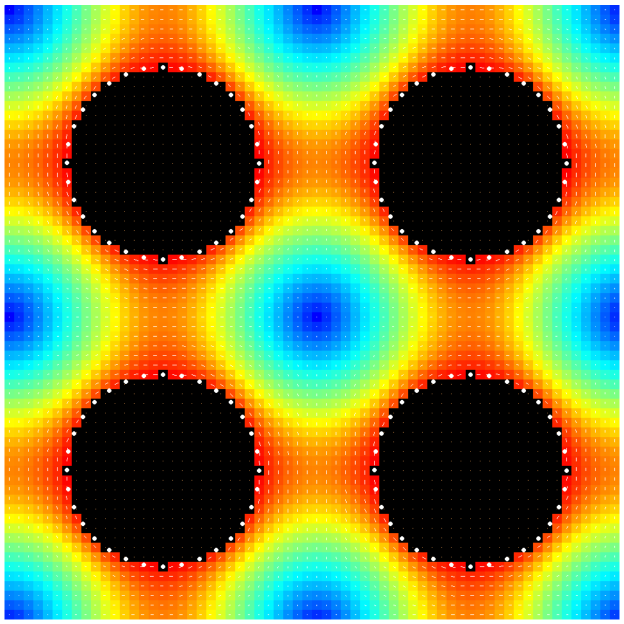
Bringing GPU Parallelization and Complex Boundaries to a Computational Fluid Dynamics Solver



MSc thesis in Game and Media Technology
for Utrecht University

Author
Valentijn van Zwieten
(6260691)

Supervisors
Dr. Deb Panja
Dr. Joost de Graaf

November 30, 2023

Abstract

We present optimizations and flexibility improvements to an existing computational fluid dynamics solver based on fast Fourier transforms. The existing solver was implemented partly on the CPU and partly on the GPU, leading a large performance overhead due to context switches. This, among other performance issues, made it unfeasible for practical application. Additionally, the grid-based manner in which the boundaries for solids were implemented limited the range of problems it could accurately depict and solve. Moreover, the abundance of hard-coded features made it cumbersome to use. In this thesis, these problems are resolved as follows: Nearly the entire solver has been ported to the GPU, eliminating the performance overhead introduced by context switches. A host of additional runtime and memory optimizations are implemented into the program. The implementation of boundaries has been improved to handle arbitrary geometries. Finally, a basic command-line interface has been introduced to set a range of solver parameters when launching the program. In optimal situations the runtime is decreased by a factor exceeding 1000$\times$ and the memory footprint is reduced by up to a petabyte. In more general cases the runtime decrease is often in the triple digits and the footprint is reduced by many gigabytes. With the introduced improvements the way is paved for the solver to be a tool that is used by researchers in the field of fluid dynamics.

Cover figure: Visual output of the solver. Four cylinders, indicated by the black cells, are submerged in a fluid. The cylinders rotate around their axis which creates a flow. How fast the fluid flows is indicated by a color spectrum, with blue being slowest and red being fastest.

Contents

Summary of Notations

Notation	Definition
N, M, L	Width, height, and depth of system, respectively
$\vec{U}_a$	Velocity field (on axis a)
$\vec{F}_b$	Force field (on axis b)
$\widetilde{U}_a, \widetilde{F}_b$	Fourier Transformed velocity and force fields
B	Set of boundary points
V	Set of virtual forces
C	Cost function
$G_{a,b}$	$\frac{\partial U}{\partial F}$ (on combination of axis a, b)
$\vec{U}_{a,b}^{(1)}$	Unit force velocity field (on combination of axis a, b)
γ_t	Learning rate at iteration t

The exact definition of the above quantities is given in the upcoming sections.

1 Introduction

In the field of fluid dynamics, research is often carried out using a tool known as a "solver". This refers to a program that is able to solve for the flow of a fluid given a certain configuration. These configurations may, for example, contain solids around which the fluid is to flow or external forces that pull or push the fluid in a certain direction. Using this, researchers may model a variety of scenarios such as blood flowing through an artery or magma flowing through the earth. This can aid research into these topics and lead to new insights that can be applied in the real world.

In this thesis, we build our work on the research carried out by Bart Stam[6] and Florian Gaeremynck[4]. They created a novel solver that is capable of finding a solution to the Stokes equation in systems containing incompressible fluids, solids, and point forces. This solver is intended to be used in a scientific-research setting in upcoming projects in the groups of Dr. Deb Panja and Dr. Joost de Graaf. To be suitable for use in the field, further development is required, specifically on the fronts of speed and flexibility. A brief overview of the existing solver's methodology and the corresponding physics is provided below to give background to our research questions. An in-depth explanation will be provided in Sec. 2.

1.1 Physics Background

The Navier-Stokes equations are central to the field computational fluid dynamics. When describing incompressible fluids these are as follows:

$$\nabla \cdot \vec{u} = \vec{0}; \tag{1}$$

$$\underbrace{-\nabla \vec{p} + \vec{f} + \mu \nabla^2 \vec{u}}_{\propto \text{ F}} = \underbrace{\rho}_{\propto \text{ m}} \underbrace{\left(\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} \right)}_{\propto \text{ a}}, \tag{2}$$

where:

- $\vec{u}$ is the velocity of the fluid,
- $\vec{f}$ is the external force acting on the fluid,
- $\vec{p}$ is the pressure in the fluid,
- μ is the viscosity of the fluid,
- ρ is the density of the fluid, and
- t is time.

Equation 1 describes the incompressibility by stating that there is no divergence in the flow. This implies that for any volume in the fluid the amount of fluid flowing in is exactly the same as the amount of fluid flowing out. As such, there

Figure 1: A visual comparison between non-Stokes flow (left [1]) and Stokes flow (right [7]). The surface of the ocean near a beach is usually highly turbulent while magma has no turbulence whatsoever.

is no point where the fluid is amassing, which would in turn imply compression. Equation 2 is an application of Newton's second law of motion, $F = ma$ (force is mass times acceleration), describing the flow. Solving the Navier-Stokes equations is not a simple task, due to the non-linear partial differential in the acceleration term of Eq. 2. Usually analytical approaches are not possible and and iterative methods are required, such as those shown in Ref. [3].

Under specific circumstances, however, the acceleration term becomes increasingly less significant when calculating the flow, such that it may be ignored on physical grounds. This phenomenon occurs when the inertial forces in the fluid are substantially smaller than the viscous ones. As a consequence non-linearities, such as turbulence, do not occur. Fluid flow under these conditions is known as Stokes flow. An intuition for the difference between non-linear Navier-Stokes flow and linear Stokes flow is provided in Fig. 1. A more mathematical definition of when fluid flow is considered a Stokes flow can be giving using what is known as the fluid's Reynolds (Re) and Strouhal (St) numbers. These are defined as:

$$Re = \frac{\rho U L}{\mu}; \tag{3}$$

$$St = \frac{L}{UT}, \tag{4}$$

where L is a characteristic length – a dimension that defines the scale of the system – T is a characteristic time (of an external effect), and U is a characteristic velocity.

Re and St are both dimensionless. The Reynolds number describes the ratio between inertial and viscous forces. At high Reynolds numbers, the inertial forces dominate and the flow becomes turbulent, while at low Reynolds numbers, the viscous forces dominate and the flow becomes "laminar". The Strouhal number describes the ratio between inertial forces due to localized acceleration and inertial forces due to global acceleration. At high Strouhal numbers, there is a competition between the time it takes for the fluid to move and the time at which it is driven, while at low Strouhal numbers, the fluid conforms to the driving (nearly) instantaneously. Stokes flow occurs when $Re \ll 1$ and $ReSt \ll 1$. It is described by the linear Stokes equations, which are defined as follows:

$$\nabla \cdot \vec{u} = \vec{0}; \tag{5}$$

$$\underbrace{-\nabla \vec{p} + \vec{f} + \mu \nabla^2 \vec{u}}_{\propto \text{ F}} = \underbrace{\vec{0}}_{\propto \text{ ma}}. \tag{6}$$

Here, the acceleration term (and thus the mass term) are no longer present.

From the linear, time-independent Stokes equations (LTIS), we can much more easily obtain a velocity field given a force field. This is because what remains to be solved is a set of linear partial differential equations. Solving these is still not a simple task, however. In essence we need to get the derivative of a very intricate function. The method by which operations on intricate functions are usually done, involves breaking the function up into separate parts that are individually simple and orthogonal to each other. When this is the case we can apply our operation of choice to all the simple parts instead, after which we can re-combine them.

For this a Fourier transform is used. This linear transformation brings a function into the frequency domain, where it is described by a set of sine and cosine waves. An example of this can be seen in Fig. 2. Given that the derivative of a sine and cosine is well-known, solving the partial differential is now elementary. Once performed, the result can be transformed back into the real domain using an inverse Fourier transform. During this process no accuracy is lost and we obtain the exact solution.

With LTIS we can obtain a velocity field given a force field. However, it does not account for the presence of solid objects (which we will now refer to as "solids"). When these exist in the system we have to accommodate for this through other means. The key lies in what is known as the no-slip boundary condition, which states that the velocity of a fluid on the boundary of a solid must be $\vec{0}$ relative to that of the solid's surface[2]. As such, when introducing solids into the system, we introduce a constraint on the velocity field. This constraint is implemented in an "immersed" way. That is, the parts of the system that are considered solid are still modeled as a fluid by the solver. However, in these parts, the force field is set in such a way that the resulting flow satisfies the no-slip boundary condition. Since the solution to the LTIS is unique, a system in which we satisfy the no-slip boundary condition is identical to the "real" solution, where fluid does not flow inside the solid. With this we know how our Solver should go about finding its solution.

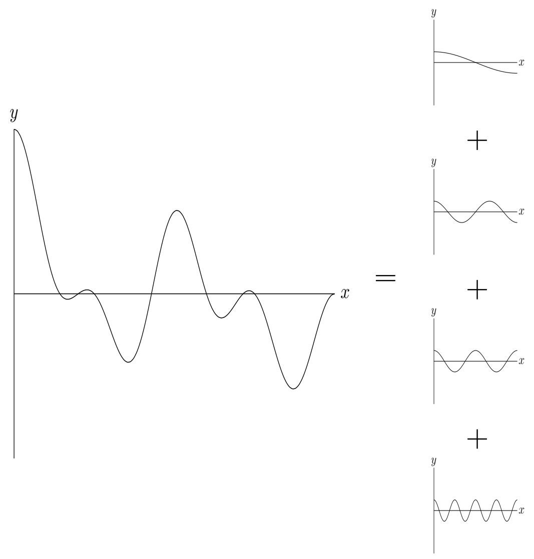


Figure 2: An intricate function (left) and its simple, orthogonal components in the frequency domain (right). The frequency components that make up any arbitrary function can be found using the Fourier transform.

1.2 Solver Overview

The solver is build to solve for Stokes flow on a periodic discrete grid. As mentioned in Sec. 1.1, to solve a system we use forces inside solids that direct the fluid flow in such a way that the no-slip boundary condition is satisfied. We call these "virtual forces", as they are not part of the configuration as set by the researcher, but rather placed by the solver itself to find the solution. Finding the correct values for these virtual forces such that we obtain our solution is a complex problem, given that every virtual force influences every cell on the velocity field. To accomplish this we use a technique called gradient descent. With this we minimize the distance to the solution by iteratively tweaking the virtual forces and observing how close the resulting velocity field is to satisfying the no-slip boundary condition. A schematic high-level overview of the solving process is provided in Fig. 3.

Before this research took place, the existing solver was already able to converge to an accurate solution for various problems. However, there are several problems that prevent it from a tool that is useful in practice. The most major of these is its computational efficiency. Calculating a solution can take up to dozens of minutes in some cases, whereas other solvers on the market are able to perform this in seconds or less. The main culprit was identified to be expensive context switches between the CPU and GPU during execution. Fixing this demands for (close to) the entire algorithm to be ported to the GPU. Other areas of the solver that show potential for performance gains have been identified as well, as will be shown in the upcoming section. Besides this, the grid-based manner in which the boundaries for solids were implemented limited the range of problems it could accurately depict and solve. Additionally, the abundance of hard-coded features made it cumbersome to use. All these problems will be addressed in this research.

1.3 Research Questions

In this thesis project, we aim to address the following question: *How can we optimize and improve flexibility of a computational fluid dynamics solver based on fast Fourier transforms?* Starting with the solver as created by Bart Stam[6] and Florian Gaeremynck[4] we identify these subproblems:

1. The solver converges on a solution iteratively. Currently, every one of these iteration is performed partly on the CPU and partly on the GPU. These context switches are computationally expensive. *Can we avoid or limit expensive context switches during the iterative section of the solving process?*

2. Every iteration of the solver is performed partly in regular space and partly in Fourier space. These transformations are computationally expensive. *Can we avoid, limit, or using domain-specific tricks, optimize expensive (inverse) discrete Fourier transforms?*

3. The solver only requires the velocities on solid surfaces in the system to converge to a solution. Some operations performed by the solver operate on all cells in the system's grid. *Can we perform the solving process exclusively for cells on a solid surface?*

4. The solver's degrees of freedom along which it finds a solution are the virtual forces. Less degrees of freedom means less computations required to find their correct values. *Can we limit the amount of virtual forces in a system while still converging to an acceptable solution?*

5. The virtual forces are initiated as zero vectors. Starting with a more representative value, even if it is only an estimation, would help our gradient descent converge faster. *Is there a way to determine an approximate initialization value for virtual forces?*

6. Memory usage is a major bottleneck of the solver. *How can we handle the data required by the solver most efficiently?*

7. Solids currently reside on the system's grid. To increase the solver's accuracy, arbitrary boundaries of solids are desirable. *Can we introduce arbitrary solid boundaries in the solver?*

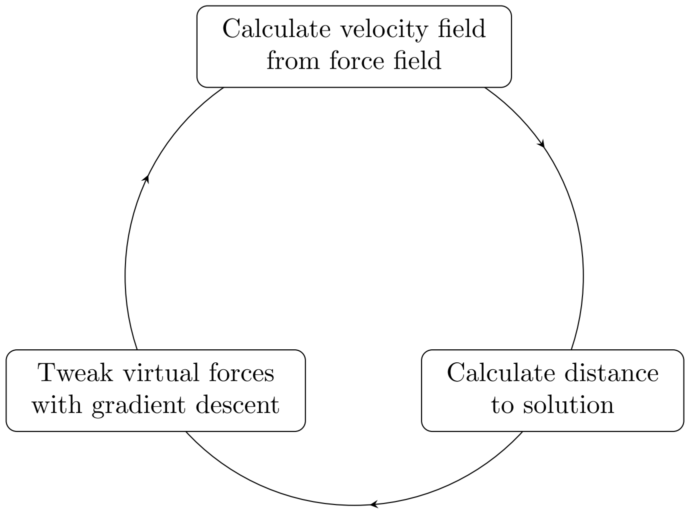

Calculate velocity field from force field
Calculate distance to solution
Tweak virtual forces with gradient descent

Figure 3: A high-level overview of the solving process, consisting of three steps. These steps are repeated until the desired solution is found.

2 Methodology

2.1 Velocities and Forces

The solver works on a periodic discrete N by M by L grid. On this grid exist a velocity field $\vec{U}$ and force field $\vec{F}$. By default there exists no fluid flow and thus all cells in $\vec{U}$ and $\vec{F}$ are $\vec{0}$. When a cell in $\vec{F}$ is set as non-zero, flow is created. To calculate the velocity field corresponding to a force field, we use a discrete Fourier space form of LTIS (Eq. 5 and 6). It is as follows:

$$\widetilde{U} = \frac{1}{\mu k^2}(I_3 - \widetilde{k} \otimes \widetilde{k})\widetilde{F}. \tag{7}$$

For the full derivations leading to this equation, we refer the user to Stam's work[6]. We calculate the Fourier-space equivalent of the real force field and real equivalent of the Fourier-space velocity field using the discrete Fourier transform and its inverse counterpart, respectively:

$$\begin{aligned} \widetilde{f}[q,r,s] &= \sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\sum_{l=0}^{L-1} \exp\left(2\pi\left(\frac{nq}{N} + \frac{mr}{M} + \frac{ls}{L}\right)\right) f[n,m,l], \\ \widetilde{f}[n,m,l] &= \sum_{q=0}^{Q-1}\sum_{r=0}^{R-1}\sum_{s=0}^{S-1} \exp\left(-2\pi\left(\frac{nq}{Q} + \frac{mr}{R} + \frac{ls}{S}\right)\right) f[q,r,s]. \end{aligned} \tag{8}$$

The full equations as given here are for reference purposes only. In reality fast (inverse) Fourier transforms are utilized, as mentioned in Sec. 3.1.

2.2 Boundaries

Solids in the system are enclosed by a boundary surface. This surface is made up of a set of points in 3D space. These "boundary points" are not restricted to being on the solver's grid. The boundary points themselves cannot move around in the system, but can simulate movement on the surface they approximate. Each boundary point has a desired velocity at which the surface is intended to move at that point. This can be envisioned as a conveyor belt, where only the surface of the object is in motion. A stationary boundary point has a desired velocity of $\vec{0}$. We denote the set of all boundary points as B. An example of the boundary system can be seen in Fig. 4.

Solving the system means satisfying the no-slip boundary condition on the surface of solids. This implies that for every boundary point, the difference between their desired velocity and the actual velocity in $\vec{U}$ at that point is $\vec{0}$. As such boundary points can be put at any point where $\vec{U}$ is desired to have a specific value, as this is what the solver will try to converge to. To check whether we satisfy the desired velocities in a system, we have to know the current velocity of each boundary point. When a boundary point exists in between cells of the discrete velocity grid the velocity at its coordinates is linearly interpolated. Boundary points are not connected to each other in any way; if there is too large of a gap between a pair of boundary points a fluid can flow "through" an intended surface in the solution. The remedy is to add more boundary points to better approximate this surface. When boundary points are spaced close together multiple boundary points interpolate their velocity from the same cells. On especially dense surfaces or low resolution grids this may make it difficult for the solver to converge, as the velocity in those cells has to satisfy the no-slip boundary condition of many boundary points.

In the original system by Stam and Florian, boundaries were defined as all the points between pairs of neighbouring fluid and solid cells. As such the boundaries were made up of exclusively discrete points. These "boundary pairs" had a desired velocity as with the new boundary points. To calculate the current fluid velocity on a boundary pair the average velocity of the two cells is taken. An example of the original boundary system can also be seen in Fig. 4.

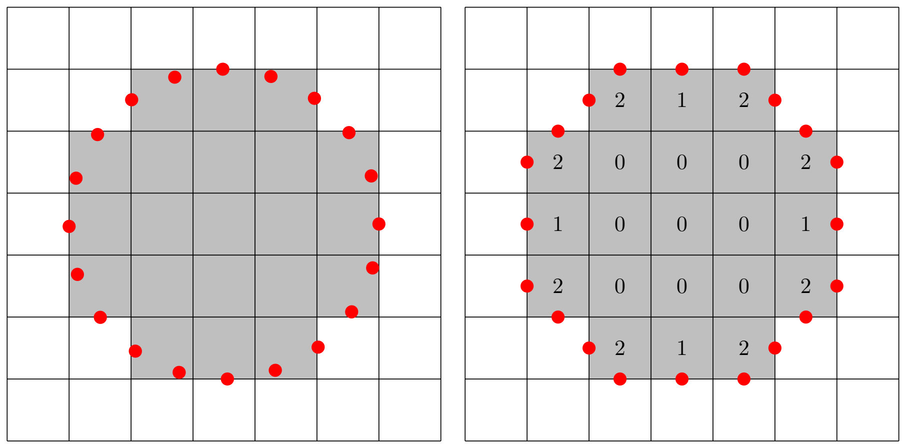


Figure 4: 2D examples of the current (left) and original (right) boundary systems when creating a boundary around a circular solid (the grey cells). In the current boundary system a boundary is defined as a point in space as visualized by the red dots. Note that this point does not have to be on the grid. In the original boundary system a boundary is defined as a pair of neighbouring fluid and solid cells. The boundary is localised in between these as visualized by the red dots. Note that one solid cell may be part of multiple boundary pairs. The number of boundary pairs a given solid belongs to is displayed in each solid cell.

2.3 Solids and Virtual Forces

To satisfy the no-slip boundary condition, we somehow have to direct the fluid flow in such a way that the velocity at every boundary point is equal to that boundary point's desired velocity. LTIS (Eq. 5 and 6) tells us that the only way to achieve this is by tweaking the force field. This is done using virtual forces; forces that are not part of the configuration but placed by the solver itself. For the solution to be correct these must be placed carefully. A cell on the solver's grid is either inside a solid or fluid. In our solution, only the velocities of the fluid are relevant. When a non-zero virtual force exists inside a fluid, we introduce foreign elements into the fluid which leads to an incorrect solution. When a virtual force exists inside a solid this is not the case. As such cells inside a solid are suitable points for our virtual forces. We denote the set of all virtual forces as V. An example of virtual forces in action can be seen in Fig. 5.

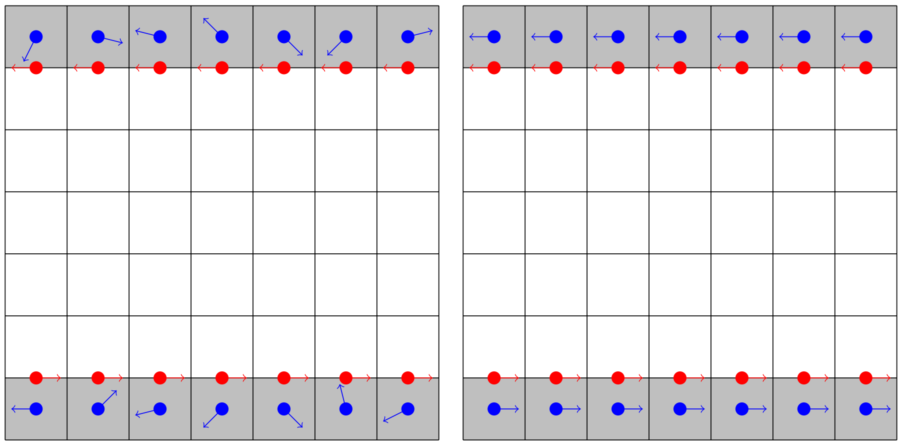


Figure 5: 2D examples of a bad (left) and good (right) choice for virtual force values (the blue dots). Arrows protruding from the boundary points and virtual forces indicate the direction of the desired velocities and forces, respectively. Their magnitudes are ignored in these examples. In the bad example the direction of the virtual force would not lead to a flow that matches the desired velocity in the boundary points, meaning the no-slip boundary condition is not satisfied. In the good example this would be the case.

2.4 Cost and Gradient Descent

The distance between any arbitrary situation and the solution can be expressed through a cost function:

$$C = \sqrt{\sum_{b \in B} (\vec{u}_c(b) - \vec{u}_d(b))^2}, \tag{9}$$

where $u_c(b)$ is the current velocity at boundary point b and $u_d(b)$ is the desired velocity at boundary point b. As is custom for cost functions, the closer C is to 0 the closer the current situation is to the solution.

To set the virtual forces in such a way that $C = 0$ we must know how a change in any given virtual forces influences the velocity of any given boundary point. The gradient that describes this is defined as follows:

$$G_{a,b}[i, j] = \frac{\partial U_a[i]}{\partial F_b[j]}, \tag{10}$$

where:

- G is the gradient,
- i is the index of the velocity cell,
- j is the index of the virtual force cell,
- a is the axis of the velocity, and
- b is the axis of the force.

The vector G has a total of $|B||V|$ entries. Given that the axis of the velocity and the axis of the force can vary independently there are a total of $3^2 = 9$ of such gradients. The methodology used to calculate how a force influences the velocity field can be seen in Sec. 2.1.

To find our solution the Gradient Descent algorithm is used. In Gradient Descent, a function is minimized by following its gradient in the opposite direction in a step-wise manner. An intuitive way of thinking about this is walking downhill across the surface of a function in the hopes of finding the lowest point. The function that is being minimized is C. While one would be quick to assume the gradient that used to determine in which direction a step is taken is given by G, this is not quite the case. The gradient that we concern ourselves with in the context of gradient descent, is one that describes the change in C with regards to a change in virtual force:

$$\nabla V = \left[\tfrac{\partial C}{\partial V[j]} \right], \tag{11}$$

where j is the index of a virtual force.

Every iteration t, a step in direction $-\hat{V}$ of size γ_t is taken. Choosing the value of γ_t, also known as the learning rate, is a tricky task and is highly dependent on the properties of the problem. If the learning rate is set too small, it can take a large number of iterations to reach the minima. If it is set too large, the solver might "overshoot" the minima by stepping over it. For most problems where gradient descent is used, local minima must also be accounted for. Here, the algorithm might get stuck hovering around a local minima and not converge to the sought after global minima. Tricks such as restarting at random points on the function and special learning rate schemes can be used to combat this. For us this is not a problem however, as C is convex, meaning that there is only one minima. This does not mean our gradient is simple to traverse and we still have to take care when selecting a learning rate scheme to converge (fully).

The following learning rate schemes are available in the solver:

- Constant
  The learning rate is kept at a set constant value.
$$\gamma_t = \gamma_{t-1} \tag{12}$$
- Halving
  The learning rate is initialized at a set value. Whenever the cost does not decrease when compared to the previous step, the learning rate is halved.
$$\gamma_t = \begin{cases} \gamma_{t-1} & \text{if } C_t < C_{t-1} \\ \dfrac{\gamma_{t-1}}{2} & \text{if } C_t \geqslant C_{t-1} \end{cases} \tag{13}$$
- Polyak's Length
  Polyak's length is designed specifically for convex optimization problems and should theoretically always find the global minimum[5]. It is defined as follows:
$$\gamma_t = \frac{C_t}{|\nabla V|^2}. \tag{14}$$
- Dynamic Polyak's Length
  An adaptation of Polyak's Length by Florian Gaeremynck, this scheme takes advantage of the observation that in our use cases Polyak's length spends the vast majority of its time converging the last few percentiles of the solution. To remedy this attempts to take bigger steps in consistently decreasing areas of the function are made. Polyak's Length is calculated, after which a multiplying factor is applied[4]. It is defined as follows:
$$\gamma_t = \frac{C_t}{|\nabla V|^2} * m_t, \tag{15}$$
$$m_t = \begin{cases} 1 & \text{if } t = 0 \\ m_t * 10 & \text{if } C_t < C_{t-1} \text{for 100 consecutive } t \\ m_t * 0.1 & \text{if } C_t \geqslant C_{t-1} \text{for 2 consecutive } t \end{cases} \tag{16}$$

3 Implementation

The program flow of the solver can be broken up into 7 distinct parts. These parts and their relation to each other can be seen in Appendix A. Originally only the initialization of $\frac{\partial U}{\partial F}$ and transforming the force field into the velocity field was performed partly on the GPU. Now every step of the main loop, save for updating the learning rate, takes place on the GPU. This removes the necessity for copying large buffers to and from the GPU in every iteration. In this section, each of the parts of the solver will be discussed in further detail. First an overview of the tools and libraries utilized by the program will be given.

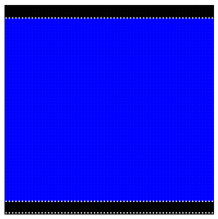
3.1 Tools and Libraries

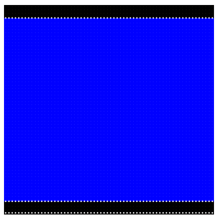
The majority of existing code on which this project is build upon is written in C++. This will remain the programming language of choice for new additions. C++ allows for a more low-level control of the system which is beneficial for runtime optimization. Existing GPGPU code is written using NVIDIA's Compute Unified Device Architecture (CUDA) language. This will similarly be adopted. NVIDIA GPU's are commonly used in the scientific community which makes the language a natural choice. Several libraries are used in various parts of the program. These are:

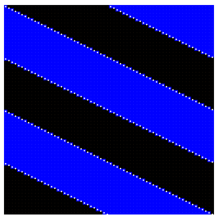
- *argparse* for better handling of launch parameters (see Sec. 3.2). This library handles the parsing of complex launch parameters through features such as optional and positional arguments. It also automatically generates a help command that explains how to use the program.
- *fftw* and *cuFFT* for the execution of Fourier transforms (see Sec. 3.4). These libraries perform Fast Fourier Transforms (FFT) on the CPU and GPU respectively. A FFT is multiple orders of magnitude faster than a "regular" Fourier transform and yields the same results.
- *CImg* for the rendering of the final flow field (see Sec. 3.9). This library supplies a pixel buffer which can be drawn to in various ways such as with elementary shapes. This buffer can then be displayed in a window.
- *CUB* to facilitate kernel programming. This library provides reusable kernel code for common operations done in GPGPU programming.

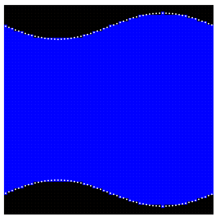
3.2 Launch

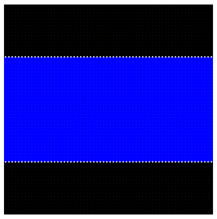
The solver allows for several parameters to be set by the user on launch. This can be done when the program is launched from the command line. The supported launch parameters and commands can be seen in Tab. 1.

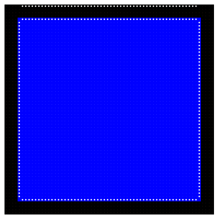

(a) 2 Moving Plates (2mp)


(b) Couette Flow (cou)


(c) Tilted Moving Plates (tmp)


(d) Sinusoidal Plates (sp)


(e) Pressure Pipe (pp)


(f) Lid Driven Cavity (ldc)

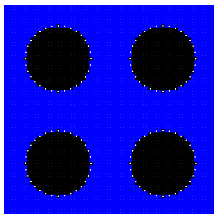

(g) 4 Roller Mill (4rm)

Figure 6: The configurations present in the solver, depicted on a $64 \times 64 \times 1$ grid.

The user can select one of several configurations. These are displayed in Fig. 6. Among them are several well-known fluid dynamics benchmark geometries. These include the following:

- Couette Flow (cou)
  Two xz-plane aligned plates with the upper plate moving tangentially to the lower plate enclose a fluid.

- Pressure Pipe (pp)
  Two xz-plane aligned still plates enclose a fluid. A force field creates a flow tangentially to the plates.

- Lid Driven Cavity (ldc)
  Plates enclose a cube of a fluid. The upper plate moves tangentially to the lower plate.

- 4 Roller Mill (4rm)
  Four cylindrical “rollers” are situated in the middle of each quadrant of the xy-plane. The upper right and lower left rollers rotate clockwise while the upper left and bottom right rollers rotate counter-clockwise.

Short Form	Long Form	Description	Default Value
-h	–help	Displays all the available launch parameters and exits	*n/a*
-v	–version	Displays versioning information and exits	*n/a*
-c	–config	Sets simulation configuration	tmp
-d	–dims	Sets simulation dimensions	32 32 1
-l	–learning-rate	Sets initial learning rate	0.05
-i	–iterations	Sets maximum iterations	10000
-o	–output	Sets output path	LastSimulationOutput.txt
-s	–show	Enables result rendering	*n/a*

Table 1: The launch parameters and commands supported by the program.

In addition, we provide several more geometrically interesting configurations. These are as follows:

- 2 Moving Plates (2mp)
  Two xz-plane plates moving tangentially in opposite direction enclose a fluid.
- Tilted Moving Plates (tmp)
  Two xz-plane plates tilted around the z-axis moving tangentially in opposite direction enclose a fluid.
- Sinusoidal Plates (sp)
  Two waves on the xz-plane mirrored on the x-axis with their surfaces moving tangentially in the same direction enclose a fluid.

3.3 System Initialization

With the desired parameters and configuration set by the user, the solver initializes everything it needs to enter the main solving loop. Among others, buffers for the real- and Fourier-space velocity and force fields are created. While virtual forces behave differently than "regular" forces, there is no separate buffer for virtual forces. We track which cells on the grid contain virtual forces and which cells in $\vec{F}$ thus can be manipulated. For all cells that do not contain virtual forces the corresponding force in $\vec{F}$ is part of the user-defined configuration and thus must be left alone. Additionally "plans" for our Fourier transforms are made (see Sec. 3.4) and the CUDA device is selected.

By default, a virtual force is placed on each cell marked as solid (see Sec. 2.3). In an effort to reduce the amount of virtual forces, those that we believe to have a limited impact on any of the boundary points can be omitted. This reduces the computations required in the gradient-descent step, as well as the size of the $\frac{\partial U}{\partial F}$ gradient (see Sec. 2.4). Virtual forces are omitted by the simple metric of whether they are at least a given distance from any boundary point.

To give the virtual forces an initial starting value, a scaled-down subsystem may be run. By solving a smaller version of the current system we get a force field which is similar to but not identical to the force field that our original system will produce. We can sample this smaller force field for starting values for our virtual forces. If the amount of processing time lost to running the subsystem is less than the time gained from our better initial starting values we have a net run-time improvement.

3.4 Force Field to Velocity Field

When using Eq. 7, all of $\vec{F}$ is taken and transformed into $\vec{U}$. Since $\vec{F}$ is a combination of user-defined forces and virtual forces, both of which must be preserved, we cannot get around the fact that we have to do a transform over the full force field when going into Fourier space. However, when calculating C we only need to know the current fluid velocity in cells from which a boundary point interpolates its own. This is illustrated in Fig. 7. As such, we only have to do a transform over a select set of cells when going back into real space. This allows us to perform a *Non-Uniform* Inverse Discrete Fourier Transform (NUIDFT). Disregarding part of the input theoretically reflects itself in a performance increase, as less data has to be processed.

The NUIDFT is implemented in matrix form. When running the solver on the CPU this matrix will be generated on initialization. On the GPU the matrix is calculated on the fly due to the performance impact of retrieving it from global memory. Each row of the matrix corresponds to a cell which is present in at least one boundary point, those being the only cells that need to be transformed back to real space for the solver to run. The contents of the rows are such that when multiplying the row vector with the (flattened) complex velocity field, we obtain the real velocity in the corresponding cell (see Sec. 2.1):

$$NUIDFT[i,j] = \exp\left(2\pi\left(\frac{c_x j_x}{N} + \frac{c_y j_y}{M} + \frac{c_z j_z}{L}\right)\right), \tag{17}$$

where c is the cell corresponding to row i. This implies the NUIDFT matrix is of size $|B_{cells}| \times NML$. However, due to the Hermitian symmetry $\vec{U}_{k_i} = \vec{U}_{k_{|\vec{U}_k|-i}}$ present in the Fourier-transformed velocity field, half of the values are redundant. In fact, *fftw* and *cuFFT* do not calculate these values at all when performing a Fourier transform. This means that the size of the Fourier space velocity and force buffers are $(\frac{N}{2}+1)ML$. In the NUIDFT matrix this is accounted for by cutting all columns in the reflected part and multiplying all columns that correspond to a cell in the reflected part by a factor of 2. The NUIDFT matrix is thus of size $|B_{cells}| \times (\frac{N}{2}+1)ML$.

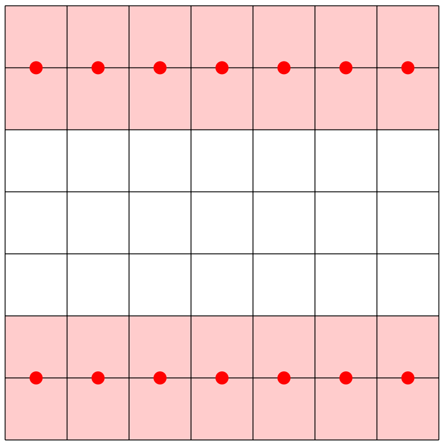


Figure 7: 2D example indicating which cells we need to know the velocity of to calculate C (the light red cells). The velocity of the remaining cells is only relevant once the final solution is calculated (after the final virtual force values have been found).

3.5 Cost Calculation

Calculating C is a simple square root over the sum of the squared difference between each boundary point's current velocity and its desired velocity. Obtaining the current velocity requires interpolation over $\vec{U}$. Since we also need the current velocity in the gradient descent step (Sec. 3.7) this value is cached.

3.6 Stopping Conditions

The user specifies how many iterations the solver is allowed to run when launching the program. Usually running this exact amount of iterations is not the goal when attempting to solve a configuration. Instead an attempt is made to reach a point where the current solution is "sufficiently" converged. What is sufficient depends on the use case. When this point is reached the solver gets to stop early and skip the remaining iterations. The solver provides two metrics to measure how far the current solution has converged; the total percentile reduction in C and the maximum difference between any boundary point's desired and actual velocity. For example, one can opt to stop iterating once C has been reduced by 99.99% or the velocity error of each boundary points is below 0.01. Another stopping condition exists which is based on the amount of iterations that have been performed since a decrease in C has occurred. As this number grows larger it becomes more likely that the system will not be able to converge any further and has gotten "stuck". For example, one can opt to stop iterating once there has not been a decrease in C for 500 iterations straight. Finally, the system always stops when $\gamma_t < 10^{-7}$. At this point the learning rate has shrunk below the margin of error for a floating point and thus no more meaningful progress can

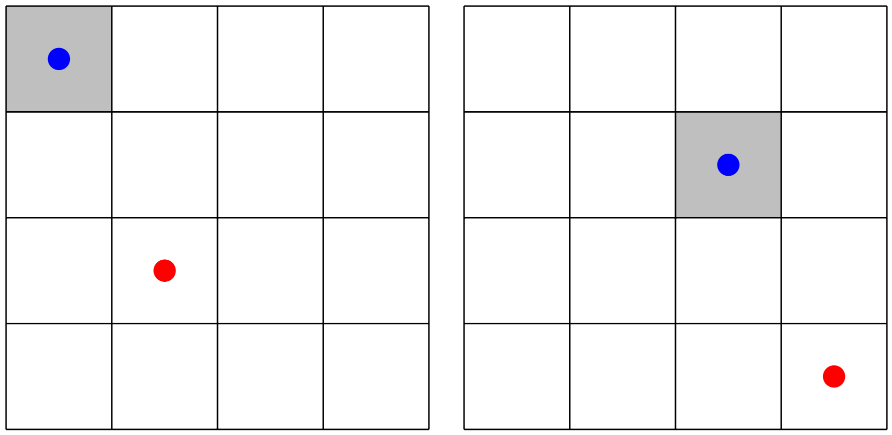


Figure 8: 2D examples of two virtual force - boundary point pairs with different absolute coordinates but identical relative coordinates, meaning that $\frac{\partial U}{\partial F}$ is identical as well.

be made. Of the stopping conditions mentioned, only the percentile reduction was present in the old solver.

3.7 Gradient Descent Step

To perform a gradient descent step we first need a way to access G. As described in Sec. 3.4, calculating $\frac{\partial U}{\partial F}$ requires a Fourier transform of the entire real force field and at least a partial inverse Fourier transform of the complex velocity field. At first glance this is a process that must be performed for every entry in G; from every virtual force to every boundary point on every combination of axes. This would lead to a total of $9|B||V|$ Fourier transforms. For configurations with a larger grid size this quickly becomes unfeasible.

This can be avoided through the observation that a change in velocity in one cell on the grid from a change in force in another depends on the relative distance between the two cells, not the absolute distance. Proof for this is given in Appendix B. This is visualised in Fig. 8. With this in mind it becomes apparent that for most configurations $\frac{\partial U}{\partial F}$ contains many duplicate values. This can be seen in Fig. 9. What we really only need to know is the velocity in a (boundary) point that is a certain distance away from a point force. Because of this we only need to calculate the velocity fields resulting from a singe unit force at the origin, on every combination of axes. From this $\frac{\partial U}{\partial F}$ can be reconstructed. For example: $\vec{U}[i,j,k]$ contains the velocity in a boundary point as a result from a virtual force with a relative distance of (i,j,k). We call these unit force velocity fields $\vec{U}^{(1)}_{a,b}$, where a, b is the combination of axis.

With $\vec{U}^{(1)}_{a,b}$ all of G can be constructed. When running the solver on the GPU, G is not explicitly saved in memory however. Instead $\vec{U}^{(1)}_{a,b}$ is read directly. This

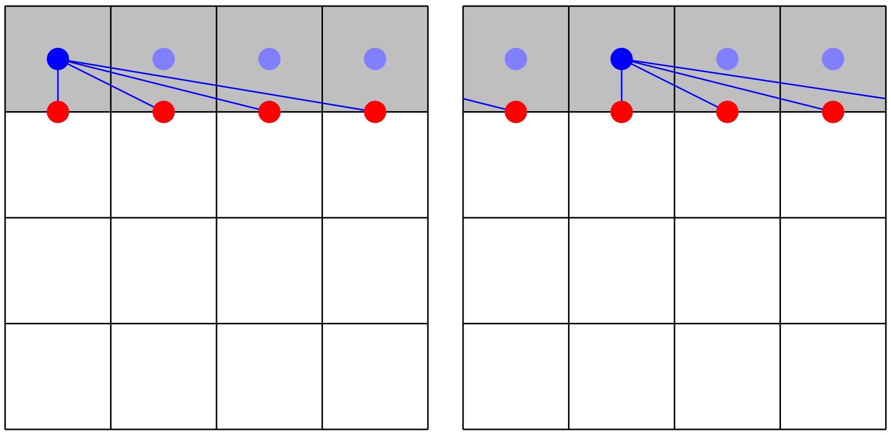


Figure 9: 2D examples of the relative distances of two virtual forces to all boundary points in a configuration, showing that these are all identical and thus would lead to many duplicate values in $\frac{\partial U}{\partial F}$. The same symmetry exists for the other virtual forces as well.

leads to a significant decrease in memory footprint, as saving all of G would require $9|B||V|$ floats. In contrast, all of $\vec{U}^{(1)}_{a,b}$ requires only $9NML$ floats. An intuitive way to look at this memory gain is that each boundary point that accesses the same coordinates of $\vec{U}^{(1)}_{a,b}$ has an identical offset and thus an identical value in G, meaning there are many duplicate cells. This memory optimization is especially important on the GPU given the slow nature and relatively low size of its global memory. Obtaining the velocity of a boundary point, which is interpolated from $\vec{U}$, has next to no overhead thanks to CUDA's hardware accelerated linear texture filtering. Hardware filtering is not present on the CPU. Additionally, memory latency and size is less of an issue here. As such when running the solver on the CPU G is generated on initialization. Its entries are placed in the order in which the gradient descent step accesses them to improve caching efficiency. That is on axis first, boundary point second and virtual force last.

In the original system G was fully generated on initialization. It would perform a separate Fourier transform for every entry in G leading to a total of $9|B||V|$ Fourier transforms and a large memory footprint (see Sec. 3.10). Using $\vec{U}^{(1)}_{a,b}$ has the additional benefit of easily enabling the implementation of moving solids, a future goal of the project. In the original system the gradient would have to be (partly) recalculated when a boundary point or virtual force moved to a new cell. This scenario now has no additional overhead whatsoever.

When the chosen stopping condition has not been satisfied (or there is no stopping condition in place) a gradient descent step will be performed by the solver. Each virtual force is nudged in the direction in which the velocity error in each boundary point looks to be decreased most.

3.8 Learning Rate Update

Following a gradient descent step γ_t is updated. When using the halving learning rate scheme the previous and current costs are factored in. When using the (dynamic) Polyak learning rate scheme the sum of the step length is used. As such in this case the learning rate update happens during the gradient descent step.

3.9 Result Logging

When the solver has reached its final solution the results will be displayed (if requested) and written to disk. An example of how the system will be displayed can be seen on the title page. $\vec{U}$ will then be written to a text file. The format that is used can be seen in Tab. 2. Additional logging can be enabled to save γ_t and C_t for every iteration as well. This allows the user to see how these values evolve over time.

Content	How often?
$N\ M\ L$	Once
$U_x[i]\ U_y[i]\ U_z[i]$	For every cell i in order of axis

Table 2: The format of the result output.

3.10 Memory Usage

To give an idea of the memory footprint of the solver on the GPU an overview of the main buffers is provided in Tab. 3. The real memory footprint is slightly larger than the sum of these buffers due to scalar variables and miscellaneous data that is stored.

Buffer	Count	Size	Type (Size)	
Real velocity fields	3	NML	float (4b)	
Real force fields	3	NML	float (4b)	
Unit force velocity fields	9	NML	float (4b)	
Complex velocity fields	3	$(\frac{N}{2}+1)ML$	complex (8b)	
Complex force fields	3	$(\frac{N}{2}+1)ML$	complex (8b)	
Boundary Points	1	$\|B\|$	BoundaryPoint (64b)	
Virtual Force Indices	1	$\|V\|$	unsigned int (4b)	+
		$(60NML + 48(\frac{N}{2}+1)ML + 64\|B\| + 4\|V\|)$b		

Table 3: The buffers used by the solver on the GPU and their memory footprint.

The memory footprint can be further illustrated by an example case. We take the 2 Moving Plates configuration at a size of $N = M = L = x$. As the plates are essentially 2 hyperplanes existing on the xz-axis with boundary point placed at a 1 unit distance from each other we have $|B| = 2NL$. With the hyperplanes at an offset of $\frac{M}{10}$ from the upper and lower limits of the grid we have $|V| = \frac{2M}{10}NL$. The resulting memory footprint as a function of x is plotted in Fig. 10.

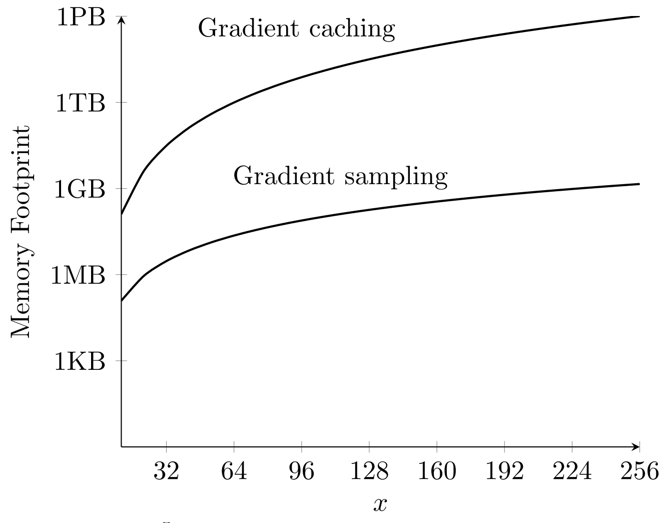
4 Results

The two points on which the solver is to perform are speed and convergence. Both will be benchmarked and compared to the old version of the program. First, an overview of the initialization, solving, and total runtime of the old and "base" new solvers are given. Next, additional optimizations for the new solver will be applied individually to observe their effects in isolation. All of the previously mentioned will use no stopping condition unless specified otherwise to make it easier to compare runtime. Finally, a "complete" benchmark will be given with all the successful optimizations and a stopping condition enabled. Benchmarking will be done on a range of configurations. These are given by their abbreviated name, which can be seen in Sec. 3.2. Some of these are not present in the old solver and as such these results are denoted by a dash (-). The system used to run the benchmarks can be seen in Tab. 4.

System Specifications	
CPU	AMD Ryzen 5 7600
GPU	NVIDIA GeForce RTX 3060 Ti GDDR6X
RAM	Kingston Fury Beast KF560C36BBEK2-16

Table 4: The specifications of the system on which the benchmarks are run.

Unfortunately, 3D configurations are currently nonfunctional. As such, all results will be run on quasi-2D configurations.



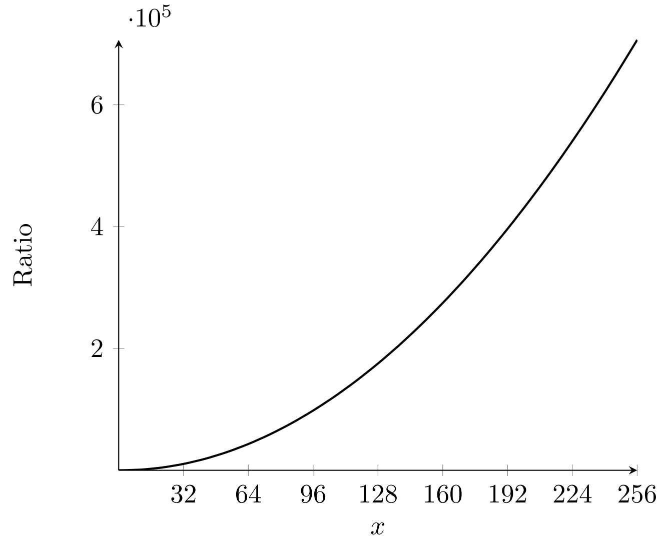


Figure 10: The approximate memory footprint of the 2 Moving Plates configuration given $N = M = L = x$, shown with the gradient caching (old) and gradient sampling (new) methods. The upper graph shows the size of the footprints while the lower graph shows the ratio between the two.

		Old (s)			New (s)			Improvement ($\times$)		
		Init.	Solv.	Total	Init.	Solv.	Total	Init.	Solv.	Total
2mp	32^2	0.007	5.089	5.096	0.109	0.521	0.629	0.064	9.768	8.368
	64^2	0.199	13.440	24.749	0.120	0.953	1.073	1.658	14.103	12.711
	128^2	5.835	94.199	100.034	0.117	1.578	1.695	49.872	59.695	59.017
	256^2	13.957	304.216	318.173	0.117	3.981	4.098	119.291	76.417	77.641
	512^2	-	-	-	0.186	31.039	31.225	-	-	-
tmp	32^2	0.059	12.715	12.774	0.108	1.037	1.145	0.546	12.261	11.156
	64^2	1.499	76.863	78.362	0.134	1.327	1.461	11.187	57.922	53.636
	128^2	44.871	489.202	534.073	0.104	2.552	2.656	431.452	191.694	201.082
	256^2	619.039	1612.193	2231.232	0.129	24.029	24.158	4798.752	67.094	92.360
	512^2	-	-	-	0.171	130.820	130.991	-	-	-
sp	32^2	0.016	5.994	6.010	0.109	0.908	1.017	0.147	6.601	5.910
	64^2	0.326	18.711	19.037	0.115	1.035	1.150	2.835	18.078	16.554
	128^2	7.810	97.890	105.700	0.112	1.592	1.704	69.732	61.489	62.031
	256^2	104.108	319.538	423.646	0.118	6.659	6.777	882.271	47.986	62.512
	512^2	-	-	-	0.173	47.866	48.039	-	-	-
4rm	32^2	-	-	-	0.093	1.112	1.205	-	-	-
	64^2	-	-	-	0.093	1.045	1.138	-	-	-
	128^2	-	-	-	0.098	1.528	1.626	-	-	-
	256^2	-	-	-	0.102	3.547	3.649	-	-	-
	512^2	-	-	-	0.172	17.209	17.381	-	-	-
ldc	32^2	-	-	-	0.107	1.073	1.180	-	-	-
	64^2	-	-	-	0.096	1.321	1.417	-	-	-
	128^2	-	-	-	0.093	1.965	2.058	-	-	-
	256^2	-	-	-	0.103	9.301	9.404	-	-	-
	512^2	-	-	-	0.171	69.651	69.822	-	-	-

Table 5: Overview of the runtime of the old and base new solvers. Any given runtime is the average of 3 separate runs of 10000 iterations.

4.1 Base

		New	
		10000	*max*
2mp	32^2	100.000%	100.000%
	64^2	99.999%	99.999%
	128^2	99.999%	99.999%
	256^2	99.996%	99.996%
	512^2	99.984%	99.984%
tmp	32^2	99.612%	99.744%
	64^2	99.763%	99.763%
	128^2	99.942%	99.942%
	256^2	99.982%	99.982%
	512^2	99.976%	99.976%
sp	32^2	98.608%	99.687%
	64^2	98.321%	99.259%
	128^2	98.658%	99.207%
	256^2	98.783%	99.354%
	512^2	98.817%	99.253%
4rm	32^2	99.498%	99.985%
	64^2	99.849%	99.874%
	128^2	99.943%	99.949%
	256^2	99.923%	99.966%
	512^2	99.811%	99.828%
ldc	32^2	95.824%	98.898%
	64^2	93.196%	97.814%
	128^2	92.326%	96.984%
	256^2	91.836%	95.631%
	512^2	91.952%	95.767%

Table 6: Overview of the convergence of the base new system. The results given are as calculated after 10000 iterations with no stopping condition and when run with unlimited iterations and the "stuck" stopping conditions, respectively.

An overview of the runtime of the old and base new system is provided in Tab. 5. This is also visualized in Fig. 11. Using the specified configurations and dimensions, a range of speedups between $5\times$ and $200\times$ is observed. The initialization shows a decrease in performance for a dimension of 32^2. However, where the old system's initialization time rapidly goes into dozens of seconds or more as the dimensions increase, the new system stays consistently within the 100 to 200 millisecond range. At no point is there a decrease in performance for the solving step. In nearly every case, a larger dimensionality leads to a greater speedup, indicating that the system's scalability has generally been improved. An outlier exists in the tmp configuration, where the improvement decreases past dimensions of 128^2. This is visible as a steep slope of the blue triangle plot

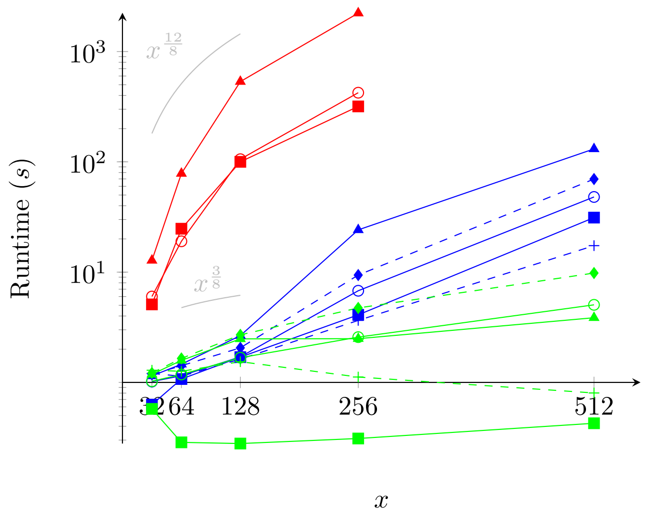


Figure 11: The run time in seconds as a function of the system size x for a quasi-2D configuration with dimensions $x \times x \times 1$. The use of colors and symbols indicates the 2mp (square), tmp (triangle), sp (circle), 4rm (plus) and ldc (diamond) configuration for the old (red), base new (blue), and total new (green) solvers as given in Table 5 and 10. A dashed line indicates that this configuration is not present in the old solver.

in Fig. 11. As can be seen in Fig. 6, this configuration contains relatively many solids, which can explain this poorer relative performance as the dimensions increase. This theory is backed up by the significantly improved results with limited virtual forces enabled (see Sec. 4.3 and the much flatter slope of the green triangle plot in Fig. 11.

An overview of the convergence of the base new system is provided in Tab. 6. Getting the old solver to properly converge proved problematic, and as such, we cannot provide our own data for its results. Florian Gaeremynck reported the convergence to be in the 99.000% to 100.000% range for the configurations available at the time[4]. As for our results, with the 2mp, tmp and 4rm configurations the new system never falls below 99.000% convergence with either 10000 or unlimited iterations. The latter provides only a minute improvement over the former. In the case of the sp and ldc configurations this difference becomes a lot more pronounced. Evidently the solver needs relatively more iterations here to converge to its best solutions. For ldc even the maximum convergence remains relatively low with no result reaching 99% convergence. We hypothesize that this is due to the large difference in velocities that the different boundary points have to satisfy. The upper lid has a velocity of $(5, 0, 0)$, while the others are all $\vec{0}$. Due to this discrepancy it might be hard to converge using one singular learning rate. However, at this point, this is speculation and this would need to be verified in future work.

		Runtime (s)	Convergence
2mp	32^2	3.901	100.000%
	64^2	14.930	100.000%
	128^2	54.875	99.999%
tmp	32^2	5.560	99.581%
	64^2	18.350	99.762%
	128^2	67.233	99.942%
sp	32^2	3.945	98.601%
	64^2	12.945	98.324%
	128^2	49.998	98.650%
4rm	32^2	3.409	99.480%
	64^2	10.554	99.834%
	128^2	33.723	99.943%
ldc	32^2	3.955	95.832%
	64^2	12.620	93.269%
	128^2	49.003	92.288%

Table 7: Overview of the runtime and convergence of the new system using NUIDFT's. The results given are as calculated after 10000 iterations.

4.2 NUIDFT

An overview of the runtime and convergence of the new system using NUIDFT's is provided in Tab. 7. From the data it is abundantly clear that using NUIDFT's in its current form has a large detrimental effect on the runtime. Despite this, we do not believe that the concept itself is to blame, but rather the implementation. Modern FFT libraries have a level of optimization that is near impossible to replicate, and even though our NUIDFT implementation is going up against a Fourier transform over the entire grid, it is still not able to eke out a runtime advantage. Further research is required to establish whether this statement holds true.

		Runtime (s)	Convergence
2mp	32^2	1.055	100.000%
	64^2	1.108	100.000%
	128^2	1.756	100.000%
	256^2	2.531	100.000%
	512^2	4.391	99.999%
tmp	32^2	1.302	99.551%
	64^2	1.621	99.426%
	128^2	2.486	99.776%
	256^2	4.124	99.876%
	512^2	7.901	99.936%
sp	32^2	1.084	99.071%
	64^2	1.280	99.006%
	128^2	1.680	98.952%
	256^2	2.695	99.193%
	512^2	5.049	99.491%
4rm	32^2	1.300	99.527%
	64^2	1.263	99.949%
	128^2	1.553	99.992%
	256^2	1.825	99.998%
	512^2	2.871	99.998%
ldc	32^2	1.212	95.363%
	64^2	1.672	94.110%
	128^2	2.791	93.895%
	256^2	4.719	93.359%
	512^2	9.823	94.068%

Table 8: Overview of the runtime and convergence of the new solver with virtual force culling enabled. Any given runtime is the average of 3 separate runs of 10000 iterations.

4.3 Limiting Virtual Forces

An overview of the runtime and convergence of the new system with limited virtual forces enabled is provided in Tab. 8. Once again the 2mp, tmp, and 4rm configuration converge well and there is no major difference between their base counterpart. For sp this difference is slightly larger, approaching 0.500%. With ldc this becomes several percents. We suspect this has the same cause as given for its lower convergence in Sec. 4.1. The difference in runtime is very prominent however, going up to over 5× as fast. This effect is noticeable everywhere, but especially on configurations that have a lot of virtual forces as can be seen in Fig. 6. The worsened runtime scaling for tmp as discussed in Sec. 4.1 has been completely eliminated. This indicates that the optimization has the desired effect of tempering the effect large solids have on the runtime.

		Runtime (s)	Convergence
2mp	32^2	1.220	100.000%
	64^2	1.311	99.999%
	128^2	2.107	99.999%
	256^2	4.569	99.998%
tmp	32^2	1.576	99.561%
	64^2	1.952	99.762%
	128^2	3.694	99.942%
	256^2	29.695	99.984%
sp	32^2	1.434	98.638%
	64^2	1.668	98.368%
	128^2	2.309	98.789%
	256^2	8.287	98.810%
4rm	32^2	1.656	99.371%
	64^2	1.656	99.912%
	128^2	2.118	99.707%
	256^2	4.511	99.882%
ldc	32^2	1.555	95.829%
	64^2	1.906	93.234%
	128^2	2.810	92.044%
	256^2	11.676	92.162%

Table 9: Overview of the runtime and convergence of the new solver with improved initialization enabled. Any given runtime is the average of 3 separate runs of 10000 iterations.

4.4 Improved Initialization

An overview of the runtime and convergence of the new system with improved initialization enabled is provided in Tab. 9. Convergence is not majorly impacted, generally hovering at around a few tenths of a percent from the base values. However, the runtime has increased across the board. We suspect that the overhead of setting up and running a smaller subsystem is greater than the time gained from the better subsequent initialization of the actual system.

		Runtime (s)	Convergence
2mp	32^2	0.575	100.000%
	64^2	0.286	100.000%
	128^2	0.279	100.000%
	256^2	0.310	99.999%
	512^2	0.426	99.999%
tmp	32^2	1.183	99.551%
	64^2	1.583	99.426%
	128^2	2.481	99.776%
	256^2	2.491	99.876%
	512^2	3.859	99.936%
sp	32^2	1.024	99.071%
	64^2	1.180	99.006%
	128^2	1.675	98.952%
	256^2	2.573	99.193%
	512^2	5.051	99.491%
4rm	32^2	1.289	99.527%
	64^2	1.261	99.949%
	128^2	1.541	99.992%
	256^2	1.119	99.998%
	512^2	0.802	99.998%
ldc	32^2	1.218	95.363%
	64^2	1.649	94.110%
	128^2	2.699	93.895%
	256^2	4.733	93.359%
	512^2	9.811	94.068%

Table 10: Overview of the runtime and convergence of the new solver with virtual force culling enabled. The vanished learning rate and stuck stopping conditions are enabled. Any given runtime is the average of 3 separate runs of 10000 iterations.

4.5 Complete

An overview of the runtime and convergence of the new system with all the successful optimizations and two stopping conditions enabled is provided in Tab. 10. The 2mp, tmp, and 4rm configurations all see improvements with the added stopping conditions as compared to using limited virtual forces. These are up to 10×, 2× and 3× respectively, which brings the first two up to a total improvement of 1026× and 895× over their old counterparts. Interestingly, the runtime for 4rm goes down as the dimensions increase. We currently have no good explanation for this phenomenon.

4.6 Research Questions

Due to the many subproblems that are posed in Sec. 1.3 and covered in this research, it can be tough to find a concrete answer to any specific one of them. Each one will be briefly covered using the previously presented results.

1. *Can we avoid or limit expensive context switches during the iterative section of the solving process?*
   By executing nearly the entire algorithm on the GPU and leaving all the relevant data in its device-side memory, the overhead of context switching has been reduced to a minimum. The usage of the GPU additionally provides an inherent performance gain. Speedups of up to 200× are observed for tested configurations and dimensions utilizing no further optimizations aside from memory improvements. Described in Sec. 3 and benchmarked in Sec. 4.

2. *Can we avoid, limit, or using domain-specific tricks, optimize expensive (inverse) discrete Fourier transforms?*
   While this is possible using an NUIDFT on $\widetilde{U}$, in its current form it does not lead to an increase in performance. We believe this to be due to the implementation rather than the theory itself. Described in Sec. 3.4 and benchmarked in Sec. 4.2.

3. *Can we perform the solving process exclusively for cells on a solid surface?*
   Partly, using the NUIDFT described in the previous question. Besides this, we have not discovered an operation that acts on the entire grid and can be performed in such a way where this is not the case. Described in Sec. 3.4.

4. *Can we limit the amount of virtual forces in a system while still converging to an acceptable solution?*
   Omitting virtual forces that are relatively far removed from any boundary point in the system, and thus most likely have only a minute influence over the solution, has shown to be a viable approach. Speedups of up to 5× are observed for tested configurations and dimension and no notable convergence problems arise on configurations that did not already have those to begin with. Described in Sec. 3.3 and benchmarked in Sec. 4.3.

5. *Is there a way to determine an approximate initialization value for virtual forces?*
   While this is possible using a scaled-down subsystem, in its current form it does not lead to an increase in performance for any configuration and metric. We consider it unlikely that a better implementation would lead to better results. Described in Sec. 3.3 and benchmarked in Sec. 4.4.

6. *How can we handle the data required by the solver most efficiently?*
Data no longer gets copied between the host and device during runtime. Additionally the highly inefficient method in which gradients were saved has been replaced by one that has a much more manageable footprint, especially with regards to scaling. An example is given where the old system would approach a petabyte of memory versus a gigabyte for the new system. Described in Sec. 3 and Sec. 3.7 and illustrated in Sec. 3.10.

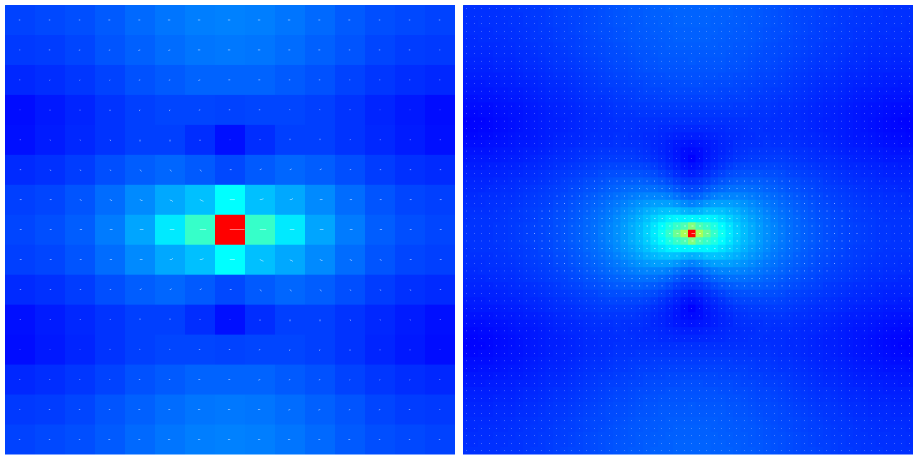
7. *Can we introduce arbitrary solid boundaries in the solver?*
Arbitrary solid boundaries have been introduced using an interpolation system over the discrete grid. This allows for smoother and more complex boundaries. Described in Sec. 2.2.

4.7 Relation to other Solvers

Comparing the performance of different fluid dynamics solvers is generally done using a metric which is known as Lattice (or cell) Updates per Second (LUPS). However, due to the novel nature of this solver, a direct comparison would not give an accurate representation of its relative performance. Other solvers tend to employ a time-dependent strategy, while we converge to our solution directly. The consequence of this is that even if other solvers have higher LUPS, this does not necessarily mean they obtain their solution faster. This is because our solver requires less total LUPS to reach its solution. Properly determining this solver's performance relative to other solvers thus requires more in depth research into the characteristics of both methods.

5 Conclusion

We have presented our work on the Stokes solver built by Bart Stam [6] and Florian Gaeremynck [4]. A variety of strategies are utilized in an attempt to reduce the solver's runtime and memory usage. Not all of these are successful (see Sec. 4.6). Nonetheless, through the remaining strategies large improvements are achieved in both runtime and memory usage. A decrease in overall runtime by a factor exceeding 1000× is observed in optimal situations. In more general cases this number is often in the triple digits. As the dimensions of the system grow, these factors only increase. Similarly, memory usage has been decreased by up to a petabyte of data for larger dimensions. Improvements have also been made to the solver's flexibility. A more versatile boundary system has been implemented which allows the solver to handle arbitrary geometries. Furthermore, a new command-line interface removes the need to deal with a set of hard-coded simulation parameters. Through the optimizations and improved flexibility obtained with this research the solver becomes a much more viable tool for use in a research environment, paving the way for practical application.



(a) $15 \times 15 \times 1$ (b) $60 \times 60 \times 1$

Figure 12: A visualization of $U^1_{x/xyz}$, translated by $(\frac{N}{2}, \frac{M}{2}, \frac{L}{2})$ for clarity. When not transformed the highest velocity is at $(0, 0, 0)$, the location of the point force.

6 Future Work

While the use of NUIDFT's did not provide a runtime improvement, we believe this is due to the implementation more than the concept itself. Time can be invested into creating a better implementation or a (CUDA-based) NUIDFT library can be used. However, at the time of writing the latter does, to our knowledge, not exist.

One of the major breakthroughs in improving performance and memory usage is the removal of a full representation of G in favor of sampling $\vec{U}^{(1)}_{a,b}$. Intuitively and when judged by eye there appears to be a vertical and horizontal symmetry present in $\vec{U}^{(1)}_{a,b}$ as can be seen in Fig. 12. If these symmetries are mathematically proven the memory footprint of $\vec{U}^{(1)}_{a,b}$ can be reduced by 75%.

Moving solids are intended to be implemented in the project and with the new gradient system (see Sec. 3.7) this is made much easier. To implement this feature boundary points would have to be grouped internally so they can move through the fluid as a single object. The forces acting on the object can be calculated from $\vec{U}$.

The additions made to the project include both high level algorithmic and structural changes and low level code optimizations. On the low level side, while care has been put into creating GPU code that runs efficiently there are indubitably performance gains to be made purely by optimizing the existing code. One promising avenue that can be taken is consolidating the entire algorithm into one mega-kernel. By doing this several global memory loads can be eliminated by keeping the relevant data in thread and shared memory during the entire solving process.

To make working with the solver easier a graphical user interface could be developed. Not everyone is adept with working from the command line and this could increase accessibility. In similar vein a Python wrapper could be created. Python is a language that many researchers are familiar with and it would make it easier for external programs to interact with the solver.

All configurations are hard-coded aside from their dimensionality. To make the solver usable in real research scenarios a much more flexible configuration system is required. This includes more pre-made configurations of common problems as well as ways to create custom configurations. If a graphical user interface is made the option could be given to have users "draw" their desired boundaries by hand. Another option is to give the user the ability to import an image of a configuration from which the solver infers the boundaries through edge-detection algorithms.

Different configurations can benefit from different learning rate schemes. To better find solutions for a wide variety of configurations additional learning rate schemes can be implemented. More tricks can be used to improve convergence; For instance, adding weighting to the error in each boundary points so virtual forces prioritise the most important parts of a configuration.

Visualizing the solution is done on the CPU using CImg. This is fine in the current context of "static" simulations where the output is a single flow field. However, once moving solids are implemented, having an animated window is preferable. In this scenario it would be better to draw the visualization on the GPU. Otherwise, the flow field would have to be copied back to the CPU at every frame which would re-introduce the context switching bottleneck. Besides, all the data required for the visualization is already on the GPU, and GPU's are naturally equipped to handle these sorts of tasks. Making the visualizer 3D would be especially great for interpreting the systems that are being run and their solutions.

A Program flow

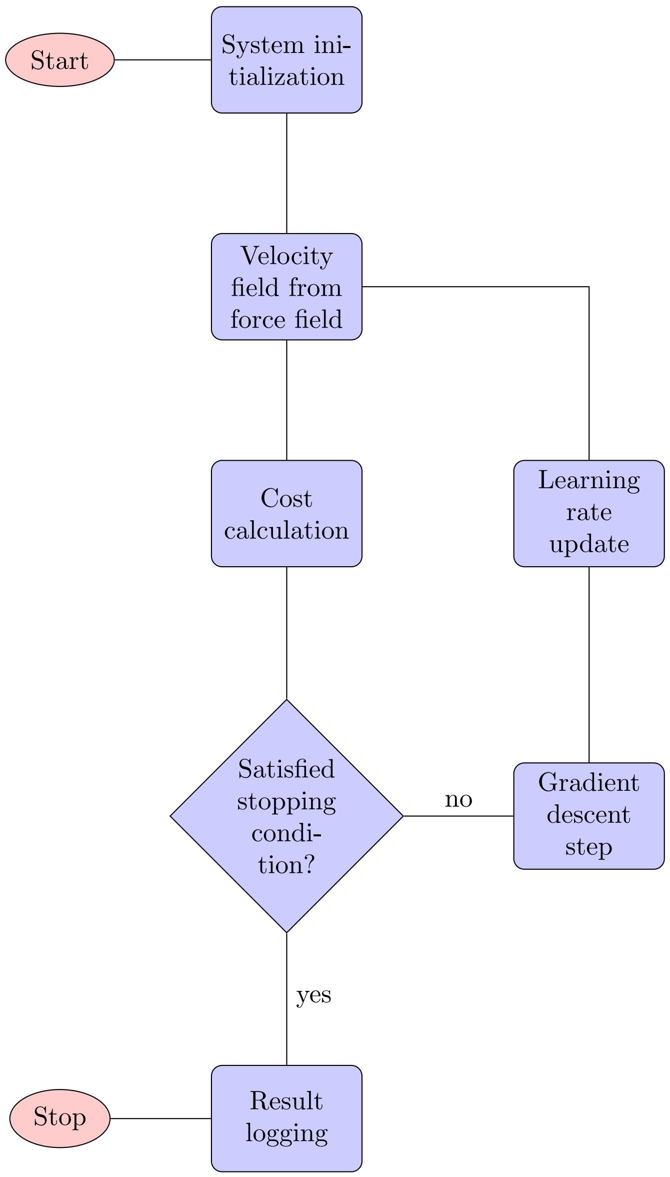


Figure 13: The program flow of the solver. Round red nodes indicate the entry and exit points. Square blue nodes indicate steps in the program flow. Diamond blue nodes indicate a conditional branching.

B Relative relation between forces and velocities

$$U[q,r,s] = \frac{1}{NML} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} \sum_{l=0}^{L-1} \exp\left(2\pi\left(\frac{nq}{N} + \frac{mr}{M} + \frac{ls}{L}\right)\right) \widetilde{U}[n,m,l], \quad (18)$$

$$\widetilde{U}[n,m,l] = A_{nml}\widetilde{F}[n,m,l], \quad (19)$$

$$\widetilde{F}[n,m,l] = \sum_{q'=0}^{N-1} \sum_{r'=0}^{M-1} \sum_{s'=0}^{L-1} \exp\left(-2\pi\left(\frac{nq'}{N} + \frac{mr'}{M} + \frac{ls'}{L}\right)\right) F[q',r',s'], \quad (20)$$

$$\frac{\partial \widetilde{U}[q,r,s]}{\partial \widetilde{F}[q',r',s']} = \frac{1}{NML} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} \sum_{l=0}^{L-1} \sum_{q'=0}^{N-1} \sum_{r'=0}^{M-1} \sum_{s'=0}^{L-1}$$
$$\exp\left(2\pi\left(\frac{n(q-q')}{N}\right)\right) \exp\left(2\pi\left(\frac{m(r-r')}{M}\right)\right) \exp\left(2\pi\left(\frac{l(s-s')}{L}\right)\right) A_{nml}. \quad (21)$$

where A is a matrix containing the transformation from $\widetilde{F}$ to $\widetilde{U}$ as described in Eq. 7.

References

[1] Yinan Chen. *Free stock photos and public domain pictures of beach, ocean, and the City of Daytona Beach, Florida Waves come crashing in from the Atlantic Ocean.* 2013. URL: `https://commons.wikimedia.org/wiki/File:Gfp-florida-daytona-beach-ocean-waves.jpg`.

[2] Michael A. Day. "The no-slip condition of fluid dynamics". In: *Erkenntnis* 33.3 (Nov. 1990), pp. 285–296. ISSN: 1572-8420. DOI: `10.1007/BF00717588`. URL: `https://doi.org/10.1007/BF00717588`.

[3] Sergey V Ershkov et al. "Towards understanding the algorithms for solving the Navier–Stokes equations". In: *Fluid Dynamics Research* 53.4 (July 2021), p. 044501. DOI: `10.1088/1873-7005/ac10f0`. URL: `https://dx.doi.org/10.1088/1873-7005/ac10f0`.

[4] Florian Gaeremynck. *Improved methods on GPU based versatile and efficient hydrodynamics code for scientific applications.* Dec. 2022.

[5] Boris Polyak. *Introduction to Optimization.* July 2020.

[6] Bart Stam. *A GPU-based versatile and efficient hydrodynamics code for scientific applications.* July 2021.

[7] www.Pixel.la. *Molten lava flowing.* 2016. URL: `https://commons.wikimedia.org/wiki/File:Molten_lava_flowing.jpg`.