DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

GPU-Accelerated Pressure Solver for Deep Learning

Martin Vincent Wepner

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

GPU-Accelerated Pressure Solver for Deep Learning

GPU-Beschleunigter Druck Löser für Deep Learning

Author:	Martin Vincent Wepner
Supervisor:	Prof. Dr. Nils Thürey
Advisor:	Philipp Holl
Submission Date:	15. March 2019

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15. March 2019 Martin Vincent Wepner

Acknowledgments

todo

Abstract

TODO: Abstract

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
1.1 Φ_{Flow} . . . 1
1.2 Using TensorFlow Custom Operations for efficient computation . . . 2
1.3 Pressure Solve and Conjugate Gradient . . . 2
1.4 TODO: Ausblick . . . 2

2 Related Work **3**

3 Navier Stokes Equations **4**
3.1 Staggered MAC-grids . . . 5
3.2 The role of the pressure solve in incompressible Navier Stokes equations 5
3.2.1 Discretization and derivation of the pressure equation . . . 5
3.2.2 Boundary Conditions . . . 6
3.3 The Laplace Matrix . . . 8

4 CUDA Pressure Solver **10**
4.1 Laplace Matrix generation . . . 10
4.1.1 Optimization 1: Abandonment of CSR *rowcnt* array . . . 11
4.1.2 Optimization 2: Merge CSR *data* and *columnptr* arrays . . . 12

5 Results **14**
5.1 Section . . . 14
5.1.1 Subsection . . . 14

List of Figures **16**

List of Tables **17**

Bibliography **18**

1 Introduction

The simulation of fluids like the behavior of water in a waterfall or smoke ascending from a fire has become a key element of visual effects in the past years increasing the immersion of movies and computer games. Two main approaches have arisen to achieve realistic results: The Lagrangian method, which computes the dynamics of individual particles in respect to each other and the Eulerian method, that uses a fixed grid and computes how mass is moved within the grid. Existing solutions are often highly computational expensive leading to trade-offs between speed and accuracy, density or size and thus believability of the simulation.

Some recent approaches tackling this issue by leveraging the power of Deep Neural Networks and transforming it into a data-driven unsupervised learning problem. Until recently only Eulerian methods were fully differentiable making them suitable for deep-learning. Using simulation data such as velocity and density fields of each timestep, Deep Neural Networks can be trained to mimic the concepts of the Navier-Stokes equation [Paper of Jonathan Tompson, Ken Perlin].

For larger simulations, the amount of physical memory needed to store the training data gets huge quickly. To avoid this, running data generation and the training process itself can be done in parallel: After every time-step is calculated the network's weights can be adjusted.

1.1 Φ_{Flow}

Φ_{Flow} is a toolkit written in Python and currently under development at the TUM Chair of Computer Graphics and Visualization for solving and visualizing n-dimensional fluid simulations. It focuses on to keep every simulation step differentiable which makes it suitable for backpropagation. Backpropagation, on the other hand, is required to calculate the gradient of the loss function of Neural Networks to adjust the weight of the network. Being developed on top of TensorFlow, it is not only capable of running on CPU, but also completely on GPU. TensorFlow simplifies the implementation of Machine Learning algorithms including the use of Neural Networks for predictions and classification. This makes Φ_{Flow} a powerful tool to train Neural Networks on fluid simulations and also visualize them by its interactive GUI running in the browser.

1.2 Using TensorFlow Custom Operations for efficient computation

This work proposes an implementation to solve the pressure equation - the most computationally intensive part of Eulerian fluid simulations. Being as efficient as possible is crucial for fast simulations and thus most importantly for the training of neural networks. Φ_{Flow} already comes with its own pressure solver running directly in TensorFlow. However, TensorFlows dataflow graph is unnecessarily complex which brings overhead to the computation.

TensorFlow offers the possibility to write custom operations ("Custom ops") in C++ to tackle this issue: Writing efficient code to solve a specific problem within the dataflow graph. Furthermore, the CUDA API by Nvidia makes it possible to write low-level CUDA code that runs natively on GPUs. Combining Custom Ops and CUDA kernels lead to a solution that is both efficient and transparent and also compliant to TensorFlow's Tensors which is required within Φ_{Flow}. It can then be compared to Φ_{Flows} built-in solution.

1.3 Pressure Solve and Conjugate Gradient

The Pressure Solve Problem can be expressed as a system of linear equations whose solution is the exact pressure value of each cell on a grid to be compliant with the incompressibility constraint of fluids. The matrix of the linear system is symmetric and positive-definite which makes it suitable to be solved by the Conjugate Gradient method. GPUs are well suited for parallel computations, especially linear algebra. The cg-method consists of only some vector operations and matrix-vector multiplications and can, therefore, be run on GPUs. Being a sparse matrix gives additional room for improvement, by not only keeping the required memory low but also decreasing the amount of data, that needs to be sent between the GPU's global memory to the on-chip local memory and vice versa. Compared to computation, memory operations are very costly. By not changing the boundary conditions of the simulation the matrix stays constant. Costly transfers between CPU and GPU can, therefore, kept low.

1.4 TODO: Ausblick

2 Related Work

TODO: Related Work

3 Navier Stokes Equations

The fundamentals of any fluid simulation lay in the Navier Stokes equations:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{F} + \nu \nabla \cdot \nabla \vec{u} \tag{3.1}$$

$$\nabla \cdot u = 0 \tag{3.2}$$

The letter $\vec{u}$ is the velocity vector of the fluid.
The Greek letter Rho ρ stands for the density of the fluid. E.g. $\rho_{Water} \approx 1000 kg/m^3$.
The Arabic p, on the other hand, holds the pressure, which is the force per unit the fluid exerts on its surroundings. Note $\rho \neq p$.
External forces like gravity and buoyancy are collected in the letter $\vec{F}$.
The Greek letter Nu ν is called kinematic viscosity. The higher the viscosity of a fluid is the higher its inherent inertia. For example, honey has a higher viscosity than water.

The first of those equations - the "momentum equation" - very briefly describes how the fluid accelerates due to the forces acting on it. The second is called "incompressibility condition" and means that the "volume?!" of the fluid doesn't change.

However, real fluids are compressible, so why do we assume them to be incompressible? Liquid fluids are in theory compressible, but it requires a lot of force to do so, and it is also only possible until a small degree of compression. Gases like air are easier to compress, but only with the help of pumps or extreme situations like blast waves caused by for example explosions. Those kind of scenarios are called "compressible flow" and are expensive to simulate. Additionally, they are only visible on a macroscopic level and hence irrelevant for the animation of fluids.

Viscosity needs to be considered only in those cases where animating viscous fluids like honey is desired. For most cases, however, it plays a minor role and can be dropped. Those ideal fluids without viscosity are called "inviscid". The Navier Stokes equations can then be simplified to the following, the so-called "Euler Equations":

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho} \nabla p = \vec{F} \tag{3.3}$$

$$\nabla \cdot u = 0 \tag{3.4}$$

3.1 Staggered MAC-grids

Grid-based fluid simulations have been discussed in this work already, but not yet clarified what's exactly meant by "grid". To numerically approximate the Navier Stokes equations, quantities like pressure, velocity and fluid density needs to be computed through space. As the name suggests, those quantities are laid out on some regular grid. [Harlow and Welch 1965] have developed the Marker-and-Cell (MAC) grid technique which many simulators still use. Instead of tracking all quantities on the same grid, they are staggered on the grid. This staggering avoids biased or null spaced central differences for the pressure gradient.

TODO Figures of grids

3.2 The role of the pressure solve in incompressible Navier Stokes equations

The pressure solve is part of the Navier Stokes equations:

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho}\nabla p = 0 \quad s.t. \quad \nabla \cdot \vec{u} = 0 \tag{3.5}$$

This means, that the pressure always needs to be so, that it advects the velocity field $\vec{u}$ divergence-free and thus makes the fluid incompressible. [TODO: Maybe write *what exactly* the pressure is and what it means for the simulation]

3.2.1 Discretization and derivation of the pressure equation

In this section, we briefly derive a numerical approximation for the pressure equation of the Navier Stokes equations with the help of a MAC-grid for two-dimensional simulations. Bridson [Bri15] [BM07] already explained this for both 2D and 3D in detail.
Discretizing 3.5 gives the following equations:

$$\vec{u}^{n+1} = \vec{u}^{n} - \Delta t \frac{1}{\rho}\nabla p \quad s.t. \quad \nabla \cdot \vec{u}^{n+1} = 0 \tag{3.6}$$

The divergence of a 2 dimensional vector $\vec{u} = (u, v)$ is defined as

$$\nabla \cdot \vec{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \tag{3.7}$$

So for a fluid grid cell (i, j):

$$(\nabla \cdot \vec{u})_{i,j} \approx \frac{u^{n+1}_{i+1/2,j} - u^{n+1}_{i-1/2,j}}{\Delta x} + \frac{v^{n+1}_{i,j+1/2} - v^{n+1}_{i,j-1/2}}{\Delta x} \tag{3.8}$$

Using the central difference for $\frac{\partial p}{\partial x}$ and $\frac{\partial p}{\partial y}$ the velocity update is:

$$u^{n+1}_{i+1/2,j} = u^{n}_{i+1/2,j} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j} - p_{i,j}}{\Delta x} \tag{3.9}$$

$$v^{n+1}_{i+1/2,j} = v^{n}_{i,j+1/2} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1/2} - p_{i,j}}{\Delta x} \tag{3.10}$$

Substituting 3.9 and 3.10 into 3.8 finally gives:

$$\frac{1}{\Delta x}\left[\left(u^{n}_{i+1/2,j} - \Delta t \frac{1}{\rho}\frac{p_{i+1,j} - p_{i,j}}{\Delta x}\right) - \left(u^{n}_{i-1/2,j} - \Delta t \frac{1}{\rho}\frac{p_{i-1,j} - p_{i,j}}{\Delta x}\right) + \left(v^{n}_{i,j+1/2} - \Delta t \frac{1}{\rho}\frac{p_{i,j+1} - p_{i,j}}{\Delta x}\right) - \left(v^{n}_{i,j-1/2} - \Delta t \frac{1}{\rho}\frac{p_{i,j-1} - p_{i,j}}{\Delta x}\right)\right] = 0$$

$$-4p_{i,j} + p_{i+1,j} + p_{i,j+1} + p_{i-1,j} + p_{i,j-1} = \frac{\rho \Delta x^2}{\Delta t}\left(\frac{u^{n}_{i+1/2,j} - u^{n}_{i-1/2,j}}{\Delta x} + \frac{v^{n}_{i,j+1/2} - v^{n}_{i,j-1/2}}{\Delta x}\right) \tag{3.11}$$

Now we have a bunch of linear equations in the form of $\mathbf{A}p = \frac{\rho \Delta x^2}{\Delta t} \nabla \cdot \vec{u}^n$. $\mathbf{A}$ is symmetric and each row represents one cell (i, j) on the grid.

3.2.2 Boundary Conditions

Solids

If an inviscid fluid is in contact with a wall respective solid, it should move in the tangential direction of the solids normal. To achieve this the velocity components of the solid normal needs to match the speed of the fluid at this position:

$$\vec{u} \cdot \hat{n} = \vec{u}_{solid} \cdot \hat{n} \tag{3.12}$$

Concerning the pressure update, assume grid cell (i, j) to be fluid and its neighbor $(i + 1, j)$ to be solid. Then $u^{n+1}_{i+1/2,j} = u_{solid}$ and 3.9 can be rearranged to

$$p_{i+1,j} = p_{i,j} + \frac{\rho \Delta x}{\Delta t}(u_{i+1/2,j} - u_{solid}) \tag{3.13}$$

This is called Neumann boundary condition.

Air

Compared to water, air is 700 times lighter. For simplicity, the pressure p of air regions may be set to zero since it has only minimal impact and therefore no significant visual effect. Since we directly specify the value of the pressure at those air cells it is called a Dirichlet boundary condition.

Extended MAC-grid

As we figured in 3.11 for one pressure update of one cell we need to consider all its neighbors. What about those grid cells laying on the edge of the grid? Technically those cells have fewer neighbors than the others. To solve this issue, we extend the grid that holds the information about the state (fluid, solid, air) on every dimension. So for a (n x m) sized simulation, we need an (n+2 x m+2) grid that holds the information. We differ between open boundaries where those extended cells are just air cells and closed boundaries where solid cells surround the grid.

TODO figure of those masks

3.3 The Laplace Matrix

Taking a closer look and with some rearrangements on equation 3.11 shows that it is an approximation of the Poisson problem $-\Delta t/\rho \nabla \cdot \nabla p = -\nabla \cdot \vec{u}$. Because of that I call **A** *Laplace Matrix* in the following.

To implement a method that extracts the Laplace Matrix, we need to take a look on the boundary conditions first. Assume a grid cell (i, j) of which one neighbor $(i-1, j)$ is an air cell ($p_{i-1,j} = 0$) and another $(i+1, j)$ is a solid cell ($p_{i+1,j} = p_{i,j} + \frac{\rho \Delta x}{\Delta t}(u_{i+1/2,j} - u_{solid})$). The remaining neighbors are fluid cells. Knowing that, we can rearrange 3.11 to the following:

$$\begin{aligned} &-4p_{i,j} + \left[p_{i,j} + \frac{\rho \Delta x}{\Delta t}(u_{i+1/2,j} - u_{solid})\right] + p_{i,j+1} + 0 + p_{i,j-1} \\ &= \frac{\rho \Delta x^2}{\Delta t}\left(\frac{u^n_{i+1/2,j} - u^n_{i-1/2,j}}{\Delta x} + \frac{v^n_{i,j+1/2} - v^n_{i,j-1/2}}{\Delta x}\right) \end{aligned} \tag{3.14}$$

which results in

$$-3p_{i,j} + p_{i,j+1} + p_{i,j-1} = \frac{\rho \Delta x^2}{\Delta t}\left(\frac{u^n_{solid} - u^n_{i-1/2,j}}{\Delta x} + \frac{v^n_{i,j+1/2} - v^n_{i,j-1/2}}{\Delta x}\right) \tag{3.15}$$

We can make some very important observations from 3.11, 3.14 and 3.15 for building the Laplace Matrix in code:

- **A** is similar to an adjacency matrix. Every row represents one cell (i, j) and every column all cells on the MAC-grid. Non-neighbor entries of **A** are 0.
- Note that the coefficient of $p_{i,j}$ in 3.11 matches the negative number of neighbors of a cell (i, j). Let's call this coefficient $k_{diagonal} = -dim(grid) * 2$, since it is always on the diagonal of **A**'s entries.
- Every neighbor of cell (i, j) is 1 in the corresponding row of **A**.
- For every neighbor of cell (i, j) that is a solid cell decrement $k_{diagonal}$ and set the corresponding column to 0.
- For every neighbor of cell (i, j) that is an air cell set the corresponding column to 0.

With that in mind, we can derive an algorithm which builds **A**. Note that most of **A** is zero so it is a sparse matrix. Also, because of the symmetry of neighboring cells, **A** is symmetric.

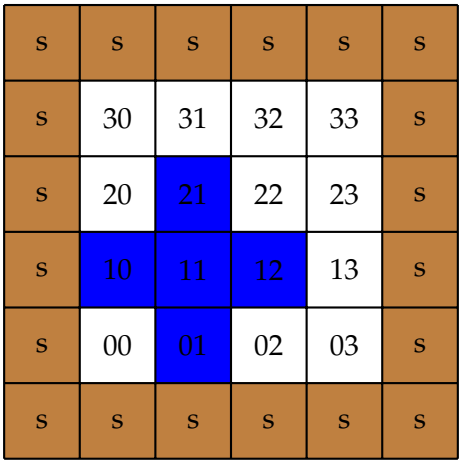

Figure 3.1: Example grid with fluid (blue), air (white) and solid (brown) cells and a closed boundary.

$$
\begin{array}{c}
 \\
00 \\
01 \\
02 \\
03 \\
10 \\
11 \\
12 \\
13 \\
20 \\
21 \\
22 \\
23 \\
30 \\
31 \\
32 \\
33
\end{array}
\begin{array}{c}
\begin{array}{cccccccccccccccc}
00 & 01 & 02 & 03 & 10 & 11 & 12 & 13 & 20 & 21 & 22 & 23 & 30 & 31 & 32 & 33
\end{array} \\
\begin{pmatrix}
-2 & 1 & & & 1 & & & & & & & & & & & \\
0 & -3 & 0 & & & 1 & & & & & & & & & & \\
 & 1 & -3 & 0 & & & 1 & & & & & & & & & \\
 & & 0 & -2 & & & & 0 & & & & & & & & \\
0 & & & & -3 & 1 & & & 0 & & & & & & & \\
 & 1 & & & 1 & -4 & 1 & & & 1 & & & & & & \\
 & & 0 & & & 1 & -4 & 0 & & & 0 & & & & & \\
 & & & 0 & & & 1 & -3 & & & & 0 & & & & \\
 & & & & 0 & & & & -3 & 1 & & & 0 & & & \\
 & & & & & 1 & & & 0 & -4 & 0 & & & 0 & & \\
 & & & & & & 1 & & & 1 & -4 & 0 & & & 0 & \\
 & & & & & & & 0 & & & 0 & -3 & & & & 0 \\
 & & & & & & & & 0 & & & & -2 & 0 & & \\
 & & & & & & & & & 1 & & & 0 & -3 & 0 & \\
 & & & & & & & & & & 0 & & & 0 & -3 & 0 \\
 & & & & & & & & & & & 0 & & & 0 & -2
\end{pmatrix}
\end{array}
\cdot
\begin{pmatrix}
p_{0,0} \\ p_{0,1} \\ p_{0,2} \\ p_{0,3} \\ p_{1,0} \\ p_{1,1} \\ p_{1,2} \\ p_{1,3} \\ p_{2,0} \\ p_{2,1} \\ p_{2,2} \\ p_{2,3} \\ p_{3,0} \\ p_{3,1} \\ p_{3,2} \\ p_{3,3}
\end{pmatrix}
$$

Figure 3.2: Laplace Matrix **A** derived from Fig. 3.1 with corresponding pressure vector. Diagonal are marked as red, neighbors as blue and non neighbors are left out.

4 CUDA Pressure Solver

In this chapter, we find an efficient numerical approach to solve the system of linear equations $\mathbf{A}p = -\frac{\rho \Delta x^2}{\Delta t} \nabla \cdot \vec{u}^n$ using TensorFlows Custom Ops and Nvidia CUDA Kernels. First, we discuss how $\mathbf{A}$ is built and stored in memory and then how the Conjugate Gradient method can be used to solve it.

We will use some recurring variables which I clarify here:
dimensions is an array which holds the size of each dimension of the grid.
dimSize holds the number of dimensions.
dimProduct is the product $\prod_{i=0}^{dimSize} dimensions[i]$ of all dimensions.
mask holds the information of fluid, air and solid cells including the boundary cells.
maskDimensions is an array that holds the dimensions of the mask.

4.1 Laplace Matrix generation

The goal of this section is to find a parallelizable algorithm that creates the Laplace Matrix A, so that a GPU can compute exactly one cell (i, j) or one row in $\mathbf{A}$ per thread.

GPU memory is very limited. For a simulation of the size 64 x 64 x 64 the dimension of the Laplace Matrix is 262.144 x 262.144. This would require 256 Gigabyte of GPU-memory if we tried to cache the matrix naively, which is not possible without further ado at the time of this writing.

Take a look back on the example Laplace Matrix in Fig. 3.2. Notice, that most of it is zero. In fact, there are only at most as many non-zero entries as the number of neighbors of a cell plus one for the cell itself. For 2D this means at most five and for 3D at most seven entries per row. Or in general $maxNonZerosPerRow = dimSize * 2 + 1$.

The *compressed sparse row* (CSR) [TODO: No paper found] format tackles this problem by only storing information of non-zero entries of a sparse Matrix. It consists of three arrays:

data:	stores all non-zero entries in row-major order.
columnptr:	stores for all non-zero entries their column index in row-major order.
rowcnt:	stores ranges of *data* entries for indexing the corresponding row.

$$\begin{pmatrix} 4 & 0 & 0 & 0 & 0 & 0 \\ 1 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ 3 & 0 & 0 & 4 & 0 & 0 \end{pmatrix} \quad \begin{aligned} \textit{data} &= [4,1,4,4,3,4] \\ \textit{columnptr} &= [0,0,1,2,0,3] \\ \textit{rowcnt} &= [0,1,3,4,6] \end{aligned}$$

Figure 4.1: Example Matrix in CSR format

Because **A** is sparse to a vast extent, the CSR format reduces memory consumption dramatically. However, it requires three memory accesses to get all data of one entry. Compared to pure computation, memory accesses are very costly on GPUs and should be kept as low as possible [Fan+18][Fuj+13]. As we will later see, when we use the Laplace Matrix for solving the equation system, it is crucial to reduce those accesses to improve speed.

[TODO: Insert naive algorithm here]

4.1.1 Optimization 1: Abandonment of CSR *rowcnt* array

The *i*th element of the *rowcnt* array points to the index in *data* of the first non-zero entry of the *i*th row. So for every row i we know that all non-zero entries of this row lay in the range $\left[\textit{rowcnt}[\text{i}]\ ;\ \textit{rowcnt}[\text{i+1}]\right[$ in the *data* array.

The maximum number of non-zero entries per row in the Laplace Matrix is $maxNonZerosPerRow$ as defined above. It is only the maximum possible number of non-zeros because neighbor cells of (i,j) might be air or solid cells, thus zero in the (i,j)th row of the Laplace Matrix. For cells at the edge of the grid, we set the corresponding value for neighbor cells that lay outside the grid to zero in the *data* array as well. The CSR format will remove those zero entries but if we keep them, we have a regular arrangement of *data* entries per row. As a trade-of this will require slightly more memory. However now we are able calculate the index of the row i_{row} of any *data* entry by its index i_{data}:

$$i_{row} = \left\lfloor \frac{i_{data}}{maxNonZerosPerRow} \right\rfloor \tag{4.1}$$

[...]

4.1.2 Optimization 2: Merge CSR *data* and *columnptr* arrays

Another observation we can make from equation 3.11 and Fig. 3.2 is that the values of *data* in the range $[-2 \cdot dimSize\ ; 1]$. Using an $INT32$ with 32 bits is a waste of memory. Instead a $INT8$ is more than enough (except in the unlikely event that you want to simulate more than 127 dimensions).

$$INT32 : \underbrace{0000\,1111\ 0000\,1111\ 0000\,1111}_{\text{columnidx}}\ \underbrace{0000\,1111}_{\text{data}}$$

Algorithm 1 My algorithm

```
1: procedure MyProcedure(1,2,3)
2:     stringlen ← length of string
3:     i ← patlen
4: top:
5:     if i > stringlen then return false
6:     j ← patlen
7: loop:
8:     if string(i) = path(j) then
9:         j ← j − 1.
10:        i ← i − 1.
11:        goto loop.
12:        close;
13:    i ← i + max(delta1(string(i)), delta2(j)).
14:    goto top.
```

```
if i ≥ maxval then
    i ← 0
else
    if i + k ≤ maxval then
        i ← i + k
```

5 Results

5.1 Section

Citation test [Lam94].

5.1.1 Subsection

See Table 5.1, Figure 5.1, Figure 5.2, Figure 5.3.

Table 5.1: An example for a simple table.

A	B	C	D
1	2	1	2
2	3	2	3

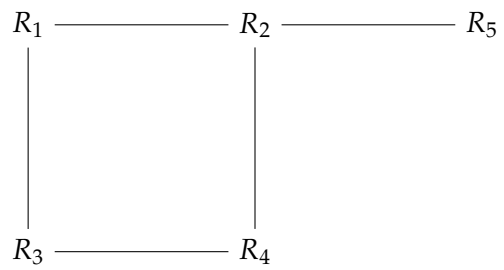

Figure 5.1: An example for a simple drawing.

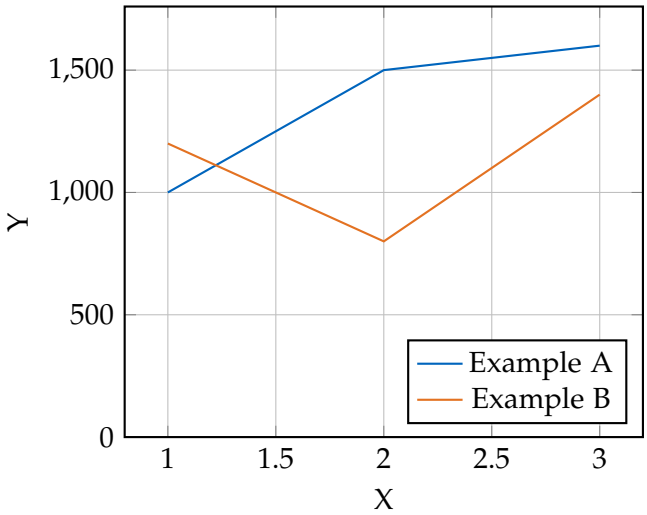

Figure 5.2: An example for a simple plot.

```
SELECT * FROM tbl WHERE tbl.str = "str"
```

Figure 5.3: An example for a source code listing.

List of Figures

3.1 Example grid with fluid (blue), air (white) and solid (brown) cells and a closed boundary. 9
3.2 Laplace Matrix **A** derived from Fig. 3.1 with corresponding pressure vector. Diagonal are marked as red, neighbors as blue and non neighbors are left out. 9

4.1 Example Matrix in CSR format 11

5.1 Example drawing 14
5.2 Example plot 15
5.3 Example listing 15

List of Tables

5.1 Example table 14

Bibliography

[BM07] R. Bridson and M. Müller-Fischer. "Fluid simulation: SIGGRAPH 2007 course notes Video files associated with this course are available from the citation page." In: *ACM SIGGRAPH 2007 courses*. ACM. 2007, pp. 1–81.

[Bri15] R. Bridson. *Fluid simulation for computer graphics*. AK Peters/CRC Press, 2015.

[Fan+18] M. Fang, J. Fang, W. Zhang, H. Zhou, J. Liao, and Y. Wang. "Benchmarking the GPU memory at the warp level." In: *Parallel Computing* 71 (2018), pp. 23–41.

[Fuj+13] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro. "Data transfer matters for GPU computing." In: *2013 International Conference on Parallel and Distributed Systems*. IEEE. 2013, pp. 275–282.

[Lam94] L. Lamport. *LaTeX : A Documentation Preparation System User's Guide and Reference Manual*. Addison-Wesley Professional, 1994.