UNIVERSITÉ DE GENÈVE FACULTÉ DES SCIENCES
Section de mathématiques

Simulation of Jet D'Eau from A to Y: Numerical algorithm to solve incompressible Navier-Stokes equations with marker and cell based method

Pablo Strasser

Master thesis under the supervision of
Prof. Martin Gander and Dr. Felix Kwok

Geneva, March 3, 2013

Abstract

We will in this work present how to solve the Navier-Stokes equations with a marker and cell method. We use the direct the Navier-Stokes approach without other models than the Navier-Stokes equations. We emphasize the mathematical modularity which allows to work separately on every piece of this work. Our method works for every Runge-Kutta method (explicit or implicit). If we denote by k the order of the chosen Runge-Kutta method, our method is of order k if we ignore boundary conditions, of order k in the better case and of order 1 for specific complicated case. Our method is exactly divergence free (up to roundoff error).

We will specifically be interested in free surface boundary conditions. The final goal is to simulate a Jet d'Eau. This goal was only partially achieved, but the needed basics are there.

Contents

Introduction

This work consists of the following chapters:

1. A notation and basic math property chapter.
2. A chapter that describes the problem analytically. In this chapter no discretization is done, but equations are written in the good form to motivate discretization.
3. A chapter that focuses on the solution of Navier-Stokes equations if the domain of computation is fixed.
4. A chapter that indicates the change needed to allow domain change.
5. To end, a conclusion chapter which will contain hint and discussion to how to upgrade the scheme used.

The two numerical chapters contain pseudo code that implement the described algorithm.

The original **C++11** (c++ code respecting the 2011 norm ISO/IEC 14882:2011) code can be found on `https://github.com/Strasser-Pablo/Mac` but is not needed to understand this work, the pseudo code is sufficient.

All the chapters are self contained, but further information and inspiration come from:

- Algorithmic ideas like the layer variable and the first scheme come from [2].
- Analytical comprehension and better formulation come from [5].
- Extrapolation for boundary conditions come from [7].
- Analytical solutions are taken from [1] and [3].
- A review of marker and cell based methods can be found in [6].

Chapter 1

Notation and basic property

Contents

1.1 Vector

We denote a vector with the dimension of physical space (2d or 3d) by $\boldsymbol{v}$. We only use Cartesian coordinate systems. We indicate a given component of a vector with indices, e.g. $\boldsymbol{v}_1$ for the first component.

1.2 Differential operator

We denote derivatives by ∂_i, where i is the spatial component with respect to which we take the derivative,

$$\partial_i v(\boldsymbol{x}) = \frac{\partial v(\boldsymbol{x})}{\partial \boldsymbol{x}_i}. \tag{1.1}$$

Definition 1.2.1 (Nabla)**.** *In Cartesian coordinates, the $\boldsymbol{\nabla}$ is a vector that cannot commute (e.g. $\nabla \cdot \boldsymbol{v} \neq \boldsymbol{v} \cdot \nabla$). Its expression is given by:*

$$\boldsymbol{\nabla}_i = \partial_i. \tag{1.2}$$

With the $\boldsymbol{\nabla}$ we can form the following expression. With n the dimension of the vector:

$$\boldsymbol{\nabla} p = \begin{pmatrix} \partial_1 p \\ \vdots \\ \partial_n p \end{pmatrix}, \tag{1.3}$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{v} = \sum_{i=1}^{n} \partial_i \boldsymbol{v}_i, \tag{1.4}$$

$$((\boldsymbol{v} \cdot \boldsymbol{\nabla})\boldsymbol{v})_i = \sum_{j=1}^{n} v_j \partial_j v_i. \tag{1.5}$$

Definition 1.2.2 (Laplacian)**.** *The Laplacian Δ consists of a scalar operator:*

$$\Delta p = \boldsymbol{\nabla} \cdot \boldsymbol{\nabla} p = \sum_{i=1}^{n} \partial_i \partial_i p. \tag{1.6}$$

Definition 1.2.3 (Vector Laplacian)**.** *A vector version of the Laplacian consists of applying the Laplacian to all components of the vector.*

$$\Delta \boldsymbol{v} = \begin{pmatrix} \Delta v_1 \\ \vdots \\ \Delta v_n \end{pmatrix}. \tag{1.7}$$

1.3 Divergence free and rotational free space

Definition 1.3.1 (Divergence)**.** *The divergence of a vector consists of*

$$\boldsymbol{\nabla} \cdot \boldsymbol{v}. \tag{1.8}$$

Definition 1.3.2 (Curl)**.** *The curl of vector consists of*

$$\boldsymbol{\nabla} \times \boldsymbol{v}. \tag{1.9}$$

Definition 1.3.3 (Gradient)**.** *The gradient of a scalar consists of*

$$\boldsymbol{\nabla} p. \tag{1.10}$$

Property 1. *The gradient is curl free:*

$$\boldsymbol{\nabla} \times \boldsymbol{\nabla} p = \mathbf{0}. \tag{1.11}$$

Property 2. *The curl is divergence free:*

$$\boldsymbol{\nabla} \cdot \boldsymbol{\nabla} \times \boldsymbol{v} = 0. \tag{1.12}$$

1.4 Projection

Property 3 (Projection of a divergence free space)**.** *Every vector field can be projected onto a divergence free space without changing its curl with the following change:*

$$\boldsymbol{\nabla} \boldsymbol{v}_{\boldsymbol{new}} = \boldsymbol{v} - \boldsymbol{\nabla} p, \tag{1.13}$$

$$\Delta p = \boldsymbol{\nabla} \cdot \boldsymbol{v}. \tag{1.14}$$

Proof Taking the divergence of equation (1.13) and using (1.14)

$$\boldsymbol{\nabla} \cdot \boldsymbol{v}_{\boldsymbol{new}} = \boldsymbol{\nabla} \cdot \boldsymbol{v} - \Delta p = \boldsymbol{\nabla} \cdot \boldsymbol{v} - \boldsymbol{\nabla} \cdot \boldsymbol{v} = 0. \tag{1.15}$$

Taking the curl of equation (1.13) and using (1.14)

$$\boldsymbol{\nabla} \times \boldsymbol{v}_{\boldsymbol{new}} = \boldsymbol{\nabla} \times \boldsymbol{v} - \boldsymbol{\nabla} \times \boldsymbol{\nabla} p = \boldsymbol{\nabla} \times \boldsymbol{v}. \tag{1.16}$$

■

Remark 1.4.1. *We have a Poisson's equation to solve. With Dirichlet and Neumann boundary condition we have a unique solution (if we are not Neumann everywhere).*

Remark 1.4.2. *This works as long as*

$$\Delta = \boldsymbol{\nabla} \cdot \boldsymbol{\nabla}. \tag{1.17}$$

So this can also be used in the discretized case and will be exact as long as (1.17) is respected.

Definition 1.4.1 (Linear projection operator)**.** *We define the projection operator* $P(\boldsymbol{v})$

$$P(\boldsymbol{v}) = \boldsymbol{\nabla}\Delta^{-1}\boldsymbol{\nabla} \cdot \boldsymbol{v}. \tag{1.18}$$

The projection of section 1.4 is then

$$\boldsymbol{v}_{new} = \boldsymbol{v} - P(\boldsymbol{v}). \tag{1.19}$$

In case we use boundary conditions that are not homogeneous Dirichlet or Neumann boundary condition, the projection operator is no longer linear, but affine.

Definition 1.4.2 (Affine projection operator)**.** *We denote the affine projection by* $P(\boldsymbol{v}, \boldsymbol{T}) = A\boldsymbol{v} + \boldsymbol{T}$*, where* A *is a matrice and* $\boldsymbol{T}$ *a vector.*

The Poisson equation is given by

$$\Delta p = \boldsymbol{\nabla} \cdot \boldsymbol{v}. \tag{1.20}$$

This can be rewritten for Dirchlet or Neumann boundary conditions by

$$Bp = \boldsymbol{\nabla} \cdot \boldsymbol{v} - \boldsymbol{c}, \tag{1.21}$$

where B *is a matrix form of* Δ *and* $\boldsymbol{c}$ *is given by the boundary conditions.* p *is then given by*

$$p = B^{-1}\boldsymbol{\nabla} \cdot \boldsymbol{v} - B^{-1}\boldsymbol{c}, \tag{1.22}$$

and the projection is

$$P(\boldsymbol{v}, \boldsymbol{T}) = -\boldsymbol{\nabla}p = -\boldsymbol{\nabla}B^{-1}\boldsymbol{\nabla} \cdot \boldsymbol{v} + \boldsymbol{\nabla}B^{-1}\boldsymbol{c}. \tag{1.23}$$

This can be rewritten as

$$P(\boldsymbol{v}, \boldsymbol{T}) = A\boldsymbol{v} + \boldsymbol{T}, \tag{1.24}$$

$$A = \boldsymbol{\nabla}B^{-1}\boldsymbol{\nabla} \cdot, \tag{1.25}$$

$$\boldsymbol{T} = \boldsymbol{\nabla}B^{-1}\boldsymbol{c}. \tag{1.26}$$

The important change is that the projection of a divergence free vector field is no more 0,

$$P(\boldsymbol{v}, \boldsymbol{T}) = A\boldsymbol{v} + \boldsymbol{T} = 0 + \boldsymbol{T} = \boldsymbol{T}. \tag{1.27}$$

Chapter 2

Analytical problem and analytical result

Contents

Introduction

In this chapter, we will discuss the analytical problem that we want to solve. The other numerical chapters will consist of numerical schemes to solve the problem given in this chapter.

2.1 Variables and parameters

The problem is to find a function of space and time in a certain domain respecting some conditions and equations.

We begin to define space, time and domain:

The *space* consists of a vector in fixed Cartesian coordinates with dimension typically 2 or 3. Space is denoted by $\boldsymbol{x}$.

The *time* consists of a parameter indicating the evolution of the function in time. Time is denoted by t.

The *domain* is the spatial region where a field is defined. This consists physically in the spatial region where water lies. Boundary conditions need to be provided. The domain can vary with respect to time and can vary with respect to the solution at previous times. The domain does not need to be connected, but can be decomposed into a union of connected subdomains.

Example 2.1.1. *Take a drop of water falling into a glass of water, the domain consists of the region where the water lies. At the beginning we have a domain consisting of two disconnected subdomains (the water in the drop and the water in the glass) that after a given time will merge into one domain.*

The domain is generally denoted by Ω or $\Omega(t)$ to emphasize that it can be variable with respect to time.

The variable of the equation is named *Field* and consists of functions with respect to space in the spatial domain and time in the temporal domain,

$$F(x,t) \qquad \text{with } x \in \Omega(t) \text{ and } t_0 \leq t \leq t_f. \tag{2.1}$$

We have the following fields:

The *speed*: A vector field consisting of the speed in space and time. The speed is denoted by $\boldsymbol{v}$.

The *pressure*: A scalar field consisting of an internal force per unit area in the fluid. We will not use the pressure in this work, but the pressure divided by density. If we denote the pressure by $\tilde{p}$ we use

$$p = \frac{\tilde{p}}{\rho}. \tag{2.2}$$

Because we use the incompressible Navier-Stokes equation, ρ is constant. This can be viewed as a change of units.

The following constant parameters are used:

The *viscosity* represents how much water particles stick together. The higher the viscosity, the less abrupt speed changes are possible. The viscosity is denoted by μ. We use ν that is the viscosity divided by the density.

$$\nu = \frac{\mu}{\rho}. \tag{2.3}$$

The *density* is a mass per unit volume. Density plays the same role as mass in Newtonian mechanics. In the incompressible Navier-Stokes equation density is a constant because incompressibility tells us that density does not change. The density is denoted by ρ.

The *external force* is a user defined vector field that represents the external force applied to the water. It can possibly depend on the solution at previous times. The force is denoted by $\boldsymbol{F}$.

2.2 Navier-Stokes equations

2.2.1 Fixed Eulerian topology

The incompressible Navier-Stokes equations are given by

$$\boldsymbol{\nabla} \cdot \boldsymbol{v}(\boldsymbol{x},t) = 0, \tag{2.4a}$$

$$\partial_t \boldsymbol{v}(\boldsymbol{x},t) + (\boldsymbol{v}(\boldsymbol{x},t) \cdot \boldsymbol{\nabla})\boldsymbol{v}(\boldsymbol{x},t) = -\boldsymbol{\nabla} p(\boldsymbol{x},t) + \frac{\boldsymbol{F}(\boldsymbol{x},t)}{\rho(\boldsymbol{x},t)} + \nu \Delta \boldsymbol{v}(\boldsymbol{x},t). \tag{2.4b}$$

Equation (2.4a), tells us that the speed is a divergence free field. Equation (2.4b) is an equation of evolution. The presence of p allows to maintain the speed always divergence free. Taking the curl of (2.4b) makes disappear the pressure term (because the curl of a gradient is 0).

The pressure term can be seen as a correction of the divergence without changing the curl.

We can rewrite the Navier-Stokes equations as

$$\boldsymbol{\nabla} \cdot \boldsymbol{v}(\boldsymbol{x},t) = 0, \tag{2.5a}$$
$$\partial_t \boldsymbol{v}(\boldsymbol{x},t) = f(\boldsymbol{v}(\boldsymbol{x},t)) - \boldsymbol{\nabla} p, \tag{2.5b}$$

where

$$f(\boldsymbol{v}(\boldsymbol{x},t)) = -(\boldsymbol{v}(\boldsymbol{x},t) \cdot \boldsymbol{\nabla})\boldsymbol{v}(\boldsymbol{x},t) + \frac{\boldsymbol{F}(\boldsymbol{x},t)}{\rho(\boldsymbol{x},t)} + \nu \Delta \boldsymbol{v}(\boldsymbol{x},t). \tag{2.5c}$$

Taking the divergence of equation (2.5b) we find

$$\Delta p = \boldsymbol{\nabla} \cdot f(\boldsymbol{v}). \tag{2.6}$$

But using the notation of (1.4.1):

$$\boldsymbol{\nabla} p = P(f(\boldsymbol{v})). \tag{2.7}$$

The equations of Navier-Stokes are then equivalent to solving

$$\partial_t \boldsymbol{v}(\boldsymbol{x},t) = f(\boldsymbol{v}(\boldsymbol{x},t)) - P(f(\boldsymbol{v}(\boldsymbol{x},t)) = (1-P) f(\boldsymbol{v}(\boldsymbol{x},t)). \tag{2.8}$$

$(1-P)$ is a projector onto a divergence free space without changing the rotational.

2.2.2 Mobile eulerian topology

When topology is variable, we need to add another equation to tell us where the domain of computation lies. A natural boundary condition is the free surface, which moves with the speed at a given point on the boundary.

We define λ a system of representatives, and $\boldsymbol{\xi}_\lambda$ the position of points given by λ. The domain is given by the union of $\boldsymbol{\xi}$ for all λ.

The Navier-Stokes equations are now

$$\boldsymbol{\nabla} \cdot \boldsymbol{v}(\boldsymbol{x},t) = 0, \tag{2.9a}$$
$$\partial_t \boldsymbol{v}(\boldsymbol{x},t) + (\boldsymbol{v}(\boldsymbol{x},t) \cdot \boldsymbol{\nabla})\boldsymbol{v}(\boldsymbol{x},t) = -\boldsymbol{\nabla} p(\boldsymbol{x},t) + \frac{\boldsymbol{F}(\boldsymbol{x},t)}{\rho(\boldsymbol{x},t)} + \nu \boldsymbol{\Delta} \boldsymbol{v}(\boldsymbol{x},t), \tag{2.9b}$$
$$\partial_t \boldsymbol{\xi}_\lambda(t) = \boldsymbol{v}(\boldsymbol{\xi}_\lambda, t). \tag{2.9c}$$

We can as in section 2.2.1 eliminate the constraint on the divergence with the same notation

$$\partial_t \boldsymbol{v}(\boldsymbol{x},t) = f(\boldsymbol{v}(\boldsymbol{x},t)) + P(f(\boldsymbol{v}(\boldsymbol{x},t)) = (1+P) f(\boldsymbol{v}(\boldsymbol{x},t)), \tag{2.10a}$$
$$\partial_t \boldsymbol{\xi}_\lambda(t) = \boldsymbol{v}(\boldsymbol{\xi}_\lambda, t). \tag{2.10b}$$

2.2.3 Lagrangian form of Navier-Stokes equations

We will in this section rewrite (2.4b) in another form. We begin to define a new position system (which is similar to the particle position used in the previous section) $\boldsymbol{\xi}_\lambda(t)$:

$$\partial_t \boldsymbol{\xi}_\lambda(t) = \boldsymbol{v}(\boldsymbol{\xi}_\lambda(t), t), \tag{2.11a}$$

$$\boldsymbol{\xi}_\lambda(t_0) = \boldsymbol{\xi}^0_\lambda. \tag{2.11b}$$

Where λ is a system of representatives that is initialized by an initial condition. For a given λ, $\boldsymbol{\xi}_\lambda(t)$ follows the speed at the given point. This position system is called particle coordinate or Lagrangian. $\boldsymbol{\xi}_\lambda(t)$ is called the characteristic.

We then can define the Lagrangian speed

$$\boldsymbol{u}_\lambda(t) = \boldsymbol{v}(\boldsymbol{\xi}_\lambda(t), t). \tag{2.12}$$

The Lagrangian speed is the speed following a characteristic.

Taking the total derivative (the derivative of everything that depend on time) of the Lagrangian speed with respect of time, we have

$$\frac{d\boldsymbol{u}_\lambda(t)}{dt} = \frac{d\boldsymbol{v}(\xi_\lambda, t)}{dt} = \partial_t \boldsymbol{v} + \left(\frac{\partial \boldsymbol{\xi}_\lambda(t)}{\partial t} \cdot \boldsymbol{\nabla}\right) \boldsymbol{v}. \tag{2.13}$$

This can be rewritten as

$$\frac{d\boldsymbol{u}_\lambda(t)}{dt} = \partial_t \boldsymbol{v} + (\boldsymbol{v} \cdot \boldsymbol{\nabla})\, \boldsymbol{v}. \tag{2.14}$$

This is exactly the lhs of equation (2.4b).

We then can rewrite the Navier-Stokes equations as

$$\boldsymbol{\nabla} \cdot \boldsymbol{u}_\lambda(t) = 0, \tag{2.15a}$$

$$\frac{d\boldsymbol{u}_\lambda(t)}{dt} = -\boldsymbol{\nabla} p(\boldsymbol{\xi}_\lambda(t), t) + \frac{\boldsymbol{F}(\boldsymbol{\xi}_\lambda(t), t)}{\rho(\boldsymbol{x}, t)} + \nu \Delta \boldsymbol{u}_\lambda(t). \tag{2.15b}$$

Remark 2.2.1. *We remark that with this notation some expression like spatial derivative of $\boldsymbol{u}$ is not defined. In an analytical procedure, we can transform $\boldsymbol{u}$ to $\boldsymbol{v}$ with the help of the expression of $\boldsymbol{\xi}$, then the derivative is well defined. But in a numerical procedure, the point where the values are known are particles. This requires us to define derivatives on unstructured grids or to interpolate at grid points from the particles.*

2.3 Boundary conditions

We will be interested in two kinds of boundary conditions.

***Inflow* Boundary:** An inflow boundary is a source of water.

***Free surface* Boundary:** A free surface boundary is a freely moving boundary where the other fluid is inert (no density and no viscosity).

2.3.1 Inflow

Inflow boundary conditions have a constant prescribed speed. This gives the following constraint on pressure:

$$\nabla p(\boldsymbol{x}, t) = f(\boldsymbol{v}(\boldsymbol{x}, t)). \tag{2.16}$$

The following alternative boundary conditions have been used. Set the speed on the surface artificially to constant and use homogeneous Neumann boundary conditions for pressure.

2.3.2 Free surface

To define the free surface boundary conditions, we need to define the stress tensor

$$\sigma_{ij} = -p\delta_{ij} + \nu \left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} \right). \tag{2.17}$$

The boundary conditions are then given by (for 3d):

$$\sum_{i,j} \sigma_{ij} n_i n_j = 0, \tag{2.18}$$

$$\sum_{i,j} \sigma_{ij} t^1_i n_j = 0, \tag{2.19}$$

$$\sum_{i,j} \sigma_{ij} t^2_i n_j = 0. \tag{2.20}$$

$$\tag{2.21}$$

where $\boldsymbol{n}$ is the normal vector and $\boldsymbol{t}^1$ and $\boldsymbol{t}^2$ are two non-colinear tangent vectors. For 2d only one tangent vector is needed.

The first equation can be used to find boundary conditions for p. The other equations only depend on speed, because $\boldsymbol{n}$ is orthogonal to $\boldsymbol{t}^1$ and $\boldsymbol{t}^2$. The value of ν is not relevant for the condition (a global non zero constant). The constraint is linear.

Chapter 3

Numerical treatment of the fixed domain case

Contents

3.1 Introduction

We will in this chapter be interested in the case of fixed domain Navier-Stokes with grid discretization. This case is easier than the variable domain case and serves as a basis for more complicated cases.

We will treat the discretization of time and space separately. We will first begin by time discretization. The space discretization will only be used to evaluate derivatives.

We will in this chapter present two ways to solve the Navier-Stokes equations. We will show that these two methods are equivalent if a Runge-Kutta method is used, and that we are exactly divergence free.

In the next subsection we will present an overview of the two methods.

3.1.1 Analytical expression for pressure

We use the work of section 2.2.1 that Navier-Stokes equations are analytically equivalent to

$$\partial_t \boldsymbol{v}(\boldsymbol{x},t) = (1-P)f(\boldsymbol{v}(\boldsymbol{x},t)). \tag{3.1}$$

$1-P$ projects a non divergence free vector to obtain a divergence free vector. The solution $\boldsymbol{v}$ is a divergence free vector as long that initial conditions are divergence free, because the integral of a divergence free vector is again a divergence free vector

$$\boldsymbol{v}(t) = \int_{t_0}^{t} \partial_{\tilde{t}} \boldsymbol{v}(\boldsymbol{x},\tilde{t}) = \int_{t_0}^{t} (1-P)f(\boldsymbol{v}(\boldsymbol{x},\tilde{t}))d\tilde{t} + \boldsymbol{v}_0,$$

$$\boldsymbol{\nabla}\cdot\boldsymbol{v}(t) = \boldsymbol{\nabla}\cdot\int_{t_0}^{t}(1-P)f(\boldsymbol{v}(\boldsymbol{x},\tilde{t}))d\tilde{t} + \boldsymbol{\nabla}\cdot\boldsymbol{v}_0 = \int_{t_0}^{t}\boldsymbol{\nabla}\cdot(1-P)f(\boldsymbol{v}(\boldsymbol{x},\tilde{t}))d\tilde{t} = \int_{t_0}^{t} 0 d\tilde{t} = 0.$$

But a priori the numerical solution is not necessarily divergence free.

3.1.2 Projection of speed

We can project every speed in a divergence free space

$$\partial_t \tilde{\boldsymbol{v}}(\boldsymbol{x},t) = f(\boldsymbol{v}(\boldsymbol{x},t)), \tag{3.2a}$$

$$\boldsymbol{v}(\boldsymbol{x},t) = (1-P)\tilde{\boldsymbol{v}}. \tag{3.2b}$$

$\boldsymbol{v}$ is certain to be always divergence free by construction.

3.2 Time discretization and integration

3.2.1 Runge-Kutta method

A Runge-Kutta method is a method to solve systems of ODEs (We drop the $\boldsymbol{x}$ dependence, but it is understood that this equation is for a vector formed in concatenating every speed)

$$\partial_t \boldsymbol{v}(t) = f(\boldsymbol{v}). \tag{3.3}$$

The solution at time $n+1$ is given from the solution at time n with Δt the time step and a, b parameters of the method,

$$\boldsymbol{v}_{n+1} = \boldsymbol{v}_n + \sum_{i=1}^{s} b_i \boldsymbol{k}_i, \tag{3.4a}$$

$$\boldsymbol{k}_i = \Delta t f(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\boldsymbol{k}_j). \tag{3.4b}$$

3.2.2 Runge-Kutta based linear projection

We now integrate equations given by methods (3.1.1) and (3.1.2) with a Runge-Kutta method.

The numerical scheme for the exact divergence free speed is

$$\tilde{\boldsymbol{v}}_{n+1} = (1-P)\tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \tilde{\boldsymbol{k}}_i, \tag{3.5a}$$

$$\tilde{\boldsymbol{k}}_i = \Delta t f\left((1-P)\left(\tilde{\boldsymbol{v}}_n + \sum_{j=1}^{i-1} a_{ij}\tilde{\boldsymbol{k}}_j\right)\right). \tag{3.5b}$$

We project at every input of the non linear f function and at the final speed, which by construction, makes every speed divergence free.

The scheme for the analytical expression for the pressure

$$\boldsymbol{v}_{n+1} = \boldsymbol{v}_n + \sum_{i=1}^{s} b_i \boldsymbol{k}_i, \tag{3.6a}$$

$$k_i = \Delta t f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\boldsymbol{k}_j\right) - \Delta t P\left(f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\boldsymbol{k}_j\right)\right). \tag{3.6b}$$

The following theorem shows that the two schemes are the same.

Theorem 3.1. *If $\boldsymbol{v}_n = \tilde{\boldsymbol{v}}_n$ and $\boldsymbol{v}_n$ is divergence free then $\boldsymbol{v}_{n+1} = \tilde{\boldsymbol{v}}_{n+1}$*

Proof We begin by proving the following lemma.

Lemma 3.2.

$$\boldsymbol{k}_i = \tilde{\boldsymbol{k}}_i - P(\tilde{\boldsymbol{k}}_i) \tag{3.7}$$

Proof By induction on i. We begin with $i = 1$:

$$\boldsymbol{k}_1 = \Delta t f(\boldsymbol{v}_n) - \Delta t P(f(\boldsymbol{v}_n)), \tag{3.8}$$

$$\tilde{\boldsymbol{k}}_1 = \Delta t f(\boldsymbol{v}_n), \tag{3.9}$$

$$\boldsymbol{k}_1 = \Delta t f(\boldsymbol{v}_n) - \Delta t P(f(\boldsymbol{v}_n)) = \tilde{\boldsymbol{k}}_1 - P\left(\tilde{\boldsymbol{k}}_1\right). \tag{3.10}$$

By induction hypothesis we are true for smaller i:

$$\boldsymbol{k}_i = \Delta t f\left(\boldsymbol{v}_n + \sum_{j=1}^{n} a_{ij} k_j\right) - \Delta t P\left(f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\boldsymbol{k}_j\right)\right).$$

Using the recurrence relation we have:

$$\boldsymbol{k}_i = \Delta t f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\left(\tilde{\boldsymbol{k}}_j - P\left(\tilde{\boldsymbol{k}}_j\right)\right)\right) - \Delta t P\left(f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\left(\tilde{\boldsymbol{k}}_j + P\left(\tilde{\boldsymbol{k}}_j\right)\right)\right)\right),$$

$$\tilde{\boldsymbol{k}}_i = \Delta t f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\tilde{\boldsymbol{k}}_j - P\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\tilde{\boldsymbol{k}}_j\right)\right).$$

Using the linearity of P and that $P\boldsymbol{v}_n = 0$:

$$\boldsymbol{k}_i = \Delta t f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\left(\tilde{\boldsymbol{k}}_j - P\left(\tilde{\boldsymbol{k}}_j\right)\right)\right),$$

$$\boldsymbol{k}_i = \tilde{\boldsymbol{k}}_i - P\left(\tilde{\boldsymbol{k}}_i\right).$$

■

We now write the expression for $\boldsymbol{v}_{n+1}$ and $\tilde{\boldsymbol{v}}_{n+1}$ using the lemma:

$$\begin{aligned}
\boldsymbol{v}_{n+1} &= \boldsymbol{v}_n + \sum_{i=1}^{s} b_i \boldsymbol{k}_i \\
&= \boldsymbol{v}_n + \sum_{i=1}^{s} b_i \left(\tilde{\boldsymbol{k}}_i - P\left(\tilde{\boldsymbol{k}}_i \right) \right), \\
\tilde{\boldsymbol{v}}_{n+1} &= \tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \tilde{\boldsymbol{k}}_i - P\left(\tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \tilde{\boldsymbol{k}}_i \right).
\end{aligned}$$

Using the linearity of P and that $P\tilde{\boldsymbol{v}}_n = 0$

$$\tilde{\boldsymbol{v}}_{n+1} = \tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \left(\tilde{\boldsymbol{k}}_i - P\left(\tilde{\boldsymbol{k}}_i \right) \right).$$

Using that $\boldsymbol{v}_n = \tilde{\boldsymbol{v}}_n$

$$\boldsymbol{v}_{n+1} = \tilde{\boldsymbol{v}}_{n+1}.$$

■

Theorem 3.3. *For the implicit Runge-Kutta case, where the sum to $i-1$ goes now to s, if $\boldsymbol{v}_n = \tilde{\boldsymbol{v}}_n$ and $\boldsymbol{v}_n$ is divergence free then $\boldsymbol{v}_{n+1} = \tilde{\boldsymbol{v}}_{n+1}$*

Proof We begin by proving the following lemma.

Lemma 3.4. *After relabeling noting $\boldsymbol{k}_i^j$ the j-th solution of $\boldsymbol{k}_i$. We have for every j,*

$$\boldsymbol{k}_i^j = \tilde{\boldsymbol{k}}_i^j + P(\tilde{\boldsymbol{k}}_i^j). \tag{3.11}$$

This does not say that we have existence and uniqueness of the solution only that if we have a set of solution for $\boldsymbol{k}$ we have a bijection to solutions of $\tilde{\boldsymbol{k}}$.

Proof Because the sum goes to s and not $i-1$ we cannot use recurrence. We now will use vector notation for $\boldsymbol{k}$.

We construct k and $\tilde{k}$ vector with the vector $\boldsymbol{k}_i$ and $\tilde{\boldsymbol{k}}_i$

$$k = \begin{pmatrix} \boldsymbol{k}_1 \\ \vdots \\ \boldsymbol{k}_s \end{pmatrix}, \tag{3.12}$$

$$\tilde{k} = \begin{pmatrix} \tilde{\boldsymbol{k}}_1 \\ \vdots \\ \tilde{\boldsymbol{k}}_s \end{pmatrix}. \tag{3.13}$$

We form $\hat{P}$ and $\hat{f}$ from a Kronecker product

$$\hat{P} = 1_s \bigotimes P = \begin{pmatrix} P & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & P \end{pmatrix}, \tag{3.14}$$

$$\hat{f} = 1_s \bigotimes f = \begin{pmatrix} f & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & f \end{pmatrix}, \tag{3.15}$$

where for $\hat{f}$ we have used the notation

$$\hat{f}\,(k) = \begin{pmatrix} f(\boldsymbol{k}_1) \\ \vdots \\ f(\boldsymbol{k}_n) \end{pmatrix}. \tag{3.16}$$

We now define $\hat{v}_n$

$$\hat{v}_n = \begin{pmatrix} \boldsymbol{v}_n \\ \vdots \\ \boldsymbol{v}_n \end{pmatrix}. \tag{3.17}$$

We now define $\hat{A}$ with the following Kronecker product

$$\hat{A} = \begin{pmatrix} a_{11} & \dots & a_{1s} \\ \vdots & \ddots & \vdots \\ a_{s1} & \dots & a_{ss} \end{pmatrix} \bigotimes 1 = A \bigotimes 1 = \begin{pmatrix} a_{11}1 & \dots & a_{1s}1 \\ \vdots & \ddots & \vdots \\ a_{s1}1 & \dots & a_{ss}1 \end{pmatrix}. \tag{3.18}$$

$\hat{A}$ and $\hat{P}$ commute, because of the mixed-product property

$$\hat{A}\hat{P} = \left(A \bigotimes 1\right)\left(1_s \bigotimes P\right) = A \bigotimes P, \tag{3.19}$$

$$\hat{P}\hat{A} = \left(1_s \bigotimes P\right)\left(A \bigotimes 1\right) = A \bigotimes P. \tag{3.20}$$

The two expressions can now be written:

$$k = (1 - \hat{P})\Delta t \hat{f}(\hat{v}_n + \hat{A}k), \tag{3.21}$$

$$\tilde{k} = \Delta t \hat{f}((1 - \hat{P})(\hat{v}_n + \hat{A}k)). \tag{3.22}$$

We begin to define 2 functions:

$$k = g(k), \tag{3.23}$$

$$\tilde{k} = \tilde{g}(\tilde{k}). \tag{3.24}$$

$$g(x) = (1 - \hat{P})\Delta t \hat{f}(\hat{v}_n + \hat{A}x), \tag{3.25}$$

$$\tilde{g}(x) = \Delta t \hat{f}((1 - \hat{P})(\hat{v}_n + \hat{A}x)). \tag{3.26}$$

k and $\tilde{k}$ are fixed points of functions g and $\tilde{g}$.
We now calculate:

$$(1 - \hat{P})\tilde{g}(x) = (1 - \hat{P})\Delta t \hat{f}((1 - \hat{P})(\hat{v}_n + \hat{A}x)). \tag{3.27}$$

Using

$$\hat{P}\hat{v}_n = 0. \tag{3.28}$$

Using that $\hat{A}$ and $\hat{P}$ commute

$$(1-\hat{P})\tilde{g}(x) = (1-\hat{P})\Delta t\hat{f}(\hat{v}_n + \hat{A}(1-\hat{P})x) \tag{3.29}$$
$$= g((1-\hat{P})x). \tag{3.30}$$

We now substitute x with $\tilde{k}$

$$(1-\hat{P})\tilde{g}(\tilde{k}) = g((1-\hat{P})\tilde{k}), \tag{3.31}$$

by definition $\tilde{k}$ is a fixed point of $\tilde{g}$

$$(1-\hat{P})\tilde{k} = g((1-\hat{P})\tilde{k}), \tag{3.32}$$

$(1-\hat{P})\tilde{k}$ is a fixed point of g. But fixed point of g is k.

With relabeling of solution if needed, this directly gives the result with $\boldsymbol{k}^j$ the j-th solution:

$$\boldsymbol{k}^j = (1-\hat{P})\tilde{\boldsymbol{k}}^j. \tag{3.33}$$

This is exactly the vector version of what we wanted to prove. ■

The rest is exactly the same as in the first case. To not overload the proof with notation, we drop the j label to indicate which solution we take.

We now write the expression for $\boldsymbol{v}_{n+1}$ and $\tilde{\boldsymbol{v}}_{n+1}$ using the lemma.

$$\begin{aligned}
\boldsymbol{v}_{n+1} &= \boldsymbol{v}_n + \sum_{i=1}^{s} b_i \boldsymbol{k}_i \\
&= \boldsymbol{v}_n + \sum_{i=1}^{s} b_i \left(\tilde{\boldsymbol{k}}_i - P\left(\tilde{\boldsymbol{k}}_i\right)\right). \\
\tilde{\boldsymbol{v}}_{n+1} &= \tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \tilde{\boldsymbol{k}}_i - P\left(\tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \tilde{\boldsymbol{k}}_i\right) \\
&= \tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \left(\tilde{\boldsymbol{k}}_i - P\left(\tilde{\boldsymbol{k}}_i\right)\right).
\end{aligned}$$

Using that $\boldsymbol{v}_n = \tilde{\boldsymbol{v}_n}$

$$\boldsymbol{v}_{n+1} = \tilde{\boldsymbol{v}}_{n+1}.$$

■

Corollary 3.5. *Solving Navier-Stokes equation with the analytical good choice of pressure is exactly divergence free at every approximation of speed.*

Proof The expression of $\boldsymbol{v}$ is the solution of the Navier-Stokes equation with the good choice of pressure. $\tilde{\boldsymbol{v}}$ is the solution of the Runge-Kutta equation without pressure which is projected at every speed estimation. $\boldsymbol{v}$ and $\tilde{\boldsymbol{v}}$ are the same by the theorem. ■

Corollary 3.6. *Using the expression for $\boldsymbol{v}$ or $\tilde{\boldsymbol{v}}$ we are k order precise in time, where k is the order of the Runge-Kutta method.*

Proof $\boldsymbol{v}$ and $\tilde{\boldsymbol{v}}$ are the same. $\boldsymbol{v}$ consists only of a Runge-Kutta method with a good chosen function. So the order in time of $\boldsymbol{v}$ and $\tilde{\boldsymbol{v}}$ is the same as the method order. ■

3.2.3 Runge-Kutta based affine projection

We will repeat the work in section 3.2.2 but where the projection is an affine operator. We use the notation of 1.4.1.

The scheme for the projection at every evaluation of the speed is

$$\tilde{\boldsymbol{v}}_{n+1} = \tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \tilde{\boldsymbol{k}}_i - P\left(\tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \tilde{\boldsymbol{k}}_i, \tilde{\boldsymbol{T}}_0\right), \tag{3.34a}$$

$$\tilde{\boldsymbol{k}}_i = \Delta t f\left(\tilde{\boldsymbol{v}}_n + \sum_{j=1}^{i-1} a_{ij} \tilde{\boldsymbol{k}}_j - P\left(\tilde{\boldsymbol{v}}_n + \sum_{j=1}^{i-1} a_{ij} \tilde{\boldsymbol{k}}_j, \tilde{\boldsymbol{T}}_i\right)\right). \tag{3.34b}$$

We project at every estimation of speed. Making by construction every speed divergence free.

Scheme for analytical expression for pressure:

$$\boldsymbol{v}_{n+1} = \boldsymbol{v}_n + \sum_{i=1}^{s} b_i \boldsymbol{k}_i, \tag{3.35a}$$

$$\boldsymbol{k}_i = \Delta t f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij} \boldsymbol{k}_j\right) - \Delta t P\left(f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij} \boldsymbol{k}_j\right), \boldsymbol{T}_i\right). \tag{3.35b}$$

The following theorem shows that the two schemes are the same and give the relation between T and $\tilde{T}$.

Theorem 3.7. *If $\boldsymbol{v}_n = \tilde{\boldsymbol{v}}_n$ and $\boldsymbol{v}_n$ are divergence free then $\boldsymbol{v}_{n+1} = \tilde{\boldsymbol{v}}_{n+1}$. With the relation for the translation:*

For $0 < i \leq s$

$$\tilde{\boldsymbol{T}}_i = \sum_{j=1}^{i-1} a_{ij} \Delta t \boldsymbol{T}_j, \tag{3.36}$$

$$\tilde{\boldsymbol{T}}_0 = \sum_{i=1}^{s} b_i \Delta t \boldsymbol{T}_i. \tag{3.37}$$

Proof We begin to prove the following lemma.

Lemma 3.8.

$$\boldsymbol{k}_i = \tilde{\boldsymbol{k}}_i - P(\tilde{\boldsymbol{k}}_i, \Delta t \boldsymbol{T}_i) \tag{3.38}$$

Proof By induction on i. We begin with $i = 1$:

$$\boldsymbol{k}_1 = \Delta t f\left(\boldsymbol{v}_n\right) - \Delta t P\left(f\left(\boldsymbol{v}_n\right), \boldsymbol{T}_1\right), \tag{3.39}$$
$$\tilde{\boldsymbol{k}}_1 = \Delta t f\left(\boldsymbol{v}_n\right), \tag{3.40}$$
$$\boldsymbol{k}_1 = \Delta t f\left(\boldsymbol{v}_n\right) - \Delta t P\left(f\left(\boldsymbol{v}_n\right), \boldsymbol{T}_1\right) = \tilde{\boldsymbol{k}}_1 - P\left(\tilde{\boldsymbol{k}}_1, \Delta t \boldsymbol{T}_1\right). \tag{3.41}$$

By induction hypothesis we are true for smaller i:

$$\begin{aligned}
\boldsymbol{k}_i &= \Delta t f\left(\boldsymbol{v}_n + \sum_{j=1}^{n} a_{ij}\boldsymbol{k}_j\right) - \Delta t P\left(f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\boldsymbol{k}_j\right), \boldsymbol{T}_i\right) \\
&= \Delta t f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\left(\tilde{\boldsymbol{k}}_j - P\left(\tilde{\boldsymbol{k}}_j, \Delta t \boldsymbol{T}_j\right)\right)\right) - \Delta t P\left(f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\left(\tilde{\boldsymbol{k}}_j - P\left(\tilde{\boldsymbol{k}}_j, \Delta t \boldsymbol{T}_j\right)\right)\right), \boldsymbol{T}_i\right) \\
\tilde{\boldsymbol{k}}_i &= \Delta t f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\tilde{\boldsymbol{k}}_j - P\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\tilde{\boldsymbol{k}}_j, \tilde{\boldsymbol{T}}_i\right)\right).
\end{aligned}$$

Using the linearity of P and $P\boldsymbol{v}_n = 0$ and defining $\hat{\boldsymbol{T}}_j^i$

$$\begin{aligned}
\tilde{\boldsymbol{k}}_i &= \Delta t f\left(\boldsymbol{v}_n + \sum_{j=1}^{i-1} a_{ij}\left(\tilde{\boldsymbol{k}}_j - P\left(\tilde{\boldsymbol{k}}_j, \hat{\boldsymbol{T}}_j^i\right)\right)\right), \\
\tilde{\boldsymbol{T}}_i &= \sum_{j}^{i-1} a_{ij}\hat{\boldsymbol{T}}_j^i, \\
\hat{\boldsymbol{T}}_j^i &= \Delta t \boldsymbol{T}_j, \\
\boldsymbol{k}_i &= \tilde{\boldsymbol{k}}_i - P\left(\tilde{\boldsymbol{k}}_i, \Delta t \boldsymbol{T}_j\right).
\end{aligned}$$

This proves the relation and gives the relation

$$\tilde{\boldsymbol{T}}_i = \sum_{j}^{i-1} a_{ij}\Delta t \boldsymbol{T}_j. \tag{3.42}$$

■

We now write the expression for $\boldsymbol{v}_{n+1}$ and $\tilde{\boldsymbol{v}}_{n+1}$ using the lemma.

$$\begin{aligned}
\boldsymbol{v}_{n+1} &= \boldsymbol{v}_n + \sum_{i=1}^{s} b_i \boldsymbol{k}_i \\
&= \boldsymbol{v}_n + \sum_{i=1}^{s} b_i \left(\tilde{\boldsymbol{k}}_i - P\left(\tilde{\boldsymbol{k}}_i, \Delta t \boldsymbol{T}_i \right) \right), \\
\tilde{\boldsymbol{v}}_{n+1} &= \tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \tilde{\boldsymbol{k}}_i - P\left(\tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \tilde{\boldsymbol{k}}_i, \tilde{\boldsymbol{T}}_0 \right) \\
&= \tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \left(\tilde{\boldsymbol{k}}_i - P\left(\tilde{\boldsymbol{k}}_i, \hat{\boldsymbol{T}}_i^0 \right) \right), \\
\tilde{\boldsymbol{T}}_0 &= \sum_{i=1}^{s} b_i \hat{\boldsymbol{T}}_i^0, \\
\tilde{\boldsymbol{T}}_0 &= \sum_{i=1}^{s} b_i \Delta t \boldsymbol{T}_i.
\end{aligned}$$

Using that $\boldsymbol{v}_n = \tilde{\boldsymbol{v}}_n$

$$\boldsymbol{v}_{n+1} = \tilde{\boldsymbol{v}}_{n+1}.$$

■

Theorem 3.9. *For the implicit Runge-Kutta case, where the sum on $i-1$ goes now to s and a_{ij} is invertible, if $\boldsymbol{v}_n = \tilde{\boldsymbol{v}}_n$ and $\boldsymbol{v}_n$ are divergence free then $\boldsymbol{v}_{n+1} = \tilde{\boldsymbol{v}}_{n+1}$, and the translation is given by*

$$\tilde{\boldsymbol{T}}_i = \Delta t \sum_j a_{ij} \boldsymbol{T}_j, \tag{3.43}$$

$$\tilde{\boldsymbol{T}}_0 = \Delta t \sum_j b_j \boldsymbol{T}_j. \tag{3.44}$$

Proof We begin to prove the following lemma.

Lemma 3.10. *After relabeling j noting $\boldsymbol{k}_i^j$ the j solution of $\boldsymbol{k}_i$. We have for every j*

$$\boldsymbol{k}_i^j = \tilde{\boldsymbol{k}}_i^j + P(\tilde{\boldsymbol{k}}_i^j, \Delta t \boldsymbol{T}_i). \tag{3.45}$$

This doesn't say that we have existence and uniqueness of the solution only that if we have a set of solution for $\boldsymbol{k}$ we have a bijection to solution of $\tilde{\boldsymbol{k}}$

Proof Because the sum goes to s and not $i-1$ we cannot use induction. We now will use vector notation for $\boldsymbol{k}$.

We construct k and $\tilde{k}$ vectors with the vector $\boldsymbol{k}_i$ and $\tilde{\boldsymbol{k}}_i$

$$k = \begin{pmatrix} \boldsymbol{k}_1 \\ \vdots \\ \boldsymbol{k}_s \end{pmatrix}, \tag{3.46}$$

$$\tilde{k} = \begin{pmatrix} \tilde{\boldsymbol{k}}_1 \\ \vdots \\ \tilde{\boldsymbol{k}}_s \end{pmatrix}. \tag{3.47}$$

We form $\hat{L}$ the linear part of the projection from L the linear part from the projection P:

$$\hat{L} = 1_s \bigotimes L = \begin{pmatrix} L & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & L \end{pmatrix}. \tag{3.48}$$

We have the following property:

$$\hat{L}\hat{v}_n = 0. \tag{3.49}$$

This is true because $\hat{v}_n$ is divergence free. $\hat{L}$ is the linear part and consist to make the projection with 0 boundary conditions.

τ_i the translation vector from $\boldsymbol{T}_i$ the translation part of P, τ is the sum of all individual translation:

$$\tau_i = \begin{pmatrix} \boldsymbol{0} \\ \vdots \\ \boldsymbol{T}_i \\ \vdots \\ \boldsymbol{0} \end{pmatrix}, \tag{3.50}$$

$$\tau = \sum_i \tau_i = \begin{pmatrix} \boldsymbol{T}_1 \\ \vdots \\ \boldsymbol{T}_s \end{pmatrix}. \tag{3.51}$$

Same for $\tilde{\tau}_i$ with $\boldsymbol{T}_i$ changed with $\tilde{\boldsymbol{T}}_i$.

We now form the function $\hat{f}$ from f with:

$$\hat{f} = 1_s \bigotimes f = \begin{pmatrix} f & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & f \end{pmatrix}. \tag{3.52}$$

We now define $\hat{v}_n$:

$$\hat{v}_n = \begin{pmatrix} \boldsymbol{v}_n \\ \vdots \\ \boldsymbol{v}_n \end{pmatrix}. \tag{3.53}$$

We now define $\hat{A}$ with the following Kronecker product:

$$\hat{A} = \begin{pmatrix} a_{11} & \dots & a_{1s} \\ \vdots & \ddots & \vdots \\ a_{s1} & \dots & a_{ss} \end{pmatrix} \bigotimes 1 = A \bigotimes 1. \tag{3.54}$$

$\hat{A}$ and $\hat{L}$ are commutative, because of the mixed-product property

$$\hat{A}\hat{L} = \left(A \bigotimes 1\right)\left(1_s \bigotimes L\right) = A \bigotimes L, \tag{3.55}$$

$$\hat{L}\hat{A} = \left(1_s \bigotimes L\right)\left(A \bigotimes 1\right) = A \bigotimes L. \tag{3.56}$$

$$k = (1 - \hat{L})\Delta t\hat{f}(\hat{v}_n + \hat{A}k) + \Delta t\tau, \tag{3.57}$$

$$\tilde{k} = \Delta t\hat{f}((1 - \hat{L})(\hat{v}_n + \hat{A}k) + \tilde{\tau}). \tag{3.58}$$

We begin to define 2 functions:

$$k = g(k), \tag{3.59}$$

$$\tilde{k} = \tilde{g}(\tilde{k}). \tag{3.60}$$

$$g(x) = (1 - \hat{L})\Delta t\hat{f}(\hat{v}_n + \hat{A}x) + \Delta t\tau, \tag{3.61}$$

$$\tilde{g}(x) = \Delta t\hat{f}((1 - \hat{L})(\hat{v}_n + \hat{A}x) + \tilde{\tau}). \tag{3.62}$$

k and $\tilde{k}$ are fixed points of function g and $\tilde{g}$.
We now calculate using that L is linear and $L\hat{v} = 0$:

$$(1 - \hat{L})\tilde{g}(x) = (1 - \hat{L})\Delta t\hat{f}((1 - \hat{L})(\hat{v}_n + \hat{A}x) + \tilde{\tau}) = (1 - \hat{L})\Delta t\hat{f}(\hat{v}_n + \hat{A}(1 - \hat{L})x + \tilde{\tau}) \tag{3.63}$$

$$= (1 - \hat{L})\Delta t\hat{f}(\hat{v}_n + \hat{A}((1 - \hat{L})x + \hat{A}^{-1}\tilde{\tau})). \tag{3.64}$$

By definition of g we have:

$$= g((1 - \hat{L})x + \hat{A}^{-1}\tilde{\tau}) - \Delta t\tau. \tag{3.65}$$

We now substitute $\tilde{k}$ in x

$$(1 - \hat{L})\tilde{g}(\tilde{k}) = g((1 - \hat{L})\tilde{k} + \hat{A}^{-1}\tilde{\tau}) - \Delta t\tau. \tag{3.66}$$

$\tilde{k}$ is a fixed point of $\tilde{g}$

$$(1 - \hat{L})\tilde{k} = g((1 - \hat{L})\tilde{k} + \hat{A}^{-1}\tilde{\tau}) - \Delta t\tau, \tag{3.67}$$

$$(1 - \hat{L})\tilde{k} + \Delta t\tau = g((1 - \hat{L})\tilde{k} + \hat{A}^{-1}\tilde{\tau}). \tag{3.68}$$

If

$$\tilde{\tau} = \Delta t\hat{A}\tau. \tag{3.69}$$

Then

$$(1 - \hat{L})\tilde{k} + \Delta t\tau = g((1 - \hat{L})\tilde{k} + \Delta t\tau). \tag{3.70}$$

$(1 - \hat{L})\tilde{k} + \Delta t\tau$ is a fixed point of g. But fixed point of g is k.
After relabeling of solution, this directly gives the result with k^j the j-th solution:

$$k^j = (1 - \hat{L})\tilde{k}^j + \Delta t\tau. \tag{3.71}$$

This is exactly the vector version of what we wanted to prove. ■

The rest is exactly the same than in the first case. To not overload the proof with notation, we drop the j label to indicate which solution we take.

We now write the expression for $\boldsymbol{v}_{n+1}$ and $\tilde{\boldsymbol{v}}_{n+1}$ using the lemma.

$$\begin{aligned}
\boldsymbol{v}_{n+1} &= \boldsymbol{v}_n + \sum_{i=1}^{s} b_i \boldsymbol{k}_i \\
&= \boldsymbol{v}_n + \sum_{i=1}^{s} b_i \left((1+L)\, \tilde{\boldsymbol{k}}_i + \Delta t \boldsymbol{T}_i \right), \\
\tilde{\boldsymbol{v}}_{n+1} &= \tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \tilde{\boldsymbol{k}}_i - L \left(\tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \tilde{\boldsymbol{k}}_i \right) + \tilde{\boldsymbol{T}}_0 \\
&= \tilde{\boldsymbol{v}}_n + \sum_{i=1}^{s} b_i \,(1-L)\, \tilde{\boldsymbol{k}}_i + \tilde{\boldsymbol{T}}_0, \\
\tilde{\boldsymbol{T}}_0 &= \Delta t \sum_{i=1}^{s} b_i \boldsymbol{T}_i.
\end{aligned}$$

Using that $\boldsymbol{v}_n = \tilde{\boldsymbol{v}}_n$

$$\boldsymbol{v}_{n+1} = \tilde{\boldsymbol{v}}_{n+1}.$$

■

3.3 Spatial discretization

3.3.1 Non-staggered grid

On a non-staggered grid we put the variables like in figure 3.1. The problem with a non-staggered grid is that obvious centered difference discretization lead to wrong divergence free vector.

We define the derivative of a scalar by:

$$\partial_x a_i = \frac{a_{i+1} - a_{i-1}}{\Delta x}. \tag{3.72}$$

The divergence discretization is then given by (notation for 2d, but for 3d it's the same):

$$\boldsymbol{\nabla} \cdot \boldsymbol{v}_{i,j} = \frac{v_{xi+1,j} - v_{xi-1,j}}{\Delta x} + \frac{v_{yi+1,j} - v_{yi-1,j}}{\Delta y}. \tag{3.73}$$

The problem can be seen in figure 3.2 is that the divergence at the red point only uses values at blue points and not the point at the center (red point). This leads to non-physical solutions that are divergence free.

In figure 3.3 we have a solution that is numerically divergence free. But this solution has high frequency oscillations and we expect that it is not divergence free.

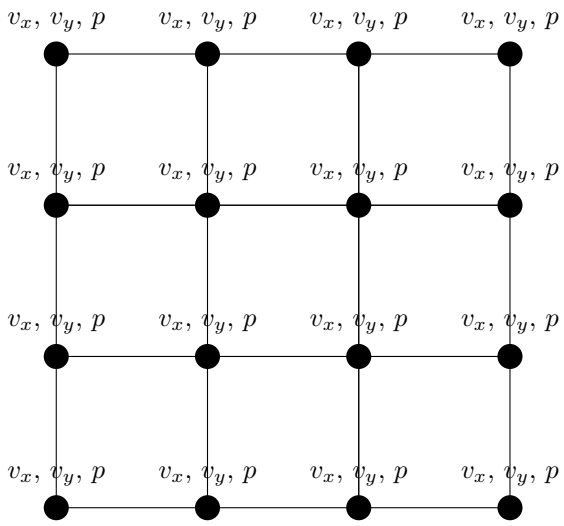


Figure 3.1: Position of variables for a Non-Staggered grid

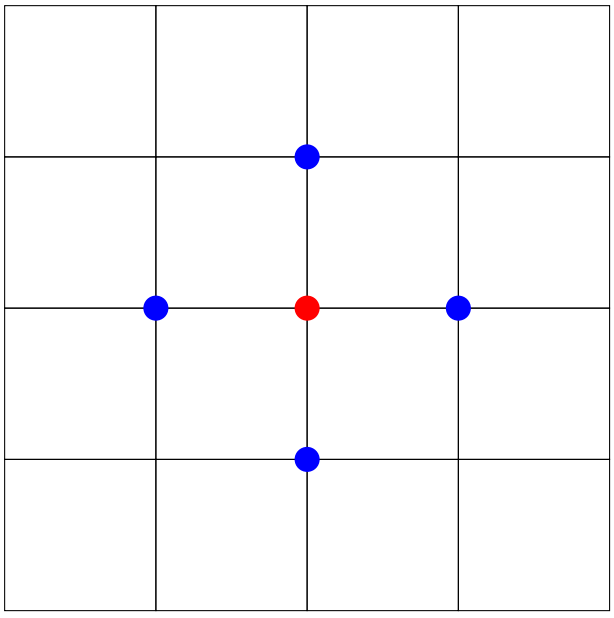


Figure 3.2: Values used to calculate the divergence at the red point are in blue

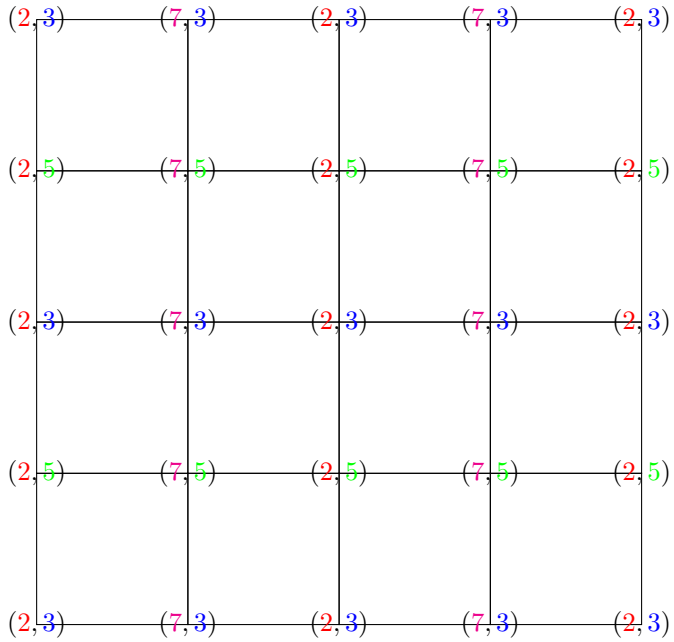


Figure 3.3: Speeds that are numerically divergence free but shouldn't

The problem with this kind of solution is that it is very stable. The projection operator P will be 0, and does not correct this kind of speed. We cannot correct it in general for every choice of f. Note that all the work done in section 3.2 is true. In particular the theorem will be true and will correctly project in a divergence free space. It is only that our divergence free space contains solutions that are not wanted. This solution is in the kernel of the discrete projetion operator.

We can recall one of the formulations of the so called Murphy's law, "If anything can go wrong, it will."

We can correct this problem with using a forward or backward expression. But this kind of expression is only first order accurate. Another solution is to use another grid.

3.3.2 Staggered grid

On a staggered grid we put the variables in the positions shown in figure 3.4.

This can seem to be strange to have different components of speed at different places. But it makes discretization more easy. To label variables, we label by cell and then take the lower left component (cf. figure 3.5).

We now define the divergence and the gradient

$$\boldsymbol{\nabla} p_{ij} = \begin{pmatrix} \frac{p_{i,j} - p_{i-1,j}}{\Delta x} \\ \frac{p_{i,j} - p_{i,j-1}}{\Delta y} \end{pmatrix}, \tag{3.74}$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{v}_{i,j} = \frac{v_{xi+1,j} - v_{xi,j}}{\Delta x} + \frac{v_{y_{i,j+1}} - v_{y_{i,j}}}{\Delta y}. \tag{3.75}$$

These two expressions are geometrically very clear.

In figure 3.6, the gradient at red speed components uses components at blue points. In contrast to the at case with non staggered grid we cannot create a none constant

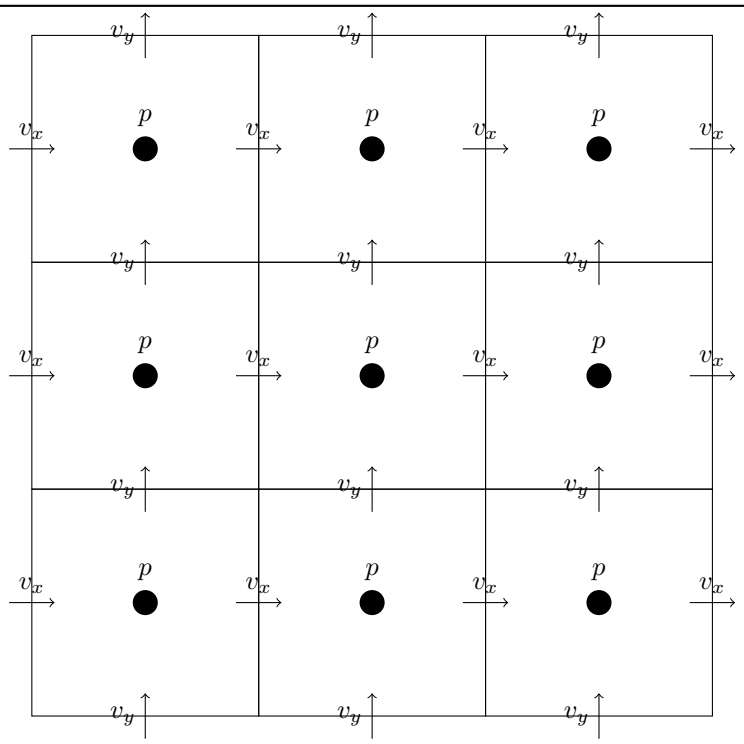


Figure 3.4: Position of variable on a staggered grid

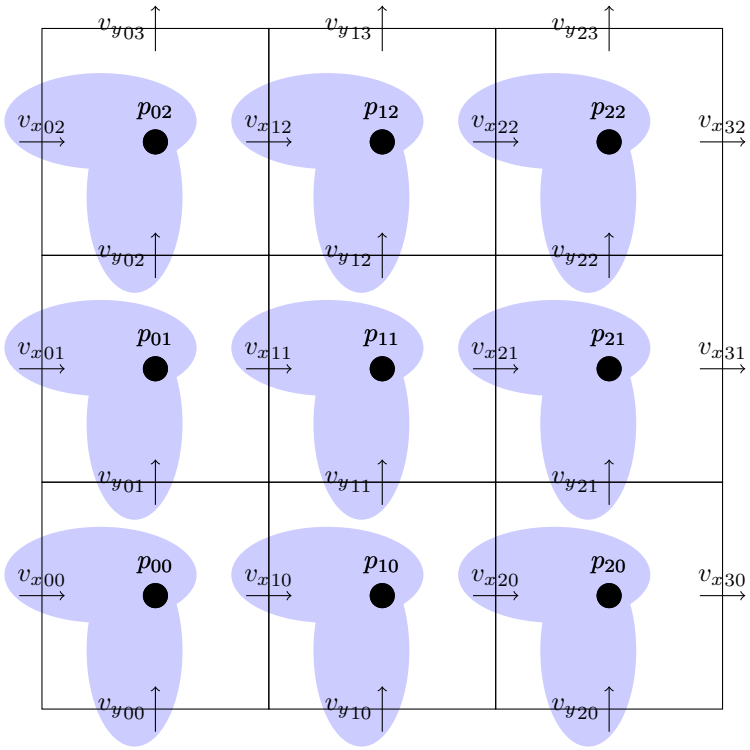


Figure 3.5: Labeling on staggered grid

vector with 0 gradient.

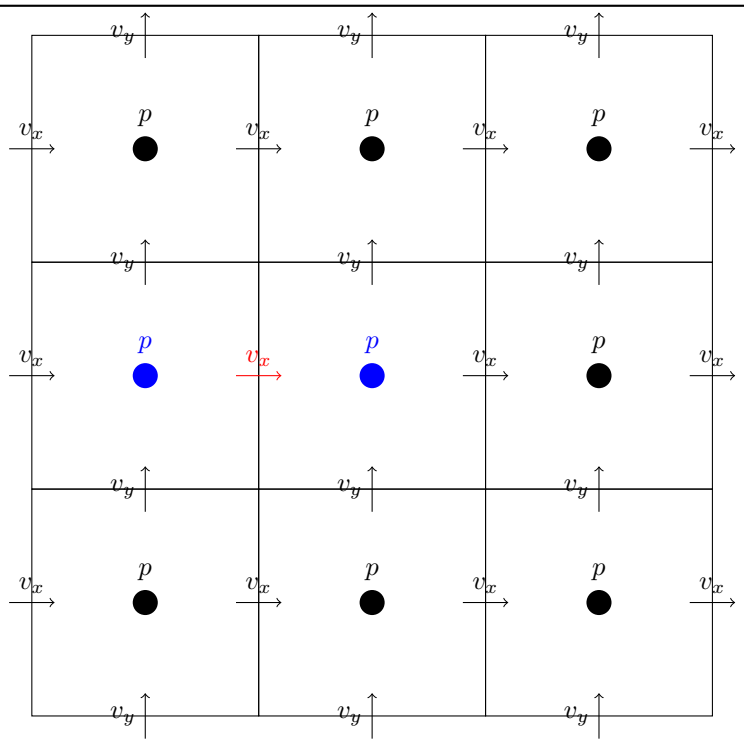


Figure 3.6: Points used to calculate the red gradient component are in blue

The divergence is given by the sum of inflow around a cell. In figure 3.7, the divergence at the red point uses the components at the blue point.

We cannot have the same situation than with non staggered grid, where a nonphysical vector is discretely divergence free.

Discretization

We now look at different possible schemes for the staggered grid discretization.

Gradient The gradient is discretized with centered differences,

$$\boldsymbol{\nabla} p_{i,j} = \begin{pmatrix} \frac{p_{i-1,j} - p_{i,j}}{\Delta x} \\ \frac{p_{i,j-1} - p_{i,j}}{\Delta y} \end{pmatrix}. \tag{3.76}$$

Centered differences are second order accurate in contrast of forward or backward difference.

Divergence The divergence is also discretized with centered differences,

$$\boldsymbol{\nabla} \cdot \boldsymbol{v}_{i,j} = \frac{v_{xi,j} - v_{xi-1,j}}{\Delta x} + \frac{v_{y_{i,j}} - v_{y_{i,j-1}}}{\Delta y}. \tag{3.77}$$

Because of centered differences it is second order accurate.

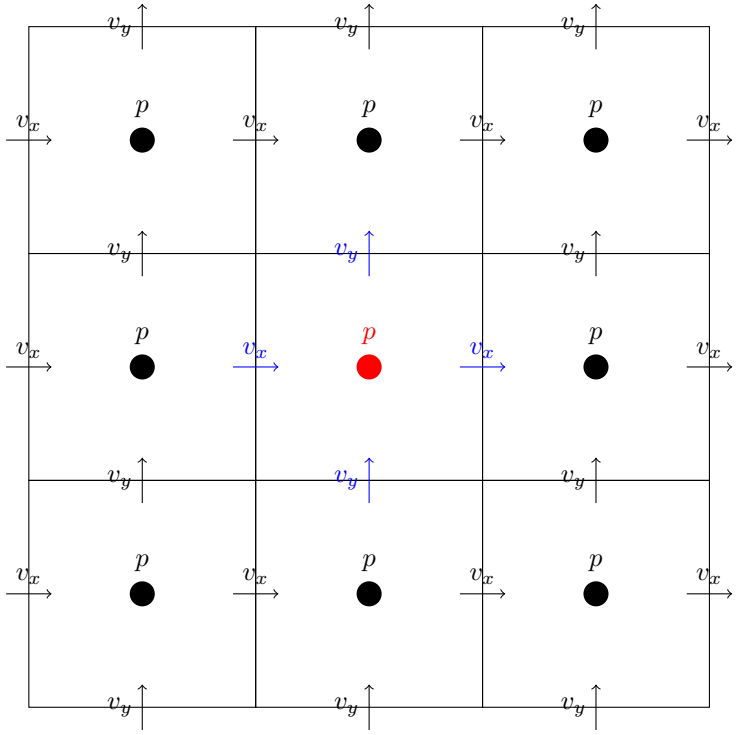


Figure 3.7: Points used to calculate the divergence approximation at the red point are in blue

Laplacian The discretization of the Laplacian is a combination of the application of the gradient and the divergence,

$$\Delta p_{i,j} = \frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{(\Delta x)^2} + \frac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{(\Delta y)^2}. \tag{3.78}$$

This formula is of order 2. We prove it for the one dimensional case by s Taylor expansion around x,

$$\begin{aligned} f(x+\Delta x) - 2f(x) + f(x-\Delta x) &= f(x) + \Delta x f'(x) + (\Delta x)^2 \frac{f''(x)}{2} + (\Delta x)^3 \frac{f'''(x)}{6} + O((\Delta x)^4) - 2f(x) \\ &\quad + f(x) - \Delta x f'(x) + (\Delta x)^2 \frac{f''(x)}{2} - (\Delta x)^3 \frac{f'''(x)}{6} \\ &= (\Delta x)^2 f''(x) + O((\Delta x)^4), \\ f''(x) &= \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{(\Delta x)^2} + O((\Delta x)^2). \end{aligned}$$

Vector Laplacian For the vector Laplacian, we use the same scheme as in section 3.3.2 but for every component of the speed. The staggered grid does not pose a problem because we do not have mixed terms with and without derivatives.

Convection The convection term is the more difficult term to treat, because it is non-linear and uses mixed terms. We need to consider two types of terms. Let $i \neq j$ and $\boldsymbol{k}$ be an integer vector of the position on the grid and $\boldsymbol{e}_i$ the i-th unit vector. We

first consider as i-th component

$$\left(v^i \partial_i v^i\right)^i_{\boldsymbol{k}}. \tag{3.79}$$

We can discretize this in two ways.

Centered difference

$$v^i{}_{\boldsymbol{k}} \frac{v^i{}_{\boldsymbol{k}+\boldsymbol{e}_i} - v^i{}_{\boldsymbol{k}-\boldsymbol{e}_i}}{2\Delta x_i}. \tag{3.80}$$

Upwind difference We use the same method, but instead of using a centered difference, we use an upwind difference:

if $v^i{}_{\boldsymbol{k}} < 0$

$$\left(v^i \partial_i v^i\right)_{\boldsymbol{k}} = v^i{}_{\boldsymbol{k}} \frac{v^i{}_{\boldsymbol{k}+\boldsymbol{e}_i} - v^i{}_{\boldsymbol{k}}}{\Delta x_i}, \tag{3.81}$$

if $v^i{}_{\boldsymbol{k}} > 0$

$$\left(v^i \partial_i v^i\right)_{\boldsymbol{k}} = v^i{}_{\boldsymbol{k}} \frac{v^i{}_{\boldsymbol{k}} - v^i{}_{\boldsymbol{k}-\boldsymbol{e}_i}}{\Delta x_i}, \tag{3.82}$$

if $v^i{}_{\boldsymbol{k}} = 0$

$$\left(v^i \partial_i v^i\right)_{\boldsymbol{k}} = 0. \tag{3.83}$$

Upwind is considered more stable (can be proved easily in one dimension for Burger's equation [5]) but more diffusive.

We now consider

$$\left(v^j \partial_j v^i\right)^i_{\boldsymbol{k}}. \tag{3.84}$$

Remark 3.3.1. *Contrary to the other case, we need to have the speed at a position not known for v^j. This arises because of the staggered grid. For this, we can interpolate the known speed of neighboring cells taking for example the average of the 4 neighbors (cf. figure 3.8). We will use as notation v^{ji} for the speed component j at position for the component i.*

We can discretize this again in two ways:

Centred difference

$$\left(v^j \partial_j v^i\right)^i_{\boldsymbol{k}} = v^{ji}{}_{\boldsymbol{k}} \frac{v^i{}_{\boldsymbol{k}+\boldsymbol{e}_j} - v^i{}_{\boldsymbol{k}-\boldsymbol{e}_j}}{2\Delta x_j}. \tag{3.85}$$

Upwind difference We use the same method, but instead to use the centered difference, we use an upwind difference:

if $v^{ji}{}_{\boldsymbol{k}} < 0$

$$\left(v^j \partial_j v^i\right)^i_{\boldsymbol{k}} = v^{ji}{}_{\boldsymbol{k}} \frac{v^i{}_{\boldsymbol{k}+\boldsymbol{e}_j} - v^i{}_{\boldsymbol{k}}}{\Delta x_j}, \tag{3.86}$$

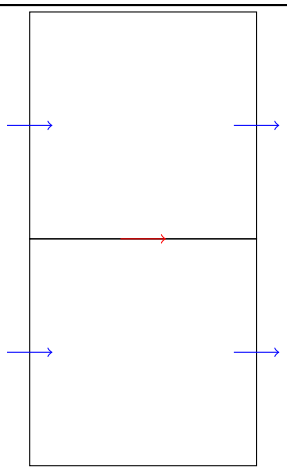


Figure 3.8: We need the value of the red arrow. For this we can interpolate from the blue arrows.

if ${v^{ji}}_{\boldsymbol{k}} > 0$

$$\left(v^j \partial_j v^i\right)^i_{\boldsymbol{k}} = {v^{ji}}_{\boldsymbol{k}} \frac{{v^i}_{\boldsymbol{k}} - {v^i}_{\boldsymbol{k}-\boldsymbol{e}_j}}{\Delta x_j}, \tag{3.87}$$

if ${v^{ji}}_{\boldsymbol{k}} = 0$

$$\left(v^j \partial_j v^i\right)^i_{\boldsymbol{k}} = 0. \tag{3.88}$$

3.4 Pseudo code

The pseudo code is not very complicated in principle. We need to put all mathematical discretizations and pieces together.

We will use the following order of operation:

1. Calculate optimal time step.
2. Use a Runge-Kutta algorithm. We now have two choices:
   (a) We project the acceleration. For this we do the following for every Runge-Kutta step:
      i. We set the speed to the wanted boundary conditions.
      ii. We calculate the acceleration from discretization of the function noted $f(\boldsymbol{v})$ in this section.
      iii. We project the given acceleration.
   (b) We project the speed. For this we apply the Runge-Kutta step as in section 3.2.2:
      i. We set the speed to the wanted boundary conditions.
      ii. We obtain a speed estimation from $f(\boldsymbol{v})$.
      iii. We project the speed estimation to be divergence free.

The principal difference of the two methods are:

- If we project on speed we are always divergence free.
- If we project on force, we need to have initial speeds that are divergence free. Numerical error can accumulate. We can eliminate them with a projection of speed.
- Boundary conditions need to be transformed as in section 3.2.3.

We will in the next section show the implementation of the principal parts.

3.4.1 Pseudo code semantic and notation.

The notation of pseudo code is inspired by **C-like** language.

Variables are introduced when used by giving their name.

Assignments are done with $\leftarrow$ which indicate that the left variable take the right value.

Boolean operations are done with standard mathematical notation ($=, \neq, <, >, \leq, \geq$).

Arithmetic operations uses the standard mathematical operations ($+, -, \cdot, \frac{x}{x}$).

The following special control flow operations are used:

Loop: An infinite loop, is only stopped by a **break** or **return** statement. In the case where we use this kind of loop we cannot have infinite loop, because we use it in case where all cell need to be treated once but in a specific order. When all cell are treated the loop stop.

For all: A loop which takes all elements of a container. **for all** expressions are normally good candidates for parallelism.

Container

The following container will be used:

Grid: Consists of all cells. Used in expression like **for all**($it \in Grid$).

Unordered Stack: A container where elements can be added and accessed in a **for all**. The order is not important. After usage, the stack is not used any more. Duplicate elements are inserted more than once. A new unordered stack is created with the function call **newunorderedstack**().

Set: A set is a container which is like a mathematical set, where elements can be added and each element accessed. The difference with a stack is that in a set duplicate elements are stored once and the set elements are not discarded after usage. A new set is created with the function call **newset**().

Data type

The first data type is the *cell*. The *cell* are elements of the grid container.
The principal methods defined on a cell are:

SpeedGet(i) **:** Which gets the speed in direction i. We use a staggered grid and consider speed components of a given cell to be below left.

SpeedGet() **:** Which gets the speed vector.

SpeedSet(i, val) **:** Sets the speed component i with value val.

SpeedSet(val) **:** Sets the speed vector to value val.

AccelerationGet(i) **:** Gets the acceleration into direction i.

AccelerationGet() **:** Gets the acceleration vector.

AccelerationSet(i, val) **:** Sets the acceleration component i to value val.

AccelerationSet(val) **:** Sets the acceleration vector to value val.

PressureGet() **:** Gets the pressure at a given cell.

PressureSet(val) **:** Sets the pressure at a given cell to val.

GetIsFluid() **:** Indicates if the cell is fluid or not.

GetIsAir() : Indicates if the cell is air or not. This is the opposite of **GetIsFluid**().

FluidSet() : Sets the cell to fluid.

AirSet() : Sets the cell to air.

GetNeighbour(dir, s) **:** Gets the neighboring cell in the direction dir positive or negative depending on the sign of s.

We define the following methods for vectors:

Get(i) Gets the value of index i of the vector.

Set(i, val) Sets the value of index i of the vector to value val.

The following global variables are used:

viscosity **:** Is the viscosity used.

dt **:** The time step used.

t **:** The time.

h **:** The spacing.

3.4.2 Acceleration evaluation

To apply gravity we simply add a constant acceleration cf. algorithm 2 and algorithm 3. We test if we need to apply gravity with a code like algorithm 1 which applies gravity to speed components that touch the fluid.

To apply the viscosity we need to calculate the Laplacian and a condition of when to apply. The same condition as for gravity works. The application is in algorithm 4, the calculation of the Laplacian is in algorithm 5.

For the convection the application is in algorithm 6, the condition is similar to the one for gravity. The convection speed derivative with upwind is done in algorithm 7 and algorithm 8.

Algorithm 1 Algorithm which tests if gravity acceleration needs to be applied

```
1: function NeedToApplyGravity(cell)
2:     if cell.GetIsFluid() then
3:         return true ▷ We are a fluid cell, so the speed component in y is bordering the fluid cell.
4:     end if
5:     cell2 ← cell.GetNeighbour(2, −1) ▷ We now consider the cell below (the gravity is applied
   in the y direction), to see if we are bordering a fluid cell.
6:     if not(cell2.IsValid()) then
7:         return false                    ▷ If the neighbor doesn't exist we are not near fluid.
8:     end if
9:     if cell2.GetIsFluid() then
10:        return true                     ▷ If the neighbor is fluid, we are bordering a fluid cell.
11:    end if
12:    return false
13: end function
```

Algorithm 2 Algorithm which applies the gravity acceleration.

```
1: function ApplyGravity(cell)
2:     if NeedToApplyGravity(cell) then
3:         cell.AccelerationSet(2, cell.AccelerationGet(2) − g) ▷ Add gravity to the acceleration.
4:     end if
5: end function
```

Algorithm 3 Algorithm which applies the gravity acceleration for every cell.

```
1: procedure ApplyGravityCell
2:     for all cell ∈ Grid do
3:         ApplyGravity(cell)
4:     end for
5: end procedure
```

3.4.3 Projection of acceleration.

The projection of acceleration is done in the following steps:

1. Calculate the divergence of the acceleration (a is the acceleration).

$$b = \nabla \cdot a. \tag{3.89}$$

Algorithm 4 Algorithm which applies the viscosity acceleration.

```
1: procedure ApplyViscosity
2:     for all cell ∈ Grid do
3:         if GetIfApplyViscosity(it) then
4:             it.AccelerationSet(it.AccelerationGet() + GetLaplacianSpeed(it) · viscosity) ▷ Add the viscosity acceleration to the acceleration.
5:         end if
6:     end for
7: end procedure
```

Algorithm 5 Algorithm which applies the vector Laplacian. The discretization explanation details are in section 3.3.2

```
1: function GetLaplacianSpeed(cell)
2:     for i = 1 to dim do ▷ A vector Laplacian consist to apply the Laplacian to all component.
3:         temp ← 0
4:         for j = 1 to dim do ▷ Calculate the Laplacian
5:             temp ← temp − 2cell.SpeedGet(i) · (h.Get(j))^2
6:             temp ← temp + cell.GetNeighbour(j, 1).SpeedGet(i) · (h.Get(j))^2
7:             temp ← temp + cell.GetNeighbour(j, −1).SpeedGet(i) · (h.Get(j))^2
8:         end for
9:         ret.Set(i, temp) ▷ Store the component in the vector ret.
10:    end for
11:    return ret ▷ return the result.
12: end function
```

Algorithm 6 Algorithm to apply the convection.

```
1: function ApplyConvection
2:     for all it ∈ Grid do
3:         for i = 1 to dim do
4:             if GetIfApplyConvection(it, i) then
5:                 it.AccelerationSet(i, it.AccelerationGet(i) + GetConvectionSpeed(it, i)) ▷ We add to the acceleration the convection.
6:             end if
7:         end for
8:     end for
9: end function
```

Algorithm 7 Algorithm which calculates $u_j \partial_j u_i$ with the upwind scheme. The discretization explanation details are in section 3.3.2

```
1: function GetConvectionElement(i, j, cell)
2:     U ← GetSpeedAtBound(cell, i, j)        ▷ Function to get the speed at speed component position i in direction j. This can be done by average.
3:     if U > 0 then                          ▷ We test if we are upwind or not.
4:         U1 ← cell.SpeedGet(i)
5:         cell ← cell.GetNeighbour(j, −1)
6:         U2 ← cell.SpeedGet(i)
7:         U ← U(U1 − U2) · h.Get(j)
8:     else
9:         U1 ← cell.SpeedGet(i)
10:        cell ← cell.GetNeighbour(j, 1)
11:        U2 ← cell.SpeedGet(i)
12:        U ← U(U2 − U1) · h.Get(j)
13:    end if
14:    return U
15: end function
```

Algorithm 8 Algorithm which calculates the convection.

```
1: function GetConvectionSpeed(cell,i)
2:     ret ← 0
3:     for j = 1 to dim do                    ▷ We sum on j of u_j ∂_j u_i.
4:         ret ← ret − GetConvectionElement(i, j, cell)
5:     end for
6:     return ret
7: end function
```

2. Solve the linear system

$$\Delta p = b. \tag{3.90}$$

3. Change the acceleration to

$$a_{new} = a - \nabla p. \tag{3.91}$$

If we have boundary conditions, the b vector needs to be translated. The divergence is calculated by algorithm 9, the gradient by algorithm 10. A function **GetIsInDomain**(*cell*) is used, we can consider it similar to *cell*.**GetIsFluid**(). The pressure is then solved with algorithm 11.

Algorithm 9 Algorithm that calculates the divergence. The discretization explanation details are in section 3.3.2

```
1: function GetDivergence(cell)
2:     ret ← 0
3:     for i = 1 to dim do
4:         ret ← ret − cell.SpeedGet(i) · h.Get(i)
5:         ret ← ret + cell.GetNeighbour(i, 1).SpeedGet(i) · h.Get(i)
6:     end for
7:     return ret
8: end function
```

Algorithm 10 Algorithm that calculates the gradient. The discretization explanation details are in section 3.3.2.

```
1: function GetGradiant(cell, i)
2:     ret ← cell.PressureGet() · h.Get(i)
3:     ret ← ret − cell.GetNeighbour(i, −1).PressureGet() · h.Get(i)
4:     return ret
5: end function
```

Algorithm 11 Algorithm to calculate the correction to acceleration to make the velocity divergence free. We assume a case with 0 Dirichlet boundary conditions.

```
1: procedure ProjectAcceleration
2:     for all it ∈ Grid do        ▷ We calculate the divergence for every cell and store it in a array. it.ID() is a unique numbering of the array for all cell.
3:         if it.GetIsInDomain() then
4:             b[it.ID()] ← GetDivergence(it)
5:         end if
6:     end for
7:     Grid.SolveLinear(b, p)      ▷ We solve the Laplacian system. p is the result and b the second member. The discretization of Laplacian is shown in section 3.3.2.
8:     for all it ∈ Grid do
9:         if it.GetIsInDomain() then    ▷ If we are in the domain for the Laplacian. We copy the result in the pressure.
10:            it.PressureSet(p[it.ID()])
11:        else                          ▷ For Dirichlet boundary conditions, we set pressure to 0.
12:            PressureSet(0)
13:        end if
14:    end for
15:    for all it ∈ Grid do                                  ▷ We correct the acceleration.
16:        for i = 1 to dim do
17:            it.AccelerationSet(i, it.AccelerationGet(i) − GetGradiant(it, i))
18:        end for
19:    end for
20: end procedure
```

3.4.4 Runge-Kutta integration

To implement the Runge-Kutta method, some new elements are needed. We need to have a copy of speed. To do this we add a method **GetArray**(i) which gets the array i. We need to set the default array if no array is given; this is done by the function $SetArraySpeed(i)$. By default if not changed array 0 is used. The function **CalculateAcceleration**() calculates the acceleration using the default array. The only part of the code that uses explicitly array is this one.

The function **CalculateAcceleration**() consists in calculating the acceleration and then do the projection. This is shown in algorithm 12.

The Runge-Kutta algorithm of order 4 is shown in algorithm 13. Depending on the Runge-Kutta scheme chosen the number of copies of speed needed changes.

Algorithm 12 Algorithm which calculates the acceleration.

```
1: procedure CalculateAcceleration
2:     ApplyConvection()
3:     ApplyViscosity()
4:     ApplyGravityCell()
5:     ProjectAcceleration()
6: end procedure
```

3.4.5 Complexity

The complexity is dominated by the 4 calls to **CalculateAcceleration**(). The function **CalculateAcceleration**() then calls the function to calculate acceleration. This function consists for cell where the acceleration apply to calculate a term which depend of the cell itself and his neighbor. The complexity is then linear in the number of cells. The complexity for the projection of acceleration will depend on the complexity to solve the Laplacian which depending on the method used can be linear (multigrid for example is linear). The complexity to calculate the gradient is linear too. The complexity is then dominated by the complexity to solve the Laplacian.

A more technical complexity is the storage of data on the grid. For fixed topology we can use a fixed size array. But for variable topology we need to choose how we store the data. To be robust, our method needs to be able to do dynamic allocation of memory to be able to represent all possible topology. Depending on the choice made the average access time can be constant or $\log(n)$ and the worst case can be n. The method **GetNeighbour**(dir, s) was used instead of index vector manipulation because it allows us to create storage with fast access to neighboring cells.

3.4.6 Parallelism

We remark that we can easily parallelisable the **for all** loop, because we have no dependency:

- The calculation of the acceleration does not modify the speed and only reads there propper acceleration.
- The projection of acceleration when applying the gradient uses the already calculated pressure.

Algorithm 13 Algorithm that integrates with the Runge Kutta method.

procedure RungeKutta
  for all $it \in Grid$ **do**
    it.**AccelerationSet**(0) ▷ We set the acceleration to 0. So that we can calculate the acceleration after.
  end for
  CalculateAcceleration() ▷ This function will add the acceleration to the acceleration array.
  for all $it \in Grid$ **do**
    it.**GetArray**(2).**SpeedSet**(it.**GetArray**(0).**SpeedGet**() + it.**AccelerationGet**() $\cdot$ dt $\cdot$ 0.5) ▷ Array 2 is set to the speed for the next evaluation. We do not use array 0 because we need it.
    it.**GetArray**(1).**SpeedSet**(it.**GetArray**(0).**SpeedGet**()+it.**AccelerationGet**() $\cdot$ $\frac{dt}{6}$) ▷ Array 1 is set to the accumulation of the result.
  end for
  for all $it \in Grid$ **do**
    it.**AccelerationSet**(0)
  end for
  SetArraySpeed(2) ▷ The speed for the next evaluation is in array 2. We set array 2 as default.
  $t \leftarrow t + 0.5 \cdot dt$
  CalculateAcceleration()
  for all $it \in Grid$ **do**
    it.**GetArray**(2).**SpeedSet**(it.**GetArray**(0).**SpeedGet**()+it.**AccelerationGet**() $\cdot$ $\frac{dt}{2}$) ▷ Array 2 is set to the speed for the next evaluation. We do not use array 0 because we need it.
    it.**GetArray**(1).**SpeedSet**(it.**GetArray**(1).**SpeedGet**()+it.**AccelerationGet**() $\cdot$ $\frac{dt}{3}$) ▷ Array 1 is set to the accumulation of the result.
  end for
  for all $it \in Grid$ **do**
    it.**AccelerationSet**(0)
  end for
  CalculateAcceleration()
  for all $it \in Grid$ **do**
    it.**GetArray**(2).**SpeedSet**(it.**GetArray**(0).**SpeedGet**()+it.**AccelerationGet**() $\cdot$ dt)
    it.**GetArray**(1).**SpeedSet**(it.**GetArray**(1).**SpeedGet**()+it.**AccelerationGet**() $\cdot$ $\frac{dt}{3}$)
  end for
  for all $it \in Grid$ **do**
    it.**AccelerationSet**(0)
  end for
  $t \leftarrow t + \frac{dt}{2}$
  CalculateAcceleration()
  for all $it \in Grid$ **do**
    it.**GetArray**(0).**SpeedSet**(it.**GetArray**(1).**SpeedGet**()+it.**AccelerationGet**() $\cdot$ $\frac{dt}{6}$) ▷ Array 0 now contain the new speed.
  end for
  SetArraySpeed(0) ▷ We set the default array again to 0 to be ready for the next time step.
end procedure

- The parallelism to solve the linear system will depend on the method used.

But technically this is only true if the method of manipulation of speed is thread safe.

3.4.7 Conclusion

We have shown a working pseudo code that solves the Navier-Stokes equation for fixed boundary conditions. We have shown that the complexity is dominated by the resolution of a Laplacian and can be linear. To have a real implementation we need to provide some storage related function to access neighbors for example.

This pseudo code is part of the simplified version of the original **C++** code. As linear solver the Umfpack direct solver was used. Other iterative libraries were tested too. The storage methods are constructed around the **C++11** standard library **Unordered Map** container which consist of a hash table.

3.5 Numerical experiments

We will now do some numerical experiments with fixed boundary conditions to show how the code works.

3.5.1 Heat equation

The more complicated term in the Navier-Stokes equation in grid representation is the convection term, because it is non linear,

$$(\boldsymbol{v} \cdot \boldsymbol{\nabla})\boldsymbol{v}. \tag{3.92}$$

We can have this term 0 everywhere, if we ensure that the direction of variation is orthogonal to the direction of speed.

For example, for a two dimensional problem we work with a flow with a speed only in the y direction. And the value depending only on the position on the x axis,

$$\boldsymbol{v} = \begin{pmatrix} 0 \\ f(x,t) \end{pmatrix}. \tag{3.93}$$

The convection term is then:

$$(f(x,t)\partial_y) \begin{pmatrix} 0 \\ f(x,t) \end{pmatrix} = 0. \tag{3.94}$$

The Navier-Stokes equations reduces then to:

$$\partial_y f(x,t) = 0, \tag{3.95}$$

$$\frac{\partial f(x,t)}{\partial t} = F(x,t) + \nu \frac{\partial^2 f(x,t)}{\partial x^2} - \boldsymbol{\nabla} p. \tag{3.96}$$

Equation (3.95) is automatically true because we have no dependence on x. The pressure term role is only to cancel non divergence free parts coming from the force, because the part with the viscosity ν is a linear operator of a divergence free term. For a potential force like gravity, the force can be absorbed in the pressure with a proper

account of boundary conditions. This equation is a heat equation. We consider no external force. We will consider as domain $x = [0, 2L]$ with fixed speed at $x = 0$ and $x = L$, Neumann boundary conditions for pressure at $x = 0$ and $x = L$. Analytically the domain in y expand from $-\infty$ to ∞. Numerically, we use Dirichlet boundary conditions at $y = [0, 2L]$ for pressure.

$$\begin{aligned} v(x = 0, t) &= 1, \\ v(x = 2L, t) &= -1, \\ v(x, 0) &= \begin{cases} 1 & \text{if } x < L, \\ -1 & \text{if } x \geq L. \end{cases} \end{aligned}$$

Analytical solution

$\boldsymbol{\nabla} p$ is everywhere 0 because the linear combination of a divergence free field is divergence free,

$$\frac{\partial f(x,t)}{\partial t} = \nu \frac{\partial^2 f(x,t)}{\partial x^2}. \tag{3.97}$$

We first look for 0 boundary conditions. We will denote this case by f_0. We first use as ansatz that we can separate variables,

$$f_0(x,t) = X(x)T(t). \tag{3.98}$$

The equation then gives

$$\frac{\dot{T}}{\nu T} = \frac{X''}{X}. \tag{3.99}$$

Because the rhs only depends on x and the lhs only depends on t, this needs to equal a constant value $-\lambda$. This gives use two uncoupled equations depending on λ,

$$\dot{T} = -\lambda \nu T, \tag{3.100}$$

$$X'' = -\lambda X. \tag{3.101}$$

The solution for T is

$$T \propto e^{-\lambda \nu t}. \tag{3.102}$$

The solution for X is

$$X \propto \sin(\sqrt{\lambda} x). \tag{3.103}$$

We only have sin because $\sin(0) = 0$. Because we ask $\sin(2L) = 0$

$$0 = \sin(2\sqrt{\lambda} L) = \sin\left(n \frac{\pi}{2L}\right). \tag{3.104}$$

This then gives the condition on λ

$$\sqrt{\lambda} = n \frac{\pi}{2L}. \tag{3.105}$$

A linear combination of this for different n is solution too. We find the good linear combination from the Fourier series of the initial condition $f(0, t)$,

$$f_0(x,t) = \sum_{n=1}^{\infty} C_n \sin\left(\frac{n\pi}{2L} x\right) e^{-\frac{n^2 \pi^2 \nu t}{4L^2}}, \tag{3.106}$$

$$C_n = \frac{1}{L} \int_0^{2L} f(x, 0) \sin\left(\frac{n\pi}{2L} x\right) dx. \tag{3.107}$$

For a non 0 boundary conditions, we need to add a particular solution, so that $f_p(0,0) = 1$ and $f_p(2L,0) = -1$.

$$f_p(x,t) = -\frac{x-L}{L}. \tag{3.108}$$

$f_0(x,0)$ is then given by:

$$f_0(x,0) = f(x,0) - f_p(x,t) = \begin{cases} \frac{x}{L} & \text{if } x < L, \\ \frac{x-L}{L} - 1 & \text{if } x \geq L. \end{cases} \tag{3.109}$$

The general solution is then:

$$f(x,t) = -\frac{x-L}{L} + \sum_{n=1}^{\infty} C_n \sin\left(\frac{n\pi}{2L}x\right) e^{-\frac{n^2\pi^2\nu t}{4L^2}}, \tag{3.110}$$

$$C_n = \frac{1}{L}\int_0^{2L} f(x,0)\sin\left(\frac{n\pi}{2L}x\right)dx. \tag{3.111}$$

This is plotted for $2L = 0.1$ m and $\nu = 1.307 \cdot 10^{-6}$ m^2/s which is the viscosity of water for 10 °C and for $n_{max} = 1'000'000$ in figure 3.9.

Note that the fact that we have truncated the sum can be considered as a slight change of initial condition. Numerically the truncation error is of the level of the numerical precision.

The analytical solution shows that the effect of viscosity is to smooth the speed distribution. The time needed to do that is of 500 s which is a very long time with respect to the distance traveled in this time (500 m). But this does not say that viscosity is not important, because it is effect is inversely proportional to the square of L.

Numerical solution

We now use our numerical program to solve the same problem. The effective problem that we will solve is the heat equation with as solver Runge-Kutta. The other part of the program as convection and projection are done but should have no effect.

The solution for different times t are shown in figures 3.10, 3.11, 3.12 ,3.13. We have an excellent agreement with the analytical results if the timestep used is sufficiently small. For $dt = 2$s for the Euler method and $dt = 3$ s for the Runge-Kutta method we have divergence (result completely wrong). The CFL condition (for convection) is $dt_{max} = hu_{max} = 0.002$ s. The CFL condition for heat equation is $dt_{max} = \frac{h^2}{2\nu} = 1.53$ s. The CFL condition equation are detailed in [4]. Our chosen timestep doesn't respect the CFL condition for convection but convection term is normally 0. Our chosen timestep correctly diverge when we approach the CFL condition for the heat equation.

The solutions with low resolution for different times t are shown in figures 3.14, 3.15, 3.16 ,3.17. As can be seen in the comparatif figure 3.18 better integration methods does not change precision but allows higher timesteps, the reason is because the spatial error is dominating. Smaller spatial steps improve discretization. The error at early time is bigger because of the steep change in the initial condition.

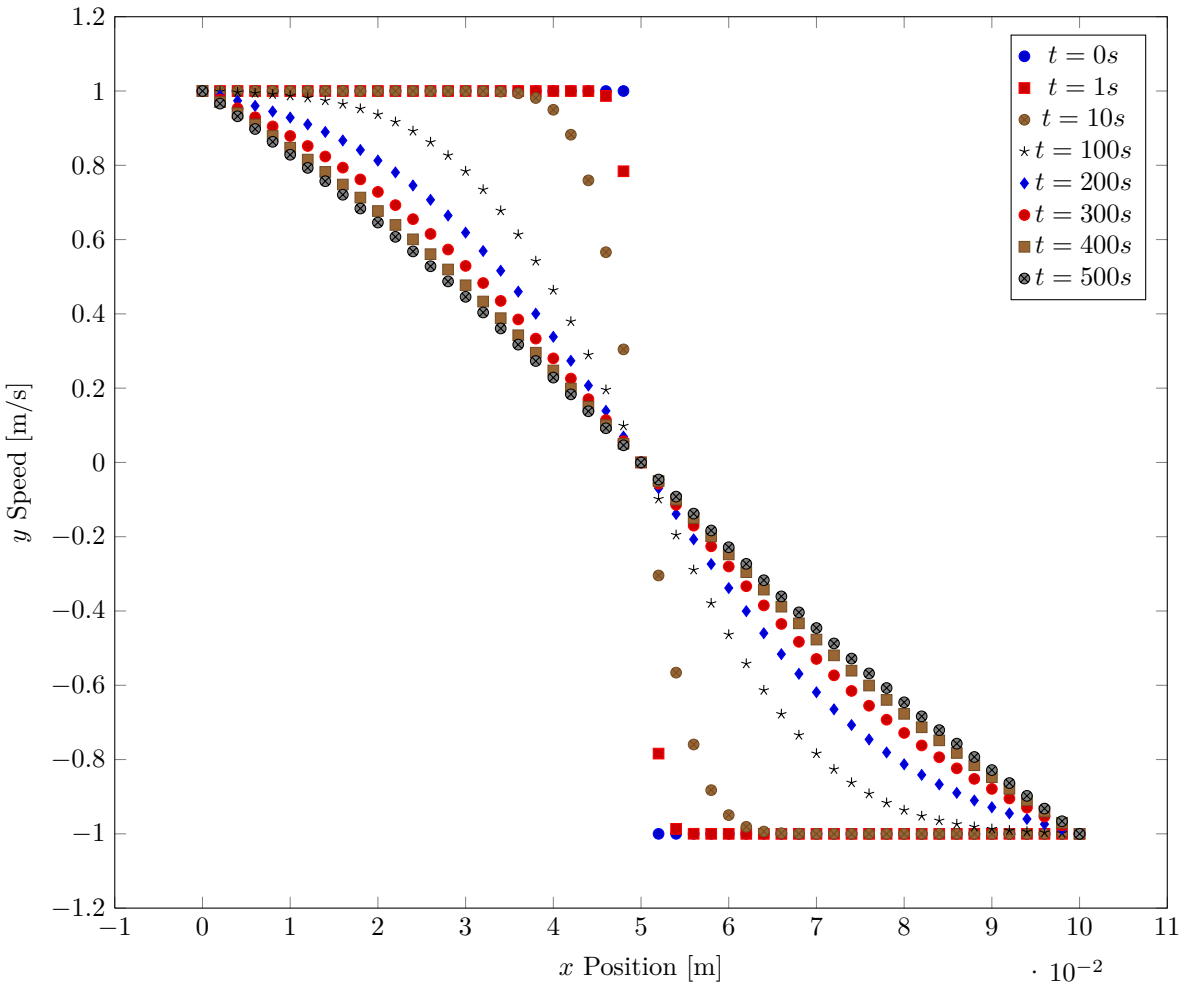


Figure 3.9: Plot for the analytical solution truncated to $n_{max} = 1'000'000$ with $2L = 0.1$ m and $\nu = 1.307 \cdot 10^{-6}$ m^2/s. At the following times.

$t = 0$ s $t = 100$ s $t = 400$ s
$t = 1$ s $t = 200$ s $t = 500$ s
$t = 10$ s $t = 300$ s

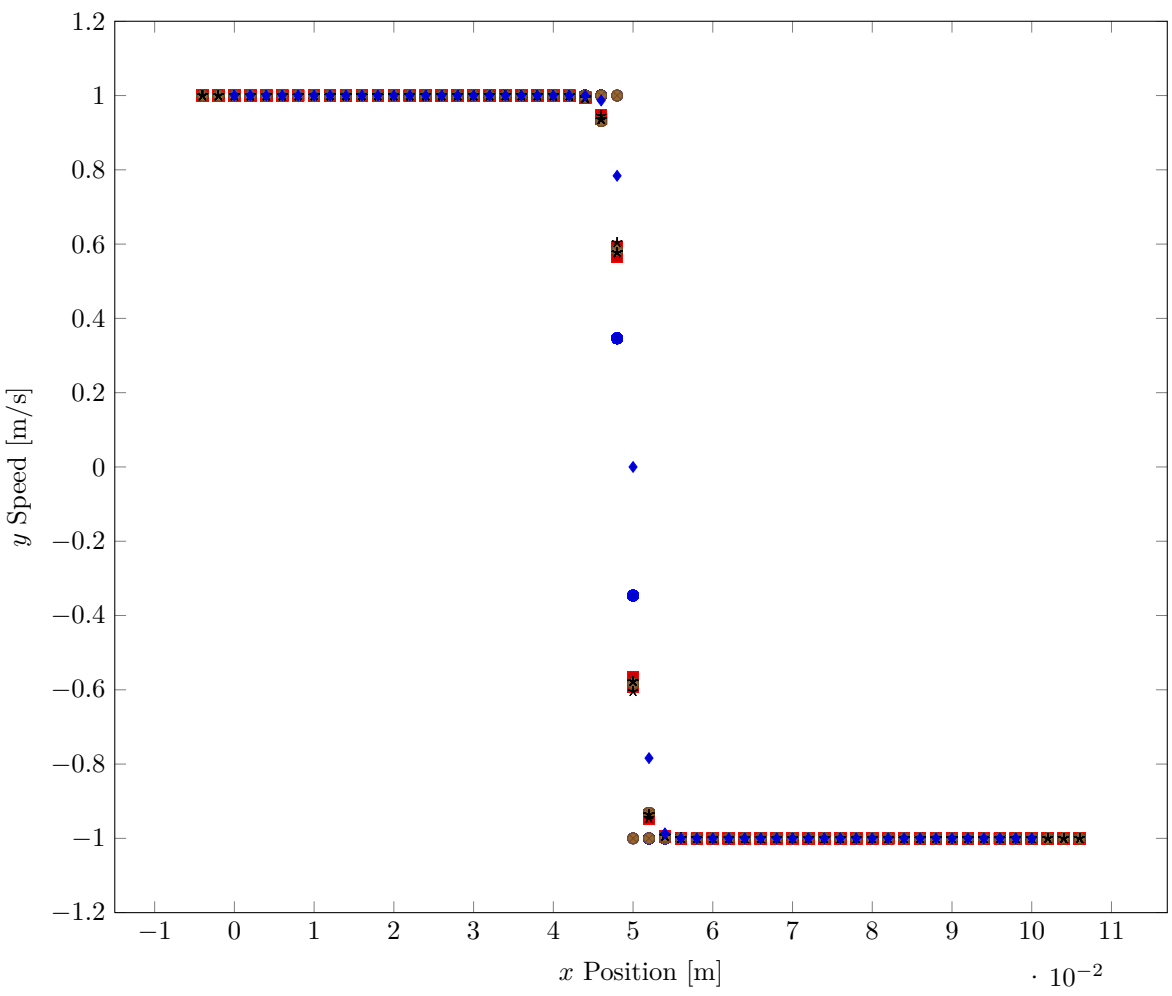


Figure 3.10: Numerical solution for $t = 1$ s, for the following methods:

Euler
$dt = 0.1$ s
$dt = 1$ s

Runge-Kutta
$dt = 0.1$ s
$dt = 1$ s

Analytical

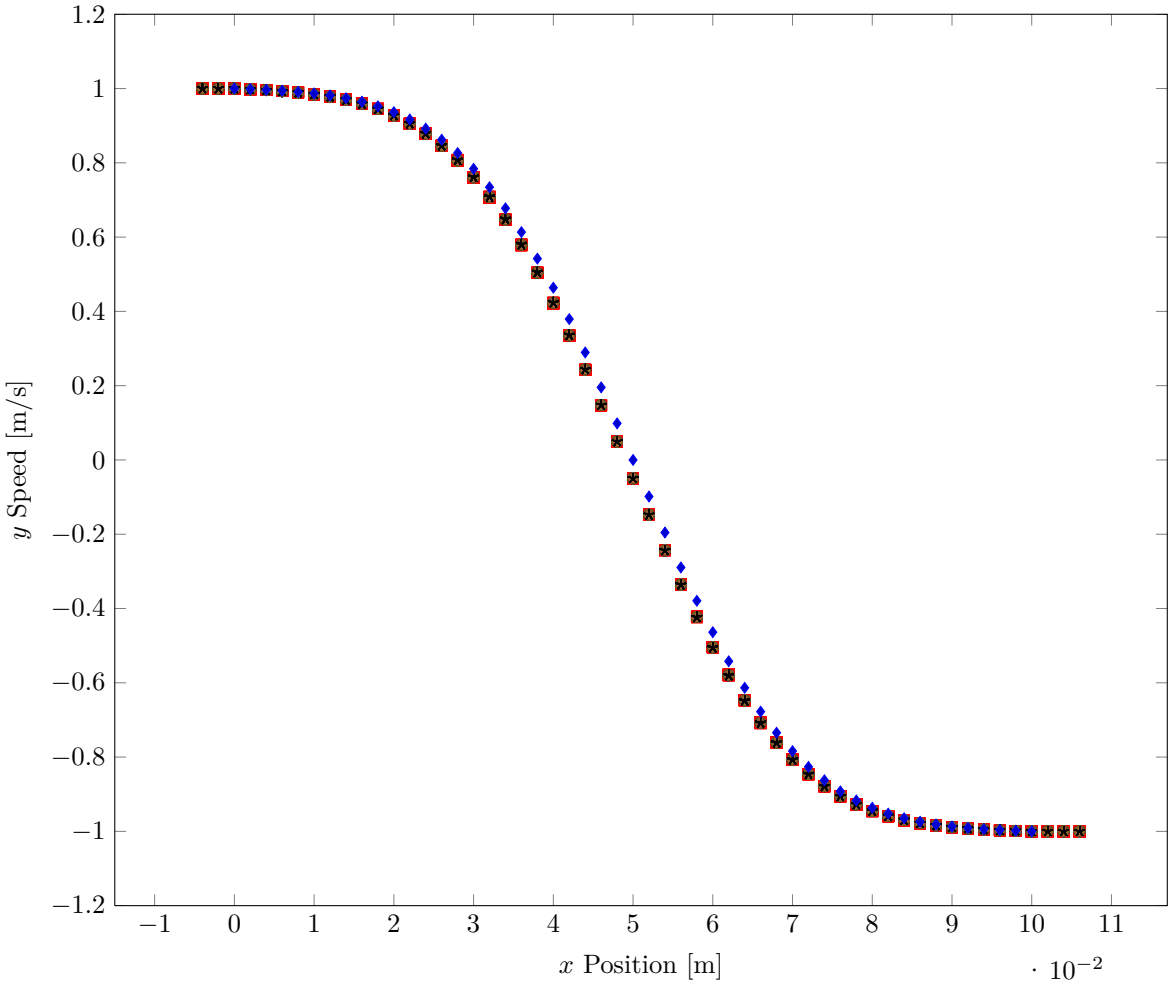


Figure 3.11: Numerical solution for $t = 100$ s, for the following methods:

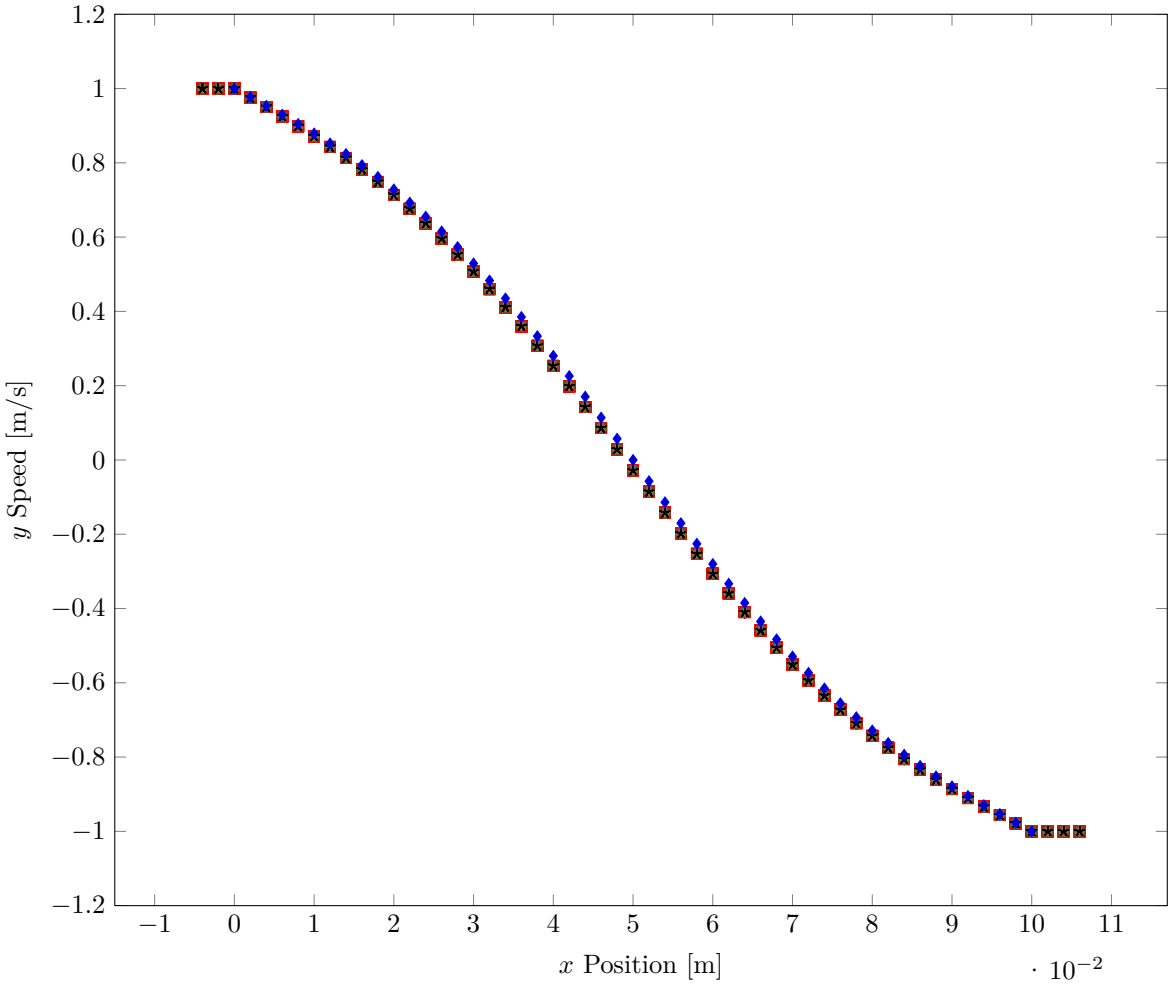


Figure 3.12: Numerical solution for $t = 300$ s, for the following methods:

Euler
$dt = 0.1$ s
$dt = 1$ s

Runge-Kutta
$dt = 0.1$ s
$dt = 1$ s

Analytical

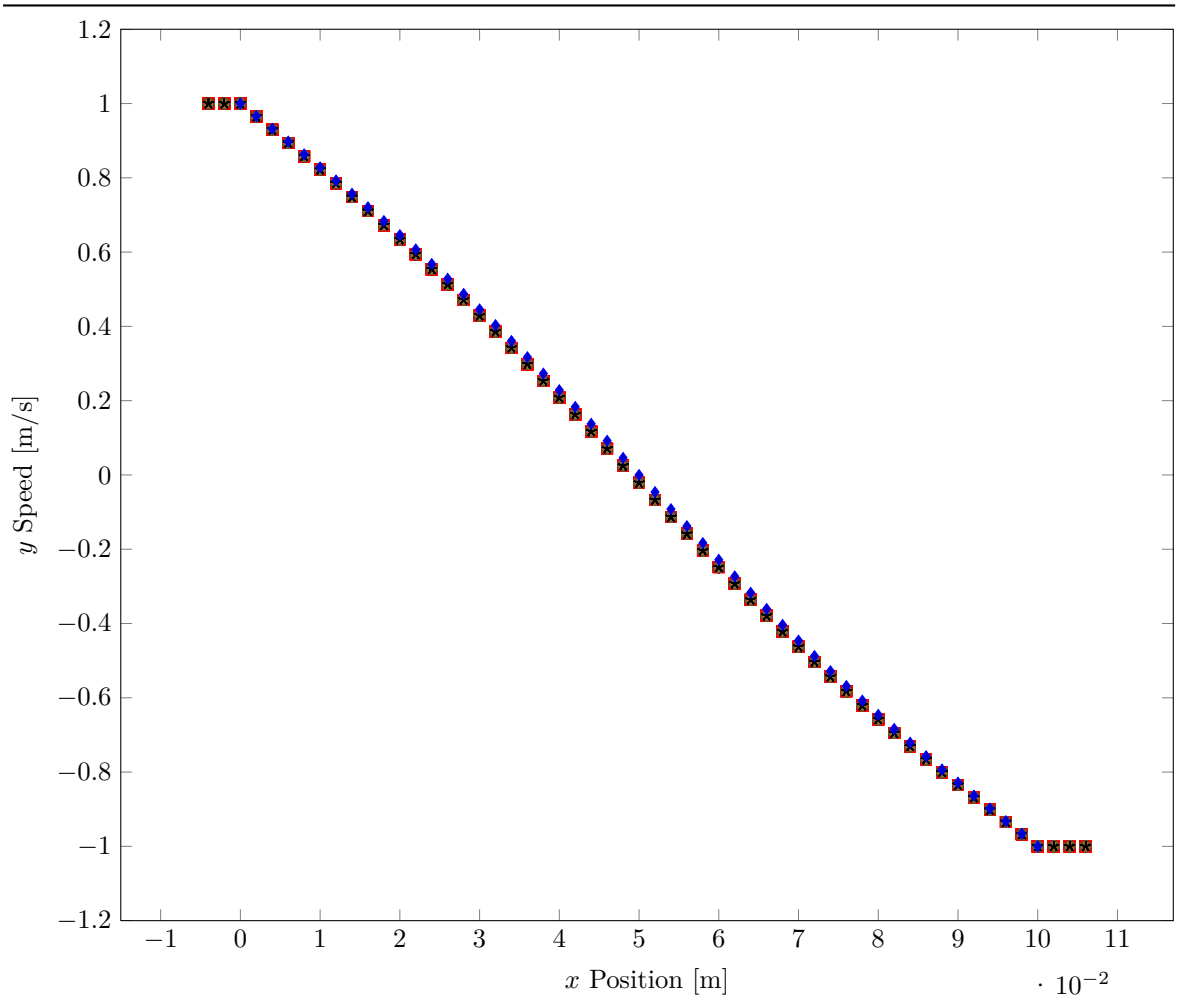
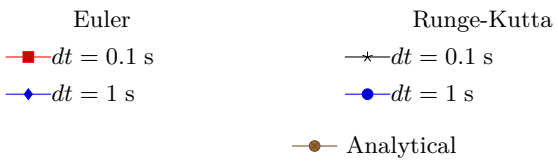


Figure 3.13: Numerical solution for $t = 500$ s, for the following methods:



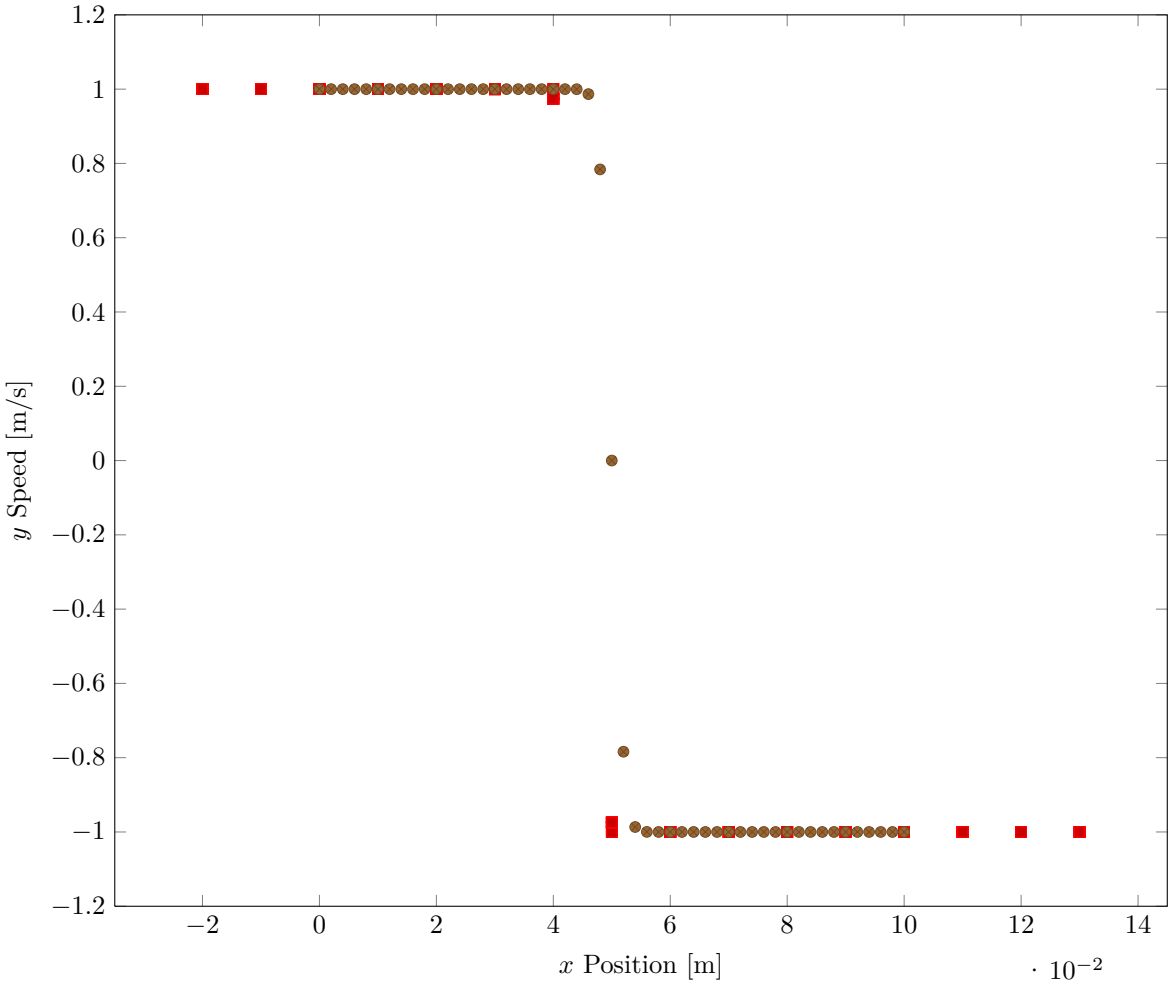


Figure 3.14: Numerical solution for $t = 1$ s. For the following method:

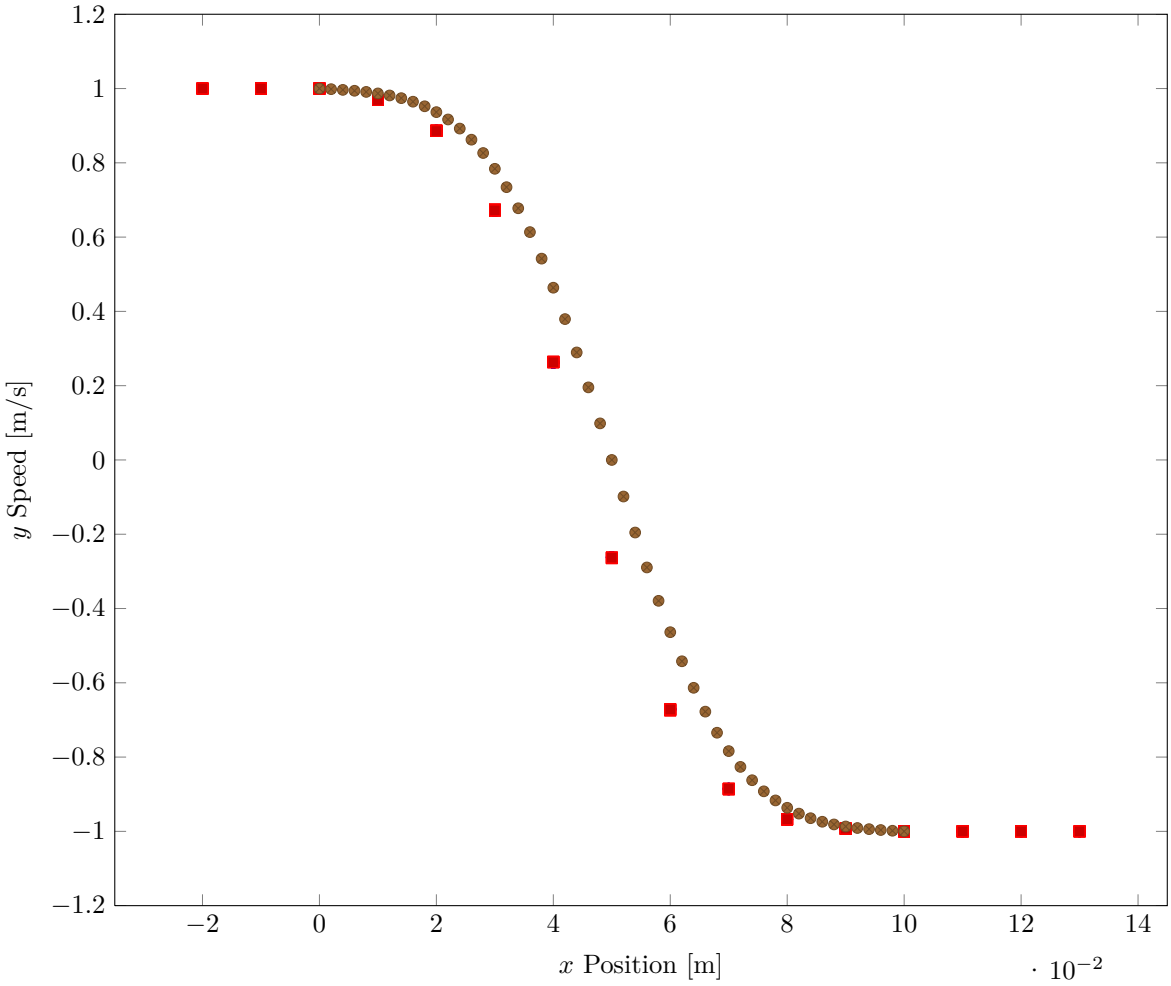


Figure 3.15: Numerical solution for $t = 100$ s. For the following method:

Euler — $dt = 1$ s

Runge-Kutta — $dt = 1$ s

Analytical

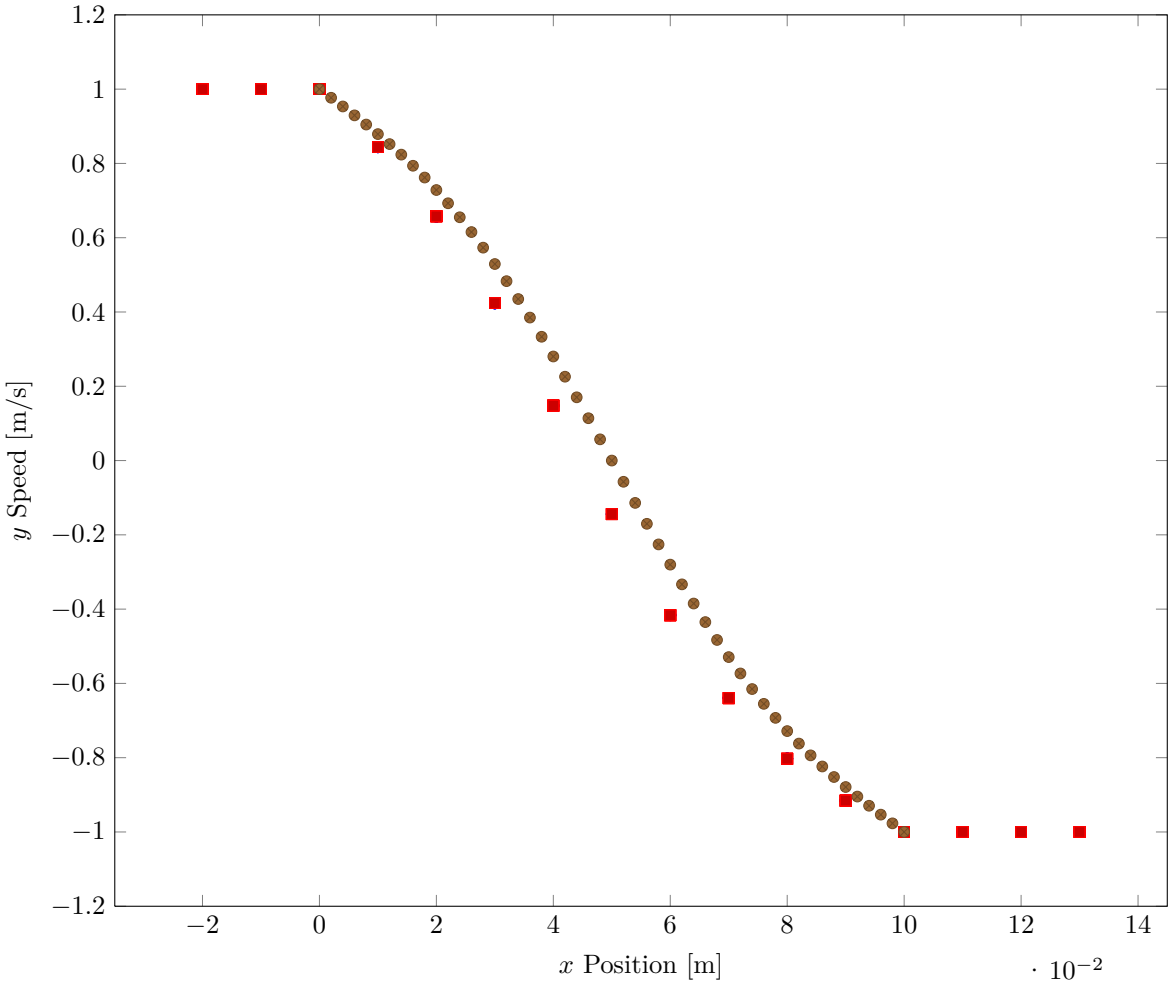


Figure 3.16: Numerical solution for $t = 300$ s, for the following methods:

Euler
$dt = 1$ s

Runge-Kutta
$dt = 1$ s

Analytical

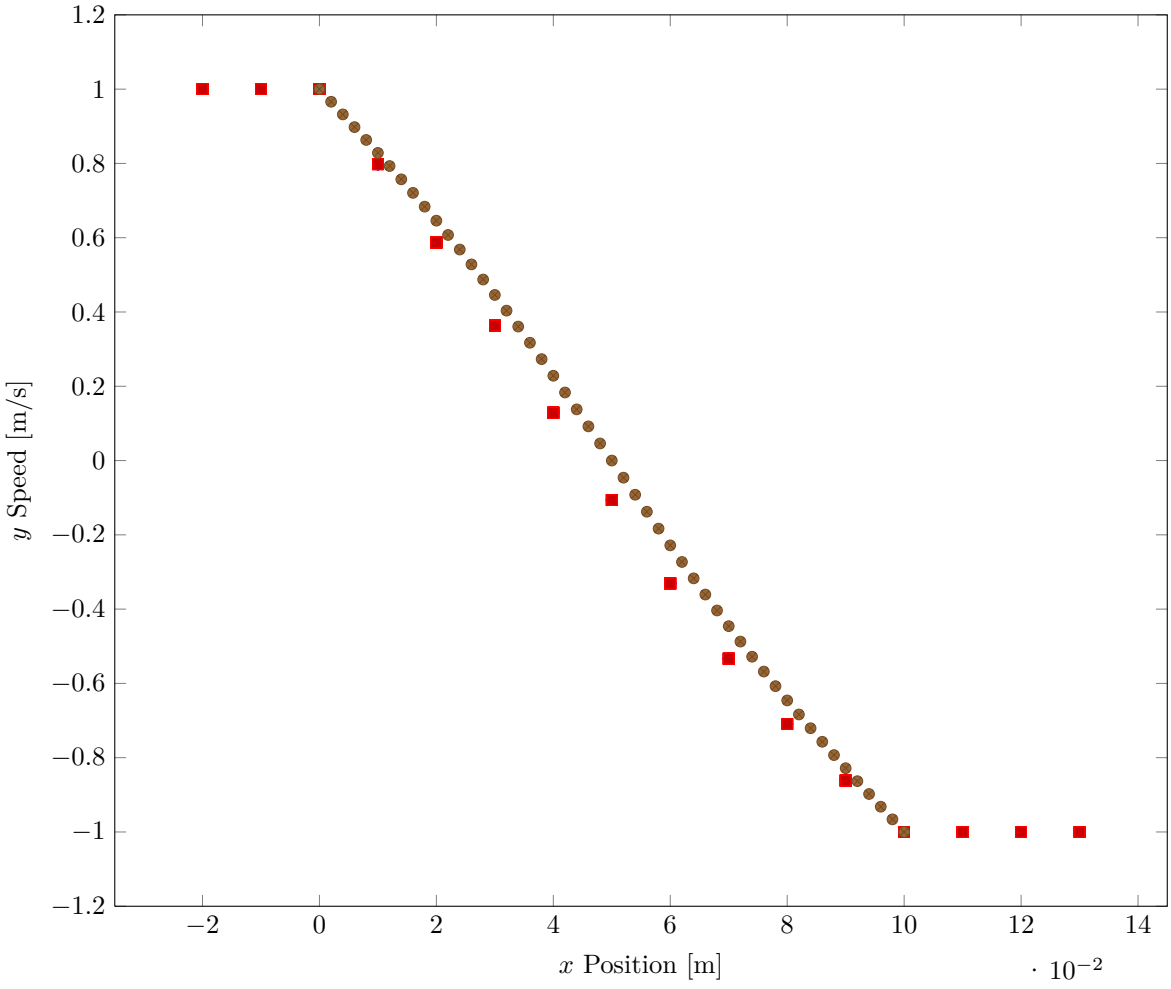


Figure 3.17: Numerical solution for $t = 500$ s, for the following methods:

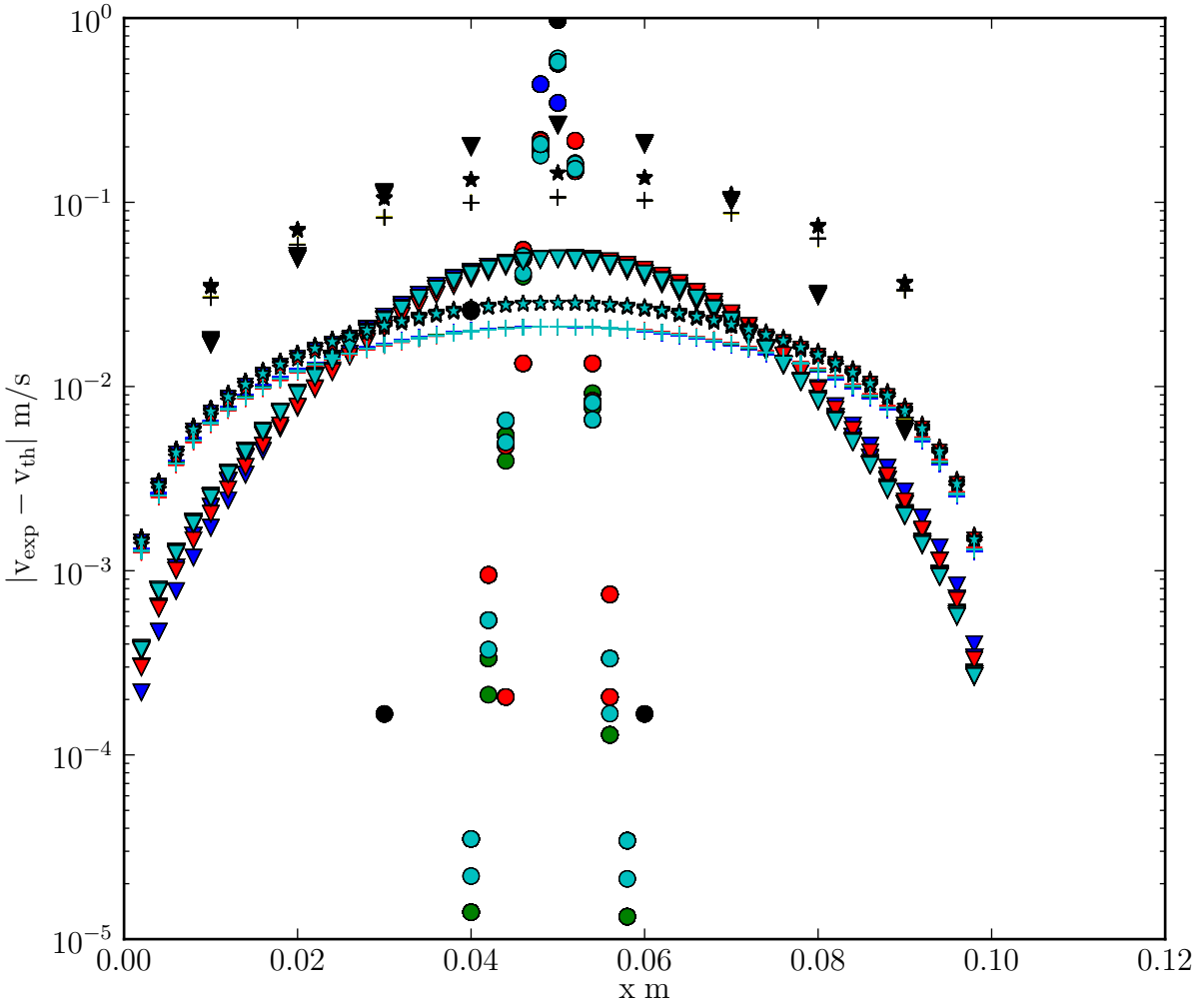


Figure 3.18: Comparison of the solution to the analytical solution. The form of the marker indicate the time:

$t = 1$ s	$t = 300$ s
$t = 100$ s	$t = 500$ s

The color of the marker indicate the method:

Euler	Runge-Kutta
$h = 0.002$	
$dt = 0.1$ s	$dt = 0.1$ s
$dt = 1$ s	$dt = 1$ s
$h = 0.01$	
$dt = 0.1$ s	$dt = 0.1$ s

3.5.2 Oseen vortex decay

Analytical solution

An analytical solution for the Navier-Stokes equation without gravity is given in polar coordinates by,

$$v_\theta = \frac{\Gamma}{2\pi r}\left(1 - e^{-\frac{r^2}{4\nu t}}\right), \tag{3.112}$$

$$v_r = 0. \tag{3.113}$$

This can be proved in polar coordinates in the vorticity version of the Navier-Stokes equation (not used in this work). In the vorticity version of the equation it consists only of a time derivative and a diffusive term. The convective term is 0.

The pressure solution can be found inserting the equations (3.112) into the Navier-Stokes equations and finding ∇p.

We now need to choose a domain and boundary conditions. As domain, we take a circle centered at 0. Because of symmetry, the boundary condition for ∇p is constant on the circle. If not we would not have radial symmetry for the speed. Constant Dirichlet boundary condtions are then a good boundary condition. The choice of the constant will not change the gradient.

Numerical results

This problem is interesting for numerical results because we have a simple analytical solution but in polar coordinates and which is found in vortex formulation, this solution is very far away of the cartesian speed pressure formulation used to derive the Navier-Stokes solver.

We use as domain a disque of radius 1 m. We use a viscosity of 1 m^2/s. We use as boundary conditions for pressure 0 Dirichlet boundary conditions. We use an initial time of $t = 0.01$ because at $t = 0$ the initial conditions have a singularity which will not be fair for the comparison of different grid spacings. N is the radius in cell of the problem. α is the condition so that a particle with the maximun speed of the fluid cannot traverse more of an α fraction of a cell.

For the speed we use the analytical solution (cf. equations (3.112)) outside the domain.

Because the analytical solution is radial, it is better to compare in polar coordinates. For this, we do the following: from the numerical solution on the staggered grid, we average the speed per cell. Then in the postprocessing stage, we calculate the cartesian analytical solution on the staggered grid and average like for the numerical solution. We then transform the numerical and analytical solution to polar coordinates and compare them.

The results are plotted in figures 3.19 and 3.20.

The first interesting result is that we have no better error when we do not diverge with respect to a better time step and better integration method. This is a sign that the error is dominated by the spatial discretization. Only changing the grid spacing changes the precision. But a better method and time step gives a better stability.

We see that the numerical results are not too good and need a finer grid to become better. The difficulty in this numerical problem is that the chosen grid is not appropriate for the problem (cartesian grid for a polar problem).

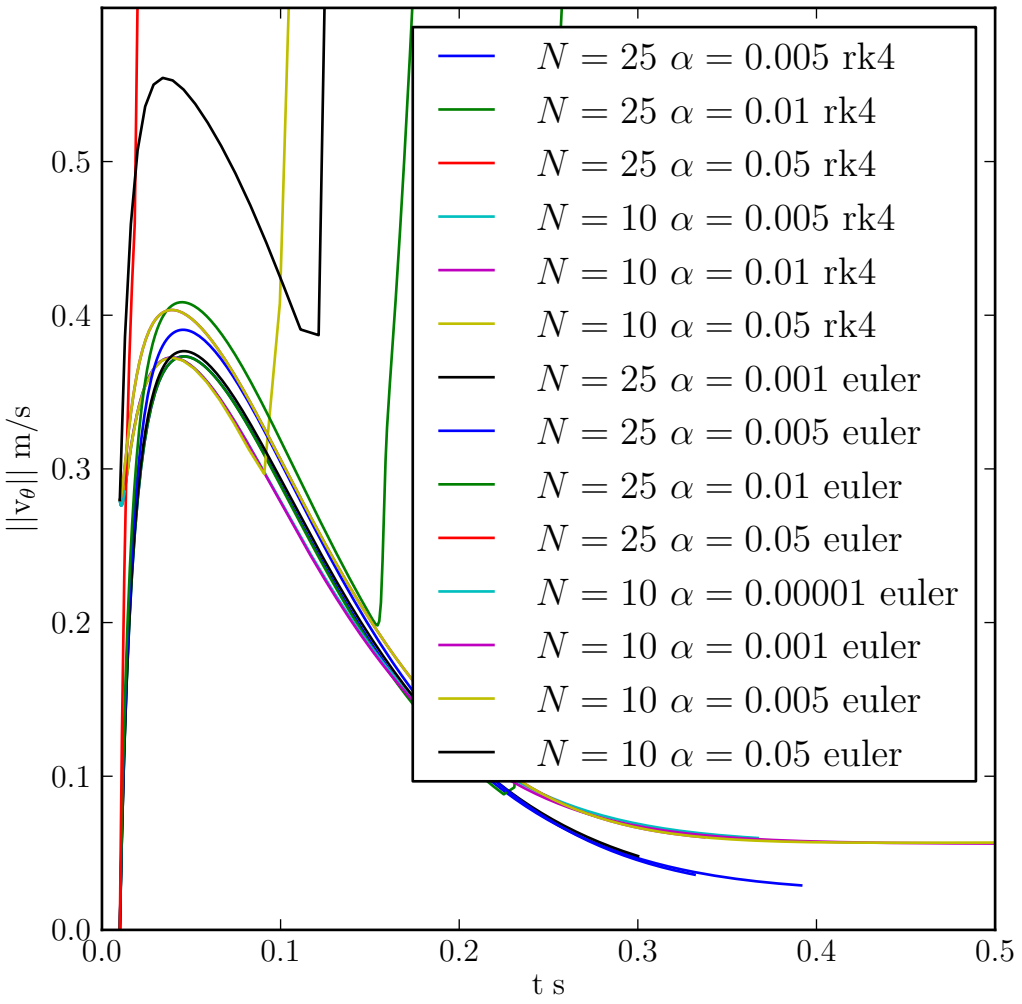


Figure 3.19: Error norm for angular component.

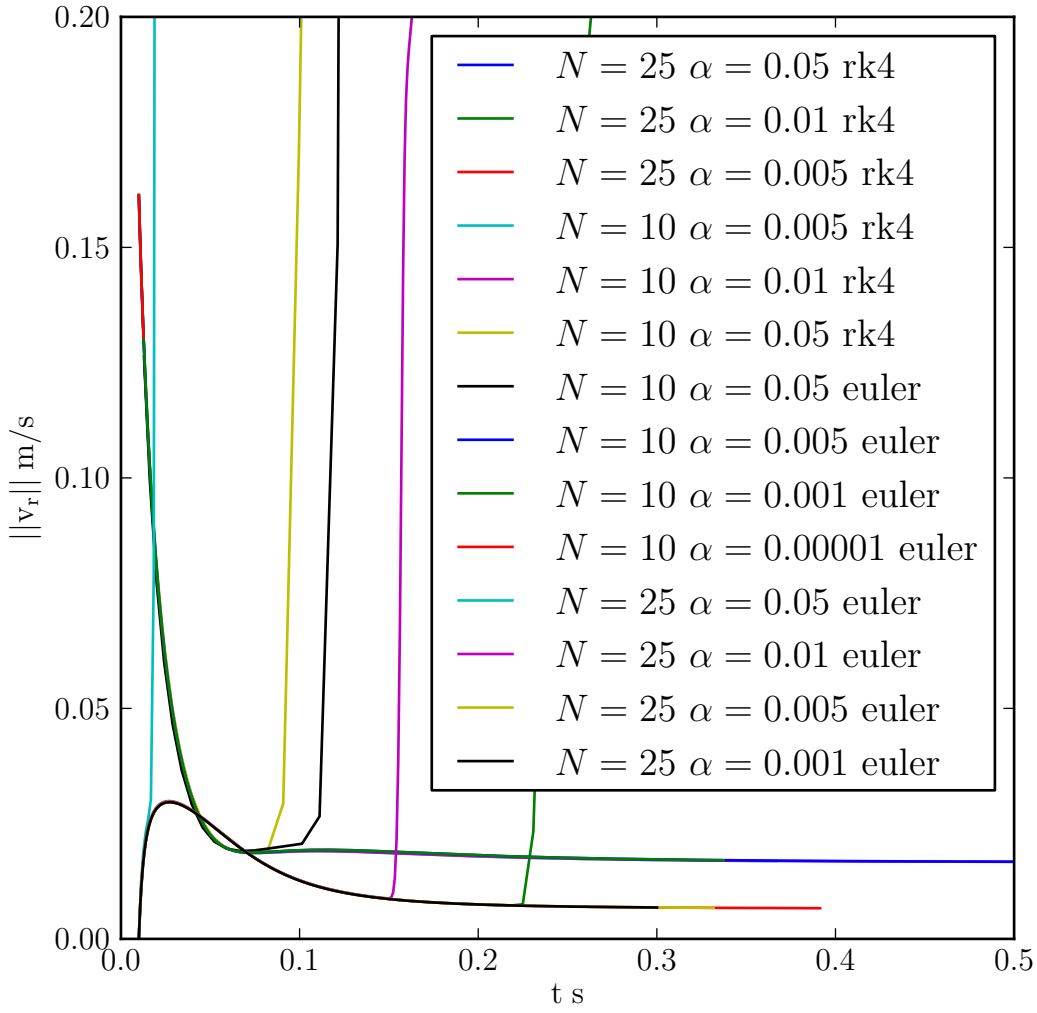


Figure 3.20: Error norm for radial component.

Chapter 4

Numerical treatment of the non fixed case

Contents

4.1 Introduction

In the previous chapter all numerical schemes were done on the same domain. We will see here the changes that we need to make if we want to have a variable topology. We will need to take care of two additional things:

1. Knowing the speed in cell that were not in the domain but are in the new time step.

2. Knowing the position of the different boundary.

We are mainly interested in a mobile boundary consisting of a free surface boundary because it will be the only boundary that moves depending on the solution.

The motion of the domain and the boundary are given by:

$$\dot{\boldsymbol{x}} = \boldsymbol{v}, \tag{4.1}$$

where $\boldsymbol{v}$ is the speed at the given position.

The spatial space is discretized into a set of cells. Cells coincide with cells in the staggered grid discretization. The domain is a union of cells. To discretize this equation, we will use particles placed in every cell. A cell where a particle lies is in the domain. A cell without particle is outside of the domain. But particles move with the speed given at their position, which is not on a grid point. Because of this, we need to interpolate the speed at the given point. When a particle moves to a new cell, we need to know the initial speed at this position. We analytically have boundary conditions for speed at boundary. We need to extrapolate this boundary condition one cell away.

4.2 Method to obtain an ODE

We can write a non discretized particle method equation as:

$$\begin{aligned}
\dot{v} &= (1 - P(D(\boldsymbol{x}))) f(\boldsymbol{v}, \boldsymbol{x}), \\
\dot{x} &= v, \\
x &= x_0,
\end{aligned}$$

where x is a continuous representation of the space with particle. $D(\boldsymbol{x})$ is the domain with particle in it. The notation $P(D(\boldsymbol{x}))$ says that we consider the projection operator with boundary conditions in the domain given by $D(\boldsymbol{x})$.

We need to add to this boundary conditions, boundary conditions for free surface, that are given by a linear differential operator A,

$$A\boldsymbol{v} = 0. \tag{4.2}$$

A contains derivatives with respect to space but not with respect to time.

When we discretize, we will need to interpolate because we need the speed at position not on the grid (for a staggered grid we have):

$$\begin{aligned}
\dot{\boldsymbol{v}}|_{\in D_v(D(\boldsymbol{x}))} &= (1 + P(D(\boldsymbol{x}))) \; f(E(\boldsymbol{v}, D_v(D(\boldsymbol{x}))), D_v(D(\boldsymbol{x})))|_{\in D_v(D(\boldsymbol{x}))}\,, \\
\dot{x} &= I\left(\boldsymbol{x}, E(\boldsymbol{v}, D_v(D(\boldsymbol{x})))\right), \\
\boldsymbol{v}|_{\notin D_v(D(\boldsymbol{x}))} &= E(\boldsymbol{v}, D_v(D(\boldsymbol{x})))|_{\notin D_v(D(\boldsymbol{x}))}\,.
\end{aligned}$$

For mathematical simplicity we will assume that $\boldsymbol{v}$ consists of a speed defined everywhere, but in the computation only the speeds that are accessed in the time step are needed (only the speed in the domain and in its neighbour). We can model this by considering non used cells with a special value like NaN that propagates in every operation.

$D(\boldsymbol{x})$ indicates the domain in which at least one particle is found. This corresponds at the same time to the topology for pressure (Pressure is cell centered).

D_v is a function depending on the topology given in general by $D(\boldsymbol{x})$ and indicates which speed components are in the domain.

f is a function that takes as argument the topology for pressure given by $D(\boldsymbol{x})$ and the topology for speed given by $D_v(D(\boldsymbol{x}))$ and the speed everywhere.

E extends the velocity field defined in the domain to the total spatial space. This is implemented by a extrapolation operator which takes as argument the speed everywhere and returns a speed everywhere, but where the speed in the speed domain is not changed. This speed is the speed that the fluid had if it were there. The distance to which this extrapolation operator needs to give values depends on the choice of the time step. The extrapolation operator needs to take into account two things:

1. The correct boundary conditions.
2. Extrapolate speed farther away than the boundary conditions if a particle moves more than one cell.

I is an interpolation operator which from the speed at discrete position finds the speed at a given position.

The use of two topologies is needed because on a staggered grid the position of speed and pressure is not the same.

Remark 4.2.1. *D_v needs to be well chosen to avoid pathological cases. For example for a domain with only one cell. We will normally consider all speed as boundary. But with only boundary we have no speed data. For this reason in this case, we consider all boundary cell as being in the domain.*

4.2.1 Splitting

A simple, but only of order 1 (in general) method to solve these equations, is to consider two different problems:

1. Advance speed for a given topology.
2. Advance the particle from the given speed.

The first step is to solve:

$$\dot{\boldsymbol{v}}|_{\in D_v(D(\boldsymbol{x}))} = (1 + P(D(\boldsymbol{x}))) \, f(E(\boldsymbol{v}, D_v(D(\boldsymbol{x}))), D_v(D(\boldsymbol{x})))|_{\in D_v(D(\boldsymbol{x}))} \,, \tag{4.3}$$

$$\boldsymbol{v_{t_0}}|_{\notin D_{v_{t_0}}(D(\boldsymbol{x}))} = E(\boldsymbol{v_{t_0}}, D_v(D(\boldsymbol{x})))|_{\notin D_v(D(\boldsymbol{x}))} \,, \tag{4.4}$$

which can be done like in the fixed topology case with a Runge-Kutta method. The second equation indicates the boundary conditions. Numerically we can calculate the speed at the boundary at every evaluation of f.

We then move the particles with the calculated speed,

$$\dot{x} = I\left(\boldsymbol{x}, E(\boldsymbol{v}, D_v(D(\boldsymbol{x})))\right). \tag{4.5}$$

This method is only of order 1.

Example 4.2.1. *Consider only one particle. With as initial condition* 0 *speed without viscosity.*

The extrapolation operator in this case will copy the speed vertically, the speed will be constant and the only non zero component is in the y direction. The speed will be given by the integration of the gravity force, the convective term is 0 *because speed is constant. The speed will be calculated with an accuracy of the order of the Runge-Kutta method used (4 for rk4). The exact solution for speed (which is constant in space) is:*

$$v_y(t) = -gt^2. \tag{4.6}$$

But the position will be calculated from the advance of a constant speed for every time step. This is then equivalent to the Euler method of order 1.

In conclusion, our speed will be correct to order k for a given time step but the position will be inacurate.

This error doesn't depend on the precision of integration for the speed or position problem. It comes from the splitting.

4.2.2 One ODE method

We will now show how we can rewrite this differential equation in only one differential equation. For this, we need an expression for the derivative of the boundary condition. When we have it, we only have to use a Runge-Kutta method.

We will now only be interested in the case where E is a linear operator with the two following constraints:

- Its effect for speed in the domain (for speed) is the identity.
- Its effect for speed outside of the domain (for speed) depends only on the speed of the domain (for speed).

The domain for speed for E is denoted by $E(D_v(D(\boldsymbol{x})))$.

In matrix notation it tells us that some blocks need to be 0 or 1. If we consider that indices from 1 to n_d are in the domain (for speed) and $n_d + 1$ to n are outside of the domain (for speed), the constraint for a line of the matrix is, using as notation the : as in matlab in indicating by an indice its position in the vector:

For $i < n_d + 1$

$$E_{i:} = (0 \dots 1_i \dots 0)\,. \tag{4.7}$$

For $i > n_d$

$$E_{i:} = \left(e^i_1 \dots e^i_{n_d} 0 \dots 0\right). \tag{4.8}$$

$$\tag{4.9}$$

Taking the derivative of $E(D_v(D(\boldsymbol{x})))\boldsymbol{v}$ has two terms, the first gives:

$$E(D_v(D(\boldsymbol{x})))\dot{\boldsymbol{v}}, \tag{4.10}$$

which uses $\dot{\boldsymbol{v}}$ but needs it only for the domain because outside the given column of the E matrix is zero.

The second term is the derivative of the matrix E with respect to change of time, which will lead to the derivative with respect to the change of topology. This will give a delta of Dirac. The reason of this is because of the discontinuity in time of the boundary speed, when we change topology. We will neglect this Dirac delta. Neglecting this Dirac delta is a dangerous steps but the problem is not as big at it could seem at first. It will be explained in details after.

The equation is then:

$$\dot{\boldsymbol{v}}|_{\in D_v(D(\boldsymbol{x}))} = (1 + P(D(\boldsymbol{x}))) \, f(E(D_v(D(\boldsymbol{x})))\boldsymbol{v}, D_v(D(\boldsymbol{x})))|_{\in D_v(D(\boldsymbol{x}))}\,, \tag{4.11}$$

$$\dot{\boldsymbol{v}}|_{\notin D_v(D(\boldsymbol{x}))} = E(D_v(D(\boldsymbol{x}))) \, (1 + P(D(\boldsymbol{x}))) \, f(E(D_v(D(\boldsymbol{x})))\boldsymbol{v}, D_v(D(\boldsymbol{x})))|_{\in D_v(D(\boldsymbol{x}))}\,, \tag{4.12}$$

$$\boldsymbol{v_{t_0}}|_{\notin D_{v_{t_0}}(D(\boldsymbol{x}))} = \left(E(D_v(D(\boldsymbol{x}))) \; \boldsymbol{v_{t_0}}|_{\in D_v(D(\boldsymbol{x}))}\right)\Big|_{\notin D_v(D(\boldsymbol{x}))}\,, \tag{4.13}$$

$$\dot{x} = I\left(\boldsymbol{x}, E(D_v(D(\boldsymbol{x})))\boldsymbol{v}\right), \tag{4.14}$$

$$\tag{4.15}$$

where we have used the notation that vector can be extended with 0 when needed because the extension is not needed because the good columns are zero, which can be rewritten as:

$$\dot{\boldsymbol{v}} = E(D_v(D(\boldsymbol{x}))) \, (1 + P(D(\boldsymbol{x}))) \, f(E(D_v(D(\boldsymbol{x})))\boldsymbol{v}, D_v(D(\boldsymbol{x})))|_{\in D_v(D(\boldsymbol{x}))}\,, \tag{4.16}$$

$$\boldsymbol{v_{t_0}}|_{\notin D_{v_{t_0}}(D(\boldsymbol{x}))} = \left(E(D_v(D(\boldsymbol{x}))) \; \boldsymbol{v_{t_0}}|_{\in D_v(D(\boldsymbol{x}))}\right)\Big|_{\notin D_v(D(\boldsymbol{x}))}\,, \tag{4.17}$$

$$\dot{x} = I\left(\boldsymbol{x}, E(D_v(D(\boldsymbol{x})))\boldsymbol{v}\right), \tag{4.18}$$

$$\tag{4.19}$$

where we have used that E is the identity in the domain. This equation can be integrated with a Runge-Kutta method. For maximal precision, the boundary speed is taken from the extrapolation at every complete Runge-Kutta step. In the argument of f we have used an extrapolation of speed which does not depend on the input in boundary speed (because the E matrix has zero columns where needed). The integrated boundary speed is only used in the case where a cell becomes a domain cell in a time step.

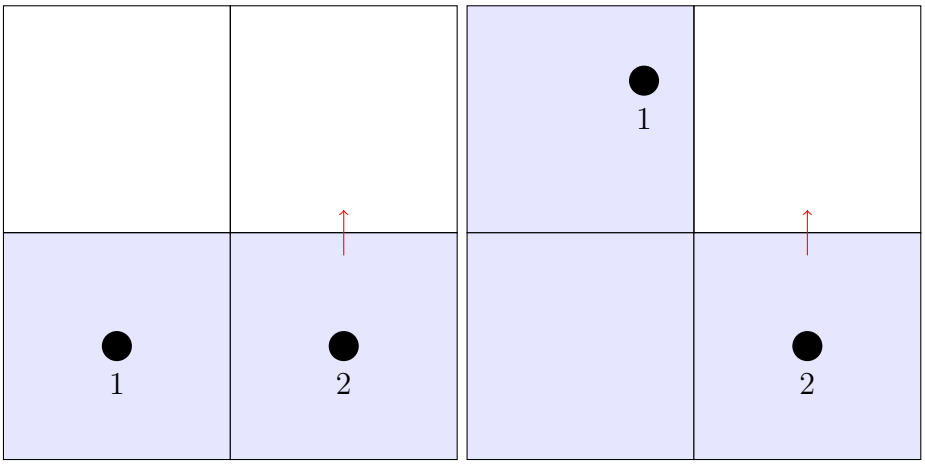
The neglected Dirac delta will only have an effect on a boundary speed component in the case where in a Runge-Kutta step the following happens:

1. A neighboring cell (cell which E matrix has component with) fills. This will change the value of boundary speed with a jump (this jump will be neglected).
2. The given speed is used because a particle fills the domain. This is needed because in the contrary the speed calculated will not be used.

This effect is shown in figure 4.1. This effect will change the precision when it happens from order k to order 1. And this error will only be on the boundary conditions being not exactly respected for a short time. The force will be integrated correctly.

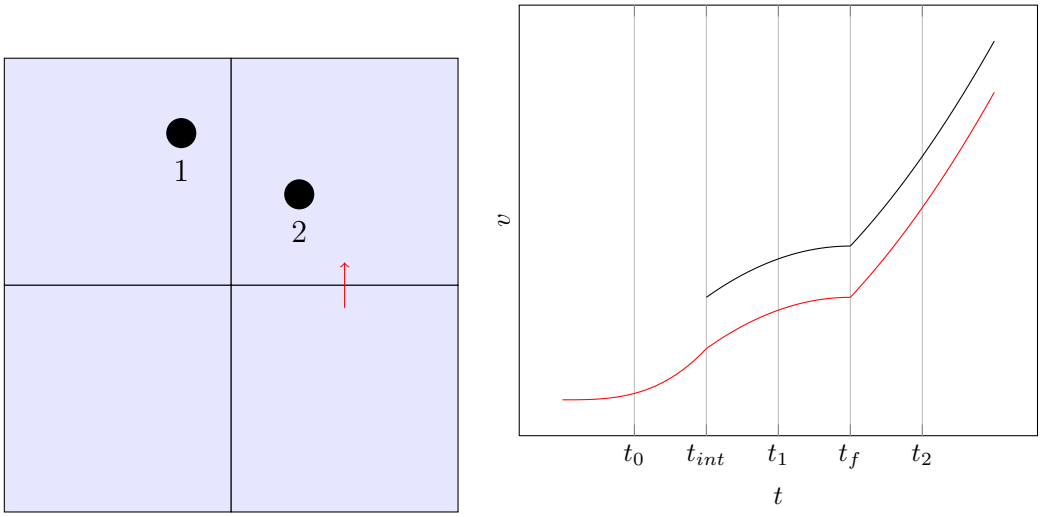
The direct filling of a cell does not pose a problem because we do not have a discontinuity in speed, because in the first region we force speed, in the second the derivative. The jump is only caused because we force speed differently for the two regions which give a discontinuity.

Example 4.2.2. *One particle with 0 initial speed, without viscosity. The speed will be constant. So only the gravity force acts and the extrapolation operator will copy speed vertically. The result will then be equivalent to the integration of the pure particle problem where the speed change is calculated at particle position. The order of the method is then k. The calculation of this example is done in section 4.6.1.*



(a) t_0, the red speed vector is not in the domain and is obtained by extrapolation.

(b) t_1, because the first particle has moved the domain haves changed, the speed extrapolation haves changed at the red point.



(c) t_2, because the red point is now in the domain, the speed is given by the acceleration from the Navier-Stokes equations.

(d) Cartoon of speed, the topology at time t_0, t_1 and t_2 are given in 4.1a , 4.1b and 4.1c. t_{in} is the moment when particle 1 enters in the new cell.t_f is the moment when particle 2 enters in the new cell. The red curve is what we will integrate if we neglect the delta of Dirac, the black curve is the exact result.

Figure 4.1: Figure of the situation where the neglected delta has a effect.

4.3 Extrapolation

We will now discuss how to define the extrapolation operator. We have seen that the extrapolation operator needs to do two things:

1. Ensure correct boundary conditions.
2. Extrapolate the rest of speed.

We now are interested in enforcing the boundary condition. Numerically this will be done by extrapolating the value at exterior from the boundary conditions on the derivative. One important point is how we enforce the boundary conditions on the discrete grid. We recall our equation of section 2.3.2:

$$\sum_{i,j} \sigma_{ij} n_i n_j = 0, \tag{4.20}$$

$$\sum_{i,j} \sigma_{ij} t_i^1 n_j = 0, \tag{4.21}$$

$$\sum_{i,j} \sigma_{ij} t_i^2 n_j = 0. \tag{4.22}$$

$$\tag{4.23}$$

Because the constraints are linear, with sufficiently additional equation (divergence free) and know value we can find an unique solution.

We now will treat the different possible cases in 2d. This will then be generalized to 3d. We will use different discretisation points for pressure and speed. We begin with the discretisation for speed.

4.3.1 2d

Plane

The first case is a surface which is a plane following one of the cell boundaries. Without loss of generality we consider the plane in the x direction (the other case are just rotations of this case). The normal and tangent vectors are then given by:

$$\boldsymbol{n} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \tag{4.24}$$

$$\boldsymbol{t} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}. \tag{4.25}$$

The equation is then given by:

$$\sum_{i,j} \sigma_{ij} t_i n_j = 0, \tag{4.26}$$

$$\sigma_{12} = 0, \tag{4.27}$$

$$-p\delta_{12} + \nu \left(\frac{\partial v_1}{\partial x_2} + \frac{\partial v_2}{\partial x_1} \right) = 0, \tag{4.28}$$

$$\frac{\partial v_1}{\partial x_2} + \frac{\partial v_2}{\partial x_1} = 0. \tag{4.29}$$

$$\tag{4.30}$$

We discretize this equation at point $i+\frac{1}{2}, j+\frac{1}{2}$:

$$\frac{v^1_{i+\frac{1}{2},j+1} - v^1_{i+\frac{1}{2},j}}{\Delta x_2} + \frac{v^2_{i+1,j+\frac{1}{2}} - v^2_{i,j+\frac{1}{2}}}{\Delta x_1} = 0, \tag{4.31}$$

where Δx_1 and Δx_2 are the spacings in x and y. We have one equation for 3 unknowns, only $v^1_{i+\frac{1}{2},j}$ are known. But we can add two equations using the divergence free condition in the fluid for cell i, j and $i+1, j$. This gives us:

$$0 = \frac{v^1_{i+\frac{1}{2},j} - v^1_{i-\frac{1}{2}}, j}{\Delta x_1} + \frac{v^2_{i,j+\frac{1}{2}} - v^2_{i,j-\frac{1}{2}}}{\Delta x_2}, \tag{4.32}$$

$$v^2_{i,j+\frac{1}{2}} = v^2_{i,j-\frac{1}{2}} - \frac{v^1_{i+\frac{1}{2},j} - v^1_{i-\frac{1}{2},j}}{\Delta x_1}\Delta x_2, \tag{4.33}$$

$$0 = \frac{v^1_{i+\frac{3}{2},j} - v^1_{i+\frac{1}{2}}, j}{\Delta x_1} + \frac{v^2_{i+1,j+\frac{1}{2}} - v^2_{i,j-\frac{1}{2}}}{\Delta x_2}, \tag{4.34}$$

$$v^2_{i+1,j+\frac{1}{2}} = v^2_{i+1,j-\frac{1}{2}} - \frac{v^1_{i+\frac{3}{2},j} - v^1_{i+\frac{1}{2},j}}{\Delta x_1}\Delta x_2. \tag{4.35}$$

We substitute the two equations (4.33) and (4.35) in equation (4.31), which will now only depend on one unknown $v^1_{i+\frac{1}{2},j+1}$ which depends on a quantity that can be found using the speed in the domain. Figure 4.2 shows the variables used and the unknowns.

We can do the things that we have done only if we have a depth of minimal two cells. If we have only one cell, we have no y speed in the domain of calculation. For this case, consider the speed y at the boundary as in the domain and let it evolve without constraint.

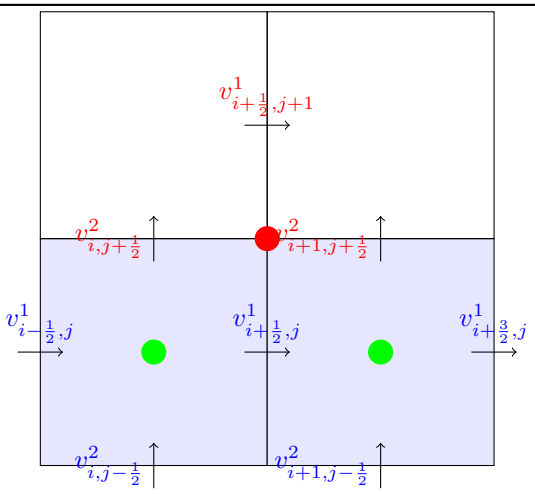


Figure 4.2: The red point is the place where the boundary condition is discretized. The green dots are cells where the divergence free rule is enforced. Red vectors are calculated with respect to the blue one.

45° plane

If we have a cell in the domain with two adjacent cells which are not in the domain, we take as normal a 45° plane.

The normal and tangent vectors are then:

$$\boldsymbol{n} = \begin{pmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{pmatrix}, \tag{4.36}$$

$$\boldsymbol{t} = \begin{pmatrix} \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \end{pmatrix}. \tag{4.37}$$

The constraint is given by:

$$\sum_{ij} \sigma_{ij} n_i t_j = 0, \tag{4.38}$$

$$\sigma_{11} n_1 t_1 + \sigma_{22} n_2 t_2 + \sigma_{12}(n_1 t_2 + n_2 t_1) = 0, \tag{4.39}$$

$$2\frac{\partial v_1}{\partial x_1}\frac{1}{2} - 2\frac{\partial v_2}{\partial x_2}\frac{1}{2} = 0, \tag{4.40}$$

$$\frac{\partial v_1}{\partial x_1} - \frac{\partial v_2}{\partial x_2} = 0. \tag{4.41}$$

We discretize at cell center i, j which gives:

$$\frac{v^1_{i+\frac{1}{2},j} - v^1_{i-\frac{1}{2},j}}{\Delta x_1} - \frac{v^2_{i,j+\frac{1}{2}} - v^2_{i,j-\frac{1}{2}}}{\Delta x_2} = 0. \tag{4.42}$$

We have two unknowns $v^1_{i+\frac{1}{2},j}$ and $v^2_{i,j+\frac{1}{2}}$. We add an additional equation using the divergence free condition at cell i, j:

$$\frac{v^1_{i+\frac{1}{2},j} - v^1_{i-\frac{1}{2}}, j}{\Delta x_1} + \frac{v^2_{i,j+\frac{1}{2}} - v^2_{i,j-\frac{1}{2}}}{\Delta x_2} = 0. \tag{4.43}$$

Figure 4.3 shows the position of variables and their elimination.

For this to work, we need to have the left and bottom boundary speed to be known. The boundary needs to be like that because in the contrary we will have no data in the x or y directions.

4.3.2 3d

The method used in 2d is generalized to 3d. The only change will be that the expressions are more complicated, and that we have more variables. The two cases that we have seen in 2d will be similar for the 2d variable but with added dependency in the z variable.

Plane

The first case is a surface which is a plane following one of the cell boundaries. Without loss of generality we consider the plane in the x and z directions. The normal and

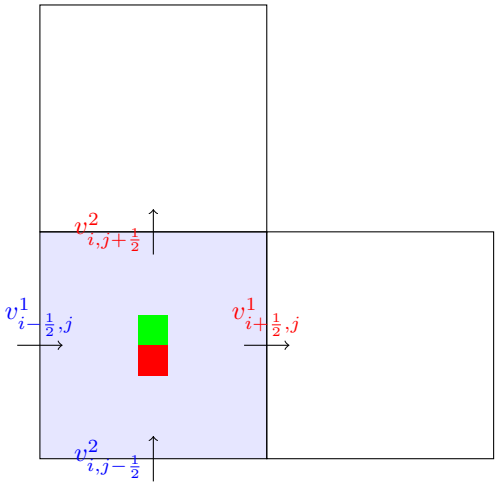


Figure 4.3: The red green rectangles are the place where the boundary equation and divergence equation are used. Red vector are expressed with respect of blue one.

tangent vectors are then given by:

$$\boldsymbol{n} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \tag{4.44}$$

$$\boldsymbol{t^1} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \tag{4.45}$$

$$\boldsymbol{t^2} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}. \tag{4.46}$$

The equations are then given by:

For $\boldsymbol{t^1}$

$$\sum_{i,j} \sigma_{ij} t^1_i n_j = 0, \tag{4.47}$$

$$\sigma_{12} = 0, \tag{4.48}$$

$$-p\delta_{12} + \nu \left(\frac{\partial v_1}{\partial x_2} + \frac{\partial v_2}{\partial x_1} \right) = 0, \tag{4.49}$$

$$\frac{\partial v_1}{\partial x_2} + \frac{\partial v_2}{\partial x_1} = 0. \tag{4.50}$$

For $\boldsymbol{t^2}$

$$\sum_{i,j} \sigma_{ij} t_i^2 n_j = 0, \tag{4.51}$$

$$\sigma_{32} = 0, \tag{4.52}$$

$$-p\delta_{32} + \nu \left(\frac{\partial v_3}{\partial x_2} + \frac{\partial v_2}{\partial x_3} \right) = 0, \tag{4.53}$$

$$\frac{\partial v_3}{\partial x_2} + \frac{\partial v_2}{\partial x_3} = 0. \tag{4.54}$$

We discretize equation (4.50) at point $i + \frac{1}{2}, j + \frac{1}{2}, k$:

$$\frac{v^1_{i+\frac{1}{2},j+1,k} - v^1_{i+\frac{1}{2},j,k+\frac{1}{2}}}{\Delta x_2} + \frac{v^2_{i+1,j+\frac{1}{2},k} - v^2_{i,j+\frac{1}{2},k}}{\Delta x_1} = 0. \tag{4.55}$$

We discretize equation (4.54) at point $i, j + \frac{1}{2}, k + \frac{1}{2}$:

$$\frac{v^3_{i,j+1,k+\frac{1}{2}} - v^3_{i,j,k+\frac{1}{2}}}{\Delta x_2} + \frac{v^2_{i,j+\frac{1}{2},k+1} - v^2_{i,j+\frac{1}{2},k}}{\Delta x_3} = 0, \tag{4.56}$$

where Δx_1, Δx_2 and Δx_3 are the spacing in x, y and z. We have 2 equations for 5 unknowns, only $v^1_{i+\frac{1}{2},j,k}$ and $v^3_{i,j,k+\frac{1}{2}}$ are known. But we can add 3 equations using the divergence free condition in the fluid for cell i, j, k, $i+1, j, k$ and $i, j, k+1$. This gives us:

$$0 = \frac{v^1_{i+\frac{1}{2},j,k} - v^1_{i-\frac{1}{2},j,k}}{\Delta x_1} + \frac{v^2_{i,j+\frac{1}{2},k} - v^2_{i,j-\frac{1}{2},k}}{\Delta x_2} + \frac{v^3_{i,j,k+\frac{1}{2}} - v^3_{i,j,k-\frac{1}{2}}}{\Delta x_3}, \tag{4.57}$$

$$v^2_{i,j+\frac{1}{2},k} = v^2_{i,j-\frac{1}{2},k} + \Delta x_2 \left(-\frac{v^1_{i+\frac{1}{2},j,k} - v^1_{i-\frac{1}{2},j,k}}{\Delta x_1} - \frac{v^3_{i,j,k+\frac{1}{2}} - v^3_{i,j,k-\frac{1}{2}}}{\Delta x_3} \right), \tag{4.58}$$

$$0 = \frac{v^1_{i+\frac{3}{2},j,k} - v^1_{i+\frac{1}{2},j,k}}{\Delta x_1} + \frac{v^2_{i+1,j+\frac{1}{2},k} - v^2_{i,j-\frac{1}{2},k}}{\Delta x_2} + \frac{v^3_{i+\frac{1}{2},j,k+\frac{1}{2}} - v^3_{i+\frac{1}{2},j,k-\frac{1}{2}}}{\Delta x_3}, \tag{4.59}$$

$$v^2_{i+1,j+\frac{1}{2},k} = v^2_{i+1,j-\frac{1}{2},k} + \Delta x_2 \left(-\frac{v^1_{i+\frac{3}{2},j,k} - v^1_{i+\frac{1}{2},j,k}}{\Delta x_1} - \frac{v^3_{i+\frac{1}{2},j,k+\frac{1}{2}} - v^3_{i+\frac{1}{2},j,k-\frac{1}{2}}}{\Delta x_3} \right), \tag{4.60}$$

$$0 = \frac{v^1_{i+\frac{1}{2},j,k+1} - v^1_{i-\frac{1}{2},j,k+1}}{\Delta x_1} + \frac{v^2_{i,j+\frac{1}{2},k+1} - v^2_{i,j-\frac{1}{2},k+1}}{\Delta x_2} + \frac{v^3_{i,j,k+\frac{3}{2}} - v^3_{i,j,k+\frac{1}{2}}}{\Delta x_3}, \tag{4.61}$$

$$v^2_{i,j+\frac{1}{2},k+1} = v^2_{i,j-\frac{1}{2},k+1} + \Delta x_2 \left(-\frac{v^1_{i+\frac{1}{2},j,k+1} - v^1_{i-\frac{1}{2},j,k+1}}{\Delta x_1} - \frac{v^3_{i,j,k+\frac{3}{2}} - v^3_{i,j,k+\frac{1}{2}}}{\Delta x_3} \right). \tag{4.62}$$

$$\tag{4.63}$$

We substitute the equations (4.58), (4.60) and (4.62) in equations (4.55) and (4.56), Which will now only depend on two unknowns $v^1_{i+\frac{1}{2},j+1,k}$ and $v^3_{i,j+1,k+\frac{1}{2}}$ which depend on quantities that can be found using the speed in the domain.

45° plane

If we have a cell in domain with two adjacent cells which are not in the domain, we take as normal a 45° plane. The normal and tangent vectors are then:

$$\boldsymbol{n} = \begin{pmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \\ 0 \end{pmatrix}, \tag{4.64}$$

$$\boldsymbol{t^1} = \begin{pmatrix} \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \\ 0 \end{pmatrix}, \tag{4.65}$$

$$\boldsymbol{t^2} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}. \tag{4.66}$$

The constraints are given by:

For $\boldsymbol{t^1}$

$$\sum_{ij} \sigma_{ij} n_i t_j^1 = 0, \tag{4.67}$$

$$\sigma_{11} n_1 t_1^1 + \sigma_{22} n_2 t_2^1 + \sigma_{12}(n_1 t_2^1 + n_2 t_1^1) = 0, \tag{4.68}$$

$$2\frac{\partial v_1}{\partial x_1}\frac{1}{2} - 2\frac{\partial v_2}{\partial x_2}\frac{1}{2} = 0, \tag{4.69}$$

$$\frac{\partial v_1}{\partial x_1} - \frac{\partial v_2}{\partial x_2} = 0. \tag{4.70}$$

For $\boldsymbol{t^2}$

$$\sum_{ij} \sigma_{ij} n_i t_j^2 = 0, \tag{4.71}$$

$$\sigma_{13} n_1 t_3^2 + \sigma_{23} n_2 t_3^2 = 0, \tag{4.72}$$

$$\frac{\partial v_1}{\partial x_3} + \frac{\partial v_3}{\partial x_1} + \frac{\partial v_2}{\partial x_3} + \frac{\partial v_3}{\partial x_2} = 0. \tag{4.73}$$

In this case, we are able to solve equation (4.70). We discretize at cell center i, j, k which gives:

$$\frac{v^1_{i+\frac{1}{2},j,k} - v^1_{i-\frac{1}{2},j,k}}{\Delta x_1} - \frac{v^2_{i,j+\frac{1}{2},k} - v^2_{i,j-\frac{1}{2},k}}{\Delta x_2} = 0. \tag{4.74}$$

We have two unknowns $v^1_{i+\frac{1}{2},j}$ and $v^2_{i,j+\frac{1}{2}}$. We add an additional equation using the divergence free condition at cell i, j, k:

$$\frac{v^1_{i+\frac{1}{2},j,k} - v^1_{i-\frac{1}{2},j,k}}{\Delta x_1} + \frac{v^2_{i,j+\frac{1}{2},k} - v^2_{i,j-\frac{1}{2},k}}{\Delta x_2} + \frac{v^3_{i,j,k+\frac{1}{2}} - v^3_{i,j,k-\frac{1}{2}}}{\Delta x_3} = 0. \tag{4.75}$$

If we want to be exact, we will need to consider in this equation $v^3_{i,j,k+\frac{1}{2}}$ and $v^3_{i,j,k-\frac{1}{2}}$ as unknown and try to determine them from equation (4.73). But a good discretisation will need to use speed outside of the cell. Which will lead to a recursive solving.

A first approximation is to ignore equation (4.73) and consider $v^3_{i,j,k+\frac{1}{2}}$ and $v^3_{i,j,k-\frac{1}{2}}$ as known.

For this to work, we need to have the left and below boundary speed to be known. The boundary needs to be like that because in the contrary we will have no data in the x or y directions.

Diagonal sloped plane

We now look at a really 3d case, where the plane is diagonal on the 3 axes. The normal and tangent vectors are:

$$\boldsymbol{n} = \begin{pmatrix} \frac{\sqrt{3}}{3} \\ \frac{\sqrt{3}}{3} \\ \frac{\sqrt{3}}{3} \end{pmatrix}, \tag{4.76}$$

$$\boldsymbol{t}^1 = \begin{pmatrix} -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \\ 0 \end{pmatrix}, \tag{4.77}$$

$$\boldsymbol{t}^2 = \begin{pmatrix} 0 \\ \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \end{pmatrix}. \tag{4.78}$$

The constraints are then:

For $\boldsymbol{t^1}$

$$0 = \sum_{ij} \sigma_{ij} n_i t^1_j, \tag{4.79}$$

$$0 = \sigma_{11} n_1 t^1_1 + \sigma_{12} n_1 t^1_2 + \sigma_{21} n_2 t^1_1 + \sigma_{22} n_2 t^1_2 + \sigma_{31} n_3 t^1_1 + \sigma_{32} n_3 t^1_2, \tag{4.80}$$

using that $t^1_1 = -t^1_2$ and $\sigma_{ij} = \sigma_{ji}$

$$0 = n_1 t^1_1 (\sigma_{11} - \sigma_{12}) + n_2 t^1_1 (\sigma_{21} - \sigma_{22}) + n_3 t^1_1 (\sigma_{31} - \sigma_{32}), \tag{4.81}$$

using that $n_1 = n_2 = n_3$

$$0 = \sigma_{11} - \sigma_{12} + \sigma_{21} - \sigma_{22} + \sigma_{31} - \sigma_{32}, \tag{4.82}$$

$$0 = 2\frac{\partial u_1}{\partial x_1} - 2\frac{\partial u_2}{\partial x_2} + \frac{\partial u_3}{\partial x_1} + \frac{\partial u_1}{\partial x_3} - \frac{\partial u_3}{\partial x_2} - \frac{\partial u_2}{\partial x_3}. \tag{4.83}$$

For $\boldsymbol{t^2}$

$$0 = \sum_{ij} \sigma_{ij} n_i t^2_j, \tag{4.84}$$

$$0 = \sigma_{13} n_1 t^2_3 + \sigma_{12} n_1 t^2_2 + \sigma_{23} n_2 t^2_3 + \sigma_{22} n_2 t^2_2 + \sigma_{33} n_3 t^2_3 + \sigma_{32} n_3 t^2_2, \tag{4.85}$$

using that $t^2_2 = -t^2_3$ and $\sigma_{ij} = \sigma_{ji}$

$$0 = n_1 t^2_3 (\sigma_{13} - \sigma_{12}) + n_2 t^2_3 (\sigma_{23} - \sigma_{22}) + n_3 t^2_3 (\sigma_{33} - \sigma_{32}), \tag{4.86}$$

using that $n_1 = n_2 = n_3$

$$0 = \sigma_{13} - \sigma_{12} + \sigma_{23} - \sigma_{22} + \sigma_{33} - \sigma_{32}, \tag{4.87}$$

$$0 = 2\frac{\partial u_3}{\partial x_3} - 2\frac{\partial u_2}{\partial x_2} + \frac{\partial u_1}{\partial x_3} + \frac{\partial u_3}{\partial x_1} - \frac{\partial u_1}{\partial x_2} - \frac{\partial u_2}{\partial x_1}. \tag{4.88}$$

We discretize equation (4.83) at position i, j, k:

$$\begin{aligned} 0 = 2\frac{v^1_{i+\frac{1}{2},j,k} - v^1_{i-\frac{1}{2},j,k}}{\Delta x_1} - 2\frac{v^2_{i,j+\frac{1}{2}.k} - v^2_{i,j-\frac{1}{2},k}}{\Delta x_2} \\ + \frac{v^3_{i,j,k+\frac{1}{2}} + v^3_{i,j,k-\frac{1}{2}} - v^3_{i-1,j,k+\frac{1}{2}} - v^3_{i-1,j,k-\frac{1}{2}}}{2\Delta x_1} \\ + \frac{v^1_{i+\frac{1}{2},j,k} + v^1_{i-\frac{1}{2},j,k} - v^1_{i+\frac{1}{2},j,k-1} - v^1_{i-\frac{1}{2},j,k-1}}{2\Delta x_3} \\ - \frac{v^3_{i,j,k+\frac{1}{2}} + v^3_{i,j,k-\frac{1}{2}} - v^3_{i,j-1,k+\frac{1}{2}} - v^3_{i,j-1,k-\frac{1}{2}}}{2\Delta x_2} \\ - \frac{v^2_{i,j+\frac{1}{2},k} + v^2_{i,j-\frac{1}{2},k} - v^2_{i,j+\frac{1}{2},k-1} - v^2_{i,j-\frac{1}{2},k-1}}{2\Delta x_3}. \end{aligned} \tag{4.89}$$

We discretize equation 4.88 at position i, j, k:

$$\begin{aligned} 0 = 2\frac{v^3_{i,j,k+\frac{1}{2}} - v^3_{i,j,k-\frac{1}{2}}}{\Delta x_3} - 2\frac{v^2_{i,j+\frac{1}{2}.k} - v^2_{i,j-\frac{1}{2},k}}{\Delta x_2} \\ + \frac{v^3_{i,j,k+\frac{1}{2}} + v^3_{i,j,k-\frac{1}{2}} - v^3_{i-1,j,k+\frac{1}{2}} - v^3_{i-1,j,k-\frac{1}{2}}}{2\Delta x_1} \\ + \frac{v^1_{i+\frac{1}{2},j,k} + v^1_{i-\frac{1}{2},j,k} - v^1_{i+\frac{1}{2},j,k-1} - v^1_{i-\frac{1}{2},j,k-1}}{2\Delta x_3} \\ - \frac{v^1_{i+\frac{1}{2},j,k} + v^1_{i-\frac{1}{2},j,k} - v^1_{i+\frac{1}{2},j-1,k} - v^1_{i-\frac{1}{2},j-1,k}}{2\Delta x_2} \\ - \frac{v^2_{i,j+\frac{1}{2},k} + v^2_{i,j-\frac{1}{2},k} - v^2_{i-1,j+\frac{1}{2},k} - v^2_{i-1,j-\frac{1}{2},k}}{2\Delta x_1}. \end{aligned} \tag{4.90}$$

The unknowns are $v^1_{i+\frac{1}{2},j,k}$, $v^2_{i,j+\frac{1}{2},k}$ and $v^3_{i,j,k+\frac{1}{2}}$. The last equation is given by the divergence free condition at cell i, j, k:

$$0 = \frac{v^1_{i+\frac{1}{2},j,k} - v^1_{i-\frac{1}{2},j,k}}{\Delta x_1} + \frac{v^2_{i,j+\frac{1}{2},k} - v^2_{i,j-\frac{1}{2},k}}{\Delta x_2} + \frac{v^3_{i,j,k+\frac{1}{2}} - v^3_{i,j,k+\frac{1}{2}}}{\Delta x_3}. \tag{4.91}$$

Contrary to the other case we have not a direct method to solve by substitution. We will rewrite the equation as:

$$A \begin{pmatrix} v^1_{i+\frac{1}{2},j,k} \\ v^2_{i,j+\frac{1}{2},k} \\ v^3_{i,j,k+\frac{1}{2}} \end{pmatrix} = b. \tag{4.92}$$

b_3 is given by equation 4.91:

$$b_3 = \frac{v^1_{i-\frac{1}{2},j,k}}{\Delta x_1} + \frac{v^2_{i,j-\frac{1}{2},k}}{\Delta x_2} + \frac{v^3_{i,j,k-\frac{1}{2}}}{\Delta x_3}. \tag{4.93}$$

b_2 is given by equation 4.110:

$$\begin{aligned} b_2 = 2\frac{v^3_{i,j,k-\frac{1}{2}}}{\Delta x_3} - 2s_2\frac{v^2_{i,j-s_2\frac{1}{2},k}}{\Delta x_2} - \frac{v^3_{i,j,k-\frac{1}{2}} - v^3_{i-1,j,k+\frac{1}{2}} - v^3_{i-1,j,k-\frac{1}{2}}}{2\Delta x_1} \\ - \frac{v^1_{i-\frac{1}{2},j,k} - v^1_{i+\frac{1}{2},j,k-1} - v^1_{i-\frac{1}{2},j,k-1}}{2\Delta x_3} + \frac{v^1_{i-\frac{1}{2},j,k} - v^1_{i+\frac{1}{2},j-1,k} - v^1_{i-\frac{1}{2},j-1,k}}{2\Delta x_2} \\ + \frac{v^2_{i,j-\frac{1}{2},k} - v^2_{i-1,j+\frac{1}{2},k} - v^2_{i-1,j-\frac{1}{2},k}}{2\Delta x_1}. \end{aligned} \tag{4.94}$$

b_1 is given by equation 4.109:

$$\begin{aligned} b_1 = 2\frac{v^1_{i-\frac{1}{2},j,k}}{\Delta x_1} - 2\frac{v^2_{i,j-\frac{1}{2},k}}{\Delta x_2} - \frac{v^3_{i,j,k+\frac{1}{2}} + v^3_{i,j,k-\frac{1}{2}} - v^3_{i-1,j,k+\frac{1}{2}} - v^3_{i-1,j,k-\frac{1}{2}}}{2\Delta x_1} \\ - \frac{v^1_{i-\frac{1}{2},j,k} - v^1_{i+\frac{1}{2},j,k-1} - v^1_{i-\frac{1}{2},j,k-1}}{2\Delta x_3} + \frac{v^3_{i,j,k-\frac{1}{2}} - v^3_{i,j-1,k+\frac{1}{2}} - v^3_{i,j-1,k-\frac{1}{2}}}{2\Delta x_2} \\ + \frac{v^2_{i,j-\frac{1}{2},k} - v^2_{i,j+\frac{1}{2},k-1} - v^2_{i,j-\frac{1}{2},k-1}}{2\Delta x_3}. \end{aligned} \tag{4.95}$$

The matrix A is then:

$$A = \begin{pmatrix} \frac{2}{\Delta x_1} + \frac{1}{2\Delta x_3} & -\frac{2}{\Delta x_2} - \frac{1}{2\Delta x_3} & \frac{1}{2\Delta x_1} - \frac{1}{2\Delta x_2} \\ \frac{1}{2\Delta x_3} - \frac{1}{2\Delta x_2} & -\frac{2}{\Delta x_2} - \frac{1}{2\Delta x_1} & \frac{2}{\Delta x_3} + \frac{1}{2\Delta x_1} \\ \frac{1}{\Delta x_1} & \frac{1}{\Delta x_2} & \frac{1}{\Delta x_3} \end{pmatrix}. \tag{4.96}$$

This matrix only depends on the spacing so it can be calculated at initialization.

The general case is:

$$\boldsymbol{n} = \begin{pmatrix} \frac{s_1\sqrt{3}}{3} \\ \frac{s_2\sqrt{3}}{3} \\ \frac{s_3\sqrt{3}}{3} \end{pmatrix}, \tag{4.97}$$

$$\boldsymbol{t}^1 = \begin{pmatrix} -s_1\frac{\sqrt{2}}{2} \\ s_2\frac{\sqrt{2}}{2} \\ 0 \end{pmatrix}, \tag{4.98}$$

$$\boldsymbol{t}^2 = \begin{pmatrix} 0 \\ s_2\frac{\sqrt{2}}{2} \\ -s_3\frac{\sqrt{2}}{2} \end{pmatrix}. \tag{4.99}$$

The constraints are then:

For $\boldsymbol{t^1}$

$$0 = \sum_{ij} \sigma_{ij} n_i t_j^1, \tag{4.100}$$

$$0 = \frac{\sqrt{6}}{6} \left(-\sigma_{11} + \sigma_{12} s_1 s_2 - \sigma_{21} s_1 s_2 + \sigma_{22} - \sigma_{31} s_3 s_1 + \sigma_{32} s_3 s_2\right), \tag{4.101}$$

$$0 = -\sigma_{11} + \sigma_{22} - \sigma_{31} s_3 s_1 + \sigma_{32} s_3 s_2, \tag{4.102}$$

$$0 = -2\frac{\partial u_1}{\partial x_1} + 2\frac{\partial u_2}{\partial x_2} - s_3 s_1 \frac{\partial u_3}{\partial x_1} - s_3 s_1 \frac{\partial u_1}{\partial x_3} + s_3 s_2 \frac{\partial u_3}{\partial x_2} + s_3 s_2 \frac{\partial u_2}{\partial x_3}. \tag{4.103}$$

For $\boldsymbol{t^2}$

$$0 = \sum_{ij} \sigma_{ij} n_i t_j^2, \tag{4.104}$$

$$0 = \frac{\sqrt{6}}{6} \left(-\sigma_{13} s_1 s_3 + \sigma_{12} s_1 s_2 - \sigma_{23} s_2 s_3 + \sigma_{22} - \sigma_{33} + \sigma_{32} s_3 s_2\right), \tag{4.105}$$

$$0 = \frac{\sqrt{6}}{6} \left(-\sigma_{13} s_1 s_3 + \sigma_{12} s_1 s_2 + \sigma_{22} - \sigma_{33}\right), \tag{4.106}$$

$$0 = -\sigma_{13} s_1 s_3 + \sigma_{12} s_1 s_2 + \sigma_{22} - \sigma_{33}, \tag{4.107}$$

$$0 = -2\frac{\partial u_3}{\partial x_3} + 2\frac{\partial u_2}{\partial x_2} - s_1 s_3 \frac{\partial u_1}{\partial x_3} - s_1 s_3 \frac{\partial u_3}{\partial x_1} + s_1 s_2 \frac{\partial u_1}{\partial x_2} + s_2 s_1 \frac{\partial u_2}{\partial x_1}. \tag{4.108}$$

We discretize equation (4.103) at position i, j, k:

$$\begin{aligned} 0 = &-2\frac{v^1_{i+\frac{1}{2},j,k} - v^1_{i-\frac{1}{2},j,k}}{\Delta x_1} + 2\frac{v^2_{i,j+\frac{1}{2}.k} - v^2_{i,j-\frac{1}{2},k}}{\Delta x_2} \\ &- s_3 \frac{v^3_{i,j,k+\frac{1}{2}} + v^3_{i,j,k-\frac{1}{2}} - v^3_{i-s_1,j,k+\frac{1}{2}} - v^3_{i-s_1,j,k-\frac{1}{2}}}{2\Delta x_1} \\ &- s_1 \frac{v^1_{i+\frac{1}{2},j,k} + v^1_{i-\frac{1}{2},j,k} - v^1_{i+\frac{1}{2},j,k-s_3} - v^1_{i-\frac{1}{2},j,k-s_3}}{2\Delta x_3} \\ &+ s_3 \frac{v^3_{i,j,k+\frac{1}{2}} + v^3_{i,j,k-\frac{1}{2}} - v^3_{i,j-s_2,k+\frac{1}{2}} - v^3_{i,j-s_2,k-\frac{1}{2}}}{2\Delta x_2} \\ &+ s_2 \frac{v^2_{i,j+\frac{1}{2},k} + v^2_{i,j-\frac{1}{2},k} - v^2_{i,j+\frac{1}{2},k-s_3} - v^2_{i,j-\frac{1}{2},k-s_3}}{2\Delta x_3}. \end{aligned} \tag{4.109}$$

We discretize equation (4.108) at position i, j, k:

$$\begin{aligned}
0 = &-2\frac{v^3_{i,j,k+\frac{1}{2}} - v^3_{i,j,k-\frac{1}{2}}}{\Delta x_3} + 2\frac{v^2_{i,j+\frac{1}{2}.k} - v^2_{i,j-\frac{1}{2},k}}{\Delta x_2} \\
&- s_3\frac{v^3_{i,j,k+\frac{1}{2}} + v^3_{i,j,k-\frac{1}{2}} - v^3_{i-s_1,j,k+\frac{1}{2}} - v^3_{i-s_1,j,k-\frac{1}{2}}}{2\Delta x_1} \\
&- s_1\frac{v^1_{i+\frac{1}{2},j,k} + v^1_{i-\frac{1}{2},j,k} - v^1_{i+\frac{1}{2},j,k-s_3} - v^1_{i-\frac{1}{2},j,k-s_3}}{2\Delta x_3} \\
&+ s_1\frac{v^1_{i+\frac{1}{2},j,k} + v^1_{i-\frac{1}{2},j,k} - v^1_{i+\frac{1}{2},j-s_2,k} - v^1_{i-\frac{1}{2},j-s_2,k}}{2\Delta x_2} \\
&+ s_2\frac{v^2_{i,j+\frac{1}{2},k} + v^2_{i,j-\frac{1}{2},k} - v^2_{i-s_1,j+\frac{1}{2},k} - v^2_{i-s_1,j-\frac{1}{2},k}}{2\Delta x_1}.
\end{aligned} \tag{4.110}$$

The unknowns are $v^1_{i+s_1\frac{1}{2},j,k}$, $v^2_{i,j+s_2\frac{1}{2},k}$ and $v^3_{i,j,k+s_3\frac{1}{2}}$.

We can write it by matrix form:

$$A\begin{pmatrix} v^1_{i+s_1\frac{1}{2},j,k} \\ v^2_{i,j+s_2\frac{1}{2},k} \\ v^3_{i,j,k+s_3\frac{1}{2}} \end{pmatrix} = b. \tag{4.111}$$

b_3 is given by equation 4.91:

$$b_3 = s_1\frac{v^1_{i-s_1\frac{1}{2},j,k}}{\Delta x_1} + s_2\frac{v^2_{i,j-s_2\frac{1}{2},k}}{\Delta x_2} + s_3\frac{v^3_{i,j,k-s_3\frac{1}{2}}}{\Delta x_3}. \tag{4.112}$$

b_2 is given by:

$$\begin{aligned}
b_2 = &-2s_3\frac{v^3_{i,j,k-s_3\frac{1}{2}}}{\Delta x_3} + 2s_2\frac{v^2_{i,j-s_2\frac{1}{2}.k}}{\Delta x_2} + s_3\frac{v^3_{i,j,k-s_3\frac{1}{2}} - v^3_{i-s_1,j,k+\frac{1}{2}} - v^3_{i-s_1,j,k-\frac{1}{2}}}{2\Delta x_1} \\
&+ s_1\frac{v^1_{i-s_1\frac{1}{2},j,k} - v^1_{i+\frac{1}{2},j,k-s_3} - v^1_{i-\frac{1}{2},j,k-s_3}}{2\Delta x_3} - s_1\frac{v^1_{i-s_1\frac{1}{2},j,k} - v^1_{i+\frac{1}{2},j-s_2,k} - v^1_{i-\frac{1}{2},j-s_2,k}}{2\Delta x_2} \\
&- s_2\frac{v^2_{i,j-s_2\frac{1}{2},k} - v^2_{i-s_1,j+\frac{1}{2},k} - v^2_{i-s_1,j-\frac{1}{2},k}}{2\Delta x_1}.
\end{aligned} \tag{4.113}$$

b_1 is given by:

$$\begin{aligned}
b_1 = &-2s_1\frac{v^1_{i-s_1\frac{1}{2},j,k}}{\Delta x_1} + 2s_2\frac{v^2_{i,j-s_2\frac{1}{2}.k}}{\Delta x_2} + s_3\frac{v^3_{i,j,k-s_3\frac{1}{2}} - v^3_{i-s_1,j,k+\frac{1}{2}} - v^3_{i-s_1,j,k-\frac{1}{2}}}{2\Delta x_1} \\
&+ s_1\frac{v^1_{i-s_1\frac{1}{2},j,k} - v^1_{i+\frac{1}{2},j,k-s_3} - v^1_{i-\frac{1}{2},j,k-s_3}}{2\Delta x_3} - s_3\frac{v^3_{i,j,k-s_3\frac{1}{2}} - v^3_{i,j-s_2,k+\frac{1}{2}} - v^3_{i,j-s_2,k-\frac{1}{2}}}{2\Delta x_2} \\
&- s_2\frac{v^2_{i,j-s_2\frac{1}{2},k} - v^2_{i,j+\frac{1}{2},k-s_3} - v^2_{i,j-\frac{1}{2},k-s_3}}{2\Delta x_3}.
\end{aligned} \tag{4.114}$$

The matrix A is then:

$$A = \begin{pmatrix} -s_1\frac{2}{\Delta x_1} - s_1\frac{1}{2\Delta x_3} & s_2\frac{2}{\Delta x_2} + s_2\frac{1}{2\Delta x_3} & -s_3\frac{1}{2\Delta x_1} + s_3\frac{1}{2\Delta x_2} \\ -s_1\frac{1}{2\Delta x_3} + s_1\frac{1}{2\Delta x_2} & s_2\frac{2}{\Delta x_2} + s_2\frac{1}{2\Delta x_1} & -s_3\frac{2}{\Delta x_3} - s_3\frac{1}{2\Delta x_1} \\ s_1\frac{1}{\Delta x_1} & s_2\frac{1}{\Delta x_2} & s_3\frac{1}{\Delta x_3} \end{pmatrix}. \tag{4.115}$$

4.3.3 Other cases

With the different cases seen above, we cannot treat all cases. Two things can happen.

1. The domain is too thin (only one cell), in this case we consider the boundary speed as in the domain, because we need at least 2 speeds of the same component in a given direction to be able to do calculations.

2. We have a succession of plane and diagonal cells, but the plane cells are not sufficiently long to use the above case. Or other cases that cannot be decomposed into the above case. In this case we extrapolate unknown speed from the mean of the neighboring speed in the same speed component. We then evenly distribute the divergence to the unknown speed to be divergence free.

This technique allows us to define a speed to every boundary of a cell in the domain.

We will then need to extrapolate it further away with a technique shown in the next section. If a speed that is not directly at boundary is defined it will be protected from erasure for the next stage (an implementation technique for this is to put a NaN value into unknown).

4.3.4 Pressure boundary conditions

The pressure boundary conditions are applied to the cells outside of the domain. The condition comes from the study from a plane of only one cell. For example for a normal of:

$$\boldsymbol{n} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}. \tag{4.116}$$

The equations are:

$$0 = \sigma_{11} n_1 = -p + 2\nu \frac{\partial v_1}{\partial x_1}, \tag{4.117}$$

$$p = 2\nu \frac{\partial v_1}{\partial x_1}. \tag{4.118}$$

Which using the speed found from previous step gives:

$$p = 2\nu \frac{v^1_{i+\frac{1}{2},j} - v^1_{i,j}}{\Delta x_1}. \tag{4.119}$$

In general we can be neighbor of more than one of these cases. In this case we sum the given pressure.

4.3.5 Extrapolate further away

We now need to extrapolate further away to be able to use bigger time step, and for corner case to have speed defined if a particle visit a new corner:

> For every unknown speed, average from known speeds that are nearer from a surface of the fluid.

This is done by implementing first a notion of "Layer" which indicates the distance from the fluid for a given cell. The extrapolation is then done layer by layer using only values of the layer before.

The exact code is a little complicated because of the staggered grid which brings a little asymmetry in choosing which cell layer needs to be used (a speed component is between 2 cells).

4.4 Interpolation

For interpolation, we do not have constraints on the scheme. A simple one can for example be n-linear interpolation. The only thing to take care is that we are in a staggered grid so that known speeds are not at the same place for all components.

4.4.1 n-linear

A n linear interpolation is a non linear interpolation but a product of linear interpolations so that in every direction the interpolation is linear.

For 1d it is given for c_1 and c_2 the value at x_1 and x_2 by:

$$f(x,y) = c_2\frac{x-x_1}{x_2-x_1} + c_1\frac{x-x_2}{x_1-x_2}. \tag{4.120}$$

For 2d it is given by:

$$f(x,y) = c_{11}\frac{x-x_2}{x_1-x_2}\frac{y-y_2}{y_1-y_2} + c_{12}\frac{x-x_2}{x_1-x_2}\frac{y-y_1}{y_2-y_1} + c_{21}\frac{x-x_1}{x_2-x_1}\frac{y-y_2}{y_1-y_2} + c_{22}\frac{x-x_2}{x_1-x_2}\frac{y-y_2}{y_1-y_2}. \tag{4.121}$$

For 3d it is calculated similarly.

4.5 Pseudo code

We will now give pseudo code to augment the program seen in the previous chapter.

4.5.1 Data types

In addition to the data types that we have seen in the previous chapter, we add the following data types:

Layer: layers are integers stored in the cell which indicates the distance from the fluid. Two methods are defined. A method to set (**LayerSet**) and a method to get (**LayerGet**). A special value of -1 is used for indicating empty cells. This is used in the initialization algorithm before considering which values the layer have.

Depth: Depths are integers stored in the cell which indicate the depth in the fluid. This is used for optimization of the number of particles. Particles are only used near the surface. Not deep inside the fluid. Two methods are defined. A method to set (**DepthSet**) and a method to get (**DepthGet**). A special value of -1 is used for indicating an empty cell

Particle: We need a list of particles which consist only of a list of position. A method (**CellGet**()) that transforms a particle in the cell which it lays in is needed. This method can be written in terms of rounding after a scaling so that cell spacings are 1. Methods to add and remove particles in the list are needed. The order in the list is not relevant, we only need to be able to remove a particle in an arbitrary position in the list.

Interior: A special cell type, interior cell is used. Interior cells are considered as cell fluid, but where the method **GetIsInterior** returns true. Interior cells are cells which do not need to have fluid particle in there. The only mean to remove an interior cell is to "Remove" the fluid cells around it.

4.5.2 Initialisation

Overview

In this part we need to do the following:

- From the position of the particle determine which cell is fluid.
- Add particles in inflow cell.
- Create air cell around the fluid to store extrapolation data.
- Create a variable named layer which has value 0 in the fluid and the distance from the fluid in air cells. This is used in the the extrapolation method.

This part is the only part that creates cells or creates particles. The difficulty of this part is that we have values that are not updated or do not exist. The order of the algorithm is very important.

To do this we use the following steps:

1. We initialize a variable called layer on every cell to a value of -1.
2. We put layer to 0 for cells with a particle in it and set the cell type to fluid.
3. We create particles in inflow if needed.
4. We iteratively set layers away from surface and set air cells.
5. We delete cells with layer of -1.

Because of compression and for optimization of the number of particles we use another variable called depth. This variable indicates the depth in fluid from the surface. Cells which are bigger than a certain depth are interior cells. Interior cells are considered as fluid cells but which do not need particles in there. Particles are created from a certain depth in cell to allow a compression effect which will deform the sampling of particles. To have only particles at the surface allows us to have more

particles at the surface with the same performance. This is important in 3d flows where the number of particles increases with h^3 if we put particles everywhere. We can do this mathematically because an opening in the fluid can only happen when we have a discontinuity in speed. This discontinuity can be detected in the speed on the grid.

Initialize layer and depth

This algorithm initializes the layer of every cell to -1 so that we can set the layer to 0 of cells which have a particle in it. For depth we only consider cell which are in the exterior fluid cells and do not touch interior cells so that they have the maximal depth of the preceding iteration. This is shown in algorithm 14

Algorithm 14 Algorithm which initializes the layer and depth.

```
1: procedure InitializeLayerAndDepth
2:     for all it ∈ Grid do
3:         it.LayerSet(−1)                ▷ We set layer to −1 so that other method can fill the layer.
4:         if it.GetIsFluid() and not(it.GetIsInterior()) then ▷ These are fluid cells at exterior.
5:             it.DepthSet(−1)                              ▷ We set their depth to −1 to initialize.
6:         end if
7:     end for
8: end procedure
```

Initialize layer with particle and depth

This algorithm will use the position of the particles to set the layer where the particles are to 0. It will also delete particles that are in the interior more than a given depth because particles are not needed there. This can be seen in algorithm 15

Algorithm 15 Algorithm which initialize layer and depth from the particles.

```
1: procedure LayerInitialWithParticleDepth
2:     for all it ∈ Particles do                                   ▷ We loop for all particle
3:         cell ← it.CellGet()  ▷ Use of the particle position to know which cell the particle lies in.
4:         if cell.GetIsInterior() then
5:             cell.LayerSet(0)
6:             if cell.DepthGet() > depmax then      ▷ The particle is at interior of more than a
   certain depth.
7:                 Particles.erase(it)                                   ▷ We erase it.
8:             end if
9:         else
10:            cell.LayerSetLayer(0)
11:            cell.FluidSet()          ▷ We set to fluid cells because a particle lies in this position.
12:        end if
13:    end for
14: end procedure
```

Create fluid particle depth

This algorithm creates particles if needed and forces that interior cells are filled. New particles are needed if:

1. It is a inbound cell which has no particles in it (has layer -1) which represent a source of fluid of the physical problem.

2. It is an interior cell which has depth smaller than the limit to remove particles.

The needs of interior cells are for two reasons:

1. Performance optimization because with interior the only means to not be fluid is by a discontinuity on the speed. This discontinuity can be found on the speed on the grid. So we do not need particles at interior.

2. By deformation of the fluid the sampling of fluid can be insufficient and air cells can manifest because we have not sufficient particles. It allow to have a bigger number of particle by cell to be more resistant to deformation effect only where it is needed, at the surface. Particle are created in the interior region if needed to go in the surface region to add sampling.

The algorithm is shown in algorithm 16. The method **InboundNeedFilling** and **AddParticle** need to be defined. The first tests if we are an empty inbound cell. The second will create one or more particles in the cell.

An identical algorithm without creation of particles named **CreateInteriorDepth** is used (cf. algorithm 17) when we are in a Runge-Kutta step and do not want to create new particles because vectors will no more have the same length which is problematic for a Runge-Kutta method.

Algorithm 16 Algorithm which creates new particles if needed and set that interior are always filled.

```
1: procedure LayerInitialWithParticleDepth
2:     for all it ∈ Grid do
3:         if InboundNeedFilling(it) then                    ▷ We need filling.
4:             AddParticle(it)
5:             it.LayerSet(0)
6:             it.FluidSet()
7:         else if it.GetIsInterior() and it.LayerGet() = −1 and it.DepthGet() ≤ depthmax
   then          ▷ We are an interior cell which is near the surface and is empty. Create a particle.
8:             AddParticle(it)
9:             it.LayerSet(0)
10:        else if it.GetIsInterior() then▷ Interior cells always have a layer of 0. Because they are
   considered as fluid.
11:            it.LayerSet(0)
12:        end if
13:    end for
14: end procedure
```

Update cellType layer depth

We now need to create the layer variable and depth variable and create air cell around the fluid. We do this recursively beginning from the fluid and avancing neighbour by neighbour.

This is done in 3 parts. First algorithm 18 create the first list with boundary cells. Then algorithm 19 create the layer moving away from the fluid. In contrary algorithm 20 creates the depth moving into the fluid from the surface.

Algorithm 17 Algorithm which set layer to 0 if needed but does't creates particles and set that interior are always filled.

```
1: procedure CreateInteriorDepth
2:     for all it ∈ Grid do
3:         if InboundNeedFilling(it) then                    ▷ We need filling.
4:             it.LayerSet(0)
5:             it.FluidSet()
6:         else if it.GetIsInterior() then    ▷ Interior fluid always have a 0 layer. No particle is created.
7:             it.LayerSet(0)
8:         end if
9:     end for
10: end procedure
```

Algorithm 18 Algorithm which creates the first layer list with cells at the boundary.

```
1: procedure UpdateCellTypeLayerDepth
2:     stack ← newunorderedstack()
3:     for all it ∈ Grid do
4:         it.DepthSet(−1)                          ▷ Every cell begin with a depth of −1
5:         b ← false
6:         if it.LayerGet() = 0 then
7:             for i = 1 to dim do
8:                 if not(b) then                   ▷ b is a boolean used to indicate the need to exit.
9:                     for si = ±1 do
10:                        it2 ← it.GetNeighbour(i, si)
11:                        if not(it2.IsValid()) or it2.LayerGet() = −1 then    ▷ If neighbor cell doesn't exist or has an empty layer, this is a sign of a non fluid cell.
12:                            it.DepthSet(0)
13:                            stack.insert(it)                  ▷ Insert in stack the current cell.
14:                            b ← true ▷ We have added in stack so that we need not to reprocess this cell. This will only insert more than one time this cell.
15:                            break
16:                        end if
17:                    end for
18:                end if
19:            end for
20:        end if
21:    end for
```

Algorithm 19 Algorithm which use the first list to create layer recursively away from the surface.

22: *stackbackup* ← *stack* ▷ We backup the stack, this is needed because in depth part we will need the same stack.
23: **for** $lay = 1$ **to** *laymax* **do**
24: *stack*2 ← **newunorderedstack**() ▷ Stack that will be filled with the next candidate.
25: **for all** *cell* ∈ *stack* **do** ▷ We loop in all *cell* in the *stack*.
26: **for** $i = 1$ **to** *dim* **do**
27: **for** $si = \pm 1$ **do**
28: $cell2 \leftarrow it.\mathbf{GetNeighbour}(i, si)$
29: **if not**($cell2.\mathbf{IsValid}()$) **or** $cell2.\mathbf{LayerGet}() = -1$ **then** ▷ We redo the test if the neighbour cell is air or none existent.
30: $cell.\mathbf{Set}(i, it.\mathbf{Get}(i) + si)$
31: $cell.\mathbf{CreateCell}()$ ▷ We create the neighbour cell to be able to add information in it.
32: $cell.\mathbf{LayerSet}(lay)$ ▷ We set the layer of the new cell.
33: $cell.\mathbf{AirSet}()$ ▷ The new cell is an air cell.
34: $stack2.\mathbf{insert}(it)$ ▷ We insert the new cell in the new stack because as new cell it's a candidate for new air.
35: $cell.\mathbf{Set}(i, it.\mathbf{Get}(i) - si)$
36: **end if**
37: **end for**
38: **end for**
39: **end for**
40: *stack* ← *stack*2 ▷ The new list becomes the current list.
41: **end for**
42: **for all** *cell* ∈ *Stack* **do** ▷ A last loop to move in the positif direction because of the asymetry in speed.
43: **for** $i = 1$ **to** *dim* **do**
44: $cell2 \leftarrow cell.\mathbf{GetNeighbour}(i, 1)$
45: **if not**($cell2.\mathbf{IsValid}()$) **or** $cell2.\mathbf{LayerGet}() = -1$ **then**
46: $cell.\mathbf{Set}(i, it.\mathbf{Get}(i) + si)$
47: $cell.\mathbf{CreateCell}()$
48: $cell.\mathbf{LayerSet}(laymax + 1)$
49: $cell.\mathbf{AirSet}()$
50: $cell.\mathbf{Set}(i, it.\mathbf{Get}(i) - si)$
51: **end if**
52: **end for**
53: **end for**

Algorithm 20 Algorithm which uses the first list to create depth recursively in from the surface.

```
54:     depth ← 1                                                    ▷ The depth is 1.
55:     loop
56:         stack2 ← newunorderedstack()          ▷ New stack that we will fill with candidate cell.
57:         for all cell ∈ stackbackup do          ▷ We use the stackbackup because it contains at first
    iteration the content after the first step.
58:             for i = 1 to dim do
59:                 for si± do
60:                     cell2 ← cell.GetNeighbour(i, si)
61:                     if cell2.LayerGet() = 0 and cell2.DepthGet() = −1 then ▷ If the neighbour
    is fluid and has depth of −1.
62:                         cell.Set(i, cell.Get(i) + si)
63:                         cell.LayerSet(0)
64:                         cell.DepthSet(depth)
65:                         if depth < interiordepth then ▷ If the depth is smaller that the limit we set
    the cell as fluid cell.
66:                             cell.FluidSet()
67:                         else
68:                             cell.InteriorSet()  ▷ If the depth is bigger than the limit we set the cell
    as interior.
69:                         end if
70:                         stack2.insert(it)                        ▷ We add the cell in the list.
71:                         cell.Set(i, cell.Get(i) − si)
72:                     end if
73:                 end for
74:             end for
75:         end for
76:         stackbackup ← stack2                          ▷ The new list becomes the current list.
77:         if stackbackup.empty() then ▷ If the new list is empty we have ended. It's the only way
    to exit the loop.
78:             break
79:         end if
80:     end loop
81: end procedure
```

Delete macCell

We delete cells that have layerw of -1 because they are cells that are not fluid and not in the air layer around the fluid. Retaining cell with layer of -1 will not change the mathematics. They will only be dead weight. This is shown in algorithm 21.

Algorithm 21 Algorithm to delete cell if layer is -1.

```
1: procedure DeleteMacCell
2:     for all it ∈ Grid do
3:         if it.LayerGet() = −1 then
4:             Grid.erase(it)
5:         end if
6:     end for
7: end procedure
```

Calculate time step

We calculate the optimun time step. The optimun time step is given so that a particle moves a maximun of a given fraction of a cell. This is shown in algorithm 22

Algorithm 22 Calculates the optimum time step.

```
1: procedure CalculateTimeStep max=0;
2:     for all it ∈ Grid do          ▷ We calculate the maximun speed weighted from the spacing.
3:         if it.GetIsFluid() then
4:             vtemp ← 0
5:             for i = 1 to dim do          ▷ Calculation of norm 2 of speed weighted with spacing.
6:                 vtemp ← vtemp + (it.SpeedGet(i) · h.Get(i))^2
7:             end for
8:             if vtemp > max then          ▷ If we are bigger than actual maximum, we have a new
   maximun.
9:                 max ← vtemp
10:            end if
11:        end if
12:    end for
13:    dt ← factor/sqrt(max)                          ▷ New candidate time step.
14:    dt ← CheckDT(dt)          ▷ We check the time step (min, max checking for example).
15: end procedure
```

Conclusion

We now have all the algorithms to initialize the topology. We note that all these steps only involve integer calculations, this allows the compiler to do extensive optimization to simplify calculations.

We have two different initialization cases:

1. Complete initialization done before a Runge-Kutta step. We create particles and calculate the optimum time step.

2. Partial initialization done in a Runge-Kutta step. We do not create particle and do not change the time step. But the rest is the same as a complete initialization step.

The two algorithms are shown in algorithms 23 and 24.

Algorithm 23 Complete initialization

```
1: procedure Initialization
2:     InitializeLayerAndDepth()
3:     LayerInitialWithParticleDepth()
4:     UpdateCellTypeLayerDepth()
5:     DeleteMacCell()
6:     CalculateTimeStep()
7: end procedure
```

Algorithm 24 Complete initialization

```
1: procedure InitializationRungeKutta
2:     InitializeLayerAndDepth()
3:     CreateInteriorDepth()
4:     UpdateCellTypeLayerDepth()
5:     DeleteMacCell()
6: end procedure
```

We will now consider extrapolation methods.

4.5.3 Boundary conditions extrapolation

Overview

The following steps are needed to determine boundary conditions in 2d:

1. Mark all speeds between air cells to **NaN**.
2. Find boundary cells: Fluid cells that are neighbor of air cells.
3. Choose which case of section 4.3.1 and apply it. If we cannot use a case we mark the speed with **NaN**.
4. Extrapolate boundary speeds which are **NaN** and correct divergence.

Extreme care needs to be taken to only use known speeds. Topology is very important for this part.

Speed in domain

A speed component is in the domain if its two neighbor cells are in the domain or we are too thin. The function takes 2 arguments, a cell and a direction and returns a boolean. This pseudo code assumes that all cells exist and are fluid or not fluid. In the real code the existence of the cell muss be checked.

Boundary between fluid and air

We now need to know when a fluid cell is a boundary between fluid and air, and extract information of which direction is the boundary. As input, we take the cell which we are interested in. As output a boolean if we are a boundary, and 2 arrays where we return every direction and sign of the interface. The value n indicates how many interfaces we have. This is shown in algorithm 26.

Algorithm 25 Algorithm to find if a speed component is in the domain or not.

function GETISSPEEDINDOMAIN($speed, dir$) ▷ *speed* is the position of the cell, *dir* the direction
  $speed2 \leftarrow speed.$ **GetNeighbour**$(dir, -1)$ ▷ *speed*2 is now the other cell touching the given speed component.
  if $speed.$ **GetIsFluid**() **and** $speed2.$ **GetIsFluid**() **then**
    return true ▷ We are between 2 fluid cells, we are in the domain.
  end if
  if not(SPEED.GETISFLUID()) **and not**(SPEED2.GETISFLUID()) **then**
    return false ▷ We are between 2 non fluid cells, we are out the domain.
  end if
    ▷ Now one of *speed* or *speed*2 is fluid. *speed*3 is the cell at the other side of the fluid cell.
  if $speed.$ **GetIsFluid**() **then**
    $speed3 \leftarrow speed.$ **GetNeighbour**$(dir, 1)$ ▷ Because *speed* is fluid *speed*3 is the other side of *speed* than *speed*2
  else if $speed2.$ **GetIsFluid**() **then**
    $speed3 \leftarrow speed2.$ **GetNeighbour**$(dir, -1)$ ▷ Because *speed*2 is fluid *speed*3 is the other side of *speed*2 than *speed*
  end if
  if not($speed3.$ **GetIsFluid**) **then**
    return true ▷ We are in a thin fluid case, because we have 2 neighbor air cells *speed*3 and one of *speed*2 or *speed*
  end if
  return false
end function

Algorithm 26 Algorithm to find if a fluid cell is a neighbor of air cell. And have a list of interface.

function ISBOUNDARY($cell, dir[2 * dim], sign[2 * dim], n$) ▷ *dim* is the space dimension,*cell* is the position of the cell, *dir* and *sign* are two arrays which will be filled for every fluid air interface will indicate the direction and the sign toward the air. n is the number of interface and the number of element filled in the array.
  if $cell.$ **GetIsAir**() **then**
    return false ▷ We are an air cell so we are not fluid.
  end if
  $n \leftarrow 0$
  for $i = 1$ **to** dim **do** ▷ For every direction.
    for $s = \pm 1$ **do** ▷ For every sign.
      $dir[n] \leftarrow i$ ▷ We store the actual direction (it will be override if n is not incremented).
      $sign[n] \leftarrow s$ ▷ We store the actual sign (it will be override if n is not incremented).
      $cell2 \leftarrow Cell.$ **GetNeighbour**(i, s)
      if $cell2.$ **IsValid**() **then**
        if $cell2.$ **GetIsAir**() **then**
          $n \leftarrow n + 1$ ▷ We have found an interface increase the number of currently found interface. The *dir* and *sign* arrays have an entry for them that will not be erased because we have increased n.
        end if
      end if
    end for
  end for
  if $n > 0$ **then**
    return true ▷ We have at least one interface, we are a boundary.
  end if
  return false ▷ We have no interface, we are not boundary.
end function

Choose boundary case

We are now able to detect the boundary cells. We now need to recognize the different cases. For this we use the information on the number of neighbors and which neighbors:

- If we have one neighbour we are candidate for a plane case. We need then to try every neighbor along the surface to see if it has the same interface with the same direction.
- If we have two neighbor interfaces of different components, we are in a diagonal plane case. This case is the more easy because we do not need to verify other things.
- If it is not possible we put **NaN** in the given component to know for the rest of the algorithm that this speed is not known.

The application of plane rules is given in section 4.3.1. This is shown in algorithm 27.

Apply 2D plane

To Apply the 2d plane rule we do the following things:

1. We use the divergence free condition 2 times for 3 known speeds and one unknown.
2. We use the last equation (4.31) to find the last unknown.

The divergence part equation is shown in algorithm 28. The plane part is shown in algorithm 29.

Apply 2d 45° plane

In the 45° we simply copy the speed from the opposite known speed component. This is shown in algorithm 30.

Apply set NaN

We need to set to **NaN** all speed component that are not boundary with a fluid cell. This are speed component between two air cells. The **NaN** value will be used in the extrapolation method after to know which speeds are unknow. This is shown in algorithm 31.

NaN extrapolation

We have cells with **NaN** at boundary, and need to extrapolate a speed in the **NaN** in respecting the divergence free condition. For this, we to do the following for every cell.

1. We remember the position of **NaN** value.
2. We extrapolate a speed in **NaN** value.
3. We calculate the divergence in the cell.
4. We distribute the divergence to all **NaN** value.

Algorithm 27 Algorithm that finds in which boundary case we are.

```
1: function ChooseBoundaryCase2d(cell, dir[2dim], sign[2dim], n)  ▷ cell is the position of the cell, dir, sign and n are the same as IsBoundary(cell, dir, sign, n)
2:     if n = 1 then  ▷ Only one interface this is a candidate for a horizontal or vertical plane case.
3:         dir2 ← 1
4:         if dir[0] = 1 then
5:             dir2 ← 2  ▷ dir2 is now the tangent direction.
6:         end if
7:         b = true
8:         for s = ±1 do  ▷ For every sign.
9:             cell2 ← cell.GetNeighbour(dir2, s)
10:            IsBoundary(cell2, dirtemp, signtemp, ntemp)  ▷ We see the interface type of the two neighbor cells.
11:            if ntemp = 1 and dirtemp[0] = dir[0] then  ▷ If we are again a candidate for a horizontal or a vertical plane case.
12:                b ← false  ▷ Set b to indicate that we were able to have a plane with a least one direction.
13:                if i < 0 then  ▷ We change the order of the worker cell depending of the sign (will be useful later to calculate derivative)
14:                    Apply2dPlane(cell2, cell, dir[0], sign[0])
15:                else if i > 0 then ▷ If the two function calls are called, the speed at cell normal to the surface, will be changed two times but with the same expression.
16:                    Apply2dPlane(cell, cell2, dir[0], sign[0])
17:                end if
18:            end if
19:        end for
20:        if b then
21:            cell3 ← cell
22:            if sign[0] = 1 then
23:                cell3 ← cell.GetNeighbour(dir[0], 1)  ▷ Depending on the sign, we need to look the cell below because of the staggered grid.
24:            end if
25:            if not(GetIsSpeedInDomain(cell, dir[0])) then
26:                cell3.SpeedSet(dir[0], NaN)  ▷ If we don't consider the given speed as in the domain we set it to NaN so that a later algorithm can set it to another value.
27:            end if
28:        end if
29:    else if n = 2 then  ▷ We have two neighbors, we are candidate for a oblic plane case.
30:        if dir[0] ≠ dir[1] then  ▷ We test if we are oblic.
31:            sa ← sign[0]  ▷ We store the two sign.
32:            sb ← sign[1]
33:            if dir[0] > dir[1] then  ▷ We reorder the sign in increasing order.
34:                sa ← sign[1]
35:                sb ← sign[0]
36:            end if
37:            Apply2dPlane45(cell, sa, sb)  ▷ We apply the oblic case.
38:        end if
39:    else  ▷ We have none of the known case.
40:        for i = 0 to n do  ▷ For every direction where we have an interface.
41:            cell2 ← cell
42:            if sign[i] = 1 then
43:                cell2 ← cell.GetNeighbour(dir[i], 1)  ▷ Because we are in a staggered grid depending of the sign we take the neighbour.
44:            end if
45:            if not(GetIsSpeedInDomain(cell2, dir[i])) then  ▷ If we are not a boundary speed.
46:                cell2.SpeedSet(dir[i], NaN) ▷ We set the speed to NaN so that later method can change it.
47:            end if
48:        end for
49:    end if
50: end function
```

Algorithm 28 Algorithm to calculate the speed at a given point from the divergence free condition.

function ApplyDiv2D($cell, dir, sign$) ▷ *cell* is the cell were to calculate divergence. *dir* and *sign* indicate which of the 4 speed components we want to calculate.*dir* is the direction, *sign* is $+1$ for the right or above component and -1 for the left and below.
  $cell2 \leftarrow cell.$ **GetNeighbour**$(dir, 1)$ ▷ *cell*2 will be the cell where the unknown speed component lays.
  $v \leftarrow cell.$ **SpeedGet**(dir) ▷ v is the known speed component at opposite of the unknown.
  if $sign = -1$ **then** ▷ Depending of the sign we need to take other cell.
    $v \leftarrow cell2.$ **SpeedGet**(dir)
    $cell2 \leftarrow cell$
  end if
  $dir2 \leftarrow 1$ ▷ $dir2$ is the other direction than dir
  if $dir = 1$ **then**
    $dir2 \leftarrow 2$
  end if
  $v2 \leftarrow cell.$ **SpeedGet**$(dir2)$ ▷ Speed value of the first known speed.
  $v3 \leftarrow cell.$ **GetNeighbour**$(dir2, 1).$ **SpeedGet**$(dir2)$ ▷ Speed value of the second known speed opposite to the first.
  $res \leftarrow v - sign \frac{h.\,\textbf{Get}(dir)}{h.\,\textbf{Get}(dir2)}(v3 - v2)$ ▷ Solution from the divergence free condition to find one unknown from 3 known speeds.
  $cell2.$ **SpeedSet**(dir, res) ▷ Set the new known speed
end function

Algorithm 29 Algorithm to apply the calculation of a 2d plane case.

function Apply2dPlane($cell, cell2, dir, sign$) ▷ *cell* and *cell*2 are two cells in fluid marking the plane boundary. *cell* and *cell*2 are given in increasing order in the normal direction of *dir*. *dir* and *sign* indicate the direction of Air.
  ApplyDiv2d$(cell, dir, sign)$ ▷ We use the divergence condition to find the 4þspeed from 3 known.
  ApplyDiv2d$(cell2, dir, sign)$ ▷ We use the divergence condition to find the 4þspeed from 3 known.
  $dir2 \leftarrow 1$
  if $dir = 1$ **then**
    $dir2 \leftarrow 2$ ▷ $dir2$ is now the tangent direction.
  end if
  $v1 \leftarrow cell.$ **SpeedGet**(dir)
  $v2 \leftarrow cell2.$ **SpeedGet**(dir)
  $v3 \leftarrow cell2.$ **SpeedGet**$(dir2)$
  $v \leftarrow -\frac{sign}{h.\,\textbf{Get}(dir2)} h.$ **Get**$(dir)(v2 - v1) + v3$ ▷ This is equation 4.31 with other notation.
  $cell3 \leftarrow cell2.$ **GetNeighbour**$(dir, sign)$
  $cell3.$ **SpeedSet**$(dir2, v)$ ▷ We set the new speed.
end function

Algorithm 30 Algorithm to apply the 45° plane case

```
1: function Apply2dPlane45(cell ,sa,sb)
2:     v1 ← cell.SpeedGet(1)                          ▷ v1 is the known speed in direction 1.
3:     v2 ← cell.SpeedGet(2)                          ▷ v2 is the known speed in direction 2.
4:     n1 ← cell.GetNeighbour(1, 1)                   ▷ n1 is the cell of the unknown speed in direction 1.
5:     n2 ← cell.GetNeighbour(2, 1)                   ▷ n2 is the cell of the unknown speed in direction 2.
6:     if sa=-1 then  ▷ If the sign in the first direction is different, we change the expression for v1
   and n1.
7:         v1 ← cell.GetNeighbour(1, 1).SpeedGet(1)
8:         n1 ← cell
9:     end if
10:    if sb = −1 then▷ If the sign in the second direction is different, we change the expression for
   v2 and n2.
11:        v2 ← cell.GetNeighbour(2, 1).SpeedGet(2)
12:        n2 ← neigh
13:    end if
14:    n1.SpeedSet(1, v1)        ▷ In the oblic case, we simply copy the speed in opposite direction.
15:    n2.SpeedSet(2, v2)        ▷ In the oblic case, we simply copy the speed in opposite direction.
16: end function
```

Algorithm 31 Algorithm that sets speed component between air cell to **NaN**.

```
1: function ApplySetNan(cell)
2:     if cell.GetIsFluid() then                                  ▷ We only work on air cell.
3:         return
4:     end if
5:     for i = 1 to dim do                                        ▷ For every direction
6:         for s = ±1 do                                          ▷ For every sign
7:             cell2 ← cell.GetNeighbour(i, s);
8:             if cell2.IsValid() then
9:                 if neigh2.GetIsAir() then          ▷ We look that we are between two air cells
10:                    cell3 ← cell2                  ▷ We set cell3 to the cell with the speed component.
11:                    if s = −1 then
12:                        cell3 ← cell
13:                    end if
14:                    cell3.SpeedSet(i, NaN)                  ▷ We set the speed to NaN
15:                end if
16:            else if s = −1 then  ▷ If we have no neighbour in this direction when s = −1 we will
   have a speed component with only a cell in one side.
17:                cell.SpeedSet(i, NaN)
18:            end if
19:        end for
20:    end for
21: end function
```

The important point that makes that we have existence and uniqueness of a solution. Is that we cannot have a **NaN** value which has no non **NaN** neighbour. This is the case because if this happens we consider the given speed component as in the domain. For example the single fluid cell case. Could make this case if we consider the 4 speeds as out of the domain. We have no known speed to extrapolate with. But our conditions are so that in this case we consider the 4 speeds as in the domain. This is shown in Algorithm 32 and 33.

Do 2d

We now put all our pieces together. For every cell we do the following operation:

1. We set to **NaN** speed components that are between two air cells. We need to do it at the beginning because some cases are able to calculate this component, in this case we have no extrapolation because the **NaN** value will become a none **NaN** value.
2. We treat the known boundary condition case.
3. We extrapolate the speed for the **NaN** value at boundary but in having a divergence free speed.

We need to do each step in a separate foreach because we have dependance on the foreach before. But we have no dependance between the same foreach. A foreach can be called in parallel, if we ensure that concurent write at the same speed component of the same value give the given value and not garbage. And if we ensure that we can get and set speed concurently on different speeds component without problem.

Note that in all this foreach loop we do not change the speed component number, only their value. A simple fixed matrix implementation will respect all this condition.

This is shown in algorithm 34.

Conclusion

In conclusion, we have an algorithm of linear complexity (only one level of nested for each) which is parallelizable.

This algorithm is specialized for boundary conditions in straight plane and in corners. For the other case an extrapolation scheme is used. If more accurate boundary conditions are needed, the given case can be added in the choose section.

4.5.4 Extrapolation

We now need to extrapolate **NaN** values from known values which are nearer to the surface. This is done iteratively layer by layer. Exact pseudo code is a little complicated because of the staggered grid. For details look at the algorithms 35 , 36 and 37. And the figure given in the comment.

4.5.5 Interpolation

We will now show the pseudo code for interpolation for N dimension with N-linear interpolation. The principal difficulty is that on a staggered grid the position of known speed varie for all components. The algorithm needed to do the translation is shown in algorithm 39. The interpolation is shown in algorithm 38.

Algorithm 32 Algorithm that extrapolates **NaN** speed component in the boundary (first part).

```
1: function ApplyNanExtrap(cell)
2:     n ← 0
3:     for i = 1 to dim do                                   ▷ For every direction
4:         v1 ← cell.SpeedGet(i)
5:         cell_temp ← cell.GetNeighbour(i, 1)
6:         v2 ← cell_temp.SpeedGet(i)
7:         if v1 = NaN then▷ We are NaN, we stock the direction and sign in tab, where we have a
   vector of vector with 0 the direction 1 the sign.
8:             tab[0][n] ← i
9:             tab[1][n] ← 0
10:            ++n
11:        end if
12:        if v2 = NaN then▷ We are NaN, we stock the direction and sign in tab, where we have a
   vector of vector with 0 the direction 1 the sign.
13:            tab[0][n] ← i
14:            tab[1][n] ← 1
15:            ++n
16:        end if
17:    end for
18:    if n = 0 then                                  ▷ We have no NaN nothing to do. return
19:    end if
20:    for i = 0 to n do          ▷ For every NaN. We set the NaN value from extrapolation.
21:        dir ← tab[0][i]
22:        sign ← tab[1][i]
23:        cell2 ← cell
24:        if sign = 1 then
25:            cell2 ← cell.GetNeighbour(dir, 1)
26:        end if
27:        vtemp ← 0
28:        nb ← 0
29:        for dir2 = 1 to dim do                          ▷ We look all directions.
30:            for s = ±1 do                                ▷ We look all signs.
31:                add ← cell2.SpeedGet(dir)                ▷ We get the speed value.
32:                if add ≠ NaN then                        ▷ If it's not NaN we add it.
33:                    vtemp ← vtemp + add
34:                    ++nb
35:                end if
36:            end for
37:        end for
38:        if nb ≠ 0 then                                   ▷ If we have found a neighbor speed with
   non NaN speed. Note if we put NaN on boundary speed and doesn't force boundary condition
   when we are too thin, this case is always true.
39:            vtemp ← vtemp/nb                             ▷ We calculate the speed as the mean.
40:        end if
41:        cell2.SpeedSet(dir, vtemp)                       ▷ We set the speed.
42:    end for
```

Algorithm 33 Algorithm that extrapolates **NaN** speed component in the boundary (second part).

```
43:     div ← 0                                          ▷ We now calculate the divergence of speed.
44:     for i = 1 to dim do                              ▷ For every direction, iteratively calculate the divergence.
45:         v1 ← cell.SpeedGet(i)
46:         v2 ← cell.GetNeighbour(i, 1).SpeedGet(i)
47:         div ← div + 1/(h.Get(i)) * (v2 − v1)
48:     end for
49:     cor ← div/n                                      ▷ The divergence is shared between NaN speed value.
50:     for i = 0 to n do                                ▷ For every NaN speed.
51:         dir ← tab[0][i]
52:         sign ← tab[1][i]
53:         cell2 ← cell
54:         if sign = 1 then
55:             cell2 ← cell.GetNeighbour(dir, 1)
56:         end if
57:         cell2.SpeedSet(dir, neigh2.SpeedGet(dir) − (2 * sign − 1) * h.Get(dir) * cor)▷ Correct
    the speed from the divergence.
58:     end for
59: end function
```

Algorithm 34 Algorithm which calculates the extrapolation given by boundary condition.

```
1:  procedure Do2dBoundaryExtrapolation
2:      for all cell ∈ Grid do                           ▷ We don't exchange state between two
    function calls because every function call substitute its needed unknown from known speed. But
    we posibly set the same value (from the same mathematical expression) in two function calls. If
    concurrent writting of the same value give the said value. Then this loop is thread safe.
3:          ApplySetNan(cell)
4:      end for
5:      for all cell ∈ Grid do                           ▷ We don't exchange state between two
    function calls because every function call substitute its needed unknown from known speed. But
    we posibly set the same value (from the same mathematical expression) in two function calls. If
    concurrent writting of the same value give the said value. Then this loop is thread safe.
6:          if IsBoundary(cell, dir, sign, n) then
7:              ChooseBoundaryCase2d(cell, dir, sign, n)
8:          end if
9:      end for
10:     for all cell ∈ Grid do                           ▷ We don't exchange state between two
    function calls because every function call substitute its needed unknown from known speed. But
    we posibly set the same value (from the same mathematical expression) in two function calls. If
    concurrent writting of the same value give the said value. Then this loop is thread safe.
11:         if IsBoundary(cell) then
12:             ApplyNanExtrap(cell)
13:         end if
14:     end for
15: end procedure
```

Algorithm 35 Algorithm which calculates the extrapolation given by boundary conditions. First Part, with the header and the first big loop.

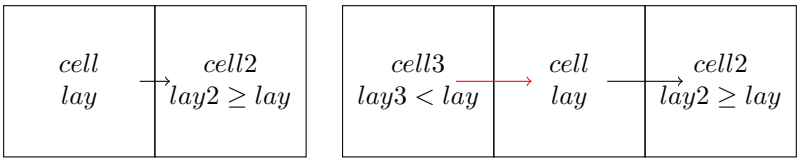
```
1: procedure AlgorithmsExtrapolateNan
2:     set ← NewSet()
3:     for all cell ∈ Grid do
4:         if cell.LayerGet() = 1 then                ▷ Cells with 1 as layer are air cells at the boundary.
5:             stack.insert(it)                       ▷ We add this cell in the list of cells to treat.
6:         end if
7:     end for
8:     lay ← 1
9:     loop
10:        set2 ← NewSet()
11:        for all it ∈ set do                        ▷ We treat all cell in the list of cell to treat.
12:            for i = 1 to dim do                    ▷ For every direction.
13:                cell2 ← cell.GetNeighbour(i, 1)    ▷ We look cell to the right.
14:                if cell2.IsValid() and not(cell2.LayerGet() = −1) then   ▷ If we can use cell2.
15:                    if cell2.LayerGet() ≥ lay then   ▷ Test if configuration like fig.4.4a .
16:                        if cell2.LayerGet() > lay then   ▷ A cell of bigger layer add it in set.
17:                            set2.insert(cell2)
18:                        end if
19:                        if cell2.SpeedGet(i) = NaN then   ▷ We only treat speed that are NaN.
20:                            n ← 0                  ▷ Will count the number of neighbour
21:                            val ← 0                ▷ Will contain the sum of the neighbour
22:                            for j = 1 to dim do    ▷ For every dimension
23:                                for s2 = ±1 do     ▷ For every sign
24:                                    if s2 = −1 and j = i then   ▷ Test if configuration like fig. 4.4b.
25:                                        cell3 ← cell.GetNeighbour(i, −1)
26:                                        if cell3.IsValid() and cell3.LayerGet() ≠ −1 then
27:                                            if cell3.LayerGet() < lay then
28:                                                n ← n + 1
29:                                                val ← val + cell.SpeedGet(i)
30:                                            end if
31:                                        end if
32:                                    else if s2 = 1 and j = i then ▷ Test if configuration like fig. 4.4c.
33:                                        cell3 ← cell2.GetNeighbour(i, 1)
34:                                        if cell3.IsValid() and not(cell3.LayerGet() = −1) then
35:                                            if cell3.LayerGet() < lay then
36:                                                n ← n + 1
37:                                                val ← val + cell3.SpeedGet(i)
38:                                            end if
39:                                        end if
40:                                    else   ▷ We are no more parallel to the direction, as in figure 4.4d.
41:                                        cell3 ← cell2.GetNeighbour(j, s2)
42:                                        cell4 ← cell.GetNeighbour(j, s2)
43:                                        if cell3.IsValid()and cell4.IsValid()and cell3.LayerGet() ≠
   −1 and neigh4.LayerGet() ≠ −1 then
44:                                            if cell3.LayerGet() < layor cell4.LayerGet() < lay then
45:                                                n ← n + 1
46:                                                val ← val + cell3.SpeedGet(i)
47:                                            end if
48:                                        end if
49:                                    end if
50:                                end for
51:                            end for
52:                            cell2.SpeedSet(i, val/n)   ▷ Set the speed to the mean of the previous
   case that apply
53:                        end if
54:                    end if
55:                end if
```

Algorithm 36 Algorithm which calculate the extrapolation given by boundary conditions. Second Part, with the second big loop.

```
56:                 cell2 ← neigh. GetNeighbour(i, −1)                ▷ We look the cell to the left.
57:                 if cell2. IsValid() and cell2. LayerGet ≠ −1 then
58:                     if cell2. LayerGet ≥ lay then  ▷ We look if we are in the situation of fig. 4.5a.
59:                         if cell2. LayerGet() > lay then  ▷ The cell has bigger layer, we add to the set.
60:                             set2. insert(cell2)
61:                         end if
62:                         if cell. SpeedGet(i) = NaN then          ▷ We only consider NaN speed.
63:                             n ← 0                                ▷ Will count the number of neighbour
64:                             val ← 0                              ▷ Will contain the sum of the neighbor
65:                             for j = 1 to dim do
66:                                 for s2 = ±1 do
67:                                     if s2 = 1 and j = i then  ▷ If we are in the situation of fig. 4.5b.
68:                                         cell3 ← cell. GetNeighbour(j, 1)
69:                                         if cell3. IsValid() and cell3. LayerGet() ≠ −1 then
70:                                             if cell3. LayerGet() < lay then
71:                                                 n ← n + 1
72:                                                 val ← val + cell3. SpeedGet(j)
73:                                             end if
74:                                         end if
75:                                     else if s2 = −1 and j == i then▷ If we are in the situation of fig. 4.5c.
76:                                         cell3 ← cell2. GetNeighbour(j, −1)
77:                                         if cell3. IsValid() and cell3. LayerGet() ≠ −1 then
78:                                             if cell3. LayerGet() < lay then
79:                                                 n ← n + 1
80:                                                 val ← val + cell2. SpeedGet(j)
81:                                             end if
82:                                         end if
83:                                     else                          ▷ If we are in the situation of fig. 4.5d .
84:                                         cell3 ← cell2. GetNeighbour(j, s2)
85:                                         cell4 ← cell. GetNeighbour(j, s2)
86:                                         if cell3. IsValid()and cell4. IsValid()and cell3. LayerGet() ≠ −1 and cell4. LayerGet() ≠ −1 then
87:                                             if cell3. LayerGet() < layor cell4. LayerGet() < lay then
88:                                                 n ← n + 1
89:                                                 val ← val + cell4. SpeedGet(i)
90:                                             end if
91:                                         end if
92:                                     end if
93:                                 end for
94:                             end for
95:                             cell. SpeedSet(i, val/n)              ▷ Set the speed to the mean value.
96:                         end if
97:                     end if
```

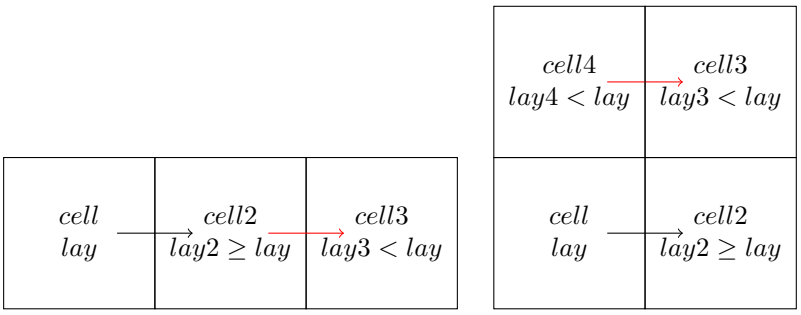
Algorithm 37 Algorithm which calculate the extrapolation given by boundary conditions. Third Part, with the thirth big loop and footer.

```
98:                 else ▷ In this scope cell2 muss not be used. cell2 doesn't exist. We are in the situation of fig. 4.6a.
99:                     if cell.SpeedGet(i) = NaN then
100:                        n ← 0
101:                        val ← 0
102:                        for j = 1 to dim do
103:                            for s2 = ±1 do
104:                                if s2 = 1 and j = i then ▷ If we are in the situation of fig. 4.6b
105:                                    cell3 ← cell.GetNeighbour(j, 1)
106:                                    if cell3.IsValid() and cell3.LayerGet() ≠ −1 then
107:                                        if cell3.LayerGet() < lay then
108:                                            n ← n + 1
109:                                            val ← val + cell3.SpeedGet(j)
110:                                        end if
111:                                    end if
112:                                else if s2 = −1 and j = i then ▷ If we are in the situation of fig. 4.6c. We need to do nothing because we cannot acces value of cell2 because it don't exist.
113:                                else ▷ If we are in the situation of fig. 4.6d
114:                                    cell4 = cell.GetNeighbour(j, s2)
115:                                    if cell4.IsValid() and cell4.LayerGet() ≠ −1 then
116:                                        cell3 ← cell4.GetNeighbour(i, −1)
117:                                        if cell3.IsValid() and cell3.LayerGet() ≠ −1 then
118:                                            if cell3.LayerGet() < lay or cell4.LayerGet() < lay then
119:                                                n ← n + 1
120:                                                val ← val + cell4.SpeedGet(i)
121:                                            end if
122:                                        end if
123:                                    end if
124:                                end if
125:                            end for
126:                        end for
127:                        cell.SpeedSet(i, val/n) ▷ We set the speed to the mean of the neighbor.
128:                    end if
129:                end if
130:            end for
131:        end for
132:        lay ← lay + 1 ▷ We increment layer.
133:        set ← set2 ▷ We swap the old set with the new one.
134:        if set.empty() then ▷ If we have a new empty set exit the algorithm. This is the only mean to exit this loop.
135:            break
136:        end if
137:    end loop
138: end procedure
```



(a) The black speed component show a case considered when *lay* has a given value.

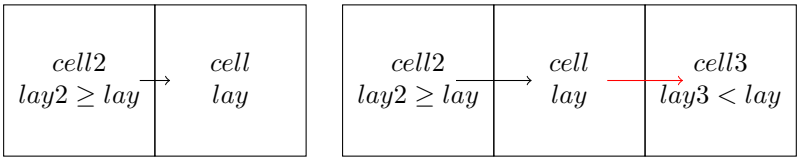
(b) This is the case where the read is done in the same direction in negatif sign of the write.



(c) This is the case where the read is done in the same direction in positif sign of the write.

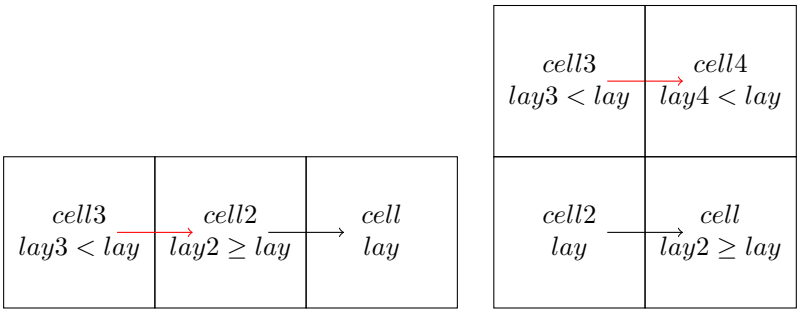
(d) This is the case where the read is done in a different direction of the write.

Figure 4.4: These are the cases used in algorithm 35. These are cases where the speed component is right the considered cell. In black is the speed component to set, in red is the speed component to read from.



(a) The black speed component show a case considered when *lay* has a given value.

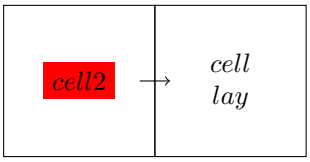
(b) This is the case where the read is done in the same direction in positif sign of the write.



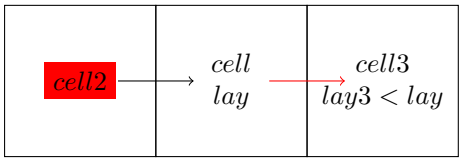
(c) This is the case where the read is done in the same direction in negatif sign of the write.

(d) This is the case where the read is done in a different direction of the write.

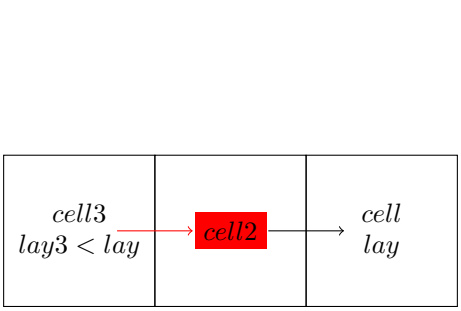
Figure 4.5: These are the cases used in algorithm 36. These are cases where the speed component is left the considered cell. In black is the speed component to set, in red is the speed component to read from.



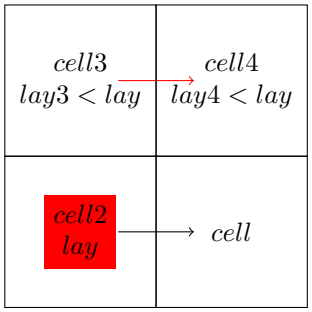
(a) The black speed component show a case considered when lay has a given value.



(b) This is the case where the read is done in the same direction in positif sign of the write.



(c) This is the case where the read is done in the same direction in negatif sign of the write.



(d) This is the case where the read is done in a different direction of the write.

Figure 4.6: These are the cases used in algorithm 37. These are the cases where the speed component is left the considered cell but where the left cell doesn't exist. In black is the speed component to set, in red is the speed component to read from. The cell marked in red doesn't exist, we cannot access left and below speed of them.

Algorithm 38 Algorithm which calculate recursively the interpolation for scaled position in the square in range $[0,1] \times [0,1]$.

1: **function** GetSpeedImpl($posscal, i, cell, k$) ▷ Get the interpolated speed in direction k at position $posscal$ for cell $cell$. i is the recursif parameter it's first value should be 1.
2: $cell2 \leftarrow cell.$ **GetNeighbour**$(i, 1)$ ▷ We set $cell2$ to access speed in the other side in direction i.
3: **if** $i < dim$ **then** ▷ We are not in a terminal case, we do a linear interpolation in direction i and recurse the linear interpolation in the rest of the dimension. The cell variable $cell2$ is used because it's "origine" of the second function call.
4: $ret \leftarrow (1 - posscal.$ **Get**$(i)) \cdot$ **GetSpeedImpl**$(posscal, i + 1, cell, k) + posscal.$ **Get**$(i) \cdot$ **GetSpeedImpl**$(posscal, i + 1, cell2, k)$
5: **return** ret
6: **else** ▷ We are in a terminal case. We acess the speed and do the linear interpolation.
7: $ret \leftarrow (1 - posscal.$ **Get**$(i)) * (cell.$ **SpeedGet**$(k)) + posscal.$ **Get**$(i) * (cell2.$ **SpeedGet**$(k))$
8: **return** ret
9: **end if**
10: **end function**

Algorithm 39 Algorithm which interpolate the speed at a given position.

1: **function** GETSPEED(pos)
2: **for** $i = 1$ **to** dim **do**
3: $posdelta.\mathbf{Set}(i, \frac{pos.\mathbf{Get}(i)}{h.\mathbf{Get}(i)})$ ▷ Scale the position.
4: $key0.\mathbf{Set}(i, round(pos_delta.GetRef(i)))$ ▷ We round to the nearest integer to find in which integer cell is the particle.
5: $posdelta.\mathbf{Set}(i, posdelta.\mathbf{Get}(i) - key0.\mathbf{Get}(i))$ ▷ Because we have rounded to the nearest integer the value of *posdelta* are in $[-0.5, 0.5[$ (the exact up or down rounding depend on the specification of the rounding function).
6: **end for**
7: **for** $k = 1$ **to** dim **do** ▷ For every speed component.
8: $posdelta.\mathbf{Set}(k, posdelta.\mathbf{Get}(k) + 0.5)$ ▷ Because we are in a staggered grid the position is translated. The 0 vector is at cell center. The range of $posdelta.\mathbf{Get}(k)$ is in range $[0, 1[$.
9: **for** $j = 1$ **to** dim **do**
10: **if** $j \neq k$ **then** ▷ We loop on every speed component different than the used one.
11: **if** $posdelta.\mathbf{Get}(j) < 0$ **then** ▷ $posdelta.\mathbf{Get}(j)$ is in range $[-0.5, 0.5[$.
12: $key.\mathbf{Set}(j, key0.\mathbf{GetRef}(j) - 1)$ ▷ We go to the cell below because we have rounded to the cell above.
13: $posdelta2.\mathbf{Set}(j, 1 + posdelta.\mathbf{Get}(j))$ ▷ We change the $posdelta2$ to the translation. $posdelta2.\mathbf{Get}(j)$ is now in range $[0, 1[$.
14: **else**
15: $key.\mathbf{Set}(j, key0.\mathbf{Get}(j))$
16: $posdelta2.\mathbf{Set}(j, posdelta.\mathbf{Get}(j))$
17: **end if**
18: **else**
19: $key.\mathbf{Set}(j, key0.\mathbf{Get}(j))$
20: $posdelta2.\mathbf{Set}(j, posdelta.\mathbf{Get}(j))$
21: **end if**
22: **end for**
23: $cell \leftarrow Grid[key]$ ▷ We get the cell at the calculated key.
24: $ret.\mathbf{Set}(k, \mathbf{GetSpeedImpl}(pos_delta2, 1, cell, k))$ ▷ We call the interpolation routine and set the result in the return vector.
25: $posdelta.\mathbf{Set}(k, posdelta.\mathbf{Get}(k) - 0.5)$
26: **end for**
27: **return** ret
28: **end function**

4.5.6 Integration

For the non split case the acceleration is given by (4.16). We use to avoid to rewrite an extrapolation fonction a method **SetSpeedToAcceleration**(*bool*) which toggles that the method to set the speed and get the speed act on acceleration instead. This is shown in algorithm 40

We now integrate with the Runge-Kutta method, the difference with respect to the case in section 3.4.4 is that we now have particles. We need to do the same that we have done having a copy of all particle positions. This is shown in algorithm 41 and 42.

A complete iteration consists of Algorithm 43

Algorithm 40 Calculate the acceleration for the non split method equation (4.16).

```
1: procedure CalculateAcceleration2
2:     InitializationRungeKutta()
3:     Do2dBoundaryExtrapolation()
4:     AlgorithmsExtrapolateNan()
5:     CalculateAcceleration()
6:     SetSpeedToAcceleration(true)
7:     Do2dBoundaryExtrapolation()
8:     AlgorithmsExtrapolateNan()
9:     SetSpeedToAcceleration(false)
10: end procedure
```

Algorithm 41 Algorithm that integrate with the Runge Kutta method first part.

1: **procedure** RungeKutta
2: **for all** $it \in Grid$ **do**
3: it. **AccelerationSet**(0) ▷ We set the acceleration to 0. So that we can after calculate the acceleration.
4: **end for**
5: **CalculateAcceleration**() ▷ This function will add the acceleration to the acceleration array.
6: **for all** $it \in Grid$ **do**
7: it. **GetArray**(2). **SpeedSet**(it. **GetArray**(0). **SpeedGet**() + it. **AccelerationGet**() $\cdot$ dt $\cdot$ 0.5) ▷ Array 2 is set to the speed for the next evaluation. We do not use array 0 because we need it.
8: it. **GetArray**(1). **SpeedSet**(it. **GetArray**(0). **SpeedGet**()+it. **AccelerationGet**() $\cdot$ $\frac{dt}{6}$) ▷ Array 1 is set to the accumulation of the result.
9: **end for**
10: **for all** $it \in Particles$ **do**
11: it. **GetList**(2). **PositionSet**(it. **GetList**(0). **PositionGet**() + it. **GetSpeed**(it.*PositionGet*()) $\cdot$ dt $\cdot$ 0.5) ▷ List 2 is set to the position for the next evaluation. We do not use list 0 because we need it.
12: it. **GetList**(1). **PositionSet**(it. **GetList**(0). **PositionGet**() + it. **GetSpeed**(it.*PositionGet*()) $\cdot$ $\frac{dt}{6}$) ▷ List 1 is set to the accumulation of the result.
13: **end for**
14: **for all** $it \in Grid$ **do**
15: it. **AccelerationSet**(0)
16: **end for**
17: **SetArraySpeed**(2) ▷ The speed for the next evaluation is in array 2. We set array 2 as default.
18: **SetListPos**(2) ▷ The position of particle for the next evaluation is in list 2. We set list 2 as default.
19: $t \leftarrow t + 0.5 \cdot dt$
20: **CalculateAcceleration**()
21: **for all** $it \in Grid$ **do**
22: it. **GetArray**(2). **SpeedSet**(it. **GetArray**(0). **SpeedGet**()+it. **AccelerationGet**() $\cdot$ $\frac{dt}{2}$) ▷ Array 2 is set to the speed for the next evaluation. We do not use array 0 because we need it.
23: it. **GetArray**(1). **SpeedSet**(it. **GetArray**(1). **SpeedGet**()+it. **AccelerationGet**() $\cdot$ $\frac{dt}{3}$) ▷ Array 1 is set to the accumulation of the result.
24: **end for**
25: **for all** $it \in Particles$ **do**
26: it. **GetList**(2). **PositionSet**(it. **GetList**(0). **PositionGet**() + it. **GetSpeed**(it.*PositionGet*()) $\cdot$ $\frac{dt}{2}$) ▷ List 2 is set to the position for the next evaluation. We do not use list 0 because we need it.
27: it. **GeList**(1). **PositionSet**(it. **GetList**(1). **PositionGet**() + it. **GetSpeed**(it.*PositionGet*()) $\cdot$ $\frac{dt}{3}$) ▷ List 1 is set to the accumulation of the result.
28: **end for**
29: **for all** $it \in Grid$ **do**
30: it. **AccelerationSet**(0)
31: **end for**
32: **CalculateAcceleration**()
33: **for all** $it \in Grid$ **do**
34: it. **GetArray**(2). **SpeedSet**(it. **GetArray**(0). **SpeedGet**()+it. **AccelerationGet**() $\cdot$ dt)
35: it. **GetArray**(1). **SpeedSet**(it. **GetArray**(1). **SpeedGet**()+it. **AccelerationGet**() $\cdot$ $\frac{dt}{3}$)
36: **end for**
37: **for all** $it \in Particles$ **do**
38: it. **GetList**(2). **PositionSet**(it. **GetList**(0). **PositionGet**() + it. **GetSpeed**(it.*PositionGet*()) $\cdot$ dt)
39: it. **GetList**(1). **PositionSet**(it. **GetList**(1). **PositionGet**() + it. **GetSpeed**(it.*PositionGet*()) $\cdot$ $\frac{dt}{3}$)
40: **end for**
41: **for all** $it \in Grid$ **do**
42: it. **AccelerationSet**(0)
43: **end for**

Algorithm 42 Algorithm that integrate with the Runge Kutta method second part.

44: $t \leftarrow t + \frac{dt}{2}$
45: **CalculateAcceleration**()
46: **for all** $it \in Grid$ **do**
47: $it.$**GetArray**(0).**SpeedSet**($it.$**GetArray**(1).**SpeedGet**()+$it.$**AccelerationGet**() $\cdot$ $\frac{dt}{6}$) ▷ Array 0 now contain the new speed.
48: **end for**
49: **for all** $it \in Particles$ **do**
50: $it.$**GetList**(0).**PositionSet**($it.$**GetList**(1).**PositionGet**() + $it.$**GetSpeed**($it.PositionGet()$) $\cdot$ $\frac{dt}{6}$) ▷ Array 0 now contain the new speed.
51: **end for**
52: **SetArraySpeed**(0) ▷ We set the default array again to 0 to be ready for the next time step.
53: **SetListPosition**(0) ▷ We set the default list again to 0 to be ready for the next time step.
54: **end procedure**

Algorithm 43 Algorithm one iteration.

1: **procedure** ITERATION
2: **Initialisation**()
3: **RungeKutta**()
4: **end procedure**

4.6 Numerical experiments

4.6.1 One particle fall

We have discussed this case qualitatively, we will now simulate it numerically.

We take as initial condition a single particle with 0 speed everywhere without viscosity falling from gravity.

The analytical solution is for the continuous problem given by:

$$v = -gt, \tag{4.122}$$

$$x = x_0 - \frac{1}{2}gt^2. \tag{4.123}$$

If we add viscosity the same is true.

Result

We see in figure 4.7 and 4.8 that the non split method coincides with the analytical solution for the integration method Runge-Kutta of order 4. The reason is that the analytical solution is a polynomial of order 2.

In contrast, because of the splitting, a splitting method will incorrectly approximate the result, because it is of order 1 at general.

When we have big time steps like $\alpha = 2$ the non split method behaves better. But it is important to recall the results on fixed grid, that having a too big time step make diverge.

4.6.2 Lateral jet

We consider a continuous source of speed in the x direction, with gravity in the y direction. The analytical solution can be found without consideration of the boundary

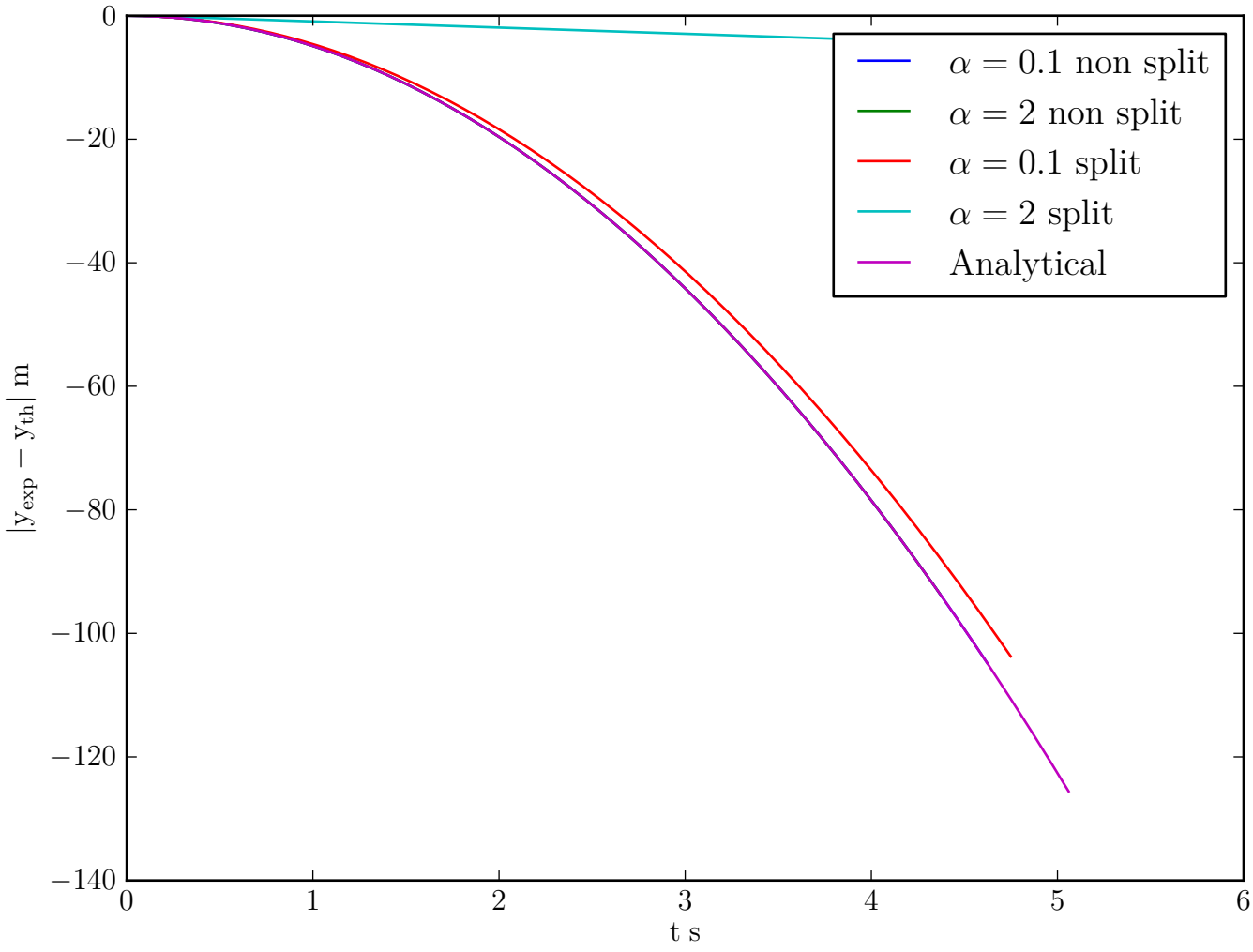


Figure 4.7: Comparison between splitting and not splitting for a single particle problem. The integration method is Runge-Kutta of order 4 for the two case. We see that the non split method is a very good approximation to the analytical solution. Instead the split method and is different than the analytical solution. For the two methods, we have used $\Delta x = 0.01$. The time step limiter was a maximal time step $dt_{max} = 0.01$.

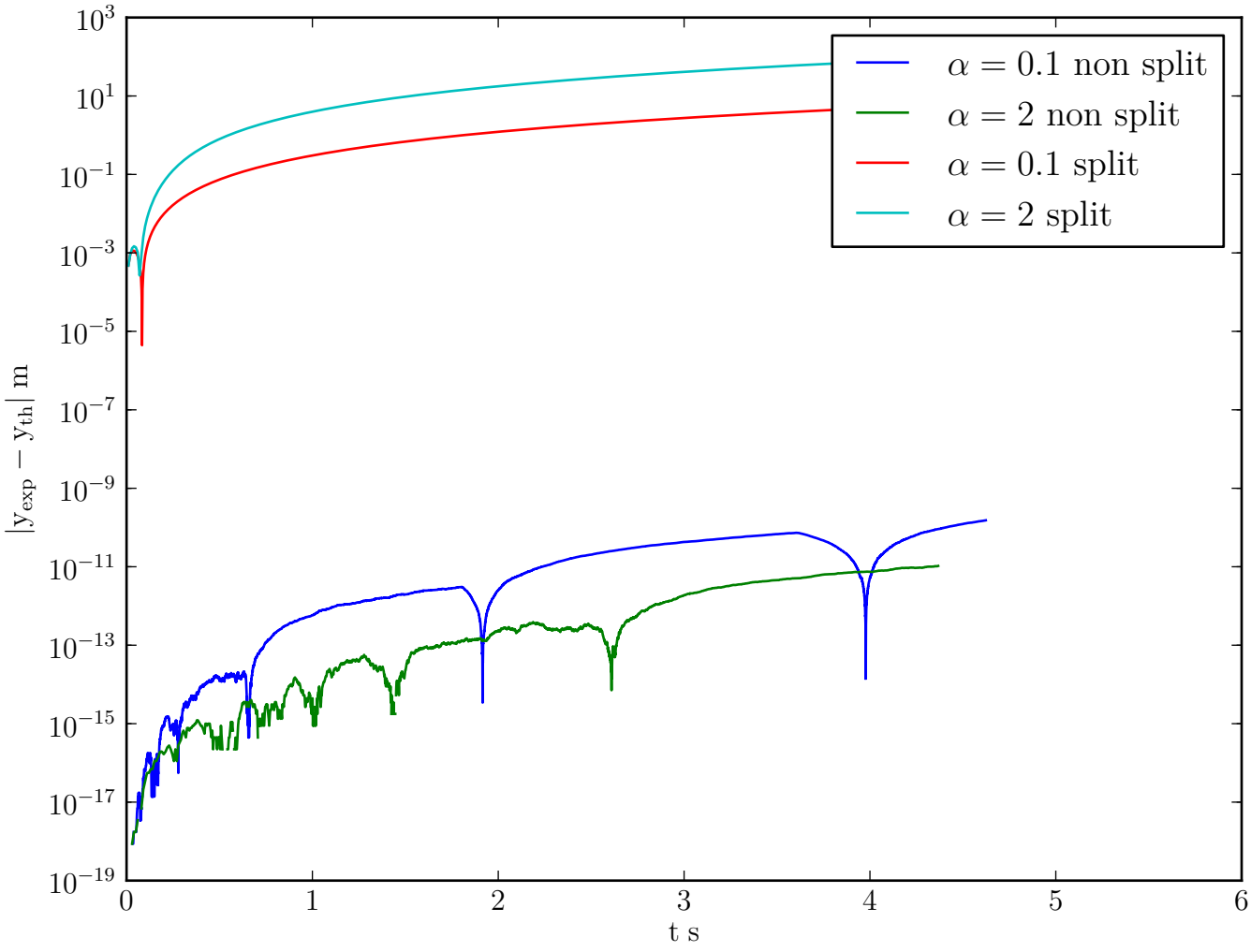


Figure 4.8: Comparison between splitting and not splitting for a single particle problem. The integration method is Runge-Kutta of order 4 for the two case. We see that the non split method is a very good approximation to the analytical solution. Instead the split method and is different than the analytical solution. For the two methods, we have used $\Delta x = 0.01$. The time step limiter was a maximal time step $dt_{max} = 0.01$. We do not know why we have small jump in precision for split method.

condition.

A single particle with initial x speed v_0 which go from y position y_0 and with initial time t_0.

The speed is:

$$v_x = v_0, \tag{4.124}$$

$$v_y = -\frac{g}{2}(t - t_0)^2. \tag{4.125}$$

The position of the particle is then:

$$x = v_0(t - t_0), \tag{4.126}$$

$$y = y_0 - \frac{g}{2}(t - t_0)^2. \tag{4.127}$$

We will now try to define the speed in a particle manner. The time is given by:

$$t - t_0 = \frac{x}{v_0}. \tag{4.128}$$

The speed is then given by:

$$v_x = v_0, \tag{4.129}$$

$$v_y = -\frac{g}{2}\frac{x^2}{v_0}. \tag{4.130}$$

This is divergence free. So we are a solution of the Navier-Stokes equation.

But we does not respect the boundary condition.

We cannot know apriori if the numerical solution with correct boundary condition will respect this solution. But we can expect that a perturbation at boundary that will respect the boundary condition, will rest in the boundary and not enter in the fluid. This approximation is called Boundary Layer.

Results

For the numerical method, we have added a small perturbation in the initial x speed.

We see in figure 4.9, 4.11 and 4.13 that lateral speed are smoothed by the viscosity and is quasi constant which is the analytical solution.

The boundary of the analytical solution is near to the numerical solution. Only the end of the fluid does not follow the analytical solution. This indicates that boundary layers are correctly approximated.

4.6.3 Jet

We now try to simulate the same problem as the lateral jet but in the same direction of gravity. We have no analytical solution in this case because the trivial particle solution is no more divergence free. Pressure is needed to push the fluid up.

We have done two final numerical runs of this problem. All numerical run were done with 50 cells at the basis. With a spacing of 0.0002 m. The viscosity is $\nu = 1.307 \cdot 10^{-6}$ m^2/s which is the viscosity of water. We use $\alpha = 0.5$. We use the non split rk4 integration method. Particles which go below the 0 altitude are supressed.

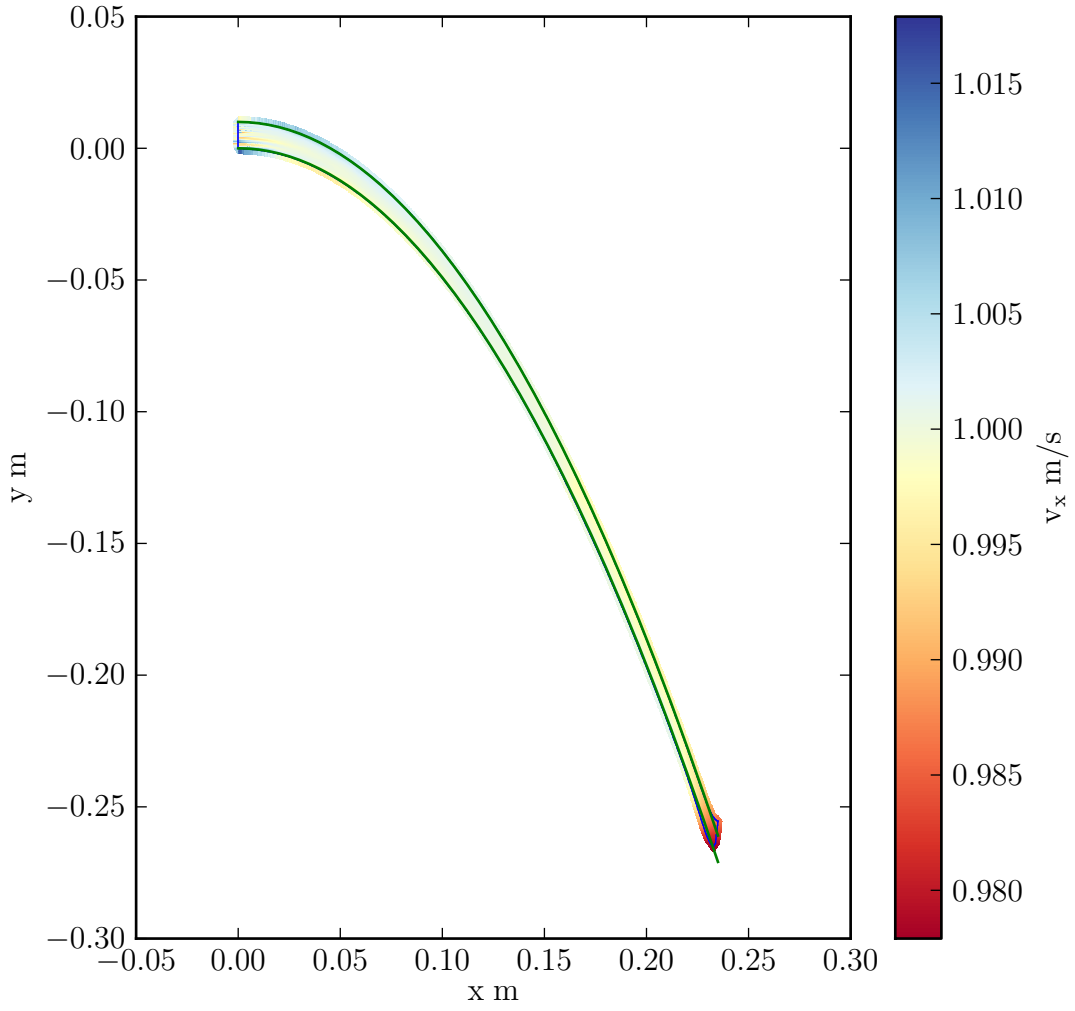


Figure 4.9: Non split result with Runge-Kutta rk4 with $\nu = 1.307\text{e}-6\ \text{m}^2/\text{s}$, $\alpha = 0.5$. This is the x speed. The green line is the analytical solution boundary for the result without boundary condition. The blue line is the calculate boundary. We see that except at the end, the two coincide. At interior the speed take the analytical value which is constant and has a little jump in the boundary. The perturbation at initial condition are smirred by the viscosity.

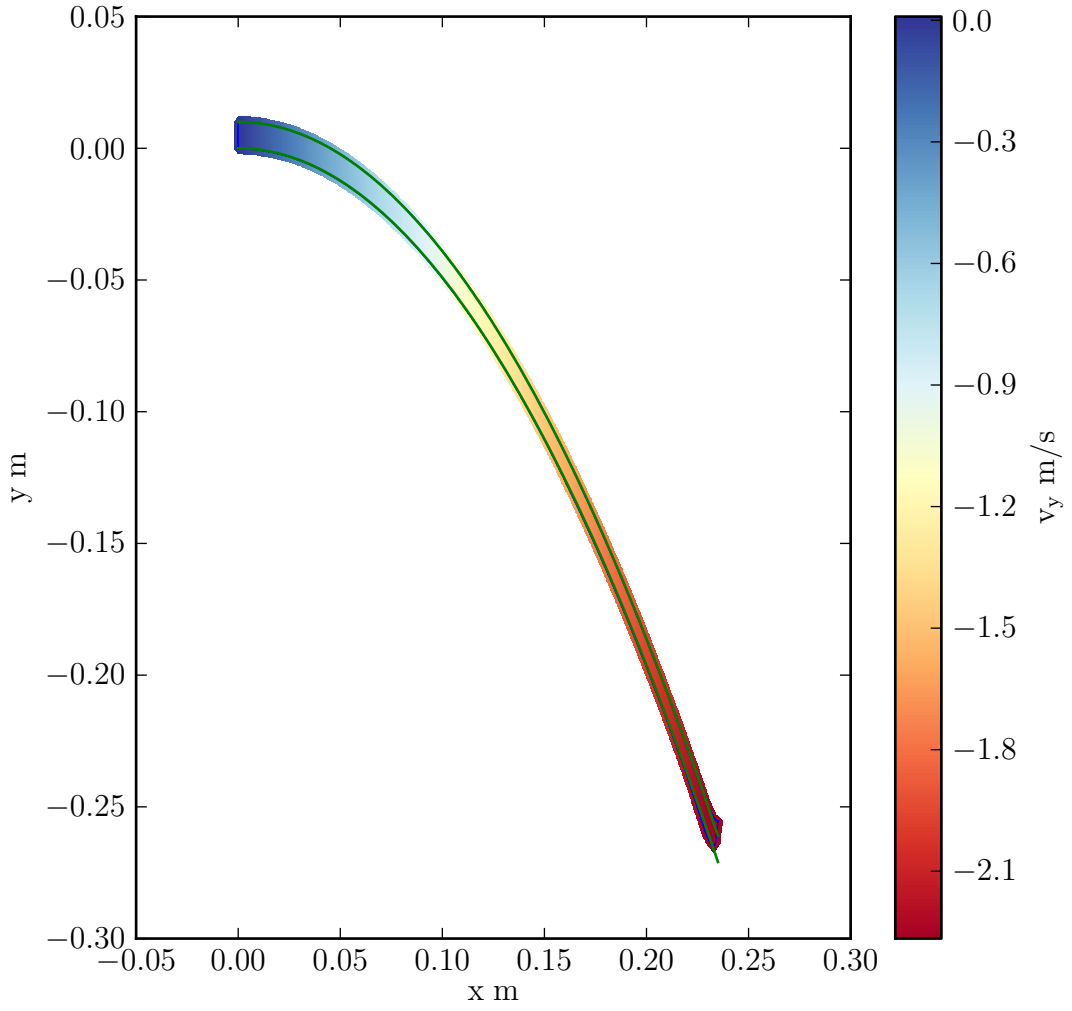


Figure 4.10: Non split result with Runge-Kutta rk4 with $\nu = 1.307\text{e}-6\ \text{m}^2/\text{s}$, $\alpha = 0.5$. This is the y speed. The green line is the analytical solution boundary for the result without boundary condition. The blue line is the calculate boundary. We see that except at the end, the two coincide. The y speed is proportional to the x^2 position, which coincide with the analytical result. The perturbation at initial condition are smirred by the viscosity.

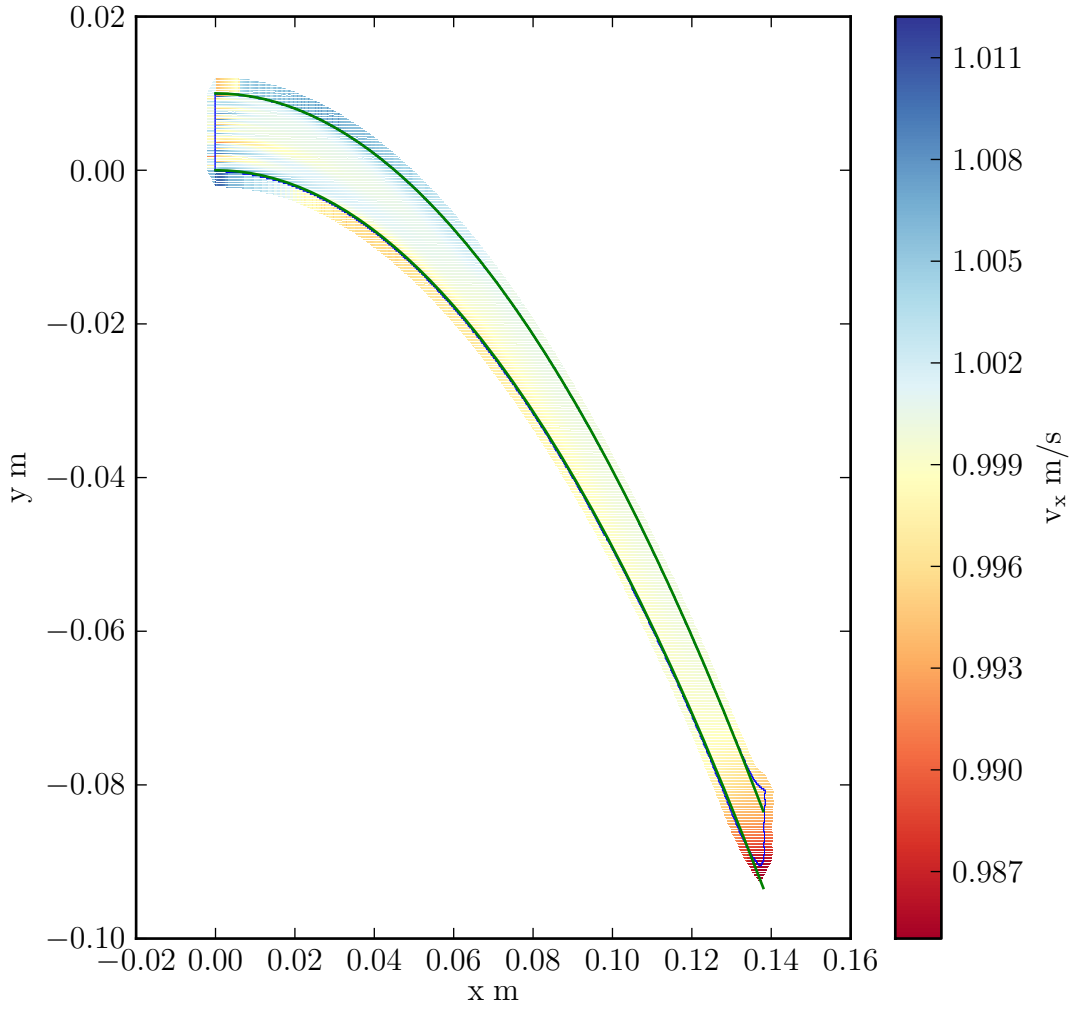


Figure 4.11: Split result with Runge-Kutta rk4 with $\nu = 1.307\text{e} - 6$ m^2/s, $\alpha = 0.5$. This is the x speed. The green line is the analytical solution boundary for the result without boundary condition. The blue line is the calculate boundary. We see that except at the end, the two coincide. At interior the speed take the analytical value which is constant and has a little jump in the boundary. The perturbation at initial condition are smirred by the viscosity.

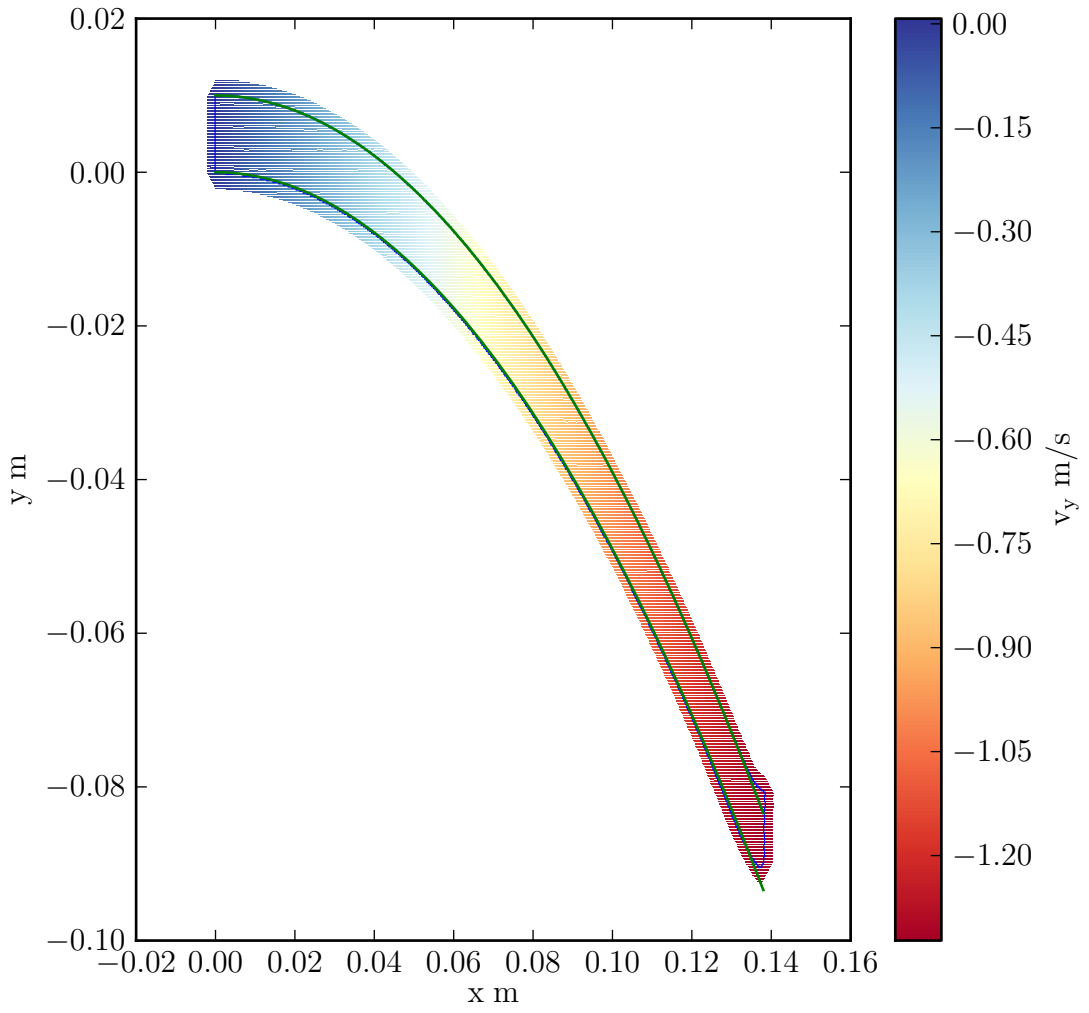


Figure 4.12: Split result with Runge-Kutta rk4 with $\nu = 1.307\text{e}-6\ \text{m}^2/\text{s}$, $\alpha = 0.5$. This is the y speed. The green line is the analytical solution boundary for the result without boundary condition. The blue line is the calculate boundary. We see that except at the end, the two coincide. The y speed is proportional to the x^2 position, which coincide with the analytical result. The perturbation at initial condition are smirred by the viscosity.

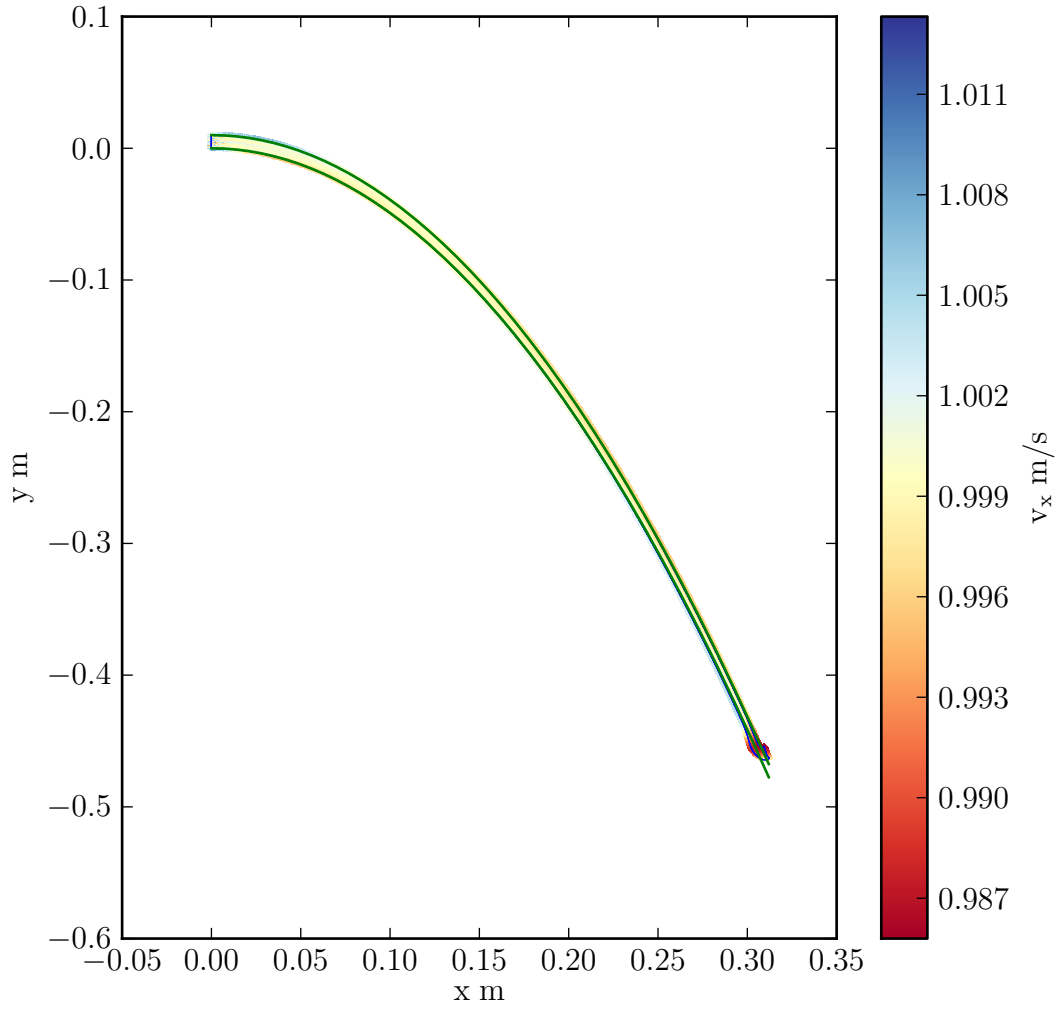


Figure 4.13: Split result with Euler with $\nu = 1.307\text{e}-6\ \text{m}^2/\text{s}$, $\alpha = 0.5$. This is the x speed. The green line is the analytical solution boundary for the result without boundary condition. The blue line is the calculate boundary. We see that except at the end, the two coincide. At interior the speed take the analytical value which is constant and has a little jump in the boundary. The perturbation at initial condition are smirred by the viscosity.

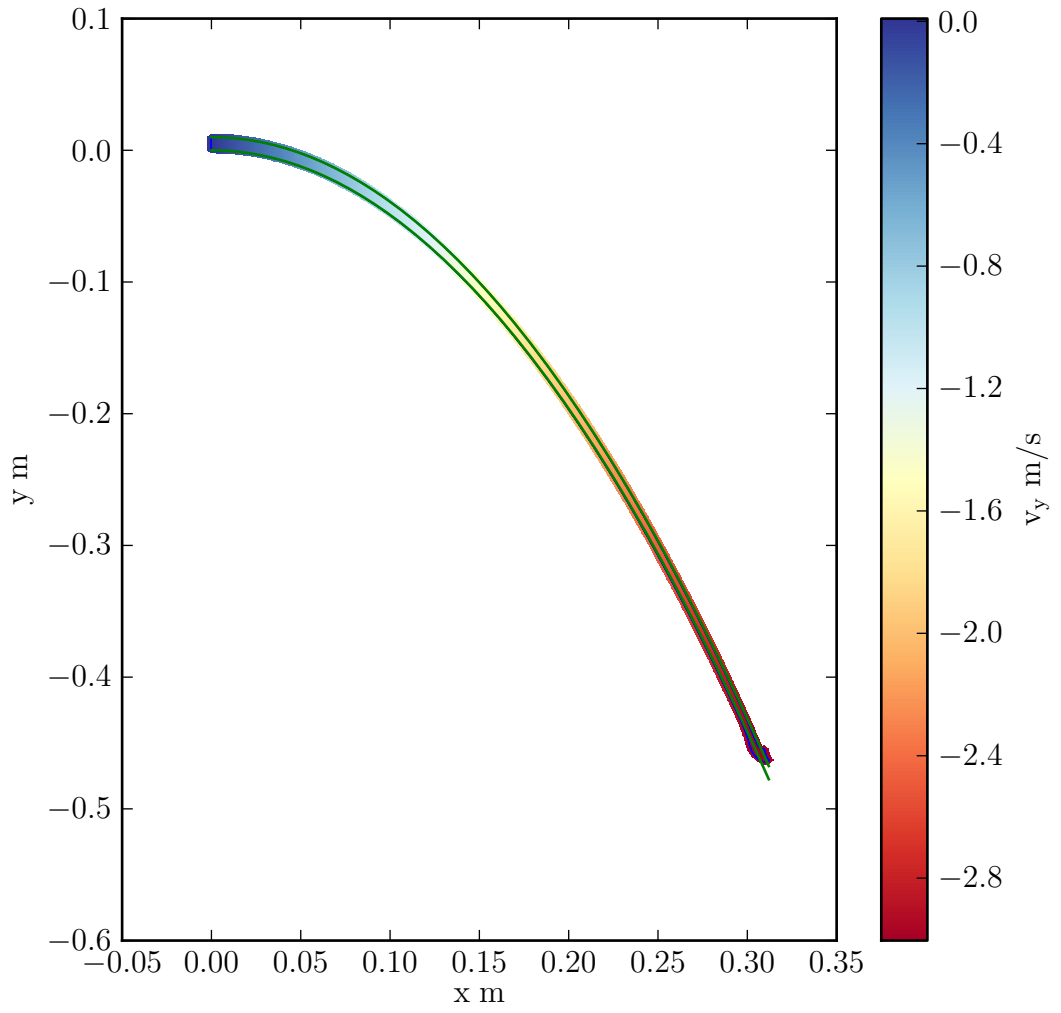


Figure 4.14: Split result with Euler with $\nu = 1.307\text{e} - 6\ \text{m}^2/\text{s}$, $\alpha = 0.5$. This is the y speed. The green line is the analytical solution boundary for the result without boundary condition. The blue line is the calculate boundary. We see that except at the end, the two coincide. The y speed is proportional to the x^2 position, which coincide with the analytical result. The perturbation at initial condition are smirred by the viscosity.

The two runs took a day to complete. 3 dimensional runs were not possible for reasons of time and computing power.

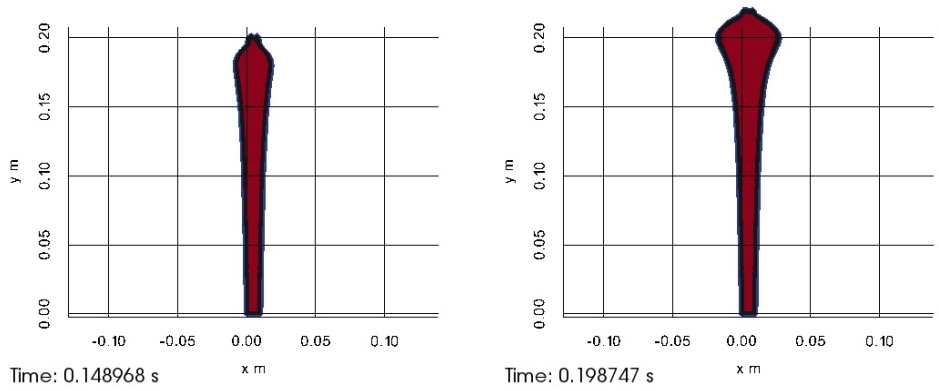
This is shown in figure 4.15, 4.16, 4.17, 4.18, 4.19 and 4.20.

The most expensive operation is to calculate the projection operator for every iteration.

We see three phases in the computed result:

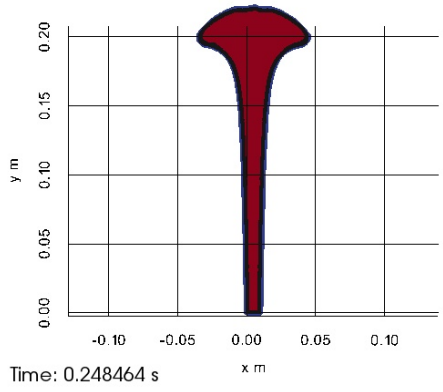
1. Vertical motion without deformation. Gravity had not the time to act a lot on the speed.
2. The top particles want to push the below particles because of gravity. But because of incompressibility the below region becomes larger. This generates a lateral speed.
3. We touch the supression particles limit and have now a stationary state.

We think that the reason why the jet lateral distance is so big is because we are in a 2d problem where we have less spatial liberty. The pressure exerted by the gravity can in 2d escape only in one dimension and not two.

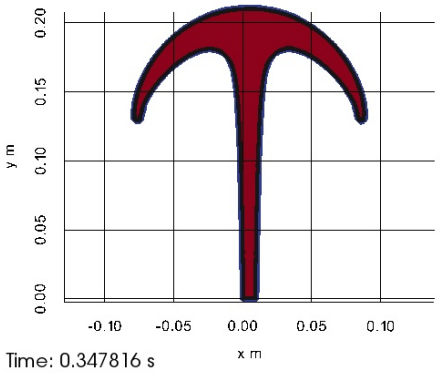


(a)

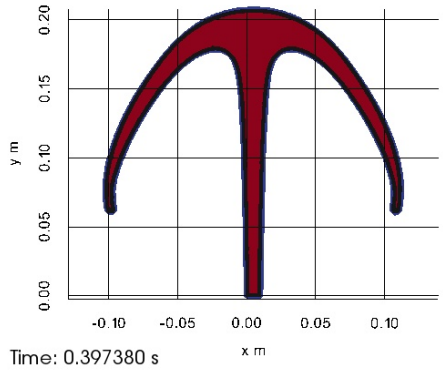
(b)



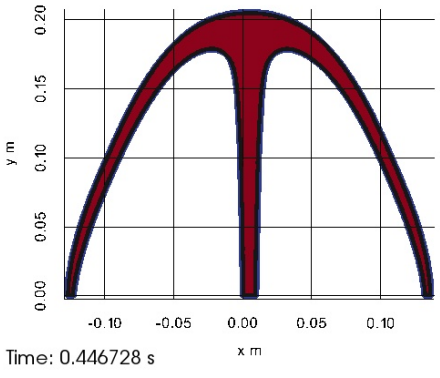
(c)



(d)



(e)



(f)

Figure 4.15: Jet simulation with an initial speed 2 m/s. The red region is the region with fluid. The black surface are in fact the particles.

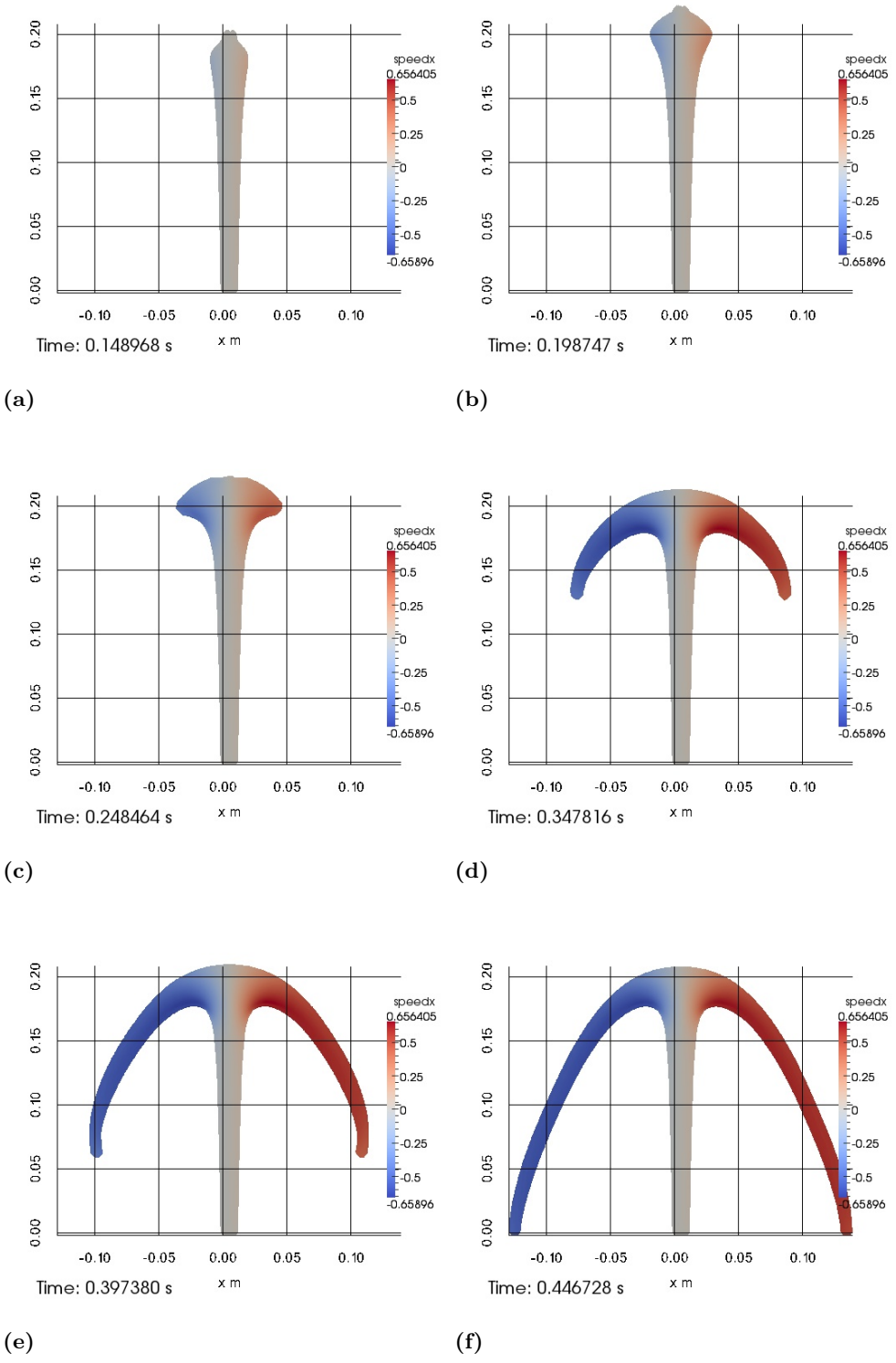


Figure 4.16: Jet simulation with an initial speed 2 m/s. This is the representation of speed in the x direction.

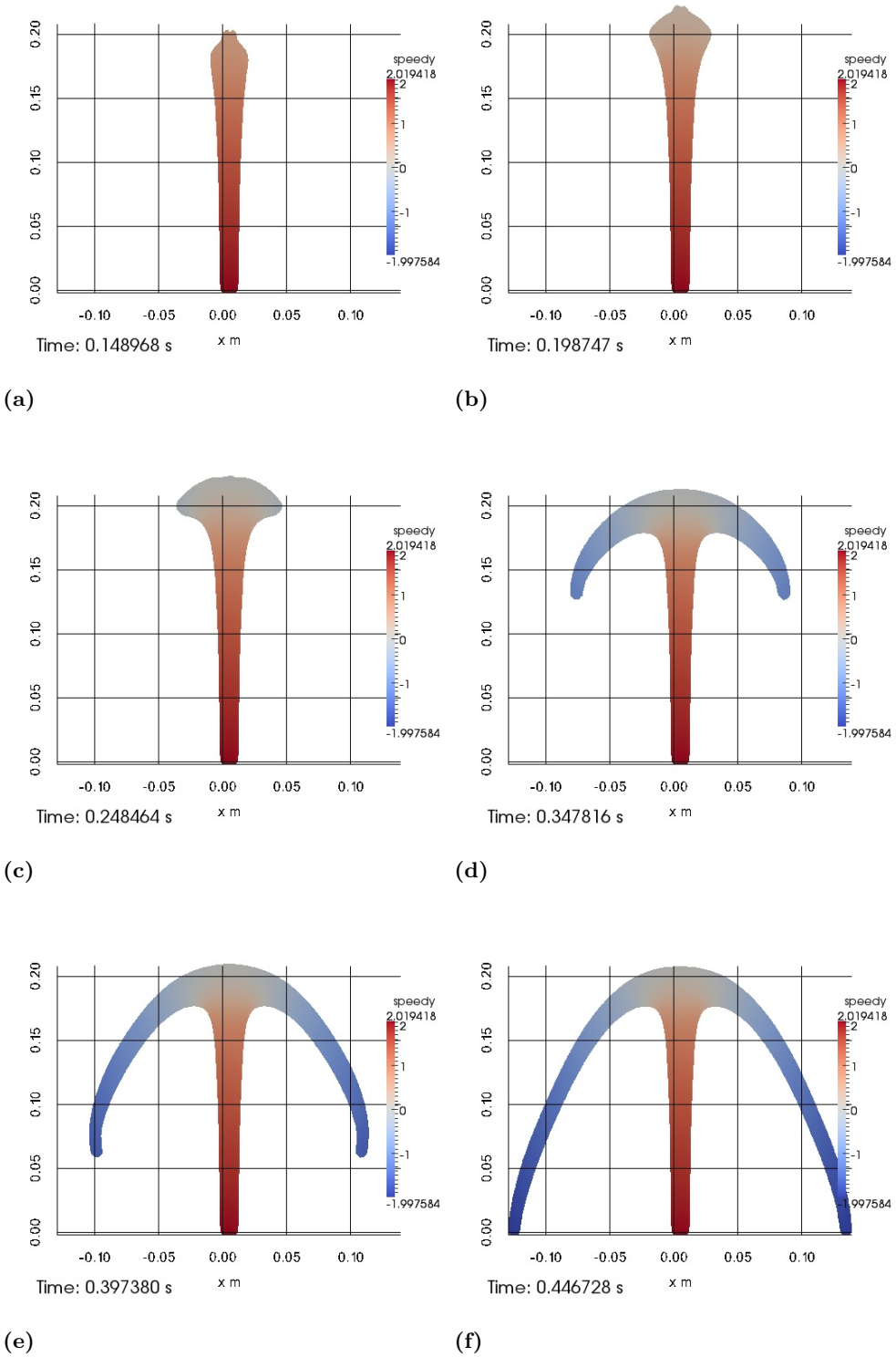


Figure 4.17: Jet simulation with an initial speed 2 m/s. This is the representation of speed in the y direction.

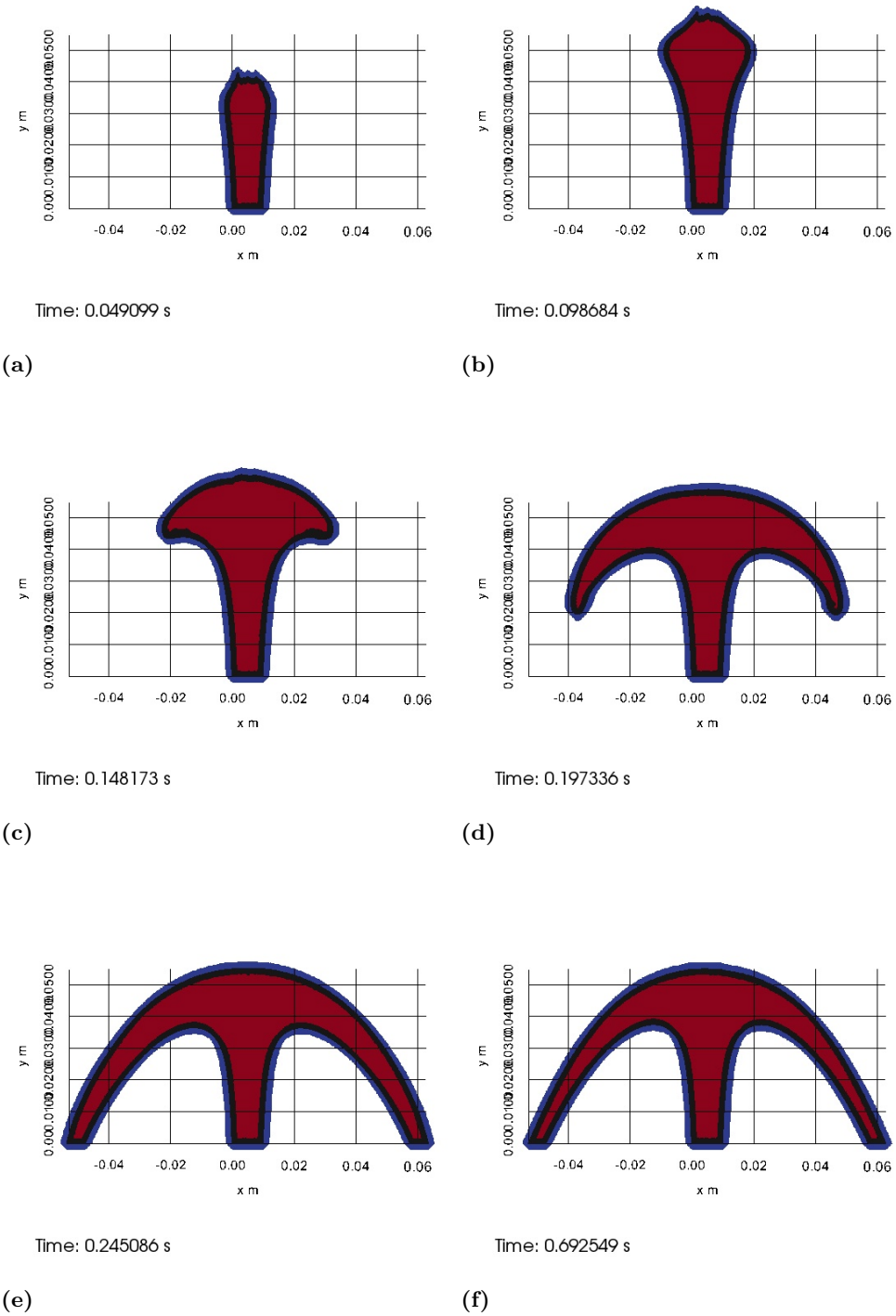


Figure 4.18: Jet simulation with an initial speed 1 m/s. The black surface are in fact the particles.

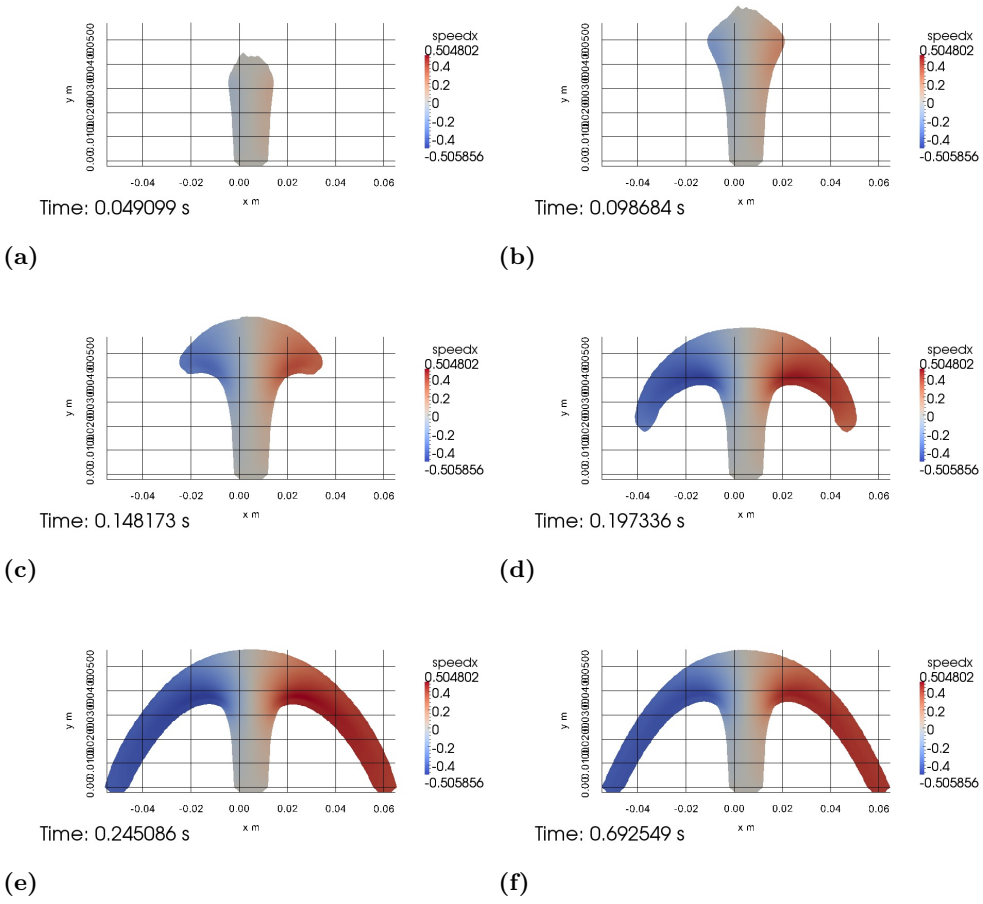
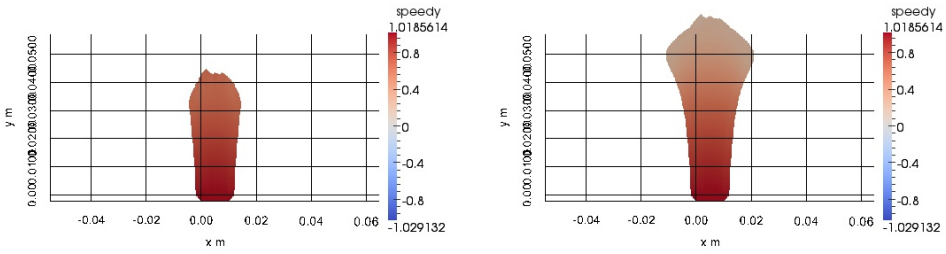


Figure 4.19: Jet simulation with an initial speed 1 m/s. This is the representation of speed in the x direction.

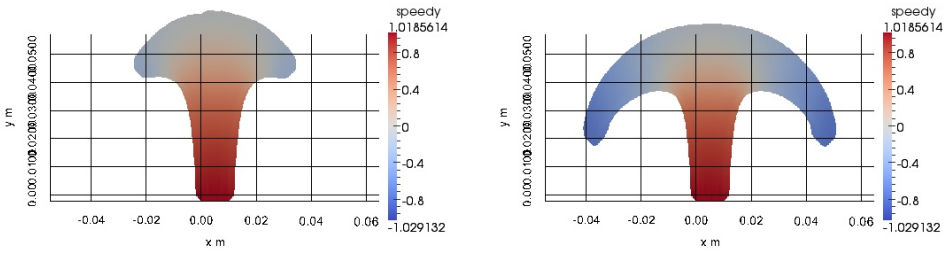


Time: 0.049099 s

(a)

Time: 0.098684 s

(b)

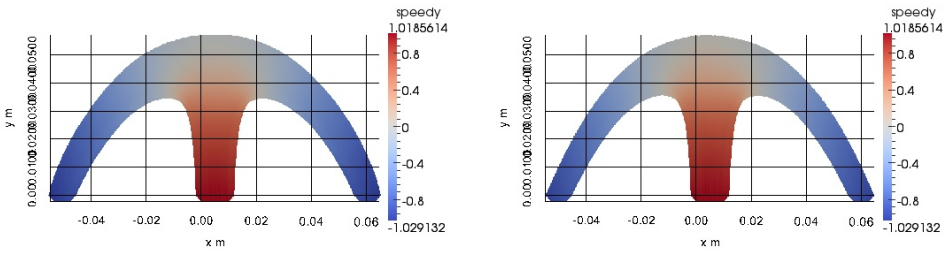


Time: 0.148173 s

(c)

Time: 0.197336 s

(d)



Time: 0.245086 s

(e)

Time: 0.692549 s

(f)

Figure 4.20: Jet simulation with an initial speed 1 m/s. This is the representation of speed in the y direction.

Chapter 5

Conclusion

In conclusion we were able to have a "simple" scheme to solve free surface incompressible Navier-Stokes equations. For this we have used the following elements:

- A spatial discretization on a staggered grid and the definition of a discretized differential operator on it.
- The reduction of the original problem to an ODE with a projected function. This needs to solve a linear system at every time step.
- A topological marker particle and interpolation method on the marker.
- An extrapolation operator that needs to ensure boundary conditions.

We were not able to launch 3d runs and are aware of the high numerical cost of direct Navier-Stokes methods for small viscosity problems, but judged that it was the better method to begin with.

5.1 Possible improvement

The following theoretical improvements are possible:

- Use higher order spatial discretization.
- Use other topological markers instead of particles like level sets or boundary surface representations.
- Have better discretization at the free surface boundary.
- Have a variant of the Runge-Kutta method that is correct if we have a Dirac delta in the function or use smooth topological change.

The following code improvements are possible:

- Have a linear solver which can use the fact that topology does not change much between iterations to be faster on average.
- Make the code run in parallel on cpus and gpus.

5.2 Software used

The code was written primary in **C++11**. The final version uses the following libraries:

- Umfpack as linear solver.
- VTK for output in the vtk format used in the visualization software Paraview.

Previous version used:

- Boost library for python binding and externalization.
- Pyamg in python for multigrid.
- VianaCl for multigrid.
- Lusol in fortran for sparse Lu updating.
- Spooles as linear solver.
- ...

For postprocessing and generate this document the following software were used:

- LuaLaTeX as LaTeX program with some lua script to generate picture.
- Pgfplots and tikz for picture and graphics.
- Matplotlib in python to generate picture and graphics.
- Paraview to visualize the data.

For debuging of the **C++11** code in addition to GDB, Valgrind was used to track memory related error and memory leak.

5.3 Thanks

I am very thankfull to:

Prof. Martin Gander: for the supervising of the work and all my numerical analysis knowledge.

Dr. Félix Kwok: for the supervising of the work and contact in practical numerical analysis.

Prof. Peter Wittwer: for the Mathematics and Physics courses in my Physics studies.

My family: Reto, Norma, Roland, Tamara: for supporting me and all the fun together. And for proof reading.

Bibliography

[1] G. K. Batchelor. *An Introduction to Fluid Dynamics (Cambridge Mathematical Library)*. Cambridge University Press, 2000. ISBN: 0521663962. URL: http://www.amazon.com/Introduction-Dynamics-Cambridge-Mathematical-Library/dp/0521663962.

[2] David Cardon Parris K. Egbert David Cline. "Fluid Flow for the Rest of Us: Tutorial of the Marker and Cell Method in Computer Graphics". URL: http://people.sc.fsu.edu/~jburkardt/pdf/fluid_flow_for_the_rest_of_us.pdf.

[3] Nikolaos A. Kampanis and John A. Ekaterinaris. "A staggered grid, high-order accurate method for the incompressible Navier-Stokes equations". In: *J. Comput. Phys.* 215.2 (July 2006), pp. 589–613. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2005.11.014. URL: http://dx.doi.org/10.1016/j.jcp.2005.11.014.

[4] Steve Marschner and Jose Guerra. "Lecture 13: Partial Differential Equations". URL: http://www.cs.cornell.edu/courses/cs667/2005sp/notes/13guerra.pdf.

[5] J. M. McDonough. "Lectures in Computational Fluid Dynamics of Incompressible Flow:Mathematics, Algorithms and Implementations". URL: http://www.engr.uky.edu/%5C~%7B%7Dacfd/me699-lctr-nts.pdf.

[6] S. McKee et al. "The MAC method". In: *Computers & Fluids* 37.8 (Sept. 2008), pp. 907–930. ISSN: 00457930. DOI: 10.1016/j.compfluid.2007.10.006. URL: http://dx.doi.org/10.1016/j.compfluid.2007.10.006.

[7] M.F. Tomé et al. "GENSMAC3D: a numerical method for solving unsteady three-dimensional free surface flows". In: *International Journal for Numerical Methods in Fluids* 37.7 (2001), pp. 747–796. ISSN: 1097-0363. DOI: 10.1002/fld.148. URL: http://dx.doi.org/10.1002/fld.148.