**GYMNASIUM JANA KEPLERA**

Parléřova 2/118, 169 00 Praha 6

# Fluid Flow Simulator

Maturita Project

| | |
|---|---|
| Author: | Viktor Fukala |
| Class: | R8.A |
| School year: | 2020/2021 |
| Subject: | Computer Science |
| Supervisor: | Šimon Schierreich |

Prague, 2021

GYMNASIUM JANA KEPLERA
*Kabinet informatiky*

# ZADÁNÍ MATURITNÍ PRÁCE

| | |
|---|---|
| *Student:* | **Viktor Fukala** |
| *Třída:* | **R8.A** |
| *Školní rok:* | **2020/2021** |
| *Platnost zadání:* | **30. 9. 2021** |
| *Vedoucí práce:* | **Šimon Schierreich** |
| *Název práce:* | **Simulátor proudění tekutin** |

*Pokyny pro vypracování:*

**Vytvořte nativní aplikaci pro operační systémy Linux a Windows simulující laminární proudění tekutin kolem pevných překážek v dvoudimenzionálním prostoru. Mezi její hlavní funkce bude patřit: (a) grafické rozhraní pro nastavení umístění překážek a dalších parametrů simulace a pro průběžné přehrávání výsledku simulace, (b) export výsledku do souboru jako video, (c) schopnost detekovat kolaps výpočetního modelu a simulaci v tom případě předčasně ukončit.**

*Doporučená literatura:*

**[1] JEŽEK, Jan, Blanka VÁRADIOVÁ a Josef ADAMEC. Mechanika tekutin. 3. přepr. vyd. Praha: České vysoké učení technické, 1998. ISBN 80-01-01615-3.**

**[2] MARTIN, Robert C. Design Principles and Design Patterns. 1. vyd. www.objectmentor.com, 2000. Dostupné z: `https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf`.**

**[3] FOWLER, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley Professional, 2003. ISBN 978-0321127426.**

*URL repozitáře:*

**`https://github.com/mimo31/brandy0`**

*vedoucí práce*

*student*

*V Praze dne 21. 10. 2020*

## Declaration of Authorship

I hereby declare that I created this project unaidedly and that all sources I used are listed in the attached bibliography. I consent to this project being made available in accordance with the Czech copyright law, Law No. 121/2000 (*zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon)* in Czech).

Prague, April 7, 2021 Viktor Fukala

## Acknowledgements

The creation of this project in its ultimate form would not be possible without the help, support, and advice from my supervisor, Šimon Schierreich. I would like to thank him for supervising this project and always being ready to answer my questions and clear my doubts.

## Abstract

The software developed in this project is a native application for Linux and Windows that simulates and visualizes the flow of an incompressible fluid around arbitrary solid obstacles and under various conditions in two-dimensional space. It aims to deepen the basic understanding of fluid flow among its end users by presenting physically accurate visualizations without the sophisticated interface that often comes with fluid simulation software used in the industry. It simulates the flow based on the shapes of the obstacles and the boundary conditions for pressure and velocity, all of which can be configured by the end user before starting the simulation. The software works the fastest and most reliably for low Reynolds numbers (laminar flow), whereas for high Reynolds numbers (turbulent flow), the amount of required computational power (both in time and space) increases rapidly. Nevertheless, the software achieves its goal by being able to simulate a wide range of laminar flows and phenomena (such as the development of vortices) at the transition to turbulent flows.

### Keywords

computational fluid dynamics, incompressible Navier-Stokes equations, data visualization, computer simulation

## Abstrakt

Software vyvinutý v rámci této práce je nativní aplikací pro operační systémy Linux a Windows, která simuluje a vizualizuje proudění nestlačitelné tekutiny kolem libovolných pevných překážek ve dvourozměrném prostoru za různých podmínek. Jeho cílem je prohloubit základní povědomí o proudění tekutin u koncových uživatelů pomocí fyzikálně přesných vizualizací bez nutnosti ovládat složitá rozhraní, jež bývají součástí průmyslového software pro simulaci tekutin. Proudění tekutiny se simuluje na základě tvarů překážek a okrajových podmínek pro tlak a pro rychlost, což koncový uživatel zadává před spuštěním simulace. Nejrychleji a nejspolehlivěji náš software pracuje při nízkých Reynoldsových číslech (laminární proudění), zatímco při vysokých Reynoldsových číslech (turbulentní proudění) začíná potřebný výpočetní výkon (v čase i prostoru) prudce narůstat. I přesto náš software dosahuje svého cíle tím, že je schopen simulovat širokou škálu laminárních proudění a jevů při přechodu k turbulentnímu proudění (např. vznik vírů).

### Klíčová slova

výpočetní dynamika tekutin, Navier-Stokesovy rovnice pro nestalčitelné tekutiny, vizualizace dat, počítačové simulace

# Contents

# 1. Theoretical Background

## 1.1 Introduction

Computational fluid dynamics (CFD) is the application of numerical methods to fluid flow problems. As of today, it plays a vital role in engineering areas and the related industries such as aeronautics, the automotive industry, environmental engineering, bioengineering (simulating the flow of blood or the air flow in lungs, for example), or even video games, where it is used to produce realistic video material for fluids. Computer-simulated fluid flow has replaced many expensive or technically infeasible real-world experiments, and its importance is only expected to grow with future advancements in both computing power and the efficiency of the relevant algorithms.

In this project however, we are not focused on any of these direct applications but rather on providing the end user with the fundamental understanding of fluid flow phenomena and a high degree of freedom in experimenting – i.e. setting up simulations to their liking. We aim for software that visualizes the computed results in an attractive and immediate manner. Therefore, our goals differ greatly from those of the software used in the industry, as that often prioritizes precision and efficiency and comes with an elaborate user interface and special functionality for the one or the other engineering or industry application.

My main motivation for committing myself to work on this project was my fascination with the computers' ability to simulate (parts of) the real world. My curiosity as to how far one can get with only the general knowledge of physics and the fundamentals of numerical analysis and no prior expertise in CFD specifically also played an important role.

Our software computes and visualizes the temporal evolution of the state of a fluid inside a two-dimensional, rectangular container. The state of the fluid is given by the velocity vector field and the pressure scalar field, each of which has a value at every point inside the container. This state at some given time after the start of the simulation depends on a multitude of parameters:

- **the shape of the container,**
  This includes the dimensions of the enclosing rectangular container and also the shapes and dimensions of the solid obstacles inside it. Both can be freely configured by the end user before a simulation.
- **the mechanical properties of the fluid,**
  This includes the density and the viscosity of the fluid, both of which can be freely configured by the end user.
- **the boundary conditions at the boundary of the container, and**
  At the four sides of the enclosing container, the user can specify what boundary conditions will be enforced. For both velocity and pressure, they can choose between a Dirichlet type or a Neumann type boundary condition. For the Dirichlet type boundary condition, they can also specify the value (of pressure or velocity) at the boundary. Under a Neumann type boundary condition, we specifically understand the Neumann boundary condition which requires the derivative along the normal to the surface (here, the side of the container) to be zero.

  At the inner boundaries, i.e. at the surface of the solid obstacles inside the container, the no-slip condition is enforced for velocity (meaning that a Dirichlet type boundary condition setting the velocity to the zero vector holds) and a Neumann type boundary condition is set for pressure.

- **the initial conditions.**
  I.e. the state of the fluid at $t = 0$. The state of the fluid at $t = 0$ is set to zero velocity and zero pressure everywhere (except at the boundary).

## 1.2 Physical Model

To model the behavior of the fluid, we use the Navier-Stokes equations. Let $\mathbf{u}$ denote the velocity of the fluid and $p$ its pressure at all the points inside the container. We assume that the fluid is incompressible, so we get the incompressibility equation

$$\nabla \cdot \mathbf{u} = 0. \tag{1.1}$$

Then we have the Navier-Stokes momentum equation

$$\rho \frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = -\nabla p + \mu \nabla^2 \mathbf{u} + \rho \mathbf{g},$$

where $\rho$ is the density of the fluid, $\mu$ its (dynamic) viscosity, $\mathbf{g}$ is the acceleration due to external forces (such as gravity) and $\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t}$ denotes the material (also known as *convective*) derivative of the velocity.

$$\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u}$$

We assume no external forces, so $\mathbf{g} = 0$. Hence, the momentum equation that we work with takes the form

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u}, \tag{1.2}$$

where $\nu = \mu/\rho$ is the *kinematic* viscosity.

## 1.3 Numerical Model

Our way of numerically solving the above equations is not the most accurate or computationally efficient, but it is relatively simple to implement and to work with. It is essentially the Chorin's projection method introduced in [4]. What we have implemented in an explicit, finite difference method.

During the implementation, Lorena A. Barba's practical examples of solving the Navier-Stokes equations (see [3]) as well the recordings of her CFD course at Boston University (see [2]) were of great help. The solver used in [3] is also an implementation of Chorin's projection method, and is therefore similar to our implementation, but there are a few notable differences such as the fact that the solver in [3] neglects some of the terms on the right-hand side of the pressure Poisson equation[1], does not incorporate any obstacles inside the enclosing rectangular boundary, and does not use the general upwind differencing scheme[2] as we do.

In order to computationally represent, process, and manipulate the state of the fluid, we must discretize in both space and time. As for the temporal discretization, we only compute the state of the fluid

[1]See 1.3.1.
[2]See 1.3.2.

at times that are (non-negative) integer multiples of some small time step $\Delta t$. These are computed successively with a state at any particular time depending on the state directly before. Each of the states must also be discretized spatially. For that, we use a uniform Cartesian grid with a spacing of $\Delta x$ and $\Delta y$ in the $x$ and the $y$ direction respectively and we store values of the fluid state fields only at these grid points.

### 1.3.1 Temporal Discretization

What now follows is a brief description of Chorin's projection method that we use in the core of our simulator.

Let us denote the value of a particular quantity at the $i$-th iteration by $i$ in the superscript. That is, let $\mathbf{u}^i$ be the velocity field at the $i$-th iteration, $p^i$ the pressure field at the $i$-th iteration, $\left(\frac{\partial \mathbf{u}}{\partial t}\right)^i$ the time derivative of the velocity field at the $i$-th iteration. Then we approximate the derivative of $\mathbf{u}$ by the forward difference.

$$\left(\frac{\partial \mathbf{u}}{\partial t}\right)^i = \frac{\mathbf{u}^{i+1} - \mathbf{u}^i}{\Delta t}$$

So at the $i$-th iteration, (1.2) takes the form

$$\frac{\mathbf{u}^{i+1} - \mathbf{u}^i}{\Delta t} + (\mathbf{u}^i \cdot \nabla)\mathbf{u}^i = -\frac{1}{\rho}\nabla p^i + \nu \nabla^2 \mathbf{u}^i. \tag{1.3}$$

Together with the equation $\nabla \cdot \mathbf{u}^{i+1} = 0$ (that is (1.1) at the $(i+1)$-th iteration), this allows us to solve for $\mathbf{u}^{i+1}$ and $p^i$. From (1.3), we express

$$\mathbf{u}^{i+1} = -\frac{\Delta t}{\rho}\nabla p^i + \Delta t\left(\nu \nabla^2 \mathbf{u}^i - (\mathbf{u}^i \cdot \nabla)\mathbf{u}^i\right) + \mathbf{u}^i.$$

During the computation, we have yet to solve for $p^i$, so we only calculate the following intermediate vector field $\mathbf{w}^i$.

$$\mathbf{w}^i := \Delta t\left(\nu \nabla^2 \mathbf{u}^i - (\mathbf{u}^i \cdot \nabla)\mathbf{u}^i\right) + \mathbf{u}^i$$

The above formula gives the value of the field in the interior of the grid and we use the boundary conditions for velocity to calculate its values at the boundary. Then

$$-\frac{\Delta t}{\rho}\nabla p^i + \mathbf{w}^i = \mathbf{u}^{i+1}$$

and

$$-\frac{\Delta t}{\rho}\nabla^2 p^i + \nabla \cdot \mathbf{w}^i = \nabla \cdot \mathbf{u}^{i+1} = 0.$$

Hence, we have the following Poisson equation for $p^i$

$$\nabla^2 p^i = \frac{\rho}{\Delta t}\nabla \cdot \mathbf{w}^i.$$

We solve this Poisson equation[3] for $p^i$ in the interior of our grid and apply the boundary conditions for pressure to obtain $p^i$ at the boundary.

[3]There exist several well-known methods for solving a Poisson equation numerically. They include, most notably, the Jacobi method, the Gauss-Seidel method, successive over-relaxation, multigrid methods, and the conjugate gradient method. Multigrid methods and the conjugate gradient method generally converge faster, but are noticeably more complex and difficult to implement than the other approaches mentioned. Of those, we use successive over-relaxation by default as it tends to have the fastest convergence. See section 9.3.3 of [8] for more details about the methods.

At last, we compute $\mathbf{u}^{i+1}$ given by

$$\mathbf{u}^{i+1} = -\frac{\Delta t}{\rho}\nabla p^i + \mathbf{w}^i$$

and the boundary conditions for velocity.

### 1.3.2 Spatial Discretization

It remains to clarify how we compute the space derivatives of the various fields. As mentioned above, we use the method of finite differences. In particular, for the Laplacian, we use the five-point stencil

$$\begin{aligned}
(\nabla^2\mathbf{u}^i)_{j,k} &= \frac{\mathbf{u}^i_{j+1,k} - 2\mathbf{u}^i_{j,k} + \mathbf{u}^i_{j-1,k}}{(\Delta x)^2} + \frac{\mathbf{u}^i_{j,k+1} - 2\mathbf{u}^i_{j,k} + \mathbf{u}^i_{j,k-1}}{(\Delta y)^2} \\
(\nabla^2 p^i)_{j,k} &= \frac{p^i_{j+1,k} - 2p^i_{j,k} + p^i_{j-1,k}}{(\Delta x)^2} + \frac{p^i_{j,k+1} - 2p^i_{j,k} + p^i_{j,k-1}}{(\Delta y)^2},
\end{aligned}$$

where $_{j,k}$ in the subscript denotes the value of the field at the $j$-th grid point in the $x$ direction and the $k$-th in the $y$ direction. For gradient and divergence, we use the central difference

$$\begin{aligned}
(\nabla\cdot\mathbf{w}^i)_{j,k} &= \frac{(\mathbf{w}^i_x)_{j+1,k} - (\mathbf{w}^i_x)_{j-1,k}}{2\Delta x} + \frac{(\mathbf{w}^i_y)_{j,k+1} - (\mathbf{w}^i_y)_{j,k-1}}{2\Delta y} \\
((\nabla p)_x)_{j,k} &= \frac{p_{j+1,k} - p_{j-1,k}}{2\Delta x} \\
((\nabla p)_y)_{j,k} &= \frac{p_{j,k+1} - p_{j,k-1}}{2\Delta y},
\end{aligned}$$

where $_x$ (or $_y$) in the subscript refers to the $x$ (or the $y$) component of a vector field.

The discretization of the advection term $((\mathbf{u}^i\cdot\nabla)\mathbf{u}^i)$ is slightly more involved. We use upwind differencing[4] to ensure that the information about the fluid velocity is properly transmitted in the direction of the flow. That means that we take

$$(\mathbf{u}^i_x)_{j,k}\left(\frac{\partial\mathbf{u}^i}{\partial x}\right)_{j,k} = (\mathbf{u}^i_x)_{j,k}\cdot\frac{1}{\Delta x}\cdot\begin{cases}\mathbf{u}^i_{j,k} - \mathbf{u}^i_{j-1,k} & \text{for } (\mathbf{u}^i_x)_{j,k} > 0 \\ \mathbf{u}^i_{j+1,k} - \mathbf{u}^i_{j,k} & \text{for } (\mathbf{u}^i_x)_{j,k} < 0\end{cases} \tag{1.4}$$

and

$$(\mathbf{u}^i_y)_{j,k}\left(\frac{\partial\mathbf{u}^i}{\partial y}\right)_{j,k} = (\mathbf{u}^i_y)_{j,k}\cdot\frac{1}{\Delta y}\cdot\begin{cases}\mathbf{u}^i_{j,k} - \mathbf{u}^i_{j,k-1} & \text{for } (\mathbf{u}^i_y)_{j,k} > 0 \\ \mathbf{u}^i_{j,k+1} - \mathbf{u}^i_{j,k} & \text{for } (\mathbf{u}^i_y)_{j,k} < 0\end{cases}. \tag{1.5}$$

By definition, the advection term is the sum of the terms in (1.4) and (1.5).

$$((\mathbf{u}^i\cdot\nabla)\mathbf{u}^i)_{j,k} = (\mathbf{u}^i_x)_{j,k}\left(\frac{\partial\mathbf{u}^i}{\partial x}\right)_{j,k} + (\mathbf{u}^i_y)_{j,k}\left(\frac{\partial\mathbf{u}^i}{\partial y}\right)_{j,k}$$

[4]For a more detailed discussion of upwind differencing and some more advanced (and accurate) methods with the same purpose (albeit in a more general setting), see chapter 3 of [15] or only section 3.2 for upwind differencing specifically.

# 2. Implementation

## 2.1 Used Technologies

### 2.1.1 Base Programming Language: C++

The software is written mostly in C++[1]. C++ is a compiled programming language with multiple freely available compilers, which are capable of many optimizations, so that the resulting programs are one of the fastest when compared to other programming languages. This is important for us because computational resources (and their efficient use) are a limiting factor during the simulations. Moreover, C++ code can be written in a highly structured manner (e.g. using objects and classes, namespaces, templates, `constexpr`, `const`, static assertions, etc.), which decreases the likelihood of errors altogether or at least enables better static analysis and the discovery of potential errors at compile time (an improvement when compared to the C programming language, for example).

### 2.1.2 User Interface, Widget Toolkit

The most visible technology used in the software is gtkmm 3 [2], a C++ wrapper around GTK+ 3 [3] user interface widget toolkit library. It is one of the two most widely used user interface libraries for C++, the other being Qt[4]. Qt is written natively in C++ (in contrast to GTK being a C library), but the gtkmm wrapper for C++ makes good use of C++ paradigms, so that gtkmm interoperates with C++ code just as easily. Hence, both gtkmm and Qt seemed adequate for our software and the final decision was based on my personal preference for the GNOME ecosystem.

### 2.1.3 OpenGL for Visualizations

Specifically for the visualization of the simulated fluid flow, we use the OpenGL API[5] to easily produce detailed images accurately representing the computed states and to do so fast enough to be able to display the images as a continuous video. No complex graphics was necessary for our software, so only 4 relatively simple shaders written in the OpenGL Shading Language are a part of the source code.

### 2.1.4 FFmpeg for Writing Video Files

Since the software was required to be able to produce video files with the visualized flow of the fluid, we used the FFmpeg libraries[6] for video manipulation. See section 3.1.1 for the exact information

[1]See [7] for a good C++ reference.

[2]The official tutorial can be found in [1] and the official reference manual in [11].

[3]The official reference manual including some examples and an explanation of the GTK+ general principles is in [10]. When working with gtkmm, it is rarely useful – only in the exceptional cases when the gtkmm reference is not comprehensive enough.

[4]See [6].

[5]Reference in [18].

[6]Documentation is to be found in [19].

about the FFmpeg libraries used. Even though these are C libraries and so their API violates the principles of good design in C++, we use them because they are one of the most widely used tools for this kind of tasks and we are able to keep the interaction with their C API local within our codebase.

### 2.1.5 CMake for Build Automation

Compiling a C++ project can quickly become a demanding task as the complexity of the project increases. It is due to long compile times and also the necessary configuration that has to be done by whoever is trying to compile the project. CMake[7] saves time in both of these by not rebuilding the entire project on each build and by allowing its user to define build targets and organize the project configuration into flexible CMake configuration files.

## 2.2 Course of Development

At the beginning (September 2020 – November 2020), we searched for and tried multiple physical (e.g. incompressible or compressible fluid) and numerical models (various computation schemes) and we researched other software that already solves problems similar to what we aimed to do. There certainly exist other similar programs such as Flowsquare[8] or OpenFOAM[9]. Both of these generally allow for more sophisticated simulations, but they also typically require configuration outside of the program's own interface (e.g. preparation of bitmap files with solid obstacles (Flowsquare) or writing the complete simulation setup into a configuration file (OpenFOAM)). Our software tries to be more accessible for less technically skilled users.

For the sake of simplicity, we started to focus only on incompressible flows. Besides Wikipedia articles such as [23], [22], [21], and [24], the most useful source of practical information were Lorena A. Barba's materials on CFD (refer to [3]) and the recordings of her CFD course at Boston University (see [2]). Throughout these initial stages of the development, we needed to perform many experiments in order to determine how reliably our numerical models work. A sizeable portion of the models were unstable and resulted in the computation diverging. Generally, we can say that none of our models (including the final one) could efficiently simulate turbulence (flows with a high Reynolds number).

In December, the development continued with the implementation of the main parts of the simulation configuration interface and the OpenGL visualization. The interface for the configuration of solid obstacles was implemented mainly in January and after that, in February, controls for the video playback and the option to export video were added. Throughout March, parts of the numerical model were improved (including its efficiency), more options for the configuration of boundary conditions were made possible and various minor parts of the software were improved, expanded, or fixed.

## 2.3 Elegant and Efficient Solutions during Development

This section discusses a few parts of the project that could be regarded as problems elegantly solved.

[7]Documentation in [5].

[8]See [13].

[9]See [12].

### 2.3.1 Obstacle Configuration Interface

One could argue that the obstacle configuration interface is, in contrast to other components of the software, elegant and user-friendly. It displays to the user the obstacle they are about to add and the grid points that will become solid as a result of that, it allows them to sequentially undo and redo the addition of obstacles or to cancel the addition of a new obstacle. However, this might be, in part, caused by the relative its isolation and clearly defined boundaries of what this interface is supposed to do in the first place, given that the rest of the software is rather volatile (in the sense that simulations may diverge, take very long to compute, etc.).

Nevertheless, there is room for improvement even in this interface. For example, the performance (and thus the smoothness of the UI) worsens as the number of points grows large enough. Furthermore, the editor lacks any ability to copy or move already added obstacles.

### 2.3.2 Use of C++ Paradigms

It is a highly debatable claim to say that in general, the structure of the codebase is well designed. But let me draw the attention to at least a few specific parts of the code that one may regard as designed well.

Firstly, notice the structs defined in the `lib` directory – namely `Grid`, `vec2d`, `Point`. The `Grid` struct is a generic two-dimensional array of fixed size and is used to represent the values of a quantity at the grid points inside the simulated container. The `vec2d` struct is a representation of a two-dimensional vector (with two coordinates of type `double`) and `Point` of a point with two integer coordinates. All these structs make heavy use of operator overloading, so that they can be used more easily and naturally. That includes a natural way of addressing the data in a `Grid`, so that statements like

```
g(3, 4) = g(0, 1)
```

or even

```
g(p1) = g(p0),
```

where `g` is of type `Grid<T>` for any type `T` that is copy-assignable and `p0`, `p1` are of type `Point`, can be used (provided that the specified indices or points are within the bounds of the grid). Naturally, standard vector operation are also available through overloaded operators, so

```
p = (u.dot(a) / u.dot(u)) * u
d = (a + b) / 2 + c * 1.5
```

are valid statements for `a`, `b`, `c`, `d`, `p`, `u` of type `vec2d` unless `u` is the zero vector.

On a similar note, we implemented a generic class `Hideable` (in `hideable.hpp`), which allows us to hide a GTK widget while it still takes up space in its parent container. The elegance of this generic class is that `Hideable<T>` inherits from `T`, so that a widget of type `W` can be declared with type `Hideable<W>` and all its methods and attributes are still accessible normally while it now also has the new methods that hide and show (unhide) it. This construct might resemble the decorator pattern, but in a decorator pattern we would typically have to override all methods of some widget subclass to invoke the respective methods on a widget object that is stored as an attribute. And since this is a templated class, we do not even know all the methods of the `T` type, this would not be possible with a decorator pattern.

However, other parts of the codebase are not so well designed. For example, the classes representing the program's windows and the classes representing the various stages of configuring / running a simulation try to follow a general pattern, but mainly due to the large number of different states the user interface can be in, they contain a relatively high amount of loosely structured code.

### 2.3.3 Storage of Computed Frames

When computing the frames of a simulation, our program does not know how much time (how many frames) it will run for. The time for which the simulator runs is under the full control of the user, as they can pause or resume the computation at any time. Therefore, the program needs a flexible container to store the computed frames.

The program uses a specific approach to address this problem. An approach which is elegantly simple and scalable with the number of frames, but it definitely has drawbacks too. One of the parameters of a simulation is the maximum number of frames that can be stored at once. When the simulation starts, the newly computed frames are being stored until the set capacity is reached. When the capacity is reached, every other of the already stored frames gets deleted and after this moment, only every other frame is stored. This coarsening of the temporal resolution by a factor of two is repeated every time the full capacity is reached.

This strategy prevents the consumption of too much memory and it ensures that frames representing the entire duration of the computed simulation are still available at some reasonable temporal resolution. It is also probably one of the simplest ways to achieve this. Besides the overall capacity, it does not require any configuration from the user.

Nevertheless, this approach might be inadequate in some cases. It not unusual that at the beginning of a simulation, there is a short time of drastic changes in the state of the fluid. Then, if the simulator runs for long enough, the temporal resolution might degrade so much that only a few frames, if any, show what happened during that short initial period of time. It could therefore be of value to allow for a variable temporal resolution during the simulation (typically high at the beginning, lower as the simulation progresses). Such an approach would, however, be unlikely to reach its full potential unless the user were able to configure these changes in resolution.

## 2.4 Technical Complications during Development

### 2.4.1 Segmentation Fault due to `GtkSwitch`

Our software contained a bug which sometimes lead to a segmentation fault during the closing of the program. Since this did not interfere with the software's functionality in any way, fixing the crash was not a priority until later in the development process. It then turned out that the crash occurs inside the destructor of gtkmm's `Gtk::Switch`, which internally calls some methods of the GTK's `GtkSwitch` widget.

At that time, there was already a report[10] of an issue in gtkmm that demonstrated the crash on a much simpler example. After some debugging of the gtkmm and GTK libraries, I concluded that there

[10] See [16].

was a bug in the underlying GTK library, which leads to a null pointer being dereferenced whenever the destruction method is called on a `GtkSwitch` that is still inside of some window. This has been reported[11] in the GTK issue tracker on March 5, 2021 and has not been fixed as of today. Note that this bug only affects GTK+ 3 and not the newer but incompatible GTK 4.

In our software, the crash can be avoided by removing the `Gtk::Switch` from the window before the `Switch` gets destroyed, which is exactly what the software now does.

### 2.4.2 Window with Visualizations Flickering

While developing the software, I noticed that the window that displays the visualizations of the simulated states sometimes starts to flicker. This typically happens when the user interacts with some of the widgets inside this window and some kind of the flickering is almost always visible whenever the user drags a corner of the window to resize it. The intensity of the flickering varies depending on platform the software currently runs on.

This seems to be caused by an already reported[12] bug in GTK+ 3 (again, this bug does not affect GTK 4) and it is only manifested in the window with the visualizations because that is the only window which uses the affected `GtkGLArea` widget. I was not able to implement any workaround for this bug.

The flickering certainly constitutes a considerable inconvenience, but it is almost always possible to get rid of if by some sequence of user interactions with the widgets. It often helps to maximize the window to fullscreen or to click or hover over some of the widgets.

### 2.4.3 Oversimplification of the Model

The way we model the state of the fluid and its evolution can be viewed as oversimplified in at least two ways. One could think of the physical model as oversimplified, since it does not incorporate quantities and phenomena that manifest themselves in fluids we encounter every day (e.g. changes in temperature, density, or viscosity between various regions of the same fluid). Even though this is a valid objection to our model, let us rather focus on the (over)simplification of the numerical model, since this project is primarily a computer science project.

Our numerical model – the algorithm we use to solve the incompressible Navier-Stokes equations – is arguably one of the simplest algorithms capable of this. Modern solvers use more complicated and involved algorithms, which have the advantage of being more accurate and precise, more computationally efficient, or both. Most notably perhaps, one could achieve higher efficiency by using irregular grids with higher density near the edges of solid obstacles because that is where the quantities describing the fluid change most rapidly. We could do that if we used the finite element method[13]. Although, we should keep in mind that implementing the current model correctly was already by no means trivial, so there would be a high risk of failure or serious problems should we have chosen a more complex approach.

[11]See [9].
[12]See [17].
[13]Essential information about the finite element method can be found in [20].

# 3. User Guide

## 3.1 Installation

The application can be distributed as a single executable file which dynamically links to libraries it depends on. The executable file can be compiled from source available in the project repository. Regardless of how the executable file is acquired, dependencies from the list below must be installed whenever the program runs.

The following installation instructions are platform independent, but they are accompanied by example commands for Ubuntu. The software has also been successfully compiled, run, and tested on Windows, where the installation of the dependencies may, however, be slightly more difficult.[1] In general, our software can be run on any platform on which all dependencies can be installed and for which there exists a C++17 compiler.

### 3.1.1 Dependencies Required at Run Time

For every dependency listed below, automatically include all its dependencies. (If using a package manager, they typically install automatically.)

- gtkmm 3 (a C++ interface for GTK (GTK+) – a GUI library)
  - possible installation using `apt` on Ubuntu 20.10 (similarly for other linux package managers)

    ```
    $ sudo apt install libgtkmm-3.0-1v5
    ```

  - Other versions, such as gtkmm 2 or gtkmm 4, are not compatible.
- some FFmpeg libraries
  - specifically `libavcodec`, `libavformat`, `libavutil`, `libswscale`
  - possible installation using `apt` on Ubuntu 20.10

    ```
    $ sudo add-apt-repository universe
    ```

    ```
    $ sudo apt install libavcodec58 libavformat58 libavutil56 libswscale5
    ```

### 3.1.2 Additional Dependencies Required for Compilation

- developer packages for the above run-time dependencies (unless the developer files were already included in the main package)
  - possible installation using `apt` on Ubuntu 20.10

    ```
    $ sudo apt install libgtkmm-3.0-dev libavcodec-dev libavformat-dev \
    libavutil-dev libswscale-dev
    ```

- a C++17-compliant C++ compiler

[1] I used the MSYS2 environment (see [14]) to install the dependencies and compile the program on Windows 10.

  - for example: possible installation of the GCC compiler using `apt` on Ubuntu 20.10

    ```
    $ sudo apt install g++
    ```

- CMake version 3.16 or higher (a build automation tool)
  - possible installation using `snap` on Ubuntu 20.10 (recommended)

    ```
    $ sudo snap install cmake
    ```

  - possible installation using `apt` on Ubuntu 20.10

    ```
    $ sudo apt install cmake
    ```

  - Older versions of CMake may work fine too, but then it is necessary to edit the minimum required version in `CMakeLists.txt`.
- make (a build automation tool)
  - possible installation using `apt` on Ubuntu 20.10

    ```
    $ sudo apt install make
    ```

- pkg-config (a build tool for fetching appropriate compiler flags)
  - possible installation using `apt` on Ubuntu 20.10

    ```
    $ sudo apt install pkg-config
    ```

### 3.1.3 Installation Steps

If you already have the program's executable file:

1. Install all dependencies mentioned in 3.1.1.
2. Run the executable.

If you need to compile the source:

1. Install all dependencies mentioned in 3.1.1 and 3.1.2.
2. Clone the repository. In what follows, it is assumed that `$REPO` is the root directory of the repository.
3. Decide whether you want a *debug* or a *release* build (or both, one executable for each). A release build should produce a more optimized executable, but it will probably make the compilation take longer and limit potential debugging options.
4. Create a directory for the build. In what follows, `$REPO/build/release` will be the build directory.

   ```
   $ mkdir -p $REPO/build/release
   ```

5. Let CMake generate its build environment.

   ```
   $ cd $REPO/build/release
   ```

   ```
   $ cmake -DCMAKE_BUILD_TYPE=Release $REPO/src
   ```

   (replace `Release` with `Debug` if you want a debug build)
6. Let CMake build the project. The project defines two CMake targets: `brandy0` (main application) and `brandy0-test` (tests). `brandy0` builds the project normally, whereas building `brandy0-test` produces an executable that only runs some tests and not the main application.

```
$ cmake --build .  --target brandy0
```

(run this still inside `$REPO/build/release`)

7. Run the executable. It has been generated at `$REPO/build/release/app/brandy0`.

## 3.2 Navigation and Use

When running, the application can always be thought of as being in of three states: the start state, the configuration state, or the simulation state. Each of these is described in more detail in the following sections.

### 3.2.1 The Start State

The start state is the simplest of all of the application's states. It is the active state at the start of the application. It consists of a button to continue to the configuration state and a button to show a window with information about the project.

### 3.2.2 The Configuration State

The configuration state can be reached from the start state by starting a new simulation or from the simulation state by ending the running simulation. Its main purpose is to let the user configure the parameters of a simulation.

The configuration state primarily consists of two windows. There is the main configuration window and the shape configuration window. Moreover, there is a button in the main configuration window which can open a third window – the window for selecting presets.

In the main configuration window, the user sets various parameters of the simulation, which are

- the properties of the fluid (density and viscosity),
- the boundary conditions (independently for each of the four sides of the container),
- and the computational parameters of the simulator.

For each side of the container, the boundary condition for pressure and the boundary condition for velocity must be set. Each of these two can either be a Neumann type boundary condition (then the derivative of the field along the normal to the side will be zero) or a Dirichlet type boundary condition (then the value of the field at the given side must be specified[2]). Note that all the quantities are taken to be unitless, which yields the same results as if all the quantities were, for example, in SI base units.

Computational parameters of the simulator include the resolution of the discretizing grid (number of its point along the $x$ axis (width) and along the $y$ axis (height)), the time step in one iteration of the simulator ($\Delta t$), number of iterations (steps) of the simulator between two consecutive outputted frames, and the maximum number of computed frames that can be stored at once.

[2]Note that when entering the coordinates for velocity vectors, the positive $x$ direction is always to the right and the positive $y$ direction is always towards the top.

In the shape configuration window, the shape of the container and the obstacles inside it can be set. There are entries for specifying the physical width and height of the container. Obstacles can be added by selecting one of the shape types and then clicking inside the chequered area below. The chequered area is a one-to-one image of the simulated container.

The third window in the configuration state is the window for selecting presets, which can be opened from the main configuration window. It offers a list of presets – predefined values of all simulation parameters – and sets the values of the parameters in the other two windows to those of the preset once a preset is confirmed. Trying some of the presets is the recommended way of getting to know the capabilities of the simulator.

### 3.2.3 The Simulation State

There is one primary window in the simulation state – the simulation window. It is divided into the visualization display area and the control panel. The display area visualizes computed frames of the simulation based on what is set in the control panel.

In the control panel, the computation and visualization of the simulation can be controlled. From left to right, the control panel contains

- a button for closing the current simulation and returning to the configuration state,
- controls and information related to the state of the computation, i.e., a switch for pausing and resuming the computation of the simulation, information of how many computed frames are currently stored, and how many iterations have been computed for the next frame,
- controls for what part of the computed simulation should be displayed and how and how fast it should be played (there are three playback modes: *play until end* plays the frames at the set speed until it reaches the end of the computed segment where it can play no faster than the speed of the computation, *loop* plays the frames at the set speed and starts from the beginning whenever it reaches the end of the computed segment, *last frame only* always (unless the playback is paused) displays the last computed frame),
- controls for what should be visualized – what should be visualized in the background using color (*background visual*[3]), where red color stands for high values and blue for low values, and what in the foreground (*foreground visual*),
- and a button for opening the video export dialog.

For some combinations of simulation parameters, it may happen that the computational model collapses and the simulation diverges. Indication that this happened is then displayed below the switch for pausing the computation, the computation is stopped, and it cannot be resumed.

The other window in the simulation state is the video export dialog, which can be opened from the control panel in the main window. The dialog lets the user set the start and the end times of the exported video and the playback speed. It also allows the user to preview the video that will be exported by letting the simulation play as it will in the video – the output of this preview is displayed in the display area in the main simulation window. Even when the export dialog is open, the visuals can be controlled by the control panel in the main simulation window and changes in the setting of

[3]Apart from the visuals with self-explanatory names, *relative vorticity* stands for vorticity divided by velocity magnitude and *velocity divergence* should theoretically be zero everywhere (since the fluid is incompressible), but it is not exactly zero due to approximations in the model.

the visuals are immediately reflected in the preview and in the exported video (unless the export operation has already been started). At last, the video export dialog contains entries for configuring the width, height, bitrate, and filename of the exported video.

# Conclusion

Software developed in this project completely fulfills the assigned specification. The specification was not overly concrete as to what exact features the software should have (e.g. what modes of visualization should be available, what simulation parameters should be configurable etc.), but I believe that the implemented features are fully adequate for what is required by the specification. Despite all this, even more features were originally considered – such as the simulation of compressible fluids –, but these were later excluded as the project turned out to be complex enough.

I think it would have been a good idea to do even more research at the beginning of the project – especially in the field of CFD. That could have helped me develop a reliable computational model faster. On the other hand, I had the opportunity to experiment with different models I came up with myself, which deepened my intuition in CFD and more generally in numerical methods and computer simulation too.

Overall, this was not an easy project. It was definitely one of the largest pieces of software I have ever created and it took a proportionally large amount of time, a sizeable portion of which has been spent on watching the computed simulations and trying to recognize the displayed physical phenomena. Even though I already had experience in software development before this project, it has taught me new skills in project management and in the C++ programming language.

There were a few unexpected problems, such as the already mentioned bugs in GTK+, but with these few exceptions, I think I did well and enjoyed working on the project.

# Bibliography

[1] Murray Cumming et al. *Programming with gtkmm 3*. Version 3.24. URL: `https://developer.gnome.org/gtkmm-tutorial/3.24/`.

[2] Lorena A. Barba. *ME 702 – Computational Fluid Dynamics*. URL: `https://www.youtube.com/playlist?list=PL30F4C5ABCE62CB61`.

[3] Lorena A. Barba and Gilbert F. Forsyth. “CFD Python: the 12 steps to Navier-Stokes equations”. In: *Journal of Open Source Education* 2.16 (2018), p. 21. DOI: `https://doi.org/10.21105/jose.00021`.

[4] Alexandre Joel Chorin. “Numerical Solution of the Navier-Stokes Equations”. In: *Mathematics of Computation* 22.104 (Oct. 1968), pp. 745–762. DOI: `https://doi.org/10.2307/2004575`.

[5] *CMake Reference Documentation*. URL: `https://cmake.org/cmake/help/v3.20/#`.

[6] The Qt Company. *Qt*. URL: `https://www.qt.io/`.

[7] cppreference contributors. *cppreference.com*. URL: `https://en.cppreference.com/w/cpp`.

[8] James F. Epperson. *An Introduction to Numerical Methods and Analysis*. Second Edition. Hoboken, New Jersey, USA: John Wiley & Sons, Inc., 2013. ISBN: 978-1-118-36759-9.

[9] Viktor Fukala. *gtk_widget_destroy on GtkSwitch inside a window segfaults*. Mar. 2021. URL: `https://gitlab.gnome.org/GNOME/gtk/-/issues/3720`.

[10] *GTK+ 3 Reference Manual*. URL: `https://developer.gnome.org/gtk3/stable/`.

[11] *gtkmm Reference Manual*. Version 3.24. URL: `https://developer.gnome.org/gtkmm/3.24/`.

[12] OpenCFD Ltd. *OpenFOAM. The open source CFD toolbox*. URL: `https://www.openfoam.com/`.

[13] Yuki Minamoto. *Flowsquare. The free, handy, integrated, Computational Fluid Dynamics software*. URL: `http://flowsquare.com/`.

[14] *MSYS2*. URL: `https://www.msys2.org/`.

[15] Stanley Osher and Ronald Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Vol. 153. New York, NY, USA: Springer–Verlag, 2003. ISBN: 0-387-95482-1.

[16] Marcin Osypka. *using Gtk::Switch causes segmentation faults on window close*. Dec. 2020. URL: `https://gitlab.gnome.org/GNOME/gtk/-/issues/1298`.

[17] Sergey Stepanov. *GtkGLArea drawing issues during resize*. Aug. 2018. URL: `https://gitlab.gnome.org/GNOME/gtk/-/issues/1298`.

[18] Inc. The Khronos Group. *OpenGL® 4.5 Reference Pages*. URL: `https://www.khronos.org/registry/OpenGL-Refpages/gl4/`.

[19] *Using liav**. URL: `https://trac.ffmpeg.org/wiki/Using%5C%20libav*`.

[20] Wikipedia contributors. *Finite element method — Wikipedia, The Free Encyclopedia.* [Online; accessed 2021/03/21]. 2021. URL: `https://en.wikipedia.org/wiki/Finite%5C_element%5C_method`.

[21] Wikipedia contributors. *Gauss-Seidel method — Wikipedia, The Free Encyclopedia.* [Online; accessed 2021/03/21]. 2021. URL: `https://en.wikipedia.org/wiki/Gauss%5C%E2%5C%80%5C%93Seidel%5C_method`.

[22] Wikipedia contributors. *Jacobi method — Wikipedia, The Free Encyclopedia.* [Online; accessed 2021/03/21]. 2021. URL: `https://en.wikipedia.org/wiki/Jacobi%5C_method`.

[23] Wikipedia contributors. *Navier-Stokes equations — Wikipedia, The Free Encyclopedia.* [Online; accessed 2021/03/21]. 2021. URL: `https://en.wikipedia.org/wiki/Navier%5C%E2%5C%80%5C%93Stokes_equations`.

[24] Wikipedia contributors. *Successive over-relaxation — Wikipedia, The Free Encyclopedia.* [Online; accessed 2021/03/21]. 2021. URL: `https://en.wikipedia.org/wiki/Successive%5C_over-relaxation`.