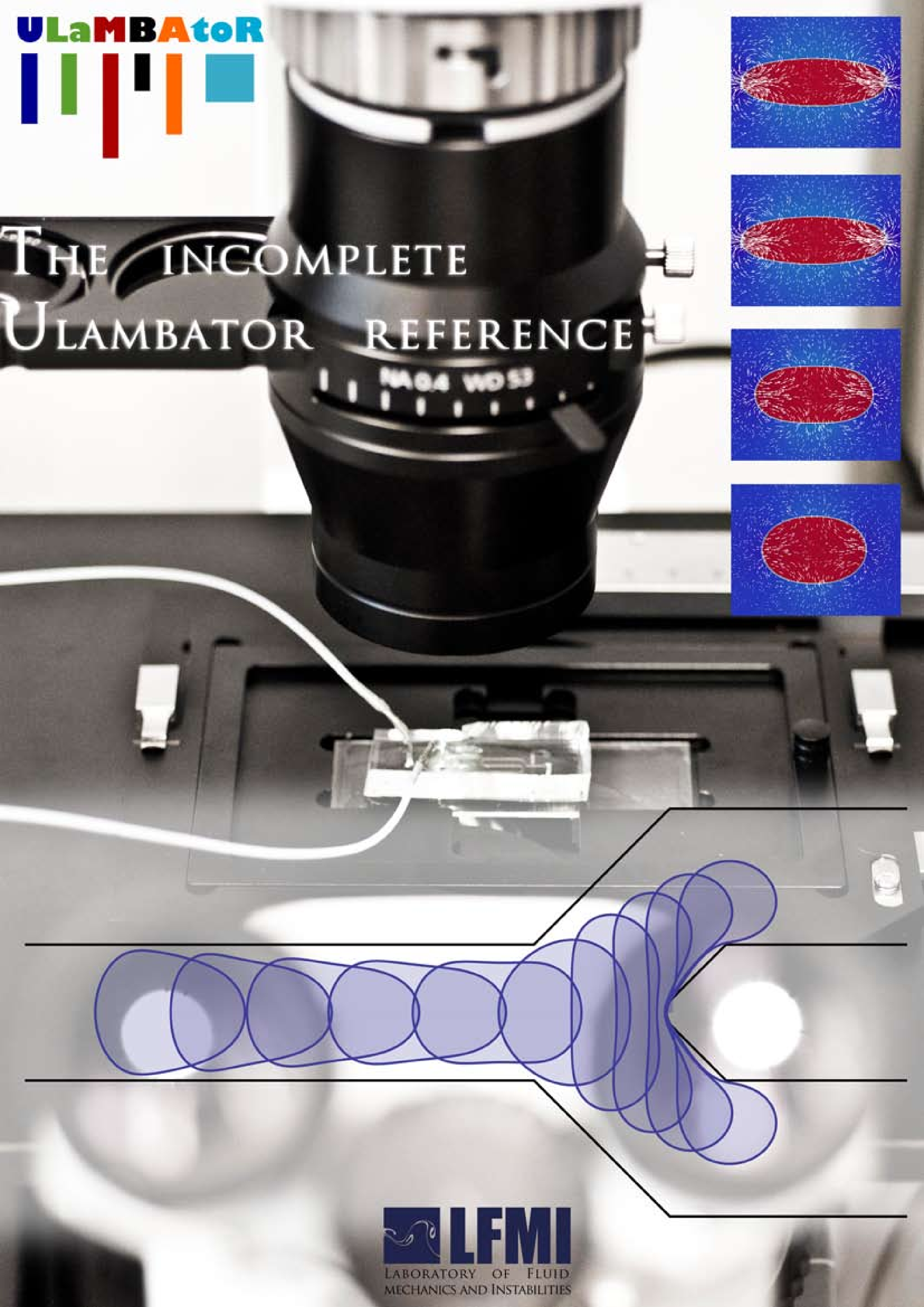
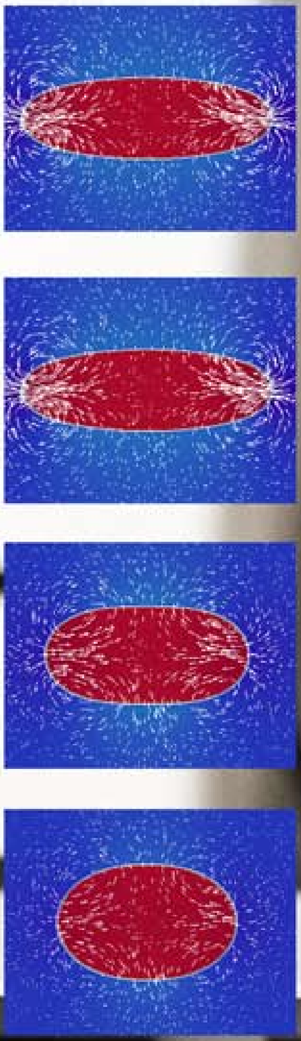
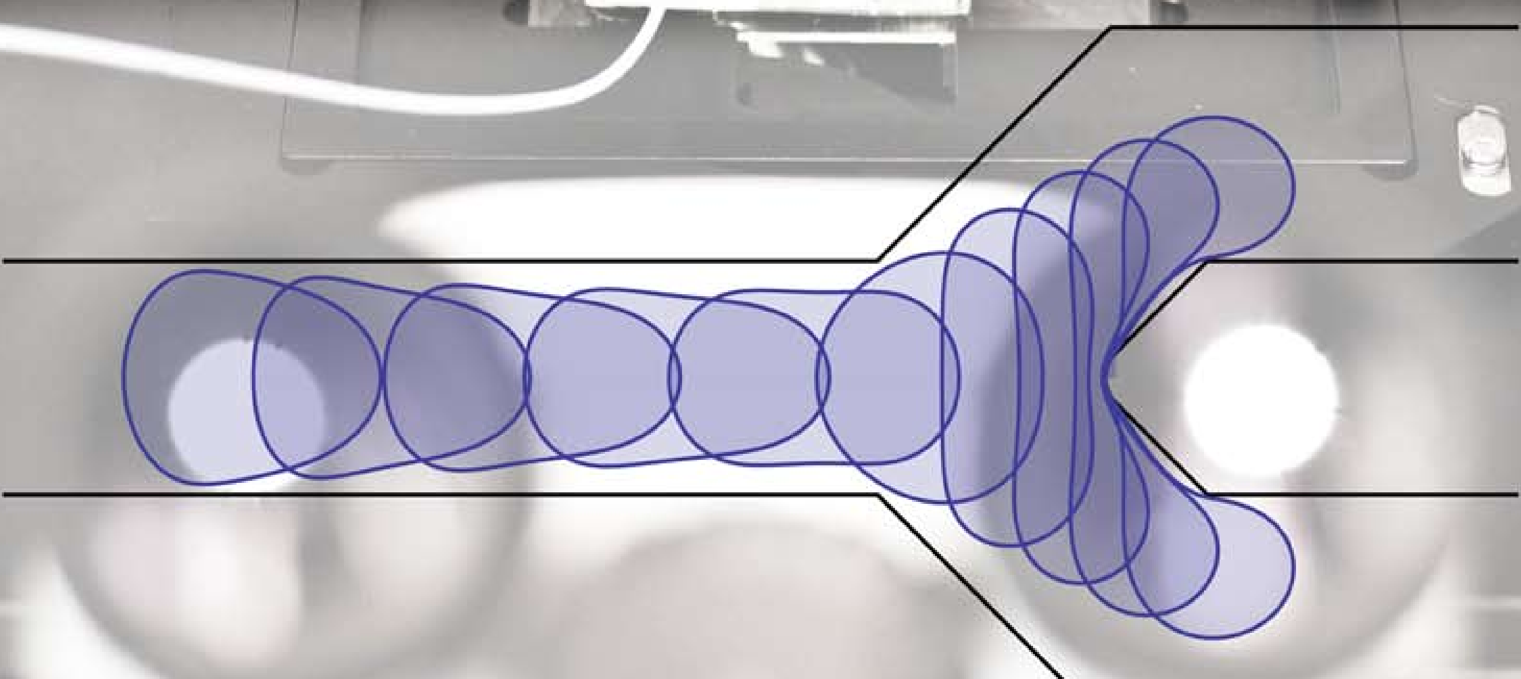
ULaMBAtoR

The incomplete Ulambator reference







LFMI

Laboratory of Fluid mechanics and Instabilities

The incomplete Ulambator Reference

Ulambator version 0.7
Manual version 0.72

December 2, 2015

The boundary element solver ULaMBAtoR has been created by Mathias Nagel as a part of his PhD thesis conducted under the supervision of Prof. François Gallaire at the Laboratory of Fluid Mechanics and Instabilities at the Federal Polytechnical Institute EPFL in Lausanne, Switzerland.

ULaMBAtoR is currently still used and extended. We make it available so that it may be used and extended by the community.

The source code and documentation can be found at:
http://lfmi.epfl.ch/ulambator
and
http://ulambator.sourceforge.net

Comments and questions can be directed to
mnagel@mat.ethz.ch

Contents

CHAPTER 1

Introduction

What is Ulambator?

ULaMBAtoR is a software tool, which is dedicated to two-phase flows in microfluidic channels that is developed at the Laboratory of Fluid Mechanics and Instabilities at the EPFL in Switzerland. The acronym **Ulambator** stands for **U**nsteady **La**minar **M**icrofluidic **BE**M **A**lgorithm **To** **O**btain **R**esults.

The numerical algorithm is based on the Boundary Element Method, which does not require a domain discretization but a boundary mesh only. Within the framework of the classical BEM no inertial terms can be accommodated and therefore the U for Unsteady highlights the deformability of the fluid interface and Laminar highlights flow at Reynolds number equal zero.

The solved flow equations are averaged over the height, leading to a system of 2D differential equation, that requires in return channels of high aspect ratio, being much wider than high.

Why and When to use Ulambator?

ULaMBAtoR solves fluid flow equations that are based on a depth-averaged model (so called Brinkman equation). A prerequisite for depth-averaging is that the fluid motion takes place in a common plane and is confined by between channel floor and ceiling. This confinement can be rationalized by the aspect ratio of droplet radius r divided by channel height h. When $r/h \gg 1$ the model does an excellent job.

In fact when $r/h \gg 1$ one could even resort to the much simpler Darcy equation. However in practical cases the confinement is often around $r/h \gtrsim 1$ and the lateral walls are not far either $w/h \gtrsim 1$, where w is the channel width. These weakly confined cases require generally the solution of the 3D Stokes (or even Navier-Stokes equation). This can be very cumbersome and may only be achieved at considerable computational costs. As a desperate try one could consider to solve instead the 2D Stokes equation, which would mean completely unconfined droplet tubes.

The Brinkman reconciles the 2D Stokes equation and the Darcy equation in establishing a model that is still dangerous for $r/h \sim 1$ but has shown results with appreciable agreement to experimental observations when $r/h \gtrsim 1$. When the channels become confined $r/h > 1$ a 3D simulation may become prohibitively expensive, where the 2D Brinkman model may provide results in a couple of minutes on a simple Desktop PC.

The numerical method underlying the solver is published in:

> Mathias Nagel, François Gallaire
> **Boundary elements method for microfluidic two-phase flows in shallow channels**
> Computers & Fluids, Volume 107, Pages 272-284, 31 January 2015
> http://dx.doi.org/10.1016/j.compfluid.2014.10.016

A preliminary version of the article can be found on arXiv (http://arxiv.org/abs/1411.2728).

A brief overview

An efficient implementation of the solver was achieved by programming the discretized scheme in the programming language C++. Although high-level programming languages like MATLAB offer fast matrix solvers, it was found that achievable computation times were much shorter using C++. The aim was to provide classes that contain high-level functions of the mathematical schemes described in our article [1]. In a similar fashion as the Finite Element Code FreeFEM++, developed by the Frederic Hecht, or Basilisk, developed by Stéphane Popinet at Institute d'Alembert, both at UMPC in Paris.

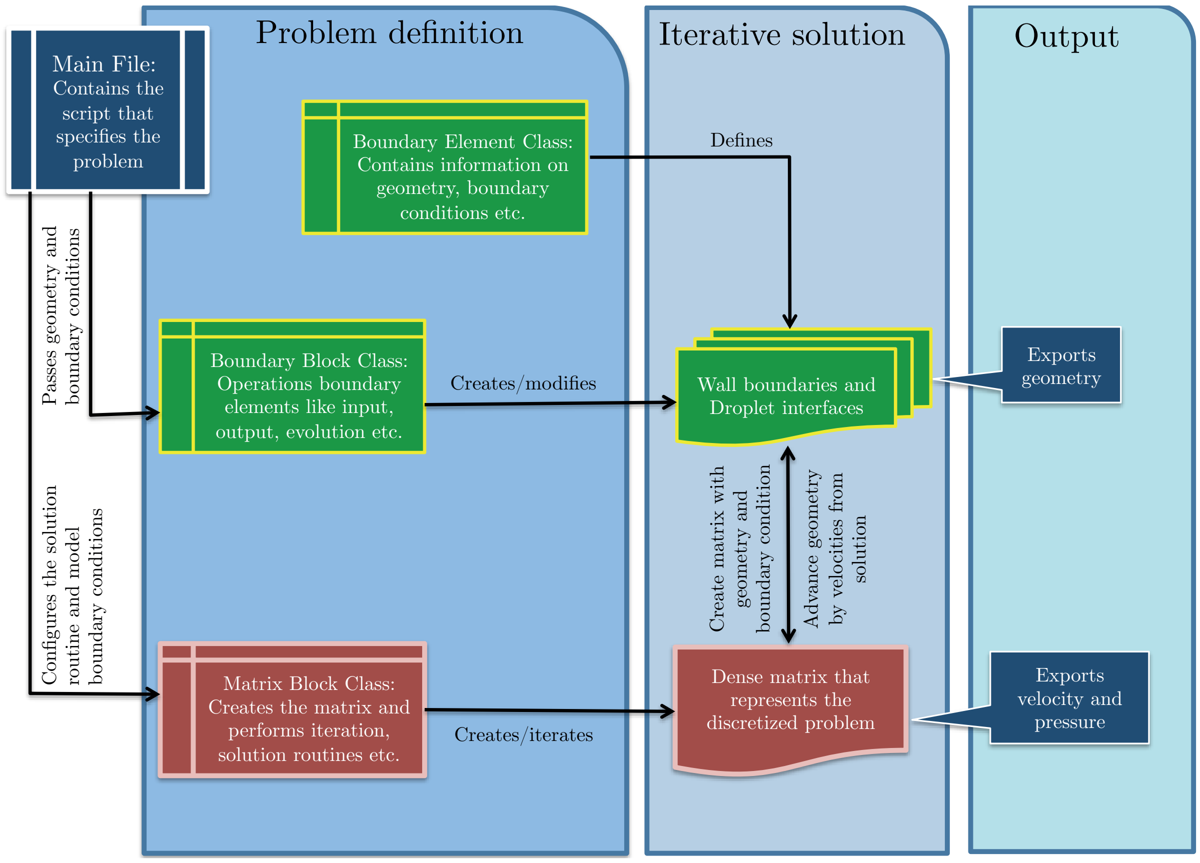

Problem definition
Iterative solution
Output
Main File: Contains the script that specifies the problem
Boundary Element Class: Contains information on geometry, boundary conditions etc.
Defines
Passes geometry and boundary conditions
Boundary Block Class: Operations boundary elements like input, output, evolution etc.
Creates/modifies
Wall boundaries and Droplet interfaces
Exports geometry
Configures the solution routine and model boundary conditions
Create matrix with geometry and boundary condition
Advance geometry by velocities from solution
Matrix Block Class: Creates the matrix and performs iteration, solution routines etc.
Creates/iterates
Dense matrix that represents the discretized problem
Exports velocity and pressure

Figure 1.1: Working principle among the Boundary Element Class, Boundary Block Class and Matrix Block Class that are used by the Main class to set up and solve the discretized problem.

A programming environment of high-level functions is a trade-off between flexibility and a user-friendly interface. C++ is an object oriented programming language, in the realm of object oriented programming three classes provide the routines and variables that are separated according to their field of application. A Boundary Element class contains all variables and functions to access information of a closed boundary curve. Several objects or boundaries can be created from this class. These boundaries are created by function calls from the Boundary Block class object. An object of this class provides all function that are necessary to read and edit boundary information, e.g. advance the droplet interface, write the geometry and pass boundary conditions to the Matrix Block class object. The Matrix Block class manages the memory of the matrix, unknowns and right hand side. In this call are specified the problem parameters: Capillary number Ca, viscosity ratio λ, aspect ratio l/h, time step Δt and the solution method. It furthermore provides all function and routines necessary to set-up the matrix from the boundary conditions and geometry of the boundaries.

On top of these three classes is the user defined main program file, which controls configures and controls the simulation using the Boundary Block and the Matrix Block. The dependence and working principle is sketched in figure 1.1.

CHAPTER 2

Getting started

Installation

The ULaMBAtoR solver needs a C++ compiler with the mathematical library LAPACK and optionally with OpenMP support. The solver runs under Linux, OS X and Windows (in theory).

Visualization requires Matlab (or a clone) and ParaView. All the software needed to run the code and view the results is available free of charge. Overall ULaMBAtoR requires only a PC and a user with a little bit of programming experience.

There is a chance that compilation works right away. If not follow the instructions below. The latest version of ULaMBAtoR can be obtained from git: git clone git://git.code.sf.net/p/ulambator/code ulambator-code

Or manually from http://sourceforge.net/projects/ulambator/files.

After copying the files into a directory of choice edit the Makefile according to your systems needs. All lines with # are comments or deactivated settings. Specify your compiler in the CPP variable and compiler options in OPTS that work on your system. These options differ between Linux, OS X and whether you use Lapack, MKL or OSXs Accelerate framework. The variables INCS and LIBS can be used to provide the path to libraries or headers.

You can first check your compiler by calling `make comp_test`. The source code will be compiled without mathematical libraries and OpenMP. The message "Compiling without OpenMP and Lapack successful" means that the program can be compiled, although it can not be executed.

Typing `make` compiles the program, if errors occur disable the OpenMP support first and try to resolve the issues that should come from LAPACK. If it does compile a series of tutorials is created that can be tried. For user defined case type `make user1` for example.

If it does not compile solve it with the indications below. If that does not help, write me an email or wait till this chapter is complemented.

LINUX: Install the GCC compiler, https://gcc.gnu.org.
And install LAPACK, http://www.netlib.org/lapack.

OS X: Either install the GCC compiler or install Xcode from the AppStore, LAPACK is included in Xcodes Accelerate framework.

Alternatively install the Intel compilers and MKL libraries for your system, https://software.intel.com/en-us/c-compilers and https://software.intel.com/en-us/intel-mkl

For Windows compiling has not been tested, but searching the internet for "Windows LAPACK" and "Windows OpenMP" seems to provide the required information.

If header files and libraries can not be found despite the fact that they have been installed one may simply try for instance 'locate omp.h' or 'locate libgomp.a' to locate the missing files and include their directories in the header search path *INCS* or the library search path *LIBS*.

Setting up a problem

Problem definitions are to be programmed by the user or certain build in functions can be used. In order to build a Boundary Element environment one first creates a Boundary Block, `BndBlock`, which is a class that handles all boundaries. This class creates all boundary objects using the `BndElement` class as a definition.

When all boundaries are defined one creates the Matrix of the discretized problem, which is done by the `MatrixBlock` class, which uses the Boundary Block to reserve memory for the unknowns and internal variables. This finishes the problem setup.

From now on actions on the boundaries and matrix object can be performed. For the boundaries its mostly writing out geometries or displacing objects. More handling is not required, because a lot of operations like advancing the solution are performed by the Matrix Block on the Boundary Block.

In order to setup **ULaMBAtoR** for a specific case you need to adapt the `main_user1.cpp` file in the `userfolder` or create a new main file and edit the Makefile to compile your version. **ULaMBAtoR** provides some high-level functions that should allow to solve a wide range of problems with only a few lines of code.

The structure of the main program is as follows:

1. Define the wall geometry and liquid interfaces.
2. Define the material parameters, time step and possibly remeshing of the interface.
3. Perform the calculation with time integration and output of data.

All the functions that are used in the following are documented in section 3.

At first the program needs headers that tells the program where to find the **ULaMBAtoR** subroutines. Some more headers may be needed to generate output to the screen etc. A collection of common headers is found in the tutorials in the appendix A. Here we show a header with the Ulambator classes `matblc1` and `bndblock` that are loaded to declare the functions `BndBlock` and `MatrixBlock`.

```
#include "../source/matblc1.h"
#include "../source/bndblock.h"
```

Define the wall geometry and liquid interfaces

One needs to tell the system how many distinct closed boundaries there are:

```
BndBlock bem(number_of_boundaries);
```

All of these closed boundaries have are initialized with given number of panels, number of elements and a given type. The type typically specifies whether the boundary is a wall or a liquid interface. The number of elements is related to the number of unknowns and is needed for memory allocation. With panels we mean portions of a closed curve with a given boundary condition. If you consider a rectangular channel for instance, it is one closed boundary composed of four panels. One inflow panel, two side wall panels and one outflow panel.

```
bem.Elements[boundary_id].Init(num_panels, num_elements, type_boundary);
```

Each of the panels needs to be initialized with a given geometry and with a given boundary condition.

```
 bem.PanelInit(boundary_id, num_elements, geo_type, bc_type, bcs, pos);
```

Define the material parameters, time step and possibly remeshing of the interface

When the boundaries are defined one uses this spatial information to create the matrix.

```
MatBlC1 mat(&bem, aspect_ratio, viscosity_ratio, Ca);
```

Together with the geometry `bem` one specifies the aspect ratio (width over height), the viscosity ratio (dispersed over bulk fluid) and the capillary number.

For deformable interface one should activate remeshing so that the points get redistributed evenly with inter-point distance specified in `element_size`. When points are too distant or close they become redistributed.

```
bem.RemeshOn(element_size, 0);
```

Time marching does not need to generate output after every iteration. Using an integration scheme allows to iterate in an inner loop for a certain number of `subtimesteps` iterations of timestep `deltaT`.

```
mat.setTimeStep(deltaT,subtimesteps);
```

Due to the non-dimensionalization based on surface tension and viscosity the time is based on a capillary time scale, which is in microfluidics usually quite fast. If the problem has a high capillary number, one might want to use a time step smaller than one because the capillar forces are not the limiting issue. On the contrary, if the capillary number is very low, the problem might evolve very slowly and one may chose a timestep larger than one. The interfacial terms are stabilized, which means the computations do not diverge, in reasonable limits, but the result maybe inaccurate.

Certain types of boundary conditions, like the capillary anchor that is demonstrated in tutorial 4, are not stabilized and will diverge if the time step larger than one.

Perform the calculation with a time integration and output of data

When the matrix object is created one can build the matrix based on the geometry and the boundary condition. One can use either `mat.Build(deltaT)` and `mat.Solve()` manually as shown in tutorial 2 or one uses an automated routine.

```
mat.SolveRK1();
```

Here `subtimesteps` iterations are performed, then the interface is exported.

```
bem.allWrite();
```

Which in this case writes down all droplet interfaces.

Compilation and execution

It is suggested to program and debug in an IDE (integrated development environment). Once the programming is finished the user compiles the program. In the `Makefile` that is provided one specifies the file name in line 35 and a name for the executable (the path after '-o'). Or one copy and pastes these lines to create user2 program. Thereafter one types `make` and hopefully the program compiles without errors. Finally one calls the program and the simulations starts.

Once the desired case is configured one compiles the program with make and runs the executable, i.e. `tutorial1.o`.

Tutorial 3 demonstrates how to configure a compiled simulation from the command line. Call the respective program for instance: `./tutorial3` or `./tutorial3 1 5 12 1` with user defined configuration and see the program running. When running on a remote system consider to add `./tutorial3 > some_text_file.txt &`, which allows you to log-off and keep the code running.

Dimensional conversion

When the simulation is configured one has to provide the non-dimensional parameters of a problem. And once the results are obtained, they might have to be dimensionalized in order to give dimensional velocities or pressures.

In contrast to the notation in the article [1] it was found more convenient to write dimensional quantities with minuscules, typical scales of dimensional values as minuscules with tilde and non-dimensional values with capital letters, *i.e.* $t = \tilde{t}\,T$.

We turn first to non-dimensional geometric parameters: The length scale $\tilde{l}$ for one unit in the simulation can be freely chosen. The aspect ratio w/h has to be set accordingly. If a channel has a width of $w = 100\mu m$ and a height of $h = 20\mu m$ and one decides to represent the channel width in the simulation by 2 units, then $\tilde{l} = 50\mu m$ and the aspect ratio $\tilde{l}/h = 2.5$.

The velocity scale is determined by the material parameters: surface tension γ and bulk fluid viscosity μ, therefore $U = \gamma/\mu$. E.g. an inflow velocity of $u = 7mm/s$ in a channel filled with water and gas bubbles ($\gamma = 70mPam, \mu = 1mPas \Rightarrow \tilde{u} = 70m/s$) the non-dimensional inflow velocity is $U = u/\tilde{u} = 0.0001$.

The droplet viscosity is non-dimensionalized by the ratio of droplet viscosity over outer fluid viscosity, $\lambda = \mu_d/\mu_c$.

The pressure is non-dimensionalized by the surface tension divided by the length scale, $\tilde{p} = \gamma/\tilde{l}$, or for single phase flow with $P = \mu\tilde{u}/\tilde{l}$.

The time is non dimensionalized by $T = \tilde{l}\mu/\gamma = \tilde{l}/\tilde{u}$.

For microchannels it is useful to recall some relations of pressure and flow rate as to impose the right boundary conditions. For parallel flow in a shallow channel one can derive the simplified relation for the dimensional volume flow:

$$\dot{v} = \Delta p \frac{h^3 w}{12\, l\, \mu}\left(1 - \frac{h}{\sqrt{3}w}\tanh\left(\sqrt{3}\frac{l}{w}\right)\right),$$

Which gives the relation between pressure drop and volume flow in a channel of width w, height h and length l.

For volume driven flow the dimensional flow rate is to be converted into a dimensional velocity and the non-dimensionalized with $\tilde{u} = \gamma/\mu$.

$$\tilde{u} = \frac{\dot{v}\,\mu}{h\,w\,\gamma} = Ca.$$

The non-dimensional inflow rate appears as a capillary number.

User guidelines

In order to understand how to apply **ULaMBAtoR** it is proposed that the user may have a look at some of the examples in the appendix A. Comments guide through the code and summarize the results. In addition the examples can be compiled from the source code and run by the user. Modifications can be made and their influence observed on the outcome of the simulation.

After one is familiarized with the structure one may find it much easier to follow the explanations that follow in the next chapter.

A word of advice: Unfortunately there are up to now only few asserts and return messages if something does not work out. It is imperative to configure ULaMBAtoR in an IDE in order to use a debug procedure to got step by step through the code to see where the inconsistency is.

In the course of time the source code will be commented through out.

CHAPTER 3

Source code reference

BndBlock Class

BndBlock

One creates a `BndBlock` and gives the number of boundaries as argument. A boundary is a closed curve that will consists of one or several panels with different boundary conditions. One panel consists of several points, in between these points the boundary elements discretize the geometry for the boundary integral equation.

```
BndBlock bem(number of boundaries);
```

BndBlock.Element.Init

Then one creates the boundaries by initializing them one after the other. The `boundary id` is an identification number to distinguish different boundaries, starting from zero.

```
bem.Element[boundary id].Init(number of panels, number of points, type);
```

Non moving boundaries need to be created first. As arguments we give the number of boundary panels, the total number of points for the discretization and the boundary type number.

Boundary type:

0 Wall boundary.

1 Moving rigid object.

2 Droplet.

3 Attached fluid interface.

The fluid domain is always to the left of the boundary when following it on its path. This means that outer boundaries are drawn counter-clockwise and inner boundaries clockwise. Moving rigid objects are also drawn clockwise while droplets are drawn counter-clockwise as they have to enclose the inner fluid. When all the boundaries are initialized we initialize all the panels.

BndBlock.PanelInit

One initializes the panels by calling:

```
bem.PanelInit(boundary id, nb of points, geometry type,
        bnd condition type, geometry array, boundary cond array);
```

The `boundary id` is the same as specified before. The number of points are the points for one panel. The geometry type is an integer that is selected according to the following list. With the coordinates specified in the `geometry array`

Geometry type:

0 Straight line, coordinates are X_0, Y_0, X_1, Y_1.

1 Curved Segment, coordinates are X_0, Y_0, X_1, Y_1 and the curvature radius R. Note that R needs to be larger than the half distance but can be positive or negative depending on the orientation.

2 A circle, coordinates are X_0, Y_0, R and a perturbation amplitude A and wave number ω. If the wavenumber is zero than a random perturbation is added.

3 An ellipse with coordinates X_0, Y_0 and axis length in x-direction L_x and y-direction L_y.

4 A rectangle with rounded caps, shaped object with coordinates X_0, Y_0, length L_y and width W_x. The rounded caps are on the north and south faces.

The point to geometry provides an array of coordinates according to the geometry type. When the geometry is not closed the solver will exit with an assert error.

As a `boundary condition type` we distinguish outer flow for boundaries, outside the droplet, and inner flow, for boundaries that are inside the droplet domain.

Boundary condition types for outer boundaries:

0 ZeroVel (the no-slip boundary condition)

1 NormVel (arguments are normal and tangential velocity)

2 Pressure (prescribe normal stress, tangential stress is zero)

3 Poiseuille (prescribe a mean velocity of a Poiseuille type flow profile)

4 FlatVel (arguments are velocities in x and y direction)

5 InfiniBox (prescribe a velocity in x and y direction at infinity)

6 Point source (Fluid injected from a point source at the origin)

Types of inner boundary conditions:

0 ZeroVel (the no-slip boundary condition)

11 Inner NormVel

12 Inner Pressure

13 Inner Poiseuille

14 Inner FlatVel

16 Inner Point source

Types of mobile interface boundary conditions

21 Droplet (Liquid-liquid interface)

22 Boos and Thess (Marangoni flow used for validation)

When the boundary is at infinity or a point source it should be of geometry type 0, initialized with zero points and given the respective boundary condition type (5,6,16).

The pointer to the boundary condition contains the values that are passed to the boundary condition. The mobile interface boundary condition does not need any parameters. It is preconfigured to have a surface tension and other option can be set afterwards. Due to the non-dimensionalization the surface tension is unitary. Its effective value is given through the inflow velocity (see section 2 – Dimensional conversion).

Optional Procedures

ReadDrop

The simulation can be restarted from a given drop position. Instead of `bem.PanelInit` one calls `bem.DropRead(int drop id, int timestep)` where the `droplet id` and the `timestep` is specified. Before this call one needs to enable write on the `BndBlock`, because the solver needs to find the corresponding files. Reading a droplet changes the timestep of the `BndBlock` to the read value. So if one wants to start from time zero after `ReadDrop` the time step needs to be set to zero with (`bem.timestep = 0;`).

correctArea

In some cases, when the viscosity ratio and the forces exerted on the droplet are high the numerical error leads to decreasing or increasing area of the droplet. One can fix this problem by providing an initial area to a droplet. The procedure `correctArea` is called in every iteration in `SolveRK1 & 2` and for objects where `initalArea` is set the area is corrected. To provide this do:

```
bem.Elements[boundary id].initialArea = bem.getArea(boundary id);
```

Btw. `getArea`(boundary id) is a function that returns the area enclosed by an interface. The `boundary id` corresponds to the numbering that is used when the interfaces are initialized. Remark: The first boundary starts with id 0 not 1.

SeedEquiDist

Usually the droplet interface gets deformed and needs remeshing once in a while. The procedure `SeedEquiDist(int dropid, double rlim)` will check if the points on the interface are evenly distributed and will remesh if needed. Alternatively one can call `bem.SeedDrop` if `bem.rlim` is specified, which will check and remesh all necessary droplets. In order for this function to work one needs to call `bem.rlim =` value;.

After remeshing one needs to call `mat.UpdateNOFM`, which tells the matrix that the number of unknowns has changed and new space may need to be allocated.

RemeshOn

This procedure is called as bem.RemeshOn(double desired spacing, int modus), where the desired spacing is reported to `bem.rlim`. The only modus available for now is a constant point distance, modus= 0. When `RemeshOn` was called the mesh will be checked and remeshed during every iteration of the integration routine `SolveRK1` or `SolveRK2`.

MatrixBlock

MatBlC1

After having initialized the boundaries one creates the Matrix Block. When calling the constructor of `MatrixBlock` one needs to pass a pointer to the `BndBlock` object and the flow parameters aspect ratio l/h of the channel, viscosity ratio λ and the characteristic capillary number. The capillary number does not influence the simulation, it is forwarded to be saved by `mat.LogCfg()`.

```
MatBlC1 mat(&bem, aspect ratio, viscosity ratio, surface tension);
```

That already finishes the problem definition. The problem parameters can be changed later on. In order to reset the viscosity ratio use `mat.Lambda =` *value* and for the capillary number `mat.Ca =` *value*. The aspect ratio is transformed into a permeability by multiplication with $\sqrt{12}$, when changing the aspect ratio use `mat.kF =` sqrt(12.0)**value*,

MatBlC1.Build and MatBlC1.Solve

The problem can now be compiled into a matrix using `mat.Build(`time step`);` and solved using `mat.Solve();`. The result of `Solve()` is a vector, which contains the *a priori* unknowns. If the velocity at a point was given, the corresponding unknown is a surface stress in x and y direction. If the surface stress was given at a point, the corresponding unknown is the velocity in x and y direction.

The procedure `Build` takes the time step as an argument because it uses a stabilization scheme for the interfacial tension. Using this scheme allows for larger time steps, setting the time step in `Build` to zero disables the stabilization.

Time marching routines

In order to advance the interface by one time step one uses `mat.AdvanceTimestep(`time step`, 1);`. Hereafter the matrix is to be rebuild and solved with `mat.Build(`time step`);` and `mat.Solve();` in an iterative fashion.

Instead of programming the loop oneself it is also possible to use a first or second order integration scheme. Before that one needs to call `setTimeStep(`time step, number of iterations`);` to specify the steps performed during one loop and the time step used. For a first order scheme use `mat.RK1Solve();` or for a second order scheme `mat.RK2Solve();` .

Optional Procedures

Subfolder

Subfolder checks if a subfolder exists, if it does the program will abort. If the folder does not exist, `Subfolder("folder name"` will create the subfolder and use this directory for the output. `EnableWrite(char*)` needs to be called. This serves essentially if a parametric study is done from one executable. The executable is launched with options, one of them being a folder id number. The check of the folder existence helps to prevent erronous overwriting of results.

Timestretch

At times the droplet can be subject to forces that would require a reduced time step. If you want to assure that the timestep times the maximal interface force remains below a certain threshold you can set the threshold by `mat.timefactor = x;`. If this value is zero no time stretching is performed.

Static Model Boundary Conditions

Static boundary conditions we name those, which act on the out–of–plane droplet meniscus and therefore influence Laplace's law. Exceptions are 8 and 9. These models are applied in the droplet boundary condition and are initialized by:

```
ChoseStatBC(int mode, double parameter 1, double parameter 2);
```

The implemented modes are:

2 Rail: Models a groove in the channel ceiling. Yet to be unspecified.

3 Hole: Models an indentation in the ceiling of the channel. The indentation works as a capillary anchor and increases the out–of–plane curvature. The model is described in an article [**?**]. Parameter p_1 is the radius of the hole, parameter p_2 is not used.

4 Hole Array: Models an array of several holes. Yet unspecified.

5 Wedge: Models a channel with a weakly non-parallel ceiling. This effect influences only the out–of–plane curvature and not the viscous effects and is therefore of dubious physical relevance. Parameter 1 is the increase in channel height with the x direction and parameter 2 with the y direction. Center is the origin. The increase is relative to the channel height. If parameter 1 is 0.1 and $x = 10$ then the channel has doubled its height.

6 Marche: A hyperbolic tangent shaped step function, less appropriate and therefore not specified.

7 Flatborder: Assuming that the interface meniscus in the out of plane direction is not a half circle but a flat interface. This is natural for an aspect ratio $r/h \approx= 0$ but unphysical for confined droplets.

8 Rotate: A microchannel that rotates around the point $x =$parameter 1, $y = 0$ at angular velocity parameter 1. The angular velocity is non dimensional and given as $Bo = \frac{\Delta\rho \tilde{l}^3, \dot{\omega}^2}{\gamma}$. The density difference is $\Delta\rho$, the length scale is $\tilde{l}$, the rotational velocity is $\dot{\omega}$ and the surface tension γ.

9 Gravity: Models the influence of gravity acting at strength parameter 1 in the x direction and of strength parameter 2 in the y direction. The non dimensional strength is $Bo = \frac{\Delta\rho\, g\, \tilde{l}^2}{\gamma}$.

Dynamic Model Boundary Conditions

Dynamic boundary conditions we name those, which act on the surface tension and therefore influence on Laplace's law. These models are applied in the droplet boundary condition and are initialized by:

```
ChoseDynBC(int mode, double parameter 1, double parameter 2);
```

The mode can be chosen between $1, 2$ and 3:

1. A surface tension gradient around a point in the coordinate center. This models for instance the effect of a focused laser beam that increases the temperature and decreases the surface tension (supposing it decreases). Then surface tension increases by parameter 1 times the nominal value of the surface tension. With a beam width of radius parameter 2. The beam is a Gaussian and the beam width is defined as the distance, where the power is halved. For parameter 1 $= -1$ the surface tension becomes zero at the center.

$$\gamma = \gamma_0 \left(1 + p_1 \exp\left(-\frac{x^2 + y^2}{p_2^2} \ln(2)\right)\right). \tag{3.1}$$

2. A linear surface tension gradient. Surface tension decreasing relative to its nominal strength in parameter 1 with the x coordinate and parameter 2 in the y coordinate.
$$\gamma = \gamma_0(1 + p_1 x + p_2 y). \tag{3.2}$$
3. A decreasing/increasing surface tension hat function that models the effect of a heating wire. Implementation not checked and thus not yet specified.

Generating output

Logging

ULaMBAtoR can create several log files that are written along with the computation. There are standard outputs and a costom output. At first you need to enable writing by providing a directory.

`mat.EnableWrite(char* dir)`

Note that this also sets `writeEnabled` in the BndBlock. The procedure `bem.EnableWrite(char*)` exists too, but is only used when writing (or reading) is needed before the matrix object is created. This is typically when loading a droplet from a file.

Next you prepare the log file. You can omit this step if you want to append data to an existing file or start a file without a header.

`mat.LogPrepare(int log id, char* header)`

An existing file will be overwritten. The header can be empty, but if it is not it should finish with a line break \n. The log files are named `log`*n*`.dat`, where the number $n =$ `log id` serves to separate different output stream.

Log files are then written using their specific routing, which starts with Log and carries as an argument the logfile id. Examples are:

mat.LogRuntime(int logid) Writes down the seconds between the moment it was first called and the moment it is called.

mat.LogMatlab This function does not requires `LogPrepare` and writes the main simulation parameters into the file `ulam_cfg.txt`. The parameters are: 1) number of time steps per RK loop, 2) initial time step, 3) aspect ratio, 4) viscosity ratio, 5) capillary number, 6) number of boundaries, 7) number of fixed boundaries and 8) number of unknowns.

mat.LogCostom(int logid, int caseswitch) Same use as for `LogCostom` in the Matrix block but defined in the Boundary Block. Depending on the output it may be more convenient to access data within a class. Since the Matrix Block can "look" into the Boundary Block this function may be redundant.

bem.LogCostom(int logid, int caseswitch) This function allows to output any data from the solver in a non-standard way, which means the user has to program the output routine himself in the matrixblock class. The argument [caseswitch] allows to have different preconfigured routines and switch by a given number between them.

bem.LogAreaPos(int logid, int blockid) Area of an object and the coordinates are written.

bem.LogTimeStep(int logid) The timestep and non-dimensional time written. In contrast to other log commands there is no line break performed. Hence this output should be used before another log routine but on the same log id.

Exporting the geometry

All geometries exist as lists of points in the boundary element object. They are written to a file using the function `bem.WritePos(int` boundary id, `int` time step`)`. In the case of a static wall boundary a file name `bnd`*a*`pos.dat` with a being the boundary id is written . And in the case of mobile interfaces `dropat`*a*`ts`*b*`.dat` with a being the boundary id and b being the time step is written

In the case of mobil interfaces the file contains in their first line the non-dimensional time and a relative position in x and y direction. Hereafter follows the list with x and y coordinates and in the case of mobile interfaces also the curvature.

The function `bem.allWrite()` performs `WritePos` for all boundaries and retrieves for the time step automatically the time step that is kept in the Boundary Block. Fixed boundaries are only written if the time step is equal to zero, because these boundaries are supposed to remain fixed.

Plotting of the boundary can be done with a simple plotting tool like gnuplot or Matlab. Examples are found in the subfolder called `tools` that contains several functions to plot the interface. In the A section one finds tutorial examples with their respective output.

Exporting the field variables

A strength of the boundary element method is that it depends only on values on the boundaries, which reduces the number of unknowns. If the velocity or pressure field is required the information of the variables in the domain will be reconstructed from the previous solution. The procedure becomes easily more costly than solving one iteration of the problem, hence the output is only created once in a while, for instance when `SolveRK1` has finished an iteration with several iterations in the nested loop.

There is one reconstruction routines but wrapped into two different output formats. If you need both at a time it maybe more elegant to group the output of both into a single routine such that the reconstruction is not performed twice.

One is called `VisMatlab`, which creates a file that is easily readable in Matlab or a Matlab clone. The other is called `VTKexport`, which creates two vtk files, one for the field variables and one for the geometry, that can be read by ParaView. Both work the same way, the user specifies: 1) time step, 2) resolution (points per unit distance), 3) left, right, bottom and top limits of the window where a visualization in desired.

It is no necessary that the window lies entirely inside the domain. Points outside of the domain will give an approximately zero result. However this is still costly to compute. Additionally points near the walls can show divergent behavior and require a rescaling of the results. These divergent results do not spoil the simulation of the interface, unless the fluid interface is very close to the wall, they only appear in the reconstruction. However if one is interested for instance in the shear stress on the wall, this divergent behavior is inconvenient. Information on the interface has to be obtained from dedicated integration schemes that cancel out the divergent terms.

Since the previous solution is used to reconstruct the field variables the geometry must not change between the moment when the problem is solved an the moment when the field variables are calculated. Unfortunately `SolveRK1` advances the interface and thus one has three possibilities. 1) Doing the iteration oneself with `Build` and `Solve` instead of `SolveRK1`, 2) calling `mat.Build(0)` and `mat.Solve()` after `SolveRK1` and before the reconstruction routine so that a solution based on the actual geometry is obtained, or 3) integrating the call of the reconstruction routine into the `SolveRK1` in the Matrix Block.

In order to visualize the data from Matlab you can run the `ulamvisual.m` file in the `tools` folder. The Matlab data is written in a quite intuitive array format, which maybe easily adaptable for other plotting tools. A wider choice of plotting options is provided by ParaView, which reads the vtk file. In the A section one finds code examples with their respective output.

Exporting Point data

In analogy to equipping the microchannel with sensors one can probe the velocity and pressure at certain locations. At first a sensor file is prepared by: `mat.SensorEnable(int sensor id)`.

During the execution of the code one calls the procedure: `mat.SensorWrite(int sensor id, double x, double y);`, which probes the velocity and pressure at the location x, y and writes the result into the file `sensor`*n*`.txt`, where n is the sensor id.

Such output is less costly than a full visualization and is well adapted to create output at every iteration and therefore with a high temporal resolution.

APPENDIX A

Beginner tutorials

Tutorial 1 - Single phase flow in a bend

In this tutorial you will learn how to create fixed boundaries and to solve a single phase flow.

```
/*
Ulambator configuration file
Date: 09.12.2014
Version 0.7
*/
```

File Header

Together with included classes we add Ulambators `matblc1.h` and `bndblock.h`, which manage the geometry and generate the matrix that solves the posed problem.

```
#include "../source/matblc1.h"
#include "../source/bndblock.h"
#include <iostream>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <math.h>
```

The program begins with `int main` and prints a greeting to the screen.

```
int main (int argc, char * const argv[]) {
    std::cout <<"Welcome to ULAMBATOR++ v.7\n";
    char* savedir;
```

Defining the geometry

The object `bem` is a boundary block, which works as a container that will manage all boundaries like fixed walls and liquid interfaces. Here we initialize the container to contain only a single boundary.

```
    int number_of_boundaries = 1;
    BndBlock bem(number_of_boundaries);
```

Two arrays are created. Position data of the boundaries in `pos` takes up to six values and boundary conditions `bcs` usually take two values.

```
    double bcs[] = {1.0, 0.0};
    double pos[6];
```

The first boundary, boundary 0, is initialized (C++ starts counting from 0 unlike Matlab). The boundary will consist of 6 panels and the number of discrete elements `num_elements` per panel is 30 per unit length. The boundary is of type 0, type 0 means fixed boundary.

```
    int num_panels = 6;
    int num_elements = 30;
    int type_boundary = 0;
    bem.Elements[0].Init(num_panels, 24*num_elements,type_boundary);
```

The first panel of boundary 0 is initialized with `num_elements` elements and it belongs to geometry type 0, a straight panel. The boundary condition type is 3, a Poiseuille inflow boundary condition. The vector `bcs` conatisn the mean inflow velocity and the `pos` vector contains the start point x,y and end point x,y coordinates.

```
    pos[0] = 0.0; pos[1]= 1.0; pos[2]= 0.0; pos[3] = 0.0;
    int geo_type = 0;
    int bc_type = 3;
    bem.PanelInit(0,num_elements, geo_type, bc_type, bcs, pos);
```

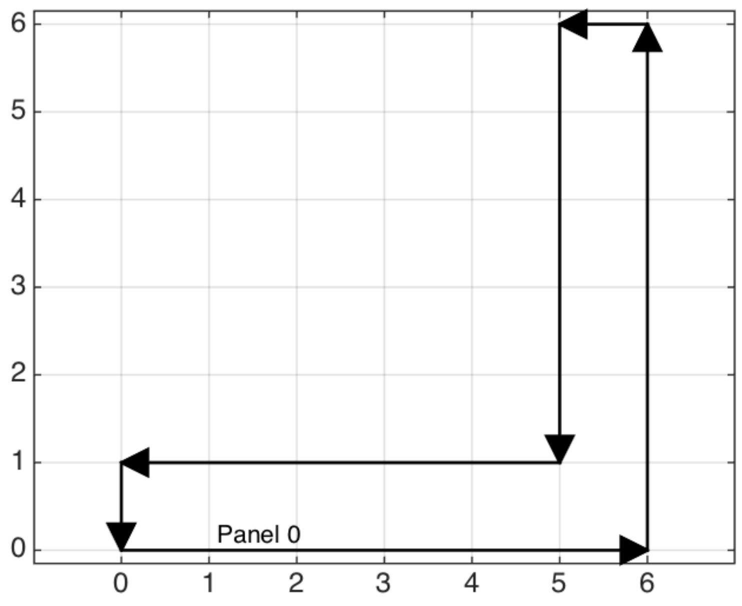


Figure A.1: Figure 1: A combination of panels forms a closed boundary. The first panel, Panel 0, is labelled. The fluid domain is on the left hand side of the arrows.

Panel 1. The boundary condition type is 0, a no-slip boundary condition. The last point of the former panel is the first point of the new panel.

```
    pos[0] = 0.0; pos[1]= 0.0; pos[2]= 6.0; pos[3] = 0.0;
    bc_type = 0;
    bem.PanelInit(0,num_elements*6, geo_type, bc_type, bcs, pos);
```

Panel 2, again with no-slip boundary condition.

```
    pos[0] = 6.0; pos[1]= 0.0; pos[2]= 6.0; pos[3] = 6.0;
    bem.PanelInit(0,num_elements*6, geo_type, bc_type, bcs, pos);
```

Panel 3, with constant normal pressure boundary condition.

```
pos[0] = 6.0; pos[1]= 6.0; pos[2]= 5.0; pos[3] = 6.0;
bc_type = 2;
bem.PanelInit(0,num_elements, geo_type, bc_type, bcs, pos);
```

Panel 4, again with no-slip boundary condition.

```
pos[0] = 5.0; pos[1]= 6.0; pos[2]= 5.0; pos[3] = 1.0;
bc_type = 0;
bem.PanelInit(0,num_elements*5, geo_type, bc_type, bcs, pos);
```

Panel 5, again with no-slip boundary condition.

```
pos[0] = 5.0; pos[1]= 1.0; pos[2]= 0.0; pos[3] = 1.0;
bem.PanelInit(0,num_elements*5, geo_type, bc_type, bcs, pos);
```

More information of different Panel types and boundary conditions is found in the Manual under PanelInit. Note that the variables can be overwritten hereafter since the information is passed on. One could also provide directly the numerical values to ‘PanelInit’, this rather explicit way was adopted to be more verbose.

Configuring the problem to solve.

At this point all boundaries and all panels are defined. This finishes the initilization of the `BndBlock` and we can create the matrix object. A matrix is created with the provided geometry bem, the aspect ratio `LH`, the viscosity ratio = zero and an unused number. A variable `LH` represents the channel aspect ratio L/H, the non-dimensional channel height is 1/`LH`.

```
double LH = 8.0;
MatBlC1 mat(&bem, LH, 0.0, 1.0);
```

The directory savedir is passed to the matrix object (and automatically to the `Bndblock`). Make sure that the folder exists.

```
savedir = new char[120];
sprintf(savedir,".");
std::cout<<savedir<<"\n"<<std::flush;
mat.EnableWrite(savedir);
```

A log file `log0.txt` is prepared with a header, if the file exists it will be deleted. If you want to append data to an existing log file don't call LogPrepare

```
mat.LogPrepare(0, "Computation time\n");
```

Write all boundaries into the savedir (or subfolder) (here: bnd0pos.dat)

```
bem.allWrite();
```

Save time step, time in the simulation and computation time to `log0.txt`

```
mat.LogRuntime(0);
```

Build, Solve and visualize the solution

Save the flowfield to a ParaView readible vtk file required to run `Build` and `Solve` before. Build the matrix, provide a timestep for stabilization (set deltaT=0 for no stabilization). Then solve the matrix to obtain stresses and velocities. The file will be called `ulam2d000.vtk`, a unit length is discretized with 30 points and the visualized window ranges from x=0 to 6 and y=0 to 6.

```
mat.Build(0.0);
mat.Solve();
mat.LogRuntime(0);
mat.VisMatlab(0, 50, 0, 6, 0, 6);
mat.LogRuntime(0);
mat.VTKexport(0, 50, 0, 6, 0, 6);
```

This provides a flowfield, limiting of the velocity and pressure may be required because the integrants are singular at the boundary and the integrals may diverge there.

A last log of the time. Make sure all files are closed to avoid data loss.

```
    mat.LogRuntime(0);
    mat.DisableWrite();
```

End of program

```
    std::cout << "Finished and Exiting\n";
    return 0;
}
```

Results

When you run the file as it is you obtain an output that can be plotted in ParaView. Here we plot the pressure field from `ulam2d000.vtk` and have ParaView compute the streamlines from the velocity field.

We use `ulamgeo2d000.vtk` to plot the outer walls in black.

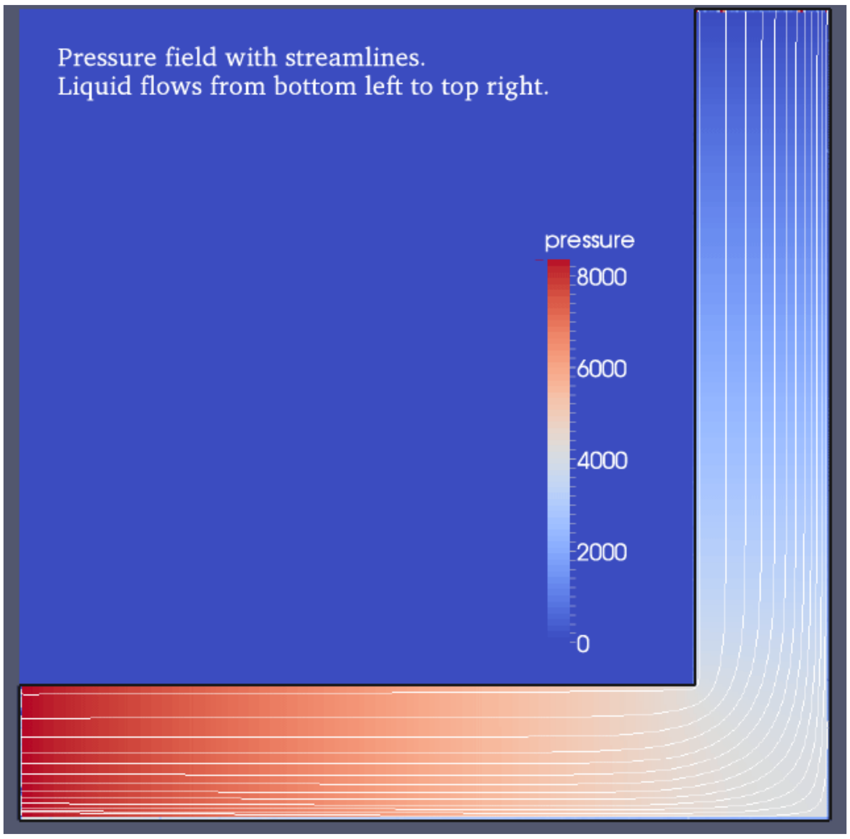
The pressure drops from about 8200 at the inlet to 0 at the outlet. An analytical result for the 2D Brinkman equation for straight channel flow at aspect ratio $W/H = 8$ would have given 827.7 per unit length. The analytical result for the 3D Stokes equation in a straight channel would have given 833.7, quite similar. When counting only the two straight segments whose length is 10 the theoretical predicted pressure drop is in the order of 8300.

We now re-equip these values with units. The dimension of the pressure is given as $P = \mu U/L$. In this example we say the viscosity is that of water $\mu = 1$ milliPas, the mean in flow velocity $U = 5mm/s$ and the channel width $W = 96\mu m$. Since the non-dimensional channel width is 1, the channel height is $12\mu m$, which verifies W/H = 8. The aspect ratio and channel geometry was used in the numerical calculation to obtain the non-dimensional pressure drop $\Delta p = -8200 \Delta P = P\,\Delta p = 0.0521\,8200 Pa = 427 Pa$ or 4.27milli bars.

Tutorial 2 - Droplet relaxation

In this tutorial you will learn how to create a liquid interface and let it evolve in time. To this purpose an elliptical interface is created and a time marching follows the relaxation of the droplet.

```
/*
Ulambator configuration file
```



```
Date: 09.12.2014
Version 0.7
*/
```

File Header

Together with included classes we add Ulambators `matblc1.h` and `bndblock.h`, which manage the geometry and generate the matrix that solves the posed problem.

```
#include "../source/matblc1.h"
#include "../source/bndblock.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <math.h>
```

The program begins with `int main` and prints a greeting to the screen.

```
int main (int argc, char * const argv[]) {
    std::cout <<"Welcome to ULAMBATOR++ v.7\n";
    char* savedir;
```

Defining the geometry

The object `bem` is a boundary block, which works as a container that will manage all boundaries like fixed walls and liquid interfaces. Here we initialize the container to contain only a single boundary.

```
    int number_of_boundaries = 1;
    BndBlock bem(number_of_boundaries);
```

We initilize three variables that allow parametrization from the command line. A variable that represents the channel aspect ratio L/H, the non-dimensional channel height is 1/`LH`. The eccentricity of the elliptical droplet and a folder identification number.

```
    double LH = 8.0;
    double eccentricity = 0.95;
    int folderid = 0;
```

Here the program catches the numbers that were passed when it was called eg. `./ulambator_main.o 1 5 0.5` sets the `folderid=1, LH=5, eccentricity=0.5`

```
    if (argc>3) {
        folderid = atof(argv[1]);
        LH = atof(argv[2]);
        eccentricity = atof(argv[3]);
    }
```

Output to the screen of the parameters.

```
    std::cout<<"Elliptic Droplet Simulation, L/H "<<LH;
    std::cout<<" eccentricity "<<eccentricity<<" folder no. "<<folderid<<".\n";
```

Two arrays are created. Position data of the boundaries in `pos` takes up to six values and boundary conditions `bcs` usually take two values.

```
double bcs[] = {0.0, 0.0};
double pos[6];
```

The first boundary, boundary 0, is initialized (C++ starts counting from 0 unlike Matlab). The boundary will consist of one single panel and the number of discrete elements `num_elements` is 300. We chose to reserve twice as much memory if due to deformation and remeshing the number of unknowns increases. The boundary is of type 2, type 2 means a droplet boundary.

```
int num_panels = 1;
int num_elements = 300;
int type_boundary = 2;
bem.Elements[0].Init(num_panels, 2*num_elements,type_boundary);
```

The first panel of boundary 0 is initialized with `num_elements` elements and it belongs to geometry type 3, an ellipse. The boundary condition type is 21, a liquid interface. The array `bcs` has no influence and the `pos` array contains the center coordinate x = 0, y=0, major axis=1, minor axis according to eccentricity.

```
pos[0] = 0.0; pos[1]= 0.0; pos[2]= 1.0;
pos[3] = sqrt(1.0-eccentricity*eccentricity);
int geo_type = 3;
int bc_type = 21;
bem.PanelInit(0,num_elements, geo_type, bc_type, bcs, pos);
```

More information of different Panel types and boundary conditions is found in the Manual under PanelInit. Note that the variables can be overwritten hereafter since the information is passed on. One could also provide directly the numerical values to 'PanelInit', this rather explicit way was adopted to be more verbose.

Configuring the problem to solve.

At this point all boundaries and all panels are defined. This finishes the initilization of the `BndBlock` and we can create the matrix object. A matrix is created with the provided geometry bem, the aspect ratio LH, the viscosity ratio and unused number. The viscosity ratio is $\lambda = \mu_{inside}/\mu_{outside}$.

```
double lambda = 0.2;
MatBlC1 mat(&bem, LH, lambda, 0.0);
```

The directory savedir is passed to the matrix object (and automatically to the `Bndblock`). Make sure that the folder exists.

```
savedir = new char[120];
sprintf(savedir,".");
mat.EnableWrite(savedir);
```

If a `folderid` bigger than 0 is provided the following code will create a subfolder in `savedir` is created. The subfolder will be named "test" followed by a number. If the folder already exist the program is stopped to prevent overwriting.

```
if (folderid>0) {
    sprintf(savedir,"test%d", folderid);
    mat.SubFolder(savedir);
}
```

A log file `log0.txt` is prepared with a header, if the file exists it will be deleted. If you want to append data to an existing log file don't call LogPrepare

```
mat.LogPrepare(0, "Computation time\n");
```

Prepare another logfile without header. If the file does not exist you don't need to call LogPrepare

```
mat.LogPrepare(1, "Cardinal Curvatures\n");
```

Write all boundaries into the savedir (or subfolder) (here: bnd0pos.dat and drop1ts0.dat)

```
    bem.allWrite();
```

Turn remeshing on, droplets will be remeshed with a resolution $2\pi/N$ when one or more elements become twice or half as long this reference size.

```
    bem.RemeshOn(2.0*M_PI/num_elements, 0);
```

Save time step, time in the simulation and computation time to `log0.txt`

```
    mat.LogRuntime(0);
```

Solving the problem by time stepping

Set the timestep `deltaT` to 0.25 and set 10 iterations between output. An outer iteration loop is performed `nnSteps` times. A total of `nSteps*nnSteps` are done and the simulated time is `deltaT*nSteps*nnSteps`.

```
    double deltaT = 0.25;
    int nSteps = 10;
    int nnSteps = 50;
    for (int j=1; j<=nnSteps; j++) {
        for (int k=1; k<=nSteps; k++) {
```

Solving the problem with a one-step explicit time integration scheme. The `SeedDrop()` function checks and remeshes the interface if needed. `UpdateNOFM()` adapts the matrix and right hand side size. Then the matrix is build and solved.

```
            bem.SeedDrop();
            mat.UpdateNOFM();

            mat.Build(deltaT);
            mat.Solve();
```

On the last iteration the velocity field is computed before the interface is advanced. `VTKexport`writes to file j, with a resolution 50 points per unit length in a window of x=-1.2 … 1.2, y=-1 … 1. Alternatively comment VTKexport and uncomment VisMatlab.

```
            if (k==nSteps) {
                mat.VTKexport(j, 50, -1.2, 1.2, -1.0, 1.0);
        // mat.VisMatlab(j, 50, -1.2, 1.2, -1.0, 1.0);
            }
```

The interface is advanced by a time step `deltaT` with a 1-step scheme. Thereafter the curvatures at the cardinal points (north, south, east, west) are written to a `log1.txt`.

```
            mat.AdvanceTimestep(deltaT, 1);
            bem.LogCustom(1,3);
            std::cout<<"."<<std::flush;
        }
```

Increase the iteration counter and write the interface geometry

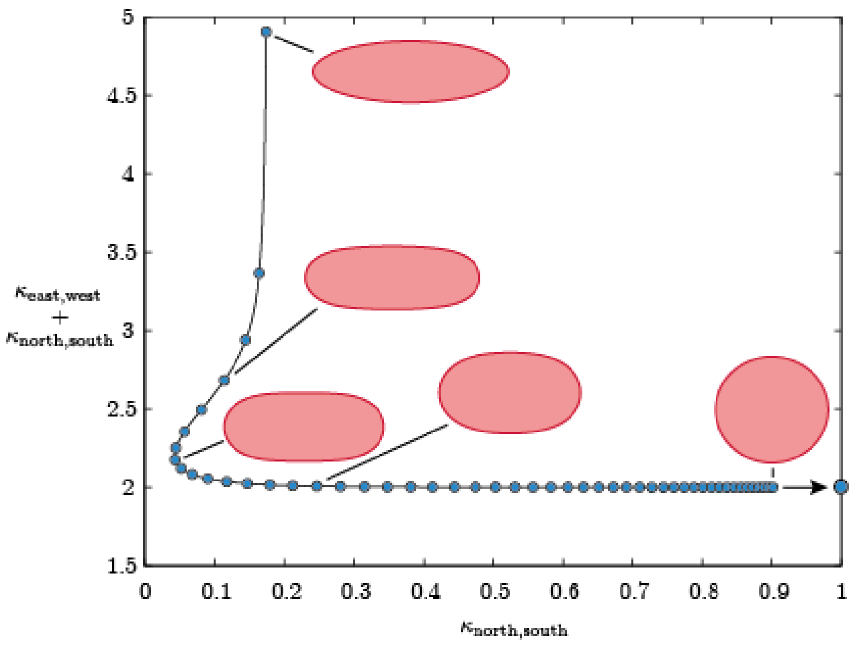
```
        bem.runstep++;
        bem.allWrite();
        mat.LogRuntime(0);
    }
```

A last log of the time. Make sure all files are closed to avoid data loss.

```
    mat.LogRuntime(0);
    mat.DisableWrite();
```

End of program

```
    std::cout << "Finished and Exiting\n";
    return 0;
}
```



Results

We now look at two dimensional properties. The pressure difference inside/outside the relaxed circular droplet and the time needed for relaxation. The pressure is dimensionalized with $P = \gamma/L$. The final droplet will have a radius 1 non-dimensional units. If you consider an air bubble or radius $100\mu m$ in water (although the viscosity ratio is not appropriate), the surface tension is $\gamma = 72$ milli Pam and hence the pressure scale $P = 720Pa$.

In the example the relaxed droplet should have an inner pressure about $2/RH + \pi/4$ non-dimensional pressure units higher than the outer phase, which is also found numerically. Hence the pressure jump for $RH = 8$ is $p = 12085Pa$ or $0.12bars$. The main contribution comes from the curvature in the thin channel direction, where $H = 12.5\mu m$ and thus the radius of curvature is $6.25\mu m$! Laplace law determines the pressure jump as surface tension times curvature: $\Delta p = \gamma\kappa$, with $\kappa = 1/6.25\mu m$.

In order to determine the relaxation time one needs a criteria when to consider the droplet as relaxed, because it tends asymptotically to a circle and theoretically never reaches the completely circular state. A very small eccentricity could be such a criteria. In this example we consider that the droplet is relaxed after 50 times 10 iterations. The time step was 0.25, so the final non-dimensional time is 125.

The time scale is $T = L/U$ with $U = \gamma/mu$, that is $T = L\mu/\gamma$. Assuming as before water in the continuous phase $T = 100\mu m \cdot 1\text{milli}Pas/72\text{milli}Pam = 1.39\mu s$. So the relaxation happens effectively after $t = 0.17$milli seconds.

In the context of a publication we plot also the cardinal curvatures that has been logged by mat.LogCostom(1,3). Column 2 and 4 in log1.txt contain the curvatures at every timestep. We normalize the curvature by the curvature of the circular interface. The north and east curvature tends to 1, hence the final state in the diagram is (2,1), however we observe that sum of curvatures equal 2 is attained much earlier.

P.-T. Brun, Mathias Nagel, and François Gallaire
Generic path for droplet relaxation in microfluidic channels

Phys. Rev. E **88**, 043009, 2013.

Tutorial 3 - Droplet stretching

This tutorial shows interaction of outer boundaries and a liquid interface. Droplet stretching in a cross flow geometry as published in experiments by Ulloa et al. is simulated.

```
/*
 Ulambator configuration file
 Date: 19.12.2014
 Version 0.7
*/
```

File Header

```
#include <iostream>
#include <stdio.h>
#include "../source/matblc1.h"
#include "../source/bndblock.h"
#include <sys/time.h>
#include <time.h>
#include <math.h>
#include <stdlib.h>

int main (int argc, char * const argv[]) {
    std::cout<<"Welcome to ULAMBATOR++ v0.7\n";
    char* savedir;
    savedir = new char[120];
    sprintf(savedir,".");
```

Reading problem parameters

A folder id to number the output folder, the viscosity ratio $\lambda = \mu_{drop}/\mu_{bulk}$, theoretical shear rate G and droplet radius rdrop. The aspect ratio is $200\mu m$ over $58\mu m$.

```
    int folderid = 0;
    double lambda = 0.008;
    double G = 6.5;
    double WH = 3.45;
    int num_elements = 400;
```

The radius is given in micrometers and non-dimensionalized later with the half channel width. Ulloa et al. used two different kind of microchannels, here we configure channel II with a width of $400\mu m$. The non-dimenseional channel width is 2, thus one unit corresponds to $200\mu m$.

```
double width = 400.0;
double rdrop = 100.0;
```

Reading in data from command line, then non-dimensionalizing the droplet radius.

```
if (argc>4) {
    folderid = atoi(argv[1]);
    WH = atof(argv[2]);
    G =  atof(argv[3]);
    rdrop = atof(argv[4]);
}
rdrop = 2*rdrop/width;
```

The theoretical shear rate G is transformed into a inflow velocity: $u = G \cdot W$, where the non-dimensional velocity is $U = \frac{\gamma}{\mu}$. So the non-dimensional inflow rate defines a capillary number $Ca = \frac{\mu G W}{\gamma}$. The viscosity of the continuous phase is $120mPas$ and the surface tension $48mPam$.

```
double Ca = G*1e-6*width*120.0/48;

std::cout<<"folder:"<<folderid<<" W/2H:"<<WH<<" G:"<<G<<" Rd:"<<rdrop<<"\n";
```

Defining the geometry

A boundary block with two boundries is created.

```
BndBlock bem(2);
```

Decide to use boundaries or an infinite hyperbolic flow profile without outer boundaries. Without outer boundaries it is faster but requires a correction from the theoretical shear rate in the channel to the shear rate of the infinite flow.

```
    double bcs[] = {Ca, 0.0};
    double pos[6];
    int no_walls = 0;

    if (no_walls==1) {
        std::cout<<"In infinite linear flow\n";
        bcs[0] = -Ca/1.1;
        bem.Elements[0].Init(1, 10, 0);
        bem.PanelInit(0, 0, 0, 7, bcs, pos);
    } else {
        std::cout<<"In a cross channel flow\n";
```

Initiate a boundary with 12 panels and 2200 elements. The final zero sets fixed outer boundaries. Following entries set the panel coordinates and initialize every panel.

```
        bem.Elements[0].Init(12, 2200, 0);

        pos[0] = -6; pos[1]= 1.0; pos[2]= -6.0; pos[3] = -1.0;
        bem.PanelInit(0,100, 0, 3, bcs, pos);   // an inflow panel

        pos[0] = -6.0; pos[1]= -1.0; pos[2]= -1.0; pos[3] = -1.0;
        bem.PanelInit(0,250, 0, 0, bcs, pos);

        pos[0] = -1.0; pos[1] = -1.0; pos[2] =-1.0; pos[3]= -4.0;
        bem.PanelInit(0,150, 0, 0, bcs, pos);

        bcs[0] = 0.0;
        pos[0] = -1.0; pos[1]= -4.0; pos[2]= 1.0; pos[3] = -4.0;
        bem.PanelInit(0,100, 0, 2, bcs, pos);   // an outflow panel

        pos[0] = 1.0; pos[1]= -4.0; pos[2]= 1.0; pos[3] = -1.0;
        bem.PanelInit(0,150, 0, 0, bcs, pos);

        pos[0] = 1.0; pos[1]= -1.0; pos[2]= 6.0; pos[3] = -1.0;
        bem.PanelInit(0,250, 0, 0, bcs, pos);

        bcs[0] = Ca;
        pos[0] = 6.0; pos[1]= -1.0; pos[2]= 6.0; pos[3] = 1.0;
        bem.PanelInit(0,100, 0, 3, bcs, pos);   // an inflow panel

        pos[0] = 6.0; pos[1]= 1.0; pos[2]= 1.0; pos[3] = 1.0;
        bem.PanelInit(0,250, 0, 0, bcs, pos);
        pos[0] = 1.0; pos[1]= 1.0; pos[2]= 1.0; pos[3] = 4.0;
        bem.PanelInit(0,150, 0, 0, bcs, pos);

        bcs[0] = 0.0;
        pos[0] = 1.0; pos[1]= 4.0; pos[2]= -1.0; pos[3] = 4.0;
        bem.PanelInit(0,100, 0, 2, bcs, pos);   // an outflow panel

        pos[0] = -1.0; pos[1]= 4.0; pos[2]= -1.0; pos[3] = 1.0;
        bem.PanelInit(0,150, 0, 0, bcs, pos);

        pos[0] = -1.0; pos[1]= 1.0; pos[2]= -6.0; pos[3] = 1.0;
        bem.PanelInit(0,250, 0, 0, bcs, pos);
    }
```

Initialize the droplet interface, with one panel, with an excess number of elements and a droplet, type 2.

```
    bem.Elements[1].Init(1,2*num_elements,2);
```

The initial position $x = -4, y = 10^{-6}$ facilitates the drift from the center position. We initialize boundary 1, `num_elements` elements, geometry type 2 (a circle), boundary condition type 21 (a droplet) and the geometric specifications pos and boundary values bc (not used).

```
    pos[0] = -4.0; pos[1]= 1e-6; pos[2]= rdrop; pos[3] = 0.0; pos[4] = 1.0;
    bem.PanelInit(1,num_elements , 2, 21, bcs, pos);
```

Remeshing is activated with standard density according to method 0 (default, homogeneous distribution). The initial area of boundary is saved and maintained.

```
    bem.RemeshOn(2*M_PI*rdrop/num_elements, 0);
    bem.Elements[1].initialArea = bem.getArea(1);
```

Setting up the matrix and solver properties

The matrix object is created with the geometry object, aspect ratio `WH`, viscosity ratio `lambda` and cpaillary number `Ca`. the capillary number is only passed to the `LogMatlab()` function and does not influence the simulation.

```
MatBlC1 mat(&bem, WH, lambda, Ca);
```

Enable writing and prepare log files.

```
mat.EnableWrite(savedir);
char *title;
title = new char[120];
if(folderid>0){
    sprintf(title,"cordero%d", folderid);
    mat.SubFolder(title);
}

sprintf(title,"Machine Time\n");
mat.LogPrepare(0, title);
mat.LogPrepare(1, "");

bem.LogTimeStep(0); // Log simulation time (no line break)
mat.LogRuntime(0);  // Log real time
bem.LogCustom(1,2);  // to logfile 1, with option 2
mat.LogMatlab();
bem.allWrite();
```

To accelerate the simulation in the presence of outer walls the invariant parts of the matrix can be inverted to allow a Gauss block algorithm to decrease the final matrix size.

```
mat.PreCondense();
```

Solving the problem by time stepping

Set the timestep `deltaT` to 0.5 and set 10 iterations between output. An outer iteration loop is performed `nnSteps` times. A total of `nSteps*nnSteps` are done and the simulated time is `deltaT*nSteps*nnSteps`.

```
double deltaT = 0.25;
int nSteps = 20;
int nnSteps = 110;
mat.setTimeStep(deltaT,nSteps);
```

Integrate in time with a Euler explicit one step scheme. Then log data and write geometry. No field variables output is produced, in this version it is incompatible with `PreCondense()`.

```
    for (int j=0; j<nnSteps; j++) {
        mat.SolveRK1();
        bem.LogTimeStep(0);
        mat.LogRuntime(0);
        bem.LogCustom(1, 2);
        bem.allWrite();
    }

    mat.LogRuntime(0);
    mat.DisableWrite();
```

End of program

```
    std::cout << "Finished and Exiting\n";
    return 0;
}
```

Results

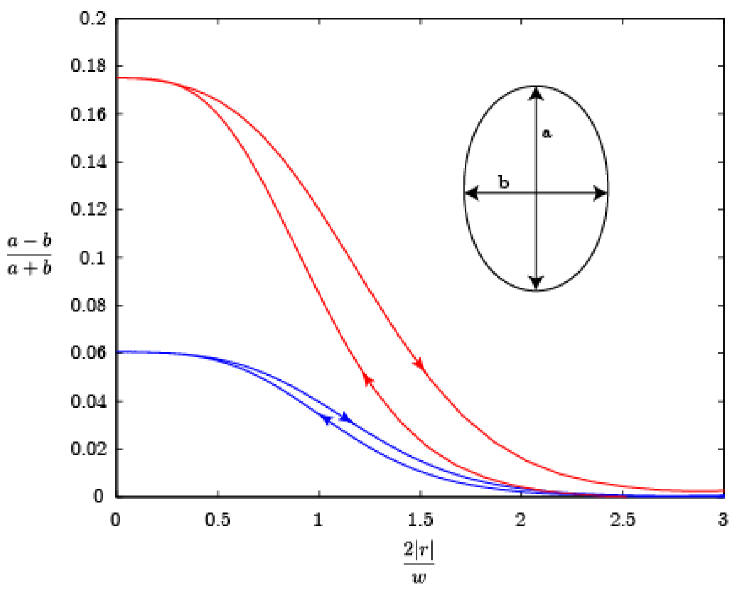
Principle result is the deformation of the droplet in a shear flow. The deformation is defined as the difference of the major and minor axis over the sum of major and minor axis. The blue curve corresponds to the predefined case with $G = 6.5s^{-1}, R = 100\mu m$ (corresponds to Figure 4 (a) in Ulloa et al.). The red curve is for $G = 18s^{-1}$ and $R = 100\mu m$.

Camilo Ulloa, Alberto Ahumada and María Luisa Cordero
Effect of confinement on the deformation of microfluidic drops
Phys. Rev. E **89**, 033004, 2014.

Tutorial 4 - Droplet anchoring

This tutorial shows how to use a model boundary condition on the liquid interface that incorporates the presence of an indentation in the channel floor as studied experimentally by Dangla et al. It is furthermore shown how to simulate several droplets.

```
/*
 Ulambator configuration file
 Date: 10.02.2014
```



```
Version 0.7
*/
```

File Header

```
#include <iostream>
#include <stdio.h>
#include "../source/matblc1.h"
#include "../source/bndblock.h"
#include <sys/time.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>


int main (int argc, char * const argv[]) {
    std::cout<<"Welcome to ULAMBATOR++ v0.7\n";
    std::cout<<"Unsteady LAminar Microfluidic Boundary element Algorithm To Obtain Results\r
```

Reading problem parameters

Working directory, viscosity ratio, capillary number, aspect ratio and a folder id.

```
    char* savedir;
    savedir = new char[120];
    sprintf(savedir,".");
    double lambda = 0.1;
    double Ca = 0.001;
    double RH = 3.0;
    int folderid = 0;
```

Additionally the radius of the indentation `r_hole` is provided. Here we set `r_hole` equal to the channel height.

```
    double r_hole = 1.0/RH;
```

Optional reading of data from command line.

```
    if (argc>5) {
        folderid = atoi(argv[1]);
        RH = atof(argv[3]);
        Ca = atof(argv[2]);
        lambda = atof(argv[4]);
        r_hole = atof(argv[5]);
    }
    std::cout<<"Configuration folder:"<<folderid<<" R/H:"<<RH<<" Ca:"<<Ca<<"\n";
    std::cout<<"lambda:"<<lambda<<" and indentation radius:"<<r_hole<<"\n";
```

Setting up the geometry

Three boundaries, one for the outer flow and two for the droplets.

```
    BndBlock bem = BndBlock(3);
```

Create variables for geometric and boundary value information and initialize the first boundary with 1 panel. The first boundary will be an open domain with constant flow at infinity.

```
    double pos[6];
    double bcs[2];
    int bnd_id = 0;      // first boundary
    int nb_panels = 1;   // number of panels for this boundary
    int nb_elements=0;   // boundary at infinity needs no elements
    int bnd_type=0;      // a fixed wall
    bem.Elements[bnd_id].Init(nb_panels, nb_elements, bnd_type);
```

Initialize panel 0 of boundary 0 as a boundary at infinity.

```
    int wall_type = 0;       // a straight line wall, but unused since there are zero elemen
    int bc_type = 5;         // constant flow at infinity
    bcs[0] = Ca;         // velocity at infinity in x-direction
    bcs[1] = 0.0;        // no vertical velocity component
    bem.PanelInit(bnd_id, nb_elements, wall_type, bc_type, bcs, pos);
```

Initialize boundary 1 as a liquid interface. Twice as much memory is reserved to be on the save side when the deformed drop is remeshed.

```
    bnd_id = 1;
    nb_elements = 500;
    bnd_type = 2;
    bem.Elements[bnd_id].Init(nb_panels,2*nb_elements,bnd_type);
```

Specify the geometric parameters for the circular liquid interface: x, y, radius, perpurbation amplitude and perturbation wave number. Here a droplet with radius one is created at the origin. The interface has no perturbation.

```
    pos[0] = 0.0; pos[1]= 0.0; pos[2]= 1.0; pos[3] = 0.0; pos[4] = 1.0;
    wall_type = 2;       // a circle
    bc_type = 21;        // a liquid-liquid interface
    bem.PanelInit(bnd_id,nb_elements, wall_type, bc_type, bcs, pos);
```

Initialize boundary 2 also as a liquid interface. Twice as much memory is reserved to be on the save side when the deformed drop is remeshed.

```
bnd_id = 2;
nb_elements = 500;
bnd_type = 2;
bem.Elements[bnd_id].Init(nb_panels,2*nb_elements,bnd_type);
pos[2] = 2.01;
bem.PanelInit(bnd_id,nb_elements, wall_type, bc_type, bcs, pos);
```

Save the initial area to enforce area conservation.

```
bem.Elements[1].initialArea = bem.getArea(1);
bem.Elements[2].initialArea = bem.getArea(2);
```

Matrix initialization

The matrix object is created and a pointer to the geometry is passed as well as the aspect ratio, viscosity ratio and capillary number.

```
MatBlC1 mat(&bem, RH, lambda, Ca);
mat.EnableWrite(savedir);
```

The model boundary condition for the indentation, option 3, is set up. One parameter is passed, which is the radius of the identation. All other parameters are as the effective depth of the hole are calculated from a model that we presented in an article.

```
mat.ChoseStatBC(3, r_hole, 0.0);
```

A Subfolder is created if the index is bigger than zero.

```
if (folderid>0) {
    sprintf(savedir,"casestudy%d", folderid);
    mat.SubFolder(savedir);
}
```

Log files are prepared.

```
mat.LogPrepare(0, "Machine time\n");
mat.LogPrepare(1, "Drop 1: Area and X,Y Position\n");
mat.LogPrepare(2, "Drop 2: Area and X,Y Position\n");
```

Initial lof files and boundaries are written.

```
mat.LogCfg();                // configuration
bem.LogTimeStep(0);          // step number and time of the simulation
mat.LogRuntime(0);           // real time
bem.LogAreaPos(1,1);         // area and position of drop 1 to file 1
bem.LogAreaPos(2,2);         // area and position of drop 2 to file 2
bem.allWrite();              // geometry
```

Time stepping is specified and remeshing activated.

```
double deltaT = 1.0;
int subtimesteps = 50;
mat.setTimeStep(deltaT,subtimesteps);
bem.RemeshOn(6.28/nb_elements,0);
```

Time marching

The internal loop in `SolveRK2` is repeated 100 times, then the boundary and log files are written.

```
for(int k=0; k<100; k++){
    mat.SolveRK2();
    bem.allWrite();
    bem.LogTimeStep(0);
    mat.LogRuntime(0);
    bem.LogAreaPos(1,1);
    bem.LogAreaPos(2,2);
}
```

End of the program.

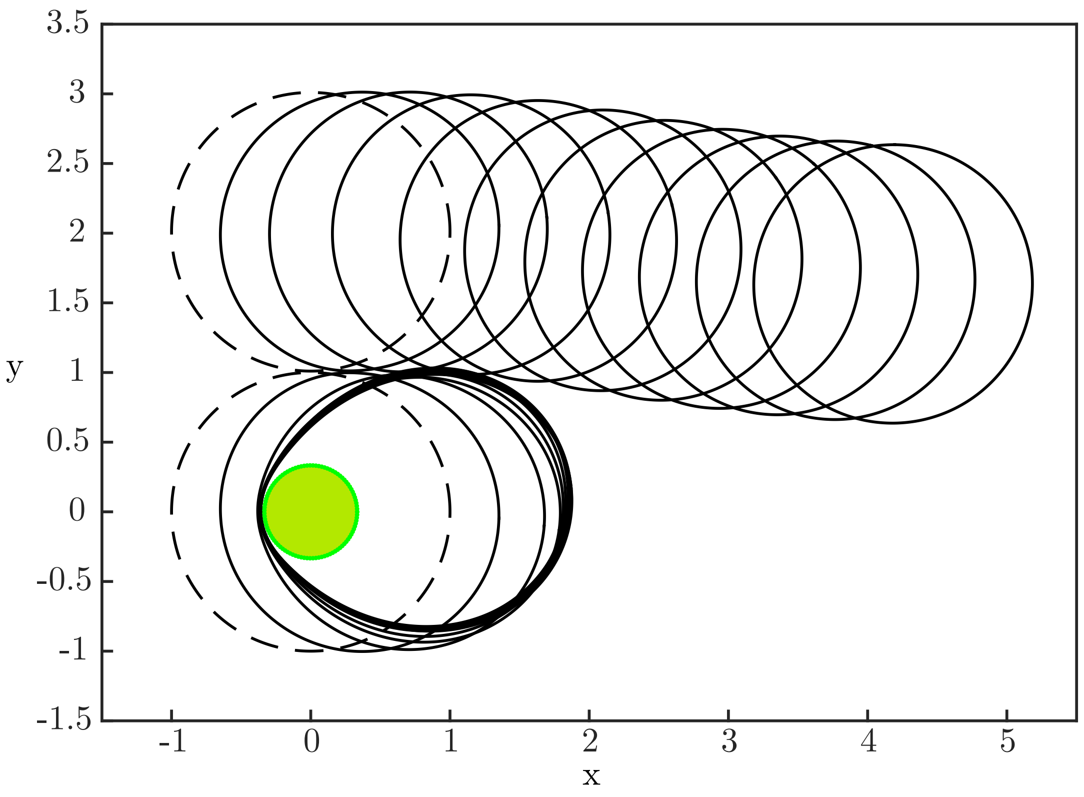


Figure A.2: Two droplets streamed from left to right, initial position marked by a dashed line. The lower droplet is a anchored on a indentation, while the upper droplet continues its motion.

```
    mat.DisableWrite();
    std::cout << "Finished and Exiting\n";
    return 0;
}
```

Results

In this simulation two droplets are emitted to a flow at low capillary number. Both droplets have a radius of 1 and are one hundredth of their radius apart. One of the droplets will pass over a indentation in the channel ceiling, which functions as a capillary anchor. One observes in the time lapse image, figure A, that the lower of the two droplets is stopped while the upper droplet continues its way. Due to the small distance between the droplets there is a hydrodynamic coupling. As the lower droplet is stopped the thin gap between both droplets is filled with liquid, which induces a pressure drop and therefore leads to an attraction between both droplets.

We see in figure A that the lower droplet becomes anchored and remains fixed. The displacement of the droplets relative to their initial position in the y coordinate (b) shows that the droplets first show a slight repulsion, maybe induce by deformation

R. Dangla, S. Lee and C. N. Baroud

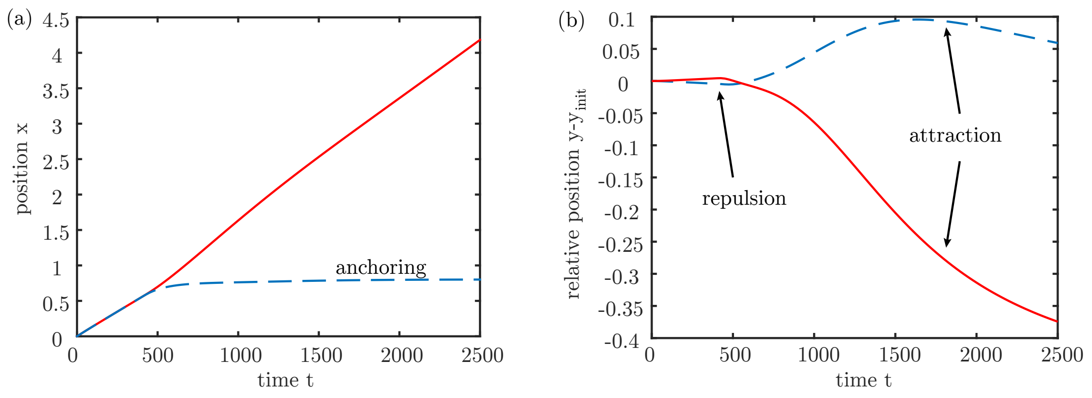


Figure A.3: Droplet displacement against time, upper droplet marked by a dashed red line and lower droplet marked by a solid blue line. (a) Center of mass moving in x direction with time, (b) center of mass relative to the initial position in y direction.

Trapping Microfluidic Drops in Wells of Surface Energy
Physical Review Letters **107** (12), 124501, 2011.

M. Nagel, P.-T. Brun and F. Gallaire
A numerical study of droplet trapping in microfluidic devices Physics of Fluids, 26(3), 032002, 2014.

Tutorial 5 - A posterior visualization

This tutorial shows how to restart a simulation from a previously saved configuration. Using the reading of a previous configuration we will compute a posteriori the flow field from a simulation that is already finished. The exectuable needs to be copied into the folder where the initial simulation took place.

```
/*
 Ulambator configuration file
 Date: 24.2.2014
 Version 0.7
*/
```

File Header

```
#include <iostream>
#include <stdio.h>
#include "../source/matblc1.h"
#include "../source/bndblock.h"
#include <sys/time.h>
```

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>


int main (int argc, char * const argv[]) {
    std::cout<<"Welcome to ULAMBATOR++ v0.7\n";
    std::cout<<"Unsteady LAminar Microfluidic Boundary element Algorithm To Obtain Results\
```

Reading problem parameters

Working directory, viscosity ratio, capillary number, aspect ratio and a folder id. Since we restart a previous simulation these parameters should match those of the initial simulation.

```

    char* savedir;
    savedir = new char[120];
    sprintf(savedir,".");
    double lambda = 0.1;
    double Ca = 0.001;
    double RH = 3.0;
    int folderid = 0;
```

Additionally the radius of the indentation `r_hole` is provided. Here we set `r_hole` equal to the channel height.

```

    double r_hole = 1.0/RH;

    std::cout<<"Configuration folder:"<<folderid<<" R/H:"<<RH<<" Ca:"<<Ca<<"\n";
    std::cout<<"lambda:"<<lambda<<" and indentation radius:"<<r_hole<<"\n";
```

Setting up the geometry

Three boundaries, one for the outer flow and two for the droplets.

```

    BndBlock bem = BndBlock(3);
```

Create variables for geometric and boundary value information and initialize the first boundary with 1 panel. The first boundary will be an open domain with constant flow at infinity.

```
double pos[6];
double bcs[2];
int bnd_id = 0;     // first boundary
int nb_panels = 1;  // number of panels for this boundary
int nb_elements=0;  // boundary at infinity needs no elements
int bnd_type=0;     // a fixed wall
bem.Elements[bnd_id].Init(nb_panels, nb_elements, bnd_type);
```

Initialize panel 0 of boundary 0 as a boundary at infinity.

```
int wall_type = 0;      // a straight line wall, but unused since there are zero elemen
int bc_type = 5;        // constant flow at infinity
bcs[0] = Ca;        // velocity at infinity in x-direction
bcs[1] = 0.0;       // no vertical velocity component
bem.PanelInit(bnd_id, nb_elements, wall_type, bc_type, bcs, pos);
```

Initialize boundary 1 as a liquid interface.

```
bnd_id = 1;
nb_elements = 500;
bnd_type = 2;
bem.Elements[bnd_id].Init(nb_panels,2*nb_elements,bnd_type);
```

Writing is enabled in order to provide a folder from where the data is read. The droplet geometry of droplet 1 is read from a previous timestep, here timestep 0.

```
bem.EnableWrite(savedir);
int timestep = 0;
bem.DropRead(bnd_id, timestep);
```

Initialize boundary 2 also as a liquid interface.

```
bnd_id = 2;
nb_elements = 500;
bnd_type = 2;
bem.Elements[bnd_id].Init(nb_panels,2*nb_elements,bnd_type);
bem.DropRead(bnd_id, timestep);
```

Matrix initialization

The matrix object is created and a pointer to the geometry is passed as well as the aspect ratio, viscosity ratio and capillary number.

```
MatBlC1 mat(&bem, RH, lambda, Ca);
mat.EnableWrite(savedir);
```

The model boundary condition for the indentation, option 3, is set up. One parameter is passed, which is the radius of the identation. All other parameters are as the effective depth of the hole are calculated from a model that we presented in an article.

```
mat.ChoseStatBC(3, r_hole, 0.0);
```

The time step for stabilization is equal to that in the initial simulation.

```
double deltaT = 1.0;
```

Remeshing is activated not for remeshing purposes but because the approximate inter point distance serves to avoid divergent points near the interface. Points closer to the interface than the inter point distance are not evaluated by integration but are assigned the values obtained on the interface.

```
bem.RemeshOn(6.28/nb_elements,0);
```

Post–Processing

The Visual output is generated for all 100 times. The droplet geometry is loaded, the unknowns determined and then the flow field constructed.

```
    for(int k=0; k<=100; k++){
        bem.DropRead(1, k);
        bem.DropRead(2, k);
        mat.Build(deltaT);
        mat.Solve();
        mat.VTKexport(k, 40, -1, 3, -1, 3);
    }
```

End of the program.

```
    mat.DisableWrite();
    std::cout << "Finished and Exiting\n";
    return 0;
}
```

This post–processing routine fits to the simulation in tutorial 4. After compiling tutorial 5 copy the executable into the folder, where the results of the simulation from tutorial 4 are saved. Make sure that the parameters like capillary number, aspect ratio etc. are the same. Then run the code. You can view the resulting `ulam2d`NNNN`.vtk` and `ulamgeo`NNNN`.vtk` with ParaView.

Bibliography

[1] M. Nagel and F. Gallaire. Boundary elements method for microfluidic two-phase flows in shallow channels. *Computers & Fluids*, 107(0):272 – 284, 2015.