script, and CMakeLists.txt
COPY src/ /app/src/
COPY runSolver.sh /app/
COPY CMakeLists.txt /app/


# Run the application
CMD ["bash","./runSolver.sh"]
```

For continuous integration and continuous deployment (CI/CD), I established a pipeline using GitLab's capabilities. This pipeline utilized the Docker image stored in the GitLab registry. The CI/CD pipeline was structured to automate several key tasks: firstly, building the project using CMake to ensure consistency and reliability across different environments. Secondly, I added a further step which is pushing the latest version of the dockerfile. which will facilitate the devlopement for future new features and Lastly executed tests on a parameter variation bash script against a manually created reference solution, ensuring accuracy and performance.

2.3 Challenges and problemes

One of the biggest challenges I faced during this project was time constraints. Balancing a full-time job as a software engineer, which often involved handling numerous long discusions over complex decisions to make and urgent tasks, left me with limited time to dedicate to this project. Underestimating the workload compounded this challenge, making it difficult to allocate sufficient time and focus.

Another significant hurdle was grasping the mathematical concepts underlying the implicit solver and its algorithm. Understanding these intricate details required extensive research and learning, which added complexity to the development process.

Additionally, I encountered issues with the GitLab runner not executing as expected, leading to considerable time lost troubleshooting. Ultimately, the problem stemmed from system permissions; I discovered that running commands with sudo was necessary to overcome these issues and ensure the pipeline operated correctly.

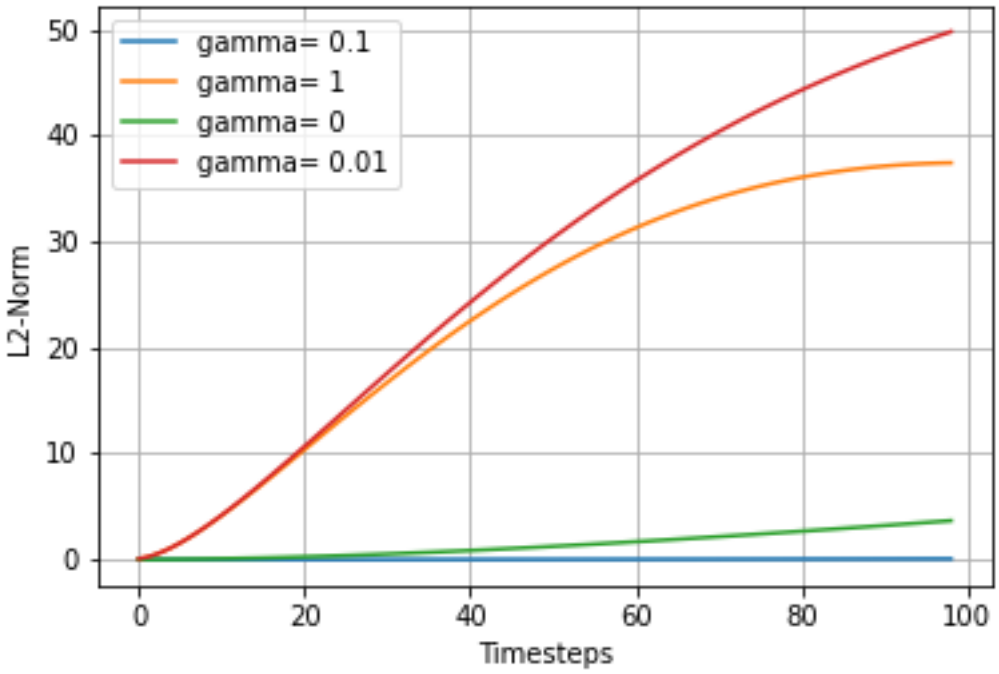
Navigating these challenges underscored the importance of thorough planning, time management, and technical proficiency in executing complex projects effectively. Each obstacle presented valuable learning opportunities and reinforced the importance of resilience and problem-solving skills in overcoming project hurdles. Moreover, despite my efforts, the solver did not yield the expected results. It

exhibited unexpectedly quick convergence, indicating potential errors in the field updating process for subsequent timesteps. This discrepancy was evident in the large L2 error observed during validation, which will be detailed in forthcoming plots.

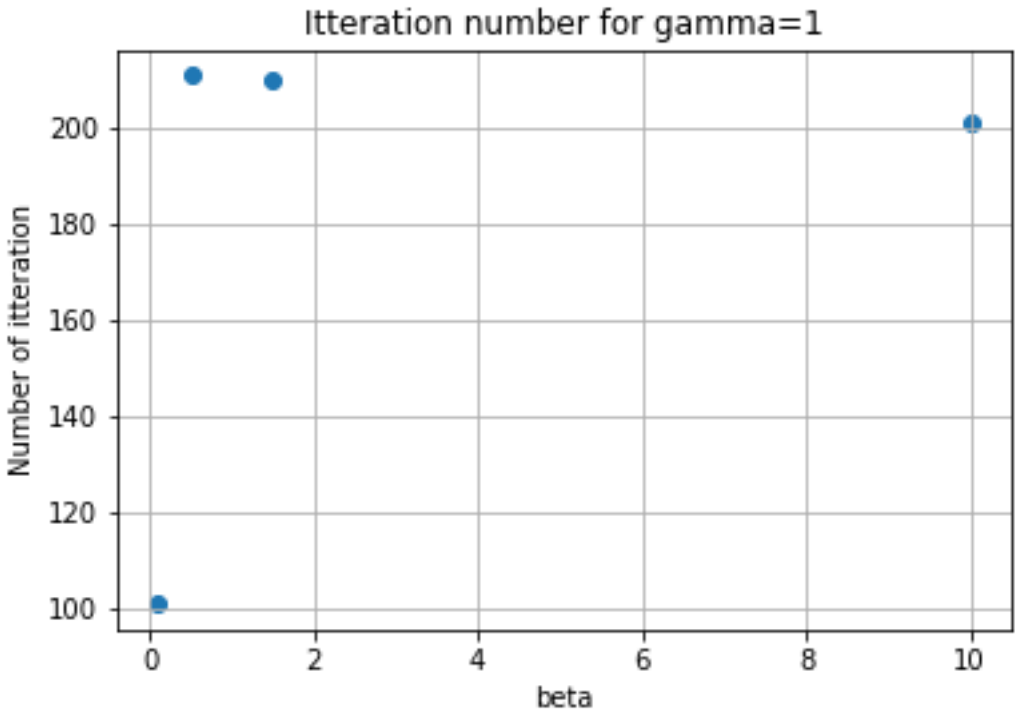
Navigating these challenges highlighted the importance of meticulous planning, effective time management, and technical proficiency in executing complex projects. Each obstacle presented valuable learning opportunities, underscoring the significance of resilience and adept problem-solving skills in overcoming project hurdles effectively.

3 Results and Conclusion

3.1 results



The calculated L2-Norm for different Gamma and $beta = 0$



Number of iteration based on different beta values

These results show clear false result. The L2-norm grows with time while the number of itteration for constant gaamma is around 210 from the 50 that are exepected. This leads me to think that somewhere in my algorithm something went wrong and with more time, I am sure that I will be able to find the mistake.

3.2 Outlook and conclusion

Despite my best efforts, I was unable to complete the development of the code due to time pressures stemming from my full-time job and personal problems. As a result, the solver's output is clearly incorrect, as evidenced by the number of iterations and the L2 norm for the linear case. The numerical solution fails to converge to the analytical solution, indicating a mistake in the implementation of the algorithm.

However, I appreciated the opportunity to work on a university project that exposed students to tools commonly used in software development. I also discovered the Eigen library, which was a valuable addition to my knowledge. Although the project was enjoyable to program, I underestimated the time required to complete it, given that it was only worth 4 CPs. Despite the challenges, the experience was educational and highlighted the importance of careful planning and time management.

References

[1] Various Authors. *cppreference.com*. 2024. URL: https://en.cppreference.com/w/.

[2] Various Authors. *Linux Documentation Project*. 2024. URL: https://www.tldp.org/.

[3] The Linux Foundation. *The Linux Kernel Documentation*. 2024. URL: https://www.kernel.org/doc/html/latest/.

[4] GitHub. *GitHub CI/CD Resources*. 2024. URL: https://docs.github.com/en/actions.

[5] Docker Inc. *Docker Docs*. 2024. URL: https://docs.docker.com/.

[6] Docker Inc. *Docker Reference Documentation*. 2024. URL: https://docs.docker.com/reference/.

[7] GitLab Inc. *GitLab CI/CD Documentation*. 2024. URL: https://docs.gitlab.com/ee/ci/.

[8] Kitware. *CMake Documentation*. 2024. URL: https://cmake.org/documentation/.

[9] Kitware. *CMake Reference Documentation*. 2024. URL: https://cmake.org/cmake/help/latest/.

[10] The Git Project. *Git Documentation*. 2024. URL: https://git-scm.com/doc.

[11] Juan Soulie. *C++ Language Tutorial*. 2024. URL: https://www.cplusplus.com/doc/tutorial/.