# A Nonlinear Differential Equation Solver With Potential Application To Pelton Turbines

**Jesper Tveit**


Dissertation for the degree of philosophiae doctor (PhD)
at the University of Bergen

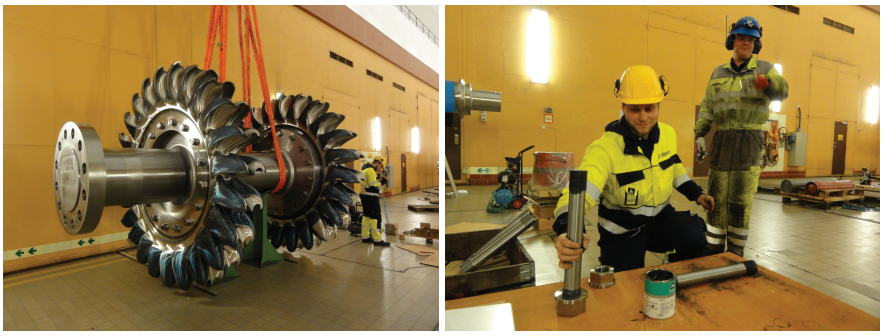
2016

Dissertation date: 02.05.2016

# Scientific environment

This thesis is funded through the *Industrial PHD Scheme*, an initiative by the Research Council of Norway which aims to build relations between academia and industry. Under this scheme, the supervison is shared by an academic institution and a company within a relevant industry. In this case, these are the University of Bergen and BKK Production AS. The funding is shared equally between the Reasearch Council of Norway and BKK Production AS.


The Research Council of Norway



Above we see a Pelton-turbine runner at one of BKK's power plants being bolted onto the runner shaft. The author can be seen in the foreground of the right image.

BKK[1] is an energy and infrastructure company based in Bergen. BKK owns 32 hydro-electric power-plants in western Norway, constituting BKK's core business. With an average annual production of 6.7 TWh, BKK is the fifth largest producer of electric power in Norway.

BKK is a technology driven company with expressed effort in contributing to innovation and technological development. This thesis is a part of that effort.

[1] *BKK* was originally an acronym for *Bergenhalvøens Kommunale Kraftselskap* (Muncipal Power Company of the Bergen-Peninsula) but has since been privatized with BKK as the official name.

# Acknowledgements

The author would like to thank the supervisors Alex C. Hoffmann[2], Jan S. Vaagen[3], Laszlo Csernai[4] and Arne Småbrekke[5] for valuable insights and discussion. Their contribution is greatly appreciated.

[2]Professor, University of Bergen, MAE
[3]Professor, University of Bergen, MAE
[4]Professor, University of Bergen, MAE
[5]Department Manager, BKK Production

# Précis

The underlying motivation of this work is to better understand the flow in turbines, aiming at a simplified design process and improved energy efficiency.

Based on this motivation there are many potential strategies and areas of interest to consider. For example testing of scale models, developing measurement techniques or the development of turbulence models. But in this thesis the focus has been on the development of numerical methods for solving nonlinear differential equations and the application to the governing equations in fluid dynamics – computational fluid dynamics (CFD). The reasoning behind this is the progressive improvement in computational hardware, giving ever increasing possibilities at predicting the behavior of fairly complex systems based on fundamental physical principles.

There are, however, large challenges within the field of CFD. Most notably issues with regard to the scale of the problems one would solve. Currently there are no methods which allow direct numerical simulation of a complete Pelton-turbine. A rough estimation of the Kolmogorov length scale in a typical Pelton-turbine yields a length scale on the order of $10^{-5}$ m in some regions. Given a computational domain of several cubic meters, an ordinary uniform grid discretization nearing this resolution would have on the order of $10^{23}$ grid points and require on the order of $10^{9}$ petabyte of storage. As a comparison, the Cray Titan supercomputer currently (2015) has less than one petabyte working memory and 40 petabyte of storage capacity.

There are several ways to tackle this problem. Sub-grid modeling, averaging theory, adaptive grids and particle based solvers to mention a few. We will study a different approach, which deals with how we represent a continuous medium as discrete quantities of digital data.

# Abstract

The focus in this thesis is the development and implementation of a new method for solving nonlinear differential equations on a grid.

The method's novelty lies in the way it represents continuously distributed variables by discrete information stored in a grid. The grid contains information about both the values and the values of the derivatives of the unknown functions at the grid points in the computational domain. With this method the derivatives are thus explicitly defined at each grid point rather than, as in conventional numerical schemes, implicitly given by the function values at the surrounding grid points.

By using piecewise polynomial interpolation, functions can be represented with an arbitrary order of continuity over the entire computational domain.

A mathematical framework is defined and the details of the polynomial interpolation is discussed, leading to the definition of particular sets of basis function which have especially favorable numerical properties for use with the current method.

It is shown how this method is used to formulate sets of differential equations as algebraic equations. With special focus on the Navier-Stokes equations.

A solution algorithm is developed for parallel computation on graphics processor units using C++ and OpenCL. Tests are performed on low cost hardware, and the implementation is developed to meet constraints in memory capacity and processing speed. The algorithm is a residual-minimizing type and is based on the finite element method. Test cases in 1D, 2D and 3D are numerically solved using the developed code. Two conference presentations are based on the method developed in this thesis, showing application to simulation of a rising bubble under gravity in three dimensions and a fluid flow in a Pelton turbine cup, also in three dimensions.

# Contents

# List of Figures

# Listings

# Chapter 1

# Introduction and Background

## 1.1 Fields and Discretization

Physical systems are described and explained as continuous fields in time and space. Even fluids and matter, which are composed of particles, are treated as fields in the macroscopic case. The behavior of these fields is governed by differential equations based on physical principles. In some cases, these differential equations may be solved analytically. However, in many cases of practical interest, the solutions must be found numerically.

This requires a way to represent a field numerically which can be stored in computer memory. An arbitrary quantity, continuously distributed in time and space, will in general not be accurately representable in a finite amount of computer memory. Even though matter and fluids are in reality composed of particles, the sheer number of particles prohibits individual treatment in the macroscopic case.

In numerical computing several different approaches are used to represent a continuously distributed variable. We will divide the most common approaches into *grid discretization*, *particle systems* and *function approximation*. Combinations of these are also possible.

## 1.2 Grid Discretization

A grid is a division of space and time into individual segments, each representing an interval or point in space and time. The shape of these segments may be defined so that the grid fits well to a particular geometric region. Figure 1.1 shows an example of a two dimensional grid adjusted to fit with a nonlinear geometry.[1]

[1] In Figure 1.1 the grid-cell is defined to be a region surrounded by a set of grid-points. This definition is used throughout the thesis. Other definitions of the grid-cell also exists. One may, for example, define the grid points to be in the center of each grid-cell.

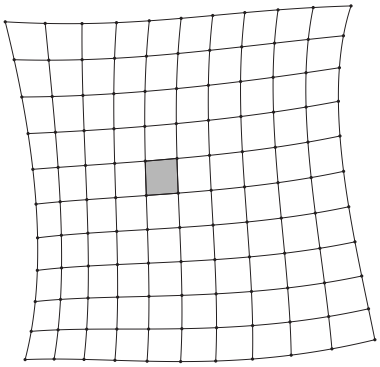


Figure 1.1: This figure shows a two dimensional region discretized by a grid. The line intersections are called grid-points and the enclosed individual regions are called grid-cells (one grid-cell is grayed out as an illustration)

## 1.3 Particle Based Discretization

In this case, the field is approximated by a group of particles. The particles are not ordered into a grid, but each has a definite position. Polyhedral grid-cells may be constructed based on the distribution of the particles, but more common is the use of a smoothing function, where the influence of different particles at a specific position is weighted based on the distance from the particle to this position. In the CFD context, *Smoothed Particle Hydrodynamics* (see [6] for further details) is currently state of the art within particle based methods.

## 1.4 Function Approximation

A field may be approximated by a weighted sum of a set of basis functions, defined on the region of interest. This may be very efficient if the basis functions also are approximate solutions to the equations governing the behavior of the field.[2] The *Spectral Method* is an example of this approach which is commonly used in fluid dynamics (see [5] for details).

## 1.5 Continuity Based Approach

The method developed in this thesis is a combination of grid-discretization and function approximation. The field in each grid-cell is described by a function approximation, not unlike a spectral element approach. But in the current approach the information of the derivatives, up to a given order, is explicitly given at each grid-point and define the order of continuity of the grid (see Chapter 2 for details).[3] By examining Figure

[2] For example, when solving a problem with oscillating or wave-like properties, one might benefit from using wave functions to approximate the solution rather than, say, polynomials.

[3] The current approach shares some of the qualities of the Cubic Interpolated Propagation (CIP) method [7] (an alternative version of the CIP acronym is used here), which includes the gradient of unknown quantities as a free parameter. The CIP method is a third order method used successfully, for example, to simulate acoustic

1.2, we see that a better approximation can be produced, using the same quantity of data, by increasing the order of continuity instead of the density of grid-points. This provides a motivation for developing this approach. Additionally we may observe that, for low orders of continuity, the position of the grid-points have a significant effect on the accuracy of the approximation. But for higher orders of continuity, this dependency is not as strong.

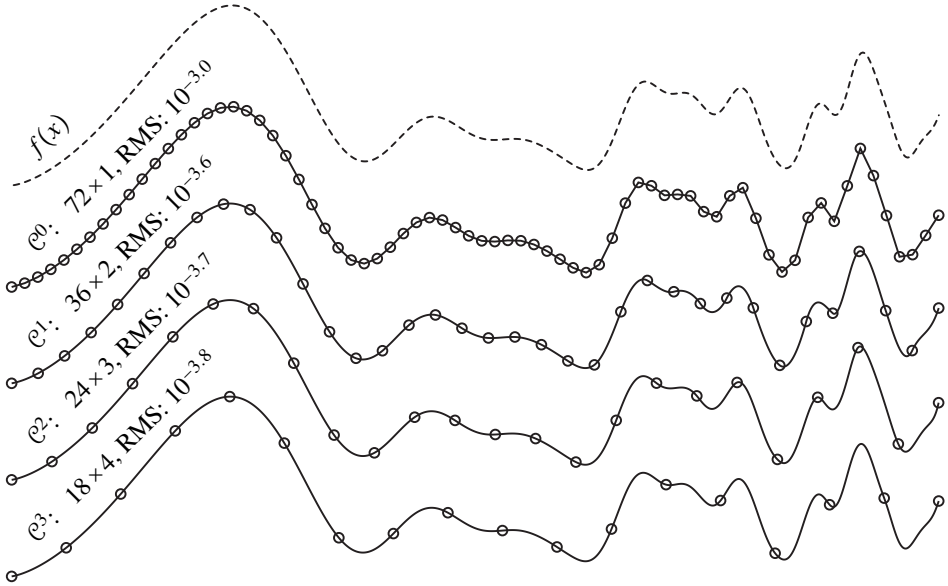



Figure 1.2: This figure shows four different approximations of a function, $f(x)$. Each approximation requires the same amount of floating point numbers and approximately the same number of operations to compute. The $\mathcal{C}^0$ continuous approximation has 72 grid-points where a single grid value is stored, while the $\mathcal{C}^1$ continuous approximation has 36 grid-points where both the function value and the derivative is stored giving 72 floating point numbers in total. Further, the $\mathcal{C}^2$ and $\mathcal{C}^3$ continuous approximations uses $3\times36$ and $4\times18$ floating point numbers, respectively. The root-mean-square (RMS) deviation from the original function, $f(x)$, is given and shows a decreasing trend as the order of continuity increases.

wave propagation.

# Chapter 2

# Notation and Mathematical Framework

## 2.1 Notation

Square brackets will be used to identify components in matrices. A component in a two-dimensional matrix, $\mathbf{A}$, will thus be referred to as $\mathbf{A}[r,c]$, where $r$ is the row index and $c$ is the column index. Indices in an $R \times C$ matrix are defined to go from 0 to $R-1$ (rows) and 0 to $C-1$ (columns). If $C = 1$, then the matrix may be referred to as a column vector and if $R = 1$, then the matrix may be referred to as a row-vector. $\mathbf{A}[:,c]$ refers to the $c$'th column and $\mathbf{A}[r,:]$ refers to the $r$'th row. A higher dimensional matrix may be written equivalently as a *matrix of matrices*. For example a four-dimensional matrix, $\mathbf{Q}$, where $\mathbf{Q}[r,c][\tau,\nu] \stackrel{\text{def}}{=} \mathbf{Q}[r,c,\tau,\nu]$. Square brackets are also used for closed continuous intervals. If a matrix, $\mathbf{A}$, is square and nonsingular, its inverse will be written as $\mathbf{A}^{-1}$. The transpose of a two dimensional matrix (or a tensor), $\mathbf{A}$, will be written $\mathbf{A}^T$. The *pseudo-inverse* is defined in (2.1):

$$\mathbf{A}^+ \stackrel{\text{def}}{=} \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T \tag{2.1}$$

Mapping of indices from multiple index form to single index form will, unless otherwise stated, be on the form $a = b + cD$, where $D$ is a positive integer and $b,c \in \{0,\ldots,D-1\}$ and $a \in \{0,\ldots,D^2-1\}$. The indices $b$ and $c$ may also be single index forms of other index tuples, in which case the mapping will recursively follow the given form.

## 2.2 Order of Continuity

Consider a discretization of a function, $f(x)$, on a grid. Let the matrix $\mathbf{x}$ be composed of the positions of a set of *grid-points*. The grid-points are assumed to be uniformly placed with a separation of unity, i.e. $\mathbf{x}[k+1] - \mathbf{x}[k] = 1$. Let the value of the function, $f(x)$, and its derivatives up to, and including, the $(\Omega-1)$'th order be explicitly defined for each grid-point in terms of the matrix, $\mathbf{F}$, with components given by (2.2):

$$a_\alpha \mathbf{F}[k,\alpha] \stackrel{\text{def}}{=} \left.\frac{\partial^\alpha f(x)}{\partial x^\alpha}\right|_{x=\mathbf{x}[k]} \tag{2.2}$$

where the index $k$ identifies the grid-point and $\alpha \in \{0,\ldots,\Omega-1\}$. The discretization is then by definition continuous and has continuous derivatives up to $(\Omega-1)$'th order at

the grid-points $\mathbf{x}[k]$. This is referred to as $\mathcal{C}^{\Omega-1}$ continuity. The coefficient $a_\alpha$ is used as a normalization constant. This will allow us to scale the values stored in the grid, avoiding large variations in their numerical value (see Chapter 2.4.2).

## 2.3 Grid Structure

Consider the interval between two grid-points, $k$ and $k+1$, constituting a one-dimensional *grid-cell*.

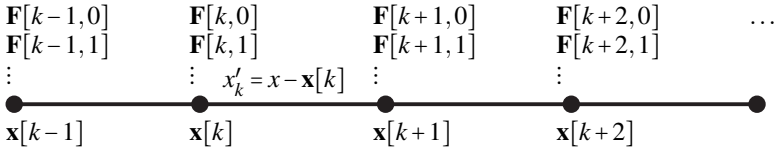



Figure 2.1: Uniformly spaced grid-points in one dimension. $\mathbf{F}[k]$ contains the grid components corresponding to the function, $f$, at the $k$'th grid-point. The sub-grid coordinate, $x'_k$, is defined on the interval $x \in [\mathbf{x}[k], \mathbf{x}[k+1]]$.

Further, let $\mathbf{b}_N(x)$ be a column vector of polynomial basis functions with $\deg(\mathbf{b}_N) < N$ on the interval $x \in [0,1]$, where the total number of basis functions in the set is $N$ and $\mathbf{b}_N(x)[n]$ refers to each function. Let the *sub-grid coordinate*, $x'_k$, be defined by (2.3):

$$x'_k = x - \mathbf{x}[k] \tag{2.3}$$

The function, $f(x)$, may then be approximated by a weighted sum of these basis functions, as shown in (2.4), where $\boldsymbol{\kappa}_k$ is a column vector with the weight of each basis function.

$$f(x) = \mathbf{b}_{2\Omega}^T(x'_k)\,\boldsymbol{\kappa}_k + \mathcal{O}\left(x_k'^{2\Omega}\right), \quad x' \in [0,1] \tag{2.4}$$

We then require, in (2.5), that the approximation conforms with the order of continuity given by (2.2):

$$\left.\frac{\partial^\alpha \mathbf{b}_{2\Omega}^T(x)\,\boldsymbol{\kappa}_k}{\partial x^\alpha}\right|_{x=0} = a_\alpha \mathbf{F}[k,\alpha] \tag{2.5a}$$

$$\left.\frac{\partial^\alpha \mathbf{b}_{2\Omega}^T(x)\,\boldsymbol{\kappa}_k}{\partial x^\alpha}\right|_{x=1} = a_\alpha \mathbf{F}[k+1,\alpha] \tag{2.5b}$$

We will use the term *component* when referring to individual entries in grid-points. We will let *boundary-components* refer to components given by the boundary conditions and *interior-components* refer to the remaining, unknown components. Further, we will use the term *grid-cell-component* refer to a component in a particular grid-cell (without regard to its interior/boundary status). Note that this naming convention then implies that the sets of interior and boundary components does not intersect, while a set of grid-cell-components may intersect with other grid-cells as well as with interior and boundary components (see Figure 2.2).

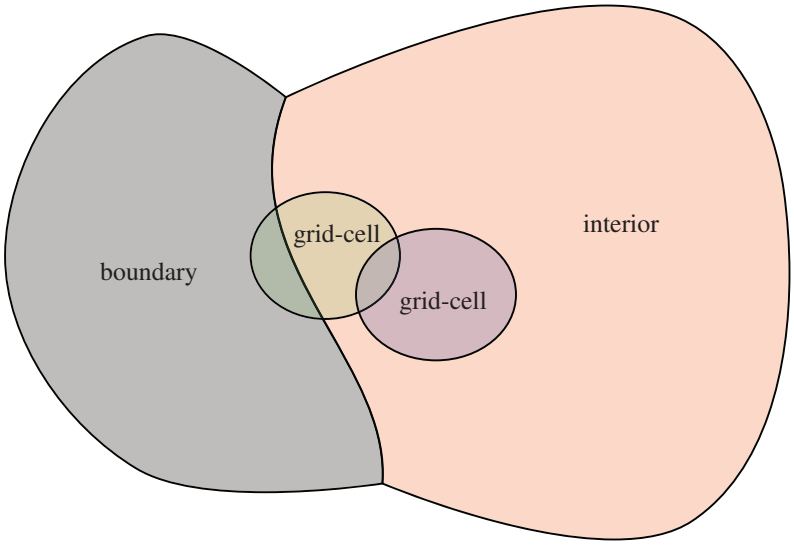



Figure 2.2: This figure shows a Venn diagram of the relations between the sets of interior, boundary and grid-cell components.

## 2.4 Choice of Basis functions

### 2.4.1 Polynomial Basis Functions and Conditioning

Any set of polynomial basis functions spanning the same polynomial vector space would give the same (analytical) approximation. However, floating point errors differ greatly depending on the choice of basis functions. Consider (2.5) arranged into the matrix equation (2.6), where the matrix product $\mathbf{B}\kappa_k$ is equivalent to the summation on the left hand side of (2.5) and the matrix $\mathbf{A}$ is diagonal, consisting of the constants $a_\alpha$.

$$\underbrace{\begin{bmatrix} \mathbf{b}_{2\Omega}^T(x) & |_{x=0} \\ \frac{\partial}{\partial x}\mathbf{b}_{2\Omega}^T(x) & |_{x=0} \\ \vdots & \\ \mathbf{b}_{2\Omega}^T(x) & |_{x=1} \\ \frac{\partial}{\partial x}\mathbf{b}_{2\Omega}^T(x) & |_{x=1} \\ \vdots & \end{bmatrix}}_{\mathbf{B}} \kappa_k = \underbrace{\begin{bmatrix} a_0 & 0 & \dots & & & \\ 0 & a_1 & & & & \\ \vdots & & \ddots & & & \\ & & & a_0 & 0 & \dots \\ & & & 0 & a_1 & \\ & & & \vdots & & \ddots \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} \mathbf{F}[k,0] \\ \mathbf{F}[k,1] \\ \vdots \\ \mathbf{F}[k+1,0] \\ \mathbf{F}[k+1,1] \\ \vdots \end{bmatrix}}_{\mathbf{f}_k} \tag{2.6}$$

The coefficients, $\kappa_k$, are given by $\kappa_k = \mathbf{B}^{-1}\mathbf{A}\mathbf{f}_k$. The *condition number*, $\text{cond}(\mathbf{B})$, defined in Eq.(2.7), gives an estimate of the relative numerical accuracy of the matrix product, $\kappa_k = \mathbf{B}^{-1}\mathbf{A}\mathbf{f}_k$ – i.e.: the numerical accuracy of the weighting of the basis functions. This is important because, as a high polynomial order gives high accuracy in exact

arithmetic, the numerical accuracy tends to give the opposite effect.

$$\mathrm{cond}\left(\mathbf{B}\right) \stackrel{\mathrm{def}}{=} \frac{\sigma_{\max}\left(\mathbf{B}\right)}{\sigma_{\min}\left(\mathbf{B}\right)} \tag{2.7}$$

In Eq.(2.7), $\sigma_{\max}\left(\mathbf{B}\right)$ is the largest singular value of $\mathbf{B}$, and $\sigma_{\min}\left(\mathbf{B}\right)$ is the smallest singular value of $\mathbf{B}$. When numerically solving the linear system, $\mathbf{A}\mathbf{f}_k = \mathbf{B}\kappa_k$, using floating point numbers with machine precision, $\varepsilon_m$, an error of order $\mathcal{O}\left(\varepsilon_m \mathrm{cond}\left(\mathbf{B}\right)\right)$ should be expected.[1] It is clear that the condition number of $\mathbf{B}$ is important in the relation between the numerical values stored in the grid ($\mathbf{F}$) and the resulting approximation of $f(x)$. By examining Table 2.1 we see that condition numbers rise exponentially with an increase of the order of continuity.

| | cond $(\mathbf{B})$ | | |
|---|---|---|---|
| $\Omega$ | monomials | Bernstein | Hermite |
| 2 | $2.4 \times 10^{1}$ | $6.2 \times 10^{0}$ | $5.2 \times 10^{2}$ |
| 3 | $7.6 \times 10^{2}$ | $8.7 \times 10^{1}$ | $4.7 \times 10^{5}$ |
| 4 | $4.8 \times 10^{4}$ | $1.9 \times 10^{3}$ | $1.0 \times 10^{9}$ |
| 5 | $4.4 \times 10^{6}$ | $5.8 \times 10^{4}$ | $3.4 \times 10^{12}$ |
| 6 | $5.5 \times 10^{8}$ | $2.2 \times 10^{6}$ | $2.9 \times 10^{16}$ |

Table 2.1: Estimated floating point errors for different values of $\Omega$ for a selection of common basis function sets. The matrix, $\mathbf{B}$, is defined in (2.6) and the condition number, $\mathrm{cond}\left(\mathbf{B}\right)$, is defined by (2.7). This is the estimated precision of the numerical computation of the quantity $\mathbf{B}^{-1}\mathbf{A}\mathbf{f}_k$ relative to the machine precision, $\varepsilon_m$. The first column shows different values of $\Omega$, corresponding to $\mathcal{C}^{\Omega-1}$ continuity. The second column shows expected loss in precision when the monomial basis functions, $x^n$ for $n \in \{0, \ldots, 2\Omega - 1\}$, are used to construct the matrix $\mathbf{B}$. The third column shows the expected loss in precision when the Bernstein polynomial basis functions are used to construct the matrix $\mathbf{B}$. The fourth column shows the expected loss in precision when the Hermite polynomial basis functions (translated to the interval $[0, 1]$) are used to construct the matrix $\mathbf{B}$.

### 2.4.2 Optimal Set of Basis Functions

The best possible set of basis functions in this respect are those that give the smallest possible condition number, i.e. $\mathrm{cond}\left(\mathbf{B}\right) = 1$. This is the case, for example, with $\mathbf{B} = \mathbf{I}$. We will therefore require that $\mathbf{B} = \mathbf{I}$. Since the the basis functions are polynomials of degree $2\Omega - 1$, this requirement together with the form of $\mathbf{B}$ given in (2.7) yields the

[1] The reader may refer to Trefethen and Bau [8, pg. 95] for a more detailed explanation of the condition number and numerical accuracy of linear equation systems.

linear system (2.8):

$$\begin{bmatrix} 1 & x & x^2 & x^3 & \dots & x^{2\Omega-1} & |_{x=0} \\ 0 & 1 & 2x & 3x^2 & \dots & & |_{x=0} \\ 0 & 0 & 2 & 6x & \dots & & |_{x=0} \\ \vdots & & & & & & \\ 1 & x & x^2 & x^3 & \dots & & |_{x=1} \\ 0 & 1 & 2x & 3x^2 & \dots & & |_{x=1} \\ 0 & 0 & 2 & 6x & \dots & & |_{x=1} \\ \vdots & & & & & & \end{bmatrix} \mathbf{D} = \mathbf{I} \tag{2.8}$$

The matrix, $\mathbf{D}$, then contains the coefficients for each power of $x$ in each basis function. As a result the basis functions are

$$\mathbf{b}_{2\Omega}(x) = \begin{bmatrix} 1 \, x \, x^2 \, \dots \, x^{2\Omega-1} \end{bmatrix} \mathbf{D}, \quad \text{with} \quad \mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots \\ 0 & 1 & 0 & 0 & \dots \\ 0 & 0 & 2 & 0 & \dots \\ \vdots & & & & \\ 1 & 1 & 1 & 1 & \dots \\ 0 & 1 & 2 & 3 & \dots \\ 0 & 0 & 2 & 6 & \dots \\ \vdots & & & & \end{bmatrix}^{-1} \tag{2.9}$$

The inverse of the matrix, $\mathbf{D}$, is found with exact arithmetic without loss of precision. By studying the basis functions, $\mathbf{b}_{2\Omega}(x)$, carefully, the following closed form presents itself (2.10):

$$\mathbf{b}_{2\Omega}[\alpha](x) \propto (1-x)^{\Omega} x^{\alpha} \left( \sum_{n=0}^{\Omega-1-\alpha} \mathbf{c}_{\Omega}[n] x^n \right) \tag{2.10a}$$

$$\mathbf{b}_{2\Omega}[\Omega+\alpha](x) = (-1)^{\alpha} \mathbf{b}_{2\Omega}[\alpha](1-x) \tag{2.10b}$$

$$\alpha \in \{0, \Omega-1\} \tag{2.10c}$$

where the set of coefficients, $\mathbf{c}_N$, is defined in terms of the recursive relations (2.11). Table 2.4.2 shows some computed coefficients.

$$\mathbf{c}_1[0] = 1 \tag{2.11a}$$

$$\mathbf{c}_N[n] = \sum_{m=0}^{n} \mathbf{c}_{N-1}[m], \quad n < N-1 \tag{2.11b}$$

$$\mathbf{c}_N[N-1] = 2\mathbf{c}_N[N-2] \tag{2.11c}$$

| $N$ | $\mathbf{c}_N[0]$ | $\mathbf{c}_N[1]$ | $\mathbf{c}_N[2]$ | $\mathbf{c}_N[3]$ | $\mathbf{c}_N[4]$ |
|---|---|---|---|---|---|
| 1 | 1 | | | | |
| 2 | 1 | 2 | | | |
| 3 | 1 | 3 | 6 | | |
| 4 | 1 | 4 | 10 | 20 | |
| 5 | 1 | 5 | 15 | 35 | 70 |

Table 2.2: This table shows some computed values of the coefficients defined in (2.11).

### 2.4.3 Normalization

The final degree of freedom in the choice of basis functions lies in the constant, $a_\alpha$, which we determine by normalizing the basis functions (2.12a):

$$\hat{\mathbf{b}}_N[\alpha](x) \stackrel{\text{def}}{=} \frac{\mathbf{b}_N[\alpha](x)}{a_\alpha} \tag{2.12a}$$

$$a_\alpha \stackrel{\text{def}}{=} 2\left|\int_{x=0}^{1} \mathbf{b}_N[\alpha](x)\mathrm{d}x\right| \tag{2.12b}$$

thus

$$\left|\int_{x=0}^{1} \hat{\mathbf{b}}_N[\alpha](x)\mathrm{d}x\right| = \frac{1}{2} \tag{2.13}$$

Importantly, we have a symmetry relation between $\alpha$ and $\Omega+\alpha$ which gives us, pairwise, the same normalization constant for these basis functions. This ensures that the normalization is consistent over the entire grid. By choosing to scale the functions to the constant $1/2$ we end up with $a_0 = 1$ for all orders of continuity. The approximation given in (2.4) may now be given directly in terms of the grid values and the basis functions as shown in (2.14):

$$f(x) = \hat{\mathbf{b}}_{2\Omega}^T(x'_k)\mathbf{f} + \mathcal{O}\left(x_k'^{2\Omega}\right), \quad x'_k \in [0,1] \tag{2.14}$$

Due to the special properties of the basis functions, the weighting factors, $\kappa_k$, are no longer needed; the grid-cell-components now *are* the weights. Table 2.3 shows the resulting normalized basis functions for different orders of continuity. Figures 2.3 - 2.5 shows plots of the normalized basis functions, $\hat{\mathbf{b}}_4$, $\hat{\mathbf{b}}_6$ and $\hat{\mathbf{b}}_8$. The symmetry/anti-symmetry relations given in (2.10) can be observed.

| | | |
|---|---|---|
| $\Omega = 2$ | $\hat{\mathbf{b}}_4[0](x)$ | $2x^3 - 3x^2 + 1$ |
| $(\mathcal{C}^1)$ | $\hat{\mathbf{b}}_4[1](x)$ | $6x^3 - 12x^2 + 6x$ |
| | $\hat{\mathbf{b}}_4[2](x)$ | $3x^2 - 2x^3$ |
| | $\hat{\mathbf{b}}_4[3](x)$ | $6x^3 - 6x^2$ |
| $\Omega = 3$ | $\hat{\mathbf{b}}_6[0](x)$ | $-6x^5 + 15x^4 - 10x^3 + 1$ |
| $(\mathcal{C}^2)$ | $\hat{\mathbf{b}}_6[1](x)$ | $-15x^5 + 40x^4 - 30x^3 + 5x$ |
| | $\hat{\mathbf{b}}_6[2](x)$ | $-30x^5 + 90x^4 - 90x^3 + 30x^2$ |
| | $\hat{\mathbf{b}}_6[3](x)$ | $6x^5 - 15x^4 + 10x^3$ |
| | $\hat{\mathbf{b}}_6[4](x)$ | $-15x^5 + 35x^4 - 20x^3$ |
| | $\hat{\mathbf{b}}_6[5](x)$ | $30x^5 - 60x^4 + 30x^3$ |
| $\Omega = 4$ | $\hat{\mathbf{b}}_8[0](x)$ | $20x^7 - 70x^6 + 84x^5 - 35x^4 + 1$ |
| $(\mathcal{C}^3)$ | $\hat{\mathbf{b}}_8[1](x)$ | $\frac{140}{3}x^7 - 168x^6 + 210x^5 - \frac{280}{3}x^4 + \frac{14}{3}x$ |
| | $\hat{\mathbf{b}}_8[2](x)$ | $84x^7 - 315x^6 + 420x^5 - 210x^4 + 21x^2$ |
| | $\hat{\mathbf{b}}_8[3](x)$ | $140x^7 - 560x^6 + 840x^5 - 560x^4 + 140x^3$ |
| | $\hat{\mathbf{b}}_8[4](x)$ | $-20x^7 + 70x^6 - 84x^5 + 35x^4$ |
| | $\hat{\mathbf{b}}_8[5](x)$ | $\frac{140}{3}x^7 - \frac{476}{3}x^6 + 182x^5 - 70x^4$ |
| | $\hat{\mathbf{b}}_8[6](x)$ | $-84x^7 + 273x^6 - 294x^5 + 105x^4$ |
| | $\hat{\mathbf{b}}_8[7](x)$ | $140x^7 - 420x^6 + 420x^5 - 140x^4$ |
| $\Omega = 5$ | $\hat{\mathbf{b}}_{10}[0](x)$ | $-70x^9 + 315x^8 - 540x^7 + 420x^6 - 126x^5 + 1$ |
| $(\mathcal{C}^4)$ | $\hat{\mathbf{b}}_{10}[1](x)$ | $-\frac{315}{2}x^9 + 720x^8 - 1260x^7 + 1008x^6 - 315x^5 + \frac{9}{2}x$ |
| | $\hat{\mathbf{b}}_{10}[2](x)$ | $-270x^9 + 1260x^8 - 2268x^7 + 1890x^6 - 630x^5 + 18x^2$ |
| | $\hat{\mathbf{b}}_{10}[3](x)$ | $-420x^9 + 2016x^8 - 3780x^7 + 3360x^6 - 1260x^5 + 84x^3$ |
| | $\hat{\mathbf{b}}_{10}[4](x)$ | $-630x^9 + 3150x^8 - 6300x^7 + 6300x^6 - 3150x^5 + 630x^4$ |
| | $\hat{\mathbf{b}}_{10}[5](x)$ | $70x^9 - 315x^8 + 540x^7 - 420x^6 + 126x^5$ |
| | $\hat{\mathbf{b}}_{10}[6](x)$ | $-\frac{315}{2}x^9 + \frac{1395}{2}x^8 - 1170x^7 + 882x^6 - 252x^5$ |
| | $\hat{\mathbf{b}}_{10}[7](x)$ | $270x^9 - 1170x^8 + 1908x^7 - 1386x^6 + 378x^5$ |
| | $\hat{\mathbf{b}}_{10}[8](x)$ | $-420x^9 + 1764x^8 - 2772x^7 + 1932x^6 - 504x^5$ |
| | $\hat{\mathbf{b}}_{10}[9](x)$ | $630x^9 - 2520x^8 + 3780x^7 - 2520x^6 + 630x^5$ |
| $\Omega = 6$ | $\hat{\mathbf{b}}_{12}[0](x)$ | $252x^{11} - 1386x^{10} + 3080x^9 - 3465x^8 + 1980x^7 - 462x^6 + 1$ |
| $\left(\mathcal{C}^5\right)$ | $\hat{\mathbf{b}}_{12}[1](x)$ | $\frac{2772}{5}x^{11} - 3080x^{10} + 6930x^9 - 7920x^8 + 4620x^7 - \frac{5544}{5}x^6 + \frac{22}{5}x$ |
| | $\hat{\mathbf{b}}_{12}[2](x)$ | $924x^{11} - \frac{10395}{2}x^{10} + 11880x^9 - 13860x^8 + 8316x^7 - 2079x^6 + \frac{33}{2}x^2$ |
| | $\hat{\mathbf{b}}_{12}[3](x)$ | $1386x^{11} - 7920x^{10} + 18480x^9 - 22176x^8 + 13860x^7 - 3696x^6 + 66x^3$ |
| | $\hat{\mathbf{b}}_{12}[4](x)$ | $1980x^{11} - 11550x^{10} + 27720x^9 - 34650x^8 + 23100x^7 - 6930x^6 + 330x^4$ |
| | $\hat{\mathbf{b}}_{12}[5](x)$ | $2772x^{11} - 16632x^{10} + 41580x^9 - 55440x^8 + 41580x^7 - 16632x^6 + 2772x^5$ |
| | $\hat{\mathbf{b}}_{12}[6](x)$ | $-252x^{11} + 1386x^{10} - 3080x^9 + 3465x^8 - 1980x^7 + 462x^6$ |
| | $\hat{\mathbf{b}}_{12}[7](x)$ | $\frac{2772}{5}x^{11} - \frac{15092}{5}x^{10} + 6622x^9 - 7326x^8 + 4092x^7 - 924x^6$ |
| | $\hat{\mathbf{b}}_{12}[8](x)$ | $-924x^{11} + \frac{9933}{2}x^{10} - 10725x^9 + \frac{23265}{2}x^8 - 6336x^7 + 1386x^6$ |
| | $\hat{\mathbf{b}}_{12}[9](x)$ | $1386x^{11} - 7326x^{10} + 15510x^9 - 16434x^8 + 8712x^7 - 1848x^6$ |
| | $\hat{\mathbf{b}}_{12}[10](x)$ | $-1980x^{11} + 10230x^{10} - 21120x^9 + 21780x^8 - 11220x^7 + 2310x^6$ |
| | $\hat{\mathbf{b}}_{12}[11](x)$ | $2772x^{11} - 13860x^{10} + 27720x^9 - 27720x^8 + 13860x^7 - 2772x^6$ |

Table 2.3: This table shows normalized basis functions for different orders of continuity. The functions are normalized to the constant value of $1/2$ over the interval $[0,1]$. Note that in an implementation it is beneficial to factorize the polynomials to reduce round-off errors occurring when subtracting one large number from another large number.

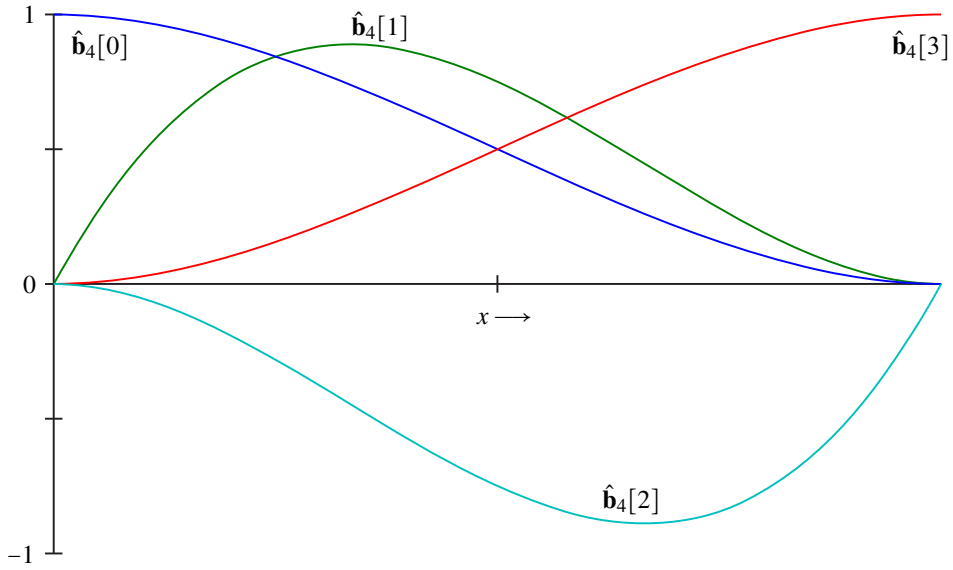



Figure 2.3: This figure shows a plot of the normalized basis functions for $\Omega = 2$ in the interval $x \in [0, 1]$.

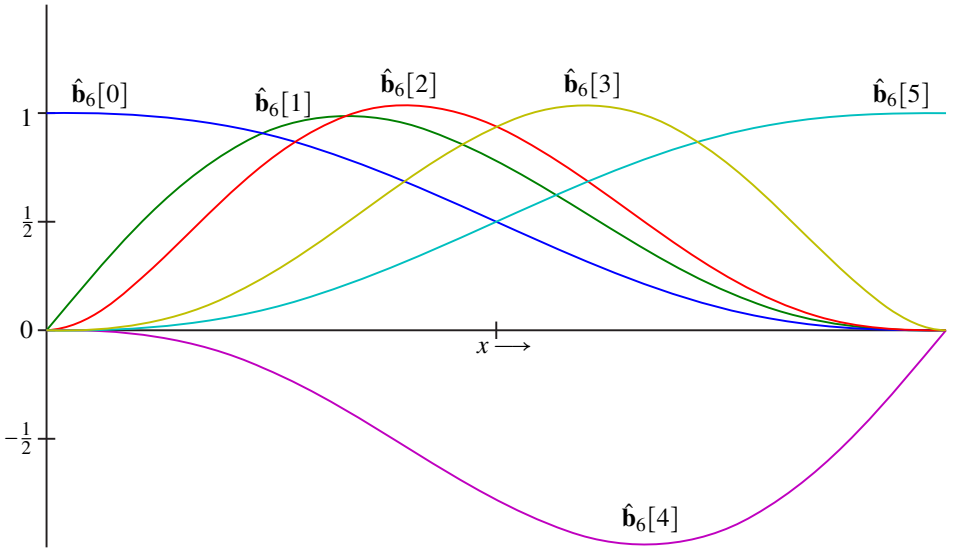



Figure 2.4: This figure shows a plot of the normalized basis functions for $\Omega = 3$ in the interval $x \in [0, 1]$.

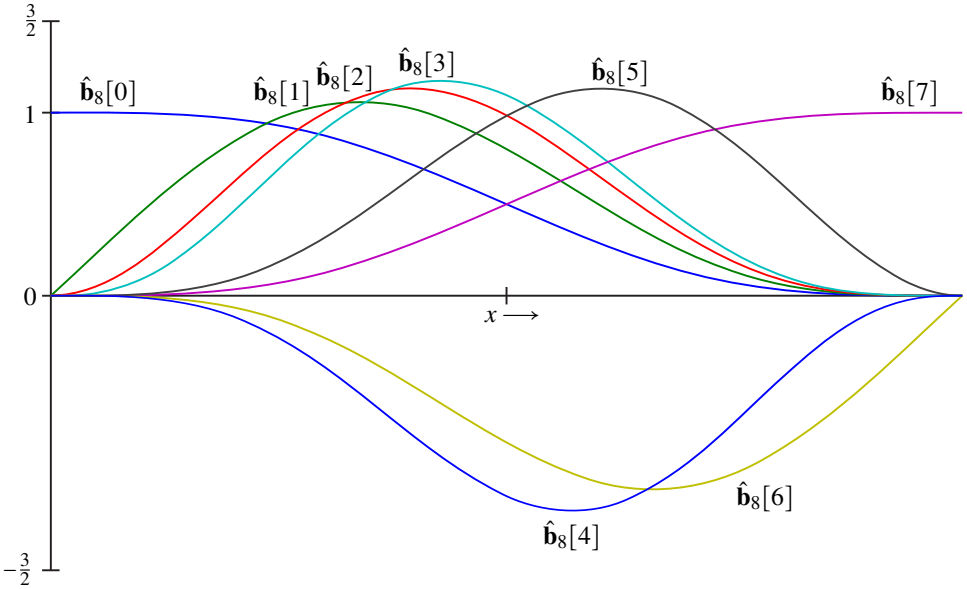



Figure 2.5: This figure shows a plot of the normalized basis functions for $\Omega = 4$ in the interval $x \in [0,1]$.

## 2.5 Generalization to higher dimensions

### 2.5.1 Basis Functions

The one-dimensional grid is generalized into higher dimension by using the product of different basis functions corresponding to each spatial or temporal dimension. We will focus on the three spatial dimensions with the three-dimensional basis functions defined in (2.15). The general principles remain the same as in the one-dimensional case. The main task here is to consistently map the components of the grid to the three dimensional basis functions. Note that the order of continuity may be different for different spatial directions, and for the sake of brevity we omit the indication of continuity order for the multi-dimensional basis functions.

$$\hat{\hat{\mathbf{b}}}[n](x,y,z) \stackrel{\text{def}}{=} \hat{\mathbf{b}}_{2\Omega_x}[n_x](x)\hat{\mathbf{b}}_{2\Omega_y}[n_y](y)\hat{\mathbf{b}}_{2\Omega_z}[n_z](z)\,, \quad n = n_x + 2\Omega_x(n_y + 2\Omega_y n_z) \tag{2.15}$$

### 2.5.2 Grid Structure

As in the one-dimensional case we assume a uniform grid with an interval spacing between the grid-points equal to unity. The primed coordinates, defined in (2.16), are cell-specific coordinates on the interval $[0,1]^3$.

$$x'_k \stackrel{\text{def}}{=} x - \mathbf{x}[k]\,, \quad y'_l \stackrel{\text{def}}{=} y - \mathbf{y}[l]\,, \quad z'_m \stackrel{\text{def}}{=} z - \mathbf{z}[m]\,, \quad (x'_k, y'_l, z'_m) \in [0,1]^3 \tag{2.16}$$

### 2.5.3 Grid-cell Approximation

The grid now contains information about derivatives, including mixed derivatives, up to a given order in each spatial direction. The three dimensional version of the grid definition is given in (2.17), and the three-dimensional function approximation is given in (2.19) (corresponding to the one dimensional definitions (2.2) and (2.14)).

$$\left.\frac{\partial^\alpha \partial^\beta \partial^\gamma f(x,y,z)}{\partial x^\alpha \partial y^\beta \partial z^\gamma}\right|_{x=\mathbf{x}[k],y=\mathbf{y}[l],z=\mathbf{z}[l]} \stackrel{\text{def}}{=} a_\alpha b_\beta c_\gamma \mathbf{F}[k,l,m][\alpha,\beta,\gamma] \tag{2.17}$$

$$\alpha \in \{0,\ldots,\Omega_x-1\},\ \ \beta \in \{0,\ldots,\Omega_y-1\},\ \ \gamma \in \{0,\ldots,\Omega_z-1\} \tag{2.18}$$

$$f(x,y,z) = \hat{\hat{\mathbf{b}}}^T(x'_k,y'_l,z'_m)\mathbf{f}_{k,l,m} + \mathcal{O}\left(x_k'^{2\Omega_x} + y_l'^{2\Omega_y} + z_m'^{2\Omega_z}\right) \tag{2.19}$$

All the grid data belonging to a particular grid-cell is stored in the column vector, $\mathbf{f}_{k,l,m}$, defined in (2.20). This is the three-dimensional generalization of $\mathbf{f}_k$ which was used in the one-dimensional case (2.6).

$$\mathbf{f}_{k,l,m}[\Omega_x\Omega_y\Omega_z i+\tau] \stackrel{\text{def}}{=} \mathbf{F}[k+i_x,l+i_y,m+i_z][\alpha,\beta,\gamma] \tag{2.20}$$

where

$$i = i_x + 2(i_y + 2i_z) \ \text{ and } \ \tau = \alpha + \Omega_x(\beta + \Omega_y\gamma). \tag{2.21}$$

Note that the index mappings, in (2.20) and (2.21), must agree with the definition of the basis functions for three dimensions, (2.15), in order to express the approximation as a single product of vectors shown in (2.19).

# Chapter 3

# Algebraic Formulation of Differential Equations

## 3.1 Governing Equations and Grid Scale

As an example we will use the time-dependent Navier–Stokes equations for incompressible flow for mass conservation and momentum conservation (differential form) without gravity. The dimensionless Navier–Stokes equations are commonly expressed in a reference frame where a unit of length corresponds to a characteristic physical length, $x_0$, of the system. We will let this characteristic length scale correspond to the size of our computational domain in the spatial $x$-direction. Given $K \times L \times M$ grid points, placed at unit intervals, the corresponding size of the grid then is $K-1$. Figure 3.1 illustrates these relations. As (3.1) is expressed in the reference frame of the computational domain, spatial differentiation then generates a factor, $K-1 \stackrel{\text{def}}{=} \eta$, in the governing equations. In the continuity equation, (3.1a), these factors all cancel out since each term has exactly one spatial derivative. In the momentum equations, (3.1b-3.1d), the diffusion and time derivative terms will be scaled with $\eta$ and $\eta^{-1}$, respectively.

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{3.1a}$$

$$\eta^{-1}\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -\frac{\partial p}{\partial x} + \frac{\eta}{\mathrm{Re}}\nabla^2 u \tag{3.1b}$$

$$\eta^{-1}\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = -\frac{\partial p}{\partial y} + \frac{\eta}{\mathrm{Re}}\nabla^2 v \tag{3.1c}$$

$$\eta^{-1}\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = -\frac{\partial p}{\partial z} + \frac{\eta}{\mathrm{Re}}\nabla^2 w \tag{3.1d}$$

The scaling of the Navier – Stokes equations is further discussed in Appendix A, where independent scaling of the temporal dimension is also shown.

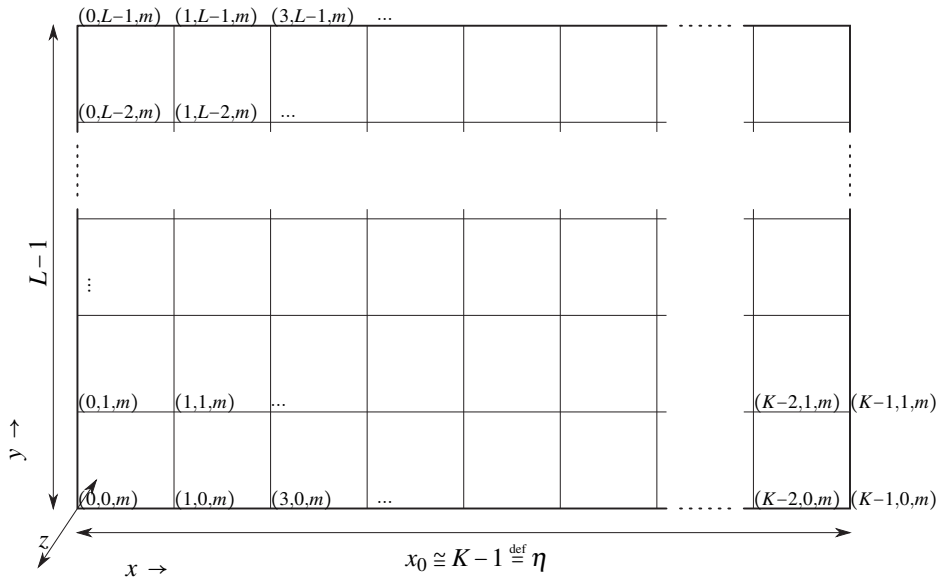



Figure 3.1: This figure shows the different scales the computational domain corresponds to. $x_0$ is the physical characteristic size which the Reynolds number is based upon. This length corresponds to $K-1$ in the computational domain.

## 3.2 Algebraic Form

First we will express the differential equations as algebraic equation systems defined for each grid-cell given by the indices $k,l$ and $m$. The grid-cell systems are coupled and this must be taken into account when solving for the equation for all the grid-components. The velocity and pressure are taken to be continuously distributed variables approximated as defined in (2.19), where $\mathbf{p}_{k,l,m}$ contains the pressure components for the grid-cell and $\mathbf{u}_{k,l,m}$, $\mathbf{v}_{k,l,m}$ and $\mathbf{w}_{k,l,m}$ contain the velocity components for the spatial $x,y$ and $z$ directions, respectively (see Chapter 2.3). The approximation is shown in (3.2) for all the fluid components.

$$p(x,y,z) \approx \hat{\hat{\mathbf{b}}}^T(x'_k,y'_l,z'_m)\mathbf{p}_{k,l,m} \tag{3.2a}$$

$$u(x,y,z) \approx \hat{\hat{\mathbf{b}}}^T(x'_k,y'_l,z'_m)\mathbf{u}_{k,l,m} \tag{3.2b}$$

$$v(x,y,z) \approx \hat{\hat{\mathbf{b}}}^T(x'_k,y'_l,z'_m)\mathbf{v}_{k,l,m} \tag{3.2c}$$

$$w(x,y,z) \approx \hat{\hat{\mathbf{b}}}^T(x'_k,y'_l,z'_m)\mathbf{w}_{k,l,m} \tag{3.2d}$$

Further, a short-hand notation for the derivatives of the basis functions is defined in (3.3):

$$\mathbf{b}_\mu \stackrel{\text{def}}{=} \frac{\partial}{\partial\mu}\hat{\hat{\mathbf{b}}}(x'_k,y'_l,z'_m), \quad \mathbf{b}_{\mu\mu} \stackrel{\text{def}}{=} \frac{\partial^2}{\partial\mu^2}\hat{\hat{\mathbf{b}}}(x'_k,y'_l,z'_m), \quad \ldots \tag{3.3}$$

By substituting the approximated pressure and velocity (3.2) into the Navier–Stokes equations (3.1) and applying the proper scaling we arrive at the algebraic form of the differential equations, (3.4), for each grid-cell. Note that the cell-indices, $k,l,m$ are ommitted in (3.4) for the sake of brevity.

$$\mathbf{b}_x^T\mathbf{u}+\mathbf{b}_y^T\mathbf{v}+\mathbf{b}_z^T\mathbf{w}=0 \tag{3.4a}$$

$$\frac{1}{\eta}\frac{\partial}{\partial t}\mathbf{b}^T\mathbf{u}+\left(\mathbf{b}^T\mathbf{u}\mathbf{b}_x^T+\mathbf{b}^T\mathbf{v}\mathbf{b}_y^T+\mathbf{b}^T\mathbf{w}\mathbf{b}_z^T\right)\mathbf{u}=-\mathbf{b}_x^T\mathbf{p}+\frac{\eta}{\mathrm{Re}}\left(\mathbf{b}_{xx}^T+\mathbf{b}_{yy}^T+\mathbf{b}_{zz}^T\right)\mathbf{u} \tag{3.4b}$$

$$\frac{1}{\eta}\frac{\partial}{\partial t}\mathbf{b}^T\mathbf{v}+\left(\mathbf{b}^T\mathbf{u}\mathbf{b}_x^T+\mathbf{b}^T\mathbf{v}\mathbf{b}_y^T+\mathbf{b}^T\mathbf{w}\mathbf{b}_z^T\right)\mathbf{v}=-\mathbf{b}_y^T\mathbf{p}+\frac{\eta}{\mathrm{Re}}\left(\mathbf{b}_{xx}^T+\mathbf{b}_{yy}^T+\mathbf{b}_{zz}^T\right)\mathbf{v} \tag{3.4c}$$

$$\frac{1}{\eta}\frac{\partial}{\partial t}\mathbf{b}^T\mathbf{w}+\left(\mathbf{b}^T\mathbf{u}\mathbf{b}_x^T+\mathbf{b}^T\mathbf{v}\mathbf{b}_y^T+\mathbf{b}^T\mathbf{w}\mathbf{b}_z^T\right)\mathbf{w}=-\mathbf{b}_z^T\mathbf{p}+\frac{\eta}{\mathrm{Re}}\left(\mathbf{b}_{xx}^T+\mathbf{b}_{yy}^T+\mathbf{b}_{zz}^T\right)\mathbf{w} \tag{3.4d}$$

## 3.3 Solution Algorithms

### 3.3.1 Linearization and Iterative Steady State Solution

In order to solve the nonlinear system of equations, (3.4), one possible approach is to linearize the system and solve iteratively. In the steady state case, the time derivatives are by definition zero, so the partial time derivative term is discarded. In (3.5) we collect the remaining nonlinear convection terms, and in (3.6) we collect the viscous stress terms.

$$\mathbf{m}_n^T\overset{\text{def}}{=}\mathbf{b}^T\mathbf{u}\mathbf{b}_x^T+\mathbf{b}^T\mathbf{v}\mathbf{b}_y^T+\mathbf{b}^T\mathbf{w}\mathbf{b}_z^T \tag{3.5}$$

$$\mathbf{s}^T\overset{\text{def}}{=}-\frac{\eta}{\mathrm{Re}}\left(\mathbf{b}_{xx}^T+\mathbf{b}_{yy}^T+\mathbf{b}_{zz}^T\right) \tag{3.6}$$

The equation set for each grid-cell can thus be written on matrix form as shown in (3.7) and (3.8).

$$\underbrace{\begin{bmatrix}0 & \mathbf{b}_x^T & \mathbf{b}_y^T & \mathbf{b}_z^T\\ \mathbf{b}_x^T & \mathbf{m}^T+\mathbf{s}^T & 0 & 0\\ \mathbf{b}_y^T & 0 & \mathbf{m}^T+\mathbf{s}^T & 0\\ \mathbf{b}_z^T & 0 & 0 & \mathbf{m}^T+\mathbf{s}^T\end{bmatrix}}_{\mathbf{A}_{k,l,m}}\underbrace{\begin{bmatrix}\mathbf{p}\\ \mathbf{u}\\ \mathbf{v}\\ \mathbf{w}\end{bmatrix}}_{\mathbf{x}_{k,l,m}}=\mathbf{0} \tag{3.7}$$

$$\mathbf{A}_{k,l,m}\mathbf{x}_{k,l,m}=\mathbf{0} \tag{3.8}$$

As shown in (3.7) we let $\mathbf{x}_{k,l,m}$ be constructed from the grid-cell components. The grid-cell systems, (3.8), appear to be homogeneous. However, boundary conditions have not yet been taken into account. All the grid-cell matrices, $\mathbf{A}_{k,l,m}$, are gathered into one matrix, $\Pi$, as shown in (3.9):

$$\begin{bmatrix}\mathbf{A}_{0,0,0} & 0 & \cdots & & \\ 0 & \mathbf{A}_{0,0,1} & & & \\ \vdots & & \ddots & & \\ & & & \mathbf{A}_{k,l,m} & \\ & & & & \ddots\end{bmatrix}\overset{\text{def}}{=}\Pi \tag{3.9}$$

We will let some of the grid-components be constant due to boundary conditions, and define the two column matrices, $\mathbf{g}$ and $\mathbf{c}$, containing all the non-boundary components

and the boundary components of the grid, respectively. Further, we will define two mapping matrices, $\mathcal{I}$ and $\mathcal{B}$ which map the interior and boundary components of the grid to the grid-cell components, $\mathbf{x}_{k,l,m}$ as shown in (3.10):

$$\begin{bmatrix} \mathbf{x}_{0,0,0} \\ \mathbf{x}_{0,0,1} \\ \vdots \\ \mathbf{x}_{k,l,m} \\ \vdots \end{bmatrix} = \mathcal{I}\mathbf{g} + \mathcal{B}\mathbf{c} \tag{3.10}$$

Each row in the mapping matrices, $\mathcal{I}$ and $\mathcal{B}$, contains exactly one nonzero element, equal to unity, which maps one element in $\mathbf{g}$ or $\mathbf{c}$ to its corresponding grid-cell component. All the grid-cell systems may then be written as one matrix equation, shown in (3.11):

$$\Pi\left(\mathcal{I}\mathbf{g} + \mathcal{B}\mathbf{c}\right) = \mathbf{0} \tag{3.11}$$

We would like to find a solution which satisfies (3.11) as well as possible in the entire computational domain.

$$\nabla_{\mathbf{g}} \int_V \left(\mathcal{I}\mathbf{g} + \mathcal{B}\mathbf{c}\right)^T \Pi^T \Pi \left(\mathcal{I}\mathbf{g} + \mathcal{B}\mathbf{c}\right) \mathrm{d}V = \mathbf{0} \tag{3.12}$$

Recall that $\Pi$ depends on the grid-components, which are unknown. In order to solve (3.12) as a linear system, we will use an iteration where $\Pi$ is taken as constant, constructed using an initial guess and subsequently updated with the latest approximation of the grid components.

$$\underbrace{\int_V \mathcal{I}^T \Pi_{(n)}^T \Pi_{(n)} \mathcal{I} \mathrm{d}V}_{\Phi_{(n)}^T \Phi_{(n)}} \mathbf{g}_{(n+1)} = \underbrace{-\int_V \mathcal{I}^T \Pi_{(n)}^T \Pi_{(n)} \mathcal{B} \mathrm{d}V \mathbf{c}}_{\omega_{(n)}} \tag{3.13}$$

$$\Phi_{(n)}^T \Phi_{(n)} \mathbf{g}_{(n+1)} = \omega_{(n)} \tag{3.14}$$

The iteration[1] defined in (3.14) constitutes a fully coupled scheme since it solves for the velocity and pressure simultaneously and will, under conditions which will be studied further, give solutions for $\mathbf{g}$ which converge toward a constant value until floating point errors become dominant. This approach is successfully used in [9] (with variable relaxation factors).

### 3.3.2 Linearization and Implicit Time Marching

Time marching can be approached in much the same way as in the steady state case. We are using an incompressible fluid in the current example, so explicit time marching is unstable. One way to deal with this is to use artificial compressibility. Another way is to use implicit time marching, which we will go through here using backward

[1] A type of *fixed point iteration.*

Euler integration (used to simulate two-phase incompressible 3D flow in Chapter 5 and Chapter 6.2).

$$\underbrace{\mathbf{f}|_{t=1}}_{\mathbf{f}^{(1)}} = \underbrace{\mathbf{f}|_{t=0}}_{\mathbf{f}^{(0)}} + h \underbrace{\left.\frac{\partial \mathbf{f}}{\partial t}\right|_{t=h}}_{\dot{\mathbf{f}}^{(1)}} \tag{3.15}$$

Thus we substitute $\mathbf{u}^{(i-1)} + h\dot{\mathbf{u}}^{(i)}$ for $\mathbf{u}$, $\mathbf{v}^{(i-1)} + h\dot{\mathbf{v}}^{(i)}$ for $\mathbf{v}$, $\mathbf{w}^{(i-1)} + h\dot{\mathbf{w}}^{(i)}$ for $\mathbf{w}$ and $\mathbf{p}^{(i-1)} + h\dot{\mathbf{p}}^{(i)}$ for $\mathbf{p}$ in (3.4). In this case $\dot{\mathbf{u}}^{(i)}$, $\dot{\mathbf{v}}^{(i)}$, $\dot{\mathbf{w}}^{(i)}$ and $\dot{\mathbf{p}}^{(i)}$ are the unknown quantities. As before we collect nonlinear convection terms and viscous stress terms. However, we must add the time derivative on the left hand side and subtract the previous timestep on the right hand side.

$$\mathbf{m}_n^{(i)T} \stackrel{\text{def}}{=} \mathbf{b}^T \mathbf{u}^{(i)} \mathbf{b}_x^T + \mathbf{b}^T \mathbf{v}^{(i)} \mathbf{b}_y^T + \mathbf{b}^T \mathbf{w}^{(i)} \mathbf{b}_z^T \tag{3.16}$$

In this case we let the pressure term be treated as constant when solving for the velocity and vice versa. This gives us the two systems (3.17) and (3.18), which are smaller than the corresponding system (3.7) which was used in the steady state case and may be solved using a semi-coupled scheme or an uncoupled scheme.

$$\begin{bmatrix} h\mathbf{b}_x^T & h\mathbf{b}_y^T & h\mathbf{b}_z^T \\ \frac{\mathbf{b}^T}{\eta} + h\left(\mathbf{m}^{(i)T} + \mathbf{s}^T\right) & 0 & 0 \\ 0 & \frac{\mathbf{b}^T}{\eta} + h\left(\mathbf{m}^{(i)T} + \mathbf{s}^T\right) & 0 \\ 0 & 0 & \frac{\mathbf{b}^T}{\eta} + h\left(\mathbf{m}^{(i)T} + \mathbf{s}^T\right) \end{bmatrix} \begin{bmatrix} \dot{\mathbf{u}}^{(i)} \\ \dot{\mathbf{v}}^{(i)} \\ \dot{\mathbf{w}}^{(i)} \end{bmatrix} = \begin{bmatrix} -\left(\mathbf{b}_x^T \mathbf{u}^{(i-1)} + \mathbf{b}_y^T \mathbf{v}^{(i-1)} + \mathbf{b}_z^T \mathbf{w}^{(i-1)}\right) \\ -\mathbf{b}_x^T \mathbf{p}^{(i)} - \left(\mathbf{m}^{(i)T} + \mathbf{s}^T\right) \mathbf{u}^{(i-1)} \\ -\mathbf{b}_y^T \mathbf{p}^{(i)} - \left(\mathbf{m}^{(i)T} + \mathbf{s}^T\right) \mathbf{v}^{(i-1)} \\ -\mathbf{b}_z^T \mathbf{p}^{(i)} - \left(\mathbf{m}^{(i)T} + \mathbf{s}^T\right) \mathbf{w}^{(i-1)} \end{bmatrix} \tag{3.17}$$

$$\begin{bmatrix} h\mathbf{b}_x^T \\ h\mathbf{b}_y^T \\ h\mathbf{b}_z^T \end{bmatrix} \dot{\mathbf{p}}^{(i)} = \begin{bmatrix} -\mathbf{b}_x^T \mathbf{p}^{(i-1)} - \frac{\mathbf{b}^T \dot{\mathbf{u}}^{(i)}}{\eta} - \left(\mathbf{m}^{(i)T} + \mathbf{s}^T\right) \mathbf{u}^{(i)} \\ -\mathbf{b}_y^T \mathbf{p}^{(i-1)} - \frac{\mathbf{b}^T \dot{\mathbf{v}}^{(i)}}{\eta} - \left(\mathbf{m}^{(i)T} + \mathbf{s}^T\right) \mathbf{v}^{(i)} \\ -\mathbf{b}_y^T \mathbf{p}^{(i-1)} - \frac{\mathbf{b}^T \dot{\mathbf{w}}^{(i)}}{\eta} - \left(\mathbf{m}^{(i)T} + \mathbf{s}^T\right) \mathbf{w}^{(i)} \end{bmatrix} \tag{3.18}$$

Now, recalling that the system is nonlinear – unknown quantities are present in the cell matrices as well as in the solution vector, we employ the same strategy as with the steady state case. The only differences are that we have two systems, and that the unknowns now are the time derivatives, $\dot{\mathbf{u}}$, $\dot{\mathbf{v}}$, $\dot{\mathbf{w}}$ and $\dot{\mathbf{p}}$. Since we have two systems, two sets of mapping matrices are needed. Let the velocity system be $\mathcal{I}_v \mathbf{g} + \mathcal{B}_v \mathbf{c}$ and let the pressure system be $\mathcal{I}_p \mathbf{h} + \mathcal{B}_p \mathbf{d}$. Further, let $\Pi$ be the matrix of the grid-wide system for the velocity and $\Xi$ be the matrix of the grid wide system for the pressure. We omit the time-step indication, $(i)$, and let it be understood that the equation sets deal with the $i$'th time step where the $i$'th interior components are unknown.

$$\Pi\left(\mathcal{I}_v \mathbf{g}_v + \mathcal{B}_v \mathbf{c}_v\right) = \boldsymbol{\pi} \tag{3.19}$$

$$\Xi\left(\mathcal{I}_p \mathbf{g}_p + \mathcal{B}_p \mathbf{c}_p\right) = \boldsymbol{\xi} \tag{3.20}$$

A least squares solution is then formulated for both systems.

$$\underbrace{\int_V \mathcal{I}_v^T \Pi_{(n)}^T \Pi_{(n)} \mathcal{I}_v \mathrm{d}V}_{\Phi_{(n)}^T \Phi_{(n)}} \mathbf{g}_{(n+1)} = \underbrace{\int_V \mathcal{I}_v^T \Pi_{(n)}^T \left(\boldsymbol{\pi}_{(n)} - \Pi_{(n)} \mathcal{B}_v \mathbf{c}_{(n)}\right) \mathrm{d}V}_{\omega_{(n)}} \tag{3.21}$$

$$\underbrace{\int_V \mathcal{I}_p^T \Xi_{(n)}^T \Xi_{(n)} \mathcal{I}_p \mathrm{d}V}_{\Theta_{(n)}^T \Theta_{(n)}} \mathbf{h}_{(n+1)} = \underbrace{\int_V \mathcal{I}_p^T \Xi_{(n)}^T \left(\boldsymbol{\xi}_{(n)} - \Xi_{(n)} \mathcal{B}_p \mathbf{d}_{(n)}\right) \mathrm{d}V}_{\psi_{(n)}} \tag{3.22}$$

For each time-step, we have the following linearized iterations

$$\Phi_{(n)}^T \Phi_{(n)} \mathbf{g}_{(n+1)} = \omega_{(n)} \tag{3.23}$$

$$\Theta_{(m)}^T \Theta_{(m)} \mathbf{h}_{(m+1)} = \psi_{(m)} \tag{3.24}$$

We may choose to run these iterations interchangeably, always using the latest approximation of one when solving the other (semi-coupled scheme), or we may choose to solve the velocity iteration completely and then solving the pressure system in one step since it is linear once the velocity is taken as constant (uncoupled scheme).

### 3.3.3 Notes on Boundary Conditions

A grid-point on the boundary may contain both quantities which are known (given by the boundary conditions) but may, at the same time, also contain unknown quantities. For example, assume we have no-slip Diriclet boundary conditions for the velocity at a boundary in the *xy*-plane. In this case the velocity components, including derivatives in the $x$ and $y$ directions will be given by the boundary conditions, but any derivatives in the $z$ direction, including mixed derivatives, will be unknown because they do not describe the condition *on* the boundary – rather the condition as we move away from the boundary. Von Neumann boundary conditions may similarly be applied directly by simply defining the derivatives in the grid points at the boundary as known quantities.

In the incompressible case there must also be a constraint for the pressure. This can be implemented by assigning a constant pressure to a fixed point, or by adding an extra conservation-like equation to the pressure.

## 3.4 Mapping Matrices

The mapping matrix $\mathcal{I}$ is a highly sparse matrix with either exactly one or none nonzero entries in each row which is equal to unity. Each row then corresponds to one element in one particular grid cell and each column corresponds to one of the interior-components (hence it may be zero if the element in the grid cell is a boundary-component). The mapping matrix $\mathcal{B}$ is structured in the same way, except that in this case the columns correspond to boundary-components. Figure 3.2 illustrates the logical layout of the mapping matrices.

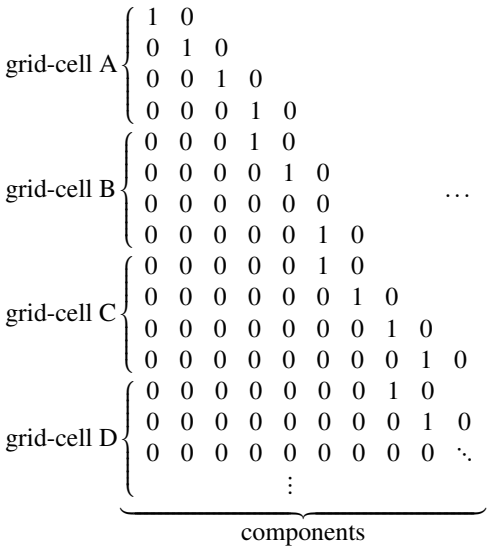



Figure 3.2: The logical structure of the mapping matrices. If an interior (boundary) component maps to a specific grid-cell component, there is a one in the row corresponding to the grid-cell component and the column corresponding to the interior (boundary) component. A row with all zeros (see third row of grid-cell B) means that the grid-cell component has no corresponding interior (boundary) component and is thus a boundary (interior) component. Not that all columns must have at least one nonzero entry, since all interior (boundary) components maps to at least one grid-cell component.

In practice, the mapping matrices are not formed explicitly. Instead they are implemented as algorithms or as one-dimensional index arrays (see Figure 3.3). The matrix form is useful when formulating the equation system and the index/algorithmic approach is useful for the actual computation as it is efficient in terms of storage and operation count.

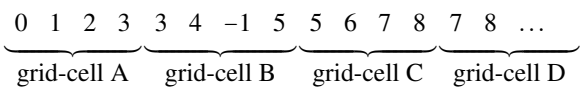



Figure 3.3: Mapping matrix represented by an index array. The number in the array is column index of the mapping matrix.

## 3.5 Sampling and Integration

### 3.5.1 Numeric Versus Analytic

The integration over the computational domain, (3.13), may in some cases be computed analytically (see for example the test case in Chapter 4.1). For general purposes,

however, this integration must be performed numerically.

### 3.5.2 Sampling

The numeric integration is done for each grid-cell separately and then added to the product ((3.14), (3.23) or (3.24)) using the mapping matrices. A set of sample positions for each grid-cell are defined and the value of the integral is taken to be approximately the sum of the value of the integrand at each point, divided by the number of points. Both uniform and randomly distributed sample points have been tested.[2] This summation is suitable for parallel computation and we will go through the implementation on GPU in Chapter 4.3.

### 3.5.3 Sub-grid Features

The numeric integration allows us to apply different governing equations independent of the grid resolution and orientation. Different fluid properties (i.e. multiphase flow) and solid objects may thus be included without adapting the grid to fit a variable geometry. Figure 3.4 shows how different regions may intersect with a grid-cell. At each sample position one select the appropriate governing equation and apply its contribution to the integral.

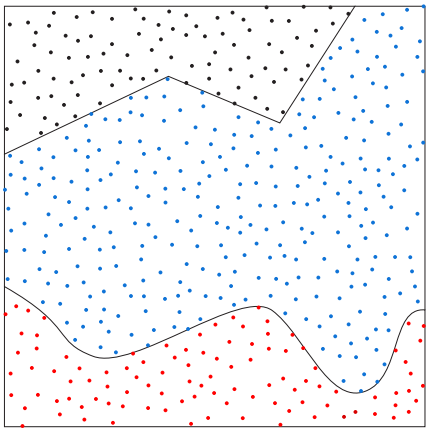


Figure 3.4: Illustration of different regions within a grid-cell where different governing equations are applied. In this example the sample points in black falls within a solid region, the blue and red sample points represent fluids with different properties. Since the integration is performed numerically, the different regions may have any shape. However, features very small compared to the grid-cell will be approximated poorly unless the order of continuity is increased to compensate.

[2] Uniform sample positions are used in Chapter 6.1, while randomly distributed sample positions are used in Chapter 5 and Chapter 6.2.

## 3.6 Further Development on The Solution Algorithm

The use of iterative linearization, together with sparse matrix schemes, in order to solve the algebraic equation system was originally implemented for the two dimensional steady state case. As the dimension increases to 3 and 4, the memory requirement becomes an obstacle, even with the benefit of sparse matrix storage schemes. In particular GPU memory is a limiting factor. For this purpose, the next step in developing this method should be the implementation of a pure nonlinear algorithm, e.g. nonlinear conjugate gradients, which does not require a linearized equation system to be created. This would enable higher resolution in the three-dimensional simulations and would also enable us to formulate a time dependent problem on a four-dimensional grid without using a marching scheme.

# Chapter 4

# Implementation

## 4.1 Spherical Laplace Equation

Before diving into the Navier–Stokes Equations in higher dimensions we will go through the implementation of a simple example; the Laplace equation in spherical coordinates with no angular dependence. The problem is discretized for a general resolution ($K$ grid-points) and a fixed $\mathcal{C}^2$ continuity, i.e. $\Omega = 3$.

### 4.1.1 Governing Equations

The problem is defined by the differential equation (4.1) and the boundary conditions (4.2). Together these yield the analytical solution (4.3).

$$\frac{2}{r}\frac{\partial f}{\partial r} + \frac{\partial^2 f}{\partial r^2} = 0 \tag{4.1}$$

$$f(r_0) = f_0\,, \quad f(r_1) = f_1 \tag{4.2}$$

$$f(r) = \frac{r_1 f_1 - r_0 f_0}{r_1 - r_0} - \frac{r_0 r_1 (f_1 - f_0)}{r_1 - r_0} r^{-1} \tag{4.3}$$

We now turn to the numerical solution. In accordance with the mathematical framework, we represent $f$ on a uniform grid with a spacing between the grid-points equal to unity. Let there be $K$ grid-points, with coordinates $\mathbf{x}[k] = k$ for $k \in \{0, \dots, K-1\}$. Thus we have

$$r = \frac{r_1 - r_0}{K-1} x + r_0 \tag{4.4a}$$

$$\frac{\partial f}{\partial r} = \frac{K-1}{r_1 - r_0}\frac{\partial f}{\partial x} \tag{4.4b}$$

$$\frac{\partial^2 f}{\partial r^2} = \left(\frac{K-1}{r_1 - r_0}\right)^2 \frac{\partial^2 f}{\partial x^2} \tag{4.4c}$$

We write (4.1) in grid coordinates as

$$\frac{\partial f}{\partial x} + \frac{x + A r_0}{2}\frac{\partial f^2}{\partial x^2} = 0 \quad \text{where } A \stackrel{\text{def}}{=} \frac{K-1}{r_1 - r_0} \tag{4.5}$$

| | | | | | |
|---|---|---|---|---|---|
| $\frac{8c^2+4c+1}{7/15}$ | $\frac{240c^2+96c+17}{28/5}$ | $\frac{24c^2+16c+3}{14/15}$ | $-\frac{8c^2+4c+1}{7/15}$ | $\frac{240c^2+144c+29}{7/15}$ | $-\frac{24c^2+8c+1}{14/15}$ |
| $\iddots$ | $\frac{192c^2+31c+12}{7/5}$ | $\frac{88c^2+48c+9}{14/15}$ | $-\frac{240c^2+96c+17}{28/5}$ | $\frac{432c^2+216c+37}{28/5}$ | $-\frac{(4c-1)(4c+1)}{7/15}$ |
| $\iddots$ | $\iddots$ | $\frac{24c^2+6c+1}{7/90}$ | $-\frac{24c^2+16c+3}{14/15}$ | $\frac{(4c+1)(4c+3)}{7/15}$ | $\frac{8c^2+4c+1}{7/45}$ |
| $\iddots$ | $\iddots$ | $\iddots$ | $\frac{8c^2+4c+1}{7/15}$ | $-\frac{240c^2+144c+29}{28/5}$ | $\frac{24c^2+8c+1}{14/15}$ |
| $\iddots$ | $\iddots$ | $\iddots$ | $\iddots$ | $\frac{384c^2+322c+89}{14/5}$ | $-\frac{88c^2+40c+7}{14/15}$ |
| $\iddots$ | $\iddots$ | $\iddots$ | $\iddots$ | $\iddots$ | $\frac{12c^2+9c+2}{7/180}$ |

Table 4.1: Analytically computed elements of the symmetric matrix, $\mathbf{L}_k$, where $c \stackrel{\text{def}}{=} (\mathbf{x}[k]+Ar_0)/2$

Using the approximation given in (2.14), we arrive at the following algebraic equation for the $k$'th grid-cell (4.6):

$$\left(\frac{\partial \mathbf{b}_{2\Omega}^T(x_k')}{\partial x}+\frac{x+Ar_0}{2}\frac{\partial^2 \mathbf{b}_{2\Omega}^T(x_k')}{\partial x^2}\right)\mathbf{f}_k=0 \quad \text{where} \quad x_k'=x-\mathbf{x}[k] \tag{4.6}$$

We can write (4.6) as

$$\mathbf{l}_k^T(x_k')\mathbf{f}_k=0 \tag{4.7}$$

The grid-cell system then is

$$\underbrace{\sum_{k=0}^{K-2}\int_0^1 \mathbf{l}_k(x_k')\mathbf{l}_k^T(x_k')\mathrm{d}x_k'}_{\mathbf{L}_k}\mathbf{f}_k=\mathbf{0} \tag{4.8}$$

$$\sum_{k=0}^{K-2}\mathbf{L}_k\mathbf{f}_k=\mathbf{0} \tag{4.9}$$

The integrand in (4.8) is a symmetric $2\Omega\times 2\Omega$ matrix with polynomial entries. The integral is computed analytically. The entries of $\mathbf{L}_k$ is shown in Table 4.1. The unknown interior-components are ordered into a column-vector as shown in (4.10):

$$\begin{bmatrix}\mathbf{F}[1,0]\\ \mathbf{F}[1,1]\\ \vdots\\ \mathbf{F}[1,\Omega-1]\\ \mathbf{F}[2,0]\\ \mathbf{F}[2,1]\\ \vdots\\ \mathbf{F}[K-2,\Omega-2]\\ \mathbf{F}[K-2,\Omega-1]\\ \mathbf{F}[K-1,1]\\ \mathbf{F}[K-1,2]\\ \vdots\\ \mathbf{F}[K-1,\Omega-1]\end{bmatrix} \tag{4.10}$$

Note that the the boundary-components, $\mathbf{F}[0,0]$ and $\mathbf{F}[K-1,0]$, are not present in (4.10). The ordering of the grid-components is in principle arbitrary. However, for larger systems it is convenient to arrange them so that the resulting system becomes narrow banded. The mapping matrices are not formed explicitly but implemented as algorithms. The equation system is linear and is solved directly using Gaussian elimination.

### 4.1.2 Implementation Details

The implementation is kept simple by using a fixed order of continuity of 2 (thus $\Omega = 3$). The grid resolution is variable. The code is written in C++ using standard libraries only. The complete source code is given in Listing B.1 and is tested with the MinGW 64 bit compiler. We will go through the important details and explain how they relate to the mathematical framework. Grid data, boundary values and other grid properties are stored as global data shown in Listing 4.1. We use 64 bit double precision for all real numbers and 32 bit signed integers for indices and counters.

Listing 4.1: Global Data

```
// grid properties
const int Omega = 3;                  // number of derivatives
const double f0 = 1.0;                // lower boundary value
const double f1 = 10.0;               // upper boundary value
const double r0 = 1.0;                // lower boundary radius
const double r1 = 10.0;               // upper boundary radius
int K = 5;                            // number of grid-points
double A = (K-1)/(r1-r0);             // reference-frame change
vector<double> F(K*Omega,0.0);        // grid values
```

The algoritm is implemented using the set of functions shown in Listing 4.2.

Listing 4.2: Functions

```
// Computes normalized basis functions for Omega = 3.
// The first argument is the grid cell coordinate.
// The basis functions are evaluated and stored in the
// array given in the second argument
void basis6(const double, double [6]);

// Evaluates a function based on grid-cell components
// given in an array (second argument), at coordinate
// given in the first argument
double evaluate6(const double, const double [6]);

// Returns the analytic reference solution:
// f(r) = a + b/r
double analytic(const double);

// Returns the computed solution
double computed(const double);
```

```
// Computes the laplacian vector product integrated over
// the k'th grid-cell. Result is stored in the array,
// which is treated as a 6 by 6 matrix (stored row-wise)
void sphLaplacianIntegral(const int, double [36]);

// Maps grid-components to non-boundary components
// returns -1 if given a boundary component
int mapComp(const int);

// Maps the non-boundary components to the grid-components
int unmapComp(const int);

// Computes the solution to our test problem for a given
// resolution
void solveSystem(const int);

// Solves a symmetric, linear equation system
void solveSym(const int,          // size
              vector<double> &,   // matrix
              vector<double> &);  // result

// Compares the computed solution to the analytic solution
double getRMSerror(void);
```

The main function, Listing 4.3, computes solutions for a range of resolutions and prints the RMS error to the console.

Listing 4.3: Main Function

```
// Compute solutions for different grid resolutions
// print the RMS error for each resolution
int main()
{
    for(int i = 5; i <= 20; i++)
    {
        cout << "K = " << setw(3) << i;
        solveSystem(i);
        cout << ", RMS error: "
             << setw(12) << getRMSerror() << std::endl;
    }

    return 0;
}
```

Listing 4.4 shows the implementation of the basis functions, $\hat{\mathbf{b}}_6(x)$ in Table 2.3. The polynomials are factorized in order to reduce round-off errors. For $\Omega = 3$, there are six basis functions. These are needed together, thus the function evaluates them all and stores the result in an array.

Listing 4.4: Basis Function Set

```
// x is the coordinate, result is stored in B[]
void basis6(const double x, double B[6])
{
    const double x2 = x*x;
    const double x3 = x2*x;
    const double y = 1 - x;
    const double y2 = y*y;
    const double y3 = y2*y;

    B[0] = y3*(3*(2*x + 1)*x + 1);
    B[1] = 5*y3*x*(3*x + 1);
    B[2] = 30*y3*x2;
    B[3] = x3*(3*(2*y + 1)*y + 1);
    B[4] = -5*x3*y*(3*y + 1);
    B[5] = 30*x3*y2;
}
```

Listing 4.5 shows the implementation the assembled algorithm. A grid wide system is instantiated, and a loop over all the grid cells adds the grid-cell systems, (4.9), to the grid-wide system. The system is solved (Gaussian elimination) and copied back to the grid.

Listing 4.5: System Assembly and Solution

```
// Computes the solution to our test problem
void solveSystem(const int num_gridpoints)
{
    K = num_gridpoints;
    F.resize(K*Omega,0.0);
    A = (K-1)/(r1-r0);

    // Set boundary components (a0 = 1)
    F[0] = f0;
    F[Omega*(K-1)] = f1;

    // Instantiate and initialize matrices
    const int N = Omega*K - 2; // system size
    vector<double> M(N*N,0.0); // system matrix
    vector<double> B(N,0.0);   // system result
    double LLT[4*Omega*Omega]; // cell system

    // Loop over all the grid-cells
    for(int k = 0; k < K-1; k++)
    {
        // Get the laplacian integral for current cell
        sphLaplacianIntegral(k,LLT);
```

```
        // Add the cell-system to the grid-wide system
        const int i0 = Omega*k;
        for(int i = 0; i < 2*Omega; i++)
        {
            const int row = mapComp(i0 + i);

            // ignore rows corresponding to the boundary
            if(row >= 0)
                for(int j = 0; j < 2*Omega; j++)
                {
                    const int col = mapComp(i0 + j);
                    const double Lij = LLT[2*Omega*i + j];

                    // col >= 0:
                    // non-boundary-> add to matrix

                    // col < 0:
                    // boundary-> add to solution

                    if(col >= 0)
                        M[N*row + col] += Lij;
                    else
                        B[row] -= Lij*F[i0 + j];
                }
        }
    }

    // solve the linear system
    solveSym(N,M,B);

    // copy solution back to grid
    for(int i = 0; i < N; i++)
        F[unmapComp(i)] = B[i];
}
```

### 4.1.3 Output and Plots

Listing 4.6 shows the console output of the program and Figure 4.1 shows a plot of the RMS error on a logarithmic scale. As expected, the RMS error decreases exponentially with an increasing grid resolution. Figure 4.2 shows plots of different computed solutions together with the reference.

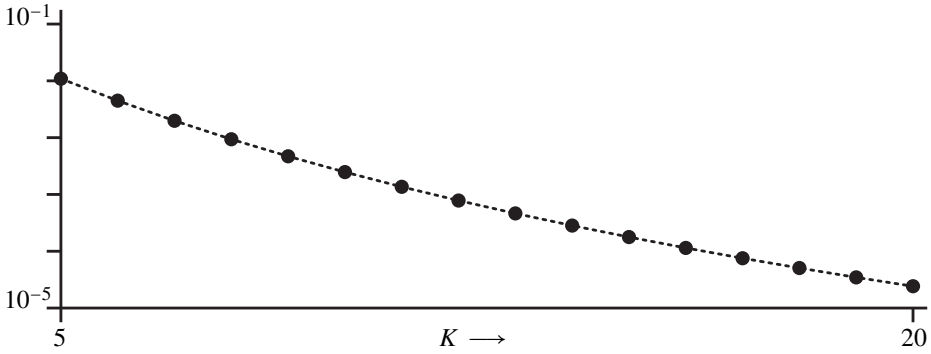



Figure 4.1: This figure shows a plot of the RMS error from Table 4.6 on a logarithmic scale.

Listing 4.6: Console Output

```
K =   5, RMS error:     0.109125
K =   6, RMS error:    0.0447779
K =   7, RMS error:    0.0198103
K =   8, RMS error:   0.00936651
K =   9, RMS error:   0.00469804
K =  10, RMS error:   0.00247644
K =  11, RMS error:   0.00136312
K =  12, RMS error:  0.000781452
K =  13, RMS error:  0.000463423
K =  14, RMS error:  0.000283197
K =  15, RMS error:  0.000177757
K =  16, RMS error:  0.000114372
K =  17, RMS error: 7.53395e-005
K =  18, RMS error: 5.06704e-005
K =  19, RMS error: 3.46625e-005
K =  20, RMS error: 2.42191e-005
```

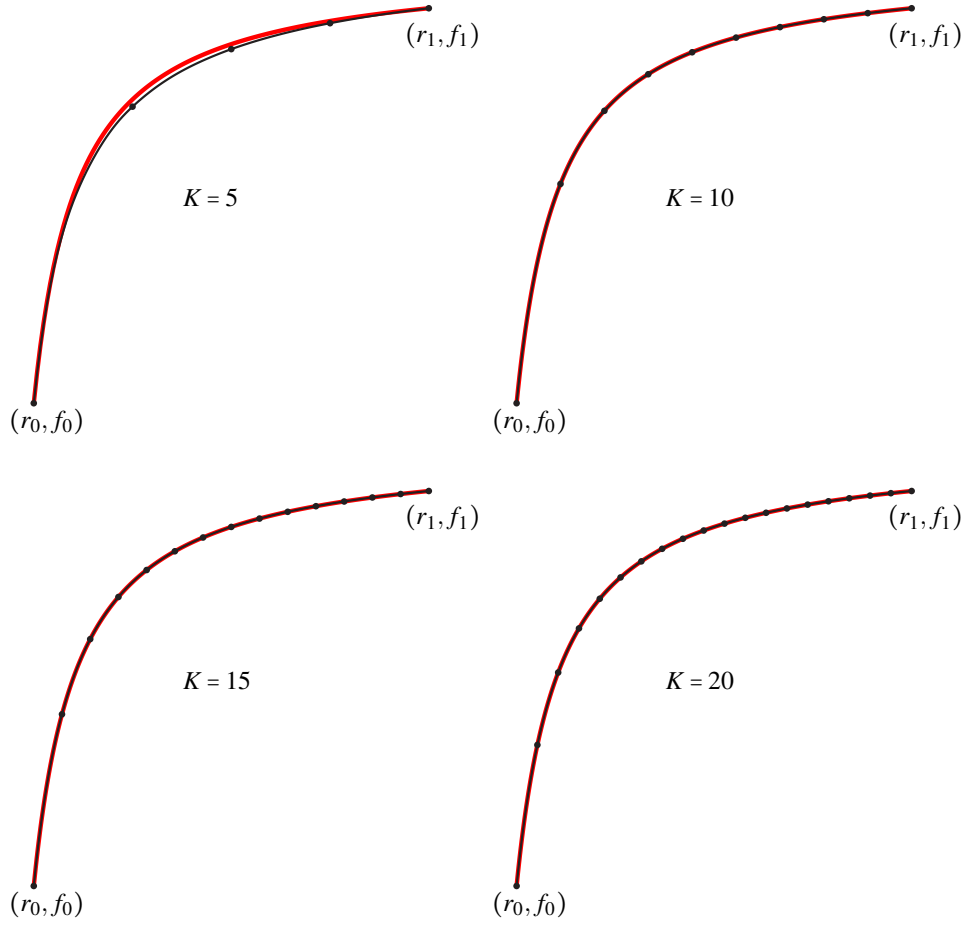



Figure 4.2: Plots of the computed solution, $\hat{\mathbf{b}}^T(x'_k)\mathbf{f}_k$, for four different grid resolutions (black lines with dots at the grid-points) compared with the analytic reference solution (red line).

## 4.2 Code Structure and Data Layout

### 4.2.1 Solver Application Structure

The code environment used to implement and test the current method was chosen based on requirements on versatility, memory management control, low level control and available application programming interfaces (APIs). In order to achieve acceptable computation speed with low-cost hardware it was necessary to implement algorithms for use in parallel on GPU devices. The Open Compute Library (OpenCL) was used for this purpose. Visualization, which is essential for debugging and verification, was implemented using the Open Graphics Library (OpenGL) for real time visualization and ray-tracing for higher quality rendered graphics.

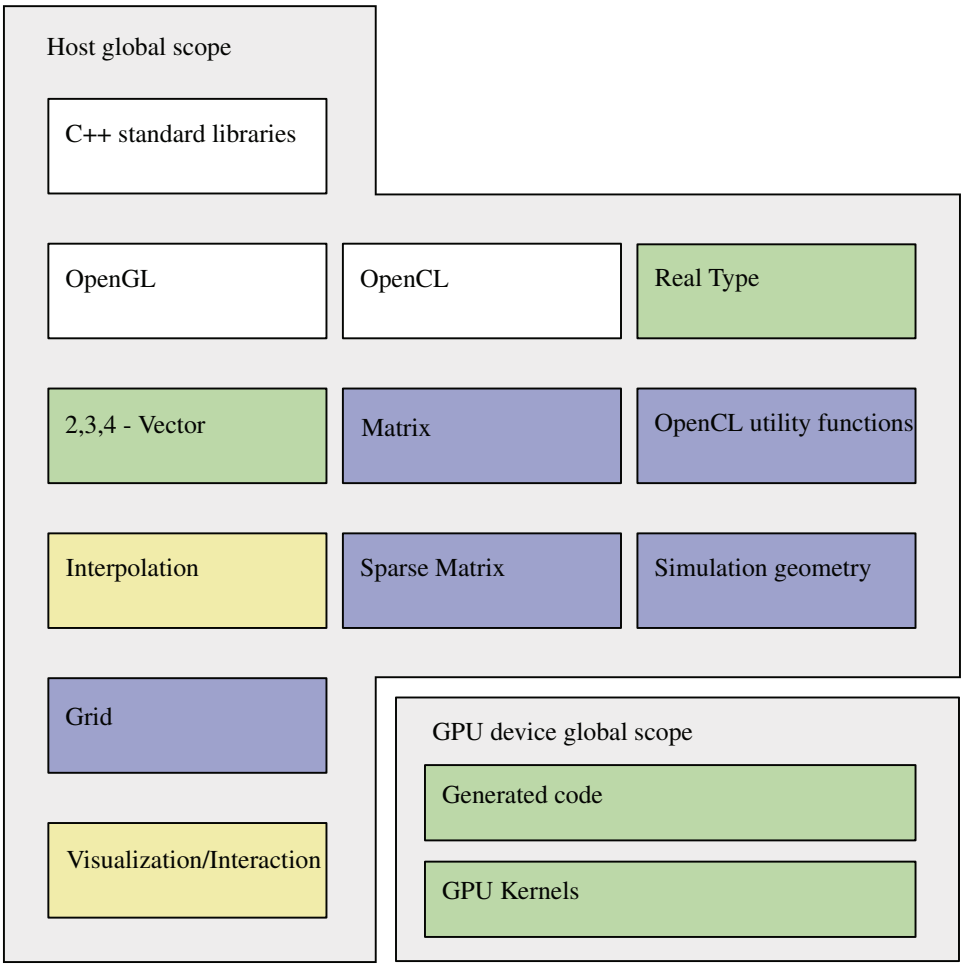



Figure 4.3: A diagram showing the basic layout of the application code which was developed to test the current method. Dependencies points upward within a given scope. Public open-source code is indicated with white background. Code developed especially for this thesis is placed on colored background. Green indicates plain old data (POD) or C99 standard code (compliant with OpenCL GPU source code), blue is used for C++ classes and yellow for C++ namespaces.

Figure 4.3 shows the basic layout of the application code used for in the simulation of unsteady three dimensional flow. Some alterations were made for the different computed results presented in Chapters 5 - 6. Partly improvements and partly modifications to suit each test case. A complete functioning implementation consists of between 20000 and 50000 lines of code, which is too extensive to be listed in this thesis. We will go through parts of the source-code with emphasis on the program structure and key sections necessary to create an efficient implementation.

### 4.2.2 Scalar and Vector Types

All computed results presented in this work have been obtained using 64-bit floating point numbers. However, tests with 32-bit and 128-bit[1] floating point numbers has been made during the development process. Preprocessor directives define a general floating point type used for real numbers[2]. Listing 4.7 shows the preprocessor definitions used when compiling for 64-bit precision. The floating point precision can thus be altered without rewriting any major part of the code.

Listing 4.7: Real Type Definitions

```
#ifdef FLOAT64
#define REAL_TYPE      double
#define REAL_ONE       1.0
#define REAL_TWO       2.0
#define REAL_ZERO      0.0
#define REAL_HALF      0.5
#define REAL_PI        3.14159265358979323846264338327950
#define REAL_TWOPI     6.28318530717958647692528676655 9
#define REAL_EPSILON   2.2204460492503131e-16
#endif // FLOAT64
```

Vector types of constant size for spatial and temporal coordinates and indices are defined (Listing 4.8) in accordance with the Plain Old Data (POD) specification. This ensures that the data is contiguous in memory and is compatible with common implementations of the OpenCL built in data types.

Listing 4.8: Vector Structures

```
struct vec2 { REAL_TYPE x, y; };
struct vec3 { REAL_TYPE x, y, z; };
struct vec4 { REAL_TYPE x, y, z, t; };
struct int2 { int k, l; };
struct int3 { int k, l, m; };
struct int4 { int k, l, m, n; };
```

### 4.2.3 Classes and Namespaces

The matrix class implements a variable sized dense matrix (dynamic memory allocation) based on the real type (Listing 4.7) with arithmetic operations, some common factorizations and partially pivoted Gaussian elimination (see Listing B.2).

The sparse matrix class implements different sparse matrix storage schemes for symmetric matrices and common matrix operations optimized for sparse matrices (see Listing B.3 in the appendix).

The OpenCL utility functions is a class containing a collection of functions which facilitates initialization of OpenCL and encapsulates the handles to the GPU device,

[1] Current GPU's are limited to 32-bit or 64-bit floating point numbers. As a consequence, the tests with 128-bit floating point precision were performed on CPU only.

[2] C++ templates could have been used instead of preprocessing. But this would have reduced compatibility with OpenCL source code.

context and the GPU programs on the test system. Additionally it simplifies parsing of the source code and debugging information between the GPU compiler and the host system.

The simulation geometry class contains information about the computational domain, e.g. which equations to use in different regions and the boundary conditions.

The grid class contains the grid data and implements the algorithm used for the iterative solution approach. It also contains the host code necessary to instruct the GPU device in doing the numeric integration.

The interpolation name-space contains functions which generates OpenCL source-code associated with the basis functions and grid-cell function approximation.

The visualization name-space implements real-time visualization using OpenGL (glfw 3) and user interaction.

### 4.2.4 OpenCL sourcecode

The OpenCL source-code consists of a set of kernel functions which are compiled and executed on a OpenCL compatible device via instructions from the main program (the host). Multiple instances of each kernel may be executed in parallel and (preferably) performs computations on its own limited chunk of data. Each kernel also has access to globally defined functions and data. The global data is accessible to all kernels but, when reading from and writing to global memory, potential access conflicts arise since several kernels are executed simultaneously. A large part of the GPU source code is generated by the host program upon its execution in order to ensure consistency between the host code and the OpenCL code. We will take a closer look at the GPU computations in Chapter 4.3.

### 4.2.5 Grid Data Memory Management and Indexing

The way the grid data is aligned in memory affects how efficiently it can be accessed and may thus affect computational speed in principle. In practice, the use of precomputed indexes (the mapping matrices) ensures that reading or writing grid data does not greatly affect the computational speed in current implementations. All grid data is dynamically allocated as needed and accessed through index lists and pointers. In Listing 4.9 we see the parts of the grid-class which deals with memory allocation and indexing. For each time-step there is a pointer to an array of grid-point structs. Each grid-point struct also contains pointers to grid-components – one for each function which is used in the Navier–Stokes equations.

Listing 4.9: Grid Class Memory Managment and Indexing

```
class Grid {
public:
    enum Vartype { X_VEL=0, Y_VEL=1, Z_VEL=2, PRESSURE=3, PHASE=4 };

    // each grid point contains pointers to the grid data of this point
    struct Gridpoint
    {
        REAL_TYPE *U; // x-velocity
        REAL_TYPE *V; // y-velocity
        REAL_TYPE *W; // z-velocity
```

```
    REAL_TYPE *P; // pressure
    REAL_TYPE *F; // phase
};

    // indices giving a specific grid component
struct ComponentID
{
    int3 i;       // grid-point
    int3 s;       // derivative order
    Vartype ty;   // type (e.g. x-vel, y-vel, pressure etc)
};

// indexes of components in a grid-cell
struct CellIndex
{
    int3 i; // cell index
    std::vector<int> vInd; // position in interior index
    std::vector<int> pInd;
    std::vector<int> fInd;
    std::vector<ComponentID> vCmp; // position in grid
    std::vector<ComponentID> pCmp;
    std::vector<ComponentID> fCmp;
};

// returns the number of grid-points
int numGridpoints(void) const { return int3_prod(num_gpts); }

// returns the number of components (of a type) in each grid-point
int numDerivatives(void) const { return int3_prod(num_sdrv); }

// returns the number components (of a type) in each grid-cell
int numCellVar(void) const { return int3_prod(2*num_sdrv); }

// allocates memory for a set of grid points and returns a
// pointer to the data in memory
Gridpoint * allocateGridpoints(void) const
{
    const int N_G = numGridpoints();
    const int N_S = numDerivatives();

    Gridpoint *gp = new Gridpoint[N_G];

    for(int i = 0; i < N_G; i++)
    {
        gp[i].U = new REAL_TYPE[N_S];
        gp[i].V = new REAL_TYPE[N_S];
        gp[i].W = new REAL_TYPE[N_S];
        gp[i].P = new REAL_TYPE[N_S];
        gp[i].F = new REAL_TYPE[N_S];
    }

    return gp;
}

/*
member functions snipped for brevity
*/
```

```
    // size of the grid in spatial directions
    int3 num_gpts;

    // number of derivatives in each spatial direction
    int3 num_sdrv;

    // as grid-data for each time-step is allocated, a pointer is appended
    // at the end to the gridData array
    std::vector<Gridpoint *> gridData;

    // the following indexes are used to efficiently access the grid data
    std::vector<ComponentID> velocityIndex;
    std::vector<ComponentID> pressureIndex;
    std::vector<ComponentID> phaseIndex;
    std::vector<CellIndex> cellIndex;
};
```

## 4.3 Numeric Integration on GPU

In Chapter 4.1 the integration over the grid-cells was performed analytically. This can not be done in the general case. In the two dimensional case, numerical integration using single threaded CPU computation is feasible, however for three or four dimensional grids, this is would require too much computation time to be practical.

**Algorithm 1** This pseudo-code illustrates how the GPU device is instructed to perform the numeric integration and how the result is mapped back to the linearized system, (3.14), here represented by $\mathbf{Lx} = \mathbf{z}$. The GPU computation is performed in two steps – expansion and summation, which is explained further in Chapter 4.3.1.

**for** $0 \leq s <$ numGridCells() **do**
  $\mathbf{G}_{host} \leftarrow$ gridCellData($s$)
  $\mathbf{G}_{dev} \leftarrow$ copyHostToDevice($\mathbf{G}_{host}$)
  $\mathbf{A}_{dev} \leftarrow$ gpuExpansionStep($\mathbf{G}_{dev}$)
  $\mathbf{A}^T_{dev}\mathbf{A}_{dev} \leftarrow$ gpuSummationStep($\mathbf{A}_{dev}$)
  $\mathbf{S}_{host} \leftarrow$ copyDeviceToHost($\mathbf{A}^T_{dev}\mathbf{A}_{dev}$)
  $N \leftarrow$ numGridCellComponents($s$)
  **for** $0 \leq k < N$ **do**
    $r \leftarrow$ mapGridCellComponent($k$)
    **if** $0 \leq r$ **then**
      **for** $0 \leq l < N$ **do**
        $r \leftarrow$ mapGridCellComponent($l$)
        **if** $0 \leq c$ **then**
          $\mathbf{L}[r,c] \leftarrow \mathbf{L}[r,c] + \mathbf{S}[k,l]$
        **else**
          $\mathbf{z}[r] \leftarrow \mathbf{z}[r] - \mathbf{S}[k,l]\mathbf{G}_{host}[k]$
        **end if**
      **end for**
    **end if**
  **end for**
**end for**

### 4.3.1 Numeric Integration Scheme

An *expansion-summation* scheme was used for the numeric integration on the GPU device. If we study (3.9) and (3.13) we see that the products $\mathbf{A}^T_{k,l,m}\mathbf{A}_{k,l,m}$ may be integrated independently for each grid-cell. The expansion-summation scheme entails that we first compute the value of $\mathbf{A}_{k,l,m}$ for a range of sample positions in parallel. This is the expansion step (see Figure 4.4). Then the summation for each component is computed in parallel. This is the summation step (Figure 4.5). The numeric integral for each grid cell is then added to the system of interior components or to the boundary components using the mapping matrices (see Algorithm 1). The advantage of doing the numeric integral in two steps has to do with GPU-memory accessing – the expansion part is done most efficiently in parallel for the sample points, where the result for each sample adds to the matrix entries of the grid-cell components. Thus there would be a conflict in memory access if one attempted to do the summation in the expansion step, while doing the expansion in the summation step would require evaluation of the basis functions at the given sample position several times.

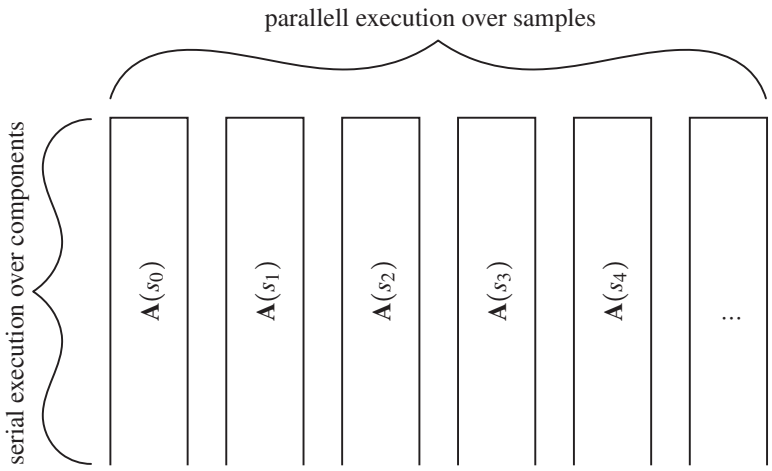



Figure 4.4: Execution order of the expansion step. Each OpenCL kernel is given a unique sample position, $s_i$, of which the contribution to the integral for all grid-cell components is computed.

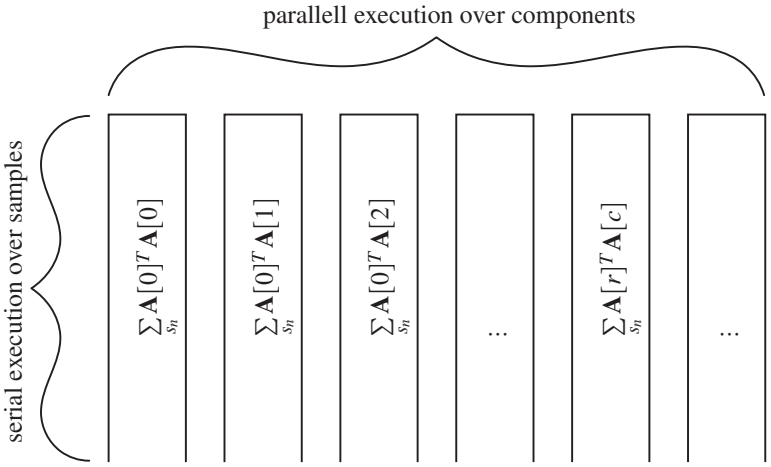



Figure 4.5: Execution order of the summation step. Each OpenCL kernel is given a unique row, $r$, and column, $c$, and computes the sum over all sample positions for this matrix entry.

## 4.4 Solving The Linearized System

### 4.4.1 Sparse Matrix Storage Scheme

Once the integration over all grid-cells is performed and mapped to the linearized systems for the interior components (see (3.14),(3.23) or (3.24)), the remaining task is to solve these systems. These systems tend to become quite large as the number of matrix entries grows proportionally to the resolution and the spatial order to the power of $2 \times d$, where $d$ is the number of dimensions. For example, with $\Omega = 3$ on a $10 \times 10 \times 10$ grid, using a dense storage scheme with 64 bit floating point numbers would require approximately 89 GB of system memory (or 49 GB for the velocity system if solved independently). However, since the linear systems are symmetric and sparse (typical sparsity around 2 % - 3 % depending on the order of continuity, boundary components and grid resolution) when using the indexed sparse matrix storage scheme (see Listing B.3) the required memory is reduced by nearly two orders of magnitude.

### 4.4.2 Direct Solution Versus Iterative

In the presented 3D simulations the semi-coupled scheme was used. For the velocity system, the conjugate gradient iteration was found to be very efficient if preconditioned properly (see Chapter 6.2). The smaller pressure system was most efficiently solved directly (Gaussian elimination without pivoting).

# Chapter 5

# Pelton Bucket Simulation

## 5.1 Outline

The material in the current chapter was presented at the Coupled Problems 2015 [1] conference and aims to demonstrate a feasible potential for use of the current method for simulating two phase flow, describing the basic mechanical property of flow in a Pelton-turbine bucket. As previously discussed, we can not hope to achieve an accurate realistic simulation without resorting to some form of modeling to compensate for the lacking resolution of the grid. Instead we will construct idealized situation with one stationary bucket and two incompressible fluids which differ only in density. The bucket is symmetric, but we are not exploiting this. The bucket has a discontinuous curvature in the symmetry plane, so using the symmetry plane to simplify computations could obscure potential issues with the interaction of the fluid and the bucket in this region. We employ the methodology as described in Chapter 3.3.2 (and also used in [10]).

## 5.2 Geometry and Computational Domain

For the purpose of this idealized simulation it is convenient to use an idealized bucket design. We will define an analytic model of the turbine bucket.[1] The analytic bucket is designed to be mathematically simple, yet capable of performing the mechanical task of a real turbine bucket.

$$r(y') = S_x\sqrt{1+\left(\frac{y'}{S_y}\right)^2} \tag{5.1a}$$

$$z'(x',r) = -x'\sqrt{\frac{r}{x'}-1} \tag{5.1b}$$

$$1 = \left(\frac{x'}{S_x+\delta}\right)^2 + \left(\frac{y'}{S_y+\delta}\right)^2 + \left(\frac{z'}{S_x+2\delta}\right)^2 \tag{5.1c}$$

The inner surface is a half-circle in the $xz$-plane extruded along an ellipse in the $xy$-plane. The ellipse is defined in (5.1a), and (5.1b) gives the half-circle. The primed

[1] A parametric or polygon based model of an actual modern bucket design may be hard to obtain and may also require further processing in order to be implemented in a numeric solver.

coordinates, $x', y'$ and $z'$, are here relative to the center of the bucket rather than relative to the origin in the computational domain (not to be confused with sub-grid coordinates). A cylindric section is subtracted from the tip of the bucket, allowing a rotation into the beam without obstructing the previous bucket. Figure 5.2 shows cross sections of the inner surface and Figure 5.1 shows perspective renderings from different angles. The outer surface of the bucket is the $xy$-plane at $z' = 0$ and a spheroid defined by (5.1c). The parameter $\delta$ determines the thickness. The different parameters in (5.1) and Figure 5.2 are given in Table 5.1.

| $S_x$ | $S_y$ | $\delta$ | $r_b$ |
|---|---|---|---|
| 0.35 | 0.35 | 0.05 | 0.1 |

Table 5.1: Analytic bucket parameters

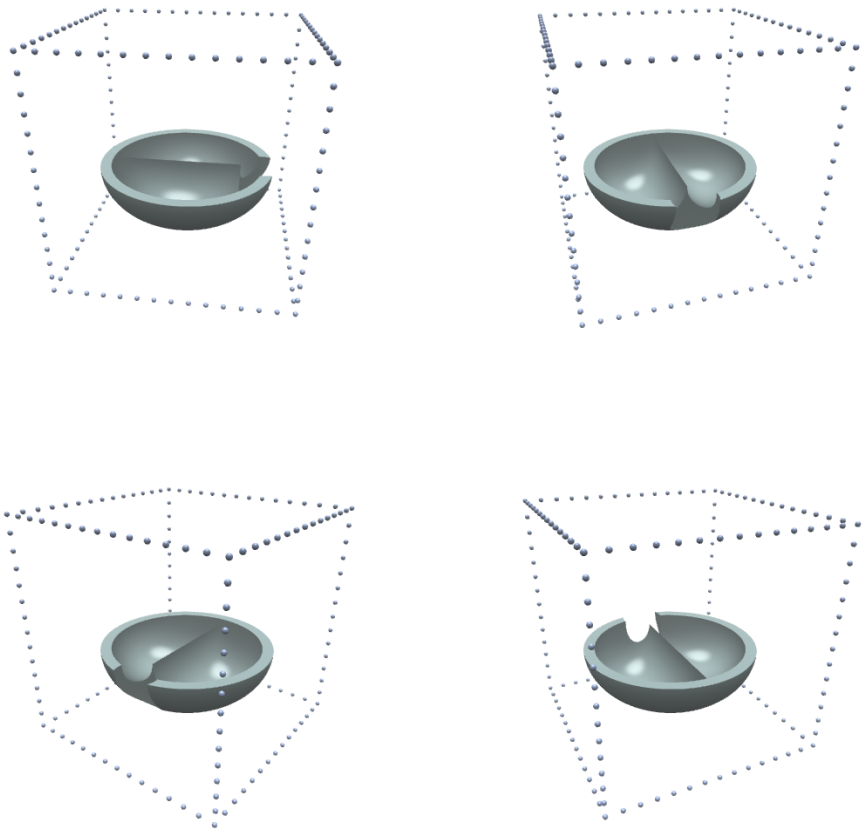


Figure 5.1: Perspective renderings of the bucket (generated with ray-tracing). The edges of the computational domain and grid resolution are given by the small spheres.

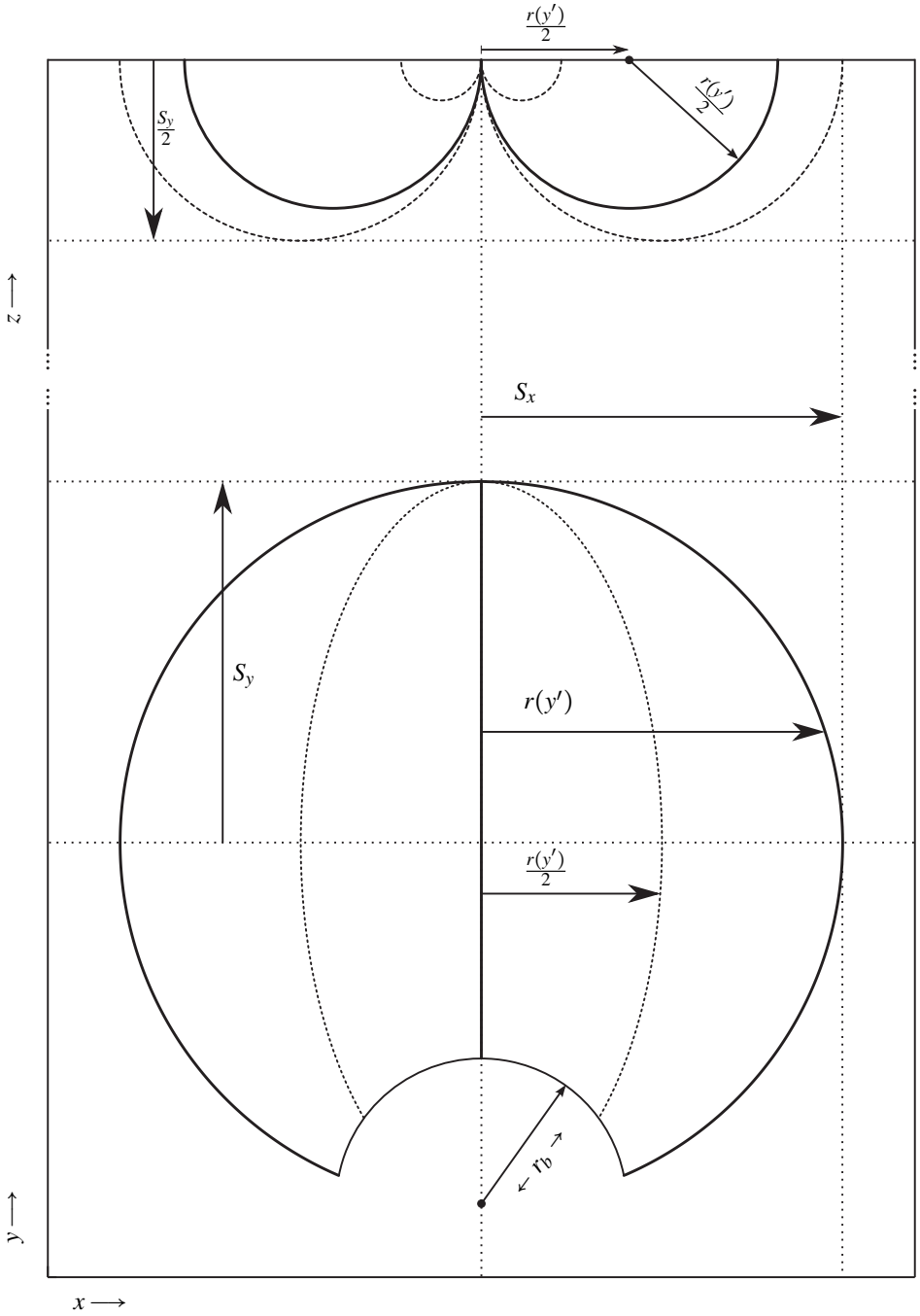



Figure 5.2: The upper graph shows a cross section in the *xz*-plane. The lower graph shows the outline and center of the ellipse the inner surface is extruded along. The subtracted cylindrical section is also indicated.

### 5.2.1 Boundary Conditions and Simulation Parameters

The simulation parameters are given in Table 5.2. Figure 5.3 shows a cross-section of the initial phase distribution.

| | |
|---|---|
| $L$ | 15 |
| $\eta$ | 14 |
| $\tau$ | 42/14 |
| $\Omega$ | 2 |
| Re | 1000 |
| $\beta_{heavy}$ | $10^{-2}$ |
| $\beta_{light}$ | 10 |

Table 5.2: Simulation parameters with notation in accordance with the formulation of the governing equations in Appendix A.

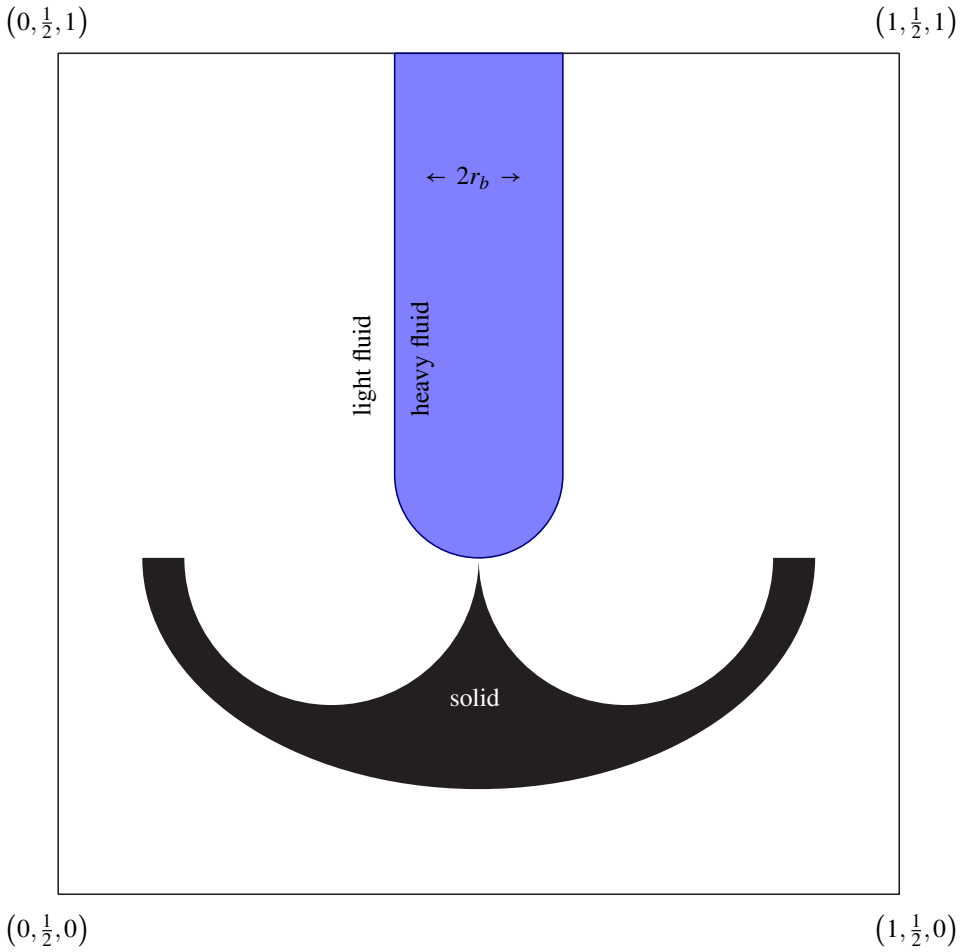



Figure 5.3: A cross section in the *xz*-plane showing the initial phase distribution.

The $z$-component of the velocity is set to a constant value of one along the vertical boundaries. The top boundary ($z = 1$) and bottom boundary ($z = 0$) have (time) constant $z$-velocity given as a function of the distance from the horizontal axis center of the computational domain (see (5.2) and Figure 5.4)

$$r = \sqrt{(x-1/2)^2 + (y-1/2)^2} \tag{5.2a}$$
$$a = \max(2r, 1) \tag{5.2b}$$
$$b = 1 - a \tag{5.2c}$$
$$w_{top} = -(a^5 + 5a^4 b) + (10a^3 b^2 + 10a^2 b^3 + 5ab^4 + b^5) \tag{5.2d}$$
$$w_{bottom} = (a^3 + 3a^2 b)/21 + 3ab^2 + b^3 \tag{5.2e}$$

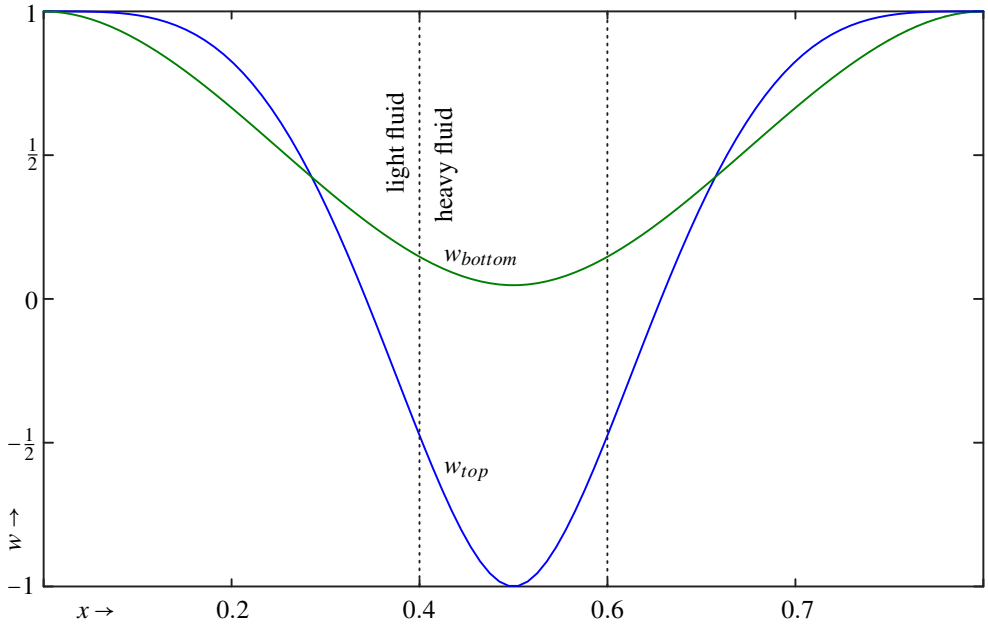



Figure 5.4: The graphs show the $z$-component of the velocity at the upper and lower boundary through the center of the computational domain.

Note that the total fluid volume in the computational domain is conserved with the boundary conditions given in (5.2), i.e. (5.3) is satisfied:

$$\int_{r=0}^{1/2} 2\pi r w_{top}(r)\mathrm{d}r = \int_{r=0}^{1/2} 2\pi r w_{bottom}(r)\mathrm{d}r \tag{5.3}$$

## 5.3 Simulation Results

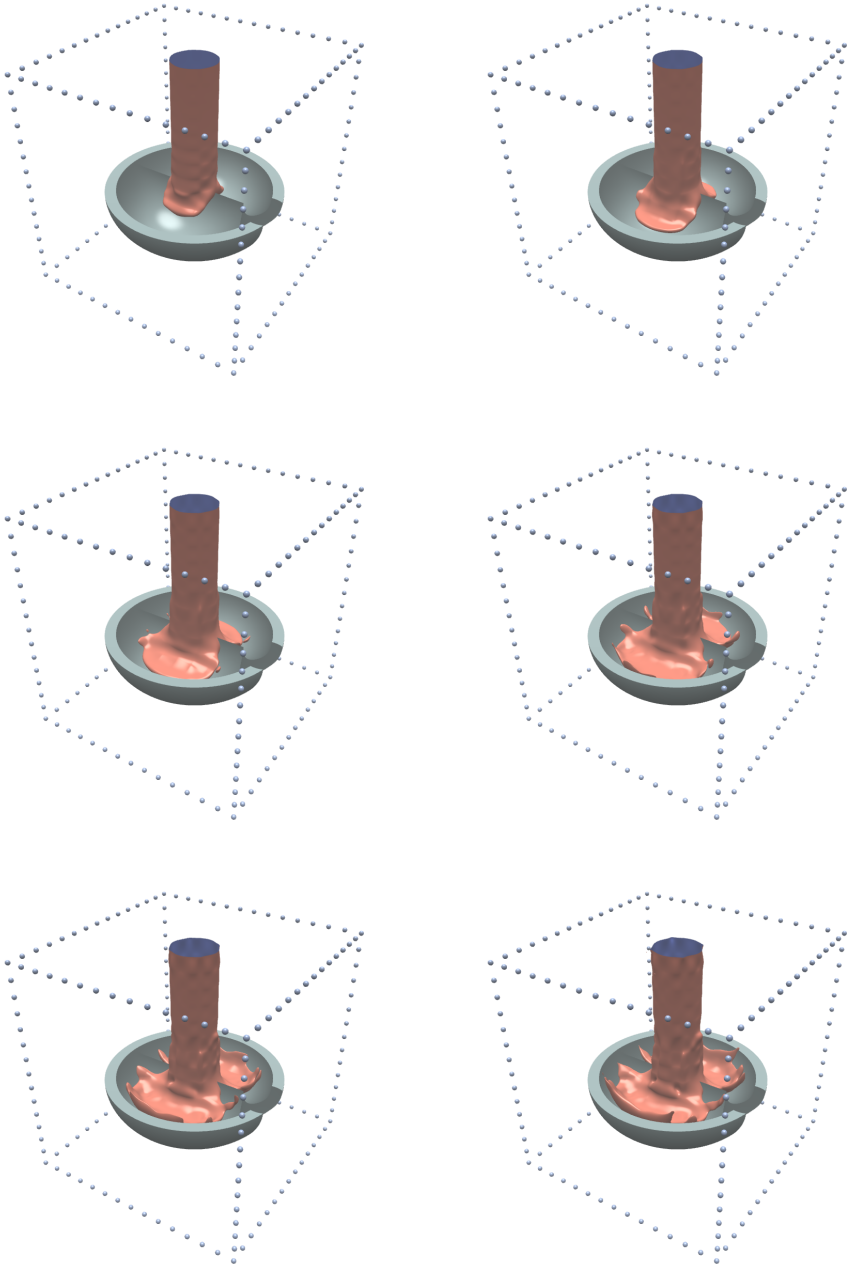


Figure 5.5: Surface Visualization at timesteps: 10,20,30,40,50 and 60 (each time-step corresponding to 1/42'th of a time unit). The visualizations are generated using ray-tracing. In the last frames surface artifacts appear.

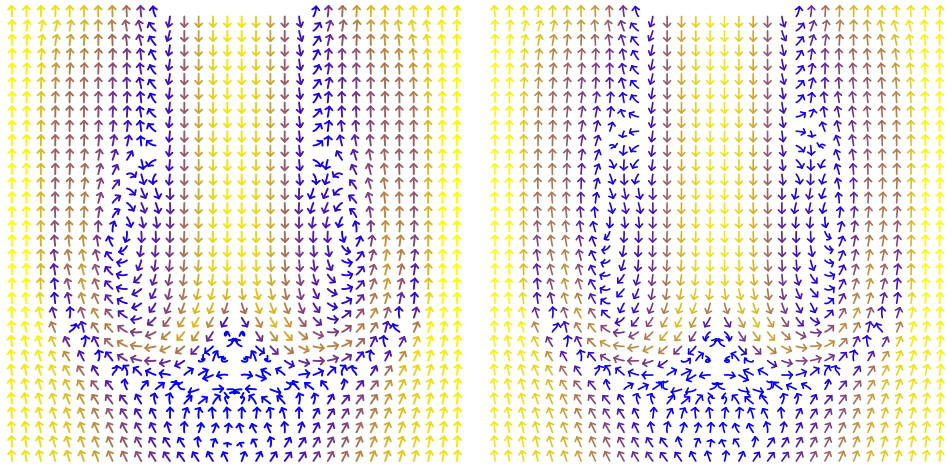


Figure 5.6: Visualization of the flow velocity through slices in the xz-plane at time-steps 60 (left) and time-step 90 (right). The arrow length is constant and the color goes from blue at zero velocity to yellow at the highest velocity.

The simulation successfully describes the splitting of the incoming beam by the bucket. As each half of the beam is deflected within the bucket it spreads out to gradually thinner layers. As a result the details of the surface geometry reaches a scale much smaller than the grid resolution. As this happens surface artifacts appear and the simulation no longer gives an good description of the flow (see Figure 5.5). The velocity configuration does not vary greatly with time and shows the expected $180^o$ deflection of the flow (see Figure 5.6). The convergence profile of the iteration is essentially the same as observed in the simulations discussed in Chapter 6.2. Each step in the iteration required typically 280 s ± 10 s (using the same hardware configuration).

## 5.4 Outlook on Pelton-Turbine Simulation

The Pelton bucket simulation shows that the method has potential use for this kind of problem, but the following steps need to be taken i) The algorithms must be optimized in terms of memory usage to allow higher resolution. ii) For a full scale turbine simulation direct numerical simulation is not feasible, instead phase averaging would be a suitable approach combined with the current method.

# Chapter 6

# Papers

## 6.1 Lid-Driven Cavity

The paper entitled *A Numerical Approach to Solving Nonlinear Differential Equations on a Grid with Potential Applicability to Computational Fluid Dynamics* [9], is an early version of the current approach. It validates the method for two dimensional steady flow by comparing it with previously obtained results from different sources [4], [3] and [11]. This paper also demonstrates the advantage of increasing the order of continuity versus grid refinement. The method has since been updated to use an optimal set of polynomial basis functions which improves the conditioning of the linear stage of the solution algorithm. The *lid-driven cavity* is defined in Chapter 3 and Figure 1 of [9].[1]

[1] Note that in [9], page 13, first paragraph a reference is made to Figure 7. It should be Figure 6.

# A Numerical Approach to Solving Nonlinear Differential Equations on a Grid with Potential Applicability to Computational Fluid Dynamics*

Jesper Tveit†

*Department of Physics and Technology, University of Bergen, Norway*
*BKK Production, Kokstad, Norway*

November 7, 2014

**Abstract**

A finite element method for solving nonlinear differential equations on a grid, with potential applicability to computational fluid dynamics (CFD), is developed and tested.

The current method facilitates the computation of solutions of a high polynomial degree on a grid. A high polynomial degree is achieved by interpolating both the value, and the value of the derivatives up to a given order, of continuously distributed unknown variables.

The two-dimensional lid-driven cavity, a common benchmark problem for CFD methods, is used as a test case. It is shown that increasing the polynomial degree has some advantages, compared to increasing the number of grid-points, when solving the given benchmark problem using the current method. The current method yields results which agree well with previously published results for this test case.

## 1 Introduction

Through development and testing in the well known case of lid-driven cavity flow (see Figure 1 for details) the current method is shown to have potential applicability to CFD. Steady state solutions of the Navier-Stokes equations for incompressible two-dimensional flow are computed for this test case with Reynolds number $\mathrm{Re} \leq 4 \times 10^4$.

Obtaining a steady-state flow solution in a two-dimensional lid-driven cavity becomes increasingly challenging as the Reynolds number increases. Computing a steady-state flow solution is therefore useful as a bench-mark for the quality of numerical schemes, although the solution does not necessarily describe a physical fluid. In order to compute a solution, which accurately represents details at high Reynolds numbers, the most successful approaches have been using very high grid resolutions [Erturk et al., 2005, Wahba, 2012]. The current approach obtains comparable results with much lower grid resolutions.

The current solutions have up to 9'th order (polynomial degree) of spatial accuracy and the highest grid resolution is 135 by 135 grid points. Several other high order solutions to the lid-driven cavity have previously been presented. Barragy and Carey [1996] present solutions for the lid-driven cavity up to $\mathrm{Re} = 12500$ (as well as an under-resolved solution for $\mathrm{Re} = 16500$)

*Previously published at arxiv.org (2014), arXiv:1409.1072 [physics.flu-dyn]
†email: jtv001@uib.no

with spatial accuracies from 6'th to 8'th order. Other works which present high order solutions include Schreiber and Keller [1982] (8'th order), and Nishida and Satofuka [1992] (10'th order).

Wahba [2012] present reliable steady state solutions of comparably high Reynolds numbers ($\mathrm{Re} \leq 35 \times 10^3$) but this is the first time reliable high order (above fourth, in polynomial degree) steady state solutions for Reynolds number, $\mathrm{Re} \geq 20000$, to the lid-driven cavity in two dimensions have been presented.

The explicit definition of derivatives employed by the current method is a feature which is partially shared by the CIP method [Takewaki et al., 1984], since the CIP method includes the gradient of unknown quantities as a free parameter. The CIP method is a third order method used successfully, for example, to simulate acoustic wave propagation.

The current method is a finite element type of approach and solves nonlinear differential equations through several steps. First a discretization is defined to contain information about the unknown functions in a given set of differential equations. This information includes both the value, and the value of the derivatives, of the unknown functions at specific positions (grid points) in a computational domain. Next, the differential equations are formulated as a nonlinear system of equations (weak form) depending on the information contained in the grid. This system of equations is solved through an iteration, which minimizes the square of a uniformly weighted residual. Each iteration has both a linear and a nonlinear stage, and finds an approximate solution that improves the previous approximate solution.

The specific details involved in each of these steps will be thoroughly explained in Sections 2 - 5. Results of particular interest will be presented in Section 6. The complete data of all the computed results can be obtained from the author upon request.

## 2 Notation and Mathematical Framework

### 2.1 Notation

Square brackets will be used to identify components in matrices. A component in a two-dimensional matrix, $\mathbf{A}$, will thus be referred to as $\mathbf{A}[r, c]$, where $r$ is the row index and $c$ is the column index. Indices in an $R \times C$ matrix are defined to go from 0 to $R-1$ (rows) and 0 to $C-1$ (columns). If $C = 1$, then the matrix may be referred to as a column vector and if $R = 1$, then the matrix may be referred to as a row-vector. A *matrix of matrices* will be equivalent to a four dimensional matrix, $\mathbf{Q}$, where $\mathbf{Q}[r, c][\tau, \nu] \equiv \mathbf{Q}[r, c, \tau, \nu]$. If a matrix, $\mathbf{A}$, is square and nonsingular, its inverse will be written as $\mathbf{A}^{-1}$.

For brevity, the evaluation of a derivative of a function, $\Phi(\zeta)$, at a certain point, $\zeta_0$, will in unambiguous cases be written as shown on the right-hand side of Eq.(1):

$$\left.\frac{\partial \Phi(\zeta)}{\partial \zeta}\right|_{\zeta=\zeta_0} \equiv \frac{\partial \Phi(\zeta_0)}{\partial \zeta} \tag{1}$$

For functions of two variables, the first and the second argument will be referred to as the $x$-, and the $y$- component (or variable), respectively.

Mapping of indices from double index form to single index form will, unless otherwise stated, be on the form $a = b + cD$, where $D$ is a positive integer and $b, c \in \{0, \ldots, D-1\}$ and $a \in \{0, \ldots, D^2 - 1\}$. The indices, $b$ and $c$, may also a single index form of other index tuples, in which case the mapping will recursively follow the given form.

## 2.2 Order of Continuity

Consider a discretization of a function, $f(x,y)$, on a uniform two-dimensional grid. Let the matrices $\mathbf{x}$ and $\mathbf{y}$ be composed of the $x$ and $y$ components, respectively, of the position of the grid points. Let the value of the function, $f(x,y)$, and its derivatives up to, and including, the $(\Omega-1)$'th order in each direction be explicitly defined for each grid point in terms of the (four dimensional) matrix, $\mathbf{F}$, with components given by Eq.(2):

$$\frac{\partial^{\alpha+\beta} f\left(\mathbf{x}\left[k,l\right],\mathbf{y}\left[k,l\right]\right)}{\partial x^{\alpha}\partial y^{\beta}} \equiv \mathbf{F}\left[k,l,\alpha,\beta\right] \tag{2}$$

where $\alpha \in \{0,\ldots,\Omega-1\}$, $\beta \in \{0,\ldots,\Omega-1\}$, and the indices, $k$ and $l$, identify the grid point. The discretization is then by definition continuous and has continuous derivatives up to $(\Omega-1)$'th order at the grid points $(\mathbf{x}\left[k,l\right],\mathbf{y}\left[k,l\right])$. This is referred to as $\mathcal{C}^{\Omega-1}$ continuity.

## 2.3 Grid Structure

For the sake of simplicity the grid will be oriented and scaled such that the location of each grid point is uniquely determined by its indices, $k$ and $l$, as shown in Eq.(3):

$$\left(\mathbf{x}\left[k,l\right],\mathbf{y}\left[k,l\right]\right) \equiv (k,l) \tag{3}$$

defining a uniform square grid.

## 2.4 Polynomial Basis-function Expansion

Let the functions, $b_{m,n}(x,y)$, be a set of polynomial basis functions where the value of $m$ is the polynomial degree of the first variable, $x$, and the value of $n$ is the polynomial degree of the second variable, $y$.

Let the matrix of column vectors, $\mathbf{f}$, the row vector-function, $\mathbf{b}(x,y)$, and the matrix, $\mathbf{B}$, be defined as shown in Eqs.(4-6), respectively:

$$\mathbf{f}\left[k,l\right]\left[\tau,0\right] \equiv \mathbf{F}\left[k+i,l+j,\alpha,\beta\right] \tag{4}$$

$$\mathbf{b}[0,m+Nn](x,y) \equiv b_{m,n}(x,y) \tag{5}$$

$$\mathbf{B}\left[\tau,m+Nn\right] \equiv \frac{\partial^{\alpha+\beta} b_{m,n}(i,j)}{\partial x^{\alpha}\partial y^{\beta}} \tag{6}$$

where $\tau = \alpha+\beta\Omega+\Omega^2(i+2j)$, $i \in \{0,1\}$, $j \in \{0,1\}$, $m \in \{0,\ldots,N-1\}$, $n \in \{0,\ldots,N-1\}$ and $N = 2\Omega$. It follows that each column vector in the matrix, $\mathbf{f}$, then has $(2\Omega)^2 = N^2$ components, that the matrix, $\mathbf{B}$, is a $N^2 \times N^2$ square matrix and that the row vector-function, $\mathbf{b}(x,y)$, has $N^2$ components.

The four neighboring points: $(\mathbf{x}\left[k,l\right],\mathbf{y}\left[k,l\right])$, $(\mathbf{x}\left[k+1,l\right],\mathbf{y}\left[k+1,l\right])$, $(\mathbf{x}\left[k,l+1\right],\mathbf{y}\left[k,l+1\right])$ and $(\mathbf{x}\left[k+1,l+1\right],\mathbf{y}\left[k+1,l+1\right])$, surround a square region which will be referred to as the $k,l$'th *grid-cell*.

Within the $k,l$'th grid-cell, the function, $f(x,y)$, may be approximated by a weighted sum of the polynomial basis functions, $b_{m,n}(x,y)$, written in matrix form in Eq.(7):

$$f(x,y) = \mathbf{b}(x',y')\mathbf{B}^{-1}\mathbf{f}\left[k,l\right] + \mathcal{O}(x'^{N}+y'^{N}) \approx \mathbf{b}(x',y')\mathbf{B}^{-1}\mathbf{f}\left[k,l\right] \tag{7}$$

where $x' = x - \mathbf{x}\left[k,l\right]$ and $y' = y - \mathbf{y}\left[k,l\right]$.

From the definitions, Eqs.(4-6), it is clear that the approximation, $\mathbf{b}(x', y')\mathbf{B}^{-1}\mathbf{f}[k, l]$, matches up exactly with the discretization, $\mathbf{F}$, at the four grid points, i.e.:

$$\mathbf{F}[k + i, l + j, \alpha, \beta] = \frac{\partial^{\alpha+\beta}\mathbf{b}(i, j)}{\partial x^{\alpha}\partial y^{\beta}}\mathbf{B}^{-1}\mathbf{f}[k, l] \tag{8}$$

where $i \in \{0, 1\}$ and $j \in \{0, 1\}$ as previously.

## 2.5 Hermite Splines

The idea of approximating a function by sampling both its value and the value of its derivatives is known as Hermite interpolation. The approximation, $\mathbf{b}(x', y')\mathbf{B}^{-1}\mathbf{f}[k, l]$, of the function, $f(x, y)$, is a two-dimensional generalization of a Hermite spline, equivalent to recursively interpolating a set of Hermite splines.

## 2.6 Choice of Basis Functions

The *condition number*, cond($\mathbf{B}$), defined in Eq.(9), gives an estimate of the relative numerical accuracy of the matrix product, $\mathbf{B}^{-1}\mathbf{f}[k, l]$.

$$\text{cond}(\mathbf{B}) \equiv \frac{\sigma_{\max}(\mathbf{B})}{\sigma_{\min}(\mathbf{B})} \tag{9}$$

In Eq.(9), $\sigma_{\max}(\mathbf{B})$ is the largest singular value of $\mathbf{B}$, and $\sigma_{\min}(\mathbf{B})$ is the smallest singular value of $\mathbf{B}$. When numerically solving a linear system, $\mathbf{f} = \mathbf{B}\mathbf{c}$, using floating point numbers with machine precision, $\epsilon_m$, an error of order $\mathcal{O}(\epsilon_m \text{cond}(\mathbf{B}))$ should be expected. The reader may refer to Trefethen and Bau [1997, pg. 95] for a more detailed explanation of the condition number and numerical accuracy of linear equation systems.

The condition number, cond($\mathbf{B}$), depends on the choice of basis functions, $b_{m,n}(x, y)$, and on the order of continuity (in other words, the value of $\Omega$).

For the computations in this paper, the basis functions, $b_{m,n}(x, y)$, are defined in terms of the Bernstein polynomials, $\mathcal{B}_{\lambda,\Lambda}(x)$, given in Eq.(10):

$$\mathcal{B}_{\lambda,\Lambda}(x) = \binom{\Lambda}{\lambda} x^{\lambda}(1 - x)^{\Lambda-\lambda},\ \lambda \in \{0, \ldots, \Lambda\} \tag{10}$$

as

$$b_{m,n}(x, y) \equiv \mathcal{B}_{m,\Omega-1}(x)\mathcal{B}_{n,\Omega-1}(y) \tag{11}$$

Table 1 shows that the condition number of the matrix, $\mathbf{B}$, increases exponentially with the value of $\Omega$. The machine precision is a limiting factor for the computation of the function approximation, Eq.(7). For higher orders of continuity, it may be considered an ill-conditioned system. As a result one should not expect the coefficients of the function approximation, Eq.(7), to be accurate down to machine epsilon. For this reason, the main results presented in this paper have been computed using 64 bit floating point numbers (*double* in C++ syntax) rather than the more common 32 bit float. The corresponding ISO C standard definition of machine epsilon is $2^{-52} \approx 2.2 \times 10^{-16}$, referred to in this paper as $\epsilon_{64}$.

| $\Omega$ | $\epsilon_{64}\text{cond}\,(\mathbf{B})$ with $b_{m,n}(x,y)$ | $\epsilon_{64}\text{cond}\,(\mathbf{B})$ with $x^m y^n$ |
|---|---|---|
| 2 | $8.5 \times 10^{-15}$ | $1.3 \times 10^{-13}$ |
| 3 | $1.7 \times 10^{-12}$ | $1.3 \times 10^{-10}$ |
| 4 | $8.2 \times 10^{-10}$ | $5.0 \times 10^{-7}$ |
| 5 | $7.5 \times 10^{-7}$ | $4.4 \times 10^{-3}$ |
| 6 | $1.1 \times 10^{-3}$ | 23.5 |

Table 1: Estimated floating point errors for different values of $\Omega$. The matrix, $\mathbf{B}$, is defined in Eq.(6) and the condition number, $\text{cond}\,(\mathbf{B})$, is defined by Eq.(9). This is the estimated precision of the numerical computation of the quantity $\mathbf{B}^{-1}\mathbf{f}$ using 64 bit floating point numbers with machine precision, $\epsilon_{64} \approx 2.2 \times 10^{-16}$, (ISO C standard). The first column shows different values of $\Omega$, corresponding to $\mathcal{C}^{\Omega-1}$ continuity. The second column shows, $\epsilon_{64}\text{cond}\,(\mathbf{B})$, constructed with the basis functions, $b_{m,n}(x,y)$, as given by Eq.(11). The third column shows what the expected precision would be if the monomial basis functions, $x^m y^n$, of equal degree were used to construct the matrix $\mathbf{B}$ instead of $b_{m,n}(x,y)$.

## 3 Discretization the Navier–Stokes Equations

### 3.1 Navier–Stokes Equations for Steady State Incompressible Flow in Two Dimensions

The grid has $L \times L$ grid-points at positions defined in Eq.(3). The indices, $k$ and $l$, then have values ranging from 0 to $L-1$ and the grid is square with length and width equal to $L-1$. The computational domain is thus the two dimensional interval $[0, L-1] \times [0, L-1]$ (see Figure 1).

The Navier–Stokes Equations for steady state incompressible flow in two dimensions, where the physical variables have been scaled with appropriate scales, read

$$0 = u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + \frac{\partial p}{\partial x} - \frac{1}{\text{Re}'}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{12a}$$

$$0 = u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + \frac{\partial p}{\partial y} - \frac{1}{\text{Re}'}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \tag{12b}$$

$$0 = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \tag{12c}$$

where $u$ and $v$ are the $x$– and $y$–components of the flow velocity, respectively, $p$ is the pressure and $\text{Re}' = \text{Re}/(L-1)$. The definition of the Reynolds number, Re, is the same as in the references [Ghia et al., 1982, Erturk et al., 2005, Wahba, 2012]. However, since the the computational domain used by the references is the two dimensional interval $[0,1] \times [0,1]$, the Reynolds number, Re, must be scaled with the size of the current domain, $L-1$, so that Eqs.(12a-12c) remain mathematically equivalent to the equations solved in the references.

The variables $u$, $v$ and $p$ will be referred to as the flow-variables and are functions of the two variables $x$ and $y$. Additionally, the pair $(u, v)$ will be referred to as the flow-velocity.

The pressure-velocity form of the Navier–Stokes equations (Eqs.(12a-12c)) is usually transformed into an equivalent *vorticity-streamfunction* form which, in the two dimensional case, has one less flow variable to deal with. Most published papers dealing with the lid-driven cavity in two dimensions use the vorticity-streamfunction form. In the current work, however, we solve for the pressure and velocity directly.

### 3.2 Boundary Conditions

The no-slip Dirichlet boundary condition is imposed on the flow velocity, $(u(x,y), v(x,y))$. Figure 1 shows the details of the boundary of the computational domain of the grid. This test case is known as the lid-driven cavity for two dimensions. No boundary values are required for the pressure during the iterative solution process. The origin of the pressure is implicitly determined by the initial condition for the iteration (see Section 5).

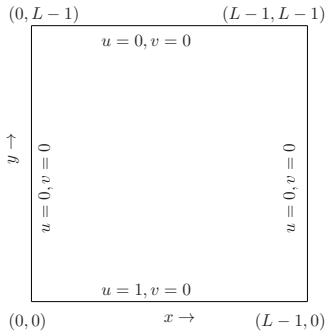



Figure 1: This figure shows the computational domain of the grid and its boundary conditions. The flow velocity, $u(x,y)$ and $v(x,y)$, along the edges is given. The lower edge, where $y = 0$, is a "lid" which drives the flow by sliding horizontally (positive $x$- direction) at a constant speed equal to one. At the three remaining edges, both components of the flow velocity are zero. This system is referred to as a lid-driven cavity.

### 3.3 Navier-Stokes Equations in Matrix Form

The flow-variables, velocity and pressure, are discretized as shown in Section 2, with $\mathbf{U}$,$\mathbf{V}$, and $\mathbf{P}$ being the discrete counterparts of $u(x,y)$, $v(x,y)$ and $p(x,y)$, respectively. In each grid-cell, the flow variables are approximated by

$$u(x,y) = \mathbf{b}(x',y')\mathbf{B}^{-1}\mathbf{u}\,[k,l] + \mathcal{O}(x'^N + y'^N) \tag{13a}$$
$$v(x,y) = \mathbf{b}(x',y')\mathbf{B}^{-1}\mathbf{v}\,[k,l] + \mathcal{O}(x'^N + y'^N) \tag{13b}$$
$$p(x,y) = \mathbf{b}(x',y')\mathbf{B}^{-1}\mathbf{p}\,[k,l] + \mathcal{O}(x'^N + y'^N) \tag{13c}$$

where $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{p}$ are defined in terms of $\mathbf{U}$,$\mathbf{V}$, and $\mathbf{P}$ in the same way as $\mathbf{f}$ was defined in terms of $\mathbf{F}$ in Section 2.

Let the row vector-functions, $\mathbf{c}_{\alpha,\beta}(x,y)$ and $\mathbf{s}(x,y)$, and the matrix of row vector-functions, $\mathbf{m}(x,y)$, be defined as shown in Eq.(14), Eq.(15) and Eq.(16):

$$\mathbf{c}_{\alpha,\beta}(x,y) \equiv \frac{\partial^{\alpha+\beta}\mathbf{b}(x,y)}{\partial x^\alpha \partial y^\beta}\mathbf{B}^{-1} \tag{14}$$

$$\mathbf{s}(x,y) \equiv -\frac{L-1}{\mathrm{Re}}\left(\mathbf{c}_{2,0}(x,y) + \mathbf{c}_{0,2}(x,y)\right) \tag{15}$$

$$\mathbf{m}\,[k,l]\,(x,y) \equiv \left(\mathbf{c}_{0,0}(x,y)\mathbf{u}[k,l]\right)\mathbf{c}_{1,0}(x,y) + \left(\mathbf{c}_{0,0}(x,y)\mathbf{v}[k,l]\right)\mathbf{c}_{0,1}(x,y) + \mathbf{s}(x,y) \tag{16}$$

The Navier–Stokes equations, Eqs.(12a-12c), are then approximated, within a grid-cell as

$$0 = \mathbf{m}[k,l]\mathbf{u}[k,l] + \mathbf{c}_{1,0}\mathbf{p}[k,l] + \mathcal{O}(x^{N-2} + y^{N-2}) \tag{17a}$$

$$0 = \mathbf{m}[k,l]\mathbf{v}[k,l] + \mathbf{c}_{0,1}\mathbf{p}[k,l] + \mathcal{O}(x^{N-2} + y^{N-2}) \tag{17b}$$

$$0 = \mathbf{c}_{1,0}\mathbf{u}[k,l] + \mathbf{c}_{0,1}\mathbf{v}[k,l] + \mathcal{O}(x^{N-1} + y^{N-1}) \tag{17c}$$

As indicated in Eqs.(17a-17c), the formal polynomial order of accuracy is reduced due to the differentation with respect to $x$ and $y$. From now on, the indication of polynomial order of accuracy, $\mathcal{O}(\dots)$, will be omitted.

Eqs.(17a-17c), can be written in the form of a single matrix equation, as shown in Eq.(18):

$$\mathbf{0} = \mathbf{E}[k,l](x,y)\mathbf{z}[k,l] \tag{18}$$

where the $3 \times 3N^2$ matrix-functions, $\mathbf{E}[k,l](x,y)$, and the $3N^2 \times 1$ column vectors, $\mathbf{z}[k,l]$, are defined by Eq.(19) and Eq.(20), respectively, as:

$$\mathbf{E}[k,l](x,y) \equiv \begin{bmatrix} \mathbf{m}[k,l](x,y) & \mathbf{0} & \mathbf{c}_{1,0}(x,y) \\ \mathbf{0} & \mathbf{m}[k,l](x,y) & \mathbf{c}_{0,1}(x,y) \\ \mathbf{c}_{1,0}(x,y) & \mathbf{c}_{0,1}(x,y) & \mathbf{0} \end{bmatrix} \tag{19}$$

and

$$\mathbf{z}[k,l] \equiv \begin{bmatrix} \mathbf{u}[k,l] \\ \mathbf{v}[k,l] \\ \mathbf{p}[k,l] \end{bmatrix} \tag{20}$$

### 3.4 Boundary Components of the Grid

In accordance with the definition of the grid structure, Eq.(3), and the boundary conditions for the lid-driven cavity (Figure 1), a grid component of the flow velocity, $\mathbf{U}[k,l,\alpha,\beta]$ or $\mathbf{V}[k,l,\alpha,\beta]$, will be defined to be a constant *boundary component* if $(l = 0 \vee l = L-1) \wedge \beta = 0$ (boundary parallel to the $x$-axis) or if $(k = 0 \vee k = L-1) \wedge \alpha = 0$ (boundary parallel to the $y$-axis). Note that *derivatives, parallel to the boundary, at the boundary* are also defined as boundary components. The values of the boundary components are all zero, except in the cases given by Eq.(21) and Figure 2. The components which are not defined to be boundary components will be referred to as *internal components* or as *internal flow components*.

$$\mathbf{U}\left[k,0,0,0\right] = \begin{cases} 1, & \text{for } \ 0 < k < L-1 \\ \frac{1}{2}, & \text{for } \ k \in \{0, L-1\} \end{cases} \tag{21}$$

## 4 Linear Approximation of the Navier-Stokes Equations

### 4.1 Integrated Error-Squared

In this subsection (4.1), the grid-cell indices, $[k,l]$, will be omitted occasionally for the sake of brevity. Unless otherwise stated, the equations and definitions will be understood to correspond to a single, arbitrary grid-cell.

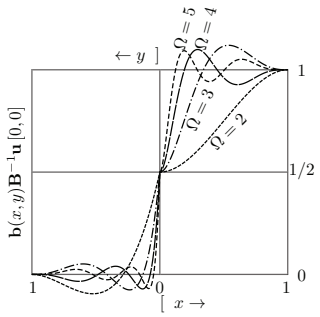



Figure 2: This figure shows how the boundary value of the $x$-component of the velocity, $u(x, y)$, is approximated near the lower left corner ($x = y = 0$) of the boundary for different values of $\Omega$. The graphs on the right hand side of the figure are for $y = 0$ out to the nearest grid point along the $x$-axis. The graphs on the left hand side of the figure are for $x = 0$ out the to the nearest grid point along the $y$-axis. By using a standard least squares algorithm (see for example Howard [2000, pg. 437]), the components $\mathbf{U}[0, 0, \{2, \ldots, \Omega - 1\}, 0]$ are set to produce the best fit to the unit boundary condition and the components $\mathbf{U}[0, 0, 0, \{1, \ldots, \Omega - 1\}]$ are set to produce the best fit to the zero boundary condition. Note that the derivative of $u$ with respect to $x$ at the corner, $\mathbf{U}[0, 0, 1, 0]$, must be zero since the derivative of $v$ with respect to $y$ at the corner, $\mathbf{V}[0, 0, 0, 1]$, is zero (otherwise the continuity equation, Eq.(12c), would not be satisfied at these points). The velocity near the lower right corner is modeled in the same way (mirrored along the $x$-axis).

The error-squared, $R^2$, for each grid-cell will be defined, using the approximated Navier-Stokes equations on matrix form (Eq.(18)), as

$$R^2 \equiv \frac{1}{2} \int_0^1 \int_0^1 \mathbf{z}^T \mathbf{E}^T(x, y) \mathbf{E}(x, y) \mathbf{z} \, \mathrm{d}x\mathrm{d}y \tag{22}$$

A linear approximation of the derivatives of the error-squared, $R^2[k, l]$, with respect to the components of the column vector, $\mathbf{z}$, is

$$\frac{\partial R^2}{\partial \mathbf{z}} \approx \left( \int_0^1 \int_0^1 \mathbf{E}^T(x, y) \mathbf{E}(x, y) \, \mathrm{d}x\mathrm{d}y \right) \mathbf{z} \tag{23}$$

The derivative, given in Eq.(23), is an approximation because the matrix, $\mathbf{E}$, is taken to be constant (while, in fact, it depends on the flow velocity through its dependence on the matrix, $\mathbf{m}[k, l]$).

It is possible to compute the integral, Eq.(23), analytically. But, by using numerical integration, the following procedure is more flexible (with future modifications and extensions in mind).

Let the points, $(x_s, y_s)$, for $s \in \{0, \ldots, S^2 - 1\}$ form a uniform set of sample positions, defined in Eq.(24):

$$(x_s, y_s) \equiv \left( \frac{1 + s_x}{1 + S}, \frac{1 + s_y}{1 + S} \right) \tag{24}$$

where $s = s_x + s_y S$ and $s_x, s_y \in \{0, \ldots, S-1\}$. From Eq.(24) it is clear that $0 < x_s < 1$ and $0 < y_s < 1$. The approximation of the derivatives of the error-squared, $R^2[k,l]$, with respect to the components of the column vector, $\mathbf{z}[k,l]$, where the integral has been replaced by a sum over the samples, $(x_s, y_s)$, read:

$$\frac{\partial R^2}{\partial \mathbf{z}} \approx \left( \frac{1}{S^2} \sum_{s=0}^{S^2-1} \mathbf{E}^T(x_s, y_s) \mathbf{E}(x_s, y_s) \right) \mathbf{z} \tag{25}$$

### 4.2 Sub-cell System to Grid-wide System

Let the $(L-1) \times (L-1) \times 3N^2 \times 3N^2$ matrix, $\mathbf{R}$, be defined by Eq.(26):

$$\frac{1}{S^2} \sum_{s=0}^{S^2-1} \mathbf{E}[k,l]^T(x_s, y_s) \mathbf{E}[k,l](x_s, y_s) \equiv \mathbf{R}[k,l] \tag{26}$$

with the matrix, $\mathbf{E}$, as defined in Eq.(19). The definition, Eq.(26), implies that the the $3N^2 \times 3N^2$ matrices, $\mathbf{R}[k,l]$, are symmetric. The grid-cell system, Eq.(25), may be written as shown in Eq.(27):

$$\frac{\partial R^2[k,l]}{\partial \mathbf{z}} \approx \mathbf{R}[k,l] \mathbf{z}[k,l] \tag{27}$$

To find an approximate minimum of the error-squared, $R^2[k,l]$, for all the grid-cells, we formulate the equation system given in Eq.(28) from which a solution for the internal flow components is implied.

$$\mathbf{R}[k,l] \mathbf{z}[k,l] = \mathbf{0} \tag{28}$$

Recall that the matrices, $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{p}$ (and thus also $\mathbf{z}$), are defined in terms of the three four-dimensional matrices $\mathbf{U}$,$\mathbf{V}$, and $\mathbf{P}$. From the definition given in Eq.(4) it is clear that there is an overlap between some of the components in the vectors, $\mathbf{u}[k,l]$, $\mathbf{v}[k,l]$ and $\mathbf{p}[k,l]$, for different values of the cell indices, $k$ and $l$ (for example, the components $\mathbf{u}[k,l][\alpha + \beta\Omega + 3\Omega^2, 0]$ and $\mathbf{u}[k+1, l+1][\alpha + \beta\Omega, 0]$, with $\alpha, \beta \in \{0, \ldots, \Omega - 1\}$, correspond to the same components in the matrix, $\mathbf{U}$, and are then by definition equal). The system given in Eq.(28) can thus not be solved independently for each grid-cell but must be solved for the entire grid.

In order to employ efficient techniques for solving linear systems, it is convenient to formulate the set of systems for each grid cell, Eq.(28), into a single system for the entire grid, given in Eq.(29):

$$\mathbf{\Pi w} = \mathbf{t} \tag{29}$$

where $\mathbf{\Pi}$ is a square, symmetric matrix and $\mathbf{w}$ and $\mathbf{t}$ are column vectors. This is done by defining a one to one index mapping from all the internal flow components to the components in the column vector, $\mathbf{w}$. Coefficients of the components in the grid cell systems (i.e. components in the symmetric matrix, $\mathbf{R}[k,l]$ in Eq.(28)), are added to $\mathbf{\Pi}$ if they correspond to internal components. If a component of $\mathbf{z}[k,l]$ is a boundary component, then it is multiplied with the corresponding row in $\mathbf{R}[k,l]$ and subtracted, forming the right hand vector, $\mathbf{t}$, in Eq.(29). This procedure is shown in detail by Algorithm 1.

It is convenient to arrange components so that the matrix, $\mathbf{\Pi}$, gets a narrow band structure, allowing more efficient computations on the system. For computations presented in this paper the order is arranged by first sorting the internal flow components according to their location in the grid (indices $k,l$), then by what type of flow component ($x$-velocity, $y$-velocity or pressure), then by the order of the derivative (indices $\alpha, \beta$).

Note that Algorithm 1 shows the entire matrix, $\mathbf{\Pi}$, being assembled. In the implementation of this algorithm the sub-diagonal elements are not stored since the matrix is symmetric.

**Algorithm 1** This pseudo-code shows the details of how the set of grid-cell systems (Eq.(28)) is reformulated into the system given by Eq.(29). The variables, $k$, $l$, $m$, $n$, $L$ and $N$ are all integers following their previous definitions from Section 2. The temporary variables $r$ and $c$ are also integers and contain the row and column indices for the matrix $\mathbf{\Pi}$. Internal flow components are ordered, first by the grid point location, then by what type of flow component, and then by the order of the derivative, into a contiguous list. The method, index(...), returns the position in this list if its arguments correspond to a an internal component. Otherwise, a negative number is returned.

```
Π ← 0, t ← 0
for 0 ≤ k < L, 0 ≤ l < L do
    for 0 ≤ m < 3N² do
        r ← index(k, l, m)
        if 0 ≤ r then
            for 0 ≤ n < 3N² do
                c ← index(k, l, n)
                if 0 ≤ c then
                    Π[r, c] ← Π[r, c] + R[k, l][m, n]
                else
                    t[r] ← t[r] − R[k, l][m, n]z[m, 0]
                end if
            end for
        end if
    end for
end for
```

## 5 Iterative Solution of the Nonlinear System of Equations

The Navier-Stokes equations on matrix form, Eq.(18), are solved by an iteration over several stages. Initially the internal flow components are either set to zero (velocity) and one (pressure) or corresponding to a solution for a lower Reynolds number, forming an initial approximate matrix, $\mathbf{\Pi}_0$, and an approximate solution, $\mathbf{w}_0$, of the system given in Eq.(29).

### 5.1 Linear Substep

At the $(\kappa-1)$'th iteration, the system, Eq.(29), is formed using the approximate values, $\mathbf{w}_{\kappa-1}$. A new approximate solution, $\mathbf{w}'_{\kappa-1}$, is found using the linear conjugate gradient iteration [see for example Trefethen and Bau, 1997, chap. 38]. The linear conjugate gradient iteration is terminated when the relative improvement factor, $\hat{r}_{\kappa-1}$, defined in Eq.(30), of the solution of the linear system reaches a predetermined value, $\hat{r}_{\kappa-1} \leq \hat{\omega} \ll 1$.

$$\frac{\left\|\mathbf{\Pi}_{\kappa-1}\mathbf{w}'_{\kappa-1} - \mathbf{t}_{\kappa-1}\right\|_2}{\left\|\mathbf{\Pi}_{\kappa-1}\mathbf{w}_{\kappa-1} - \mathbf{t}_{\kappa-1}\right\|_2} \equiv \hat{r}_{\kappa-1} \tag{30}$$

The approximation, $\mathbf{w}'_{\kappa-1}$, is used to define a search direction, $\Delta\mathbf{w}_{\kappa-1}$, as shown in Eq.(31):

$$\Delta\mathbf{w}_{\kappa-1} \equiv \mathbf{w}'_{\kappa-1} - \mathbf{w}_{\kappa-1} \tag{31}$$

The search direction is defined to have corresponding grid-cell components given by Eq.(32):

$$\Delta\mathbf{z}_{\kappa-1}[k,l] = \mathbf{z}'_{\kappa-1}[k,l] - \mathbf{z}_{\kappa-1}[k,l] \tag{32}$$

where the mapping from $\mathbf{z}'_{\kappa-1}$ to $\mathbf{w}'_{\kappa-1}$ is the same as used in Algorithm 1 for internal components. If a component, $\mathbf{z}'_{\kappa-1}[k,l][\tau,0]$, corresponds to a boundary component, then it is defined to be equal to its initial value, $\mathbf{z}_{\kappa-1}[k,l][\tau,0]$, giving $\Delta\mathbf{z}_{\kappa-1}[k,l][\tau,0]=0$ for boundary components. The updated flow components, $\mathbf{z}_\kappa[k,l]$, are given by Eq.(33):

$$\begin{bmatrix}\mathbf{u}_\kappa[k,l]\\ \mathbf{v}_\kappa[k,l]\\ \mathbf{p}_\kappa[k,l]\end{bmatrix}=\begin{bmatrix}\mathbf{u}_{\kappa-1}[k,l]\\ \mathbf{v}_{\kappa-1}[k,l]\\ \mathbf{p}_{\kappa-1}[k,l]\end{bmatrix}+\begin{bmatrix}\theta_u\Delta\mathbf{u}_{\kappa-1}[k,l]\\ \theta_v\Delta\mathbf{v}_{\kappa-1}[k,l]\\ \theta_p\Delta\mathbf{p}_{\kappa-1}[k,l]\end{bmatrix} \tag{33}$$

where $\theta_u$, $\theta_v$ and $\theta_p$ are three parameters to be determined in each iterative step and

$$\mathbf{z}_\kappa[k,l]\equiv\begin{bmatrix}\mathbf{u}_\kappa[k,l]\\ \mathbf{v}_\kappa[k,l]\\ \mathbf{p}_\kappa[k,l]\end{bmatrix},\ \mathbf{z}_{\kappa-1}[k,l]\equiv\begin{bmatrix}\mathbf{u}_{\kappa-1}[k,l]\\ \mathbf{v}_{\kappa-1}[k,l]\\ \mathbf{p}_{\kappa-1}[k,l]\end{bmatrix},\ \Delta\mathbf{z}_{\kappa-1}[k,l]\equiv\begin{bmatrix}\Delta\mathbf{u}_{\kappa-1}[k,l]\\ \Delta\mathbf{v}_{\kappa-1}[k,l]\\ \Delta\mathbf{p}_{\kappa-1}[k,l]\end{bmatrix} \tag{34}$$

in accordance with the definition given in Eq.(20).

## 5.2 Nonlinear Substep

Consider the integrand of the grid-cell residual squared (Eq.(22)) at the $\kappa$'th stage:

$$\mathbf{z}_\kappa^T[k,l]\mathbf{E}_\kappa^T[k,l](x,y)\mathbf{E}_\kappa[k,l](x,y)\mathbf{z}_\kappa[k,l]\equiv\rho_\kappa[k,l](\theta_u,\theta_v,\theta_p,x,y) \tag{35}$$

According to Eq.(33), the column vector, $\mathbf{z}_\kappa$, depends linearly on the parameters $\theta_u$, $\theta_v$ and $\theta_p$, and the matrix, $\mathbf{E}_\kappa(x,y)$, depends linearly on the parameters $\theta_u$ and $\theta_v$. Eq.(35) can thus be written as a fourth degree polynomial of $\theta_u$, $\theta_v$ and $\theta_p$, shown in Eq.(36)

$$\begin{aligned}\rho_\kappa[k,l](\theta_u,\theta_v,\theta_p,x,y)=\ & c_{000}+c_{100}\theta_u+c_{200}\theta_u^2+c_{300}\theta_u^3+c_{400}\theta_u^4+\\ & c_{010}\theta_v+c_{110}\theta_u\theta_v+c_{210}\theta_u^2\theta_v+c_{310}\theta_u^3\theta_v+\\ & c_{020}\theta_v^2+c_{120}\theta_u\theta_v^2+c_{220}\theta_u^2\theta_v^2+\\ & c_{030}\theta_v^3+c_{130}\theta_u\theta_v^3+\\ & c_{040}\theta_v^4+\\ & c_{001}\theta_p+c_{101}\theta_u\theta_p+c_{201}\theta_u^2\theta_p+\\ & c_{011}\theta_v\theta_p+c_{111}\theta_u\theta_v\theta_p+\\ & c_{021}\theta_v^2\theta_p+c_{002}\theta_p^2\end{aligned} \tag{36}$$

The coefficients, $c_{\ldots}[k,l](x,y)$, in Eq.(36) are determined from the definition of $\mathbf{E}[k,l](x,y)$ (see Eq.(19)) and $\mathbf{m}[k,l](x,y)$ (see Eq.(16)) through basic algebraic operations by substituting $\mathbf{u}_{\kappa-1}+\theta_u\Delta\mathbf{u}_{\kappa-1}$ for $\mathbf{u}$, $\mathbf{v}_{\kappa-1}+\theta_v\Delta\mathbf{v}_{\kappa-1}$ for $\mathbf{v}$ and $\mathbf{p}_{\kappa-1}+\theta_p\Delta\mathbf{p}_{\kappa-1}$ for $\mathbf{p}$ (the grid cell indices $[k,l]$ and function arguments $(x,y)$ for the coefficients, $c_{\ldots}[k,l](x,y)$, are omitted in Eq.(36) for the sake of brevity). Eq.(36) is integrated numerically over the entire grid by the sum given in Eq.(37):

$$P(\theta_u,\theta_v,\theta_p)\equiv\frac{1}{W}\sum_{\substack{k=0\\ l=0}}^{L-2}\sum_{s=0}^{S^2-1}\rho_\kappa[k,l](\theta_u,\theta_v,\theta_p,x_s,y_s) \tag{37}$$

where $W = S^2(L-1)^2$, yielding

$$\begin{aligned} P(\theta_u,\theta_v,\theta_p) = & C_{000} + C_{100}\theta_u + C_{200}\theta_u^2 + C_{300}\theta_u^3 + C_{400}\theta_u^4 + \\ & C_{010}\theta_v + C_{110}\theta_u\theta_v + C_{210}\theta_u^2\theta_v + C_{310}\theta_u^3\theta_v + \\ & C_{020}\theta_v^2 + C_{120}\theta_u\theta_v^2 + C_{220}\theta_u^2\theta_v^2 + \\ & C_{030}\theta_v^3 + C_{130}\theta_u\theta_v^3 + \\ & C_{040}\theta_v^4 + \\ & C_{001}\theta_p + C_{101}\theta_u\theta_p + C_{201}\theta_u^2\theta_p + \\ & C_{011}\theta_v\theta_p + C_{111}\theta_u\theta_v\theta_p + \\ & C_{021}\theta_v^2\theta_p + C_{002}\theta_p^2 \end{aligned} \tag{38}$$

The function, $P(\theta_u,\theta_v,\theta_p)$, is then minimized with respect to the parameters $\theta_u,\theta_v,\theta_p$. The minimization of Eq.(38) is not a computationally expensive step since the function, $P(\theta_u,\theta_v,\theta_p)$, is a fourth degree polynomial depending on only three variables. For the purposes of this paper, the nonlinear conjugate gradient iteration was sufficient. A fixed number of iterations (50) was used and the Fletcher-Reeves method determined the line search direction (the reader may refer to Shewchuk [1994] for details concerning the nonlinear conjugate gradient iteration).

With the parameters $\theta_u,\theta_v,\theta_p$ determined, the flow components are updated as shown in Eq.(33) and the iteration may be repeated until desired accuracy is reached or until errors, due to limited floating point precision or due to the approximate nature of the discretization, prevents further improvement.

## 6 Results

Solutions were computed for Reynolds numbers, $\mathrm{Re} \in \{100, 1000, 5000, 10000, 20000, 30000, 40000\}$. The data from all the computations is too extensive to be displayed in detail in this paper but is available from the author upon request. In Subsections 6.1-6.3 details of a selection of the computed solutions are discussed.

### 6.1 Velocity Profiles

The $x$-component of the velocity, $u(x,y)$, through the geometric center of the cavity, from the center of the "lid", $(x = (L-1)/2, y = 0)$, to the opposing side, $(x = (L-1)/2, y = L-1)$, is shown in Figures 3-5. This will simply be referred to as a *velocity profile* from now on.

For Reynolds number, $\mathrm{Re} = 100$, the well known results from Ghia et al. [1982] are used as a comparison (Figure 3). For Reynolds number, $\mathrm{Re} = 20000$, the current results are compared[1] with the very fine-grid solutions from Erturk et al. [2005] and Wahba [2012] (Figure 4). Additionally, the interesting features for various high Reynolds numbers from $\mathrm{Re} = 5000$ to $\mathrm{Re} = 40000$ is compared with each other (Figure 5).

Figure 3, which shows the velocity profile for Reynolds number, $\mathrm{Re} = 100$, confirms that the current results agree with established results [Ghia et al., 1982] for this Reynolds number.

Figure 4, which shows the velocity profile for Reynolds number, $\mathrm{Re} = 20000$, shows a small deviance from the reference solutions by Erturk et al. [2005] and Wahba [2012]. In this case the current solution tends to agree with the references where they coincide and tends to lie between the references where they do not coincide.

[1] Reynolds number, $\mathrm{Re} = 20000$, was the highest Reynolds number for which multiple reference results were available.

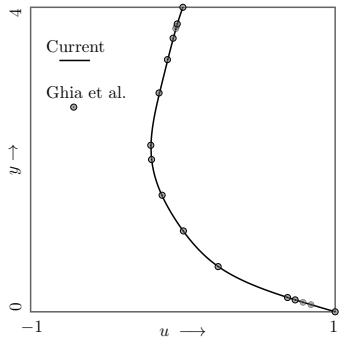



Figure 3: This figure shows the computed $x$-component of the velocity on a vertical line through the geometric center of the grid for Reynolds number, Re = 100. The line shows current results, $\mathbf{b}(x', y')\mathbf{B}^{-1}\mathbf{u}[k, l]$ for $x = (L-1)/2,\ 0 \leq y \leq L-1$, with $L = 5$ and $\Omega = 4$ where $x' = x - \mathbf{x}[k, l]$ and $y' = y - \mathbf{y}[k, l]$. The dotted circles show results presented by Ghia et al. [1982] as a comparison.

Figure 5 shows how the upper and lower parts of the velocity profile evolve as the Reynolds number increases. The upper part shows a systematic trend where minimum drops while shifting increasingly closer to the edge. The lower part shows a similar trend for Re = 5000 to Re = 20000. However from Re = 20000 to Re = 40000, the local minimum move toward the edge at a much smaller rate while increasing in magnitude at a greater rate.

Figure 7 shows the velocity profiles for Reynolds number, Re = 1000, obtained with increasing values of $\Omega$ and decreasing values of $L$, compared with the results given by Ghia et al. [1982], Erturk et al. [2005]. The value of $L$ was the lowest value which did not give any significant deviance from solutions obtained using a higher resolution. These results illustrate how an increase of the order of continuity allows for a lower grid resolution while still achieving results of similar accuracy. Also note that the amount of data contained in the grid (proportional to $L^2\Omega^2$ in a two dimensional grid) decreases with increasing values of $\Omega$.

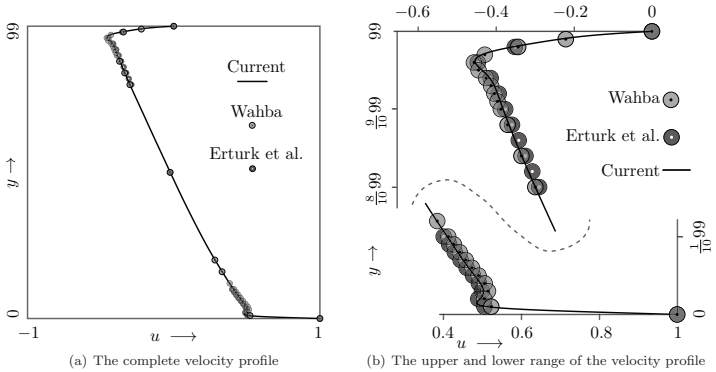
### 6.2 Flow Configurations

Figure 7 shows visualizations of computed flow configurations. Due to the high polynomial degree of the solution, flow features below grid resolution are resolved. This can be seen in the close-up plot in Subfigure 7(b). At higher Reynolds numbers the required grid resolution is higher compared to the scale of the main vortices of the flow. However, secondary, tertiary and quaternary vortices all split up in several sub-vortices at high Reynolds numbers. It seems reasonable to assume that the higher resolution requirement is connected to this phenomenon.

### 6.3 Computation Time and Convergence

The computation times were achieved on a standard desktop computer (quad core Xeon W3565 CPU at 3.2 GHz with 12 GB RAM). The computation times are not comparable to what might be achieved on a high end system utilizing parallel computing, but might have some use for internal comparison. Up to four separate computations were run simultaneously (each single threaded) each utilizing 23-25 percent of the CPU capacity.

Figure 8 shows examples of accuracy versus computation time for different Reynolds numbers,




(a) The complete velocity profile

(b) The upper and lower range of the velocity profile

Figure 4: These figures show the computed $x$-component of the velocity on a vertical line through the geometric center of the grid for Reynolds number, Re = 20000. The line shows current results, $\mathbf{b}(x', y')\mathbf{B}^{-1}\mathbf{u}[k, l]$ for $x = (L-1)/2$, $0 \leq y \leq L-1$, with $L = 100$ and $\Omega = 5$ where $x' = x - \mathbf{x}[k, l]$ and $y' = y - \mathbf{y}[k, l]$. The dotted circles show results presented by Erturk et al. [2005] and Wahba [2012] as a comparison. Subfigure 4(a) shows the plot for the entire $y$-range while subfigure 4(b) shows a larger view of the upper and lower $y$-range.

grid resolutions and order of continuity. The linear conjugate gradient iteration (see Subsection 5.1) accounted for most of the computation time (typically about 95 %). For higher Reynolds numbers, a higher grid resolution was required to achieve convergence. For Reynolds number, $Re > 5000$, computations were only carried out with $\Omega = 5$ since the required grid resolution for lower values of $\Omega$ made computations on the current system too time consuming. For the highest Reynolds number, $Re = 40000$, the computation was run for approximately 72 hours with $L = 135$ and $\Omega = 5$.

The error, $\sqrt{P}$, plotted with empty squares in Figure 8, shows a rapid initial convergence followed by a much slower rate of convergence. It is clear, however, when comparing with the reference figures from Erturk et al. [2005], plotted with solid squares in Figure 8, that the computed solutions still undergo changes during the final iterations. This may be explained by the fact that, while the quantity $\sqrt{P}$ only measures how well the solution conforms to the given differential equations (independently) at each point in the computational domain, the quantity $RMS_{ref}$ depends on the value of the solution at specific points which may be affected by the accumulation of small errors elsewhere in the computational domain. Additionally, some areas of the computational domain (e.g. near the lower corners where the velocity is discontinuous) may suffer from large errors compared to the rest of the grid, dominating the value of the quantity $\sqrt{P}$ in the later stages of the iteration. The residual of the solutions computed by the references [Erturk et al., 2005, Wahba, 2012] converge to a smaller factor than the error of the current solutions. However, the error *between* grid points is not taken into consideration by Erturk et al. [2005], Wahba [2012], whereas in the current work the error is computed over a large number of sub-grid sample points.

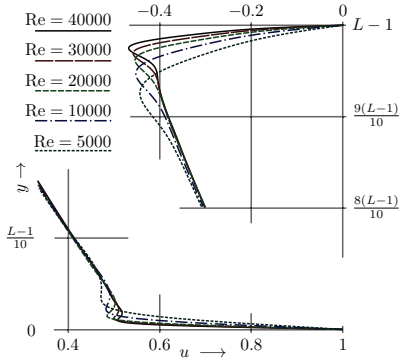



Figure 5: This figure shows the computed $x$-component of the velocity on a vertical line through the geometric center of the grid for Reynolds number, $\mathrm{Re} \in \{5000, 10000, 20000, 30000, 40000\}$. Only the upper ($y$ near $L-1$) and lower ($y$ near 0) part of the computational domain is plotted. The different lines shows current results, $\mathbf{b}(x', y')\mathbf{B}^{-1}\mathbf{u}[k, l]$ for $x = (L-1)/2$, $0 \leq y \leq L-1$, with $L \in \{40, 60, 100, 120, 135\}$ and $\Omega = 5$ where $x' = x - \mathbf{x}[k, l]$ and $y' = y - \mathbf{y}[k, l]$.

It is clear that both an increase of the grid resolution, $L$, and the order of continuity, $\Omega - 1$, increases the size of the linear system, $\mathbf{\Pi}$ (see Eq.(29)), and thus the required computation time and the required amount of memory. It should also be noted, however, that increasing the order of continuity (i.e. increasing $\Omega$) also increases the number of nonzero sub/super-diagonals of the linear system, $\mathbf{\Pi}$, which also increases memory requirements and computation time. Despite this disadvantage for higher orders of continuity, it was found that using higher values of $\Omega$ was more efficient at computing solutions for high Reynolds numbers ($\mathrm{Re} \gtrapprox 2500$), yielding solutions of acceptable accuracy for lower values of the grid resolution, $L$. The value of $\Omega$ is limited by the floating point errors, as shown in Subsection 2.6. For this reason, a maximum value of 5 was chosen (i.e. $\Omega \leq 5$ in all computations), corresponding to a spatial 4'th order of continuity ($C^4$) and a polynomial accuracy of order 9.

## 7 Conclusion and Outlook

An increase in spatial order (polynomial degree) of the grid ($p$-refinement) has advantages compared to increasing the grid resolution ($h$-refinement) in some cases when using the current method, as shown by the high Re solutions for the lid-driven cavity. These solutions, computed on an ordinary desktop computer, are among the highest Reynolds numbers at which steady state solutions for the lid-driven cavity have been published, even though obvious optimizations (e.g. mesh grading or parallel computation) were not used.

Unlike pseudo-time finite differencing approaches, the current method for arriving at a steady state solution does not yield periodic solutions as artifacts. Instability may appears if the grid resolution is insufficient, but it is chaotic, and does not resemble a periodic flow configuration.

It is clear from physical evidence that, for the high Reynolds numbers ($\mathrm{Re} \gtrapprox 5000$), the presented steady state solutions do not correspond to a physical three dimensional flow. It is, however, interesting to note that small perturbations, which are thought to initiate turbulence in a real flow, may be mimicked by numerical inaccuracies and potentially initiate turbulence or periodic behavior in simulations. The large differences in reported Reynolds number at which steady state solutions have been obtained for the lid-driven cavity in two dimensions may be

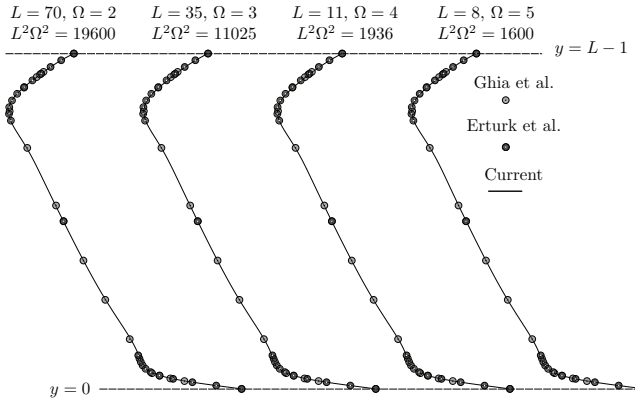



Figure 6: This figure shows a comparison of the computed $x$-component of the velocity on a vertical line through the geometric center of the grid for $(L, \Omega) \in \{(70, 2), (35, 3), (11, 4), (8, 5)\}$ with Reynolds number, $\mathrm{Re} = 1000$. At the top ($y = L - 1$) of the figure the $x$-component of the velocity is zero and at the bottom ($y = 0$) it is one (positive $x$-direction). The dotted circles show results presented by Erturk et al. [2005] and Ghia et al. [1982].

explained by the different nature and magnitude of these inaccuracies. If a periodic behavior, observed when solving the two dimensional system, was exclusively due to the mathematical qualities of the of the system (i.e. due to Poincaré–Andronov–Hopf bifurcation), it is reasonable to assume that this behavior would have occurred at similar Reynolds numbers even though different numerical schemes were used.

The grids with the highest order of continuity, $\Omega = 5$ (equivalent to a polynomial degree of 9, see Subsection 2.2-2.4), were the most efficient for computing steady state solutions for high Reynolds number flows, but the numerical accuracy imposed limitations on further increase of the order of continuity. An improvement, for example by using increased floating point precision or by finding basis functions with better numerical properties, is clearly possible.

It is clear from the mathematical framework (see Section 2) that the current method can be generalized to higher dimensions. Further, linear terms (e.g. time derivative, for unsteady flows) may also be added to the governing equations in matrix form (see Subsection 3.3) without fundamentally changing the properties of the method. With the current method, and other finite-element based methods, one obtains coupled sets of equations depending on information in a grid. The computational cost required to solve these systems tend to grow exponentially with the number of grid points. However, the computational cost of the numerical integration, which defines the equation set for the current method (see Subsection 4.1) grows linearly with the number of grid points. This is an advantage because, instead of adapting the grid to complex geometry or to different fluid phases, with the current method it is possible to select different governing equations independently at different sample points. One can also increase the density of sample points in some areas if necessary (assuming appropriate weighting is applied). Inter-

action with objects smaller than the grid scale can thus be incorporated. An interface between immiscible fluid phases can be incorporated in the same way. The latter will be demonstrated with three-dimensional unsteady flow in a forthcoming paper.

## Acknowledgments

This work was supported by BKK Production and The Research Council of Norway under The Industrial Ph.D Scheme.

Alex Hoffmann[2], Jan Vaagen[3], Laszlo Csernai[4] and Arne Småbrekke[5] are acknowledged for productive discussions and valuable suggestions.

## Bibliography

E. Barragy and G. F. Carey. Stream function-vorticity driven cavity solutions using p finite elements. *Computers and Fluids*, 26, 1996.

E. Erturk, T. C. Corke, and Gökçöl. Numercal solutions of 2-d steady incompressible driven cavity flow at high reynolds numbers. *International Journal for Numerical Methods in Fluids*, 48, 2005.

U. Ghia, K. N. Ghia, and T. C. Shin. High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method. *Journal of Computational Physics*, 48, 1982.

Anton Howard. *Elementary Linear Algebra*. Wiley, New York, 2000.

H. Nishida and N. Satofuka. Higher-order solutions of square driven cavity flow using a variable-order multigrid method. *International Journal for Numerical Methods in Engineering*, 34, 1992.

R. Schreiber and H. B. Keller. Driven cavity flows by efficient numerical techniques. *Journal of Computational Physics*, 49, 1982.

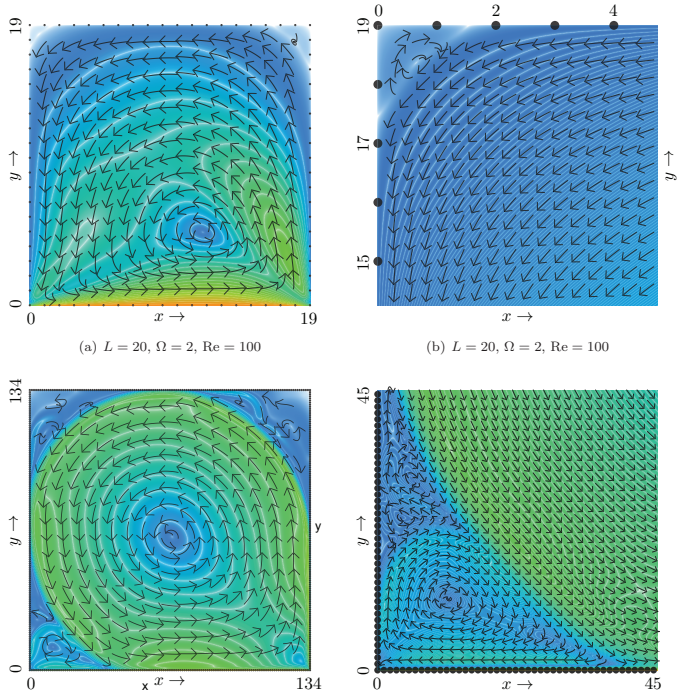
Jonathan R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.

H. Takewaki, A. Nishiguri, and T. Yabe. Cubic interpolated pseudo-particle method (CIP) for solving hyperbolic-type equations. *Journal of Computational Physics*, 61, 1984.

Lloyd N. Trefethen and David Bau, III. *Numerical Linear Algebra*. Siam, Philadelphia, 1997.

E. M. Wahba. Steady flow simulations inside a driven cavity up to reynolds number 35,000. *Computers and Fluids*, 66, 2012.

[2]Professor, University of Bergen, MAE
[3]Professor, University of Bergen, MAE
[4]Professor, University of Bergen, MAE
[5]Department Manager, BKK Production



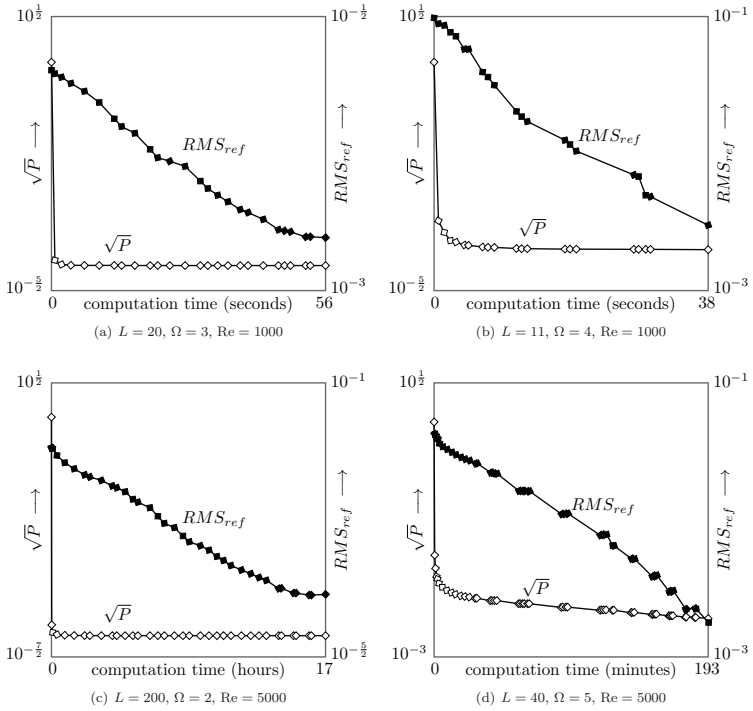

(a) $L = 20$, $\Omega = 2$, $\mathrm{Re} = 100$

(b) $L = 20$, $\Omega = 2$, $\mathrm{Re} = 100$

(c) $L = 135$, $\Omega = 5$, $\mathrm{Re} = 40000$

(d) $L = 135$, $\Omega = 5$, $\mathrm{Re} = 40000$

Figure 7: Subfigures 7(a) and 7(b) show the computed flow configuration for Reynolds number, $\mathrm{Re} = 100$, with $L = 20$ and $\Omega = 2$. Subfigure 7(a) shows the entire computational domain, while Subfigure 7(b) shows details at the upper left corner (commonly referred to as the tertiary vortex). Subfigures 7(c) and 7(d) show the computed flow configuration for Reynolds number, $\mathrm{Re} = 40000$, with $L = 135$ and $\Omega = 5$. Subfigure 7(c) shows the entire computational domain, while Subfigure 7(d) shows details at the lower left corner (commonly referred to as the quaternary vortex, however in this case it is split up into multiple sub-vortices). The color indicates the magnitude of the velocity, $\|(u, v)\|_2$, where orange is for $\|(u, v)\|_2 = 1$, green is for $\|(u, v)\|_2 = 1/2$ and bright blue is for $\|(u, v)\|_2 = 0$ (and interpolated between these colors for the intermediate values). Contour lines of the velocity magnitude are drawn in white with a contour interval of $1/20$ in Subfigure 7(a), $1/1000$ in Subfigure 7(b), $1/20$ in Subfigure 7(c) and $1/200$ in Subfigure 7(d). The arrows are of constant length in each subfigure and are drawn in a Lagrangian coordinate system, defined by the two orthogonal unit vectors $\hat{x}_l = (u, v)/\|(u, v)\|_2$ and $\hat{y}_l = (v, -u)/\|(u, v)\|_2$, pointing in the positive direction along the unit vector, $\hat{x}_l$. The grid resolution is indicated by dots along the edge of the grid.




(a) $L = 20$, $\Omega = 3$, Re = 1000

(b) $L = 11$, $\Omega = 4$, Re = 1000

(c) $L = 200$, $\Omega = 2$, Re = 5000

(d) $L = 40$, $\Omega = 5$, Re = 5000

Figure 8: These figures show the error (empty squares) and the deviance (solid squares) from a reference solution [Erturk et al., 2005] on a logarithmic scale for four separate computations, run through several iterations until comparable accuracy was reached. The error, $\sqrt{P}$, (plotted with empty squares) is the root–mean–square grid-cell error (numerically integrated over the entire grid). The quantity, $P$, is given in Eq.(37) as a function of the parameters $\theta_u$,$\theta_v$ and $\theta_p$ which are determined as explained in Subsection 5.2. The quantity, $RMS_{ref}$, (plotted with solid squares) is the root–mean–square deviance of the computed solution as compared with the figures given by Erturk et al. [2005] for the $x$-component of the velocity along a vertical line through the geometric center of the cavity. Subfigures 8(a) and 8(b) shows these quantities over 25 iterations using $L = 20, \Omega = 3$ and $L = 11, \Omega = 4$, respectively, and with Reynolds number, Re = 1000, where an approximate solution for Reynolds number, Re = 400, was used to initialize the flow components. Subfigures 8(c) and 8(d) shows these quantities over 30 and 50 iterations using $L = 200, \Omega = 2$ and $L = 40, \Omega = 5$, respectively, with Reynolds number, Re = 5000, where an approximate solution for Reynolds number, Re = 2500, was used to initialize the flow components.

## 6.2 Bubble Simulation

The paper entitled *A High order Approach to Solving Nonlinear Differential Equations Applied to Direct Numerical Simulation of Two-Phase Unsteady Flow*, was presented at the Multiphase 2015 conference [10],[2]. This paper shows the method employed to simulate bubble rising under gravitational influence (implicit time marching). Figure 1 in [10] illustrates the problem to be solved. The simulations are performed using relatively low cost hardware (CPU: Intel Xeon E5-1650v2, GPU: AMD FirePro W7000).

# A High Order Approach to Solving Nonlinear Differential Equations Applied to Direct Numerical Simulation of Two-Phase Unsteady Flow

Jesper Tveit
*University of Bergen*
*BKK Production*
*Bergen, Norway*

**Abstract**

A method for solving nonlinear differential equations, which facilitates the computation of solutions of a high polynomial degree on a grid, is tested for use in direct numerical simulation (DNS) of two-phase unsteady flow.

The method uses a grid discretization to approximate continuously distributed variables, represented by functions of time and space, in a given set of differential equations. The grid contains information about both the values and the values of the derivatives of the unknown functions at the grid points in the computational domain. With this method the derivatives are thus explicitly defined at each grid point rather than, as in conventional numerical schemes, implicitly given by the function values at the surrounding grid points. Using piecewise polynomial interpolation, functions can be represented with an arbitrary order of continuity over the entire computational domain.

The high polynomial order used in this method allows for simulation of flow features smaller than the interval separating each grid point. This reduces the required number of grid points and the need to adapt the grid to complex boundary geometry or to the interphase between different fluid phases. This simplifies grid generation and reduces the computational cost.
*Keywords: discretization, high order, direct numerical simulation, two-phase unsteady flow.*

## 1 Introduction

The mathematical framework and algorithms employed are described in detail in ref. [1], together with computed results for the lid-driven cavity test case. This method has been developed for a finite element, residual minimizing type of approach.

In the current work we apply the method to three dimensional unsteady two phase flow. Simulations of a bubble in a cubical domain are carried out as a proof of concept.

The current results are obtained after some improvements have been made. We will therefore make a short review of these, as well as the changes that have been made in order to perform two-phase flow simulations.

## 2 Adaption to Two–Phase Unsteady Flow

### 2.1 Basis Functions and Conditioning

| | | |
|---|---|---|
| $C^1$ | $b_0^4(x) =$ | $2x^3 - 3x^2 + 1$ |
| | $b_1^4(x) =$ | $6x^3 - 12x^2 + 6x$ |
| | $b_2^4(x) =$ | $-2x^3 + 3x^2$ |
| | $b_3^4(x) =$ | $-6x^3 + 6x^2$ |
| $C^2$ | $b_0^6(x) =$ | $-6x^5 + 15x^4 - 10x^3 + 1$ |
| | $b_1^6(x) =$ | $-15x^5 + 40x^4 - 30x^3 + 5x$ |
| | $b_2^6(x) =$ | $-30x^5 + 90x^4 - 90x^3 + 30x^2$ |
| | $b_3^6(x) =$ | $6x^5 - 15x^4 + 10x^3$ |
| | $b_4^6(x) =$ | $-15x^5 + 35x^4 - 20x^3$ |
| | $b_5^6(x) =$ | $30x^5 - 60x^4 + 30x^3$ |
| $C^3$ | $b_0^8(x) =$ | $20x^7 - 70x^6 + 84x^5 - 35x^4 + 1$ |
| | $b_1^8(x) =$ | $(140x^7)/3 - 168x^6 + 210x^5 - (280x^4)/3 + (14x)/3$ |
| | $b_2^8(x) =$ | $84x^7 - 315x^6 + 420x^5 - 210x^4 + 21x^2$ |
| | $b_3^8(x) =$ | $140x^7 - 560x^6 + 840x^5 - 560x^4 + 140x^3$ |
| | $b_4^8(x) =$ | $-20x^7 + 70x^6 - 84x^5 + 35x^4$ |
| | $b_5^8(x) =$ | $(140x^7)/3 - (476x^6)/3 + 182x^5 - 70x^4$ |
| | $b_6^8(x) =$ | $-84x^7 + 273x^6 - 294x^5 + 105x^4$ |
| | $b_7^8(x) =$ | $140x^7 - 420x^6 + 420x^5 - 140x^4$ |

Table 1: Polynomial basis functions, $b_\gamma^\Gamma$, for different orders of continuity.

As shown in [1] the choice of interpolating basis functions is important with respect to the numerical conditioning of the resulting system of equations. Bernstein polynomials were found to have acceptable properties. However, in the current work we use a different set of polynomials (Table 1). The polynomials given in Table 1 are chosen especially such that they produce a well conditioned system. These polynomials are constructed such that at the end points (where the interpolating variable, $x$, is either zero or one) they satisfy the conditions given in Equations (1a - 1b). Note that, for each order of continuity, there are an even number of basis functions. Of each set, the lower half ($\lambda \in \{0 \dots \Lambda/2 - 1\}$) corresponds to the point at $x = 0$ while the rest ($\lambda \in \{\Lambda/2 \dots \Lambda - 1\}$) corresponds to the point at $x = 1$.

$$\frac{\partial^k}{\partial x^k} b_\lambda^\Lambda(x)|_{x=0} = a_k \delta_{k\lambda} \tag{1a}$$

$$\frac{\partial^k}{\partial x^k} b_\lambda^\Lambda(x)|_{x=1} = a_k \delta_{(k+\Lambda/2)\lambda} \tag{1b}$$

$$\left| \int_{x=0}^{1} b_\lambda^\Lambda(x) \mathrm{d}x \right| = 1 \tag{1c}$$

$$k \in \{0 \dots \Lambda/2 - 1\} \tag{1d}$$

Here $a_k$ is a positive normalization constant chosen such that sub-equation (1c) is satisfied. Thus, a polynomial approximation of a function $f(x)$, based on the function values at the two end points (i.e. grid points) is given directly by the values of $f(x)$ and its derivatives at the end points by

$$f(x) = \sum_{\lambda=0}^{\Lambda/2-1} b_\lambda^\Lambda(x) a_\lambda \left. \frac{\partial^\lambda f(x)}{\partial x^\lambda} \right|_{x=0} + \sum_{\lambda=0}^{\Lambda/2-1} b_{\lambda+\Lambda/2}^\Lambda(x) a_\lambda \left. \frac{\partial^\lambda f(x)}{\partial x^\lambda} \right|_{x=1} + \mathcal{O}\left(x^\Lambda\right)$$

At each grid point then, the values $\{a_0 f, a_1 f', a_2 f'', \dots\} \equiv \{\hat{f}, \hat{f}', \hat{f}'', \dots\}$ up to a desired order of continuity are stored (here, prime denotes derivative and $\hat{}$ indicates a normalized quantity). A matrix inversion is no longer needed to produce the piecewise polynomial approximation for each cell. As a consequence the floating point accuracy is no longer a limiting factor (see [1] section 2). Since the higher derivatives tend to take on values of greatly varying magnitude even with small variations of the flow configuration, computing the scaled values directly rather than derivatives, improves the conditioning of the resulting equation system.

### 2.2 Basis Functions in Three Dimensions

The basis functions are generalized to higher dimensions by taking the product, $B^{\Lambda}_{k,l,m}(x,y,z) = b^{\Lambda}_{k}(x)b^{\Lambda}_{l}(y)b^{\Lambda}_{m}(z)$. In the current work we use this discretization with the same order of continuity in the three spatial dimensions, and implicit marching in the temporal direction.[1]

### 2.3 Unsteady Flow

The continuity and momentum equations depend on the fluid phase in a way which is not easily linearized. As a consequence we do not linearize all the governing equations into a single system. Instead the velocity, pressure and phase are mapped into separate linearized global equation systems, where the time derivatives of the next time-step are the unknowns (including time derivatives of the spatial derivatives, up to the given order of continuity). This is an implicit time–marching scheme (see Table 2) where the solution for each time step is found by repeatedly solving for velocity, pressure and phase, taking the previous solution as constant in each step.[2]

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| | 0 | 1 |

Table 2: Buther tableau for the implicit marching scheme.

### 2.4 Interphase Tracking

Numerical methods solving two-phase unsteady flow typically rely on either adjusting the discretization geometry of the computational domain to fit the interphase between the different fluid phases, or by using particles moving with the flow, like the Particle in Cell (PIC) method [2, 3] and its successor, the Smoothed Particle Hydrodynamics (SPH) [4] method.

The current method uses a constant grid combined with a sub-grid integration scheme to achieve sub-grid accuracy. An iso-surface of a scalar function, $f$, is used to track the interphase between different fluid phases. This approach is also employed by for example the Volume of Fluid (VOF) method [5].

[1] It is also possible to employ this discretization in the temporal dimension, and to use different order of continuity in different directions.

[2] The nonlinear optimization used in [1] was not implemented, as current procedure alone produced acceptable convergence rates.

The scalar function, $f$, is discretized in the same manner as the velocity and density. Its time evolution is determined by convection along with the fluid flow (see appendix for details). At each sample point, the distance, $r$, from the interphase is approximated by $r \approx f/\sqrt{\nabla f \cdot \nabla f}$. A smoothing function, $s(r)$, which is nonzero for small values of $r$ determines the surface effects (see appendix, Equation (7)). The smoothing function $s(r)$ is a polynomial with continuous first and second derivatives. The interphase is thus approximated by a layer near $f = 0$ of finite thickness. The necessary thickness depends on the density of the sample points. In the current work the interphase thickness was approximately $7.79 \times 10^{-3}$ (relative to the size of the computational domain).

### 2.5 Preconditioning

Solving the global equation system for the velocity is a potential bottleneck as the resolution increases (the cost of the direct solution grows as $N^3(k+1)^3$, with $N$ and $C^k$ being the number of grid points and order of continuity). However, by using the Cholesky factorization of the initial equation system as a preconditioner, the system may be solved very efficiently in the subsequent iterations using the conjugate gradient (CG) iteration (typically around five CG iterations).

## 3 Governing Equations

The differential form of the Navier–Stokes equations (dimensionless, scaled with appropriate physical quantities) are solved. Table 3 shows the numerical values of the different parameters determining the fluid properties. The grid dependent parameters are $\tau = \frac{T-1}{L-1}$and $\eta = L - 1$, where $L$ and $T$ are the spatial and temporal grid resolutions, respectively. The reader may examine the appendix for a detailed formulation of the governing equations.

## 4 Simulation

The simulation is of a fictitious fluid with high viscosity. The aim is to demonstrate the method's applicability to two-phase unsteady flow together with boundary details on a sub-grid scale. The reader may refer to [1] for a verification and comparison of the results of this method with conventional methods. Figure 1 shows the set up. The grid used in this case uses $C^2$ continuity and thus a spatial (polynomial) order of five ($\mathcal{O}(x^6)$ terms are discarded). With seven-cubed grid points we have $L = 7 \Rightarrow \eta = L - 1 = 6$. Further we let one time unit correspond to sixty steps, thus $\tau = (T-1)/(L-1) = 10$.

| | phase I | phase II |
|---|---|---|
| $\alpha$ | $10^{11}$ | $10^{11}$ |
| $\beta$ | $1/10$ | $10$ |
| Re | $100$ | $100$ |
| $\vec{g}$ | $(0, 0, -1)$ | $(0, 0, -1)$ |
| $\sigma$ | $10^{-3}$ | $10^{-3}$ |

Table 3: The numerical values of the parameters of the governing equations. The two fluids have equal properties except for the density, which is lower (by a factor 100) for the bubble (phase II), resulting in a higher value of $\beta$.

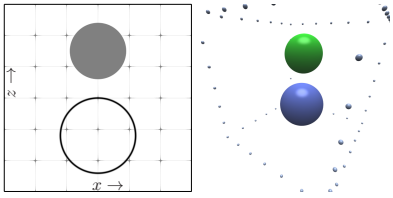



Figure 1: This figure shows the initial conditions of the system. The computational domain is a cube. A solid, spherical object with radius 0.15 is fixed at the center of the $x$-$y$ plane at a (center) height 0.75. The lightest phase is initially a sphere with radius 0.2 positioned at the center of the $x$-$y$ plane at a height 0.25 (scales relative to the size of the computational domain). A cross section of the computational domain of the grid is shown on the left hand side. On the right hand side we have a perspective rendering showing the solid and the interphase at its initial position. The initial flow velocity is zero and the initial pressure is constant. No-slip Dirichlet boundary conditions are enforced throughout the simulation. The grid resolution is indicated by dots (in this case $7^3 = 343$ grid points)

#### 4.1 System Configuration

Figure 1 describes the computational domain and the initial conditions. The initial distribution of the light fluid phase is axially symmetric and the computational domain is cubical.

### 4.2 Time Evolution

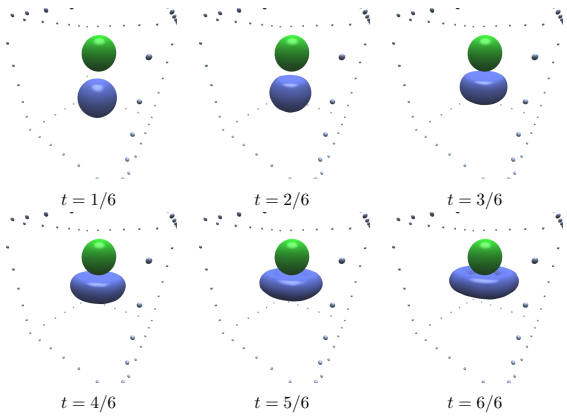



Figure 2: This figure shows a perspective rendering (obtained with ray casting) of the bubble interphase at different times. The surrounding dots are grid-points at the edge of the computational domain.

Figures 2 and 3 shows snapshots of the simulation at different times. Table 4 shows numerical values of theoretically verifiable quantities at different time-steps. As the bubble shape becomes stretched out and thinner compared to the grid resolution, an increased inaccuracy is observed.

## 5 Computational Cost

The computational cost can be divided into two parts, (i) the numeric integration over all sample points which form the linearized system of equations, and (ii) the cost of solving these equations. In this simulation the numeric integration required most time (on average 262 seconds per iteration). Less than ten percent of the time was spent on solving the systems (on average 27 seconds per iteration) due to the rapid convergence of the CG iteration. It should be noted that the cost of the numeric integration grows linearly with the number of grid-points. It is also easily parallelizable.

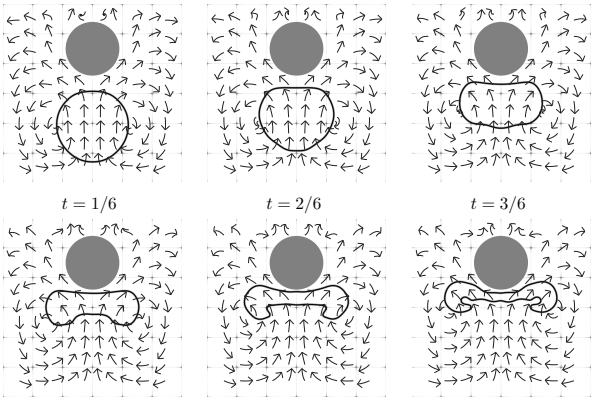



Figure 3: This figure shows the $x - z$ cross section ($y$ centered) at different times. The arrows are of constant length in each figure and are drawn in a Lagrangian coordinate system, projected into the $x - z$ plane.

#### 5.1 convergence

Figure 4 shows the convergence history for 12 different time-steps.

## 6 Conclusion and Outlook

Two main problems have been tested in these simulations. i) Two phase flow and ii) sub grid geometry. Both of these were studied simultaneously without fundamentally changing the method to fit either issue. Compared with the two dimensional computations presented in [1] we see that the main computational effort is spent on numeric integration, while solving the linear systems is comparatively cheap due to efficient use of preconditioners. Since the algorithms used for numeric integration are easily parallelizable and have a $\mathcal{O}(N)$ cost, the benefit of increasing the hardware capabilities should be high compared to other methods.

| step | mass | $x$–momentum | $y$–momentum |
|---|---|---|---|
| 0 | 9.63863 | 0 | 0 |
| 5 | 9.63871 | $-1.69635 \times 10^{-5}$ | $8.60974 \times 10^{-6}$ |
| 10 | 9.63813 | $1.31676 \times 10^{-5}$ | $1.65046 \times 10^{-5}$ |
| 15 | 9.63808 | $9.96706 \times 10^{-6}$ | $5.95129 \times 10^{-5}$ |
| 20 | 9.63750 | $9.644 \times 10^{-5}$ | $9.89 \times 10^{-5}$ |
| 25 | 9.63620 | 0.000254756 | 0.000242913 |
| 30 | 9.63827 | 0.00035846 | 0.000307487 |
| 35 | 9.62337 | 0.000497964 | 0.000484558 |
| 40 | 9.63011 | 0.000585154 | 0.000473407 |
| 45 | 9.61777 | 0.000430384 | 0.000362024 |
| 50 | 9.61049 | 0.000440684 | 0.000301131 |
| 55 | 9.60459 | 0.000514161 | 0.000194891 |
| 60 | 9.60225 | 0.000688966 | 0.000309241 |

Table 4: This table shows computed numerical values of the physically constant quantities: total mass and total horizontal momentum in $x$– and $y$–directions (obtained with Monte Carlo integration). The exact value of the mass is $10\left(1-\frac{4}{3}\pi 0.2^3\right)+\frac{4}{3}\pi 0.2^3/10 \approx 9.668$ and the exact value of the horizontal momentum is zero.

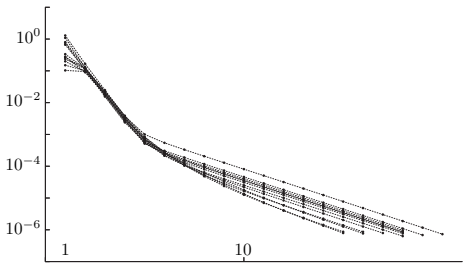



Figure 4: The convergence history of twelve different time-steps of the simulation is shown. The plotted value (dots) is the root of the sum of squares of the step length of all flow components in the grid for each iteration. Each iteration was terminated once this quantity dropped below $10^{-6}$.

# Appendix

### The Dimensionless Navier-Stokes Equations for Two Phases

The characteristic length and velocity scales, which map to unity in the computational domain, are $x_0$ and $v_0$ ($t_0 = v_0/x_0$) and define the dimensionless (non-primed) quantities:

$$\vec{v}' = v_0\vec{v} \tag{2a}$$
$$\rho' = \rho_0\rho \tag{2b}$$
$$p'_{amb} = \rho_0 v_0^2 p_{amb} \tag{2c}$$
$$p' = \rho_0 v_0^2 (p + p_{amb}) \tag{2d}$$
$$\vec{g}' = \frac{v_0^2}{x_0}\vec{g} \tag{2e}$$
$$\frac{\partial}{\partial t'} = \frac{v_0}{x_0}\frac{\partial}{\partial t} \tag{2f}$$
$$\nabla' = \frac{1}{x_0}\nabla \tag{2g}$$
$$\mathbb{T}' = \frac{\mu v_0}{x_0}\mathbb{T} = \frac{\mu v_0}{x_0}\left(\nabla\vec{v} + (\nabla\vec{v})^T + \frac{\lambda}{\mu}(\nabla\cdot\vec{v})\,\mathbb{I}\right) \tag{2h}$$
$$\sigma' = \sigma_0\sigma = \rho_0 x_0 v_0^2\sigma \tag{2i}$$
$$f' = f \tag{2j}$$

Here, $\rho$ is density, $p$ is pressure, $\vec{v}$ is velocity, $\mathbb{T}$ is the viscous stress tensor. The sign of the scalar function, $f$, defines the fluid phase. The equation of state is approximated by a linear relation between density and pressure. The superscript $^T$, is the transpose and $\mathbb{I}$ is the identity tensor. The phase dependent properties are $\mu$ and $\lambda$ (first and second viscosity coefficients), $\rho_{amb}$ (ambient density) and $k = K_{amb}/\rho_{amb}$ where $K_{amb}$ is the bulk modulus at ambient conditions.

The dimensionless phase dependent properties are determined by the dimensionless parameters Re (viscosity), $\alpha$ (compressibility) and $\beta$ (density):

$$\mathrm{Re} = \frac{x_0 v_0 \rho_{amb}}{\mu}\,,\ \alpha = \frac{k\rho_{amb}}{v_0^2\rho_0} = \frac{K_{amb}}{v_0^2\rho_0}\,,\ \beta = \frac{\rho_0}{\rho_{amb}} \tag{3}$$

We let $\lambda/\mu = -2/3$. The viscous stress term, written as an operator $\mathbb{S}$, acting on the velocity reads

$$\nabla\cdot\mathbb{T} = \mathbb{S}\cdot\vec{v} = \begin{bmatrix} \nabla^2 + \frac{1}{3}\frac{\partial^2}{\partial x^2} & \frac{2}{3}\frac{\partial^2}{\partial x\partial y} & \frac{2}{3}\frac{\partial^2}{\partial x\partial z} \\ \frac{2}{3}\frac{\partial^2}{\partial x\partial y} & \nabla^2 + \frac{1}{3}\frac{\partial^2}{3\partial y^2} & \frac{2}{3}\frac{\partial^2}{\partial y\partial z} \\ \frac{2}{3}\frac{\partial^2}{\partial x\partial z} & \frac{2}{3}\frac{\partial^2}{\partial y\partial z} & \nabla^2 + \frac{1}{3}\frac{\partial^2}{3\partial z^2} \end{bmatrix}\cdot\vec{v} \tag{4}$$

The dimensionless formulation is then:

$$0 = \frac{1}{\alpha}\left[\frac{\partial p}{\partial t} + p\nabla\cdot\vec{v} + \vec{v}\cdot\nabla p\right] + \nabla\cdot\vec{v} \tag{5a}$$

$$0 = \frac{1}{\alpha}\left[p\left(\frac{\partial\vec{v}}{\partial t} + v\cdot\nabla\vec{v} - \vec{g}\right)\right] + \left(\frac{\partial\vec{v}}{\partial t} + v\cdot\nabla\vec{v} - \vec{g}\right) + \beta\nabla p - \frac{1}{\mathrm{Re}}\mathbb{S}\cdot\vec{v} \tag{5b}$$

$$0 = \frac{\partial f}{\partial t} + \vec{v}\cdot\nabla f \tag{5c}$$

Since the fluids are assumed weakly compressible, $\alpha$ is large and the bracketed terms make only a small contribution. If the flow is incompressible ($\alpha\to\infty \Rightarrow \nabla\cdot\vec{v} = 0$) it can be shown that $\mathbb{S}$ reduces to $\mathbb{I}\nabla^2$. In the single-phase case one would choose $\rho_0 = \rho_{amb}$ yielding $\beta = 1$. In the two-phase case $\rho_0$ may be set to $\rho_{amb}$ of one of the fluids, or something in between.

### Adapting Spatial and Temporal Scales to Grid Dimensions

The spatial and temporal scales in Eq.(5) are defined so that their size is equal to the interval $[0,1]^4$ in the computational domain. If the grid is uniform, floating point round off errors might be reduced by defining the characteristic length scales so that they instead correspond to the interval $[0, L-1]^3\times[0, T-1]$ in the computational domain. With $L$ being the spatial grid resolution and $T$ the temporal grid resolution (i.e. the grid has $L\times L\times L\times T$ grid-points) the spacing between grid points becomes equal

to one. The corresponding set of equations are:

$$0 = \frac{1}{\alpha}\left[\tau\frac{\partial p}{\partial t} + p\nabla\cdot\vec{v} + \vec{v}\cdot\nabla p\right] + \nabla\cdot\vec{v} \tag{6a}$$

$$0 = \frac{1}{\alpha}\left[p\left(\tau\frac{\partial \vec{v}}{\partial t} + \vec{v}\cdot\nabla\vec{v} - \frac{\vec{g}}{\eta}\right)\right] + \left(\tau\frac{\partial \vec{v}}{\partial t} + \vec{v}\cdot\nabla\vec{v} - \frac{\vec{g}}{\eta}\right) + \beta\nabla p - \frac{\eta}{\mathrm{Re}}\mathbb{S}\cdot\vec{v} \tag{6b}$$

$$0 = \tau\frac{\partial f}{\partial t} + \vec{v}\cdot\nabla f \tag{6c}$$

where $\tau = \frac{T-1}{L-1}$ and $\eta = L - 1$.

### Surface Tension

Surface tension gives rise to a pressure discontinuity in the equilibrium case. Since the discontinuity is difficult to express accurately with continuous basis functions we add it as an additional force (source term) in the momentum equation instead of incorporating it in the pressure directly. The interphase is approximated by a small interval around $f = 0$ with a smoothing function, $s$, depending on the the distance, $r$, from the interphase. The momentum equation, with surface tension included reads

$$0 = \frac{1}{\alpha}\left[p\left(\tau\frac{\partial \vec{v}}{\partial t} + \vec{v}\cdot\nabla\vec{v} - \frac{\vec{g}}{\eta}\right)\right] + \left(\tau\frac{\partial \vec{v}}{\partial t} + \vec{v}\cdot\nabla\vec{v} - \frac{\vec{g}}{\eta}\right) + \beta\left(\nabla p - \eta\sigma\frac{\nabla f}{|\nabla f|}\frac{\partial s}{\partial r}\nabla\cdot\left(\frac{\nabla f}{|\nabla f|}\right)\right) - \frac{\eta}{\mathrm{Re}}\mathbb{S}\cdot\vec{v} \tag{7}$$

### References


[1] Tveit, J., A Numerical Approach to Solving Nonlinear Differential Equations on a Grid with Potential Applicability to Computational Fluid Dynamics. *arXiv*, 2014.

[2] Harlow, F.H. & Welch, J.E., Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, **8**, p. 2182, 1965.

[3] Harlow, F.H. & Welch, J.E., Numerical study of large amplitude free-surface motions. *Physics of Fluids*, **9**, p. 842, 1966.

[4] Hu, X.Y. & Adams, N.A., An incompressible multi-phase sph method. *Journal of Computational Physics*, **227**, pp. 264–278, 2007.

[5] Hirt, C.W. & Nichols, B.D., Volume of fluid (vof) method for the dynamics of free boundaries. *Journal of Computational Physics*, **39**, pp. 201–225, 1981.

# Chapter 7

# Conclusion and Outlook

## 7.1 Applicability

Through this thesis we have developed a method for solving nonlinear differential equations, based on a novel grid-discretization. The approach is quantitatively verified by the computed solutions to the lid-driven cavity, where the results agree with previously computed results from different independent sources. The lid-driven cavity results also show favorable scaling properties when increasing the order of continuity as indicated in the introduction. On this foundation the approach is further developed for three dimensional two-phase flow. Simulations of three-dimensional two phase flow demonstrates, in a qualitative way, the potential utilization in the study of two phase flow patters in mechanical systems such as a Pelton-turbine. From these we also learn where focus must lie in the further development and optimization of the method.

## 7.2 Further Work

Based on the experience gained in developing this method, there are three main principles which stand out as interesting in the further development. i) A pure nonlinear solution algorithm which does not require the formation of a linearized equation system. The nonlinear conjugate gradient iteration is an algorithm with potential, as it only requires the computation and storage of gradients and search directions. Additionally, the linear conjugate gradient iteration was able to solve the linearized velocity systems efficiently. ii) A hybrid particle-sample based integration scheme. In the Pelton-bucket simulation the level set method which was used to distinguish the different phases produced artifacts as the flow features became too fine compared to the grid resolution. Having the sample points act as particles following the fluid flow would counter this problem, but at the cost of not having a functional definition of the interface. iii) Optimization of the numeric integration for parallel computing. The numeric integration was the most computational expensive step in the three-dimensional simulations. As the resolution increases, the cost of the numeric integration should become less significant if implemented effectively since it is a parallelizable computation which scales linearly with the number of grid-cells.

## 7.3 Options

The methodology is based on the interpolation in a grid-cell, which was defined to cover a region enclosed by the surrounding grid-points. This is convenient since we may use the same interpolation method in all grid-cells in the same manner regardless of their location relative to the edge of the grid. However it is fully possible to implement a central-difference analogue – interpolation over three grid points (see Figure 7.1). This yields a higher polynomial order of accuracy (given the same grid information) and thus potentially more accurate solutions. However the algebraic systems will be more coupled (less sparse linearized systems) and the grid-cells near the edge of the grid must be treated as special cases, making the implementation more complicated.

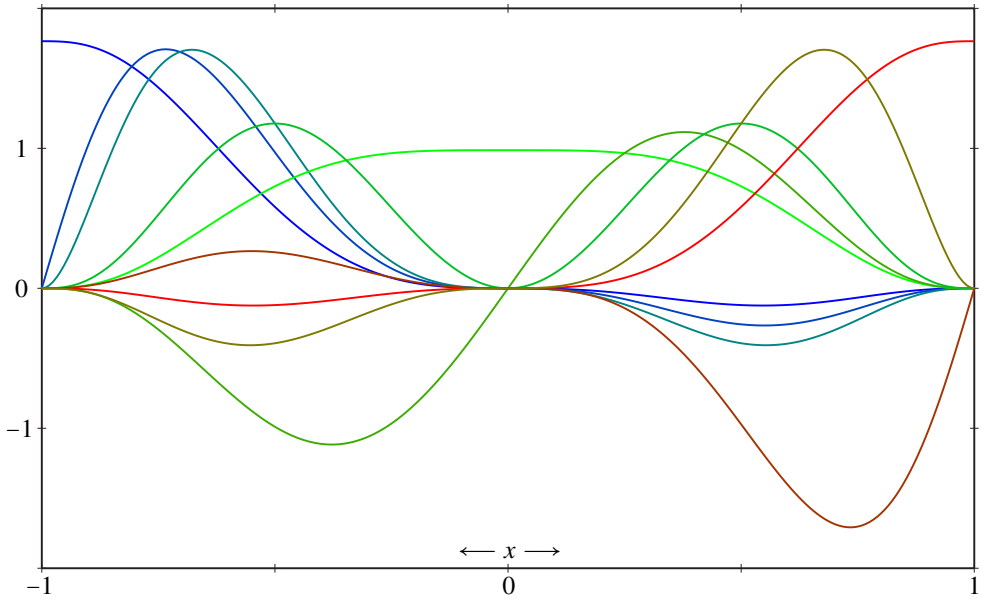



Figure 7.1: The different basis functions for a $\mathcal{C}^2$ continuous grid where 3 grid-points at positions, $x \in \{-1, 0, 1\}$, are interpolated. The polynomial order is 8 ($\mathcal{O}(x^9)$ terms are discarded).

# Appendix A

# Governing Equations

## A.1 Navier–Stokes Equations in Physical Dimensions

Weak compressibility and a Newtonian fluid is assumed and temperature dependence is ignored.

$$0 = \frac{\partial \rho'}{\partial t'} + \nabla' \cdot (\rho' \mathbf{v}') \tag{A.1a}$$

$$0 = \rho' \left( \frac{\partial \mathbf{v}'}{\partial t'} + \mathbf{v}' \cdot \nabla' \mathbf{v}' \right) + \nabla' p' - \nabla' \cdot \mathbb{T}' - \rho' \mathbf{g}' \tag{A.1b}$$

$$0 = \frac{\partial f'}{\partial t'} + \mathbf{v}' \cdot \nabla' f' \tag{A.1c}$$

In (A.1) $\rho'$ is density, $p'$ is pressure, $\mathbf{v}'$ is velocity, $\mathbb{T}'$ is the viscous stress tensor and $f'$ defines the fluid phase. If the system is closed we also have

$$0 = \int \frac{\partial (\rho' \mathbf{v}')}{\partial t'} dV' \tag{A.2a}$$

$$0 = \int \frac{\partial \rho'}{\partial t'} dV' \tag{A.2b}$$

Where $V'$ is the volume of the closed system. The equation of state is approximated by a linear relation between density and pressure:

$$p' - p'_{amb} = k(\rho' - \rho_{amb}), \text{ thus} \tag{A.3a}$$

$$\frac{\partial \rho'}{\partial t'} = \frac{1}{k} \frac{\partial p'}{\partial t'} \tag{A.3b}$$

$$\nabla' (\rho' \mathbf{v}') = \frac{1}{k} \nabla' \cdot \left[ \mathbf{v}' \left( p' - p'_{amb} \right) \right] + \rho_{amb} \nabla' \cdot \mathbf{v}' \tag{A.3c}$$

The viscous stress tensor is given by

$$\mathbb{T}' = \mu \left[ \nabla' \mathbf{v}' + (\nabla' \mathbf{v}')^T \right] + \lambda (\nabla' \cdot \mathbf{v}') \mathbb{I} \tag{A.4}$$

Where the superscript, $T$, is the transpose and $\mathbb{I}$ is the identity tensor. The phase dependent properties are $\mu$ and $\lambda$ (first and second viscosity coefficients), $\rho_{amb}$ (ambient density) and $k = K_{amb}/\rho_{amb}$ where $K_{amb}$ is the bulk modulus at ambient conditions. The

density free formulation, (A.5), is obtained by solving (A.3) for the density, $\rho'$, and replacing the density in (A.1).

$$0 = \frac{1}{k}\left[\frac{\partial p'}{\partial t'} + \nabla' \cdot \left(\mathbf{v}'\left(p' - p'_{amb}\right)\right)\right] + \rho_{amb}\nabla' \cdot \mathbf{v}' \tag{A.5a}$$

$$0 = \frac{1}{k}\left[\left(p' - p'_{amb}\right)\left(\frac{\partial \mathbf{v}'}{\partial t'} + \mathbf{v}' \cdot \nabla'\mathbf{v}' - \mathbf{g}'\right)\right] + \rho_{amb}\left(\frac{\partial \mathbf{v}'}{\partial t'} + \mathbf{v}' \cdot \nabla'\mathbf{v}' - \mathbf{g}\right) + \nabla p' - \nabla' \cdot \mathbb{T}' \tag{A.5b}$$

$$0 = \frac{\partial f'}{\partial t'} + \mathbf{v}' \cdot \nabla' f' \tag{A.5c}$$

The density free global conservation equations read

$$0 = \int \left(\frac{1}{k}\frac{1}{\partial t'}\left(\mathbf{v}'\left(p' - p'_{amb}\right)\right) + \rho_{amb}\frac{\partial \mathbf{v}'}{\partial t'}\right) dV' \tag{A.6a}$$

$$0 = \int \frac{1}{k}\frac{\partial p'}{\partial t'} dV' \tag{A.6b}$$

## A.2 The Dimensionless Navier–Stokes Equations for Two Phases

The characteristic length and velocity scales, which map to unity in the computational domain, are $x_0$ and $v_0$ ($t_0 = v_0/x_0$) and define the dimensionless (non-primed) quantities:

$$\mathbf{v}' = v_0\mathbf{v} \tag{A.7a}$$

$$\rho' = \rho_0\rho \tag{A.7b}$$

$$p'_{amb} = \rho_0 v_0^2 p_{amb} \tag{A.7c}$$

$$p' = \rho_0 v_0^2 (p + p_{amb}) \tag{A.7d}$$

$$\mathbf{g}' = \frac{v_0^2}{x_0}\mathbf{g} \tag{A.7e}$$

$$\frac{\partial}{\partial t'} = \frac{v_0}{x_0}\frac{\partial}{\partial t} \tag{A.7f}$$

$$\nabla' = \frac{1}{x_0}\nabla \tag{A.7g}$$

$$\mathbb{T}' = \frac{\mu v_0}{x_0}\mathbb{T} = \frac{\mu v_0}{x_0}\left(\nabla\mathbf{v} + (\nabla\mathbf{v})^T + \frac{\lambda}{\mu}(\nabla \cdot \mathbf{v})\mathbb{I}\right) \tag{A.7h}$$

$$\sigma' = \sigma_0\sigma = \rho_0 x_0 v_0^2\sigma \tag{A.7i}$$

$$f' = f \tag{A.7j}$$

The dimensionless surface tension is given by the parameter $\sigma$, equal to the physical surface tension, $\sigma'$, scaled with $\sigma_0 \stackrel{\text{def}}{=} \rho_0 x_0 v_0^2$. The phase dependent properties are determined by the dimensionless parameters Re (viscosity), $\alpha$ (compressibility) and $\beta$

(density):

$$\mathrm{Re} = \frac{x_0 v_0 \rho_{amb}}{\mu} \tag{A.8a}$$

$$\alpha = \frac{k\rho_{amb}}{v_0^2 \rho_0} = \frac{K_{amb}}{v_0^2 \rho_0} \tag{A.8b}$$

$$\beta = \frac{\rho_0}{\rho_{amb}} \tag{A.8c}$$

We let $\lambda/\mu = -2/3$. The viscous stress term, written as an operator, $\mathbb{S}$, on the velocity reads

$$\nabla \cdot \mathbb{T} = \mathbb{S} \cdot \mathbf{v} = \begin{bmatrix} \nabla^2 + \frac{1}{3}\frac{\partial^2}{\partial x^2} & \frac{2}{3}\frac{\partial^2}{\partial x \partial y} & \frac{2}{3}\frac{\partial^2}{\partial x \partial z} \\ \frac{2}{3}\frac{\partial^2}{\partial x \partial y} & \nabla^2 + \frac{1}{3}\frac{\partial^2}{3\partial y^2} & \frac{2}{3}\frac{\partial^2}{\partial y \partial z} \\ \frac{2}{3}\frac{\partial^2}{\partial x \partial z} & \frac{2}{3}\frac{\partial^2}{\partial y \partial z} & \nabla^2 + \frac{1}{3}\frac{\partial^2}{3\partial z^2} \end{bmatrix} \cdot \mathbf{v} \tag{A.9}$$

The dimensionless formulation is then:

$$0 = \frac{1}{\alpha}\left[\frac{\partial p}{\partial t} + p\nabla \cdot \mathbf{v} + \mathbf{v} \cdot \nabla p\right] + \nabla \cdot \mathbf{v} \tag{A.10a}$$

$$0 = \frac{1}{\alpha}\left[p\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} - \mathbf{g}\right)\right] + \left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} - \mathbf{g}\right) + \beta \nabla p - \frac{1}{\mathrm{Re}} \mathbb{S} \cdot \mathbf{v} \tag{A.10b}$$

$$0 = \frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla f \tag{A.10c}$$

Since the fluids are weakly compressible, $\alpha$ is large and the bracketed terms make only a small contribution. If the flow is incompressible ($\alpha \to \infty \Rightarrow \nabla \cdot \mathbf{v} = 0$) it can be shown that $\mathbb{S}$ reduces to $\mathbb{I}\nabla^2$. In the single-phase case one would choose $\rho_0 = \rho_{amb}$ yielding $\beta = 1$. In the two-phase case $\rho_0$ might be set to $\rho_{amb}$ of one of the fluids, or something in between. The global mass conservation, separated for two different phases, $l$ and $g$, reads:

$$0 = \sqrt{\frac{\alpha_g \beta_g}{\alpha_l \beta_l}} \int_{V_l} \frac{\partial p}{\partial t} dV + \sqrt{\frac{\alpha_l \beta_l}{\alpha_g \beta_g}} \int_{V_g} \frac{\partial p}{\partial t} dV \tag{A.11}$$

## A.3 Uniform Change of Spatial and Temporal Scales

The spatial and temporal scales in (A.10) are defined so their size is equal to the interval $[0,1]^4$ in the computational domain. If the grid is uniform, floating point round off errors might be reduced by defining the characteristic length scales so that they instead correspond to the interval $[0, L-1]^3 \times [0, T-1]$ in the computational domain. With $L$ being the spatial grid resolution and $T$ the temporal grid resolution (i.e. the grid has $L \times L \times L \times T$ grid-points) the spacing between grid points becomes equal to one. The

equivalent set of equations are:

$$0 = \frac{1}{\alpha}\left[\tau\frac{\partial p}{\partial t} + p\nabla\cdot\mathbf{v} + \mathbf{v}\cdot\nabla p\right] + \nabla\cdot\mathbf{v} \tag{A.12a}$$

$$0 = \frac{1}{\alpha}\left[p\left(\tau\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v} - \frac{\mathbf{g}}{\eta}\right)\right] + \left(\tau\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v} - \frac{\mathbf{g}}{\eta}\right) + \beta\nabla p - \frac{\eta}{\mathrm{Re}}\mathbb{S}\cdot\mathbf{v} \tag{A.12b}$$

$$0 = \tau\frac{\partial f}{\partial t} + \mathbf{v}\cdot\nabla f \tag{A.12c}$$

where (A.12d)

$$\tau = \frac{T-1}{L-1} \tag{A.12e}$$

$$\eta = L - 1 \tag{A.12f}$$

## A.4 Surface Tension

Surface tension gives rise to a pressure discontinuity in the equilibrium case and may be approximated in terms of the phase function

$$\nabla' f'\cdot\nabla' p'_{\sigma'} = -\sigma'\frac{\partial s'}{\partial f'}|\nabla' f'|^2\nabla'\cdot\left(\frac{\nabla' f'}{|\nabla' f'|}\right) \tag{A.13}$$

Here $\Delta f'$ is a small interval which approximate the interface between the two phases. The dimensionless form, scaled with the grid resolution reads

$$\nabla f\cdot\nabla p_\sigma = -\eta\sigma\frac{\partial s}{\partial f}|\nabla f|^2\nabla\cdot\left(\frac{\nabla f}{|\nabla f|}\right) \tag{A.14}$$

Where $s(f)$ is a function which approximates the discontinuity of the pressure at the interface. Since the discontinuity is difficult to express accurately with continuous basis functions it might be added as an additional force (source term) in the momentum equation instead of incorporating it in the pressure directly. The momentum equation now reads

$$0 = \frac{1}{\alpha}\left[p\left(\tau\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v} - \frac{\mathbf{g}}{\eta}\right)\right] + \left(\tau\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v} - \frac{\mathbf{g}}{\eta}\right) + \beta\left(\nabla p - \eta\sigma\nabla f\frac{\partial s}{\partial f}\nabla\cdot\left(\frac{\nabla f}{|\nabla f|}\right)\right) - \frac{\eta}{\mathrm{Re}}\mathbb{S}\cdot\mathbf{v} \tag{A.15}$$

Alternatively, the distance from the interface may be approximated from $f$ and $s$ may be taken as a function of the distance, $r$, from the interface

$$0 = \frac{1}{\alpha}\left[p\left(\tau\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v} - \frac{\mathbf{g}}{\eta}\right)\right] + \left(\tau\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v} - \frac{\mathbf{g}}{\eta}\right) + \beta\left(\nabla p - \eta\sigma\frac{\nabla f}{|\nabla f|}\frac{\partial s}{\partial r}\nabla\cdot\left(\frac{\nabla f}{|\nabla f|}\right)\right) - \frac{\eta}{\mathrm{Re}}\mathbb{S}\cdot\mathbf{v} \tag{A.16}$$

# Appendix B

# Source Codes

## B.1 Laplace Equation Solver

Listing B.1 shows the complete source-code for the Laplace-equation solver. This does not depend on external libraries other than the C++ standard and may be copied into any modern compiler and tested as it is.

Listing B.1: Laplace Equation Solver

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <cstdlib>
using namespace std;

// grid properties
const int Omega = 3;              // number of derivatives
const double f0 = 1.0;            // lower boundary value
const double f1 = 10.0;           // upper boundary value
const double r0 = 1.0;            // lower boundary radius
const double r1 = 10.0;           // upper boundary radius
int K = 5;                        // number of grid-points
double A = (K-1)/(r1-r0);         // reference-frame change
vector<double> F(K*Omega,0.0);    // grid values

// Computes normalized basis functions for Omega = 3.
// The first argument is the grid cell coordinate.
// The basis functions are evaluated and stored in the
// array given in the second argument
void basis6(const double, double [6]);

// Evaluates a function based on grid-cell components
// given in an array (second argument), at coordinate
// given in the first argument
double evaluate6(const double, const double [6]);

// Returns the analytic reference solution:
// f(r) = a + b/r
double analytic(const double);

// Returns the computed solution
double computed(const double);
```

```
// Computes the laplacian vector product integrated over
// the k'th grid-cell. Result is stored in the array,
// which is treated as a 6 by 6 matrix (stored row-wise)
void sphLaplacianIntegral(const int, double [36]);

// Maps grid-components to non-boundary components
// returns -1 if given a boundary component
int mapComp(const int);

// Maps the non-boundary components to the grid-components
int unmapComp(const int);

// Computes the solution to our test problem for a given
// resolution
void solveSystem(const int);

// Solves a symmetric, linear equation system
void solveSym(const int,           // size
              vector<double> &,    // matrix
              vector<double> &);   // result

// Compares the computed solution to the analytic solution
double getRMSerror(void);

// Compute solutions for different grid resolutions
// print the RMS error for each resolution
int main()
{
    for(int i = 5; i <= 20; i++)
    {
        cout << "K = " << setw(3) << i;
        solveSystem(i);
        cout << ", RMS error: "
             << setw(12) << getRMSerror() << std::endl;
    }

    return 0;
}
// x is the coordinate, result is stored in B[]
void basis6(const double x, double B[6])
{
    const double x2 = x*x;
    const double x3 = x2*x;
    const double y = 1 - x;
    const double y2 = y*y;
    const double y3 = y2*y;

    B[0] = y3*(3*(2*x + 1)*x + 1);
    B[1] = 5*y3*x*(3*x + 1);
    B[2] = 30*y3*x2;
    B[3] = x3*(3*(2*y + 1)*y + 1);
    B[4] = -5*x3*y*(3*y + 1);
    B[5] = 30*x3*y2;
}

// x is the coordinate, F[] is the grid-cell components
double evaluate6(const double x, const double F[6])
```

```
{
    double f = 0.0;
    double B[6];
    basis6(x,B);

    for(int i = 0; i < 6; i++)
        f += B[i]*F[i];

    return f;
}

// Returns the analytic reference solution
double analytic(const double r)
{
    const double C0 = r1*f1 - r0*f0;
    const double C1 = r0*r1*(f1 - f0);
    return (C0 - C1/r)/(r1 - r0);
}

// Returns the computed solution
double computed(const double r)
{
    const double x = (r - r0)*A;
    const int xf = static_cast<int>(floor(x));
    const int k = max(min(xf,K-2),0);

    return evaluate6(x-k,&F[Omega*k]);
}

// Result is stored in the matrix object LLT
void sphLaplacianIntegral(const int k, double LLT[36])
{
    const double C = (A*r0 + k)/2;
    const double C2 = C*C;
    const double S = 210.0;

    LLT[0] = S*(8*C2+4*C+1)/98;
    LLT[1] = S*(240*C2+96*C+17)/1176;
    LLT[2] = S*(24*C2+16*C+3)/196;
    LLT[3] = -S*(8*C2+4*C+1)/98;
    LLT[4] = S*(240*C2+144*C+29)/1176;
    LLT[5] = -S*(24*C2+8*C+1)/196;

    LLT[6] = LLT[1];
    LLT[7] = S*(192*C2+31*C+12)/294;
    LLT[8] = S*(88*C2+48*C+9)/196;
    LLT[9] = -S*(240*C2+96*C+17)/1176;
    LLT[10] = S*(432*C2+216*C+37)/1176;
    LLT[11] = -S*((4*C-1)*(4*C+1))/98;

    LLT[12] = LLT[2];
    LLT[13] = LLT[8];
    LLT[14] = S*(3*(24*C2+6*C+1))/49;
    LLT[15] = -S*(24*C2+16*C+3)/196;
    LLT[16] = S*((4*C+1)*(4*C+3))/98;
    LLT[17] = S*(3*(8*C2+4*C+1))/98;

    LLT[18] = LLT[3];
```

```
    LLT[19] = LLT[9];
    LLT[20] = LLT[15];
    LLT[21] = S*(8*C2+4*C+1)/98;
    LLT[22] = -S*(240*C2+144*C+29)/1176;
    LLT[23] = S*(24*C2+8*C+1)/196;

    LLT[24] = LLT[4];
    LLT[25] = LLT[10];
    LLT[26] = LLT[16];
    LLT[27] = LLT[22];
    LLT[28] = S*(384*C2+322*C+89)/588;
    LLT[29] = -S*(88*C2+40*C+7)/196;

    LLT[30] = LLT[5];
    LLT[31] = LLT[11];
    LLT[32] = LLT[17];
    LLT[33] = LLT[23];
    LLT[34] = LLT[29];
    LLT[35] = S*(6*(12*C2+9*C+2))/49;
}

// fi is the grid-component index
int mapComp(const int fi)
{
    const int bi = Omega*(K-1);

    if(fi < bi) return fi-1;
    if(fi > bi) return fi-2;

    return -1;
}

// si is the index of the solution vector
int unmapComp(const int si)
{
    const int bi = Omega*(K-1);

    if(si + 1 < bi) return si + 1;

    return si + 2;
}

// Computes the solution to our test problem
void solveSystem(const int num_gridpoints)
{
    K = num_gridpoints;
    F.resize(K*Omega,0.0);
    A = (K-1)/(r1-r0);

    // Set boundary components (a0 = 1)
    F[0] = f0;
    F[Omega*(K-1)] = f1;

    // Instantiate and initialize matrices
    const int N = Omega*K - 2; // system size
    vector<double> M(N*N,0.0); // system matrix
    vector<double> B(N,0.0);   // system result
    double LLT[4*Omega*Omega]; // cell system
```

```
    // Loop over all the grid-cells
    for(int k = 0; k < K-1; k++)
    {
        // Get the laplacian integral for current cell
        sphLaplacianIntegral(k,LLT);

        // Add the cell-system to the grid-wide system
        const int i0 = Omega*k;
        for(int i = 0; i < 2*Omega; i++)
        {
            const int row = mapComp(i0 + i);

            // ignore rows corresponding to the boundary
            if(row >= 0)
                for(int j = 0; j < 2*Omega; j++)
                {
                    const int col = mapComp(i0 + j);
                    const double Lij = LLT[2*Omega*i + j];

                    // col >= 0:
                    // non-boundary-> add to matrix

                    // col < 0:
                    // boundary-> add to solution

                    if(col >= 0)
                        M[N*row + col] += Lij;
                    else
                        B[row] -= Lij*F[i0 + j];
                }
        }
    }

    // solve the linear system
    solveSym(N,M,B);

    // copy solution back to grid
    for(int i = 0; i < N; i++)
        F[unmapComp(i)] = B[i];
}

// Solves the matrix equation Ax = B, where:
// A is assumed symmetric with nonzero diagonal elements,
// N is the dimension and the result is stored in B
void solveSym(const int N,          // size
              vector<double> &A,    // matrix
              vector<double> &B);   // result
{
    for(int d = 0; d < N-1; d++)
        for(int r = d+1; r < N; r++)
        {
            const double le1 = -A[N*d + r]/A[N*d + d];

            const double * const U = &A[N*d + r];
            double * const V = &A[N*r + r];
            const int M = N-r;
```

```
            for(int n = 0; n < M; n++)
                V[n] += U[n]*lel;

            B[r] += B[d]*lel;

            A[N*d + r] = lel;
        }

    for(int d = 0; d < N; d++)
        B[d] /= A[N*d + d];

    for(int r = N-2; r >= 0; r--)
        for(int i = N-1; i > r; i--)
            B[r] += A[N*r + i]*B[i];
}


// Compares the computed solution to the analytic solution
double getRMSerror(void)
{
    const int num_samp = 10000*(K-1)*Omega;
    double R2 = 0.0;

    for(int i = 0; i < num_samp; i++)
    {
        const double r = r0 + ((r1 - r0)*rand())/RAND_MAX;
        const double dR = computed(r) - analytic(r);

        R2 += dR*dR;
    }

    return sqrt(R2/num_samp);
}
```

## B.2 Matrix Classes

Listings B.2-B.3 shows the header files (member declarations) of the matrix classes developed for this thesis. The purpose of each member function is explained in the comments on the same line or above. The implementations of the matrix classes are too large to include in a printed paper but will be made available on-line or by request to the author. Some member functions are capable of using GPU accelerated parallel processing and requires initialized OpenCL context, command queue and programs.

Listing B.2: Matrix Class Header

```
#ifndef MATRIX_H_INCLUDED
#define MATRIX_H_INCLUDED

#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>

#include "real.h"
```

```
class Matrix {
public:
    Matrix(void); // creates empty matrix
    Matrix(const int, const int); // allocates storage
    Matrix(const Matrix &); // creates a copy of argument
    ~Matrix(void);
    void setSize(const int, const int);
    void setIdentity(const int); // initializes an identity matrix
    void setRandom(void); // randomizes matrix (normal distribution)
    void setRandomSym(void); // random symmetrix matrix
    void setAll(const REAL_TYPE); // set all elements equal to argument

    // opetators
    Matrix & operator = (const Matrix &);
    Matrix & operator *= (const REAL_TYPE);
    Matrix & operator -= (const Matrix &);
    Matrix & operator += (const Matrix &);
    const REAL_TYPE & operator () (const int, const int) const;
    REAL_TYPE & operator () (const int, const int);
    Matrix operator + (const Matrix &) const;
    Matrix operator - (const Matrix &) const;
    Matrix operator * (const Matrix &) const;
    Matrix operator * (const REAL_TYPE &) const;

        Matrix & solve(Matrix &); // partially pivoted gaussian elimination
    Matrix & solveSym(Matrix &); // symmetric gaussian elimination
    Matrix & solveNopiv(Matrix &); // non-pivoted gaussian elimination
    void solveChol(Matrix &); // solve via cholesky factorization
    Matrix & invert(void); // inverts via gaussian elimination on identity
    Matrix getTranspose(void) const; // returns a transposed copy
    void normalizeRows(Matrix &); // scales rows by infinity-norm

        // returns sub matrix based on argument indices
    Matrix sub(const int, const int, const int, const int) const;

        // different norm types and conditioning
    REAL_TYPE maxAbsElm(void) const;
    REAL_TYPE frobNorm(void) const;
    REAL_TYPE frobNormSq(void) const;
    REAL_TYPE svdNorm(void) const;
    REAL_TYPE conditionNumber(void) const;

    void QRlsqr(Matrix &B);  // pseudo inverse via QR factorization
    void NRMlsqr(Matrix &B); // pseudo inverse via normal equations

        // singular value decomposition (adapted from numerical recipes)
    void svdcmp(REAL_TYPE [], Matrix &);

    const int map(const int, const int) const; // memory layout

    // save to disk in various text formats
    void saveForm(const char *) const;
    void saveForm(std::ofstream &) const;
    void saveOctaveForm(const char *) const;

        // produces symmetric matrix by copying upper triangular to lower
    void copyUpperTriToSubTri(void);
```

```
    int rows,cols; // size
    REAL_TYPE *data; // pointer to data in memory
};

#endif // MATRIX_H_INCLUDED
```

Listing B.3: Sparse Matrix Class Header

```
#ifndef SPARSEMAT_H
#define SPARSEMAT_H

#include <thread>
#include <CL/cl.hpp>
#include "Matrix.h"

class Sparsemat
{
public:
    // indexed matrix element
    struct Elm
    {
        int c;
        REAL_TYPE v;
    };

    // row operation (mult p with s and add to q)
    struct RowOp
    {
        int p, q;
        REAL_TYPE s;
    };

    // householder reflection with rows and angles
    struct HouseholderRef
    {
        int p, q;
        REAL_TYPE c, s;
    };

    // banded storage scheme used for preconditioning
    struct BandedConSym
    {
        int rows, bwdt;
        std::vector<REAL_TYPE> data;
    };

    // matrix row of indexed elements starting from diagonal
    class Row
    {
    public:
        Row(const int id) : r(id) { }
        Row(void) : r(0) { }
        Row(const Row &other) : r(other.r), elm(other.elm) { }
        ~Row(void) { }
        void getRawRow(std::vector<REAL_TYPE> &) const;
        int numElm(void) const { return static_cast<int>(elm.size()); }
        int elmPos(const int) const;
        bool nonzero(const int) const;
```

```
    REAL_TYPE getElm(const int) const;
    REAL_TYPE frobNormSq(void) const;
    void setElm(const int, const REAL_TYPE);
    void addToElm(const int, const REAL_TYPE);
    void multElm(const int, const REAL_TYPE);
    void divElm(const int, const REAL_TYPE);
    void cgPrecon(const Matrix &);
    void multRow(REAL_TYPE &, const Matrix &) const;
    void multColSub(Matrix &, const REAL_TYPE) const;
    void zeroAll(void);
    void clear(void);

    int r;
    std::vector<Elm> elm;
};

    // initialization
Sparsemat(void);
Sparsemat(const int);
virtual ~Sparsemat(void);
Sparsemat(const Sparsemat &other);
Sparsemat & operator = (const Sparsemat &other);

bool exists(const int, const int) const; // is element allocated?
bool nonzero(const int, const int) const;  // true if nonzero
REAL_TYPE getElm(const int, const int) const; // returns value
int maxBand(void) const; // largest non-zero distance from diagonal

    // various operations on elements, allocates memory if necessary
void setElm(const int, const int, const REAL_TYPE);
void addToElm(const int, const int, const REAL_TYPE);
void multElm(const int, const int, const REAL_TYPE);
void divElm(const int, const int, const REAL_TYPE);

    // multiply with positive-definite matrix
const Matrix posDefMult(const Matrix &) const;

    // multiply with positive-definite matrix with an added diagonal
const Matrix posDefMultShift(const Matrix &, const Matrix &) const;

    // apply preconditioner
const Matrix cgPrecon(Matrix &, Matrix &);
const Matrix cgPreconShift(Matrix &, Matrix &, Matrix &);

    // conjugate gradient with various preconditioners and optional
    // utilization of GPU device for parallel computing
const REAL_TYPE conjugateGradient(Matrix &, const Matrix &,
                                  const REAL_TYPE, int &, const int);
const REAL_TYPE cgPre(Matrix &, Matrix &, const REAL_TYPE,
                                                  int &, const int);
const REAL_TYPE cgPre(Matrix &, Matrix &, const REAL_TYPE, int &,
          const int, cl::Context *, cl::CommandQueue *, cl::Program *);
const REAL_TYPE cgPre(Matrix &, Matrix &, const BandedConSym &,
                      cl::Context *, cl::CommandQueue *, cl::Program *);
const REAL_TYPE cgPre(Matrix &, Matrix &, int &, const BandedConSym &,
                      cl::Context *, cl::CommandQueue *, cl::Program *);
const REAL_TYPE cgPre(Matrix &, Matrix &, cl::Context *,
                                  cl::CommandQueue *, cl::Program *);
```

```
    const REAL_TYPE cgPreShift(Matrix &, Matrix &, Matrix &, cl::Context *,
                                        cl::CommandQueue *, cl::Program *);

        // expand data allocation into sub-diagonal region (by copying)
    void setExplicitSubdiag(void);

    REAL_TYPE frobNorm(void) const;

        // apply householder reflection
    void householderPrecon(std::vector<HouseholderRef> &);
        Matrix applyOperationTran(const Matrix &,
                                  const std::vector<HouseholderRef> &) const;
    Matrix applyOperation(const Matrix &,
                                  const std::vector<HouseholderRef> &) const;

        // compute cholesky factorization and save as banded preconditioner
    void choleskyFac(BandedConSym &);
    void choleskyFac(BandedConSym &, cl::Context *,
                                        cl::CommandQueue *, cl::Program *);

    Matrix solveChol(Matrix &); // solve via cholesky factorization
    Matrix solveSym(Matrix &);  // solve via symmetric gaussian elimination
    Matrix solveSymShift(Matrix &, Matrix &);
    const int dims(void) const; // get dimension
    void convertToRawData(BandedConSym &);  // convert storage scheme
    void createFromRawData(const BandedConSym &);
    void invert(void); // inverse via gaussian elimination on identity
        void saveFormatted(std::ofstream &) const; // save as text file
    void clear(void); // release memory
    void setSize(const int); // set dimension
    void zeroAll(void); // set all elements equal to zero (releases memory)

    std::vector<Row> row; // matrix data
};

#endif // SPARSEMAT_H
```

# Bibliography


[1] *Proc. Coupled Problems 2015*, 2015. CIMNE. 5.1

[2] *Proc. Multiphase Flow 2015*, volume 89 of *WIT Transactions on Engineering Sciences*, 2015. Witpress. 6.2

[3] E. Erturk, T. C. Corke, and Gökçöl. Numercal solutions of 2-d steady incompressible driven cavity flow at high reynolds numbers. *International Journal for Numerical Methods in Fluids*, 48, 2005. 6.1

[4] U. Ghia, K. N. Ghia, and T. C. Shin. High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method. *Journal of Computational Physics*, 48, 1982. 6.1

[5] M. Y. Hussaini. Spectral methods in fluid dynamics. *NASA Langley Research Center*, 1986. 1.4

[6] J. J. Monaghan. Smoothed particle hydrodynamics. *Annu. Rev. Astron. Astrophys*, 30:543–74, 1992. 1.3

[7] H. Takewaki, A. Nishiguri, and T. Yabe. Cubic interpolated pseudo-particle method (CIP) for solving hyperbolic-type equations. *Journal of Computational Physics*, 61, 1984. 3

[8] Lloyd N. Trefethen and David Bau, III. *Numerical Linear Algebra*. Siam, Philadelphia, 1997. 1

[9] Jesper Tveit. A numerical approach to solving nonlinear differential equations on a grid with potential applicability to computational fluid dynamics. 2014. 3.3.1, 6.1, 1

[10] Jesper Tveit. A high order approach to solving nonlinear differential equations applied to direct numerical simulation of two-phase undsteady flow. *Proc. Computational Methods in Multiphase Flow*, VIII, 2015. 5.1, 6.2

[11] E. M. Wahba. Steady flow simulations inside a driven cavity up to reynolds number 35,000. *Computers and Fluids*, 66, 2012. 6.1