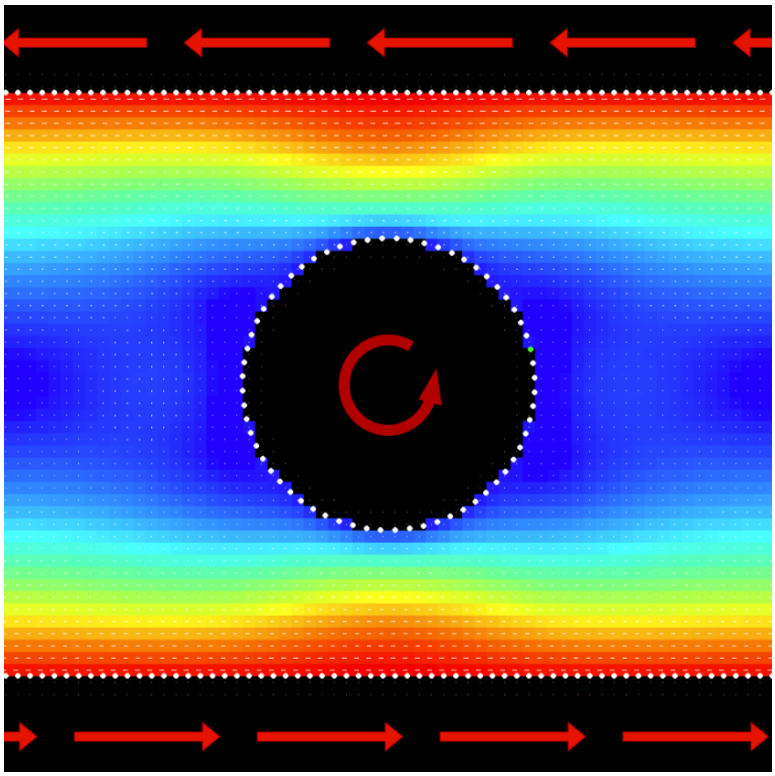
Utrecht University

Enhancing a Time-Independent CFD Solver with Moving Boundaries and Solids



Author
Pieter Michels
(5081211)

Supervisors
Dr. Deb Panja
Dr. Joost de Graaf

July 5, 2024

Abstract

Computational Fluid Dynamics solvers are a widely used tool to simulate the flow of a fluid and the motion of objects. When working with a time independent solver, computing object motion requires additional steps. In this thesis, methods are presented to introduce object rotation and translation to a solver based on the linear time-independent Stokes equations. It is shown that, using the results obtained by running such a solver, it is possible to update boundary conditions between time steps. Furthermore, a way to move said boundary conditions around in a scene is provided. Additionally, the solver used in this thesis is extended with a convenient Python side library. The newly added features and theories in this work provide a solid basis for future use in the field by researchers.

Cover figure: An example of a velocity field visualized by the solver. A disk, with some initial angular velocity, is placed in the exact center of a Couette flow configuration. Over time the disk reaches an equilibrium angular velocity as imposed by the fluid flowing around it.

Table of Contents

Section 1 **3**

Preface 3

Section 2 **4**

Introduction 4
2.1 Document Structure 4
2.2 Physics Background 5
2.3 Force to Velocity Field 6
2.4 Gradient Descent 7
2.5 Research Questions 8

Section 3 **9**

Methodology 9
3.1 Accessibility 9
3.2 GPU Porting 9
3.3 Time-Varying Boundaries 9
3.4 Moving Solid Objects 10

Section 4 **12**

Implementation 12
4.1 Tools and Libraries 12
4.2 Solver Overview 12
4.2.1 System Initialization 12
4.2.2 Building a Configuration 13
4.2.3 Configuration Locking 13
4.2.4 Locking in a Step 14
4.2.5 Velocity Field Calculation 14
4.2.6 Stopping Conditions 14
4.2.7 Gradient Descent Step 15
4.2.8 Updating the Learning Rate 15
4.3 Time Varying Boundaries 16
4.4 Moving Objects 16
4.5 Python Bindings 17
4.6 Mega Kernel Implementation 18

Section 5 **20**

Results 20
5.1 Python Bindings 20
5.2 Time Varying Boundaries 21
5.3 Object Rotation 21
5.4 Answering the Research Questions 22

Section 6 **24**
Conclusion 24

Section 7 **25**
Additional Work 25

Section A **26**
Program Flow 26

Section B **27**
Python Functions 27

Section 1

Preface

In this thesis, accessibility improvements and new features are added to an existing computational fluid dynamics solver. In previous iterations of the code, the difficulty of running it with specific, custom configurations was challenging. Furthermore, due to the nature of the solver, moving objects and conditions were yet to be explored. To solve these issues, nearly all of the remaining hard coded features were made configurable by users through an easy to use Python extension. Additionally, the solver was extended to handle an arbitrary amount of moving objects of a complexity determined completely by the user of the software. Added computational overhead by the introduction of both of these features was compensated for by further small optimizations, ensuring that the loss in performance remains marginal.

Due to the project requiring knowledge of theoretical physics in conjunction with computer science it is supervised by two people with qualifications in both fields: Dr. Deb Panja and Dr. Joost de Graaf.

Section 2

Introduction

Numerical solvers are widely used to carry out research within the field of fluid dynamics. In this context, a solver is a tool that computes a fluid flow given a set configuration of boundary conditions, which can represent the presence of particles or other solids in the system. These help provide an understanding of experimental research and verifying analytical calculations. For example, the flow of blood running through an artery [1]. In blood flow, a finely resolved model of blood, requires the modeling of the red blood cells as moving objects. Thus, not only does a solver need to deal with flow, but also how this flow affects the motion of objects suspended within it.

In this project, a solver originally put together by Bart Stam [2] and subsequently extended upon by Florian Gaeremynck [3] and Valentijn van Zwieten [4] will be used as a base for the intended development. Together they developed a solver capable of numerically approximating solutions to the incompressible Stokes equations on a regular lattice. To obtain such a flow, the forces acting on a system's fluid as well as the boundary conditions at coupling points, that lie between the lattice cells, need to be specified. The goal of this thesis is to broaden the solver's capabilities by improving its accessibility and introducing object motion. This should provide a good basis to further experiment with runtime and memory optimizations and performance comparisons with other available solvers.

2.1 Document Structure

A brief overview of the physics background and current software implementation of this solver is provided in Subsection 2.2. Following that, the research questions this thesis aims to answer are discussed in Subsection 2.5. To aid in reading, a list of notations used throughout this thesis may be consulted in Table 2.1. The methodology, found in Section 3, discusses the information needed to answer the research questions. Next, in Section 4, the solver's implementation is discussed. The results of the work done is found in Section 5, followed by a conclusion in Section 6. Any future work that should be considered for this project is briefly discussed in Section 7.

Symbol	Definition
N, M, L	The width, height and depth of a configuration, respectively
$G_{a,b}$	$\frac{\partial U}{\partial F}$ (on the axis combination a, b)
$\mathbf{F}_o$	The set of virtual forces (belonging to object o)
B_o	The set of boundary points (belonging to object o)
$\vec{U}_a$	The velocity field in real space (on axis a)
$\tilde{U}_a$	The velocity field in Fourier space (on axis a)
$\vec{F}_a$	The force field in real space (on axis a)
$\tilde{F}_a$	The force field in Fourier space (on axis a)
C_i	The cost at iteration i
γ_i	The learning rate at iteration i
$\mathcal{I}_o(\vec{p})$	A function that checks if $\vec{p}$ is inside object o
$\mathcal{F}(\vec{p}, t)$	A time-varying boundary function

Table 2.1: Notations used throughout this thesis. More exact definitions will be given in the relevant sections.

2.2 Physics Background

The Navier-Stokes equations are central to this solver and are a crucial part for solving many problems in the field of computational fluid dynamics. Of particular interest are the incompressibility condition and the momentum equation for an incompressible flow. These are defined as follows, respectively:

$$\nabla \cdot \vec{u} = 0 \tag{2.1}$$

$$-\nabla \vec{p} + \vec{f} + \mu \nabla^2 \vec{u} = \rho \left(\frac{\partial}{\partial t} + \vec{u} \cdot \nabla \right) \vec{u} \tag{2.2}$$

where:

- $\vec{u}$ is the velocity of the fluid (in $\frac{m}{s}$),
- $\vec{p}$ is the pressure in the fluid (in $\frac{kg}{ms^2}$),
- $\vec{f}$ is the external force acting on the fluid in (in $\frac{kg}{m^2 s^2}$),
- μ is the viscosity of the fluid (in $\frac{kg}{ms}$),
- ρ is the fluid density (in $\frac{kg}{m^3}$) and
- t is time in seconds (s).

Equation 2.1 states that the divergence in the fluid velocity is 0 at all points, implying that the fluid is incompressible. The physical interpretation is that there is always exactly as much fluid flowing in as there is out of any controlled volume. Equation 2.2 is a form of Newton's second law ($\vec{F} = m\vec{a}$) adapted to the domain of fluid dynamics. From this perspective, we can make the following identifications: $\vec{F} \propto -\nabla \vec{p} + \vec{f} + \mu \nabla^2 \vec{u}$, $m \propto \rho$ and $\vec{a} \propto \left(\frac{\partial}{\partial t} + \vec{u} \cdot \nabla \right) \vec{u}$.

Due to the non-linear partial differential present in the acceleration term the Navier-Stokes equations are particularly challenging to solve numerically and can only be solved analytically for very simple systems in certain flow regimes. However, it can be done, and there is good reason to do so. Among the possible ways to approach CFD problems, the Stokes based approach is particularly favored when it comes to fluids with particles, such as polymers or bacteria, in them [5], which will be a major topic in this thesis. Another approach is that of Dissipative Particle Dynamics, or DPD,

which seems to mostly focus on physicochemical interactions within geometries. Furthermore, the Stokes based approach tends to be harder to implement than for instance the lattice Boltzmann method, which naturally lends itself well to parallel computation methods that are the norm when working with GPUs.

The solver developed by Stam, Gaeremynck and van Zwieten, solves the flow equations in a regime where the viscous dissipation is dominant. That is, internal friction is significant larger than inertia. In such a regime we refer to the flow as Stokes flow. A more mathematical definition of when a flow is considered a Stokes flow is given by the fluid's Reynolds (Re) and Strouhal (St) numbers, which are defined as

$$Re = \frac{\rho U L}{\mu} \tag{2.3}$$

$$St = \frac{L}{UT} \tag{2.4}$$

Where U is the flow velocity, L is a characteristic length and T is a characteristic time. Both numbers are dimensionless. Re describes the relation between inertial and viscous forces. When $Re \ll 1$ the viscous forces dominate the inertial ones, making the flow "laminar". The Strouhal number describes the relation between inertial forces due to localized and global acceleration. A flow is considered a Stokes flow when $ReSt \ll 1$.

In the regime the solver works with, the acceleration term can be neglected and the linear time-independent Stokes equations (referred to as LTIS) are considered.

$$\nabla \cdot \vec{u} = 0 \tag{2.5}$$

$$-\nabla \vec{p} + \vec{f} + \mu \nabla^2 \vec{u} = \vec{0} \tag{2.6}$$

Note that time dependence can be introduced to these equations. Doing so can be done simply via $\vec{f}$ by running the solver multiple times. Additionally, it can be done using the boundary conditions, either by changing the boundary conditions or by moving the points where the conditions are evaluated. In theory, this process can also be optimized by priming the solver using a previously found solution, which should already approximate the solution better than a $\vec{0}$ force field if the steps are small enough.

With the LTIS it is much easier to obtain a velocity field given a force field. What remains to be done is solving a set of linear partial differential equations. That is, the derivative of a very intricate function needs to be found. There are several ways in which this can be done. In this thesis Fourier transforms and projections to obtain numerical solutions will be used.

2.3 Force to Velocity Field

To find the unique solution to the LTIS equations the solver operates on a discrete grid of size N by M by L. This grid is also periodic, which is a necessary property when working with Fourier transforms. A Fourier transform is a linear method that transforms a function into a set of sine and cosine functions that describe a frequency domain. The reason for using this Fourier space is the fact that the derivatives of sine and cosine functions are trivial to calculate. On the grid, both a force field $\vec{F}$ and a velocity field $\vec{U}$ are defined. In the default, zero flow state, both of these fields are equal to $\vec{0}$. Upon changing $\vec{F}$, to satisfy the no-slip boundary condition, the fluid flow changes accordingly. This relationship is defined as follows:

$$\tilde{U} = \frac{1}{\mu k^2} \left(I_3 - \tilde{k} \otimes \tilde{k} \right) \tilde{F} \tag{2.7}$$

where $\tilde{F}$ and $\tilde{U}$ are the force and velocity field in Fourier space respectively and I_3 is the three dimensional identity matrix. The derivation of the above relation may be found in *A GPU-based versatile and efficient hydrodynamics code for scientific applications* [2]. Calculating $\tilde{F}$ from $\vec{F}$ is done using a Fourier Transformation. This process can also be inverted, allowing the solver to compute $\vec{U}$ from $\tilde{U}$. Specifically, since the grid that is operated on is discrete, the solver makes use of (inverse) Discrete Fourier Transforms, or DFTs. These, as more thoroughly discussed in Subsubsection 4.2.5, can be computed very efficiently using Fast Fourier Transforms (FFTs).

2.4 Gradient Descent

The Fourier transform based approach converts forces into fluid velocity but it does not yet take into account any solid objects (or "solids") present in the simulation. A no-slip boundary condition [6] needs to be introduced to achieve these boundaries. This condition states that a fluid's velocity on the boundary of a solid must be $\vec{0}$ relative to the surface of the solid. In technical terms, a constraint on the velocity field is introduced within solids in the system.

To satisfy the no-slip boundary condition a set of virtual forces is introduced inside of the solids that need to be tuned to meet the boundary constraints. Since the LTIS has a unique solution, the size and direction of the virtual forces can be established using an iterative approach. In this thesis, gradient descent is used. As with any gradient descent algorithm, the goal is to minimize a cost function C. In the case of this solver, C is the Euclidean distance between the desired velocity of the boundary points and the actual velocities that $\vec{U}$ imposes on said boundary points. This gives the following function:

$$C = \sqrt{\sum_{b \in B} \|\vec{u}_b - \vec{du}_b\|^2} \tag{2.8}$$

Where $\vec{u}_b$ and $\vec{du}_b$ are the current and desired velocity of boundary point b respectively.

When performing gradient descent, the system moves along the gradient that defines the change in cost with respect to changes in the set of virtual forces $\mathbf{F}$ as follows:

$$\nabla V = \left[\frac{\partial C}{\partial f}\right] \tag{2.9}$$

Where f is a virtual force in $\mathbf{F}$. At some iteration i, a step is taken of size γ_i, in the direction $-\hat{V}$. Choosing a good initial learning rate γ_0 and a good method of updating the learning rate each iteration is a highly researched subject. Currently, the solver comes with several built-in learning rate schemes. These are discussed in more detail in Subsubsection 4.2.8. Fortunately, the cost function C is convex. As a result, scenarios such as getting stuck in local minima do not need to be accounted for.

To minimize the cost function, a way to determine the effect of nudging a virtual force on the velocity of all boundary points is needed. This is done using a gradient $G_{a,b}$ which is defined as

$$G_{a,b}[i,j] = \frac{\partial \vec{U}_a[i]}{\partial \vec{F}_b[j]} \tag{2.10}$$

where

- G is the gradient.
- i is the index of the cell in $\vec{U}$.
- j is the index of the cell in $\vec{F}$.

- a is the axis of the cell in $\vec{U}$.
- b is the axis of the cell in $\vec{F}$.

Computing G is described in more detail by van Zwieten [4], who optimized the process to only a single calculation.

2.5 Research Questions

Although the solver is capable of finding solutions already, there is still room for significant improvement in terms of performance, modularity and capability. As mentioned in the Future Work section of the research thesis *Bringing GPU Parallelization and Complex Boundaries to a Computational Fluid Dynamics Solver* [4], the solver can be expanded as follows: A Graphical User Interface (GUI) or some other way to easily interact with the program can be added, such that it is easier for an end-user to define custom configurations. In addition, extending the solver by making it capable of handling moving boundary conditions, such as solids moving the flow or inducing flow, would tremendously increase the solver's overall usefulness and will therefore be the main topic of this research.

Within this research project, the following research questions will be addressed:

1. The solver is written in the C++ programming language, which is quite challenging to edit and at present requires recompilation to run custom configurations. This problem could be avoided by introducing a GUI or by Python interface. Hence the question: *Can the solver be made more accessible with both preset and custom configurations?*
2. While the solver mostly uses the GPU for its calculations, a significant amount of data copying and context switches remains. This raises the question: *Is it possible to reduce the number of switches between CPU and GPU by implementing the algorithm as a whole on the GPU?*
3. The program solves a stationary flow field. That is, for a problem that involves moving boundaries, it solves one time step of the simulation. It would be interesting to allow for scenarios where the desired velocities of boundary points change over time. *Can time-varying boundaries be introduced to the program while maintaining performance and accuracy?*
4. Currently, the solver is only capable of handling static solids. For the experiments that are to be verified with this program it is interesting to introduce moving solids as well. *Can groups of boundary points be introduced that act as a moving object in the simulation while maintaining performance and accuracy?*

Section 3

Methodology

In this section the methodology required to answer the research questions in Subsection 2.5 is provided. The time integration needed for time-varying boundaries and moving boundary points will be the main focus.

3.1 Accessibility

To make the code easier to use several options were considered. The option of building a GUI was quickly dismissed in favor of Python bindings as Python is a widely used language in the scientific community and is quite easy to understand and learn. When setting up Python bindings there are many libraries to choose from. In this thesis, Cython [7], pybind11 [8] and nanobind [9] were considered.

Each of the above obviously has their advantages and disadvantages. Cython requires developers to keep a `.pyx` file on the side which acts as a wrapper around the C++ functions that are chosen to be available in Python. This `.pyx` file compiles to C++ code and therefore is expected to be the fastest out of all three options. Importantly though, there is no standardized way to integrate C++ dependencies, such as the CUDA Toolkit and some of its libraries, into a Cython based package. In contrast to Cython, both pybind11 and nanobind support the use of C++ dependencies in several ways, most notably CMake. Furthermore, the two have very similar conventions and syntax. However, in this case, nanobind has the edge over pybind11 as it was made with performance in mind. While it does drop some features from pybind11, such as multiple inheritance, those features are inherently poor in performance and therefore not used in the solver to begin with. Hence, nanobind will be used to generate python bindings for the solver, providing Python's package installer pip as an easy installation mechanism.

3.2 GPU Porting

The current implementation involves a loop to determine the virtual forces. Within this loop a lot of communication and synchronization between CPU and GPU happens, which substantially slow down the algorithm The conversion to a pure GPU based implementation does not require too many changes. Most of the algorithm used already runs on the GPU anyway. However, one major change will be swapping out the cuFFT library [10] with the cuFFTDx library [11]. The big difference between the two is that cuFFT is callable exclusively from the CPU, forcing any GPU implementation to work with the more hands-on cuFFTDx.

3.3 Time-Varying Boundaries

Both time-varying boundaries as well as moving groups of boundary points require the introduction of time to the simulation. Fortunately, the existing algorithm does not need to be changed in order to handle this. For each time step the current algorithm can simply be used to find a solution. Furthermore, for each time step, the solution of the previous step can be used as input. In theory, this should reduce the time needed to find solutions in consecutive runs of the solver.

3.4 Moving Solid Objects

The moving of a solid as modeled by a group of boundary points comes down to a lot more than just perturbing the desired velocities. In particular, calculating the velocity of the points approximating the moving surface is of interest. Dünweg and Ladd touch upon this in the book *Advanced Computer Simulation Approaches for Soft Matter Sciences III* [12]. Specifically, they state that the local velocity of a point b on a surface can be calculated as such:

$$\vec{u_b} = \vec{U} + \vec{\Omega} \times \left(\vec{r_b} - \vec{R}\right) \tag{3.1}$$

Where

- $\vec{U}$ is the particle's velocity,
- $\vec{\Omega}$ is the angular velocity at point b,
- $\vec{r_b}$ is the boundary node location and
- $\vec{R}$ is the particle's center of mass.

Given this method of calculating the desired velocity of all boundary points $b \in B_o$ for any object o, it is possible to compute an angular velocity $\omega_{o,e}$ for which the forces acting on o are in equilibrium with the forces the fluid flow imposes on o. That is, the point at which the torque τ_o acting on object o is equal to $\vec{0}$ needs to be found. For example, consider a disk d placed in the center of a Couette flow configuration, as shown in Figure 3.1, which is given an initial angular velocity $\omega_{d,init}$ such that $\omega_{d,init} \neq \omega_{d,e}$. With the solution found to this configuration, the sum of forces in $\mathbf{F}_d$ is not equal to $\vec{0}$. Since the only forces that can directly influence the torque on d are the virtual forces in $\mathbf{F}_d$, only those forces need to be adjusted to reach the equilibrium angular velocity $\omega_{d,e}$.

Forces that are set to satisfy boundary conditions are not the only ones influencing objects in the fluid. In addition to rotation due to torque, objects may be subject to translation as a result of pressure differences. A commonly used example is that of an airplane, which gets pushed up into the air at high velocities due to pressure difference above and below the wings. Computing the pressure field, while not trivial, can be done using the force field $\vec{F}$. Given some object o, the pressure on its surface S_o may be found by

$$\oiint_{S_o} d\vec{S}\, p(\vec{S}) \tag{3.2}$$

where $\vec{S}$ is some vector on the surface S_o and p is the pressure. Combining the knowledge that $d\vec{S} = \hat{n}dS$, with $\hat{n}$ being the surface normal, and that the integral of the pressure over S_o is equal to the force on the object, this can be derived to

$$\oiint_{S_o} dS_o\, \hat{n} \cdot (p\mathbb{I}_3) \tag{3.3}$$

Since $p\mathbb{I}_3$ is a tensor, the divergence theorem can be used to obtain

$$\iiint_{V_o} dV_o\, \left(\overline{\nabla} \cdot \mathbb{I}_3\right) \tag{3.4}$$

where V_o is the volume of object o. Rewriting the pressure tensor as $\rho\mathbb{I}_3$ further simplifies the problem:

$$\oiint_{S_o} dS_o\, \hat{n} \cdot (p\mathbb{I}_3) = \iiint_{V_o} dV_o\, \left(\overline{\nabla} \cdot (\rho\mathbb{I}_3)\right) = \iiint_{V_o} dV_o\, \left(\overline{\nabla} p\right) \tag{3.5}$$

A definition for $\left(\overline{\nabla} p\right)$ can be derived from the LTIS, which gives:

$$\Delta p = \overline{\nabla} \vec{f} \tag{3.6}$$

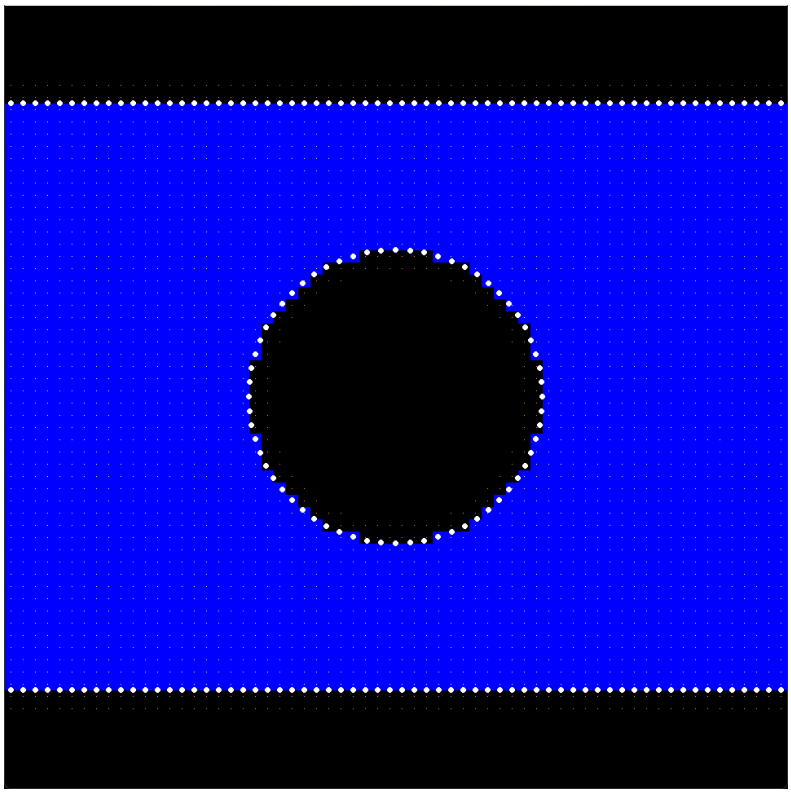


Figure 3.1: A disk placed in the exact center of a Couette flow configuration.

This can be transformed into Fourier space, like so:

$$\check{p} = -\frac{i\overline{k} \cdot \check{\overline{f}}}{k^2} \tag{3.7}$$

Joining Equation 3.6 and 3.7 a definition for $(\check{\overline{\nabla} p})$ is found as:

$$(\check{\overline{\nabla} p}) = (\hat{k} \otimes \hat{k})\check{\overline{f}} \tag{3.8}$$

Therefore, when $\overline{\nabla} p$ is needed, it can be computed as

$$(\check{\partial p}) = \frac{i}{n} \sin(kh)\check{p} \tag{3.9}$$

where $\check{p}$ is defined as:

$$\check{p} = \frac{ih\left(\sin(k_x h)\check{f}_x(N,M,L) + \sin(k_y h)\check{f}_y(N,M,L) + \sin(k_z h)\check{f}_z(N,M,L)\right)}{2\left(\cos k_x h + \cos k_y h + \cos k_z h\right)} \tag{3.10}$$

Similar to determining ω_o, e, it is now possible to determine a set of forces on object o, resulting in a velocity towards a position where the pressure will be equal to 0. Verification of this process can be achieved in the same type of configuration used for object rotation. When placing a disk inside of a Couette flow the pressure gradient should push the circle to the center of the channel.

Section 4

Implementation

In this Section, details are provided on the way the theory that was introduced in Section 3 is implemented. Firstly, any tools and libraries that the solver makes use of are presented. Secondly, an overview of the solver's execution order is given. Next, the way the Python bindings interface with the code is concisely discussed. Finally, the progress made towards converting the GPU code to a mega kernel is shown.

4.1 Tools and Libraries

Since the original code was written in C++ the decision was made very early in the project to stick with this programming language. Besides the obvious benefit of not having to port the code to another language, C++ offers great control over low-level operations and structures, which is highly beneficial for runtime optimization. Furthermore, the code that is executed on the GPU is currently written using NVIDIA's Compute Unified Device Architecture (CUDA) language. CUDA, and the NVIDIA GPU's that support it, are widely used in the scientific community. As such, there are several highly optimized libraries that are available, making the continued use of this language an obvious choice. Additionally, some CPU side libraries were used for the sake of implementation convenience. All the used libraries are:

- *argparse* which is used to handle command line arguments. Though mostly made redundant by the introduction of the Python bindings, the executable was kept updated for the sake of completeness.
- *FFTW*, *cuFFTDx* and *cuFFT* are used for the execution of FFTs. The first of the three is CPU only, whereas the other two require a GPU.
- *CImg* is used, albeit sparingly, to visualize the solver's velocity field. Although this part of the code is incredibly inefficient, it has proven to be useful for ensuring configurations are built correctly and giving a quick yet low detail overview of found solutions.
- *CUB* is used for block wide kernel programming. This library provides highly optimized and easy to use functions for several common operations. The block wide reduction was of particular interest in this project.
- *nanobind* to simplify the building of Python bindings for the solver.

4.2 Solver Overview

In this section, the implementation details of the solver are discussed. All steps of the solver, excluding the logging of results and memory freeing, are briefly explained in order of the diagram found in Appendix A.

4.2.1 System Initialization

The initialization of a solver instance currently boils down mostly to memory allocation and other setup requirements. Firstly, buffers are created for the force and velocity fields - for both real and

Fourier space - as well as the grid cell states. Secondly, the plans for the FFTs are created. Furthermore, the standard settings for stopping conditions and a learning rate scheme are set as detailed in Subsubsection 4.2.6 and Subsubsection 4.2.8 respectively. Initialization then diverges: The CPU at this point is already done. However, the GPU saves the $\frac{\partial U}{\partial F}$ gradient across a set of textures, which require separate initialization alongside the creation of underlying arrays. Moreover, additional buffer memory has to be allocated on the GPU for the force and velocity fields, again, in both real and Fourier space.

4.2.2 Building a Configuration

In the current iteration of the solver, a configuration is set either by preset or manually. For the presets, the same selection of seven as proposed by van Zwieten [4] is available. A render of these presets can be found in Figure 4.1. Manually setting a configuration can be done using the newly added Python bindings. The bindings offer an easy way to set up static parts of the configuration by setting solids and adding boundary points to the system. Furthermore, moving objects may be added, with its respective set of boundary points. This makes it substantially easier to create custom configurations. One such example is the previously used disk inside a Couette flow, as shown in Figure 3.1. Exact details on how to do this can be found in Subsection 4.5.

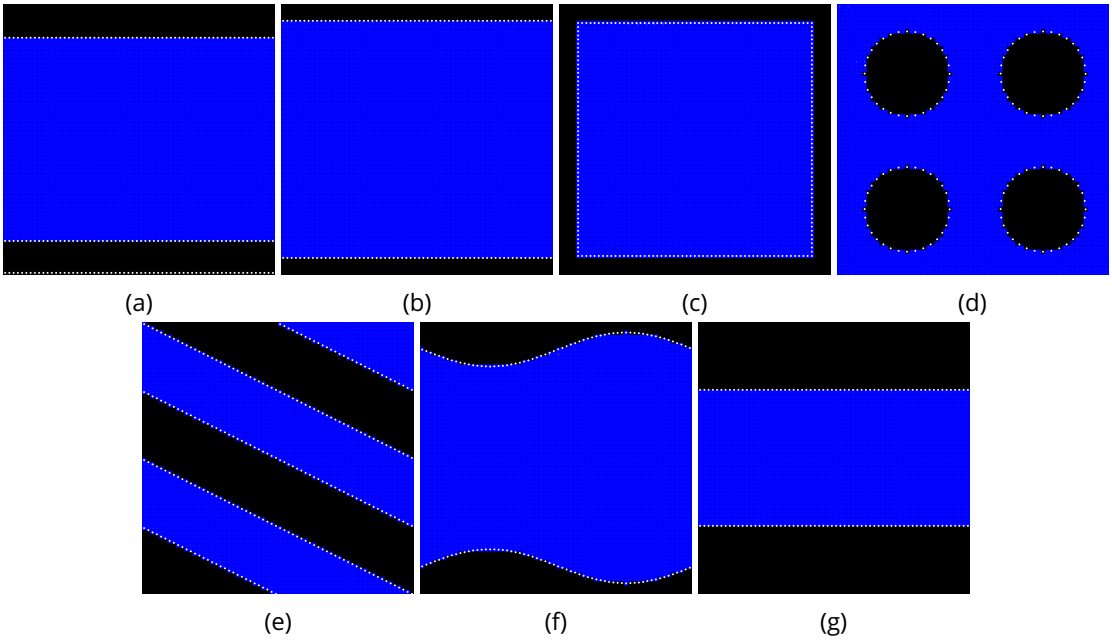


Figure 4.1: The seven available preset configurations: (a) Couette Flow (cou), (b) Two Moving Plates (2mp), (c) Lid Driven Cavity (ldc), (d) Four Roller Mill (4rm), (e) Tilted Moving Plates (tmp), (f) Sinusoidal Plates (sp) and (g) Pressure Pipe (pp). The blue areas denote liquid cells, whereas the black represent solid grid cells. The white dots are boundary points.

4.2.3 Configuration Locking

The locking of a configuration can be seen as a continuation of the initialization. It is done after initialization due to the fact that the number of boundary points or virtual, or both, need to be known at this point. During this phase of execution, any steps are taken that need to be done only once before finding a solution for a given configuration. That is, virtual forces in static objects are culled, the initial centers of mass for all objects are computed and any manually added forces are loaded in. Again, there is a split here between CPU and GPU code. On the CPU, memory is allocated

to the $\frac{\partial U}{\partial F}$ gradient now that the number of boundary points and virtual forces is known. Conversely, on the GPU, memory needs to be allocated for the centers of mass as well as the boundary points.

4.2.4 Locking in a Step

As a final step before the algorithm starts, the system locks in a step. This, as the name suggests, happens at the start of each step. During this step, the learning rate and all variables used for the dynamic Polyak system are reset to their initial values. More details on this can be found in Subsubsection 4.2.8. Furthermore, desired velocities of boundary points which are either subject to a time varying boundary condition or part of a moving solid are updated. Lastly, any solid cells belonging to a moving object, and resulting moving virtual forces, need to be updated. That is, a cell i in the system is set to be solid if there is an object o in the system for which the function $\mathcal{I}_o(\vec{p}_{i,o})$ returns true. The input $\vec{p}_{i,o}$ for this function is the coordinates of cell i with any translation and rotation to o undone. That is

$$\vec{p}_{i,o} = R_z(\vec{r}_{o,z})R_y(\vec{r}_{o,y})R_x(\vec{r}_{o,x})\,(\vec{p}_i - \vec{c}_o) \tag{4.1}$$

where $\vec{p}_i$ is the coordinates of cell i. As with any statically set solids, every solid cell will have a virtual force in it initially. The exact same method and parameters to cull these forces is used again for the virtual forces of objects, while making sure to not check statically set forces a second time. Furthermore, the velocity of boundary points belonging to each object o is updated according to its current angular velocity ω_o. That is, given the set of boundary points B_o, then for each point $b \in B_o$, located at coordinates $\vec{p}_b$, the desired velocity $\vec{du}_b$ is calculated using Equation 3.1.

4.2.5 Velocity Field Calculation

The very first step in the algorithm is computing the velocity field $\vec{U}$ from the current force field $\vec{F}$. To do this, $\vec{F}$ is transformed into Fourier space using a Fast Fourier Transform: FFTW on the CPU and cuFFT on the GPU. Using the formulas defined by Stam [2] $\vec{U}$ is then calculated from $\vec{F}$ in Fourier space. Next, with an inverse FFT, the velocity field is transformed back to real space.

4.2.6 Stopping Conditions

Checking the stopping conditions has undergone a major change. Importantly, users can now enable multiple stopping conditions at one time. In previous iterations, there was two conditions that were enabled by default: A maximum number of iterations and a learning rate that is effectively zero. In conjunction with these two, it was possible to use one additional condition. Now, only the maximum number of iterations remains as a default. The other learning rate schemes have been updated to be similarly configurable by the user:

- Error (0)
  This stopping condition checks the maximum difference in desired and actual velocity of all boundary points. That is, after an iteration, $\vec{du}_{max}$ is calculated as

  $$\vec{du}_{max} = \max_{b \in B} \|\vec{u}_b - \vec{du}_b\|_\infty \tag{4.2}$$

  Users can provide a maximum allowed error e_{max}. By default $e_{max} = 0.01$. When using this condition, the solver will consider a step solved when $du_{max} < e_{max}$.

- Percent (1)
  A rather important stopping condition is the convergence percentage check. As mentioned, the last few percentiles of convergence take the majority of the solver's time. Therefore, being

able to control when a found velocity field is considered a solution can save quite some time. Furthermore, it allows the setting of a desired quality of solution. Using this condition, the solver will exit when $C_i < C_{initial} \cdot (1 - \frac{l}{100})$, where l is the goal convergence percentage, which is 99.99% by default.

- Stuck (2)
  The stuck stopping condition is a very simple condition which stops the solving process if the cost has not decreased for j iterations. The default value of j is 500 but can once again be set by users.
- Learning rate (3)
  As the solver inches closer to the actual solution, the learning rate similarly approaches 0. Obviously, with a learning rate equal to 0, gradient descent does not actually ever improve the solution. Therefore, users can provide an r which, when this condition is used, will stop the solver when $\gamma_i < r$.

4.2.7 Gradient Descent Step

When performing gradient descent the solver attempts to nudge the virtual forces such that the resulting velocity field more closely resembles the desired velocities of the boundary points in the configuration. To do so, the virtual force at index j, referred to as $\vec{F}[j]$ is adjusted as follows:

$$\vec{F}[j] = \vec{F}[j] - \frac{\gamma_i \vec{s}[j]}{|\vec{s}[j]|} \tag{4.3}$$

where γ_i is the learning rate at iteration i and $\vec{s}[j]$ defines the size and direction of the step. It is calculated as follows:

$$\vec{s}[j] = \sum_{i=0}^{|B|} \left(\begin{bmatrix} G_{x,x}[i,j] & G_{y,x}[i,j] & G_{z,x}[i,j] \\ G_{x,y}[i,j] & G_{y,y}[i,j] & G_{z,y}[i,j] \\ G_{x,z}[i,j] & G_{y,z}[i,j] & G_{z,z}[i,j] \end{bmatrix} (\vec{u}_i - \vec{du}_i) \right) \tag{4.4}$$

4.2.8 Updating the Learning Rate

In the latest version of the solver, updating the learning rate is woven into the gradient descent step. This was done to reduce the amount of data copying needed between the CPU and GPU, especially for the Polyak methods. Currently, the following learning rate schemes are available:

- Constant (0)
  The learning rate has a constant value.

$$\gamma_i = \gamma_{i-1} \tag{4.5}$$

- Halving (1)
  The learning rate is initialized to a given value and then halved whenever the cost of the solution increases.

$$\gamma_i = \begin{cases} \gamma_{i-1} & \text{if } C_i \leqslant C_{i-1} \\ \frac{\gamma_{i-1}}{2} & \text{if } C_i > C_{i-1} \end{cases} \tag{4.6}$$

- Polyak (2)
  When dealing with convex optimization problems, the learning rate scheme as proposed by Boris T. Polyak [13] should, in theory, always find the global minimum. This Polyak's length is defined as

$$\gamma_i = \frac{C_i}{|\nabla V|^2} \tag{4.7}$$

- Dynamic Polyak (3)
  The Dynamic Polyak learning rate scheme is an extension on the Polyak learning rate introduced by Gaeremynck specifically for this solver. He made the observation that the solver tends to spend the majority of its time converging on the very last few percentiles of the solution. In an attempt to alleviate this problem, this scheme uses a multiplier in conjunction with Polyak's length to try and take bigger steps in consistently decreasing areas and smaller steps in consistently increasing areas of the function. It is defined as follows:

$$\gamma_i = \frac{C_i \cdot m_i}{|\nabla V|^2} \tag{4.8}$$

  Where the m_i is the multiplier at iteration i, defined as

$$m_i = \begin{cases} 1 & \text{if } i = 0 \\ m_{i-1} \cdot m_b & \text{if } C_i < C_{i-1} \text{ for } b \text{ consecutive } i \\ m_{i-1} \cdot m_w & \text{if } C_i \geqslant C_{i-1} \text{ for } w \text{ consecutive } i \end{cases} \tag{4.9}$$

  Where b, w, m_b and m_w are all configurable by the user. By default, these are set to 100, 2, 10 and 0.1 respectively.

4.3 Time Varying Boundaries

The first introduction of multiple steps to the solver was Time Varying Boundaries (TVBs). Users can provide a TVB function through the Python bindings, as detailed in Subsection 4.5, which will be used to scale the desired velocity of all boundary points at the start of each step. That is, given a TVB function $\mathcal{F}$, when given the coordinates $\vec{p}_b = (x_b, y_b, z_b)$ of a boundary point $b \in B$ and the accumulative step size t, the desired velocity of b at step t ($\vec{du}_{b,t}$) is calculated as

$$\vec{du}_{b,t} = \mathcal{F}(\vec{p}_b, t)\vec{du}_b \tag{4.10}$$

Where $\vec{du}_b$ is the boundary point's original desired velocity.

4.4 Moving Objects

The introduction of moving objects required additional data for both the boundary points as well as a smart way of storing information concerning the objects themselves. Firstly, the solver has to know which boundary points belong to which object. This is done using a simple id that is added to the structure. Secondly, a list of object data is required. For each object o in the scene the following data is stored:

- The center of mass $\vec{c}_o$.
- The object's current rotation $\vec{r}_o$.
- The current angular velocity $\vec{\omega}_o$.
- The number of boundary points $|B_o|$.
- The number of virtual forces $|\mathbf{F}_o|$.

Out of all of these values, only ω_o needs to be given an initial value. All the others are calculated internally. Updating objects happens in two distinct steps. Applying an object's angular velocity to

its boundary points, setting solid cells and the corresponding virtual forces happens as discussed in Subsubsection 4.2.4.

The second, and more interesting part of object updates, is of course moving them. Since this requires the solution to the current configuration, calculating object movement is done as last in each step. Unfortunately, due to time limitations, translation was not implemented. However, rotation is fully implemented. Using gradient descent, the angular velocity at the equilibrium point $\vec{\omega}_{o,e}$, as discussed in Subsection 3.4, is found for each object. This gradient descent starts with a learning rate equal to that provided for the main algorithm. To evaluate the torque working on each object, the solver is run with the same stopping conditions and learning rate schemes. However, it is only given ten percent of the maximum number of iterations. This way, performance should remain within reason and the results should be decently accurate. Using the approximated $\vec{\omega}_{o,e}$, the rotation over a step of size t is calculated as

$$\delta\vec{r}_o = \frac{t}{2}(\vec{\omega}_o + (\vec{\omega}_o + t(\vec{\omega}_{o,e} - \vec{\omega}_o))) \tag{4.11}$$

Next, the new angular velocity of each object is computed:

$$\vec{\omega}_o = \vec{\omega}_o + t(\vec{\omega}_{o,e} - \vec{\omega}_o) \tag{4.12}$$

Following this, for all boundary points $b \in B_o$, the rotation caused by the average angular velocity over the executed step is applied:

$$\vec{p}_b = R_z(\delta\vec{r}_{o,z})R_y(\delta\vec{r}_{o,y})R_x(\delta\vec{r}_{o,x})\,(\vec{p}_b - \vec{c}_o) + \vec{c}_o \tag{4.13}$$

where $\vec{p}_b$ are the coordinates of b and $R_a(\theta)$ is a rotation matrix over axis a by an angle of θ.

4.5 Python Bindings

With Python bindings becoming available for the project options that were available exclusively through C++ preprocessor directives were converted to runtime defined and configurable variables. Despite this hampering performance a tiny bit, the amount of customizability it gives to users more than makes up for this. Initially, the code had the following features locked behind directives:

- Enabling use of the Non-Uniform Discrete Fourier Transform (NUIDFT).
- Enabling virtual force approximation.
- Enabling virtual force culling.
- Execution platform.
- Floating point precision.
- Running in verbose mode.
- Used Learning Rate Method.
- Used Stop Condition.

While already an improvement over previous iterations of the solver, this still left much to be desired. In the most recent version, all but two of these options are configurable through several different functions. The floating point precision remains a compile-time defined option for the simple reason that it would require essentially duplicating the entire code base for the single precision (float) and double precision (double) types. The NUIDFT, while novel, was lacking in performance and incredibly complex. Therefore, the decision was made to remove it from the code for now.

The implementation of the Python bindings is done in three separate files:

1. CMakeLists.txt
   This file controls the compilation of the Python package, installation of dependencies and finally installing the package using nanobind's functions.
2. pyproject.toml
   Here, the configuration settings for the package such as the version number, required Python versions and dependencies and the package name are listed.
3. fft_fluid_ext.cpp
   This C++ file contains the direct bindings, connecting the C++ side functions and respective arguments to functions that will be available in the Python package.

Through these bindings, the solver's full capability is exposed. Notably, the bindings allow for setting several variables that influence the learning rate, stopping conditions and virtual force culling. Additionally, via Python, users will be able to identify multiple objects in the simulation, with the option of having them move. The Python side functions can be divided into four groups:

1. Configuration presets
   These functions expose several preset configurations as written by van Zwieten. The available presets can be found in Table B.1.
2. Manual configuration control
   Besides the presets, the bindings expose functions that allow the user to build their own configurations. A list of these can be found in Table B.2.
3. Parameter setting
   Similar to the command line arguments, the Python package offers control over various parameters influencing the solver. However, the bindings take this one step further, allowing variables to be passed through to be used during execution. The documentation for these functions is found in Table B.3.
4. Post-configuration functions
   These are functions that users may want to use once a configuration is built or once a solution has been found, such as displaying the velocity field or freeing up the allocated memory. Table B.4 contains a list of such functions.

A major advantage of the bindings is the possibility of using multiple stop conditions. Furthermore, users can provide a list of parameters for the stopping conditions (P_S) and learning rate methods (P_m) as specified in Subsubsections 4.2.6 and 4.2.8 respectively.

4.6 Mega Kernel Implementation

As mentioned in Subsection 3.2, porting the solving algorithm to the GPU in its entirety requires using NVIDIA's cuFFTDx library. To use this library, two problems need to be solved:

1. Due to the design of the library, the size of the FFTs needs to known at compile time.
2. Kernels that put the input data into the GPU's shared memory in the correct order need to be set up. Such kernels will be referred to as transpose kernels from now on.

The first issue is in immediate conflict with the current set up, in which the size of the system only gets defined at runtime. Obviously, having a list of all possible combinations of sizes is not a feasible

solution. The second idea that came to mind was zero buffering the FFTs: Always setting the FFTs to maximum size and not using anything beyond the N, M and L needed inputs. However, this approach has two major drawbacks. Firstly, it would most likely eat up all of the available memory on the GPU without ever actually using most of it. Secondly, after executing an FFT, all the empty output data would somehow have to be filtered out. Considering the complex nature of Fourier transforms, the decision was made not to do this. Another solution to the compile time known sizes presents itself in the form of another NVIDIA library: The NVIDIA Runtime Compilation library, or NVRTC. As the name suggests, this library allows for the compilation of CUDA kernels while the program is running. Since it is possible to pass parameters through this, it is possible to set up the cuFFTDx FFTs with the user provided dimensions quite easily. Most notably is the low impact on performance this has, considering that compiling these kernels only has to happen at system initialization.

Whereas the first problem was quite handily solved, figuring out the transpose kernels proved a much harder task. Running a one dimensional FFT is trivial. However, when dealing with three dimensions, rather than just one, it becomes much more problematic. The main obstacle is that, with cuFFTDx, it is not immediately possible to run anything beyond a one dimensional FFT. To execute an n dimensional FFT, the library requires n FFTs. This is exactly where the trouble with the transpose kernels comes in. When going from real to Fourier space the size of the data changes, complicating the process. Unfortunately, even with the assistance of NVIDIA employees, this issue was not solved in time.

Despite not managing to get a functioning kernel with cuFFTDx, an important discovery was made. It is most likely, except in very specific combinations of dimensions, not possible to create a singular kernel that executes the full algorithm. The FFTs that are created using cuFFTDx, and therefore by extension the transformation kernels, require very specific grid and block sizes. Therefore, unless a guarantee can be made that these are the same for all three axes, the Fourier transforms have to be split up into at least six separate kernels: Three for transforming the $\vec{F}$ into $\tilde{F}$ and three more for the inverse transform from $\tilde{U}$ to $\vec{U}$.

Section 5

Results

In this Section the achieved goals are presented. First of all, the improved accessibility as provided by the Python bindings is briefly highlighted. Secondly, some example configurations and possibilities are discussed regarding the time varying boundaries. Furthermore, the implemented rotation of moving objects is considered. Lastly, all the research questions are answered.

5.1 Python Bindings

With the introduction of Python bindings to the project, using the solver, especially with different configurations and settings, is easier than ever. Previously easy to edit items, such as the system dimensions and a preset configuration are all still available. However, nearly everywhere else the solver is now much more configurable. Firstly, setting up a configuration by hand is now possible. Users can add boundary points, or groups of boundary points in the shapes of lines, circles, spheres and superellipses, to the static parts of the configuration or any object they added. Furthermore, the necessary solids of the static part can be set. Additionally, manually adding a force that will not be touched by the solver, as long as it is not overriden by a virtual force, is now possible.

Secondly, the bindings offer a lot of control over solver variables. Users can toggle the culling of virtual forces and set the cull radius. Moreover, the solver can now make use of multiple stopping conditions, rather than just one. Better yet, as mentioned in Subsubsection 4.2.6, a lot of the variables used for these stopping conditions can now also be configured. Similarly, the Dynamic Polyak learning rate scheme can now also be edited to influence how and when the multiplier is updated.

Finally, obtaining output from running the solver has been made easier. The found velocity field can be visualized and obtained as a text file. This text output has also been extended. In addition to the velocity field, it now contains information concerning the virtual forces, boundary points and any objects that were added to the system. Furthermore, the main solve function returns an object, containing data about the following:

- The runtime of each step.
- The number of iterations run each step.
- The initial, lowest and final cost of each step.
- The learning rate at the end of each step.
- The torque and total force acting on each object, each step.
- The total runtime.

Previously, this information was only printed. Making it available in this manner allows the user to easily process the solver's output, should they wish to do so.

It is of note that the numerous tests ran and graphs generated throughout this project have all been obtained using these bindings. Setting up configurations and obtaining detailed results on the solving process is, compared to the previous editing of C++ code and the following compiling, very fast and easy. Performance seems to be on par with previous iterations of the solver, though further testing should be done to confirm this in greater detail. A full list of the available functions can be found in Appendix B.

5.2 Time Varying Boundaries

To confirm if the time varying boundaries were working two different configurations were tested, each with a separate function $\mathcal{F}$. The first was a Couette flow, which was given the function

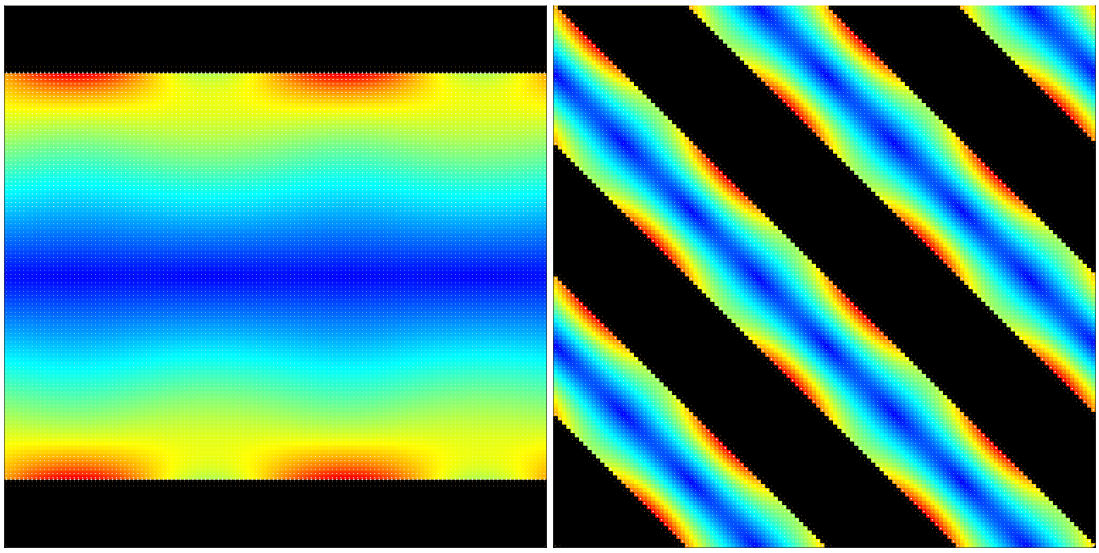
$$\mathcal{F}(\vec{p}, t) = 1 + \frac{1}{2}\sin(2\pi t)\sin\left(\frac{\pi \vec{p}_x}{M}\right) \tag{5.1}$$

Secondly, a tilted channels configuration, which was given a function that is functionally the same, just equally tilted:

$$\mathcal{F}(\vec{p}, t) = 1 + \frac{1}{2}\sin(2\pi t)\sin\left(\frac{\pi\left(\vec{p}_x + \vec{p}_y\right)}{M}\right) \tag{5.2}$$

where $\vec{p}$ is the position of a boundary point, $\vec{p}_a$ is the coordinate of p on axis a, t is the current time and M is the size of the system in the y dimension. These functions set up a sine wave over the boundary conditions. That is, the velocity field that is found should have peaks of higher velocity near the boundaries, intertwined with equally big drops in velocity. The resulting fluid velocity fields have been visualized, and can be seen in Figure 5.1. The solver appears to be capable of finding a solution in such configurations. However, it is important to note that, just like with moving objects, both speed and convergence seemed to improve upon resetting the force field between steps.

It is interesting to note that there is no functional difference between giving a solver a TVB function and executing multiple steps or changing the configuration on the Python side and executing each step separately. The only noticeable difference will be the requirement of having to initialize the system multiple times. However, the time varying boundaries served as a great way to test the how the solver responds to the introduction of time dependence by changing the boundary conditions.



(a) (b)

Figure 5.1: A Couette flow configuration (a) and a tilted channel configuration (b) when subject to a time varying boundary condition. The pictured states are both at $t = 0.25$.

5.3 Object Rotation

Before moving to testing, it was imperative to verify that an angular velocity $\omega_{o,e}$ exists for each object o in the scene where the torque on that object is equal to $\vec{0}$. To check this, three different shapes were placed in the exact center of a Couette flow configuration. By giving each shape a

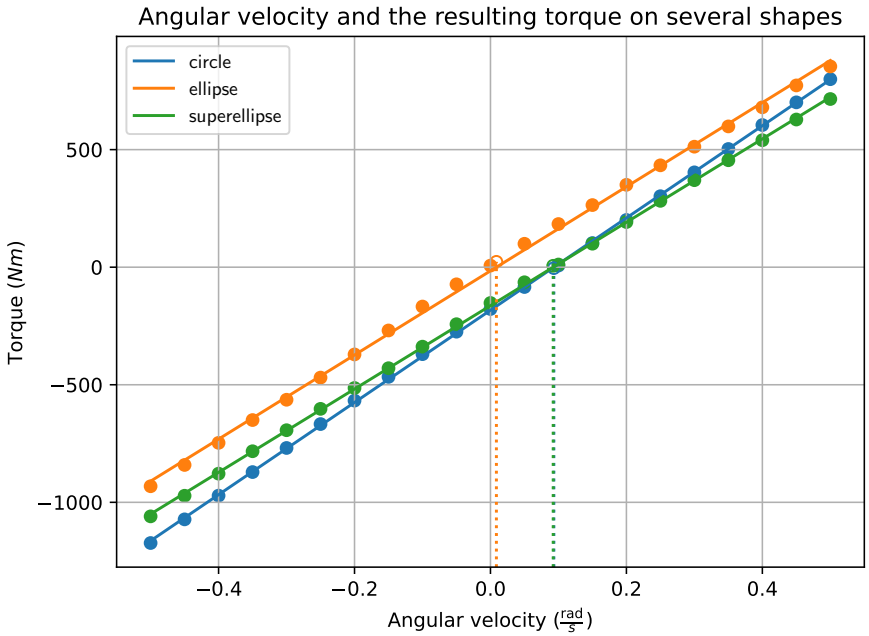


Figure 5.2: The torque on a circle, ellipse and superellipse as a result of a given angular velocity.

set of angular velocities to test against, a set of resulting torques was obtained. The results of this test can be found in Figure 5.2. As can be seen, the angular velocity and torque are in a linear relation. Sadly, as can also be seen in the Figure, in particular for the ellipse, the solver is not always accurate enough to find a point exactly on this line. Therefore, instead of interpolating $\omega_{o,e}$ from two distinct points on the line, gradient descent is used again to slowly approach this equilibrium angular velocity.

With the methodology for object rotation described in Subsection 3.4 and knowing that $\omega_{o,e}$ exists, it remains to show that finding this point indeed works. To this end, a circle with radius $r = \frac{3M}{16}$ was placed inside of a Couette flow configuration. This circle was given an initial angular velocity $\vec{\omega} = \left(0, 0, -\frac{5}{r}\right)^T$. The solver was then given 32 steps, each of size $\frac{1}{8}$ to get the configuration to its balanced state. The torque and angular velocity of the circle at each step can be found in Figure 5.3. As expected, the change in torque is exponential, and is, for this specific case, approximated by the function $-992.71e^{\frac{-x}{7.49}}$. The change in angular velocity is approximated by $0.09 - 0.51e^{-0.13x}$.

5.4 Answering the Research Questions

Given the research questions in Subsection 2.5 it is now finally possible to provide an answer to every one of them:

1. *Can the solver be made more accessible with both preset and custom configurations?*
   Yes, using a library, Python bindings have been generated, exposing the solver's full functionality as a small and easy to operate interface.
2. *Is it possible to reduce the number of switches between CPU and GPU by implementing the algorithm as a whole on the GPU?*
   While it is impossible to port the current implementation to a mega kernel structure, it is possible to convert the entire system to the GPU. However, doing so requires additional work be

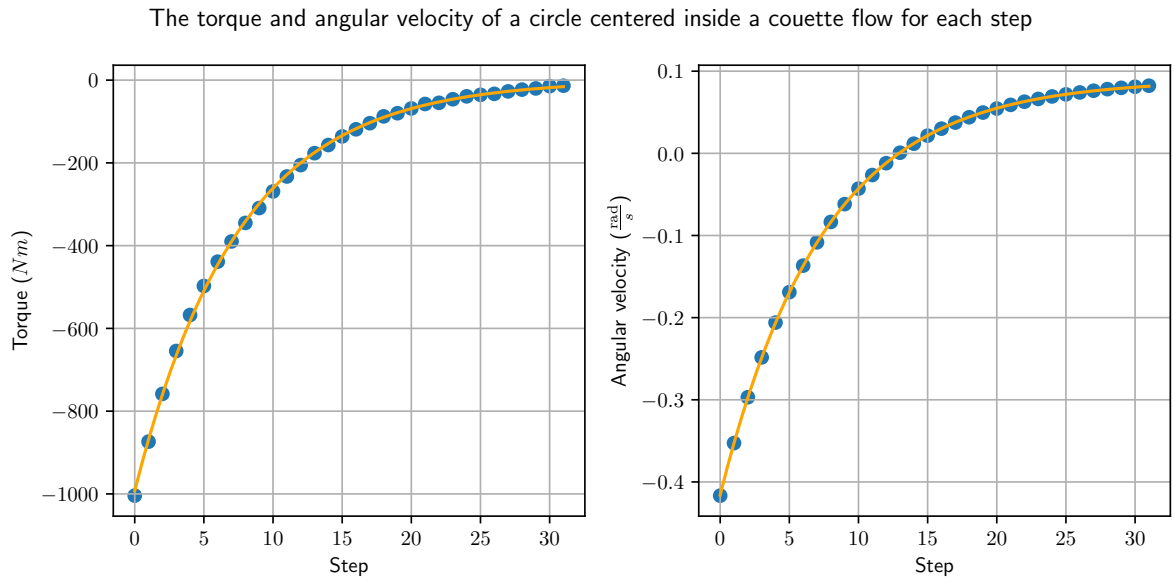


Figure 5.3: The progression of the torque and angular velocity with each step on a circle placed in the center of a Couette flow configuration. The approximating function for each is drawn in orange.

done on the details regarding transformation kernels from Subsection 4.6. Additionally, care needs to be taken to integrate the code's main loop with a GPU implementation, which at the moment is greatly dependent on the single-threaded nature of the CPU.

3. *Can time varying boundaries be introduced to the program while maintaining performance and accuracy?*
   Time varying boundaries are, due to the implementation, inherently quite poor in terms of performance. However, as they are only applied once each step, this is negligible. A good accuracy is maintained simply by keeping track of an original desired velocity, preventing error build up through incremental updates.

4. *Can groups of boundary points be introduced that act as a moving object in the simulation while maintaining performance and accuracy?*
   An intricate system has been made to keep track of several objects in a configuration. Updating boundary points belonging to these objects is largely done on the GPU, ensuring that updates take as little time as possible. However, the accuracy may be further improved by extending the boundary point structure with an original coordinate, minimizing the floating point error build up as the solver progressess.

Section 6

Conclusion

In summary, in this thesis, several aspects of the solver developed by Stam [2], Gaeremynck [3] and van Zwieten [4] have been extended. With the introduction of Python bindings, working with the code has become much easier. Not only have previous options that were available through the command line been kept available, but many new additions have been made that give users the option to finely tune the solver to their liking. Already in this project alone the bindings have been used extensively and it is hopefully something that will become available to more people in the future. Furthermore, with the introduction of both time-varying boundaries it was proven possible to introduce (time) steps to a time-independent solver by finding intermittent solutions. Continuing this development of multiple steps being executed, moving objects were introduced to the solver, which lead to the successful implementation of allowing them to rotate.

However, not all endeavours in this project were a success. The idea of a mega kernel, while in theory possible and highly performant, seems to have been shut down by the implementation specific details that cuFFTDx calls for. Additionally, there sadly was no time to fully complete the object movement, as translation is currently not implemented due to time constraints. Finally, specific to the code, some mess remains, and inevitably was introduced with the newly introduced features.

Nonetheless, the current state of the project should provide a solid basis for further development. The theory supporting object movement has already been worked out and should be decently easy to integrate into the current code. Furthermore, while the solver currently appears to find solutions in an acceptable time, there is always room for improvement in terms of low level optimizations as well as larger systems such as learning rate schemes and additional cost functions.

Section 7

Additional Work

In the wake of this project there are still improvements and additions. Most notably, adding a system to evaluate the pressure field with the purpose of adding translation, as described in Subsection 3.4, should be looked into. In theory, adding this to the existing code should be a relatively small amount of work.

Secondly, the non-uniform inverse discrete fourier transform (referred to as NUIDFT) as developed by Valentijn van Zwieten was chosen to be removed from the code. As stated in his paper the implementation had poor runtime due to implementation inefficiencies. In this project the NUIDFT was removed from the code base for the sake of readability. However, it might be a worthwhile undertaking to develop this idea further in future work.

Furthermore, looking into different learning rate schemes and cost functions might improve the solver's results. Configurations such as the four roller mill and lid driven cavity are known to make the solver converge very poorly. Moreover, the poor convergence when starting from a solution established by a previous step should be investigated. It was naively believed that starting from such a solution would increase performance. However, the opposite was found: When resetting the force field before finding a solution the solver converged both faster and further.

Finally, in the interest of performance, it remains interesting to study the mega kernel approach. Though it is not possible to have one big kernel, it is most likely a good idea to introduce a bigger split between CPU and GPU code. Despite efforts to split the two into separate functions, some mixing could not be avoided. This marginally hampers both performance as well as readability of the code. Both of which could give rise to issues with future endeavors.

Section A

Program Flow

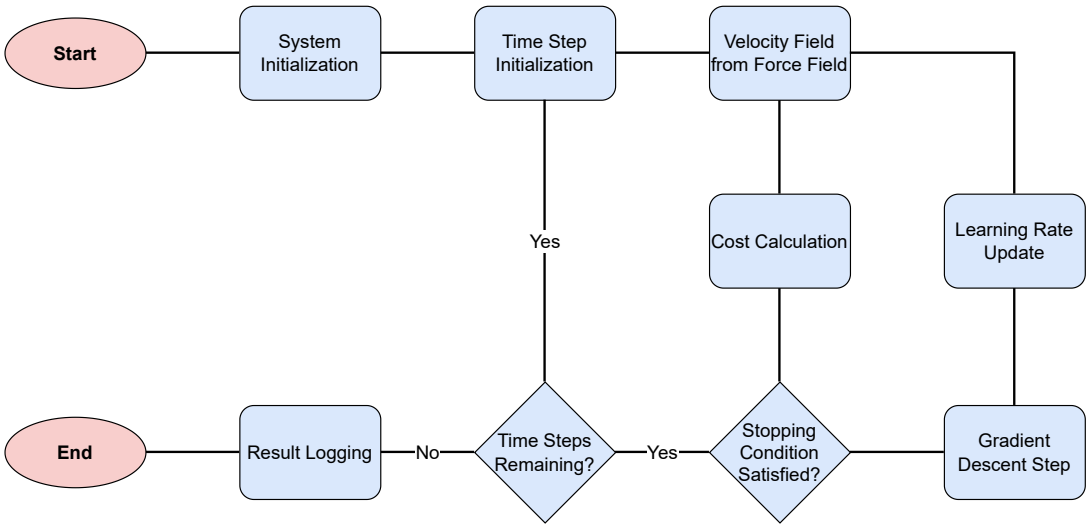


Figure A.1: The solver's flow chart. The red ellipses indicate start and end points. Blue boxes represent actions and blue diamonds indicate conditional branching of the code.

Section B

Python Functions

Function name	Parameters	Description
set_couette_flow	n/a	Sets a Couette flow configuration.
set_two_moving_plates	n/a	Sets the two moving plates configuration.
set_lid_driven_cavity	n/a	Sets a lid driven cavity configuration.
set_four_roller_mill	n/a	Sets a four roller mill configuration.
set_tilted_moving_plates	n/a	Sets the tilted moving plates configuration.
set_sinusoidal_plates	n/a	Sets the sinusoidal plates configuration.
set_pressure_pipe	n/a	Sets a pressured pipe configuration.

Table B.1: Functions that switch the solver's configuration to a preset.

Function name	Parameters	Description
set_boundary_point	x, y, z, u_x, u_y, u_z, idx	Creates a singular boundary point for object idx at point (x, y, z) with a velocity of (u_x, u_y, u_z).
set_boundary_line	$x1$, $y1$, $z1$, $x2$, $y2$, $z2$, d, u, idx	Creates a line of d boundary points for object idx from $p1 = (x1, y1, z1)$ to $p2 = (x2, y2, z2)$. The velocity of the created points will be u in the direction of $p1$ to $p2$.
set_boundary_circle	x, y, z, r, u, d, idx	Creates a circle of d boundary points for object idx on layer z with (x, y) as its center and a radius r. The velocity of the created points will be u tangent to surface of the circle.
set_boundary_sphere	x, y, z, u_x, u_y, u_z, r, d_a, d_p, idx	Creates a sphere consisting of $d_a d_p + 2$ boundary points for object idx centered at (x, y, z) with an angular velocity of (u_x, u_y, u_z) and radius r.
set_boundary_ellipse	x, y, z, u_x, u_y, u_z, a, b, ω, d, idx	Creates an ellipse of d boundary points for object idx with angular velocity $(u_x.u_y, u_z)$ on layer z with (x, y) as its center and a and b as its major and minor axis respectively.
set_boundary_super_ellipse	x, y, z, u_x, u_y, u_z, a, b, n, ω, d, idx	Creates a superellipse, with power n, of d boundary points for object idx with angular velocity $(u_x.u_y, u_z)$ on layer z with (x, y) as its center and a and b as its major and minor axis respectively.
set_solid	x, y, z	Sets the grid cell at (x, y, z) to solid.
set_force	x, y, z, f_x, f_y, f_z	Sets the force at coordinates (x, y, z) to (f_x, f_y, f_z).
add_object	$\mathcal{I}_o$, ω_x, ω_y, ω_z	Adds an object to the simulation and returns its index. The function $\mathcal{I}_o$ should, given an x, y and z coordinate of a grid cell, return whether it is inside of the object. The newly made object's initial angular velocity will be set to $(\omega_x, \omega_y, \omega_z)$.
clear_configuration	n/a	Clears the configuration, this entails the following: Clearing all boundary points, all solids, all virtual forces and all objects. Furthermore, the force field is reset to all zeroes.
clear_solution	n/a	Resets the force field to all zeroes.

Table B.2: Functions and their parameters that allow users to make their own configurations.

Function name	Parameters	Description
set_virtual_force_culling	v, r	Disables or enables virtual force culling based on v. When it is enabled, virtual force culling will remove virtual forces that are at a distance of r or more from any boundary points in the configuration.
run_verbose	n/a	Runs the solver in verbose mode, making it print the learning rate and cost at each iteration.
add_stop_conditions	S, P_S	Enables all stopping conditions specified in S. Furthermore, stop condition parameters can be set in P_S.
remove_stop_conditions	S	Disables all stopping conditions specified in S.
set_learning_method	m, P_m	Switches to the learning rate scheme specified by m. Furthermore, learning rate parameters can be set in P_m.
set_time_varying_boundary	$\mathcal{F}$	Sets the system's time varying boundary function to $\mathcal{F}$. The function f should, when given an x, y and z of a boundary point together with a time t, return a scalar by which to scale the original desired velocity of that point.
remove_time_varying_boundary	n/a	Clears the system's time varying boundary function.

Table B.3: Functions and their parameters that set solver variables.

Function name	Parameters	Description
approximate_virtual_forces	s, i_{max}, l	Runs a subsystem of scale s with an initial learning rate of l and i_{max} maximum iterations to approximate the virtual forces for the current configuration.
solve	i_{max}, l, t, ri, δt	Runs the solver. The solver will process t steps, reporting the velocity field every ri steps. Every step, with a size of δt, the learning rate will reset to l and the solver will attempt to find a solution in at most i_{max} iterations.
write_to_file	f	Outputs the current velocity field to the file specified by f.
free	n/a	Frees the memory taken up by the solver, effectively destroying it.
display_stokes	s, z	Displays the velocity field of solver s at a depth z.

Table B.4: Functions and their parameters that may be run after a configuration is set up or solved.

Bibliography

[1] Sohail Nadeem et al. “Numerical computations of blood flow through stenosed arteries via CFD tool OpenFOAM”. In: *Alexandria Engineering Journal* 69 (2023), pp. 613–637. ISSN: 1110-0168. DOI: https://doi.org/10.1016/j.aej.2023.02.005. URL: https://www.sciencedirect.com/science/article/pii/S1110016823001035.

[2] Bart Stam. “A GPU-based versatile and efficient hydrodynamics code for scientific applications”. Utrecht University, 2021.

[3] Florian Gaeremynck. “Improved methods on GPU based versatile and efficient hydrodynamics code for scientific applications”. Utrecht University, Dec. 2022. URL: https://studenttheses.uu.nl/handle/20.500.12932/43492.

[4] Valentijn van Zwieten. “Bringing GPU Parallelization and Complex Boundaries to a Computational Fluid Dynamics Solver”. Utrecht University, Nov. 2023. URL: https://studenttheses.uu.nl/handle/20.500.12932/45624.

[5] Ulf D Schiller, Timm Krüger, and Oliver Henrich. “Mesoscopic modelling and simulation of soft matter”. In: *Soft matter* 14.1 (2018), pp. 9–26.

[6] Michael A. Day. “The no-slip condition of fluid dynamics”. Version 33. In: *Erkenntnis* (Nov. 1990), pp. 285–296. DOI: https://doi.org/10.1007/BF00717588.

[7] S. Behnel et al. “Cython: The Best of Both Worlds”. In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 31–39. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.118.

[8] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. 2017. URL: https://github.com/pybind/pybind11.

[9] Wenzel Jakob. *nanobind—Seamless operability between C++17 and Python*. 2022. URL: https://github.com/wjakob/nanobind.

[10] NVIDIA. *cuFFT, the CUDA Fast Fourier Transform library*. Version 12.5. May 2024. URL: https://docs.nvidia.com/cuda/cufft/index.html#.

[11] NVIDIA. *cuFFT Device Extensions (cuFFTDx)*. Version 1.1.1. URL: https://docs.nvidia.com/cuda/cufft/index.html#.

[12] Burkhard Dünweg and Anthony J. C. Ladd. “Lattice Boltzmann Simulations of Soft Matter Systems”. In: *Advanced Computer Simulation Approaches for Soft Matter Sciences III*. Ed. by Christian Holm and Kurt Kremer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 89–166. ISBN: 978-3-540-87706-6. DOI: 10.1007/978-3-540-87706-6_2. URL: https://doi.org/10.1007/978-3-540-87706-6_2.

[13] Boris T Polyak. “Introduction to optimization”. In: (1987).