The incomplete Ulambator Reference
The Source Code Appendix

Ulambator version 0.7
Manual version 0.73

December 2, 2015

Abstract

This appendix holds the commented and hopefully reader friendly version of the source code. Commenting is still ongoing but we hope that this document provides a reference to the user.

Contents

Contents **1**
Ulambators MatrixBlock class (`matrixblock.cpp`) v0.3 2
Ulambators BoundaryBlock class (`bndblock.cpp`) v0.3 58
Ulambator BoundaryElement class (`bndelement.cpp`) v0.3 87
Further classes 89

Ulambators MatrixBlock class (`matrixblock.cpp`) v0.3

This class contains functions to create and solve a matrix of the discretized Brinkan equation. In the following the functions have been regrouped into the categories:

Contents:

- Initilisation procedure
- Matrix creation
- Dynamic and static boundary conditions
- Problem solving
- Time marching
- Writing data
- Field data export
- Ubiquitous subroutines

The header optionally contains OpenMP support.

```
#include "matrixblock.h"
#include <stdio.h>
#include <iostream>
#include <algorithm>
#include <fstream>
#include <sys/stat.h>
#include <sys/types.h>
#ifdef _WIN32
#define _USE_MATH_DEFINES // for C++
#include <cmath>
#endif
#include <math.h>
#include "combase.h"
#include <iomanip>
#ifndef _WIN32
#include <sys/time.h>
#endif
#ifdef _WIN32
#include <direct.h>
#include <Winsock2.h>
#include <Windows.h>
#endif
#include <time.h>
#include <assert.h>
#include <string.h>
```

```
#if defined (_OPENMP)
    #include <omp.h>
#endif

using namespace std;
```

Initialization

```
MatrixBlock::MatrixBlock(BndBlock *bembl, double aspectratio, double viscRatio, double capi
```

The constructor of the MatrixBlock class is called with the information from the BoundaryBlock that contains the geometrical data, the aspect ratio L/H, the viscosity ratio $\mu_{\text{dispersed}}$ over μ_{bulk} and a capillary number. A loop checks whether all declared boundaries have been set.

```
    bem = bembl;
    for (int n=0; n<bem->nbBlocks; n++) {
        cout<<"Check boundary "<<bem->Elements[n].nbPanRes<<" of "<<bem->Elements[n].nbPanel
        if (!(bem->Elements[n].nbPanels==bem->Elements[n].nbPanRes)){
            cout<<"Boundary "<<n<<" has "<<bem->Elements[n].nbPanels<<" instead of "<<bem->
            exit(0);
        }
    }
```

The initialization of the BoundaryBlock is finalized.

```
    bem->Init();
```

Variables beeing initilized. The aspect ratio L/H is changed into a permeability `kF`$= \sqrt{12}L/H$.

```
    widerange = 100.0;
    midrange = 50.0;

    parabc1 = 0;
    parabc2 = 0;
```

```
nSubSteps = 0;
kF = sqrt(12.0)*aspectratio;
Lambda = viscRatio;
Ca = capillaryNumber;
writeEnabled = 0;
nbTracers = 0;

timefactor = 0;
zerotime = 0;
prcndsing = 0;
arrest = 0;
channelheight = 1.0/aspectratio;
workdir = new char[120];
nOFm = bem->getAllSize();
```

Gauss points and integration weights are provided for 3 point and 6 point integration.

```
IGP = new double[6];
IGW = new double[6];
GP3 = new double[3];
GW3 = new double[3];

GP3[0] = -0.77459666924148337703;
GP3[1] = 0.0;
GP3[2] = -0.77459666924148337703;
GW3[0] = 0.55555555555555555555;
GW3[1] = 0.88888888888888888888;
GW3[2] = 0.55555555555555555555;

IGP[0] = -0.932469514203152;
IGP[1] = -0.661209386466265;
IGP[2] = -0.238619186083197;
IGP[3] = 0.238619186083197;
IGP[4] = 0.661209386466265;
IGP[5] = 0.932469514203152;
IGW[0] = 0.171324492379170;
IGW[1] = 0.360761573048139;
IGW[2] = 0.467913934572691;
IGW[3] = 0.467913934572691;
IGW[4] = 0.360761573048139;
IGW[5] = 0.171324492379170;
```

Memory for the matrix `A`, right hand side `rhs` and two vectors is reserved.

```
    A = (double*) calloc ((nOFm+10)*(nOFm+10),sizeof(double));
    rhs = (double*) calloc (nOFm+10,sizeof(double));
    u1vec = (double*) calloc (nOFm+10,sizeof(double));
    u2vec = (double*) calloc (nOFm+10,sizeof(double));
    cout<<"Matrix Block initiated with "<<nOFm<<" unknowns.\n"<<"Problem parameters, Ca:"<<
```

A number of pointers is initialized that provides access to some BoundaryBlock variables.

```
    dynblock = bem->dynblock;
    dynpanel = bem->dynpanels;
    ifblock = bem->ifblock;
    nbBlocks = bem->nbBlocks;
    fixsize = bem->getFixSize();
    thedropbc = 0;
    thestatbc = 0;
    applybc = &MatrixBlock::Standard;

    bndtype = bem->allBCtype;
    bndvalue1 = bem->allBCX;
    bndvalue2 = bem->allBCY;
```

If desired the number of cores can be limited here by uncommenting the `omp_set_num_threads`.

```
    #ifdef _OPENMP
            //omp_set_num_threads(2); //number_of_desired_cores
    #endif
}
```

Building the matrix

The following two routines build the matrix and right-hand-side using the geometry and boundary conditions.

```
void MatrixBlock::Build(double deltatime){
```

This routine builds the matrix, all the subsequent routines that make up for the different boundary conditions are stored in `matblc1.cpp`. First a routine checks that the matrix size did not change. Then matrix and right-hand-side are set to zero. Variables for the boundary conditions are allocated.

```
UpdateNOFM();
ResetMR(A, rhs);
double varA,varB, varC;
int varI;

for (int block = 0; block<bem->nbPanels; block++) {
```

The respective boundary condition index is stored in `varI`, if precondensation is used the index is advanced by 30.

```
    varI = bndtype[block] + 30*prcndsing;
    varA = bem->allBCX[block];
    varB = bem->allBCY[block];
    switch (varI) {
```

Fixed boundaries in the bulk fluid

```
        case 0:
            ZeroVel(block);
            break;
        case 1:
            NormVel(block, varA, varB, 1.0);
            break;
        case 2:
            Pressure(block, varA, 1.0);
            break;
        case 3:
            Poiseuille(block, varA, 1.0);
            break;
        case 4:
            varC = bem->allBCZ[block];
            FlatVel(block, varA, varB,varC, 1.0);
            break;
        case 5:
            InfiniBox(varA, varB);
```

```
            break;
        case 6:
            varC = bem->allBCZ[block];
            InfiniSource(varA, varB, varC, 1.0);
            break;
        case 7:
            InfiniLinearFlow(varA, varB, 1.0);
            break;
        case 8:
            FiniLinearFlow(block, varA, varB, 1.0);
            break;
        case 9:
            POX(block, varA, 1.0);
            break;
        case 10:
            SineFlow(block, varA, varB, 1.0);
            break;
```

Fixed boundaries in the dispersed fluid

```
        case 11:
            NormVel(block, varA, varB, Lambda);
            break;
        case 12:
            Pressure(block, varA, Lambda);
            break;
        case 13:
            Poiseuille(block, varA, Lambda);
            break;
        case 14:
            varC = bem->allBCZ[block];
            FlatVel(block, varA, varB,varC, Lambda);
        case 16:
            varC = bem->allBCZ[block];
            InfiniSource(varA, varB, varC, Lambda);
            break;
        case 19:
            POX(block, varA, Lambda);
            break;
        case 20:
            SineFlow(block, varA, varB, Lambda);
```

Liquid interface boundary conditions

```
case 21:
    CircTrac(block, deltatime);
    break;
case 22:
    BoosThess(block);
    break;
    /*    case 23:
     BordTrac(block, deltatime);
     break;  */
```

PreCondensation boundary conditions for fixed boundaries in the bulk fluid

```
case 30:
    SZeroVel(block);
    break;
case 31:
    SNormVel(block, varA, varB, 1.0);
    break;
case 32:
    SPressure(block, varA, 1.0);
    break;
case 33:
    SPoiseuille(block, varA, 1.0);
    break;
case 34:
    varC = bem->allBCZ[block];
    SFlatVel(block, varA, varB,varC, 1.0);
    break;
case 39:
    SPOX(block, varA, 1.0);
    break;
case 40:
    SSineFlow(block, varA, varB, 1.0);
```

PreCondensation boundary conditions for fixed boundaries in the dispersed fluid

```
case 41:
    NormVel(block, varA, varB, Lambda);
    break;
case 42:
```

```
                Pressure(block, varA, Lambda);
                break;
            case 43:
                SPoiseuille(block, varA, Lambda);
                break;
            case 44:
                varC = bem->allBCZ[block];
                FlatVel(block, varA, varB,varC, Lambda);
            case 46:
                varC = bem->allBCZ[block];
                InfiniSource(varA, varB, varC, Lambda);
                break;
            case 49:
                SPOX(block, varA, Lambda);
                break;
            case 50:
                SSineFlow(block, varA, varB, Lambda);
```

PreCondensation boundary conditions for liquid interfaces

```
            case 51:
                SCircTrac(block, deltatime);
                break;
                /*    case 53:
                 SBordTrac(block, deltatime);
                 break;  */
            default:
                cout<<"ERROR BC in build not found! Program stopped!\n";
                exit(0);
                break;
        }
    }
}
```

Building the right-hand-side for flow field visualisation

```
void MatrixBlock::BuildVisual(int points, double *Xpos, double *Ypos, double *U, double *V,
```

This routine is similar to `Build()` except that no matrix is created but all values that were formaly in the matrix become multiplied by the solution and accounted on the right-hand-side.

This allows to retrieve the velocity and pressure at the pole of the singularity. Near the liquid interface the singular behaviour is avoid by replacing the integrated value by the nearest velocity on the interface. The pressure near close to the interface is set zero because of ambiguity since the pressure is discontinuous there. A number of `points` and vector of `Xpos` and `Ypos` is passed. The vectors `U,V` and `P` give back the values. `Dom` indicates with an integer in domain the point is located, -1 for the bulk fluid and down for each droplet (i.e. -2 for being inside the first droplet). Positive values indicate that the velocity of a boundary node is to be used instead.

```
double varA,varB, varC;
int varI;
```

Allocate memory for the unknowns.

```
fill_n(U,points,0.0);
fill_n(V,points,0.0);
fill_n(P,points,0.0);

for (int block = 0; block<bem->nbPanels; block++) {
```

The respective boundary condition index is stored in `varI`, if precondensation is used the index is advanced by 30.

```
    varI = bndtype[block];
    varA = bem->allBCX[block];
    varB = bem->allBCY[block];
    switch (varI) {
```

Fixed boundaries in the bulk fluid

```
        case 0:
            VisuZeroVel(block, Xpos, Ypos, points, U, V, P);
            break;
        case 1:
            VisuNormVel(block, Xpos, Ypos, varA, varB, 1.0, points, U, V, P);
            break;
        case 2:
            VisuPressure(block,Xpos, Ypos, varA, 1.0, points, U, V, P);
```

```
            break;
        case 3:
            VisuPoiseuille(block,Xpos, Ypos, varA, 1.0, points, U, V, P);
            break;
        case 4:
            varC = bem->allBCZ[block];
            VisuFlatVel(block,Xpos, Ypos, varA, varB, varC, 1.0, points, U, V, P);
            break;
        case 5:
            VisuInfiniBox(Xpos, Ypos, varA, varB, points, U, V, P, Dom);
            break;
        case 6:
            VisuInfiniSource(Xpos, Ypos, varA, varB, varC, 1.0, points, U, V, P);
            break;
        case 7:
            VisuLinearFlow(Xpos, Ypos, varA, varB, points, U, V, P, Dom);
            break;
        case 8:
            VisuFiniFlow(block, Xpos, Ypos, varA, varB, 1.0, points, U, V, P);
            break;
        case 9:
            VisuPOX(block,Xpos, Ypos, varA, 1.0, points, U, V, P);
            break;
        case 10:
            VisuSineFlow(block, Xpos, Ypos, varA, varB, 1.0, points, U, V, P);
            break;
```

Fixed boundaries in the dispersed fluid

```
        case 11:
            VisuNormVel(block, Xpos, Ypos, varA, varB, Lambda, points, U, V, P);
            break;
        case 12:
            VisuPressure(block,Xpos, Ypos, varA, Lambda, points, U, V, P);
            break;
        case 13:
            VisuPoiseuille(block,Xpos, Ypos, varA, Lambda, points, U, V, P);
            break;
        case 14:
            varC = bem->allBCZ[block];
            VisuFlatVel(block,Xpos, Ypos, varA, varB, varC, Lambda, points, U, V, P);
            break;
        case 16:
            VisuInfiniSource(Xpos, Ypos, varA, varB, varC, Lambda, points, U, V, P);
        case 19:
```

```
                VisuPOX(block, Xpos, Ypos, varA, Lambda, points, U, V, P);
                break;
```

Case 20 SineFlow is missing! Liquid interface boundary conditions

```
            case 21:
                VisuCircTrac(block,Xpos, Ypos, points, U, V, P);
                break;
                /*      case 23:
                 VisuBordTrac(block,Xpos, Ypos, points, U, V, P);
                 break;  */
            default:
                cout<<"ERROR BC in BuildVisual not found!\n";
                break;
        }
    }

    for (int k=0; k<points; k++) {
```

Depending whether the point is inside a droplet or not the viscosity ratio λ needs to be added. Divison by 2π or 4π is due to the domain integral value of the singularity.

```
        P[k] = P[k]/(2*M_PI);
        if (Dom[k]<-1){
            U[k] = U[k]/(4*M_PI*Lambda);
            V[k] = V[k]/(4*M_PI*Lambda);
```

The constant pressure jump is added. We avoid to integrate this contribution along the interface because small errors in the curvature become amplified by this often large value.

```
            P[k] += 2.0/channelheight;
        } else if(Dom[k]==-1){
            U[k] = U[k]/(4*M_PI);
            V[k] = V[k]/(4*M_PI);
        } else {
```

If the point is close to the interface the integrated value is discarded and the value of the closest point on the interface is used.

```
            U[k] = rhs[2*Dom[k]];
            V[k] = rhs[2*Dom[k]+1];
            P[k] = 0;
        }
    }
}
```

Interfacial boundary conditions

Setting the static boundary condition

```
void MatrixBlock::ChoseStatBC(int choice, double para1, double para2){
```

This procedure sets a boundary condition, which modifies the liquid interface boundary condition due to geometrical models. Those include for instance indentations in the top or bottom channel confinement, which change the out-of-plane curvature of a droplet. The models are described in the manual.

```
    thestatbc = choice;
    statbc1 = para1;
    statbc2 = para2;
    switch (choice) {
```

Curvature dependence due to thin film formation. Still in debugging!

```
        case 1:
            applybc = &MatrixBlock::Meiburg;
            cout<<"Stat BC is Meiburg\n";
            break;
```

A groove in the channel walls that functions as a microfluidic rail.

```
        case 2:
            applybc = &MatrixBlock::Rail;
            cout<<"Stat BC is Rail\n";
            break;
```

An indentation in the channel top or bottom walls.

```
        case 3:
            applybc = &MatrixBlock::Hole;
            cout<<"Stat BC is Hole\n";
            break;
```

Array of holes.

```
        case 4:
            applybc = &MatrixBlock::HoleArray;
            cout<<"Stat BC is HoleArray\n";
            break;
```

A slope of the channel height.

```
        case 5:
            applybc = &MatrixBlock::Wedge;
            cout<<"Stat BC is Wedge\n";
            break;
```

A step-wise change in the channel height.

```
        case 6:
            applybc = &MatrixBlock::Marche;
            cout<<"Stat BC is Step\n";
            break;
```

Assuming the liquid menisc is not curved but flat. Visualization may be wrong. This serves to separate Marangoni effects into contributions from tangential motion and contribution from pressure jump out-of-plane curvature.

```
        case 7:
            applybc = &MatrixBlock::Flatborder;
            cout<<"Stat BC is No Meniscus\n";
            break;
```

Exceptionally not an influence of the interface shape but a centrifugal force on the droplet that is transformed into a interface force. Visualization will not get the centrifugal pressure contribution.

```
        case 8:
            applybc = &MatrixBlock::Rotate;
            cout<<"Stat BC is Rotate\n";
            break;
```

Exceptionally not an influence of the interface shape but a gravitational force on the droplet that is transformed into a interface force. Visualization will not get the centrifugal pressure contribution.

```
        case 9:
            applybc = &MatrixBlock::Gravity;
            cout<<"Stat BC is Gravity\n";
            break;
```

A custom boundary condition that can be edited by the user.

```
        case 10:
            applybc = &MatrixBlock::CustomStatBC;
            cout<<"Stat BC is Custom\n";
```

A simple menisc of curvature $\pi/4 \cdot 2/h$.

```
        default:
            applybc = &MatrixBlock::Standard;
            cout<<"Stat BC is Standard\n";
            break;
    }
}
```

Setting dynamic boundary conditions

```
void MatrixBlock::ChoseDynBC(int choice, double para1, double para2){
```

This procedure sets the droplet interfacial tension to a possibly position dependent value. If the surface tension is modelled as temperature dependent, then this is the low Peclet number regime, where a heat source is prescribed and heat diffusion is much faster then heat convection. The actual influence on the surface tension can be seen in function `LocalGam` or the manual.

```
    thedropbc = choice;
    parabc1 = para1;
    parabc2 = para2;

    switch (choice) {
```

A Gaussian of surface tension.

```
        case 1:
            cout<<"Dyn BC is Focussed\n";
            break;
```

A linear surface tension gradient.

```
        case 2:
            cout<<"Dyn BC is Gradient\n";
            break;
```

A linear hat function that models the presenced of a haeting wire.

```
        case 3:
            cout<<"Dyn BC is Wire\n";
            break;
```

A custom boundary condition that can be edited by the user.

```
        case 10:
            cout<<"Dyn BC is Custom\n";
```

A constant surface tension.

```
        default:
            cout<<"Dyn BC is Constant\n";
            break;
    }
}
```

Static boundary conditions

```
void MatrixBlock::Flatborder(double Xp, double Yp, double Nx, double Ny, double Unorm, doubl
    f1 = 0.0;
    f2 = 0.0;
}

void MatrixBlock::Standard(double Xp, double Yp, double Nx, double Ny, double Unorm, double
```

The constant curvature at constant reference surface tension is subtracted, because its net influence is zero and it may amplify errors by `1/channelheight`, which is large in the limit of flat channels. Deviations from the reference surface tension are included by `$\gamma$-1`.

```
    f1 =  -2.0*(gamma-1.0)/channelheight*Nx;
    f2 =  -2.0*(gamma-1.0)/channelheight*Ny;
}

void MatrixBlock::Hole(double Xp, double Yp, double Nx, double Ny, double Unorm, double gamm
```

Effect of a hole on the curvature. The constant curvature and surface tension are subtracted. The resulting effective curvature $\kappa_{eff} = \frac{2}{h}(-\ln(2)\frac{r_h^2}{h^2}e^{-(r^2/r_h^2)^4} + (\gamma-1)/(1+\frac{r_h^2}{h^2}e^{-\ln(2)(r^2/r_h^2)^4})$. The parameter `statbc1` prescribes the radius of the hole.

```
    double rhP2 = statbc1*statbc1;
    double kappaeff;
    double rhE2 = rhP2/(channelheight*channelheight);
    double r2 = Xp*Xp + Yp*Yp;
    kappaeff = 2.0/channelheight*(-rhE2*exp(-pow(r2/rhP2,4)*0.69315)+(gamma-1.0))/(1.0+rhE2*
```

The effective force is projected on the normal vector and `f1` will act in the x direction and `f2` in the y direction.

```
    f1 = -kappaeff*Nx;
    f2 = -kappaeff*Ny;
}


void MatrixBlock::HoleArray(double Xp, double Yp, double Nx, double Ny, double Unorm, double
```

An array of holes. Number `m` in x times number `n` in y. The parameter `statbc2` that is set in the routine `ChoseStatBC` defines the spacing in between the holes. The parameter `statbc1` prescribes the radius of the hole.

```
    int m =2, n=2;

    double rhP2 = statbc1*statbc1;
    double kappaeff;
    double rhE2 = rhP2/(channelheight*channelheight);
```

Finding the closest hole and determine the square of the distance `r2`.

```
    double r2 = 1000.0;
    double Xp0, Yp0;
    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++) {
            Xp0 = Xp - (i-(m-1)*0.5)*statbc2;
            Yp0 = Yp - (j-(n-1)*0.5)*statbc2;
            r2= min(Xp0*Xp0 + Yp0*Yp0,r2);
        }
    }
```

The rest is analog to the routine `Hole`. Whenever another configuration of holes is desired one should create a Custom boundary condition, copy paste this routine and modify it.

```
    kappaeff = 2.0/channelheight*(-rhE2*exp(-pow(r2/rhP2,4)*0.69315)+(gamma-1.0))/(1.0+rhE2*
    f1 = kappaeff*Nx;
    f2 = kappaeff*Ny;
}

void MatrixBlock::Rail(double Xp, double Yp, double Nx, double Ny, double Unorm, double gamm
```

Effect of a rail

```
    double nG;
    double rhE2 = statbc1*statbc1;
    double kappaeff = 2.0*rhE2/(channelheight*(rhE2+1.0))*(rhE2<=1) + (-1.0/(channelheight*s
    rhE2 = rhE2*channelheight*channelheight;
    double r2 = pow(Yp-2.0*sin(Xp/2.0),2);

    kappaeff = -2.0/channelheight/(1.0+rhE2*exp(-pow(r2/rhE2,4)*0.69315) );
    nG  = kappaeff;

    f1 = nG*Nx;
    f2 = nG*Ny;
}

void MatrixBlock::Meiburg(double Xp, double Yp, double Nx, double Ny, double Unorm, double i

   //  Obsolete, ForceFree in matblc1.cpp takes this functionality
```

THIS is strictly explicit and no good after remeshing

```
    double nG;
 //   nG  = fifac*pow(fabs(Unorm),0.666666)*(3.8*(Unorm>=0.0)-1.13*(Unorm<0.0));
   // nG = fifac*pow(fabs(Unorm),0.666666)*(2.465+1.335*tanh(40*Unorm))*(1-2*(Unorm<0));
   // nG = fifac*pow(fabs(Unorm),0.666666)*(3.8*(Unorm>=0.0)-1.13*(Unorm<0.0))*tanh(20*fabs
    nG = -kF/sqrt(3)*pow(fabs(Unorm),0.66666666)*(-1.13+2.465*(tanh(20*Unorm)+1.0));
//  f1 = -3*nG*Nx;
//  f2 = -3*nG*Ny;
    f1 = nG*Nx;
    f2 = nG*Ny;
}


void MatrixBlock::Wedge(double Xp, double Yp, double Nx, double Ny, double Unorm, double gar
```

Wedged channel , where the height changes gradually

```
    // double tmp = 2.0/(statbc1*Xp+statbc2*Xp+channelheight);
    double tmp = -2.0*(statbc1*Xp+statbc2*Yp)/((statbc1*Xp+statbc2*Yp+1.0)*channelheight);
    f1 = (gamma*(1.0-M_PI_4)-tmp)*Nx;
    f2 = (gamma*(1.0-M_PI_4)-tmp)*Ny;
}

void MatrixBlock::Marche(double Xp, double Yp, double Nx, double Ny, double Unorm, double ga
```

Steep step in a channel

```
    double tmp;
    Yp = Yp-1.0;

    if(Yp<-1.0*channelheight*statbc2){
        tmp = 0.0;
    } else if (Yp>3.0*channelheight*statbc2) {
        tmp = - 2.0*statbc1/(channelheight*(1.0+statbc1));
    } else {
        tmp = statbc1*(1.0+tanh(4.0*Yp/(channelheight*statbc2)-2.0));
        tmp = - 2.0*tmp/(channelheight*(2.0 + tmp));
    }
```

```
    f1 = -tmp*Nx;
    f2 = -tmp*Ny;
}

void MatrixBlock::Rotate(double Xp, double Yp, double Nx, double Ny, double Unorm, double ga
    // boundary condition for centrifugal forces para1 = 0.5*We/Rb, para2 = y0 center of ro
//  f1 = 0.5*statbc1*(Xp*Xp+(Yp-statbc2)*(Yp-statbc2))*Nx;
//  f2 = 0.5*statbc1*(Xp*Xp+(Yp-statbc2)*(Yp-statbc2))*Ny;

    f1 = 0.5*statbc1*(Yp*Yp+(Xp-statbc2)*(Xp-statbc2))*Nx;
    f2 = 0.5*statbc1*(Yp*Yp+(Xp-statbc2)*(Xp-statbc2))*Ny;
}

void MatrixBlock::Gravity(double Xp, double Yp, double Nx, double Ny, double Unorm, double g
    // boundary condition for gravitational force, statbc indicate direction and |statbc| =
    f1 = (statbc1*Xp+statbc2*Yp)*Nx;
    f2 = (statbc1*Xp+statbc2*Yp)*Ny;
}

void MatrixBlock::CustomStatBC(double Xp, double Yp, double Nx, double Ny, double Unorm, dou
    // Custom function. Nothing is defined here. We prescribe a surface tension of 1 just as
    f1 =  -2.0*(gamma-1.0)/channelheight*Nx;
    f2 =  -2.0*(gamma-1.0)/channelheight*Ny;
}
```

Dynamic boundary conditions/Variation of surface tension.

```
double MatrixBlock::LocalGam(int dropId, int nL, int nR){
```

Assignes values to the surface tension in the domain

```
    double lcGamma, varA, varB, varC;
    double Xp = 0;
    double Yp = 0;


    switch (thedropbc) {
        case 1:
```

Focussed Marangoni at (0, Yl)

```
            Xp = 0.5*(bem->allXId[dropId][nL]+bem->allXId[dropId][nR])-0.0;
            Yp = 0.5*(bem->allYId[dropId][nL]+bem->allYId[dropId][nR])-0.0;
            varB = 0.69315/(parabc2*parabc2);
            varC = exp(-varB*(Xp*Xp+Yp*Yp));
            lcGamma  = 1.0+parabc1*varC;
            break;
        case 2:
```

Gradient.

```
            Xp = 0.5*(bem->allXId[dropId][nL]+bem->allXId[dropId][nR]);
            Yp = 0.5*(bem->allYId[dropId][nL]+bem->allYId[dropId][nR]);
            lcGamma = 1.0+Xp*parabc1+Yp*parabc2;
            break;
        case 3:
```

Wire.

```
            Yp = 0.5*(bem->allYId[dropId][nL]+bem->allYId[dropId][nR]);
            varA = (Yp>(-parabc2))*(parabc2+Yp)+(Yp>0.0)*(-2.0*Yp)+(Yp>parabc2)*(-parabc2+Yp
            lcGamma = 1.0+parabc1*varA;
            break;
        case 10:
```

Custom function. Nothing is defined here. We prescribe a surface tension of 1 just as the default. This is free to be edited by the user.

```
            lcGamma = 1.0;
            break;
```

Constant surface tension.

```
        default:
            lcGamma = 1.0;
            break;
    }
    return lcGamma;

}
```

Forces due to VdW or droplet Rayleigh-Plateau instability.

```
double MatrixBlock::SpecialForces(int DropId, int Vertex){
```

Sums all contributions from pinching to VanDerWaals and other repulsion forces No Marangoni effect included

```
    double WallHamacker = 0;
    double DropHamacker = 0; // positive means repulsion
    double ForceBreakUp;
    double ForceVanDerWalls;
    double ForceVanDerFusion;
```

The force due to a droplet menisc whose curvature is higher than the half-channel height. This would be a consequence of the formation of a droplet ligament, whose diameter is smaller than the channel height. NOT YET INCLUDED.

```
    ForceBreakUp = -bem->allCdist[DropId][Vertex];
```

This very simplistic inclusion of Van-der-Waals forces considers the decay rate with a 3rd power, which is appropriate for repulsion between two parallel plates.

```
    ForceVanDerWalls = -WallHamacker*pow(bem->allWdist[DropId][Vertex],-3);
    ForceVanDerFusion = -DropHamacker*pow(bem->allRdist[DropId][Vertex],-3);

    return ForceBreakUp+ForceVanDerFusion+ForceVanDerWalls;
}
```

Problem solving

Precondensation/Guass Block-Diagonalization

If the decompose the matrix into 4 blocks, of which one is the influence of the wall on the boundary elements on the wall, you can can solve before hand the wall problem. The function `PreCondense` calculates the inverse of this wall problem, which allows by Gauss Block elimination by matrix/matrix matrix/vector multiplications to reduce the final problem. The remaining matrix that has to be solved contains only the influence of the droplet on the boundary elements on the droplet.

```
void MatrixBlock::PreCondense(){
    if (fixsize>0) {
```

The precondensation is only done if there is an outer boundary. Change the matrix size as to build only the matrix for fixed walls.

```
        int bufferPan = bem->nbPanels;
        int bufferNM = nOFm;
        int bufferblocks = nbBlocks;
        prcndsing = 0;
        bem->nbPanels = bem->dynpanels;
        nOFm = 2*fixsize;
        nbBlocks = dynblock;
        bem->nbBlocks =  dynblock;
```

Build matrix and invert it.

```
        Build(0.0);
        memcpy(u2vec, rhs, 2*fixsize*sizeof(double));
        dgeinv(A, 2*fixsize);
```

Undo prior changes such that the mobile interfaces are included again.

```
        prcndsing = 1;
        bem->nbPanels = bufferPan;
        nOFm = bufferNM;
        nbBlocks = bufferblocks;
```

```
        bem->nbBlocks  = bufferblocks;
        cout<<"Matrix PreCondensed\n";
    } else {
        cout<<"No fixed boundaries - Matrix NOT PreCondensed\n";
    }
}
```

Solve a matrix.

```
void MatrixBlock::Solve(){
```

This solves the matrix, either in a precondensation way if precondensed or in the usual way by calling the respective modified LAPACK routine. The matrix `A`, right hand side `rhs` and matrix size/submatrix sizes are passed.

```
    if (prcndsing) {
        dgesch(A, rhs, fixsize*2, nOFm-2*fixsize);
    } else {
        dgesv(A, rhs, nOFm);
    }
}

    // int MatrixBlock::RunRigidRK1(double dgap, double *move){
    // int MatrixBlock::RunRigidRK2(double dgap, double *move){
    // int MatrixBlock::RunRigidManual(double dgap, double *move){
```

A single step Euler explicit.

```
int MatrixBlock::RunOne(){
```

Do one iteration (matrix create and solve), evolve the interface and update the timestep.

```
    Build(deltaT0);
    Solve();
```

```
    bem->allEvolve(1.0, 0.0, deltaT0, rhs);
    bem->allStock();
    bem->runstep++;
    bem->runtime += deltaT0;
    return 0;
}
```

One step Euler explicit/Runge Kutta scheme.

```
int MatrixBlock::SolveRK1(){
```

Initialize variables for time adaptation.

```
    int status = 0;
    double stretch =1.0;
    double deltaT = deltaT0;
    cout<<"RK1";
```

Perform `nSubSteps` iterations. At first `SeedDrop` remeshes the droplet if necessary.

```
    for (int n=0; n<nSubSteps; ++n) {
        bem->SeedDrop();
        dominantForce = 0;
```

Create and solve the problem.

```
        Build(deltaT);
        Solve();
```

Determine if the time step should be reduced with `DetTime`, then evolve the interface.

```
        stretch = DetTime();
        deltaT = deltaT0*stretch;
        AdvanceTimestep(deltaT, 1);

        cout<<"."<<flush;
    }
    bem->runstep++;
    return status;
}

    // int MatrixBlock::SolveFilm(){
```

2nd order Runge Kutta scheme.

```
int MatrixBlock::SolveRK2(){
```

Initialize variables for time adaptation.

```
    int status = 0;
    double stretch = 1.0;
    double deltaT = deltaT0;
    cout<<"RK2";
```

Perform `nSubSteps` iterations. At first `SeedDrop` remeshes the droplet if necessary.

```
    for (int n=0; n<nSubSteps; ++n) {
        bem->SeedDrop();
```

Step 1: Create and solve the problem, then move the interface.

```
        Build(deltaT);
        Solve();
        bem->allEvolve(1.0, 0.0, deltaT, &rhs[2*fixsize]);
```

Step 2: Create and solve the problem, then advance interface and time.

```
        Build(deltaT);
        Solve();
        stretch = DetTime();
        deltaT = deltaT0*stretch;
        AdvanceTimestep(deltaT, 2);

        cout<<"."<<flush;
    }
    bem->runstep++;
    return status;
}
```

Routines dedicated to time marching

```
void MatrixBlock::setTimeStep(double dT, int nSteps){
    deltaT0 = dT;
    nSubSteps = nSteps;
    cout<<"Time stepping set, dt:"<<dT<<" with "<<nSteps<<" Substeps\n";
}

void MatrixBlock::Asifstep(int blockId, double& Xcm, double& Ycm){
    // pretend a move get center of mass and get back
    Build(deltaT0);
    Solve();
    bem->Evolve(blockId, 0.0, 1.0, deltaT0, &rhs[2*fixsize]);
    bem->getCoM(blockId, Xcm, Ycm);
    bem->Evolve(blockId, 1.0, 0.0, 0.0, &rhs[2*fixsize]);
}

double MatrixBlock::DetTime(){
    // determine a aproproate time step in case of strong forcing
    double timestretch = 1;
    if (timefactor>0) {
        timestretch = max(floor(deltaT0*dominantForce/timefactor),1.0);
    }
    return 1.0/timestretch;
}

void MatrixBlock::AdvanceTimestep(double deltaT, int modus){
```

Move tracers by solution

```
double schmidt = 100.0;  // diff length = sqrt(Sc*dt) , we multiply schmidt by two beca
double scatter;
double *uvel, *vvel, *dummy, *oldTrX, *oldTrY;
int *dommy, dom, coupe;

if (nbTracers>0) {
    oldTrX = (double*) calloc (nbTracers,sizeof(double));
    oldTrY = (double*) calloc (nbTracers,sizeof(double));
    memcpy(oldTrX, xTracers, nbTracers*sizeof(double));
    memcpy(oldTrY, yTracers, nbTracers*sizeof(double));
    uvel = (double*) calloc (nbTracers,sizeof(double));
    vvel = (double*) calloc (nbTracers,sizeof(double));
    dummy = (double*) calloc (nbTracers,sizeof(double));
    dommy = (int*) calloc (nbTracers,sizeof(int));
    schmidt = sqrt(deltaT/schmidt);

    fill_n(dommy, nbTracers, -1);

    BuildVisual(nbTracers, xTracers, yTracers, uvel, vvel, dummy, dommy);
    for (int n=0; n<nbTracers; n++) {
        scatter = rand();
        scatter = scatter*2.0*M_PI/RAND_MAX;
        xTracers[n] += deltaT*uvel[n] + schmidt*cos(scatter);
        yTracers[n] += deltaT*vvel[n] + schmidt*sin(scatter);
    }
}

if (modus==1){
    // One step advance
    bem->allEvolve(1.0, 0.0, deltaT, &rhs[2*fixsize]);
} else if (modus==2){
    // Two step 2/2 advance
    bem->allEvolve(0.5, 0.5, deltaT*0.5, &rhs[2*fixsize]);
}

bem->runtime += deltaT;
bem->allStock();

bem->correctArea();
bem->allStock();

if (nbTracers>0) {
    for (int n=0; n<nbTracers; n++) {
        dom = FigInOut(xTracers[n], yTracers[n], 0.0);
        coupe = 0;
        while (dom!=-1) {
            scatter = rand();
```

```
                scatter = scatter*10*M_PI/RAND_MAX;
                xTracers[n] = oldTrX[n] + deltaT*uvel[n] + schmidt*cos(scatter);
                yTracers[n] = oldTrY[n] + deltaT*vvel[n] + schmidt*sin(scatter);
                dom = FigInOut(xTracers[n], yTracers[n], 0.0);

                if(coupe>3){
                    cout<<" Outsider remains after 50 tries! ";
                    break;
                }
                coupe++;
            }
        }
        delete uvel;
        delete vvel;
        delete dummy;
        delete dommy;
    }
}

void MatrixBlock::DistributeTracers(int thenumTr, double Xd, double Yd, double Dd){
```

Distribute tracers

```
    if(thenumTr>0){
        srand (time(NULL));

        int circles, circum;
        nbTracers = 0;
        xTracers = (double*) calloc (thenumTr,sizeof(double));
        yTracers = (double*) calloc (thenumTr,sizeof(double));
        double alpha = 0.0;
        double raddi = Dd;

        nbTracers = 1;
        xTracers[0] = Xd;
        yTracers[0] = Yd;
        circles = ceil( -0.5+sqrt(0.25 + thenumTr*1.0/M_PI ));

        for (int n=1; n<circles; n++) {
            circum = floor(2*M_PI*n);
            for (int p=0; p<circum; p++) {
                raddi = n*Dd/circles;
                alpha = 2*p*M_PI/circum;
                xTracers[nbTracers] = Xd+raddi*cos(alpha);
                yTracers[nbTracers] = Yd+raddi*sin(alpha);
```

```
                nbTracers ++;
            }
        }
    }
}
```

Writing data

```
void MatrixBlock::EnableWrite(char* save2dir){
```

Set path for writing and reading data

```
    workdir = save2dir;
    writeEnabled = 1;
    bem->EnableWrite(save2dir);
}

void MatrixBlock::SubFolder(char *subdir){
```

Creates a subfolder and checks if it does not already exist

```
    char logurl[120];
    sprintf(logurl, "%s/%s%c",workdir.c_str(), subdir, '\0');
    cout<<"Writing enabled to: "<<workdir.c_str()<<"\n";

#ifdef _MSC_VER
    if (_mkdir(logurl) == -1) {
        cout << "\nSubfolder already exists \nAborting\n";
        DisableWrite();
        exit(0);
    }
#else
    if (mkdir(logurl, 0777) == -1) {
        cout << "\nSubfolder already exists \nAborting\n";
        DisableWrite();
        exit(0);
    }
#endif
```

```
    EnableWrite(logurl);
}

void MatrixBlock::SensorEnable(int sid){
```

Clear Sensor File

```
    char strL[128];
    string filedir;
    sprintf(strL,"%s/sensor%d.txt",workdir.c_str(),sid);
    FILE* sensorfile;
    sensorfile = fopen(strL,"w");
    fclose(sensorfile);
}


void MatrixBlock::LogPrepare(int logid, char *disclaimer){
```

Create a file for log and if exists delete its content, for appending don't call this procedure

```
    char* logurl;
    logurl = new char[120];
    sprintf(logurl,"%s/log%i.txt", workdir.c_str(),logid);
    logfile = fopen(logurl, "w");
    assert(logfile!=NULL);
    fprintf(logfile,"%s", disclaimer);
    fclose(logfile);
}

void MatrixBlock::LogRuntime(int logid){
    timeval jetzt;
#ifdef _WIN32
//  GetSystemTimeAsFileTime(&jetzt);
#else
    gettimeofday(&jetzt, 0);
#endif
    char logurl[120];
    double runtime =  clock()*1.0/CLOCKS_PER_SEC;
    sprintf(logurl,"%s/log%i.txt", workdir.c_str(),logid);
    logfile = fopen(logurl, "a");
    fprintf(logfile, "%f\n", runtime);
```

```
    fclose(logfile);
}

void MatrixBlock::LogCfg(){
```

Logs: nSubsteps, deltaT, BC1value, BC2value, nbBlocks, R/H, lambda, Ca, NULL

```
    char logurl[120];
    sprintf(logurl,"%s/ulam_cfg.txt", workdir.c_str());
    logfile = fopen(logurl, "w");
    fprintf(logfile,"%d %f %f %f %f %d %d %d", nSubSteps, deltaT0, kF/sqrt(12.0), Lambda, Ca
    fclose(logfile);
}

void MatrixBlock::LogCustom(int logid, int modusid){
    char logurl[120];
    sprintf(logurl,"%s/log%i.txt", workdir.c_str(),logid);

    switch(modusid){
        case 0:
            break;
```

Case 1: Extract Pressure Data along a line

```
        case 1:
        {
            double *Xp, *Yp, *Prs, *Uv, *Vv, theta;
            int *dom_assign;
            int the_samples = 300;

            dom_assign = new int[the_samples];
            Xp = new double[the_samples];
            Yp = new double[the_samples];
            Uv = new double[the_samples];
            Vv = new double[the_samples];
            Prs = new double[the_samples];

            for (int j =0; j<the_samples; j++) {
                if (logid==1) {
                    //Circle
                    theta = j*M_PI/(the_samples-1.0);
```

```
                Xp[j] = -1.4884*cos(theta)+1.2960;
                Yp[j] =  1.4884*sin(theta);
            } else {
                //Line
                theta = j*4.0/(the_samples-1)-0.5;
                Xp[j] = theta;
                Yp[j] = 0.0;
            }
            dom_assign[j] = FigInOut(Xp[j], Yp[j], 0.0);
        }

        BuildVisual(the_samples, Xp, Yp, Uv, Vv, Prs, dom_assign);

        logfile = fopen(logurl, "a");
        for (int j=0; j<the_samples; j++) {
            fprintf(logfile, "%f %f %f %f %f %d\n", Xp[j], Yp[j], Uv[j], Vv[j], Prs[j],
        }
        fclose(logfile);
        break;
    }
```

Case 2: output of position and angle for fibers with 800 nodes

```
    case 2:
    {
        double xcm, ycm;
        double angle = (bem->Elements[1].YId[0][320]-bem->Elements[1].YId[0][80])/sqrt(p
        // fiber with 500 nodes
        //  double angle = (bem->Elements[1].YId[0][240]-bem->Elements[1].YId[0][50])/s

        logfile = fopen(logurl, "a");
        bem->getCoM(1,xcm, ycm);
        fprintf(logfile, "%f %f %f %f\n", xcm, ycm, angle, bem->runtime);

        fclose(logfile);
        break;
    }
```

Case 3: Output of boundary data

```
    case 3:
    {
```

```
            int nbpoints = 10;

            double *ut = new double[nbpoints];
            double *vt = new double[nbpoints];
            double *pt = new double[nbpoints];
            double *xt = new double[nbpoints];
            double *yt = new double[nbpoints];
            int *subdom = new int[nbpoints];

            std::fill_n(subdom, nbpoints, -1);

            for(int j=0; j<nbpoints; j++){
                xt[j] = 0.5*(bem->Elements[1].XId[0][j]+bem->Elements[1].XId[0][j+1]);
                yt[j] = 0.5*(bem->Elements[1].YId[0][j]+bem->Elements[1].YId[0][j+1]);
            }

            BuildVisual(10, xt, yt, ut, vt, pt, subdom);

            double forza[3];
            getForce(1, 1, forza , rhs);
            logfile = fopen(logurl, "w");
            for (int j=0; j<nbpoints; j++) {
                fprintf(logfile, "%f %f %f %f %f\n", xt[j], yt[j], ut[j], vt[j], pt[j]);
            }
            fclose(logfile);
            break;
        }
    }
}

void MatrixBlock::Result(int fid){
```

Write the result to file

```
    char cname[128];
    cout<< "save results ";
    FILE* rrfile;

    sprintf(cname, "%s/results%i.dat",workdir.c_str(),fid);
    rrfile = fopen(cname,"w");

    //  fprintf(rrfile, "%i \n", (nOFm-3)/2);
    for(int n=0; n<(nOFm/2); n++){
        fprintf(rrfile, "%3.15f %3.15f\n", rhs[2*n], rhs[2*n+1]);
    }
```

```
    fclose(rrfile);
}

void MatrixBlock::BoosThessPrintAnalytic(){
```

Calculate and write to screen analytic solution of Boos and Thess

```
    double K0, K1, I1, I2;

    bessk01(kF, K0, K1);
    bessi12(kF, I1, I2);
    double r_boosthess = I2*K0/(I2*(kF*K1+2*K0)+Lambda*K0*(kF*I1-2*I2));

    double errormax = 0;
    for (int i=0; i<bem->Elements[0].PanelSize[0]; i++) {
        errormax = max(errormax, fabs(r_boosthess*0.5*(bem->Elements[0].XId[0][i]+bem->Eleme
    }

    cout<<"Boos Thess Analytic Value = "<<setprecision (15)<<r_boosthess<<"\n";
    cout<<"Numerical Value = "<<setprecision (15)<<rhs[0]<<"\n";
    cout<<"Relative Error = "<<errormax/r_boosthess<<"\n";
}

void MatrixBlock::WriteTracers(int nbSlice){
```

Write position of tracers

```
    char strL[128];
    sprintf(strL,"%s/tracers%d.dat",workdir.c_str(), nbSlice);
    FILE* pfile;
    pfile = fopen(strL,"w");
    for (int n=0; n<nbTracers; n++) {
        fprintf(pfile, "%1.6f %1.6f\n", xTracers[n], yTracers[n]);
    }
    fclose(pfile);
}

void MatrixBlock::SensorWrite(int sid, double Xp, double Yp){
```

Writes the values of the sensor

```
    char strL[128];
    double U,V, P;
    int zone;
    string filedir;

    zone = FigInOut(Xp, Yp,0.0);

    BuildVisual(1, &Xp, &Yp, &U, &V, &P, &zone);
    sprintf(strL,"%s/sensor%d.txt",workdir.c_str(),sid);
    FILE* sensorfile;
    sensorfile = fopen(strL,"a");
    fprintf(sensorfile,"%5.9f %5.9f %5.9f %5.9f\n", bem->runtime, U, V, P);
    fclose(sensorfile);
}


void MatrixBlock::writeRHS(){
```

Write the result to file

```
    char cname[128];
    cout<< "save results ";
    FILE* rrfile;

    sprintf(cname, "%s/rhsbuffer.dat",workdir.c_str());
    rrfile = fopen(cname,"w");

    fprintf(rrfile, "%i \n", nOFm/2);
    for(int n=0; n<nOFm/2; n++){
        fprintf(rrfile, "%3.9f\n", rhs[2*n]);
    }
    for(int n=0; n<nOFm/2; n++){
        fprintf(rrfile, "%3.9f\n",rhs[2*n+1]);
    }

    fclose(rrfile);
}

void MatrixBlock::PV(int atp){
```

Write one solution value to screen

```
    cout<<setprecision (15)<<u1vec[atp]<<"\n";
}

void MatrixBlock::saveMatrix(){
```

Write the matrix to file

```
    char cname[128];
    cout<< "save matrix ";
    FILE* rrfile;
    sprintf(cname, "%s/matrixbuffer.dat",workdir.c_str());
    rrfile = fopen(cname,"w");
    int M,N;


    if(prcndsing==0){
        for(int n=0; n<nOFm; n++){
            for (int m=0; m<nOFm; m++) {
                fprintf(rrfile, "%3.13f ", A[m*nOFm+n]);
            }
            fprintf(rrfile,"\n");
        }

        for(int n=0; n<nOFm; n++){
            fprintf(rrfile, "%3.13f ", rhs[n]);
        }
    } else {
        M = 2*fixsize;
        N = nOFm-2*fixsize;

        for(int n=0; n<M; n++){
            for (int m=0; m<M; m++) {
                fprintf(rrfile, "%3.13f ", A[m*M+n]);
            }
            for (int m=0; m<N; m++) {
                fprintf(rrfile, "%3.13f ", A[M*(N+M)+m*M+n]);
            }
            fprintf(rrfile,"\n");
        }
        for(int n=0; n<N; n++){
            for (int m=0; m<M; m++) {
                fprintf(rrfile, "%3.13f ", A[M*M+m*N+n]);
            }
```

```
            for (int m=0; m<N; m++) {
                fprintf(rrfile, "%3.13f ", A[M*(2*N+M)+m*N+n]);
            }
            fprintf(rrfile,"\n");
        }
        for(int n=0; n<nOFm; n++){
            fprintf(rrfile, "%3.13f ", rhs[n]);
        }

    }

    // unformated
    /*
     for(int n=0; n<nOFm*nOFm; n++){
     fprintf(rrfile, "%3.12f\n", A[n]);
     }

     for(int n=0; n<nOFm; n++){
     fprintf(rrfile, "%3.12f\n",rhs[n]);
     }
     */
    fclose(rrfile);
}

void MatrixBlock::DisableWrite(){
    if (writeEnabled) {
        writeEnabled = 0;
        bem->DisableWrite();
    }
}
```

Field data export

```
void MatrixBlock::VTKexport(int nbSlice, int resolution, double left, double right, double b
```

Writes Field output to TecPlot Binary file

```
    if(prcndsing){
        cout<<"PreCondense and Output will not work yet"<<flush;
    }
    double xls, yls;
```

```
int ifid = bem->ifid;

double xFramesize = right-left;
double yFramesize = top- bottom;

int* Varloc;
double finesse = 1.0/resolution;
Varloc = new int[5];

Varloc[0] = 1;
Varloc[1] = 1;
Varloc[2] = 1;
Varloc[3] = 1;
Varloc[4] = 1;

int dsize = bem->allSize[ifid]+1;
double* Xtmp;
double* Ytmp;
double* Utmp;
double* Vtmp;
double* Ptmp;
int *Subdom;
int Nvar;
float *Pxtmp;
float *Pytmp;
double* KC;
double* DropX;
double* DropY;
Pxtmp = new float[nOFm/2+bem->nbPanels+20];
Pytmp = new float[nOFm/2+bem->nbPanels+20];

int otmpiox, otmpioy,count;


char strL[128];
char scrD[128];
sprintf(scrD,"%s",workdir.c_str());
sprintf(strL,"%s/ulam2d%03d.vtk",workdir.c_str(),nbSlice);
```

Spatial grid and field variables are initialized

```
otmpiox = int(xFramesize*resolution);
otmpioy = int(yFramesize*resolution);
Nvar = (otmpiox+2)*(otmpioy+2);
```

```
Xtmp = (double*) calloc (Nvar,sizeof(double));
Ytmp = (double*) calloc (Nvar,sizeof(double));
Utmp = (double*) calloc (Nvar,sizeof(double));
Vtmp = (double*) calloc (Nvar,sizeof(double));
Ptmp = (double*) calloc (Nvar,sizeof(double));
Subdom = (int*) calloc (Nvar,sizeof(int));

count= 0;
for (int my=0; my<(yFramesize*resolution+0.5); my++) {
    yls = bottom+my*finesse;
    for(int mx=0; mx<(xFramesize*resolution+0.5); mx++) {
        xls = left+mx*finesse;
        Xtmp[count] = xls;
        Ytmp[count] = yls;
        Subdom[count] = FigInOut(xls,yls,finesse);
        count++;
        otmpiox = mx;
    }
    otmpioy = my;
}

otmpiox++;
otmpioy++;
```

Compute the field values U, V and P for all X and Y.

```
Nvar = count;
BuildVisual(count, Xtmp, Ytmp, Utmp, Vtmp, Ptmp, Subdom);
```

The file header is written

```
FILE* vtkfile;
vtkfile = fopen(strL,"w");
fprintf(vtkfile, "# vtk DataFile Version 2.0\n");
fprintf (vtkfile, "Ulambator Output File for Slice %d \n",nbSlice);
fprintf(vtkfile, "ASCII\n");
fprintf(vtkfile, "DATASET STRUCTURED_POINTS\n");
fprintf(vtkfile, "DIMENSIONS %d %d 1\n", otmpiox, otmpioy );
fprintf(vtkfile, "ASPECT_RATIO %f %f 1\n",1.0/resolution, 1.0/resolution);
fprintf(vtkfile, "ORIGIN %f %f 0\n",left, bottom);
fprintf(vtkfile, "POINT_DATA %d\n",otmpiox*otmpioy );
```

```
fprintf(vtkfile, "SCALARS pressure float\n");
fprintf(vtkfile, "LOOKUP_TABLE default\n");
count = 0;
for (int my=0; my<otmpioy; my++) {
    for(int mx=0; mx<otmpiox; mx++) {
        fprintf(vtkfile, "%1.9f ", Ptmp[count]);
        count++;
    }
}

fprintf(vtkfile, "SCALARS domain float\n");
fprintf(vtkfile, "LOOKUP_TABLE default\n");

count = 0;
for (int my=0; my<otmpioy; my++) {
    for(int mx=0; mx<otmpiox; mx++) {
        fprintf(vtkfile, "%d ", Subdom[count]);
        count++;
    }
}


fprintf(vtkfile, "VECTORS velocity float\n");
count = 0;
for (int my=0; my<otmpioy; my++) {
    for(int mx=0; mx<otmpiox; mx++) {
        fprintf(vtkfile, "%1.9f %1.9f 0.0\n", Utmp[count], Vtmp[count]);
        count++;
    }
}

fclose(vtkfile);

std::cout << "VTK Data written, ";
```

Write Geometry into a seperate vtk file

```
int totalsegm = 0;
count = 0;
totalsegm = bem->getAllSize()/2;

sprintf(strL,"%s/ulamgeo2d%03d.vtk",workdir.c_str(),nbSlice);
vtkfile = fopen(strL,"w");
fprintf(vtkfile, "# vtk DataFile Version 3.0\nvtk output\nASCII\nDATASET POLYDATA\n");
```

```
fprintf(vtkfile, "POINTS %d float\n", totalsegm);
for (int s=0; s<bem->nbBlocks; s++) {
    DropX = bem->Elements[s].XId[0];
    DropY = bem->Elements[s].YId[0];
    dsize = bem->Elements[s].getFullSize();
    for (int i=0; i<dsize; i++) {
        fprintf(vtkfile, "%1.8f %1.8f 0.0\n", DropX[i], DropY[i]);
    }
}

count = 0;
fprintf(vtkfile, "LINES %d %d\n", totalsegm, 3*totalsegm);
for (int s=0; s<bem->nbBlocks; s++) {
    dsize = bem->Elements[s].getFullSize();
    for (int i=0; i<dsize-1; i++) {
        fprintf(vtkfile, "2 %d %d\n", i+count, i+1+count);
    }
    if(dsize>0){
        fprintf(vtkfile, "2 %d %d \n", dsize-1+count, count);
    }
    count += dsize;
}
fprintf(vtkfile, "POINT_DATA %d\n", totalsegm);
fprintf(vtkfile, "scalars pointvar float\n");
fprintf(vtkfile, "LOOKUP_TABLE default\n");
for (int s=0; s<bem->nbBlocks; s++) {
    dsize = bem->Elements[s].getFullSize();
    if (s>=ifblock) {
        bem->UpdateCourbure(s);
        KC =  bem->Elements[s].KC;
        for (int i=0; i<dsize; i++) {
            fprintf(vtkfile, "%1.8f ",KC[i]);
        }
    }else{
        for (int i=0; i<dsize; i++) {
            fprintf(vtkfile, "0.0 ");
        }
    }
}

fclose(vtkfile);

std::cout << "VTK Geometry written.\n";

delete Xtmp;
delete Ytmp;
delete Utmp;
delete Vtmp;
```

```
    delete Ptmp;

}



void MatrixBlock::VisMatlab(int nbSlice, int resolution, double left, double right, double b
```

Writes field variables to a Matlab readable file

```
    if(prcndsing){
        cout<<"PreCondense and Output will not work yet"<<flush;
    }

    double xls, yls;
    FILE* pfile;

    double xFramesize = right-left;
    double yFramesize = top- bottom;

    int Check;
    int* Varloc;

    double finesse = 1.0/resolution;
    Varloc = new int[5];

    Varloc[0] = 1;
    Varloc[1] = 1;
    Varloc[2] = 1;
    Varloc[3] = 1;
    Varloc[4] = 1;

    double* Xtmp;
    double* Ytmp;
    double* Utmp;
    double* Vtmp;
    double* Ptmp;
    int *Subdom;
    int Nvar;
    float *Pxtmp;
    float *Pytmp;

    Pxtmp = new float[nOFm+bem->nbPanels];
    Pytmp = new float[nOFm+bem->nbPanels];
```

```
int otmpiox, otmpioy,count;


char strL[128];
char scrD[128];
sprintf(scrD,"%s",workdir.c_str());
sprintf(strL,"%s/ulamviz%d.dat",workdir.c_str(),nbSlice);
```

Here true ouput starts

```
otmpiox = int(xFramesize*resolution);
otmpioy = int(yFramesize*resolution);
Nvar = (otmpiox+2)*(otmpioy+2);

Xtmp = (double*) calloc (Nvar,sizeof(double));
Ytmp = (double*) calloc (Nvar,sizeof(double));
Utmp = (double*) calloc (Nvar,sizeof(double));
Vtmp = (double*) calloc (Nvar,sizeof(double));
Ptmp = (double*) calloc (Nvar,sizeof(double));
Subdom = (int*) calloc (Nvar,sizeof(int));

pfile = fopen(strL,"w");

count= 0;

for (int my=0; my<(yFramesize*resolution+0.5); my++) {
    yls = bottom+my*finesse;

    for(int mx=0; mx<(xFramesize*resolution+0.5); mx++) {
        xls = left+mx*finesse;
        Xtmp[count] = xls;
        Ytmp[count] = yls;
        Subdom[count] = FigInOut(xls,yls,finesse);
        count++;
        otmpiox = mx;
    }
    otmpioy = my;
}

otmpiox++;
otmpioy++;

fprintf(pfile, "Ulambator Visual Output for Matlab\n %d %d %1.6f %1.6f %1.6f %1.6f\n", c

Nvar = count;
```

```
    BuildVisual(count, Xtmp, Ytmp, Utmp, Vtmp, Ptmp, Subdom);

    for (int j=0; j<Nvar; j++) {
        fprintf(pfile, "%1.9f %1.9f %1.9f %1.9f %1.9f %d\n", Utmp[j], Vtmp[j], Ptmp[j], Xtmp
    }

    Check = fclose(pfile);

    if(Check==0){
        cout << "VisMatlab Data written, \n";
    } else {
        cout << "VisMatlab Binary Data Error, \n";
    }

    delete Xtmp;
    delete Ytmp;
    delete Utmp;
    delete Vtmp;
    delete Ptmp;

}
```

Ubiquitous subroutines

```
void MatrixBlock::ResetMR(double *Mat, double * Rhs){
```

Reset Matrix and righthandside

```
    for(int n=0; n<2*fixsize*prcndsing; n++){
            Rhs[n] = u2vec[n];
    }

    for(int n=2*fixsize*prcndsing; n<nOFm; n++){
        Rhs[n] = 0;
    }
    for(int n=4*fixsize*fixsize*prcndsing; n<nOFm*nOFm; n++){
            Mat[n] = 0;
    }
}

int MatrixBlock::UpdateNOFM(){
```

Determine the size of the matrix

```
    int status = 0;
    int allsize = bem->getAllSize();

    double *Atmp;

    if (allsize>nOFm) {
        Atmp = (double*) calloc (4*fixsize*fixsize,sizeof(double));
        memcpy(Atmp, A, 4*fixsize*fixsize*sizeof(double));

        free(A);
        free(u1vec);
        free(rhs);
        nOFm = allsize;
        A = (double*) calloc ((nOFm+10)*(nOFm+10),sizeof(double));
        u1vec = (double*) calloc (nOFm+10,sizeof(double));
        rhs = (double*) calloc (nOFm+10,sizeof(double));

        memcpy(A, Atmp, 4*fixsize*fixsize*sizeof(double));
        free(Atmp);
        cout<<"m";
    }
    nOFm = allsize;
    bem->UpdateSize();
    return status;
}

int MatrixBlock::FigInOut(double xp, double yp, double crit){
```

Figure out if inside a droplet or not, important for visual export

```
    int inside = -1;
    int size;
    double dphi;
    double *Xc, *Yc;
    double dx1, dx2, dy1, dy2,rr;
    bool unbreaked = 1;

    if (crit>0.0) {
        crit = 0.5*pow(bem->rlim,2);
    }

    for (int i=dynblock; i<nbBlocks; i++) {
```

```
        dphi = 0;
        size = bem->allSize[bem->ifid+i-ifblock];
        Xc = &bem->Elements[i].XId[0][-1];
        Yc = &bem->Elements[i].YId[0][-1];

        for (int j=0;j<size+1 ; j++) {
            dx1 = Xc[j]-xp;
            dx2 = Xc[j+1]-xp;
            dy1 = Yc[j]-yp;
            dy2 = Yc[j+1]-yp;
            rr = sqrt(dx1*dx1+dy1*dy1)*sqrt(dx2*dx2+dy2*dy2);
            if (rr<crit){
                inside = fixsize;
                for (int k=ifblock; k<i; k++) {
                    inside += bem->Elements[k].getFullSize();
                }
                inside += j;
                unbreaked = 0;
                break;
            }
            dphi += asin((dy2*dx1-dx2*dy1)/rr);
        }
        if ((fabs(dphi)>M_PI)&&(unbreaked)) {
            inside = -i-2;
            break;
        }
    }



    return inside;
}


void MatrixBlock::computeGT(int dolog, double x1, double y1, double& G11, double& G12, doubl
```

Calculate the G and T functions

```
    double x2, y2, r2, r1, kr, kr2, t, dtmp, dtmp1, K0, K1, A1, A2, A3, B1, B2, B3, B4;

    x2 = x1*x1;
    y2 = y1*y1;
    r2 = x2+y2;
    r1 = sqrt(r2);
    kr = r1*kF;
```

```
    kr2 = kF*kF*r2;
    r2 = 1.0/r2;            // only 1/r^2 is used, this redefinition save costly division o
```

Outer solution

```
    if (kr>2.0) {
        t = 2.0/kr;
        dtmp1 = exp(-kr)/sqrt(kr);
        dtmp = (((((0.00053208*t-0.0025154)*t+0.00587872)*t-0.01062446)*t+
                 0.02189568)*t-0.07832358)*t+1.25331414;
        K0 = dtmp*dtmp1;
        dtmp = (((((-0.00068245*t+0.00325614)*t-0.00780353)*t+0.01504268)*t-
                 0.0365562)*t+0.23498619)*t+1.25331414;
        K1 = dtmp*dtmp1;

        A1 = -2.*(K0+K1/kr-1./kr2)-(1-dolog)*log(r1);
        A2 = 2.*(K0+2*K1/kr-2./kr2);
        A3 = -K1*kr;
        B1 = -2.*x1*r2;
        B2 = -2.*y1*r2;
        B3 = y2*r2*(4.*A2-2.*A3);
        B4 = x2*r2*(4.*A2-2.*A3);

        T111 = B1*(-1.+A2-B3);
        T122 = B1*(-1.-A2+B3);
        T112 = B2*(A3-A2+B4);

        T211 = B2*(-1.-A2+B4);
        T222 = B2*(-1.+A2-B4);
        T212 = B1*(A3-A2+B3);

        G11 = A1+x2*r2*A2;
        G12 = x1*y1*r2*A2;
        G22 = A1+y2*r2*A2;
```

Inner solution

```
    } else if(kr>0.0){
        t = log(kr/2.0)+0.57721566490153;

        // besselk0 minus times x^2 -log(kr/2) -0.57721566490153
```

```
        K0 = 0.25*( 1.-t +kr2*0.03125*( 3.-2.*t +kr2*0.009259259*( 11.-6.*t + kr2*0.0078125*

        // besselk1 minus times x^2 1/kr + kr*0.25*( -1+2*t
        K1 =  0.0625*( -5+4*t +kr2*0.0555555555*( -5+3*t +kr2*0.002604166666*( -47+24*t +kr2

        A1 = -(2.*K0+0.5*K1)*kr2 +log(kF/2.)+dolog*log(r1)+1.0772156649;     // -log(r1)
        A2 = (2.*K0+K1);                                                    // *kr2 + 1
        A3 = -0.25*(K1*kr2+2*log(kr/2.)+0.1544313298);                      // *kr2 + 1

        B1 = 2.*(A2-A3);
        B2 = (4.*A3-8.*A2)*x1+4.*dolog*x1/kr2;
        B3 = (4.*A3-8.*A2)*y1+4.*dolog*y1/kr2;
        dtmp = x2*r2;
        dtmp1 = y2*r2;

                                                                            // dolog add
        T111 = kF*kF*( 2.*A2*x1  +2.*x1*B1+ dtmp*B2 );                       // - 4*x^3/r^4
        T122 = kF*kF*( 2.*A2*x1           + dtmp1*B2 );                     // - 4*x*y^2/r^4
        T112 = kF*kF*(          B1*y1  + dtmp*B3 );                         // - 4*y*x^2/r^4

        T211 = kF*kF*( 2.*A2*y1           + dtmp*B3 );                      // - 4*y*x^2/r^4
        T222 = kF*kF*( 2.*A2*y1  +2.*B1*y1+ dtmp1*B3 );                     // - 4*y^3/r^4
        T212 = kF*kF*(         B1*x1  + dtmp1*B2 );                         // - 4*x*y^2/r^4

                                                                            // dolog add
        A2 = A2*kr2;
        G11 = A1+x2*r2*(A2-1.0);                                            // -log(r1)
        G12 = x1*y1*r2*(A2-1.0);
        G22 = A1+y2*r2*(A2-1.0);

    } else {


        T111 = 4.0*dolog*x1*x2*r2*r2;
        T112 = 4.0*dolog*y1*x2*r2*r2;
        T122 = 4.0*dolog*x1*y2*r2*r2;

        T211 = 4.0*dolog*y1*x2*r2*r2;
        T212 = 4.0*dolog*x1*y2*r2*r2;
        T222 = 4.0*dolog*y1*y2*r2*r2;


        G11 = dolog*log(r1)-x2*r2;
        G12 = -x1*y1*r2;
        G22 = dolog*log(r1)-y2*r2;
    }
}
```

```
void MatrixBlock::computeGO(int dolog, double x1, double y1, double& G11, double& G12, doubl
```

Calculates the G test functions

```
    double x2, y2, r2, r1, kr, t, dtmp, dtmp1, K0, K1, A1, A2, kr2;

    x2 = x1*x1;
    y2 = y1*y1;
    r2 = x2+y2;
    r1 = sqrt(r2);
    kr = r1*kF;
    kr2 = kF*kF*r2;
    r2 = 1.0/r2;
```

Outer solution

```
    if (kr>2.0) {
        if (dolog==0) {
            cout<<"e"<<flush;
        }
        t = 2.0/kr;
        dtmp1 = exp(-kr)/sqrt(kr);
        dtmp = (((((0.00053208*t-0.0025154)*t+0.00587872)*t-0.01062446)*t+
                 0.02189568)*t-0.07832358)*t+1.25331414;
        K0 = dtmp*dtmp1;
        dtmp = (((((-0.00068245*t+0.00325614)*t-0.00780353)*t+0.01504268)*t-
                 0.0365562)*t+0.23498619)*t+1.25331414;
        K1 = dtmp*dtmp1;

        A1 = -2.*(K0+K1/kr-1./kr2)-(1.-dolog)*log(r1);
        A2 = 2.*(K0+2.*K1/kr-2./kr2);

        G11 = A1+x2*r2*A2;
        G12 = x1*y1*r2*A2;
        G22 = A1+y2*r2*A2;
```

Inner solution

```
    } else if (kr>0.0) {
        t = log(kr/2.0)+0.57721566490153;

        // besselk0 minus times x^2 -log(kr/2) -0.57721566490153
        K0 = 0.25*( 1-t +kr2*0.03125*( 3-2*t +kr2*0.009259259*( 11-6*t + kr2*0.0078125*( 25-


        // besselk1 minus times x^2 1/kr + kr*0.25*( -1+2*t
        K1 =  0.0625*( -5+4*t +kr2*0.0555555555*( -5+3*t +kr2*0.002604166666*( -47+24*t +kr2

        //      double BesselK0 = K0*kr2 - log(kr/2)-0.57721566490153;
        //      double BesselK1 = K1/4*kr2*kr+1.0/kr+kr*(log(kr/2)/2+0.57721566490153/2-1.0/

        A1 = -(2.*K0+0.5*K1)*kr2 +log(kF/2.)+dolog*log(r1)+1.0772156649;
        A2 = (2.*K0+K1)*kr2;

        G11 = A1+x2*r2*(A2-1.0);
        G12 = x1*y1*r2*(A2-1.0);
        G22 = A1+y2*r2*(A2-1.0);
```

Stokes solution

```
    } else {

        G11 = dolog*log(r1)-x2*r2;
        G12 = -x1*y1*r2;
        G22 = dolog*log(r1)-y2*r2;

    }
}

void MatrixBlock::computeGOH(double x1, double y1, double& G11, double& G12, double& G22){
```

Calculates the G test functions

```
    double x2, y2, r2, r1, kr, t, dtmp, dtmp1, K0, K1, A1, A2;

    x2 = x1*x1;
    y2 = y1*y1;
    r2 = x2+y2;
```

```
    r1 = sqrt(r2);
    kr = r1*kF;
```

Outer solution

```
        t = 2.0/kr;
        dtmp1 = exp(-kr)/sqrt(kr);
        dtmp = (((((0.00053208*t-0.0025154)*t+0.00587872)*t-0.01062446)*t+
                 0.02189568)*t-0.07832358)*t+1.25331414;
        K0 = dtmp*dtmp1;
        dtmp = (((((-0.00068245*t+0.00325614)*t-0.00780353)*t+0.01504268)*t-
                 0.0365562)*t+0.23498619)*t+1.25331414;
        K1 = dtmp*dtmp1;

        A1 = -2*(K0+K1/kr-1./(kF*kF*r2));
        A2 = 2*(K0+2*K1/kr-2./(kF*kF*r2));

        G11 = A1+x2/r2*A2;
        G12 = x1*y1/r2*A2;
        G22 = A1+y2/r2*A2;

}

void MatrixBlock::computeGTH(int dolog, double x1, double y1, double& G11, double& G12, doub
```

Calculates the G test functions

```
    double x2, y2, r2, r1, kr, kr2, t, dtmp, dtmp1, K0, K1, A1, A2, A3, B1, B2, B3, B4;

    x2 = x1*x1;
    y2 = y1*y1;
    r2 = x2+y2;
    r1 = sqrt(r2);
    kr = r1*kF;
    kr2 = kF*kF*r2;
    r2 = 1.0/r2;            // only 1/r^2 is used, this redefinition save costly division o
```

Outer solution

```
        t = 2.0/kr;
        dtmp1 = exp(-kr)/sqrt(kr);
        dtmp = (((((0.00053208*t-0.0025154)*t+0.00587872)*t-0.01062446)*t+
                  0.02189568)*t-0.07832358)*t+1.25331414;
        K0 = dtmp*dtmp1;
        dtmp = (((((-0.00068245*t+0.00325614)*t-0.00780353)*t+0.01504268)*t-
                  0.0365562)*t+0.23498619)*t+1.25331414;
        K1 = dtmp*dtmp1;

        A1 = -2.*(K0+K1/kr-1./kr2)-(1-dolog)*log(r1);
        A2 = 2.*(K0+2*K1/kr-2./kr2);
        A3 = -K1*kr;
        B1 = -2.*x1*r2;
        B2 = -2.*y1*r2;
        B3 = y2*r2*(4.*A2-2.*A3);
        B4 = x2*r2*(4.*A2-2.*A3);

        T111 = B1*(-1.+A2-B3);
        T122 = B1*(-1.-A2+B3);
        T112 = B2*(A3-A2+B4);

        T211 = B2*(-1.-A2+B4);
        T222 = B2*(-1.+A2-B4);
        T212 = B1*(A3-A2+B3);

        G11 = A1+x2*r2*A2;
        G12 = x1*y1*r2*A2;
        G22 = A1+y2*r2*A2;


}

void MatrixBlock::computeMP(double x1, double y1, double &M1, double &M2, double &P11, doubl
```

Calculate the M and P test function for the pressure

```
    double x2, y2, r2, r1;
    x2 = x1*x1;
    y2 = y1*y1;
    r2 = x2+y2;
    r1 = sqrt(r2);

    M1 = x1/r2;
```

```
    M2 = y1/r2;
    P11 = kF*kF*log(r1) + 2.0/r2 -4*x2/(r2*r2);
    P12 = -4*x1*y1/(r2*r2);
    P22 = kF*kF*log(r1) + 2.0/r2 -4*y2/(r2*r2);
}



void MatrixBlock::IntForce(int dropId){
```

Integrate the Forces on a droplet

```
    double tang, norl, dx, dy, r;
    int size = bem->Elements[dropId].getFullSize();
    double *XA, *YA;

    XA = bem->Elements[dropId].XId[0];
    YA = bem->Elements[dropId].YId[0];
```

Wrap solution

```
    u1vec[2*fixsize+nCK*size] = u1vec[2*fixsize];
    u1vec[2*fixsize+nCK*size+1] = u1vec[2*fixsize+1];

    tang = 0;
    norl = 0;
```

Integrate force

```
    for (int k=0; k<size; k++) {
        dx = XA[k+1]-XA[k];
        dy = YA[k+1]-YA[k];
        r = sqrt(dx*dx+dy*dy);

        norl += r*(u1vec[2*fixsize+nCK*k]);
        tang += r*(u1vec[2*fixsize+nCK*k+1]);
    }
```

```
    cout<<" Sum of Forces around the droplet are: "<<norl<<" and "<<tang<<"\n";
}

double MatrixBlock::Residuum(double *sol, double &omega, double oldcvgc){
```

Calculate the difference between two solutions

```
    double rest = 0;


    for (int k=2*fixsize; k<nOFm; k++) {
```

```
   zvec[k] = fabs(xvec[k]-u2vec[k]);
```

```
        rest += pow(sol[k-2*fixsize]-rhs[k],2);
    }
    rest = sqrt(rest)/(2.0*fixsize);
```

Relaxe if convergence fails - this was not so helpful yet

```
    if (rest>=oldcvgc) {
        omega = omega/2.0;
        cout<<","<<flush;
    }else{
        omega = min(omega*1.1,1.0);
    }
```

Relax convergence rate for stability with omega

```
    for (int k=2*fixsize; k<nOFm; k++) {
        rhs[k] = sol[k-2*fixsize]+omega*(rhs[k]-sol[k-2*fixsize]); // Smartass
//      cout<<k<<" "<<sol[k-2*fixsize]<<" "<<rhs[k]<<"\n";
//      rhs[k] += omega*(-rhs[k]+sol[k-2*fixsize]);
    // min(1.0/sqrt(rest), 1.0)
```

```
    }

    return rest;
}
```

Created by Mathias Nagel on 23.03.10.

Modified by Mathias Nagel on 16.3.2015.

Ulambators BoundaryBlock class (`bndblock.cpp`) v0.3

This class contains functions to create and store information of boundaries and interfaces

Contents:

- Initialisation
- Wall and Droplet creation
- Droplet evolution
- Special functions for drop merging, repulsion etc
- Data writing
- Drolet statistics
- Remeshing procedures
- Ubiquitous subroutines

The header optionally contains OpenMP support.

```
#include "bndblock.h"
#include "bndelement.h"
#ifdef _WIN32
#define _USE_MATH_DEFINES // for C++
#include <cmath>
#endif
#include <math.h>
#include <iostream>
#include <stdio.h>
#include "combase.h"
#include <stdlib.h>
#include <fstream>
#include <assert.h>
#ifdef _WIN32
#include <Winsock2.h>
#include <algorithm>
#include <time.h>
#endif
using namespace std;
```

Initialization

```
BndBlock::BndBlock(int blocks){
    Elements = new FlowFace[blocks]();

    nbBlocks = blocks;
    runstep = 0;
    runtime = 0.0;
    nbPanels = -1;

    rlim = 0; //elementsPerLength;
    workdir = new char[120];
    panel2block = new int[64]; // only 64 panels can exist, increase if mor are needed
    writeEnabled = 0;
    remesh_threshold = 2.0;
    SeedOption = 0; // 1 - Variable, else Constant

    IGP = new double[6];
    IGW = new double[6];

    IGP[0] = -0.932469514203152;
    IGP[1] = -0.661209386466265;
    IGP[2] = -0.238619186083197;
    IGP[3] = 0.238619186083197;
    IGP[4] = 0.661209386466265;
    IGP[5] = 0.932469514203152;
    IGW[0] = 0.171324492379170;
    IGW[1] = 0.360761573048139;
    IGW[2] = 0.467913934572691;
    IGW[3] = 0.467913934572691;
    IGW[4] = 0.360761573048139;
    IGW[5] = 0.171324492379170;

}

void BndBlock::Init(){
    int counter = 0;

    ifid = 0;
    ifblock = 0;
    dynblock = 0;
    dynpanels = 0;
    nbPanels = 0;
    for (int k=0; k<nbBlocks; k++) {
        for (int j=nbPanels; j<(nbPanels+Elements[k].nbPanels); j++) {
            panel2block[j] = k;
        }
        nbPanels += Elements[k].nbPanels;
    }
```

```
allBCX = (double*) calloc (nbPanels,sizeof(double));
allBCY = (double*) calloc (nbPanels,sizeof(double));
allBCZ = (double*) calloc (nbPanels,sizeof(double));
allXId = (double**) calloc (nbPanels,sizeof(double*));
allYId = (double**) calloc (nbPanels,sizeof(double*));
allBCtype = (int*) calloc (nbPanels,sizeof(int));
allSize = (int*) calloc (nbPanels,sizeof(int));
allRdist = (double**) calloc (nbPanels,sizeof(double*));
allCdist = (double**) calloc (nbPanels,sizeof(double*));
allWdist = (double**) calloc (nbPanels,sizeof(double*));

for (int k=0; k<nbBlocks; k++) {
    switch (Elements[k].BCkind) {
        case 0:     // Outer Boundary
            for (int j=0; j<Elements[k].nbPanels; j++) {
                allBCX[counter] = Elements[k].BCvalueX[j];
                allBCY[counter] = Elements[k].BCvalueY[j];
                allBCtype[counter] = Elements[k].BCtype[j];

                allXId[counter] = Elements[k].XId[j];
                allYId[counter] = Elements[k].YId[j];
                allSize[counter] = Elements[k].PanelSize[j];

                counter++;
            }
            break;
   /*     case 1:     // Fibers
            for (int j=0; j<Elements[k].nbPanels; j++) {
                allBCX[counter] = 0.0;
                allBCY[counter] = 0.0;
                allBCZ[counter] = 0.0;
                allBCtype[counter] = 4;

                allXId[counter] = Elements[k].XId[j];
                allYId[counter] = Elements[k].YId[j];
                allSize[counter] = Elements[k].PanelSize[j];

                counter++;
            }
            break; */
        case 2:     // Droplet
            allRdist[counter] = Elements[k].Rdist;
            allCdist[counter] = Elements[k].Cdist;
            allWdist[counter] = Elements[k].Wdist;
            for (int j=0; j<Elements[k].nbPanels; j++) {
                allBCX[counter] = 0.0;
                allBCY[counter] = 0.0;
                allBCtype[counter] = 21;
```

```
                    if (Elements[k].BCtype[j]==22) {
                        allBCtype[counter] = 22;
                    }

                    allXId[counter] = Elements[k].XId[j];
                    allYId[counter] = Elements[k].YId[j];
                    allSize[counter] = Elements[k].PanelSize[j];

                    counter++;
                }
                break;
        /*      case 3:     // Bordtrac
                allRdist[counter] = Elements[k].Rdist;
                allCdist[counter] = Elements[k].Cdist;
                allWdist[counter] = Elements[k].Wdist;
                for (int j=0; j<Elements[k].nbPanels; j++) {
                    allBCX[counter] = 0.0;
                    allBCY[counter] = 0.0;
                    allBCtype[counter] = 23;

                    allXId[counter] = Elements[k].XId[j];
                    allYId[counter] = Elements[k].YId[j];
                    allSize[counter] = Elements[k].PanelSize[j];

                    counter++;
                }
                break; */
            default:
                cout<<"No valid boundary selected !"<<flush;
                exit(0);
                break;
        }


        if (Elements[k].BCkind<2) {
            ifid += Elements[k].nbPanels;
            ifblock++;
        }
        if (Elements[k].BCkind<1) {
            dynpanels += Elements[k].nbPanels;
            dynblock++;
        }

    }

    WrapAll();
    allStock();
}
```

```
void BndBlock::PanelInit(int blockId, int Size, int ofKind, int Btype, double *BCvalue, doub
    // One could check here that the number of panels is not exceeded
    int index;
    int numPanels = Elements[blockId].nbPanels;

    if (numPanels>=1) {
        index = Elements[blockId].PanelSize[numPanels-1];
        if(!((fabs(Elements[blockId].XId[numPanels-1][index]-PosVec[0])<1e-6)&&(fabs(Element
            std::cout<<"ATTENTION: Geometry NOT closed, check boundary output!!!\n";
        }
    }

    index = 0;

    for (int k = 0; k<numPanels; k++) {
        index += Elements[blockId].PanelSize[k];
    }
    Elements[blockId].XId[numPanels] = &Elements[blockId].XE[index+5];
    Elements[blockId].YId[numPanels] = &Elements[blockId].YE[index+5];
    Elements[blockId].PanelSize[numPanels] = Size;



    switch (ofKind){
        case 0:     // Straight line for walls or in- outflow
            DrawLine(Elements[blockId].XId[numPanels], Elements[blockId].YId[numPanels], Pos
            break;
        case 1:     // Curved line for walls Pos4 is curvature positive for outwards
            DrawSegm(Elements[blockId].XId[numPanels], Elements[blockId].YId[numPanels], Pos
            break;
        case 2:     // Pancake droplet
            DoCircle(Size, PosVec[2], Elements[blockId].XId[numPanels], Elements[blockId].YI
            break;
        case 3:     // Elliptic droplet Pos0 is the major halfaxis and its inverse minor ha
            DoEllipse(Size, PosVec[2], PosVec[3], Elements[blockId].XId[numPanels], Elements
            break;
        case 4:     // droplet with recangular middle section and caps, Pos0 is length of m
            DoEgg(Size, PosVec[2], PosVec[3], Elements[blockId].XId[numPanels], Elements[blo
            break;
        case 5:     // Droplet interface for attached boundaries
            DrawLine(&Elements[blockId].XId[numPanels][-1], &Elements[blockId].YId[numPanels
            break;
        case 6:     // Hyperbolic walls
            DrawHyperbel(Elements[blockId].XId[numPanels], Elements[blockId].YId[numPanels],
        case 7:     // Sinus shape walls with amplitude and wavenumber
            DrawSine(Elements[blockId].XId[numPanels], Elements[blockId].YId[numPanels], Pos
            break;
```

```
    }

    Elements[blockId].BCtype[numPanels] = Btype;
    Elements[blockId].BCvalueX[numPanels] = BCvalue[0];
    Elements[blockId].BCvalueY[numPanels] = BCvalue[1];
    numPanels++;
    Elements[blockId].nbPanels = numPanels;

    if((numPanels>1)&&(numPanels==Elements[blockId].nbPanRes)){
        if(!((fabs(Elements[blockId].XId[numPanels-1][Size]-Elements[blockId].XId[0][0])<1e-
            std::cout<<"ATTENTION: Geometry NOT closed, check boundary output!!!\n";
        }
        Elements[blockId].XId[numPanels-1][Size+1] = Elements[blockId].XId[0][1];
        Elements[blockId].YId[numPanels-1][Size+1] = Elements[blockId].YId[0][1];
    }
    std::cout<<" Block "<<blockId<<" Panel "<<numPanels<<" of "<<Elements[blockId].nbPanRes<
}
```

Dedicated to droplet creation

```
void BndBlock::DoCircle(int prec, double radius, double *Xcir, double *Ycir, double XPos, do
```

Creates droplet, optional with perturbation single/multiple/noise

```
    double radstep = 2*M_PI/prec;
    double radus;

    srand(time(NULL));

    radus = 0.0;
    for(int n=0; n<=prec; n++){

        if (Nmode>0) {
            radus = Namp*(cos(Nmode*n*radstep));
        } else {
            radus = Namp*rand()/(float(RAND_MAX)+1);
        }

        Xcir[n] = radius*(1.0+radus)*cos(n*radstep)+ XPos;
        Ycir[n] = fabs(radius)*(1.0+radus)*sin(n*radstep)+ YPos;
    }
```

```
}

void BndBlock::DoEgg(int prec, double longeur, double largeur, double *Xcir, double *Ycir, d
```

Make droplet which is elongated by a straight section - Option 4

```
    double radius = largeur;
    double signum = 1.0-2.0*(longeur<0);
    longeur= fabs(longeur);
    int straight = (prec*longeur/(2*M_PI*radius+2*longeur));
    int curved = prec/2-straight;

    double radstep = M_PI/curved;
    int n=0;
    int keep;



    for(int j=0; j<curved; j++){
        Xcir[n] = -radius*cos(n*radstep)+X0;
        Ycir[n] = -signum*(longeur*1.0/2.0+radius*sin(n*radstep))+Y0;
        n++;
    }
    keep=n;
    for(int j=0; j<straight; j++){
        Xcir[n] = radius+X0;
        Ycir[n] = signum*longeur*((n-keep)*1.0/straight-0.5)+Y0;
        n++;
    }
    keep = n;
    for(int j= 0; j<curved; j++){
        Xcir[n] = radius*cos((n-keep)*radstep)+X0;
        Ycir[n] = signum*(longeur*1.0/2.0+radius*sin((n-keep)*radstep))+Y0;
        n++;
    }
    keep = n;
    for(int j=keep; j<prec; j++){
        Xcir[n] = -radius+X0;
        Ycir[n] = signum*longeur*(0.5-(n-keep)*1.0/straight)+Y0;
        n++;
    }
}

void BndBlock::DoEllipse(int prec, double axisx, double axisy, double *Xcir, double *Ycir, d
    // Make droplet which is elliptic major/minor axisx and axisy
```

```
    double radstep = 2*M_PI/prec;

    for(int n=0; n<prec; n++){
        Xcir[n] = axisx*cos((n)*radstep)+X0;
        Ycir[n] = axisy*sin((n)*radstep)+Y0;
    }
}



void BndBlock::DrawLine(double *Xdraw, double *Ydraw, double X0, double Y0, double X1, doubl
    // Standard routine for straight lines refined at the end. Ovstp is tunable.
    double dx = (X1-X0);
    double dy = (Y1-Y0);
    double stlen = 2.0/nElm;
    double AQ, BQ, Ovstp,Sva;
    int keep=0;
    Ovstp = 1.1;

    if (Ovstp ==1) {
        double stlen = 1.0/nElm;
        for (int j=0; j<=nElm; j++) {
            Xdraw[j] = X0+dx*stlen*j;
            Ydraw[j] = Y0+dy*stlen*j;
        }
    } else {
        AQ = 2.5*(2-Ovstp)/(Ovstp-1);
        BQ = AQ+4/(2+AQ)-2;


        for (int j=0; j<nElm/2; j++) {
            Sva = j*1.0/nElm;
            //        stlen = Sva;
            stlen = (AQ*Sva+1.0/(1.0+Sva*AQ)-1.0)/BQ;
            Xdraw[j] = X0+dx*stlen;
            Ydraw[j] = Y0+dy*stlen;
            keep = j;
        }

        for (int j=keep; j<=nElm; j++) {
            Sva = 1-j*1.0/nElm;
            //        stlen = 1.0-Sva;
            stlen = 1.0 - (AQ*Sva+1.0/(1.0+Sva*AQ)-1.0)/BQ;
            Xdraw[j] = X0+dx*stlen;
            Ydraw[j] = Y0+dy*stlen;
        }
    }
}
```

```
void BndBlock::DrawSegm(double*Xdraw, double *Ydraw, double X0, double Y0, double X1, double
    // Makes curved boundary segments, needs care when used
    double D = sqrt((X1-X0)*(X1-X0) + (Y1-Y0)*(Y1-Y0));
    double Ym = -R/fabs(R)*sqrt(R*R - D*D/4);
    double gam = asin(D/(2*R));
    double Xm = 0.5*(X0+X1) - Ym*(Y1-Y0)/D;
    double phi;
    double stlen = 1.0/nElm;
    double AQ, BQ, Ovstp,Sva;
    int keep=0;
    Ovstp = 1.3;
    AQ = 2.5*(2-Ovstp)/(Ovstp-1);
    BQ = AQ+4/(2+AQ)-2;

    Ym = 0.5*(Y0+Y1) + Ym*(X1-X0)/D;

    for (int j=0; j<nElm/2; j++) {
        Sva = j*1.0/nElm;
        stlen = (AQ*Sva+1.0/(1.0+Sva*AQ)-1.0)/BQ;
        phi = gam - stlen*gam*2.0;
        Xdraw[j] = Xm - R*(sin(phi)*(X1-X0)/D + cos(phi)*(Y1-Y0)/D);
        Ydraw[j] = Ym + R*(cos(phi)*(X1-X0)/D - sin(phi)*(Y1-Y0)/D);
        keep = j;
    }

    for (int j=keep; j<=nElm; j++) {
        Sva = 1-j*1.0/nElm;
        stlen = 1.0 - (AQ*Sva+1.0/(1.0+Sva*AQ)-1.0)/BQ;
        phi = gam - stlen*gam*2.0;
        Xdraw[j] = Xm - R*(sin(phi)*(X1-X0)/D + cos(phi)*(Y1-Y0)/D);
        Ydraw[j] = Ym + R*(cos(phi)*(X1-X0)/D - sin(phi)*(Y1-Y0)/D);
    }
}

void BndBlock::DrawSine(double*Xdraw, double *Ydraw, double X0, double Y0, double X1, double
    // Makes curved boundary segments, needs care when used
    double dx = (X1-X0);
    double dy = (Y1-Y0);
    double dabs = sqrt(dx*dx+dy*dy);
    double stlen = 2.0/nElm;
    double AQ, BQ, Ovstp,Sva;
    int keep=0;
    Ovstp = 1.0;

    if (Ovstp ==1) {
```

```
        double stlen = 1.0/nElm;
        for (int j=0; j<=nElm; j++) {
            Xdraw[j] = X0+dx*stlen*j+dy/dabs*sinamp*sin(sinewave*2.0*j*M_PI/nElm);
            Ydraw[j] = Y0+dy*stlen*j-dx/dabs*sinamp*sin(sinewave*2.0*j*M_PI/nElm);        }
    } else {
        AQ = 2.5*(2-Ovstp)/(Ovstp-1);
        BQ = AQ+4/(2+AQ)-2;


        for (int j=0; j<nElm/2; j++) {
            Sva = j*1.0/nElm;
            //        stlen = Sva;
            stlen = (AQ*Sva+1.0/(1.0+Sva*AQ)-1.0)/BQ;
            Xdraw[j] = X0+dx*stlen;
            Ydraw[j] = Y0+dy*stlen;
            keep = j;
        }

        for (int j=keep; j<=nElm; j++) {
            Sva = 1-j*1.0/nElm;
            //        stlen = 1.0-Sva;
            stlen = 1.0 - (AQ*Sva+1.0/(1.0+Sva*AQ)-1.0)/BQ;
            Xdraw[j] = X0+dx*stlen;
            Ydraw[j] = Y0+dy*stlen;
        }
    }
}

void BndBlock::DrawHyperbel(double*Xdraw, double *Ydraw, double X0, double Y0, double X1, do
    double C = X0*Y0;
    assert(C==(Y1*X1));
    double D1 = X0*X0-Y0*Y0;
    double D2 = X1*X1-Y1*Y1;
    double dd = (D2-D1)/nElm;
    for (int j=0; j<=nElm; j++){
        Ydraw[j] = (( Y0 > 0 ) - ( Y0 < 0 ))*sqrt(sqrt(pow(D1+dd*j,2)/4+C*C)-(D1+dd*j)/2);
        Xdraw[j] = C/Ydraw[j];
    }
}


void BndBlock::DropRead(int dropId, int timestep){
    // Load droplet from solution used for visualization

    if(!writeEnabled){
        std::cout<<"Writing is not enabled! No folder available.\nCall EnableWrite(dir) befo
        exit(0);
    }
```

```
char str[100];
char* thisdir = new char[120];
string kio, kio2;
string delim = " ";  // or " "
double xpos, ypos;
int npos, cutat;

Elements[dropId].XId[0] = &Elements[dropId].XE[5];
Elements[dropId].YId[0] = &Elements[dropId].YE[5];

double *XId = Elements[dropId].XId[0];
double *YId = Elements[dropId].YId[0];

sprintf(thisdir,"%s/drop%its%i.dat", workdir.c_str(), dropId, timestep);
fstream file(thisdir,ios::in);
if (file.is_open()) {
    //      Read Header
    file.getline(str,100);
    kio = str;
    cutat = kio.find_first_of(delim);
    kio2 = kio.substr(0,cutat);
    runtime = strtod(kio2.c_str(),NULL);
    kio = kio.substr(cutat+1);
    cutat = kio.find_first_of(delim);
    kio2 = kio.substr(0,cutat);
    Elements[dropId].Xrelativ = strtod(kio2.c_str(), NULL);
    kio = kio.substr(cutat+1);
    cutat = kio.find_first_of(delim);
    kio = kio.substr(0,cutat);
    Elements[0].Xrelativ = strtod(kio.c_str(), NULL);

    // Read Geometry
    npos =-1;
    while (file.eof()!=1) {
        npos++;
        file.getline(str,100);
        kio = str;
        cutat = kio.find_first_of(delim);
        kio2 = kio.substr(0,cutat);
        kio = kio.substr(cutat+1);
        cutat = kio.find_first_of(delim);
        kio = kio.substr(0,cutat);
        xpos = strtod(kio2.c_str(), NULL);
        ypos = strtod(kio.c_str(), NULL);
        XId[npos] = xpos;
        YId[npos] = ypos;
    }
```

```
        Elements[dropId].nbPanels = 1;
        Elements[dropId].PanelSize[0] = npos;
        runstep = timestep;
        if(nbPanels>=0){
            allSize[ifid+dropId-ifblock] = Elements[dropId].PanelSize[0];
        }
        file.close();
        WrapInterface(dropId);
        Stock(dropId);
        std::cout<<"Drop Loaded with "<<npos<<" Elements.\n";
    }else{
        std::cout<<"Write directory could not be opened\n";
    }
}
```

Dedicated to droplet evolution

```
void BndBlock::Evolve(int blockId, double Cold, double Cnew, double deltaT, double* xvec){
    // Evolves droplets from buffer IntfXY to XYId
    int dynNodes;
    double *Xn, *Yn, *Xo, *Yo;
    dynNodes = 0;

    dynNodes = Elements[blockId].getFullSize();
    Xn = Elements[blockId].XId[0];
    Yn = Elements[blockId].YId[0];
    Xo = &Elements[blockId].IntfX[5];
    Yo = &Elements[blockId].IntfY[5];

    for (int n=0; n<dynNodes; n++) {
        Xn[n] = Cold*Xo[n] + Cnew*Xn[n] + xvec[2*n]*deltaT;//*fabs(lnx);
        Yn[n] = Cold*Yo[n] + Cnew*Yn[n] + xvec[2*n+1]*deltaT;//*fabs(lny);
    }

    WrapInterface(blockId);
}

void BndBlock::Translate(int blockId, double Xdep, double Ydep, double Theta){
    // Displace droplet by a solid body movement
    int dynNodes = Elements[blockId].getFullSize();
    double* XA = Elements[blockId].XId[0];
    double* YA = Elements[blockId].YId[0];

    double* XT = &Elements[blockId].IntfX[5];
```

```
    double* YT = &Elements[blockId].IntfY[5];

    double BY, coz, xcm, ycm;

    getCoM(blockId, xcm, ycm);

    for (int n=0; n<dynNodes; n++) {
        BY = sqrt(pow(XT[n]-xcm,2)+pow(YT[n]-ycm,2));
        coz = acos((XT[n]-xcm)/BY)*((YT[n]-ycm)>=0)+(2*M_PI-acos((XT[n]-xcm)/BY))*((YT[n]-yc

        XA[n] = BY*cos(Theta+coz)+Xdep+xcm;
        YA[n] = BY*sin(Theta+coz)+Ydep+ycm;

    }
    WrapInterface(blockId);
}

void BndBlock::correctArea(int blockId){
    // Corrects the Area by normal displacement of the interface
    double A1 = getArea(blockId);
    double A0 = Elements[blockId].initialArea;
    double circumference = 0;
    int dynNodes = Elements[blockId].getFullSize();
    double* XA = Elements[blockId].XId[0];
    double* YA = Elements[blockId].YId[0];
    double* XT = &Elements[blockId].IntfX[5];
    double* YT = &Elements[blockId].IntfY[5];
    double lnx, lny, rr;

    if(A0!=0){
        WrapInterface(blockId);
        Stock(blockId);
        for (int n=0; n<dynNodes; n++) {
            lnx = YA[n+1]-YA[n];
            lny = XA[n+1]-XA[n];
            rr = sqrt(lnx*lnx+lny*lny);
            circumference += rr;
        }

        circumference = (A0-A1)/circumference;

        for (int n=0; n<dynNodes; n++) {
            lnx = YT[n+1]-YT[n-1];
            lny = XT[n-1]-XT[n+1];
            rr = 1.0/sqrt(lnx*lnx+lny*lny);
            XA[n] += circumference*lnx*rr;
            YA[n] += circumference*lny*rr;
        }
```

```
        WrapInterface(blockId);
    }
}

void BndBlock::correctArea(){
    for(int i=ifblock;i<nbBlocks;i++){
        correctArea(i);
    }
}

void BndBlock::Stock(int blockId){
    // Move droplet positins from XYId to buffer IntfXY
    int size = Elements[blockId].getFullSize();
    for (int n=0; n<size+10;n++){
        Elements[blockId].IntfX[n] = Elements[blockId].XE[n];
        Elements[blockId].IntfY[n] = Elements[blockId].YE[n];
    }
}

void BndBlock::UpdateCourbure(int blockId){
    double *IX, *IY;
    int size;
    double dxl, dxr, dyl, dyr, delta1, delta2;

    IX = Elements[blockId].XId[0];
    IY = Elements[blockId].YId[0];
    size = Elements[blockId].getFullSize();

    for(int n=0; n<size;n++){
        dxl = IX[n]-IX[n-1];
        dxr = IX[n+1]-IX[n];
        dyl = IY[n]-IY[n-1];
        dyr = IY[n+1]-IY[n];
        delta1 = sqrt(dxl*dxl+dyl*dyl);
        delta2 = sqrt(dxr*dxr+dyr*dyr);
        dxl = -2.0*(dxr/delta2-dxl/delta1)/(delta1+delta2);
        dyl = -2.0*(dyr/delta2-dyl/delta1)/(delta1+delta2);
        Elements[blockId].KC[n] = sqrt(dxl*dxl+dyl*dyl)*(1.0-2.0*((dxl*dyr/delta2-dyl*dxr/de
    }
}

void BndBlock::UpdateCourbure(){
    for (int k=ifblock; k<nbBlocks; k++) {
        UpdateCourbure(k);
    }
}
```

```
void BndBlock::UpdateSize(){
    int count=0;
    for (int n=ifblock; n<nbBlocks; n++) {
        for (int m=0; m<Elements[n].nbPanels; m++){
            allSize[ifid+count] = Elements[n].PanelSize[m];
            count++;
        }
    }
}

void BndBlock::WrapInterface(int blockId){
    // Sets the ghost points
    int size = Elements[blockId].getFullSize();
    double *Xp, *Yp;
    if(Elements[blockId].BCkind!=3){
        for (int n=0; n<5; n++) {
            Elements[blockId].XId[0][n-5] = Elements[blockId].XId[0][size-5+n];
            Elements[blockId].YId[0][n-5] = Elements[blockId].YId[0][size-5+n];
            Elements[blockId].XId[0][size+n] = Elements[blockId].XId[0][n];
            Elements[blockId].YId[0][size+n] = Elements[blockId].YId[0][n];
        }
    } else {
        Xp = Elements[blockId].XId[0];
        Yp = Elements[blockId].YId[0];

        // Zero Angle - Problematic because implicit curvature assumes immobile ghostpoints
        Elements[blockId].XId[0][-2] = 2.0*Xp[-1]-Xp[0];
        Elements[blockId].YId[0][-2] = 2.0*Yp[-1]-Yp[0];
        Elements[blockId].YId[0][-3] = 3.0*Yp[-1]-2.0*Yp[0];
        Elements[blockId].XId[0][-3] = 3.0*Xp[-1]-2.0*Xp[0];
        Elements[blockId].XId[0][size+1] = 2.0*Xp[size]-Xp[size-1];
        Elements[blockId].YId[0][size+1] = 2.0*Yp[size]-Yp[size-1];
        Elements[blockId].YId[0][size+2] = 3.0*Yp[size]-2.0*Yp[size-1];
        Elements[blockId].XId[0][size+2] = 3.0*Xp[size]-2.0*Xp[size-1];
    }
}
```

Special functions for topography change currently disabled

void BndBlock::WallRepuls(){

double BndBlock::DropCorrect(int Drop1, int Drop2){

double BndBlock::DropApproach(int Drop1, int Drop2){

void BndBlock::PerformFusion(int Drop1, int Drop2, int proxL, int proxR){

int BndBlock::CheckBreakUp(int blockId, double kF){

void BndBlock::PerformCut(int blockId, int cutHere1, int cutHere2){

Dedicated to data writing

```
void BndBlock::EnableWrite(char* save2dir){
    workdir = save2dir;
    writeEnabled = 1;
}

void BndBlock::WritePos(int blockId, int nbSlice){
```

Write the droplet positions and curvature

```
    if (writeEnabled) {
        char strL[128];
        int nbEle;
        double* Xpos;
        double* Ypos;
        FILE* pfile;

        int saveshed = 0;
        if(blockId == 99){
            saveshed = 1;
            blockId = 1;
        }

        nbEle = Elements[blockId].getFullSize();

        Xpos = Elements[blockId].XId[0];
        Ypos = Elements[blockId].YId[0];

        //  std::cout<<" Elements present "<<nbEle<<" ";

        if (saveshed) {
            sprintf(strL,"%s/drop1ts999.dat",workdir.c_str());
            pfile = fopen(strL,"w");
            UpdateCourbure(blockId);
            fprintf(pfile, "%1.15f %1.15f %1.15f\n", runtime, Elements[blockId].Xrelativ, El
            for (int n=nbSlice; n<nbEle; n++) {
                fprintf(pfile, "%1.15f %1.15f %1.15f\n",Xpos[n],Ypos[n],Elements[blockId].KC
            }
        }else if (Elements[blockId].BCkind<1) {
            sprintf(strL,"%s/bnd%ipos.dat", workdir.c_str(), blockId);
            pfile = fopen(strL,"w");
            for (int j=0; j<nbEle; j++) {
                fprintf(pfile, "%1.15f %1.15f \n",Xpos[j],Ypos[j]);
```

```
            }

        } else {
            sprintf(strL,"%s/drop%its%d.dat",workdir.c_str(), blockId, nbSlice);
            pfile = fopen(strL,"w");
            UpdateCourbure(blockId);
            fprintf(pfile, "%1.15f %1.15f %1.15f\n", runtime, Elements[blockId].Xrelativ, El
            for (int n=0; n<nbEle; n++) {
                fprintf(pfile, "%1.15f %1.15f %1.15f\n",Xpos[n],Ypos[n],Elements[blockId].KO
            }
        }
        fclose(pfile);
        cout <<"Position "<<nbSlice<<" written for "<<blockId<<" object"<<" at time "<<runti
    }
}

void BndBlock::LogAreaPos(int logId, int blockId){
    char logurl[120];
    FILE* logfile;
    double A0, Xp, Yp;
    A0 = getCoM(blockId, Xp, Yp);
    sprintf(logurl,"%s/log%i.txt", workdir.c_str(),logId);
    logfile = fopen(logurl, "a");
    fprintf(logfile, "%2.9f %2.9f %2.9f\n", A0, Xp, Yp);
    fclose(logfile);
}

void BndBlock::LogTimeStep(int logId){
    char logurl[120];
    FILE* logfile;
    sprintf(logurl,"%s/log%i.txt", workdir.c_str(),logId);
    logfile = fopen(logurl, "a");
    fprintf(logfile, "%i %f ", runstep, runtime);
    fclose(logfile);
}

double BndBlock::LogCustom(int logid, int modusid){
```

To write a log file with customized data. Delete or add routines as you like.

```
    char logurl[120];
    FILE* logfile;
    double xcm, ycm, perimeter;
    int imax;
```

```
switch(modusid){
    case 0:
        break;
    case 1:
    {
        //  FOR ARTULAM
        double perimeter = 0;
        for(int i=0; i<Elements[1].PanelSize[0];i++){
            perimeter += sqrt( pow(Elements[1].XId[0][i+1]-Elements[1].XId[0][i] ,2)+pow

        }

        sprintf(logurl,"%s/log%i.txt", workdir.c_str(),logid);
        logfile = fopen(logurl, "a");
        getCoM(1, xcm, ycm);
        someWay += xcm;
        fprintf(logfile, "%f %f %f %f\n", xcm, ycm, perimeter, runtime);
        fclose(logfile);
        return xcm;
        break;
    }
    case 2:
    {
        // FOR CORDERO STRETCH
        perimeter = 0;
        double xmax = -10.0;
        double xmin = 10.0;
        double ymax = -10.0;
        double ymin = 10.0;

        for(int i=0; i<Elements[1].PanelSize[0];i++){
            perimeter += sqrt( pow(Elements[1].XId[0][i+1]-Elements[1].XId[0][i] ,2)+pow
            xmax = max(xmax, Elements[1].XId[0][i]);
            xmin = min(xmin, Elements[1].XId[0][i]);
            ymax = max(ymax, Elements[1].YId[0][i]);
            ymin = min(ymin, Elements[1].YId[0][i]);

        }

        char logurl[120];
        FILE* logfile;
        sprintf(logurl,"%s/log%i.txt", workdir.c_str(),logid);
        logfile = fopen(logurl, "a");
        getCoM(1, xcm, ycm);
        someWay += xcm;
        fprintf(logfile, "%f %f %f %f %f %f\n", xcm, ycm, xmax-xmin, ymax-ymin, perimete
        fclose(logfile);
        return xcm;
```

```
        break;
    }
    case 3:
    {
        //get extreme points for relax study
        double ynord, ysud, xest, xouest, kapnord, kapsud, kapest, kapouest;
        int blockid = ifblock;  // Droplet Element
        sprintf(logurl,"%s/log%i.txt", workdir.c_str(),logid);
        logfile = fopen(logurl, "a");
        double nordmin, sudmin, estmin, ouestmin;
        int imin, jmax, jmin;
        double *XA = Elements[blockid].XId[0];
        double *YA = Elements[blockid].YId[0];

        nordmin = 10;
        sudmin = 10;
        estmin =10;
        ouestmin = 10;

        for (int j=0; j<Elements[blockid].PanelSize[0]; j++) {
            if ((YA[j]>0)&&(fabs(XA[j])<nordmin)) {
                nordmin = fabs(XA[j]);
                jmax = j;
            }
            if ((YA[j]<0)&&(fabs(XA[j])<sudmin)) {
                sudmin = fabs(XA[j]);
                jmin = j;
            }

            if ((XA[j]>0)&&(fabs(YA[j])<estmin)) {
                estmin = fabs(YA[j]);
                imax = j;
            }
            if ((XA[j]<0)&&(fabs(YA[j])<ouestmin)) {
                ouestmin = fabs(YA[j]);
                imin = j;
            }
        }

        UpdateCourbure(blockid);
        kapnord = Elements[blockid].KC[jmax];
        kapsud = Elements[blockid].KC[jmin];
        kapest = Elements[blockid].KC[imax];
        kapouest = Elements[blockid].KC[imin];

        ynord = YA[jmax];
        ysud = YA[jmin];
        xest = XA[imax];
```

```
            xouest = XA[imin];


            fprintf(logfile, "%f %f %f %f %f %f %f %f %f\n", runtime, kapnord, kapsud, kapes
            fclose(logfile);
            return kapest;
            break;
        }
    }
    return 0;
}


void BndBlock::DisableWrite(){
    writeEnabled = 0;
}
```

Dedicated to determine basic droplet statistics

```
double BndBlock::getArea(int blockId){
```

Integration of the droplet Area

```
    double xgs, ygs, dx, dy, r, lnx, lny;
    int nbElms = Elements[blockId].getFullSize();
    double A = 0.0;
    double *Xc = Elements[blockId].XId[0];
    double *Yc = Elements[blockId].YId[0];

    if (Elements[blockId].BCkind!=3)
    {
        WrapInterface(blockId);
        for(int n=0; n<nbElms; n++){
            xgs = 0.5*(Xc[n+1]+Xc[n]);
            ygs = 0.5*(Yc[n+1]+Yc[n]);
            dx = Xc[n+1]-Xc[n];
            dy = Yc[n+1]-Yc[n];
            r = dx*dx+dy*dy;
            r = sqrt(r);
            lnx = dy/r;
            lny = -dx/r;
```

```
            A += r*(xgs*lnx+ygs*lny);
        }
        A = A*0.5;    // because div(x) = 2 and GW6 integrates over 2 length

    }else{

        for(int n=-1; n<=nbElms; n++){
            xgs = 0.5*(Xc[n+1]+Xc[n]);
            ygs = 0.5*(Yc[n+1]+Yc[n]);
            dx = Xc[n+1]-Xc[n];
            dy = Yc[n+1]-Yc[n];
            r = dx*dx+dy*dy;
            r = sqrt(r);
            lnx = dy/r;
            lny = -dx/r;
            A += (ygs*lny+xgs*lnx)*r;
        }
        A = A*0.5;
    }

    return A;
}

double BndBlock::getCoM(int blockId, double& Xcm, double& Ycm){
```

Integration of center of area

```
    double xgs, ygs, dx, dy, r, lnx, lny;
    int nbElms;

    double* Xc = Elements[blockId].XId[0];
    double* Yc = Elements[blockId].YId[0];
    double A = 0.0;
    nbElms = Elements[blockId].getFullSize();

    A = 0;
    Xcm = 0;
    Ycm = 0;
    for(int n=0; n<nbElms; n++){
        xgs = 0.5*(Xc[n+1]+Xc[n]);
        ygs = 0.5*(Yc[n+1]+Yc[n]);
        dx = Xc[n+1]-Xc[n];
        dy = Yc[n+1]-Yc[n];
        r = dx*dx+dy*dy;
        r = sqrt(r);
```

```
        lnx = dy/r;
        lny = -dx/r;

        A += ygs*lny*r;
```

```
    A -= xgs*lnx*r;
```

```
        Xcm += r*xgs*ygs*lny;
        Ycm += r*0.5*ygs*ygs*lny;
    }
    Xcm = Xcm/A;
    Ycm = Ycm/A;

    return A;
}

double BndBlock::getHiMo(int blockId, double& Xcm, double& Ycm, double& Mcm){
```

Integration of center of area

```
    double xgs, ygs, dx, dy, x1, y1, r, lnx, lny;
    int nbElms;
    double* Xc = Elements[blockId].XId[0];
    double* Yc = Elements[blockId].YId[0];
    double A = 0.0;
    Xcm = 0.0;
    Ycm = 0.0;
    Mcm = 0.0;

    nbElms = Elements[blockId].getFullSize();
    for(int n=0; n<nbElms; n++){
        xgs = 0.5*(Xc[n+1]+Xc[n]);
        ygs = 0.5*(Yc[n+1]+Yc[n]);
        dx = Xc[n+1]-Xc[n];
        dy = Yc[n+1]-Yc[n];
        r = dx*dx+dy*dy;
        r = sqrt(r);
        lnx = dy/r;
        lny = -dx/r;

        for (int i=0; i<6; i++) {
```

```
        x1 = xgs+dx*IGP[i]*0.5;
        y1 = ygs+dy*IGP[i]*0.5;
        A += r*(x1*lnx+y1*lny)*0.5*IGW[i];
        Xcm += r*(y1*x1*lny)*0.5*IGW[i];
        Ycm += r*(y1*x1*lnx)*0.5*IGW[i];

    //    Mcm += r*(x1*y1*y1*lnx+x1*x1*y1*lny);
        }
    }

    Xcm = 2.0*Xcm/A;
    Ycm = 2.0*Ycm/A;
    A = -A/2.0;

    for(int n=0; n<nbElms; n++){
        xgs = 0.5*(Xc[n+1]+Xc[n])-Xcm;
        ygs = 0.5*(Yc[n+1]+Yc[n])-Ycm;
        dx = Xc[n+1]-Xc[n];
        dy = Yc[n+1]-Yc[n];
        r = dx*dx+dy*dy;
        r = sqrt(r);
        lnx = dy/r;
        lny = -dx/r;

        for (int i=0; i<6; i++) {
            x1 = xgs+dx*IGP[i]*0.5;
            y1 = ygs+dy*IGP[i]*0.5;
            Mcm += r*(x1*y1*y1*lnx+x1*x1*y1*lny)*IGW[i]*0.5;
       }
    }
    Mcm = -Mcm;
    return A;
}

void BndBlock::LoadNOrder(int blockId){
```

Compute 5th order polynomial of the droplet interfaces No longer functional because no memory assigned to polynomial coefficients

```
    int order = 5;
    double *FaceX;
    double *FaceY;
    int size;
    double *AR = new double[order*order];
    double *rhsx = new double[order];
```

```
    double *rhsy = new double[order];
    double dx,dy,rp;
    int ip,rs, frac;

    FaceX = Elements[blockId].XId[0];
    FaceY = Elements[blockId].YId[0];
    size = Elements[blockId].getFullSize();

    for (int k = 0; k<size; k++) {
        frac = (order+0.5)/2;
        ip = 0;
        rs = -1.0;
        for (int i = 0; i<order; i++) {
            if(i==frac){
                ip++;
                rs = 1;
            }
            dx = FaceX[k+ip-frac]-FaceX[k];
            dy = FaceY[k+ip-frac]-FaceY[k];
            rp = rs*sqrt(dx*dx+dy*dy);
            rhsx[i] = dx;
            rhsy[i] = dy;
            for (int j = 0; j<order; j++) {
                AR[i+j*order] = pow(rp,j+1);
            }
            ip++;
        }
        dgsev2rhs(AR, rhsx, rhsy, order);
    }
}
```

Dedicated to Remeshing

```
void BndBlock::SeedDrop(){
    if (rlim>0.0) {
        for (int j=ifblock; j<nbBlocks; j++) {
            SeedEquiDist(j, rlim);
        }
    }
}

void BndBlock::RemeshOn(double targetsize, int modus){
    rlim = targetsize;
    SeedOption = modus;
```

```
}

void BndBlock::SeedEquiDist(int blockId, double limp){
```

Redistributes the droplet elements uniformly

```
    int order = 3;
    double *XA, *XB;
    double *YA, *YB;
    double* AR = new double[order*order];
    double *ace = new double[order];
    double *bce = new double[order];
    double rd, dx, dy, newx, newy, rmin, rnorm,rp;
    int jump, size, ReDist;
    int decal = 0;
    double rmax;
    double dsremain = 0;
    int nremain=0;
    double difadap=0;

    int ip,rs, frac;

    XB = Elements[blockId].XId[0];
    YB = Elements[blockId].YId[0];
    XA = &Elements[blockId].IntfX[5];
    YA = &Elements[blockId].IntfY[5];

    jump = 0;
    size = Elements[blockId].getFullSize();
    ReDist = 0;

    rmax = remesh_threshold*limp;
    rmin = limp/remesh_threshold;

    dx = XB[-1]-XB[0];
    dy = YB[-1]-YB[0];
    rd = sqrt(dx*dx+dy*dy);

    if((rd>rmax)||(rd<rmin)){
        ReDist = 1;
    }

    int kstart = 1;
    if(Elements[blockId].BCkind==3){
```

```
        jump--;
        kstart=0;
    }

    for (int k=kstart; k<=size; k++) { // k should be 1
        dx = XB[k]-XB[k-1];
        dy = YB[k]-YB[k-1];
        rd = sqrt(dx*dx+dy*dy);

        if((rd>rmax)||(rd<rmin)){
            ReDist = 1;
            break;
        }
    }

    if(ReDist){
        Stock(blockId);

//      LoadNOrder(blockId);
        //Copy Point
        XB[decal] = XA[kstart-1];
        YB[decal] = YA[kstart-1];

        rnorm = limp;
        for (int k=kstart; k<=(size); k++) {  // k should be 1
            dx = XA[k]-XA[k-1];
            dy = YA[k]-YA[k-1];
            rd = sqrt(dx*dx+dy*dy);
```

This piece of code smoothes over the last elements the distance so no reconnection no size contrast exists

```
            if (k>=size-10) {
                dsremain = limp-rnorm;
                for (int j=k; j<=size; j++) {
                    dx = XA[k]-XA[k-1];
                    dy = YA[k]-YA[k-1];
                    dsremain += sqrt(dx*dx+dy*dy);
                }
                nremain = int(round(dsremain/limp));
                difadap = (dsremain/(nremain*limp));
                rnorm += (difadap-1.0)*limp;
                limp = limp*difadap;
            }
```

```
        {
            frac = int((order+0.5)/2.0);
            ip = 0;
            rs = -1.0;
            for (int i = 0; i<order; i++) {
                if(i==frac){
                    ip++;
                    rs = 1;
                }
                dx = XA[k+ip-frac-1]-XA[k-1];
                dy = YA[k+ip-frac-1]-YA[k-1];
                rp = rs*sqrt(dx*dx+dy*dy);
                ace[i] = dx;
                bce[i] = dy;
                for (int j = 0; j<order; j++) {
                    AR[i+j*order] = pow(rp,j+1);
                }
                ip++;
            }
        }
        dgsev2rhs(AR, ace, bce, order);


        while (rnorm<rd){
            newx = XA[k-1];
            newy = YA[k-1];
            for (int j=0; j<order; j++) {
                newx += ace[j]*pow(rnorm, j+1);
                newy += bce[j]*pow(rnorm, j+1);
            }
            jump++;
            XB[jump+decal] = newx;
            YB[jump+decal] = newy;
            rnorm += limp;

        }
        rnorm += -rd;
    }

    if (Elements[blockId].BCkind==3) {
        rd = sqrt(pow(XB[jump]-XA[size],2) + pow(YB[jump]-YA[size],2));
    } else {
        rd = sqrt(pow(XB[jump+decal]-XB[decal],2) + pow(YB[jump+decal]-YB[decal],2));
    }


    if(Elements[blockId].BCkind!=3){
        jump += (rd>(0.75*limp));
```

```
            for (int k=0; k<decal; k++) {
                XB[k] = XB[jump+k];
                YB[k] = YB[jump+k];
            }
        } else {
            jump += (rd>(0.5*limp));

            XB[-1] = XA[-1];
            YB[-1] = YA[-1];

            XB[jump] = XA[size];
            YB[jump] = YA[size];
        }

        Elements[blockId].PanelSize[0] = jump;
        std::cout<<"\br";



    }

    allSize[ifid+blockId-ifblock] = Elements[blockId].PanelSize[0];

    WrapInterface(blockId);
    Stock(blockId);
}
```

void BndBlock::SeedVarDist(int blockId, double rliming){

Ubiquitous subroutines

```
void BndBlock::allWrite(){
    for (int k=0; k<nbBlocks; k++) {
        if((Elements[k].BCkind>0)||(runstep==0)){
            WritePos(k, runstep);
        }
    }

}

void BndBlock::allStock(){
    for (int k=0; k<nbBlocks; k++) {
        if(Elements[k].BCkind>0){
            Stock(k);
```

```
        }
    }
}

void BndBlock::allEvolve(double weightOldPos, double weightNewPos, double deltaT, double *U
    int index = 0;

    /*   for (int k=0; k<ifblock; k++) {
     index += 2*Elements[k].getFullSize();
     }
     */
    for (int k=ifblock; k<nbBlocks; k++) {
        Evolve(k, weightOldPos, weightNewPos, deltaT, &Uvec[index]);
        index += 2*Elements[k].getFullSize();
    }
}

void BndBlock::WrapAll(){
    for (int k=dynblock; k<nbBlocks; k++) {
        WrapInterface(k);
    }
}

int BndBlock::getFixSize(){
    int size=0;
    for (int k=0; k<nbBlocks; k++) {
        if(Elements[k].BCkind>0){
            break;
        }
        size += Elements[k].getFullSize();
    }
    return size;
}

int BndBlock::getAllSize(){
    int size=0;
    for (int k=0; k<nbBlocks; k++) {
        size += 2*Elements[k].getFullSize();
        if (Elements[k].BCkind==1) {
            size += 3;
        }
    }
    return size;
}
```

Ulambator BoundaryElement class (`bndelement.cpp`) v0.3

This is the memory class that saves information of a boundary block

```
#include "bndelement.h"
#include <stdio.h>
#include <stdlib.h>


FlowFace::FlowFace(){
};

void FlowFace::Init(int Panels, int Size, int ofKind){
```

Constructor for the boundary interfaces. Droplets and Walls ofKind determines the kind of boundary: 0-walls, 1-fibers, 2-drops, 3-attached interfaces

```
    nbPanRes = Panels;
    nbPanels  = 0;
    Xrelativ = 0.0;
    Yrelativ = 0.0;
    XE = (double*) calloc (Size+10,sizeof(double));
    YE = (double*) calloc (Size+10,sizeof(double));
    IntfX = (double*) calloc (Size+10,sizeof(double));
    IntfY = (double*) calloc (Size+10,sizeof(double));
    BCtype = (int*) calloc (Panels,sizeof(int));
    BCvalueX = (double*) calloc (Panels,sizeof(double));
    BCvalueY = (double*) calloc (Panels,sizeof(double));

    XId = (double**) calloc (Panels,sizeof(double*));
    YId = (double**) calloc (Panels,sizeof(double*));
    PanelSize = (int*) calloc (Panels,sizeof(int));
    BCkind = ofKind;
    initialArea = 0;

    if(BCkind>0){
        Rdist = (double*) calloc (Size+2,sizeof(double));   // distance from another droplet
        Cdist = (double*) calloc (Size+2,sizeof(double));   // distance from self inflicted
        Wdist = (double*) calloc (Size+2,sizeof(double));   // distance from closest wall
        KC = (double*) calloc (Size,sizeof(double));       // curvature
        for (int k=-1; k<(Size+1); k++) {
            Wdist[k] = 1e12;
            Rdist[k] = 1e12;
        }
```

```
    }
}

void FlowFace::Kill(){
    free(XE);
    free(YE);
    free(IntfX);
    free(IntfY);
    free(Rdist);
    free(Cdist);
    free(Wdist);
    free(KC);

    nbPanels = 0;
    PanelSize[0] = 0;
}

int FlowFace::getFullSize(){
    int index = 0;
    for (int k = 0; k<nbPanels; k++) {
        index += PanelSize[k];
    }
    return index;
}
```

Created by Mathias Nagel on 19.03.10.

Further classes

The ULaMBAtoR package has two more classes: `matblc1.cpp` and `combase.cpp`. `Combase` contains the basic computational routines for matrix algebra. It actually interfaces mostly the LAPACK routines that do the intensive work. The variant `combase_ cote.cpp` is used only for compiler testing and is stripped of all LAPACK functionalities in order to test the compilation for everything that is not related to LAPACK. LAPACK is for now the only external library that is used and external libraries may cause problem, which we try to isolate here.

The `Matblc1` inheritates from `matrixblock.cpp` and contains functions that fill the matrix and that are written for specific boundary conditions. Originally it was thought to be modulable, `c1` (or C_1) would stand for peice-wise linear continuous Ansatz functions, and one functions of higher degree would have been possible. The philisophy that is maintained now is that the interface is discretized by piece-wise linear functions and the external boundaries by piece-wise constant functions. So `Matblc1` is appart for historical reasons but this is maintained since it helps to avoid one huge `matrixblock.cpp` class and sorts out routines that are not to be changed.