POLITECNICO
MILANO 1863

ADVANCED PROGRAMMING FOR SCIENTIFIC COMPUTING

Multiple Screw Extruder Analysis: Investigating Pressure and Temperature

Supervisor: Professor Nicola Paroloini

Team Members

Bharat K. Premkumar, Henrik Bang-Olsen, Ask Kvarven

Table of Content

1. Introduction

1.1 Credits

Our sincere gratitude to Professor Nicola Paroloini, Professor of department of Mathematics at Politecnico di Milano, for his thorough explanation of the concept and his willingness to adapt the project throughout the process.

1.2 Abstract

The aim of this project was to develop and implement a numerical simulation, programmed in C++, to analyze the pressure and temperature distribution within a multiple screw extruder. The primary objective was to construct a simulation framework from the ground up to facilitate an understanding of the temperatures and pressures occurring within the multiple screw extruder during operation. The multiple screw extruder consists of multiple physical modules that are connected to each other in series. During operation the multiple screw extruder's goal is to mix and move the fluid through the different modules.

Employing iterative techniques, including backward and forward sweeps for calculating pressure and temperature, the system seeks to accurately resolve these distributions. For the calculations the user can choose from three different sets of formulas. These formulas are simplified and idealized approximations intended to provide a theoretical framework for understanding changes in pressure and temperature, rather than detailed simulations of real-world properties. The team had a strong focus on creating a modular, flexible and efficient codebase, to ensure a robust foundation for further research and development.

1.3 Motivation

A screw extruder, also denoted as a mixer from here on out, is a machine used in the plastics and polymer industry to process raw materials into a homogeneous, formable mass [1] [2]. The basic mechanism involves one or more rotating screws (helicoidal shafts) inside a heated barrel. As the screws rotate, they convey the raw plastic or polymer from one end of the

extruder to the other, applying shear and compressive forces that heat, mix, melt, and pump the material. During this process, the screws also generate significant pressure within the barrel. This pressure is essential for compacting the material and ensuring the material flows evenly throughout the barrel, thus forming the desired shape as it exits the extruder [2] [3].

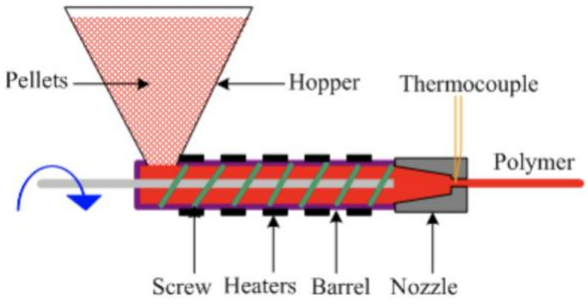


Figure 1.1: Schematic of a typical single screw extruder in the polymer industry [5]

As temperature and pressure within the extruder invariably increase during operation, understanding their distribution is essential for optimizing extrusion processes [2] [4].

2. Model and Numerical Methodology

This chapter will focus on how the mixer is designed and the numerical methodologies used in the simulations.

2.1 Shortcomings

The team would like to point out that the models and calculations used in the simulations do not directly apply to the complexities found in the real world. These models use simplified assumptions to estimate temperature and pressure distributions in the mixer, and rely on a limited set of parameters and theoretical formulas. This leads to some shortcomings in the model's capability.

The main shortcoming is that the simulations fail to consider the varying properties of different materials. The materials have fluid viscosity and density, and by not including these factors, the model provides a generalized view of fluid dynamics that does not reflect the more intricate interactions that occur under practical conditions. Additionally, the model simplifies environmental influences to a single external temperature, ignoring the possibility of other temperature variations that could result from the mixer's operation itself, which could have an effect on the results.

Moreover, the model does not take into account operational variables like the filling rate of the screw or variations in flow rates throughout the mixer. These elements are important for a thorough analysis of the mixing process but are omitted in our simulations. While these omissions help make the computations more manageable, they also restrict the model’s ability to reflect actual conditions accurately.

These limitations mean that the results of the simulations should be interpreted with an understanding that they are based on a theoretical framework that does not capture the full complexity of real operations.

2.2 Composition and properties

The model consists of a mixer that is built up by several physical modules connected in series. The physical module represents one single screw extruder and one mixer consists of

multiple physical modules. At the end of every physical module there is a stop ring where the cross section is narrower, therefore the idea is that pressure will build up further along the screw.

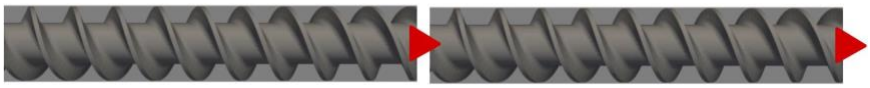


Figure 2.1: one mixer consisting of 2 physical modules. Stop ring at the end of every physical module, illustrated by a red triangle

Figure 2.2: one physical module consisting of many numerical modules.

In the transition from the first physical module to the second physical module, the temperature of the fluid is transferred. Meaning that the temperature from the outlet of the previous screw is the same as the inlet temperature of the adjacent screw. The initial temperature of the fluid, t0, is the inlet temperature of the first screw in the mixer, and is set to 293 kelvin in every test case (20 degrees celsius).

For the pressure the values are not transferred to the next screw. After the stop ring the pressure will drop drastically to the atmospheric pressure. The initial pressure of every screw is set to the atmospheric pressure. The atmospheric pressure, p0, is set to zero.

One screw/physical module consists of many numerical modules. The numerical modules have a temperature and a pressure value. The numerical module also has a RPM, representing how fast the screw spins in this position, and a Q, representing the flowrate of the fluid at this position. The RPM and Q are constants for the whole mixer.

2.3 Numerical Methodology

The numerical methodology is important for understand in the underlying mathematics happening in the code. Therefore, we will in this section we look at the functions and

constants used, the derived calculations for pressure and temperature. As well as some important notes on the iterations and convergence of the temperature and pressure.

2.3.1 Functions and constants

The numerical methodology in our project is based around the calculations of pressure and temperature in the mixer. For the calculations, these constants are being used:

- **TIn**: Input temperature used in the calculations.
- **RPM**: Constant of revolutions per minute.
- **Q**: Constant of flow rate.

The functions used to calculate the change in pressure and temperature are derived from basic formulas provided by Prof. Parolini and some additional formulas the team developed.

Basic principles:

$$\Delta P = P_{OUT} - P_{IN}$$

$$\Delta T = T_{OUT} - T_{IN}$$

$$\Delta P_{SR} = \Delta P_{OUT,SR} - \Delta P_{IN,SR}$$

The different formulas for change in temperature:

$$\Delta T_1 = 4 * \frac{RPM}{Q} * \frac{T}{500}$$

$$\Delta T_2 = 5 * \frac{RPM}{Q} * e^{\frac{300 - T}{20}} + \frac{T}{700}$$

$$\Delta T_3 = 5 * \frac{RPM}{Q} * e^{\frac{300 - T}{20}}$$

The different formulas for change in pressure:

$$\Delta P_1 = (10\,000 * RPM + 2000 * (250 - Q)) * \frac{470 - T}{150}$$

$$\Delta P_2 = (1000 * RPM + 800 * (250 - Q)) * \frac{450 - T}{150}$$

$$\Delta P_3 = (10\,000 * RPM + 2000 * (250 - Q)) * \frac{450 - T}{150}$$

The two functions for change in stop ring:

$$\Delta P_{SR,ST} = (30\,000 * (200 - RPM) + 10\,000 * Q) * \frac{500 - T}{150}$$

$$\Delta T_{SR,SP} = T_{prev} + 0.01 * T$$

As seen *stop ring special* only directly affects the temperature, while the pressure always is affected by *stop ring standard*.

2.3.2 Calculation of temperature

To map the temperature profile, we need a temperature value from each numerical module. This can be achieved by setting an initial input value and using the formula for ΔT to calculate the output temperature of each numerical module:

$$\Delta T = T_{OUT} - T_{IN} \Leftrightarrow T_{OUT} = T_{IN} + \Delta T$$

Each numerical module depends on the previous numerical module; thus, it is natural to use a sequential calculation to determine the temperature.

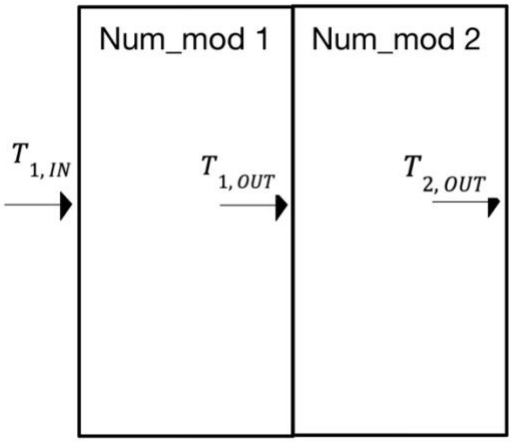


Figure 2.3: Here we have the two first numerical modules showing the temperature flow between them.

From Figure 2.3 we can see that $T_{1,OUT} = T_{2,IN}$, and this follows for all the numerical modules. From this we can index the variables, letting $T_{IN} = T_{n-1}$. Therefore, let

$$T_{OUT} = T.$$

Then,

$$T_{n,i} = T_{n-1} + \Delta T_{n,i-1},$$

where

$$T_{n,0} = T_0 = 293$$

and *n* represents numerical modules and *i* represents iterations.

2.3.3 Calculation of pressure

The calculations for the pressure profile follow the same logic as the temperature calculations, with the key difference being that the sequence iteration is performed from the end to the start, opposite to that of the temperature calculations. Given a formula for the pressure in the stop rings, which is

located at the end of each physical module, we can calculate backwards through the physical modules using the following logic:

$$\Delta P = P_{OUT} - P_{IN} \Leftrightarrow P_{IN} = P_{OUT} - \Delta P$$

Here, ΔP can always be calculated, and P_{OUT} is always provided by the input of the previous numerical module (using the same logic as for temperature, but in reverse). Therefore, let

$$P_{IN} = P.$$

Then,

$$P_n = P_{n+1} - \Delta P_n$$

where

$$P_{40} = P_{SR},$$

and n represents numerical modules 39, 38, … ,1.

2.3.4 Iterations and convergence

An important aspect of the methodology is the way we do the iterations. Firstly, we do a backward sweep, starting on the last stop ring in the last physical module. In this sweep we calculate the pressure, as described in 2.3.3. Secondly, we do a forward sweep starting in the other end of the mixer, here we calculate the temperature, as described in 2.3.2. Two such sweeps amount to one iteration. In a more complex model, we would have that the temperature and pressure functions are dependent on each other. However, for simplicity this was not implemented in our project.

We are also checking for convergence, which is obtained if the temperatures from the last iteration does not exceed temperatures of the current iterations by more than 0.0001. This is only relevant for temperature since the pressure is also only dependent on temperature. Meaning that we essentially only need to run the calculation for pressure when the temperature has converged, and we have the final values for temperature.

3. Code Organization and Architectural Choices

This chapter will focus on the code's organization and the architectural choices used to implement the simulation of temperature and pressure distribution of the mixer.

3.1 Architecture

The codebase is organized into several distinct classes that together form a comprehensive system for simulating and visualizing the extruder processes. The architecture leverages C++ for modeling the mixer and the simulation's computational aspects, while the architecture leverages Python for data visualization. This offers a robust framework for handling and displaying simulation data efficiently.

The underlying goal is to make sure the codebase design is modular, flexible and efficient. To accomplish this objective, header and source files were implemented into the codebase to enhance its manageability and scalability. This approach separates definitions in header files from implementations in source files, enhancing modularity and clarity. Modularity is essential as it allows the team to navigate and update the codebase with ease, maintaining each component separately without affecting the whole system. Encapsulation provided by header files helps secure the code by hiding implementation details and exposing only the necessary interfaces to other parts of the program. This makes the code easier to maintain and modify, as changes to the implementation do not affect parts of the program that use these interfaces.

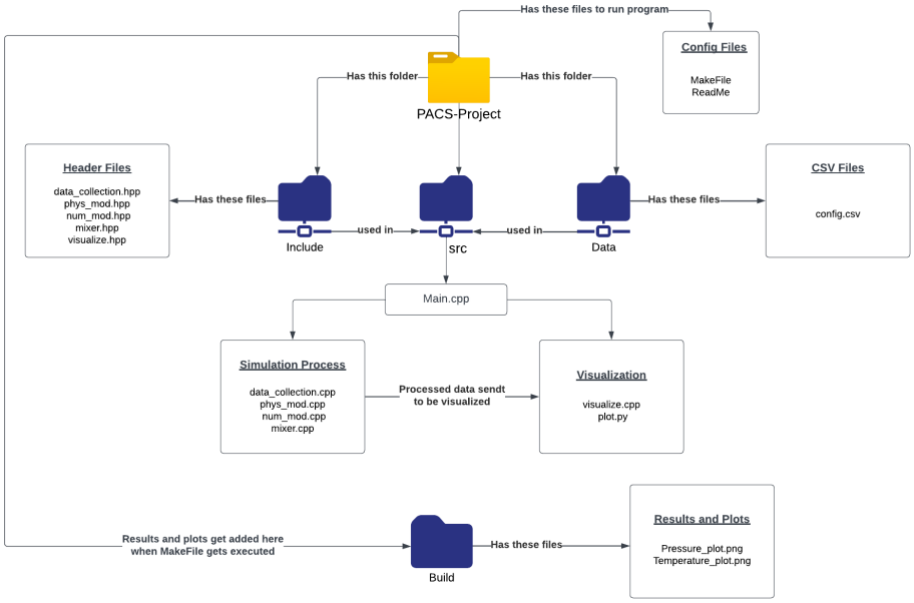


Figure 3.1: Overview of the codebase, highlighting the purpose of different folders.

3.2 Classes

The different classes work together forming a working system for simulating the mixer and displaying the results. The codebase is object-oriented, and all the classes use getters and setters for safe accessing of data. In the `main.cpp` everything is put together, and the test cases get run. The following subsections will highlight how the classes are implemented and how they form the system.

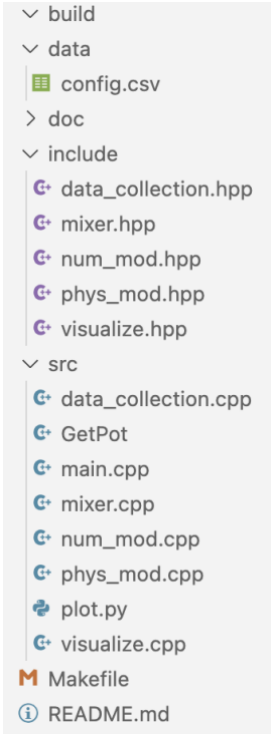


Figure 3.2: The different files and folders of the system. Header files are stored in the include folder while source files are stored in the src folder.

3.2.1 data_collection class

In `main.cpp`, the system starts by reading the configuration file and creating a `data_collection` object, handled by the `data_collection` class. This class is responsible for reading and managing configuration parameters from the `config.csv` file, which are essential for establishing the initial conditions and settings of the simulation environment. The `data_collection` object stores the data from the `config.csv` file in a structured manner that subsequent classes can easily access. The data is stored in a two-dimensional vector, so that the `main.cpp` can choose what test case (`data_line`) to run, and for the chosen `data_line` get its parameters. The `data_collection` class uses the boost library for separating strings while reading the `config.csv` file.

```
type,SRtype,npm,nnm,tIn,RPM,Q
Type1,Standard,2,40,293,100,150
Type1,Standard,2,40,293,150,250
Type1,Special,2,40,293,100,150
Type1,Special,2,40,293,150,250
Type2,Standard,2,40,293,100,150
Type2,Standard,2,40,293,150,250
Type2,Special,2,40,293,100,150
Type2,Special,2,40,293,150,250
Type3,Standard,2,40,293,100,150
Type3,Standard,2,40,293,150,250
Type3,Special,2,40,293,100,150
Type3,Special,2,40,293,150,250
```

Figure 3.3: The `config.csv` file that is being read in the `data_collection` class

3.2.2 Mixer class

The `mixer` class is responsible for the modeling and simulation process. It initializes and manages a collection of physical modules, denoted as `phys_mod`, with each physical module having multiple numerical modules. The constructor takes as input the number of physical modules (`number_pm`) and the number of how many numerical modules each physical module has (`number_nm`). The constructor updates the `std::vector<phys_mod> screw` variable, which is a vector of the physical modules in the mixer.

```
mixer::mixer(size_t number_pm, size_t number_nm)
{
    for (size_t i = 0; i < number_pm; i++)
    {
        screw.push_back(phys_mod(number_nm));
    }
}
```

Figure 3.4: The mixer constructor calls the phys_mod constructor that makes the phys_mod objects. In the phys_mod constructor the num_mod objects are created. This process creates the composition of the mixer.

The `mixer.cpp` class also has variables for “global” parameters that are used in the mixer. These are `t0` and `p0`, stored as doubles, and `RPM`, `Q`, `type` and `SRtype`, stored in the parameters map. The parameters are set in the `mixer::set_parameters()` function.

```
private:
    std::vector<phys_mod> screw;
    double t_0; // initial temperature of the fluid
    double p_0; // initial pressure
    std::map<std::string, double> parameters;
```

Figure 3.5: `mixer` parameters

The mixer class also has the important `mixer::simulate_mixer()` function where the model is simulated. The `mixer::simulate_mixer()` function is supposed to simulate the numerical methodology described in section 2.3. It utilizes the `phys_mod::update_p()` and `phys_mod::update_t()` functions to do so. The update functions will be iterated until the pressure and temperature converges, or the max number of iterations is reached. The threshold for convergence is set to 0.0001 for the temperature and 0.001 for the pressure, and the maximum number of iterations is 100.

3.2.3 Phys_mod class

The `phys_mod` class has a vector with multiple `num_mod` objects. The placement of the `num_mods` is important in the calculations, therefore the index of each `num_mod` in the vector determines its location within the mixer. The `phys_mod` also has parameters for `RPM`, `Q`, `tIn`, `type` and `SRtype`, that are used in the calculations. The `tIn` is an important variable that is used in the `phys_mod::update_t()` function. The variable is used to ensure that the temperatures are being transferred between the physical modules.

```
private:
    vector<num_mod> model;
    double RPM;
    double Q;
    double tIn;
    string type;
    string SRtype;
```

Figure 3.6: phys_mod.hpp

The phys_mod class has the functions update_t() and update_p() that are being used in the mixer:simulate_mixer() function. The update functions are used to integrate the numerical methodology described in section 2.3. The update function uses if-statements to run the right formulas specified in the config.csv, that can be *type1*, *type2* or *type3* for the type, and standard or special for the SRtype. The different types use several different functions for calculating the change in temperature and pressure, as shown in figure 3.7.

```
double dp_type1(double RPM, double Q, double T) const;
double dp_type2(double RPM, double Q, double T) const;
double dp_type3(double RPM, double Q, double T) const;
double dp_SR(double RPM, double Q, double T) const;
double dt_type1(double RPM, double Q, double T) const;
double dt_type3(double RPM, double Q, double T) const;
double dt_type2(double RPM, double Q, double T) const;
```

Figure 3.7: Functions for calculating the change in temperature and pressure

3.2.4 Num_mod class

The num_mod class is the most fundamental part of the mixer. The num_mod objects are being constructed by the phys_mod constructor. Each num_mod object has a temperature t and a pressure p, that is to be updated as the model is simulated. The num_mod also has an id, but this is only used in the mixer::print_mixer() function for printing results in the terminal.

The num_mod uses setters and getters for updating its t and p values in the phys_mod::update_t() and phys_mod::update_p() functions.

3.2.5 Visualize class

The visualization process is initiated by the visualize.cpp class. This class captures the simulation outputs from the mixer class, generating and organizing these outputs into a CSV file. The CSV file logs temperature and pressure data from various components over each num_mod, serving as the foundation for the visualization process.

```cpp
visualize::visualize(mixer m, const std::vector<std::string> &params) : parameters(params)
{
    std::vector<phys_mod> screw = m.get_screw(); // this is the mixer
    for (size_t i = 0; i < screw.size(); i++)
    {                                                    // iterates over the phys_mods
        std::vector<num_mod> model = screw[i].get_model(); // this gets the num_mods of the phys_mod
        for (size_t j = 0; j < model.size(); j++)
        {                                          // iterates over the num_mods
            t_data.push_back(model[j].get_t()); // this gets the temperature of the num_mod
            p_data.push_back(model[j].get_p()); // this gets the pressure of the num_mod
        }
    }
}
```

Figure 3.8: Code in visualize.cpp that captures the simulation outputs from the mixer class

```cpp
void visualize::vizualize_screw()
{
    std::ofstream file("build/plotting_data.csv"); // opens file

    // step 1: writes csv file
    if (file.is_open())
    {
        file << "t,p\n"; // Header for CSV

        // generate csv file with the data
        for (size_t i = 0; i < t_data.size(); i++)
        {
            file << t_data[i] << "," << p_data[i] << "\n";
        }
        file.close();
        std::cout << "CSV file has been written." << std::endl;
    }
    else
    {
        std::cout << "Unable to open file." << std::endl;
    }
```

Figure 3.9: Code in visualize.cpp that generates and organizes mixer class outputs into a CSV file.

Once the CSV file is created, it is used as input for the Python script `plot.py`. This python script reads the CSV file and generates graphical plots of the temperature and pressure profiles using matplotlib.pyplot and pandas library. These will be explained in detail in section 3.2. The Python script is designed for flexibility, allowing customization through command-line inputs to adjust aspects of the plots, such as axes labels and data granularity. This provides a modular approach for the visualization process, making it both robust and adaptable to different analysis needs.

```
42      // Execute the Python script with parameters
43      std::string command = "python src/plot.py";
44      for (const auto &param : parameters)
45      {
46          command += " " + param;
47      }
48      system(command.c_str());
49  }
```

Figure 3.10: Code in visualize.cpp that reads the organized CSV file and sends the parameters to plot.py.

In `plot.py` the CSV file gets read and the data gets plotted using Matplotlib.

3.3 Libraries

To make the simulation and calculations possible for this project, several libraries are used from both Python and C++. Standard libraries like iostream, cmath, vector, map, string etc. will not be discussed. The libraries that are being used are GetPot, Boost and Chrono for C++, and matplotlib and pandas for Python.

Note that the openMP library is also used for the numerical experiment, and will be described in section 4.3.

3.3.1 GetPot

Getpot is used for parsing command-line arguments, which makes the process of setting simulation parameters dynamically possible, increasing the flexibility of the code. Getpot is used in `main.cpp` to set the different parameters used in the simulation. This library was used a lot for testing the code, because it allowed the code to run without recompiling, saving a lot of time.

The main idea of the GetPot is to change the model variable to make the code use the different lines in the `config.csv` file. For example running `./build/main model=4` will use the 6th line in the `config.csv` file (the 6th line because of the header and that it is zero-indexed). Other parameters can also be changed, for example `./build/main model=9 RPM=133` will use the 11th line but the RPM will be set to 133.

By default, running `./build/main` or `make run`, the simulation uses the 2nd line in the `config.csv`.

```
type,SRtype,npm,nnm,tIn,RPM,Q
Type1,Standard,2,40,293,100,150
Type1,Standard,2,40,293,150,250
Type1,Special,2,40,293,100,150
Type1,Special,2,40,293,150,250
Type2,Standard,2,40,293,100,150
Type2,Standard,2,40,293,150,250
Type2,Special,2,40,293,100,150
Type2,Special,2,40,293,150,250
Type3,Standard,2,40,293,100,150
Type3,Standard,2,40,293,150,250
Type3,Special,2,40,293,100,150
Type3,Special,2,40,293,150,250
```

Figure 3.11: `config.csv` file

3.3.2 Boost

The Boost library is used for handling complex data structures and optimizing file operations. In this project it is being used for `data_collection.cpp` for reading the `config.csv` file.

3.3.3 Chrono

The Chrono library is used to measure the time taken for the simulation to run, allowing for performance benchmarks by providing high-resolution timing. It is used in `main.cpp` to measure the time taken to simulate the mixer.

```
36      auto start_time = chrono::high_resolution_clock::now(); // Start timer
37      m.simulate_mixer();
38      auto end_time = chrono::high_resolution_clock::now(); // End timer

47      cout << "Simulation took: "
48           << chrono::duration_cast<chrono::microseconds>(end_time - start_time).count()
49           << " microseconds" << endl;
```

Figure 3.12: Chrono library used in main.cpp

3.3.4 Matplotlib and pandas

On the Python side, the pandas library is used to quickly read and structure the CSV file described in section 3.1.5. Matplotlib is used to visualize the results creating a 2D plot for the

temperature and pressure distributions. Matplotlib offers extensive customisation options and the plots created can be seen in section 4.

3.4 Makefile

The Makefile is designed to manage the entire build process of this codebase. This is done by compiling and linking various C++ source files into a single executable program. It begins by defining the compiler which is denoted as g++, and the necessary compiler flags. These flags include specifying what compiler version to use and including the C++ library Boost.

Next, the Makefile sets up the build directory and specifies the target executable file, which is `main.cpp` in this project.

The Makefile has several rules implemented to make it efficient and easy to maintain. Firstly, it has a rule to make sure the build directory exists before attempting to place files in it, denoted by '`mkdir -p`'. Secondly, two simple rules for ease of use. These rules are denoted as “run” and “clean”. The “run”-rule allows the user to execute the build program directly from the command line. The “clean”-rule removes all object files and other generated files from the build directory, while preserving the directory structure itself. This helps in maintaining a clean working environment and ensuring that outdated or unnecessary files do not clutter the build directory. Below is a screenshot of this project's Makefile, illustrating the explanation above.

```
# Compiler
CXX=g++

# Compiler flags
CXXFLAGS=-Wall -std=c++20 -Iinclude -I$(mkBoostInc)

# Directory for built executables and objects
BUILDDIR=build

# Build target executable
TARGET=$(BUILDDIR)/main

# Source files
SRCS= src/main.cpp src/phys_mod.cpp src/num_mod.cpp src/mixer.cpp src/visualize.cpp src/data_collection.cpp

# Object files (stored in the build directory)
OBJS=$(SRCS:src/%.cpp=$(BUILDDIR)/%.o)

# Default rule for building the executable
all: $(TARGET)

# Ensure the build directory exists
$(BUILDDIR):
    @mkdir -p $(BUILDDIR)

# Rule for linking the final executable
$(TARGET): $(OBJS) | $(BUILDDIR)
    $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $@ $^

# Rule for compiling source files to object files
$(BUILDDIR)/%.o: src/%.cpp | $(BUILDDIR)
    $(CXX) $(CXXFLAGS) -c $< -o $@

# Run the built executable
run: $(TARGET)
    ./$(TARGET)

# Clean up by removing files in build directory, but not the directory itself
clean:
    rm -f $(BUILDDIR)/* $(BUILDDIR)/*.o

# Prevent make from doing something with files named like this
.PHONY: all clean
```

Figure 3.13: Makefile

4. Test results

This chapter focuses on the output generated from the simulations using the different test cases. There are two objectives that are wished to be addressed within this chapter. Firstly, to analyze the pressure and temperature distribution being simulated by the mixer. Secondly, to showcase how parallelization can improve the effectiveness of the code. This chapter will adress these objectives and provide a clear understanding of the findings by the methodologies used.

4.1 Discussion

The outcome of the simulations for the temperature and pressure profile distribution in the mixer varies based on the data line being read from the `config.csv` file. Different characteristic functions are used to calculate the pressure and temperature and different parameters can be set within the mixer.

The plots below illustrates the pressure and temperature profile for the 2nd line in the config.csv file (2nd line including the header). This is the test case that is being run by default by doing `./build/main` or `make run`. The formulas used for these plots are of *type1*.

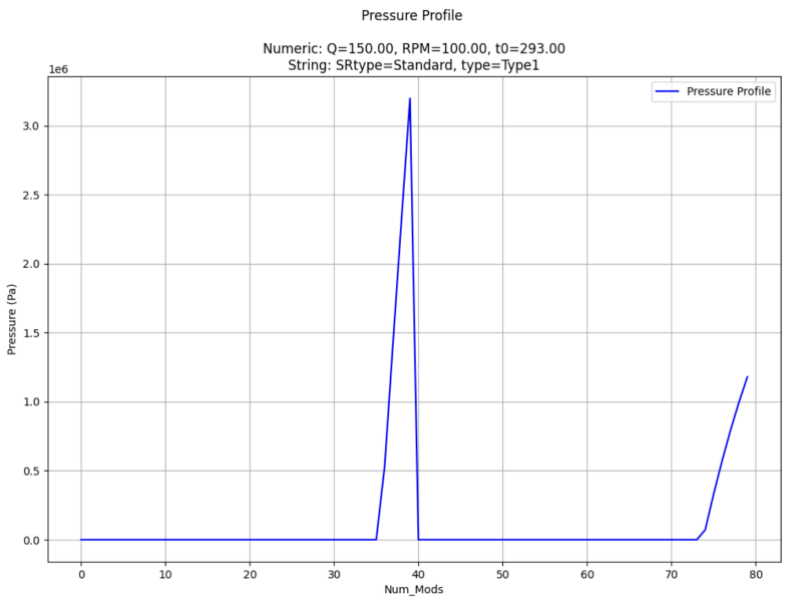


Figure 4.1: Plot of pressure profile for the default test case

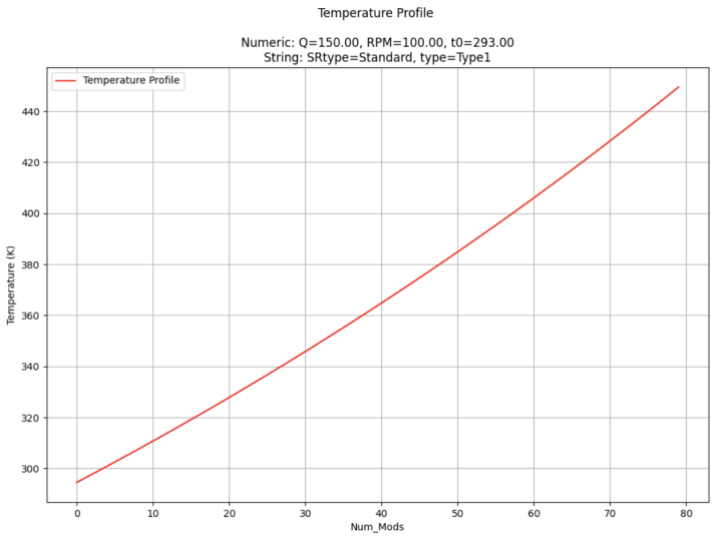


Figure 4.2: Plot of temperature profile for the default test case.

As shown in Figure 4.1, there is a notable spike in pressure from numerical module number 35 to the exit of the first physical module at numerical module number 39. This sudden increase in pressure is due to the constriction at the end of the first physical module, where the fluid is forced through a smaller opening, which increases the pressure. After the spike, the pressure drops drastically. This corresponds with the presence of a stop ring designed to reduce pressure before the fluid enters the second physical module. The decrease in pressure demonstrates the stop ring's effectiveness in controlling pressure surges within the system. The second spike in the pressure profile, although a lot smaller than the first one, is noticeable as the fluid moves through the second physical module. This moderate decrease in pressure compared to the first spike is due to an increase in temperature.

For the temperature profile shown in figure 4.2 one can see that the curve is smooth and not affected by the stop rings. This is due to the *standard* stop ring type, discussed in section 2.3.

Looking at another test case, where the stop ring type is set to *special*, it is clear that the temperature jumps at the stop ring. The *special* type stop ring is implemented to capture a steeper temperature increase in the stop rings where the cross section norrows. The next plots are simulated by running the 8th line in the config file, by doing `./build/main model=6`. This simulation will use the formulas of *type2*.

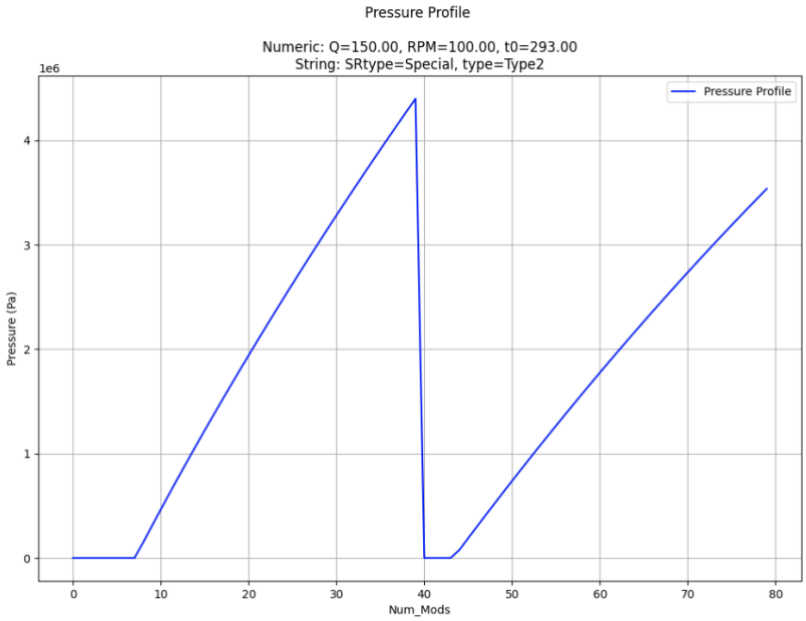
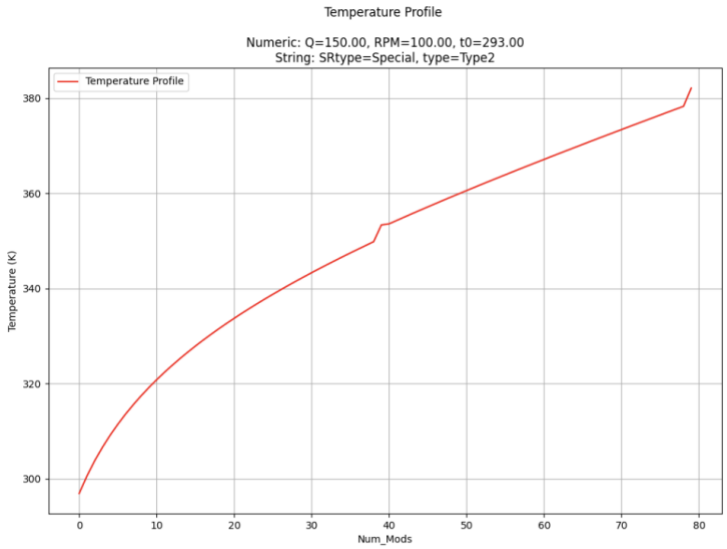


Figure 4.3: Plot for pressure using parameters from the 8th line in the config.csv file



F Figure 4.4: Plot for temperature using parameters from the 8th line in the config.csv file

Simulations using formulas of *type2* create different profiles than using the default test case, which is to be expected. As seen in section 2.3, the formula for *type2* has, in addition to the exponential function shared with *type1*, a linear term. This gives us the output seen in Figure 4.4, where we can see the function first being exponential, then linear.

4.2 Numerical experiment to highlight effectiveness of the code

The team would like to highlight how the code can be made more computationally efficient with a numerical experiment, by using parallel programming with the C++ library openMP.

For simulating the mixer the code runs the `mixer::simulate_mixer()` function, where it in every iteration updates the pressure and the temperature by calling the `phys_mod::update_p()` and `phys_mod::update_t()` functions. Our model has three different sets of formulas for `update_p()` and `update_t()`, called *type1*, *type2* and *type3*. For all of these three types the `update_t()` function is not dependent on the pressure and the `update_p()` function will converge on one iteration. Therefore, it will be possible to run these for-loops in parallel using openMP.

Figure 4.5 shows how the `mixer::simulate_mixer()` function is updated to make the updating functions run in parallel. Note that the openMP library is included with `#include <omp.h>`.

```cpp
#include <omp.h>

void mixer::simulate_mixer()
{
    // Set the initial temperature sequentially
    for (size_t i = 0; i < screw.size(); i++)
    {
        screw[i].set_tIn(parameters["t0"]);
    }

    int iter = 2000;
// Parallelize the outer loop
#pragma omp parallel for
    for (int i = 0; i < iter; i++)
    {
// Parallel region starts here
#pragma omp parallel sections
        {
#pragma omp section
            {
                // Iterate going backwards and update pressures
                for (int j = static_cast<int>(screw.size()) - 1; j >= 0; j--)
                {
                    screw[j].update_p();
                }
            }
#pragma omp section
            {
                // Update temperatures
                for (size_t j = 0; j < screw.size(); j++)
                {
                    screw[j].update_t();
                    if (j != screw.size() - 1)
                    {
                        screw[j + 1].set_tIn(screw[j].get_tOut());
                    }
                }
            }
        } // End of parallel sections
    }
}
```

Figure 4.5: Code for parallelization.

For compiling the code small adjustments need to be made to the MakeFile. At `CXXFLAGS`, `-fopenmp` needs to be added.

4.2.1 Results of Numerical Experiment to highlight Effectiveness

To get clearer results the amount of iterations per simulation is set to 2000, and the mixer is composed of 5 physical modules each containing 40 numerical modules.

```
type,SRtype,npm,nnm,tIn,RPM,Q
Type1,Standard,5,50,293,100,150
```

Figure 4.6: the data_line being executed

By running 30 simulations with and without parallelization it is clear that the runtimes are improved by parallelizing the for-loops.

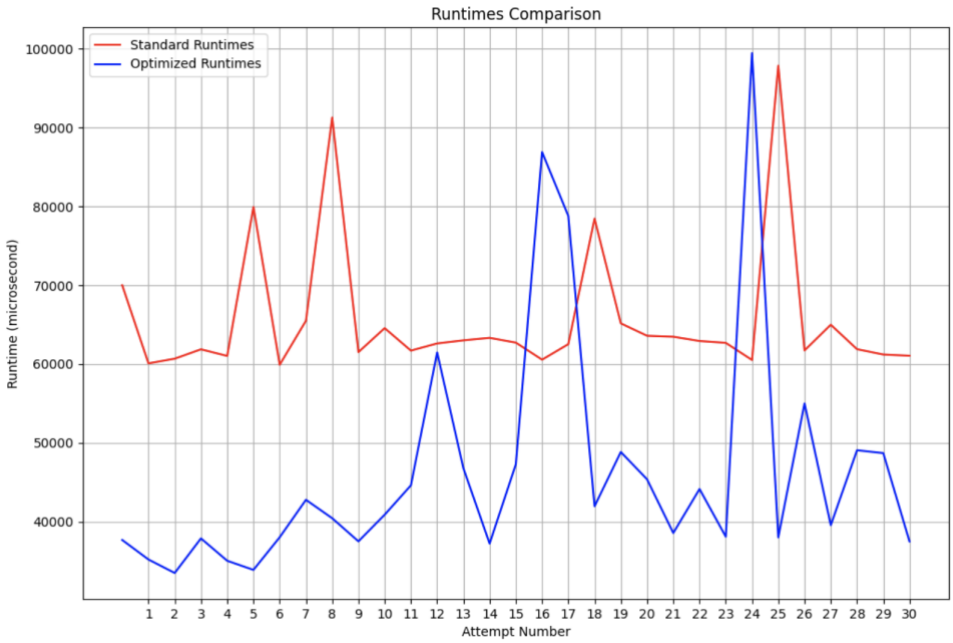


Figure 4.7: Plot for runtimes

```
Average Standard Runtime: 65752.23 microseconds
Average Optimized Runtime: 46459.97 microseconds
Optimized version is 1.42 times quicker
```

Figure 4.8: Average times showing that parallelization makes code 1.42 times quicker.

Even though parallelizing the update functions approves runtime the team has decided to not include parallel programming in the original codebase. This is due to the possibility to change the formulas for calculating pressure and temperature in the future to make them dependent on each other, better representing real life complexities.

5. Conclusion

In this project the team has developed a C++ codebase that calculates the pressure and temperature profiles from a mixer in operation. The mixer consisted of two physical modules, each consisting of 40 numerical modules, with a stop ring at each physical module. By inventing and using different characteristic functions for temperature and pressure, it was possible to calculate the temperature and pressure profiles.

Even though the numerical methodologies and models used for the simulations were simplified and idealized, the simulations managed to capture valuable insight about the mixer's temperature and pressure profiles. Allowing a clearer understanding of processes within the mixer.

In addition to this, a big focus was creating an effective and modular codebase. Therefore, the code structure was built by dividing the code into header and source files and distinct classes. The team utilized different libraries to make the code convenient and effective. The test results of the simulated mixer were visualized showing a plot of the temperature and pressure for every numerical module.

6. Tutorial for running the code

This section will give a brief tutorial on how to run the code with the different test cases.

1. For including the right C++ libraries one needs to be in the right Docker Image. An in-depth guide on how to install Docker and the right pacs-environment, in addition to building and running the environment, is given in the PACS-course Lab session 00. Mark that all the following steps need to be done within the environment. Here is a link to the tutorial on github: *https://github.com/pacs-course/pacs-Labs/tree/main/Labs/2024/00-environment_setup*
2. The project has to be downloaded and placed within the docker environment. The project can be downloaded by cloning with git from: *https://github.com/bangkorkor/PACS-project*
3. Do `module load boost` in the terminal to load the boost library.
4. Do `pip install pandas matplotlib` to install pandas and matplotlib to be used in the plotting script.
5. After completing step 4 all the necessary libraries should be installed, and one can do `make` to compile the code. Mark that this has to be done in the terminal in the lowest level of the project directory, this applies for all the remaining steps.
6. For running the code (and also compiling if not already done) do `make run`. When it comes to running the code with different test cases, it by default runs the test case with parameters from the 2nd line in the data/config.csv file.
7. For running different test cases one can specify what model to run by doing `./build/main model=7`. This will use the parameters from the 5th line of the data/config.csv file, without having to recompile the code. One can also change whatever parameter using the same technique. See Getpot in section 3.2.1 for more information.
8. The results are shown in the /build folder. If everything is done right you shall see a Pressure_plot.png and Temperature_Plot.png file in the /build folder.

7. Bibliography

[1] ScienceDirect, 2024. Screw extruder. [online] Available at: https://www.sciencedirect.com/topics/engineering/screw-extruder [Accessed 5 June 2024].

[2] Rauwendaal, C., 2014. Polymer extrusion. 5th ed. Munich: Hanser Publishers; Cincinnati: Hanser Publications. Available at: https://api.pageplace.de/preview/DT0400.9781569905395_A22613947/preview-9781569905395_A22613947.pdf [Accessed 5 June 2024].

[3] Tadmor, Z. and Gogos, C.G., 2013. *Principles of polymer processing*. Hoboken: John Wiley & Sons.

[4] Griffith, R.M., 1962. Fully developed flow in screw extruders: Theoretical and experimental study. *Industrial & Engineering Chemistry Fundamentals*, 1, pp.180–187.

Images:

[5] Replast Ltd., 2024. Features of extrusion on twin screw extruders. [online] Available at: https://www.replast-ltd.com/de/features-of-extrusion-on-twin-screw-extruders/ [Accessed 5 June 2024].