Cite as: Vasudevan, M.: Implementation of partially slip boundary conditions. In Proceedings of CFD with OpenSource Software, 2017, Edited by Nilsson. H., `http://dx.doi.org/10.17196/OS_CFD#YEAR_2017`

CFD WITH OPENSOURCE SOFTWARE

A COURSE AT CHALMERS UNIVERSITY OF TECHNOLOGY
TAUGHT BY HÅKAN NILSSON

# Implementation of partially slip boundary conditions

Developed for OpenFOAM-2.2.x

**Author:**
MADHAVAN VASUDEVAN
madvas@student.chalmers.se

**Peer reviewed by:**
MOHAMMAD ARABNEJAD
YERU SHANG

Co-supervised by: Célio Fernandes and J. Miguel Nobrega, Institute for Polymers and Composites, Minho University, Portugal

January 8, 2018

# Preface

The codes for implementation of partially slip boundary conditions for explicit formulation of Navier slip model were already developed in University of Minho, Portugal. The case files were also developed to appropriately test the codes developed. These were contributed for further work in this project. In the current project work, these codes have been adopted as the fundamental codes based on which Hatzikiriakos and Asymtotic models for explicit formulation and Navier, hatzikiriakos and Asymtotic models for semi-implicit formulation has been developed. Since the project work is aimed at providing a tutorial for the readers to develop a boundary and setup a case of their of own, a new case file replicating the given case files has been developed from scratch from an existing tutorial available in OpenFOAM installation.

# Learning Outcomes

The project work is aimed to achieve the four requirements, which are: How to use it, The theory, How it is implemented and How to modify it. The learning outcomes have been categorized under these objectives on priority.

The reader will learn:

## How to use it

- How to use Explicitly defined boundary condition for slip flows by Navier law.
- How to set up cases simple channel flow cases with slip flows.

## The theory

- The theory of slip flows.
- Concept of explicit, implicit and semi-implicit formulation of boundary conditions for slip flows.

## How it is implemented

- How to develop a boundary condition that is applied to individual faces of the patch
- How to change the member functions and data members to generate a different boundary with similar formulation (explicit formulation/ semi-implicit formulation).

## How to modify it

- How to develop a semi-implicit slip boundary condition
- How to develop boundary conditions with different governing equations

# Contents

# Chapter 1

# Partially Slip Boundary conditions

## 1.1 Introduction

The phenomena of wall slip velocity started becoming pronounced for flows with high Knudsen Number, i.e the flows with the mean free path very comparable to the length scale of the flow. These are typically the cases in flows through micro-nano channels and in rarefied flow situations assuming fluid medium as a continuum becomes non-physical. The phenomena is also very common and of immense significance in many other industrial applications like in the polymer extrusion industry. The prediction of accurate wall slip velocity becomes a necessity to define the characteristics of the overall process parameters like the throughput and quality of the final product. There have been several attempts to define these velocities analytically through both simple assumptions and including complex rheological models. The implementation of these analytical solutions in a Computational Fluid Dynamics (CFD) codes has been of prime interests and this work is one of such attempts to calculate wall slip velocity in OpenFOAM through analytically proven techniques.

First section of the report is documented with the theory involved in the prediction of wall slip velocities through different models and formulations. It is then followed by a section for explaining the framework of the code for explicit formulation of Navier-slip model (will be discussed in the upcoming section; this is already implemented in OpenFOAM). This section section will be followed by modifications to reflect other transcendental equations [1] that govern slip flows and other formulations which could reflect the same transcendental equation differently from a numerical standpoint. The final is section is concentrated on discussion of results by testing the established codes by setting up appropriate cases and solving them.

## 1.2 Theory Involved in the Problem

### 1.2.1 Explicit and Implicit Formulations in CFD

Consider a rate of change of a property $\phi$ with respect to time t. The derivative or the slope of variation could be defined as

$$\frac{\mathrm{d}\phi}{\mathrm{d}t} = \lim_{\Delta t \to 0} \frac{(\phi)^{n+1} - (\phi)^n}{\Delta t} \tag{1.1}$$

where (n+1) and (n) are the subsequent time instances which are separated by time $\Delta$t. It is of our interest here to find the property $\phi$ at the later time stage and we could formulate them in the following ways.

$$(\phi)^{n+1} = (\phi)^n + \Delta t (\frac{\mathrm{d}\phi}{\mathrm{d}t})^n \tag{1.2}$$

[1]The Navier, Hatzikiriakos and the Asymtotic slip governing laws for slip

$$(\phi)^{n+1} = (\phi)^n + \Delta t(\frac{\mathrm{d}\phi}{\mathrm{d}t})^{n+1} \tag{1.3}$$

Equation 1.2 shows the explicit formulation of calculations and 1.2 shows the implicit formulation of calculations. Both these methods are very widely used in CFD simulations based on the application and the nature of the problem being solved. The methods have both positive and negative effects namely,

Explicit methods are very stable but needs very small intervals. Relaxation of solutions becomes a necessity in this method to improve stability with larger temporal or spatial intervals

Implicit methods are not as stable as the explicit method but could accommodate higher interval ranges and relaxation could be avoided in this method.

### 1.2.2 Governing laws of slip flows and formulations

The governing equations solved vary based on the complexity of the problem. The governing equations could directly take values for the viscosity or it could solve a complex rheological model as mentioned before. In the current study, the solver used to compile the model is simpleFoam. Before describing the solver itself, it is necessary to understand the functionality of the boundary condition to be implemented so that the sequence of calculations are streamlined in report.

As discussed in the previous section, the wall slip velocity could be estimated by both implicit and explicit formulations. There is also an alternative which is novel by its idea and that is the semi-implicit formulation. The semi-implicit formulation is coupled with the simple method (Semi Implicit Method Pressure Linked Equations) solver to form a Simple Slip semi-implicit Scheme.

The main laws of wall slip velocity calculation in this process include the linear Navier slip law, the non-linear Navier slip law, the Hatzikiriakos law and the Asymtotic law. It has been found that the implicit method has been successful with the prediction of wall slip velocities for only the linear model. The explanation of the implementation of the Navier laws (both linear and non-linear) in the explicit formulation and extension of the same to Hatzikiriakos and Asymtotic laws are the first objectives of the project. The project is then focused towards formulating a semi implicit method where all the mentioned non linear models are implemented by it. The exactness of the semi implicit formulation is tested with the explicitly formulated results.

The Navier wall slip velocity is estimated through the relation

$$u_{ws} = k_{nl}(\mu(\dot{\gamma})\frac{\mathrm{d}u}{\mathrm{d}y})^m \tag{1.4}$$

where $k_{nl}$ is the slip factor and m is the order of the non-linearity by which the wall slip is estimated in the given problem. In the Navier formulation if the value of m is 1, it becomes a linear Navier slip prediction and non-linear if not for any other value .

The Hatzikiriakos wall slip velocity is estimated through the relation

$$u_{ws} = k_{h1}sinh(-k_{h2}\mu(\dot{\gamma})\frac{\mathrm{d}u}{\mathrm{d}y}) \tag{1.5}$$

where $k_{h1}$ and $k_{h2}$ are the Hatzikiriakos constants (model parameters)

The Asymtotic wall slip velocity is estimated through the relation

$$u_{ws} = k_{a1}log(1 - k_{a2}\mu(\dot{\gamma})\frac{\mathrm{d}u}{\mathrm{d}y}) \tag{1.6}$$

where $k_{a1}$ and $k_{a2}$ are the Asymtotic constants (model parameters)

The next important argument is how the gradient in the equations 1.4, 1.5 and 1.6 are formulated in the CFD codes. In all the three equations, the gradients are defined as the same way in CFD as

$$\frac{\mathrm{d}u}{\mathrm{d}y} = \frac{u_{ws} - u_p}{\Delta y_f} \tag{1.7}$$

The definition is valid for the orthogonal meshes. The $u_{ws}$ is the wall slip velocity, $u_p$ is the velocity of the cell center of the cells that attached to the boundary patch (boundary to which the boundary condition is applied) and $\Delta y_f$ is the half cell length, i.e the distance between the face of the cell (part of the boundary patch) and the cell center.

Now the implicit and explicit formulations are deployed in the definition gradient in equation 1.7.

The explicit formulation of the gradient is defined as in equation 1.8. The i in the equation 1.8 defines that the value is obtained from the previous iteration.

$$\frac{\mathrm{d}u}{\mathrm{d}y} = \frac{u_{ws}^{i} - u_{p}^{i}}{\Delta y_f} \tag{1.8}$$

The implicit formulation of the gradient is defined as in equation 1.9. The i+1 in the equation 1.9 defines that the value is obtained from the current iteration.

$$\frac{\mathrm{d}u}{\mathrm{d}y} = \frac{u_{ws}^{i+1} - u_{p}^{i+1}}{\Delta y_f} \tag{1.9}$$

The semi-implicit approach takes the value of the wall slip velocity from the current iteration and the cell center velocity from the previous iteration and hence it could be formulated as

$$\frac{\mathrm{d}u}{\mathrm{d}y} = \frac{u_{ws}^{i+1} - u_{p}^{i}}{\Delta y_f} \tag{1.10}$$

The explicit method as described earlier, relaxes the solution obtained to maintain convergence for the solution. The theory behind implementation of semi-implicit method is of our interest in this report. As it can be seen from equation 1.10 and equations 1.4, 1.5 or 1.6, the wall slip velocity in the current iteration appears on both the sides of the equation and it cannot be solved analytically. The necessity to deploy a numerical method becomes evident. Bisection method is deployed to calculate the root of the equation or solution (wall slip velocity in current iteration).

### Bisection Method

Bisection method is a numerical method to calculate the root of the equation when the function is implicit. Initially to start the process of bisection method, a range is ideally chosen and tested if the range holds the root of the equation. Consider a function f(x), and if the range chosen are a and b, it could be said that the function has a root in the range [a,b] if the product of f(a) and f(b) is negative. This typically means that the function crosses through the x axis (meaning it will be zero at that point) and that point becomes the root of the equation. So after verifying that result happens to be in the specified range, a guess value is ideally chosen as $c = 0.5(a + b)$ and f(c) is calculated. If the f(c) is positive, it means that the solution is in the range [a,c] and [b,c] if the f(c) value is negative. This process is performed in an iteration until a certain value of tolerance is reached. Note that the value of f(c) is residual that has to be minimized in the process of bisection method. The convergence of solution can be verified by both checking the difference in the values of root achieved in the subsequent iterations and also by the minimizing f(c) as it moves towards zero.

### Bisection Method in wall slip predictions - Semi Implicit Formulations

Equation 1.4, 1.5 or 1.6 is the transcendental equation for which the root $u_{ws}$ is to be obtained. Through analytical predictions, different ranges have been described for different laws. These values are vectors and the bisection method is applied along the streamline direction of the flow.

Therefore the steps involved to calculate the root of the transcendental equations could be summarized as the following.

- initialize two guess velocity vectors ( vector $\overrightarrow{a}$ and vector $\overrightarrow{b}$ ) to form the range according to the slip law employed and calculate the guess velocity vector as 0.5 ($\overrightarrow{a}$ + $\overrightarrow{b}$).

- substitute this guess velocity vector in place of $u_{ws}$ and check the streamline component of "guess vector (current root) - calculated wall slip value (estimated root) "

- Adjust the vectors (vector $\overrightarrow{a}$ and vector $\overrightarrow{b}$) with the guess vector based on the sign of the streamline component of residual from the transcendental equation.

- Continue the iterative process until magnitude residual meets a threshold value. Residual is defined as ($\overrightarrow{b}$ - $\overrightarrow{a}$)$/2^{Numberofiterations}$. For a magnitude value, the streamline component is again considered in this case for $\overrightarrow{a}$ and $\overrightarrow{b}$ .

### 1.2.3 Execution of the boundary conditions

The aim of the section is to illustrate the working of the boundary condition in connection with simpleFoam, the steady state OpenFOAM solver which works on SIMPLE algorithm for pressure-velocity couplings.

To understand the operational structure in OpenFOAM, it is very important to understand the directory organization. These set of files are compiled and added to the hash tables of the OpenFOAM library (contains many default and other custom boundary conditions) as a custom or user-defined library file. The second task would be to explain the constructors involved and the definition of the same. The third task would be to explain the member functions of the class definition. It will be of interest to note that in the subsequent tasks the directory organization remains the same throughout- The definition of member data declaration remains the same for a particular slip law by which the wall slip velocity would be computed ( i.e the same constructors initialization and definition are used for example, non-linear Navier slip law irrespective of whether it is formulated with explicit or semi-implicit formulation.)

The following are explanations about non-linear Navier slip law with explicit formulation

#### Non-linear Navier slip law

The following shows the directory structure of the Navier slip boundary condition source codes.

```
   navier
   |___ myNLSlipGenNewtonianRelaxationV2
       |___ Make
       |   |___ files
       |   |___options
       |___ nonLinNavSlipGenNewtonianRelaxationFvPatchField.C
       |___ nonLinNavSlipGenNewtonianRelaxationFvPatchField.H

3 directories, 4 files
```

The files file in the Make directory shows the name by which the boundary condition will be added to the hash table as a custom boundary condition and the options file includes the path to headers and libraries necessary for the compilation of the given boundary condition.

The options file contains the following lines

```
EXE_INC = \
    -I$(LIB_SRC)/triSurface/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/turbulenceModels \
    -I$(LIB_SRC)/turbulenceModels/incompressible/turbulenceModel \
    -I$(LIB_SRC)/turbulenceModels/incompressible/RAS/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/transportModels/incompressible/singlePhaseTransportModel \
    -I$(LIB_SRC)/fvOptions/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude

LIB_LIBS = \
    -lOpenFOAM \
    -ltriSurface \
    -lmeshTools \
    -lfiniteVolume \
    -lincompressibleTransportModels \
    -lincompressibleTurbulenceModel \
    -lincompressibleRASModels \
    -lincompressibleLESModels \
    -lfvOptions \
    -lsampling
```

The files file in the Make directory contains the following lines.

```
FOAM_USER_LIBBIN
```

is the user defined location where the library file will get updated. The changes will be made in the original OpenFOAM installations if it is mentioned as

```
FOAM_LIBBIN
```

It is recommended for modifications to be made in

```
FOAM_USER_LIBBIN.
```

```
nonLinNavSlipGenNewtonianRelaxationFvPatchField.C

LIB = $(FOAM_USER_LIBBIN)/libmynonLinNavSlipGenNewtonianRelaxation
```

### Constructors initialization

As illustrated in the directory structure of the source codes for boundary condition, one should have a header file and the main file in association with the make folder which consists of the files and the options files. The purpose of the header file is declare the necessary member data for the implementation of the necessary boundary condition class. Several constructors have been declared and defined (definition in the main file) which provides the flexibility in construction of the boundary condition class. The header file also includes all the necessary classes that the main files would make use of, for the implementation of the boundary condition. The file illustrated in the section shows the declaration of density, relaxationFactor, slipFactor and n as the necessary member data for the problem in hand. The explanation to the code is followed after the excerpt from the header file, nonLinNavSlipGenNewtonianRelaxationFvPatchField.H.

```
#ifndef nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField_H
#define nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField_H

#include "fvPatchFields.H"
#include "fixedValueFvPatchFields.H"
#include "transformFvPatchField.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{
```

```

/*---------------------------------------------------------------------------*\
            Class radiationConvectionFvPatchField Declaration
\*---------------------------------------------------------------------------*/

class nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField
:
   public fixedValueFvPatchVectorField
{
   // Private data

       //- fluid density
       scalar rho_;

       //- exponent of non-linear Navier slip condition
       scalar n_;

       //- slip factor
       scalar slipFactor_;

       //- relaxation factor
       scalar relaxationFactor_;

public:

   //- Runtime type information
   TypeName("nonLinNavSlipGenNewtonianRelaxation");


   // Constructors

       //- Construct from patch and internal field
       nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField
       (
           const fvPatch&,
           const DimensionedField<vector, volMesh>&
       );

       //- Construct from patch, internal field and dictionary
       nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField
       (
           const fvPatch&,
           const DimensionedField<vector, volMesh>&,
           const dictionary&
       );

       //- Construct by mapping given nonLinNavSlipGenNewtonianRelaxationFvPatchField
       //  onto a new patch
       nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField
       (
           const nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField&,
           const fvPatch&,
           const DimensionedField<vector, volMesh>&,
           const fvPatchFieldMapper&
       );

       //- Construct as copy
       nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField
       (
           const nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField&
       );

       //- Construct and return a clone
       virtual tmp<fvPatchVectorField> clone() const
       {
           return tmp<fvPatchVectorField>
           (
               new nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField(*this)
           );
       }

       //- Construct as copy setting internal field reference
       nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField
       (
           const nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField&,
           const DimensionedField<vector, volMesh>&
       );

       //- Construct and return a clone setting internal field reference
       virtual tmp<fvPatchVectorField> clone
       (
           const DimensionedField<vector, volMesh>& iF
       ) const
       {
           return tmp<fvPatchVectorField>
           (
               new nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField(*this, iF)
           );
       }


   // Member functions


       //- Return the fluid density
       const scalar& rho() const
       {
           return rho_;
       }

       //- Return valuefraction slipFactor
       const scalar& slipFactor() const
       {
           return slipFactor_;
       }

       //- Return n
       const scalar& n() const
       {
           return n_;
       }

       //- Return relaxationFactor

       const scalar& relaxationFactor() const
       {
           return relaxationFactor_;
       }
```

```
156
157         //- Return non-const acess to the fluid density
158         scalar& rho()
159         {
160            return rho_;
161         }
162
163         //- Return non-const acess to the valuefraction slipFactor
164         scalar& slipFactor()
165         {
166            return slipFactor_;
167         }
168
169         //- Return non-const acess to the n value
170         scalar& n()
171         {
172            return n_;
173         }
174
175         //- Return non-const acess to the relaxationFactor value
176
177         scalar& relaxationFactor()
178         {
179            return relaxationFactor_;
180         }
181
182         //- Update coefficients
183         virtual void updateCoeffs();
184
185         //- Write
186         virtual void write(Ostream&) const;
187 };
```

- line 30: include the fvPatchFields class definitions
- line 31: include the fixedValueFvPatchFields class definitions
- line 32: include the transformFvPatchFields class definitions. This class is especially used to compute the tangential components of the gradients of velocities from all the components included.
- line 43: The nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField class belongs to the base class namespace Foam.
- line 45: The nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField class is publicly derived from the fixedValueFvPatchVectorField class.
- line 49-59: The member data of the class are declared. These are the density, n (constant used for slip velocity calculations), slipFactor (constant used for the calculation of slip velocity) and the relaxationfactor (this is done for the explicit solver as explicit solving requires relaxation). All these member data are declared as private member data. This could be realized with the help of the trailing underscores, which is the OpenFOAM standard.
- line 64: The typeName is defined. This is the name of the class itself. This is the name by which the boundary condition class is identified from the set up case file.
- lines 70-74 The constructor is used to initialize when the boundary condition is set by the boundary patch properties and the internal field.

- lines 77-82 The constructor is used to initialize the member data when the boundary condition is set in 0/U file. The definition of the constructors could be seen in the Main file where the corresponding keyword searched for the particular member data is defined. Here the keywords correspond to the rho, slipFactor, n and relaxationFactor which are assigned values in the in 0/U file. This leads to the benefit of adjusting these flow and numerical parameters at the compilation time. Else, they would have to be altered every-time at the source codes and the boundary condition class needs to be compiled before running the case.
- lines 86-92: The construction through mapping
- lines 95-98: Initialization of the copy constructor is done
- lines 101-120: Initialization through other constructors
- lines 132-180: All the member data are by default declared as private member data of the class. As the member function needs access to the values of the member data and change their value, it is not readily possible when the member data are defined as private members of the class. Therefore the return function provides access for the same.
- line 183: The updateCoeffs() function is declared. This is done to ensure if the values are up-to-date before the next time the boundary condition is called for implementations.
- line 186: The write function is used to write stringed streams and constant values.

In the nonLinNavSlipGenNewtonianRelaxationFvPatchField.H, the initialization of the constructors were seen. In the main file, nonLinNavSlipGenNewtonianRelaxationFvPatchField.C the definition of the several constructors could be seen. The most relevant constructor is the one that constructs from the patch, internal field and the dictionary. As explained in the previous section, here it refers to the values looked up from the 0/U file and that is the way by which the boundary condition class is constructed throughout the project and the following excerpt shows the same. It will also be interesting to check and verify that the boundary condition class works perfectly well even without the initialization of the constructors in the header file and definition of the same in the main file.

```
44
45 // * * * * * * * * * * * * * * * * Constructors * * * * * * * * * * * * * * //
46

76
77
   nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField::nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField
78 (
79    const fvPatch& p,
80    const DimensionedField<vector, volMesh>& iF,
81    const dictionary& dict
82 )
83 :
84    fixedValueFvPatchField<vector>(p, iF),
85    rho_(readScalar(dict.lookup("rho"))),
86    n_(readScalar(dict.lookup("n"))),
87    slipFactor_(readScalar(dict.lookup("slipFactor"))),
88    relaxationFactor_(readScalar(dict.lookup("relaxationFactor")))
89 {
90    if (dict.found("value"))
91    {
92        fvPatchField<vector>::operator=
93        (
94            vectorField("value", dict, p.size())
95        );
96      }
97      else
98      {
99          // Evaluate the wall velocity
100         updateCoeffs();
101     }
102 }
103
104
```

lines 77-102: The definition shows that the class is constructed through the patch, internal field and the dictionary values. The lookup command is used for searching the corresponding keywords n, slipFactorm, relaxationFactor and rho) in the 0/U file of the set up case and if the value is found, the boundary class is evaluated based on the updated value from the U file. As mentioned earlier, this avoids the necessity of compiling the boundary condition class every time these parameters changed.

### Member Functions

The member function is where the actual modification process takes place. The member function of Navier-slip is explained as below. The explanation could lead to the implementation of other laws in the subsequent discussions. It is noted to be noted that the member functions contain two functions; updateCoeffs() and write().

```
132 // * * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * * * //
133
134 void nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField::updateCoeffs()
135 {
136     if (updated())
137     {
138         return;
139     }
140
141     //face normal vector
142     vectorField nHat = this->patch().nf();
143
144     // Du/dn
145     vectorField gradient = this->snGrad();
146
147     // only tangential components
148     gradient = transform(I - sqr(nHat), gradient);
149
150     // gradient Direction (since the pow function doesn't take vectors, we later have
    to multiply the magnitude of the gradient with the gradient direction)
151     vectorField gradientDirection = gradient / (mag(gradient) + SMALL);
152
153     // slip velocity of the last iteration
154     vectorField u_wallslip_lastIteration = (*this);
155
156     const label patchI = patch().index();
157     scalarField nuw = 1e-6*mag(patch().nf());
158
159     if (db().found("turbulenceModel"))
160     {
161         const incompressible::turbulenceModel& turbModel =
162             db().lookupObject<incompressible::turbulenceModel>
163             (
164                 "turbulenceModel"
165             );
166
167        //Info<< "\nTurbulence found\n" << endl;
168        nuw = rho_*turbModel.nu()().boundaryField()[patchI];
169     }
170     else
171     {
172        nuw = mag(patch().nf());
173        //Info<< "\nTurbulence NOT found\n" << endl;
174     }
175     //Info<<"\n nuw_val = "<< nuw <<endl;
176
177     // slip velocity in the current iteration
178     vectorField u_wallslip = -slipFactor_*(Foam::pow(nuw,
    n_))*mag(Foam::pow(mag(gradient), n_))* gradientDirection;
179
180     //Info <<"\n u_wallslip = "<< u_wallslip << endl;
181
182     //Calculate and set u_wallslip
183
    vectorField::operator=(relaxationFactor_*u_wallslip_lastIteration+(1.0-relaxationFactor_)*u_wallslip);
184      Info << u_wallslip[50] << endl;
185      fixedValueFvPatchVectorField::updateCoeffs();
186 }
187
188
189 // Write
190 void nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField::write(Ostream& os) const
191 {
192     fvPatchVectorField::write(os);
193     os.writeKeyword("rho") << rho_ << token::END_STATEMENT << nl;
194     os.writeKeyword("n") << n_ << token::END_STATEMENT << nl;
195     os.writeKeyword("slipFactor") << slipFactor_ << token::END_STATEMENT << nl;
196     os.writeKeyword("relaxationFactor") << relaxationFactor_ << token::END_STATEMENT <<
    nl;
197     writeEntry("value", os);
198 }
199
200
201 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
202
203 makePatchTypeField(fvPatchVectorField,
    nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField);
204
205 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
206
207 } // End namespace Foam
```

- line 134-139: Firstly the code checks if the values are updated, and if so, it returns the updated value to the part of the code from where this member function is accessed through the class definition inside which this member function is defined.
- line 142 : this is a pointer .nf() refers to the normal vector. patch () refers to the boundary patch to which the boundary condition is being applied. Therefore it returns an array of normal vectors to all the faces in the boundary patch.
- line 145: The gradient of the velocity is obtained for the previous iteration using the same pointer 'this' and the function snGrad. The dfinition of snGrad in the fvPatchFields.C is verified and is explicitly checked to give the same value as

```
 vectoField snGrad = ( *this - pif)*this->patch().deltaCoeffs();
```

The equivalence of the formulation in the boundary condition code is illustrated in the semi-implicit formulation where the definition becomes very essential.

- line151: the normalized gradient vector is obtained for the previous iteration.
- line154: the velocity of all the faces of the patch is defined using the pointer this, for the previous iteration
- line 156: every patch in the domain is given an index and the index of the boundary patch in consideration is obtained.
- line 159-178: The dynamic viscosity is computed using the relevant turbulence model. Since the mean flow is very Small in our case, it is ideally chosen as a laminar model (described in 'case set up')
- line178: the value of the wall slip velocity is computed according to the transcendental equation as in equation 1.4.
- line 184: As mentioned earlier, explicit methodologies are highly stable but they could diverge with large intervals. So relaxation of solution is performed for ensuring convergence.
- line 190: The write function is defined to return a constant value as stated for the individual variables; this is particularly useful when the simulations are to be resumed from latest iteration values. The values of the boundary condition are written to a specific file.
- line 193: The value of density, rho is written to a file called os.
- line 194: The value of model parameter n is written to file os.
- line 195: The value of model parameter slipFactor is written to file os.
- line 196: The value of relaxationFactor is written to the file os.
- line 197: The value of parameter calculated in the boundary condition is written to file os.

**Working of SimpleFoam**

No modification is required for the solver. In each iteration, the continuity errors are minimized to have a converged value so that the updated value of internal fields and the boundary fields get updated to the solution. The iterations are performed for sufficiently large number of steps to get a converged solution

### 1.2.4 Implementation of non-linear models by explicit formulation

**Hatzikiriakos explicit formulation**

**Procedure involved to set up the boundary condition**

Follow the prescribed steps to setup a Hatzikiriakos model from an existing Navier slip model.

-
  ```
  tar -xvzf madhavan_files.tar.gz
  ```

  Could be downloaded from the curse homepage

- `cd madhavan_files`

  All the operations in the report happen within this directory (all models/ both the semi-implicit and explicit formulations)

- cp -r navier hatzikiriakos
- cd hatzikiriakos/
- mv myNLSlipGenNewtonianRelaxationV2/ myhatziSlipGenNewtonianRelaxationV2/
- cd myhatziSlipGenNewtonianRelaxationV2/
- OF22x
- wclean
- Make/

The options file remains the same as the non-linear Navier slip explicit case The files file in Hatzikiriakos explicit formulation is written as following.

```
 hatziSlipGenNewtonianRelaxationFvPatchField.C

LIB = $(FOAM_USER_LIBBIN)/libmyhatziSlipGenNewtonianRelaxation
```

- cd ..
- mv nonLinNavSlipGenNewtonianRelaxationFvPatchField.C hatziSlipGenNewtonianRelaxation-FvPatchField.C [2]
- mv nonLinNavSlipGenNewtonianRelaxationFvPatchField.H hatziSlipGenNewtonianRelaxation-FvPatchField.H [3]
- include the new member data $k_{h1}$ and $k_{h2}$ instead of n and slipFactor in both the header and the main file; the resulting final files are made available in the report.
- change nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField to hatziSlipGenNewtoni-anRelaxationFvPatchVectorField using sed command in both the header and the main files; could be noticed in the final file made available in the report.
- Make the change in header file name in the main file include statement; could be noticed in the final file made available in the report.

### Directory Structure

As seen for the Navier slip boundary condition with explicit formulation, a similar directory structure is created for the Hatzikiriakos law as in the following. This is obtained as a result of following the steps mentioned in the previous sub-section.

```
    hatzikiriakos
    |___ myhatziSlipGenNewtonianRelaxationV2
        |___ Make
        |   |___ files
        |   |___options
        |___ hatziSlipGenNewtonianRelaxationFvPatchField.C
        |___ hatziSlipGenNewtonianRelaxationFvPatchField.H

3 directories, 4 files
```

[2]remove the hyphen after Relaxation
[3]remove the hyphen after Relaxation

**Constructors Initialization**

The constructors are initialized in the similar way as for the Navier slip law and member data to be declared are modified. In the Hatzikiriakos model, as could be seen from the transcendental equation, $k_{h1}$ and $k_{h2}$ are declared instead of the data members n and slipFactor. In the header file, hatziSlipGenNewtonianRelaxationFvPatchField.H the typneName has to be changed as well to be identified uniquely in the hash table.

```
#ifndef hatziSlipGenNewtonianRelaxationFvPatchVectorField_H
#define hatziSlipGenNewtonianRelaxationFvPatchVectorField_H

#include "fvPatchFields.H"
#include "fixedValueFvPatchFields.H"
#include "transformFvPatchField.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{

/*---------------------------------------------------------------------------*\
            Class radiationConvectionFvPatchField Declaration
\*---------------------------------------------------------------------------*/

class hatziSlipGenNewtonianRelaxationFvPatchVectorField
:
   public fixedValueFvPatchVectorField
{
   // Private data

       //- fluid density
       scalar rho_;

       //- hatzikiriakos constant 2
       scalar kh2_;

       //- hatzikriakos constant 1
       scalar kh1_;

       //- relaxation factor
       scalar relaxationFactor_;

public:

   //- Runtime type information
   TypeName("hatziSlipGenNewtonianRelaxation");


   // Constructors

       //- Construct from patch and internal field
       hatziSlipGenNewtonianRelaxationFvPatchVectorField
       (
           const fvPatch&,
           const DimensionedField<vector, volMesh>&
       );

       //- Construct from patch, internal field and dictionary
       hatziSlipGenNewtonianRelaxationFvPatchVectorField
```

```
        (
            const fvPatch&,
            const DimensionedField<vector, volMesh>&,
            const dictionary&
        );

        //- Construct by mapping given nonLinNavSlipGenNewtonianRelaxationFvPatchField
        //  onto a new patch
        hatziSlipGenNewtonianRelaxationFvPatchVectorField
        (
            const hatziSlipGenNewtonianRelaxationFvPatchVectorField&,
            const fvPatch&,
            const DimensionedField<vector, volMesh>&,
            const fvPatchFieldMapper&
        );

        //- Construct as copy
        hatziSlipGenNewtonianRelaxationFvPatchVectorField
        (
            const hatziSlipGenNewtonianRelaxationFvPatchVectorField&
        );

        //- Construct and return a clone
        virtual tmp<fvPatchVectorField> clone() const
        {
            return tmp<fvPatchVectorField>
            (
                new hatziSlipGenNewtonianRelaxationFvPatchVectorField(*this)
            );
        }

        //- Construct as copy setting internal field reference
        hatziSlipGenNewtonianRelaxationFvPatchVectorField
        (
            const hatziSlipGenNewtonianRelaxationFvPatchVectorField&,
            const DimensionedField<vector, volMesh>&
        );

        //- Construct and return a clone setting internal field reference
        virtual tmp<fvPatchVectorField> clone
        (
            const DimensionedField<vector, volMesh>& iF
        ) const
        {
            return tmp<fvPatchVectorField>
            (
                new hatziSlipGenNewtonianRelaxationFvPatchVectorField(*this, iF)
            );
        }


    // Member functions


        //- Return the fluid density
        const scalar& rho() const
        {
            return rho_;
        }

        //- Return hatzikiriakos constant1
        const scalar& kh1() const
        {
            return kh1_;
        }

        //- Return hatzikiriakos constant 1
        const scalar& kh2() const
        {
            return kh2_;
        }

        //- Return relaxationFactor

        const scalar& relaxationFactor() const
        {
            return relaxationFactor_;
        }

        //- Return non-const acess to the fluid density
        scalar& rho()
        {
            return rho_;
        }

        //- Return non-constant access to hatzikiriakos constant 1
        scalar& kh1()
        {
            return kh1_;
        }

        //- Return non-const acess to the n value
        scalar& kh2()
        {
            return kh2_;
        }

        //- Return non-const acess to the relaxationFactor value

        scalar& relaxationFactor()
        {
            return relaxationFactor_;
        }

        //- Update coefficients
        virtual void updateCoeffs();

        //- Write
        virtual void write(Ostream&) const;
};


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace Foam
```

## Member Functions

The following is the excerpt from the member function for the explicit implementation of Hatzikiriakos slip law. There are two important things to note in this case. Firstly, the member data defined in this class are different. $k_{h1}$ and $k_{h2}$ are the two member data used in the compilation of the boundary condition instead of the n and the slipFactor. This means that these member data need to be initialized in the constructor of the boundary condition class. It is also important to note that these member data are also appropriately included in the main file, hatziSlipGenNewtonianRelaxationFvPatchField.C where the definition of the constructors are made.

Secondly, the method of update of the wall slip velocity is completely changed and it follows the equation 1.5. Appropriate changes need to be made in writing the values of the model parameters and value computed in the boundary condition. The following excerpt of the code from the main file, hatziSlipGenNewtonianRelaxationFvPatchField.C, describes the definition of the member function .

---

```
// * * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * * * //

void hatziSlipGenNewtonianRelaxationFvPatchVectorField::updateCoeffs()
{
   if (updated())
   {
      return;
   }

   //face normal vector
   vectorField nHat = this->patch().nf();
   //Info << "nhat" << nHat <<endl;
   // Du/dn
   vectorField gradient = this->snGrad();
   vectorField gradient1 = this->snGrad();
   // only tangential components
   gradient = transform(I - sqr(nHat), gradient);

   // gradient Direction (since the pow function doesn't take vectors, we later have
 to multiply the magnitude of the gradient with the gradient direction)
   vectorField gradientDirection = gradient / (mag(gradient) + SMALL);
   //Info << "grad_dir" << gradientDirection << endl;
   // slip velocity of the last iteration
   vectorField u_wallslip_lastIteration = (*this);

   const label patchI = patch().index();
   scalarField nuw = 1e-6*mag(patch().nf());

   if (db().found("turbulenceModel"))
   {
      const incompressible::turbulenceModel& turbModel =
         db().lookupObject<incompressible::turbulenceModel>
         (
            "turbulenceModel"
         );

      //Info<< "\nTurbulence found\n" << endl;
      nuw = rho_*turbModel.nu()().boundaryField()[patchI];
   }
   else
   {
```

```
172         nuw = mag(patch().nf());
173         //Info<< "\nTurbulence NOT found\n" << endl;
174     }
175
176     vectorField u_wallslip = kh1_*(Foam::sinh(-
    kh2_*nuw*mag(gradient)))*gradientDirection;
177
    vectorField::operator=(relaxationFactor_*u_wallslip_lastIteration+(1.0-relaxationFactor_)*u_wallslip);
178     Info<<"u_wallslip" <<u_wallslip[50]<<endl;
179     fixedValueFvPatchVectorField::updateCoeffs();
180
181
182 // Write
183 void hatziSlipGenNewtonianRelaxationFvPatchVectorField::write(Ostream& os) const
184 {
185     fvPatchVectorField::write(os);
186     os.writeKeyword("rho") << rho_ << token::END_STATEMENT << nl;
187     os.writeKeyword("kh2") << kh2_ << token::END_STATEMENT << nl;
188     os.writeKeyword("kh1") << kh1_ << token::END_STATEMENT << nl;
189     os.writeKeyword("relaxationFactor") << relaxationFactor_ << token::END_STATEMENT <<
    nl;
190     writeEntry("value", os);
191 }
192
193
194 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

Lines prior to lin176 remain the same as the Navier slip model.

- line 176: The updating formula is changed to suit the transcendental equation 1.5
- line 177: Relaxation is performed as done for the Navier model (still explicit formulation is used and relaxation is important)
- line 184-190: To write the value of the boundary condition computed parameter and the new model parameters to the file os.

#### Asymtotic explicit formulation

#### Procedure to set up the boundary condition

Follow the prescribed steps to set up an Asymtotic model from an existing hatzikiriakos model.

- `cd madhavan_files`
- cp -r hatzikiriakos ./Asymtotic
- cd Asymtotic/
- mv myhatziSlipGenNewtonianRelaxationV2/ myasymSlipGenNewtonianRelaxationV2/
- cd myasymSlipGenNewtonianRelaxationV2/
- OF22x
- wclean
- cd Make/

The options file remains the same as the non-linear Hatzikiriakos slip explicit case. The files file in the Asymtotic explicit formulation is written as following.

```
 asymSlipGenNewtonianRelaxationFvPatchField.C

LIB = $(FOAM_USER_LIBBIN)/libmyasymSlipGenNewtonianRelaxation
~
~
```

- cd ..
- mv hatziSlipGenNewtonianRelaxationFvPatchField.C asymSlipGenNewtonianRelaxationFv-PatchField.C[4]
- mv hatziSlipGenNewtonianRelaxationFvPatchField.H asymSlipGenNewtonianRelaxationFv-PatchField.H[5]
- replace hatziSlipGenNewtonianRelaxationFvPatchVectorField by hatziSlipGenNewtonianRelaxationFvPatchVectorField in both the header file and the main file using the sed command.
- sed -i 's/kh1/ka1/g' asymSlipGenNewtonianRelaxationFvPatchField.H
- sed -i 's/kh2/ka2/g' asymSlipGenNewtonianRelaxationFvPatchField.H
- sed -i 's/kh1/ka1/g' asymSlipGenNewtonianRelaxationFvPatchField.C
- sed -i 's/kh2/ka2/g' asymSlipGenNewtonianRelaxationFvPatchField.C
- In the main file, change the name of the header file manually.
- Change the typeName in the header file manually.

### Directory Structure

For the Asymtotic slip model, a directory structure similar to the Navier and Hatzikiriakos model is obtained by following the sequence of steps mentioned. The resulting deirectory structure could be seen as the following.

```
    Asymtotic
    |___ myasymSlipGenNewtonianRelaxationV2
        |___ Make
        |   |___ files
        |   |___options
        |___ asymSlipGenNewtonianRelaxationFvPatchField.C
        |___ asymSlipGenNewtonianRelaxationFvPatchField.H

3 directories, 4 files
```

### Constructors

The member data $k_{a1}$ and $k_{a2}$ are declared in the header file, asymSlipGenNewtonianRelaxationFv-PatchField.H and are operated upon in the member functions based on the transcendental equation for Asymtotic model for wall slip.

[4]remove the hyphen after RelaxationFv
[5]remove the hyphen after RelaxationFv

### Member Functions

As in the case of the Hatzikiriakos boundary condition, the Asymtotic boundary condition also uses two different member data $k_{a1}$ and $k_{a2}$ and these objects need to be constructed before the compilation of the boundary condition. Similarly, the method of updating of the wall slip velocity vectorField is altered according to the equation 1.6 and changes have been made in the write function to appropriately suit the model parameters used. The following is definition of the member function defined in main file, asymSlipGenNewtonianRelaxationFvPatchField.C

---

```
// * * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * * * //

void asymSlipGenNewtonianRelaxationFvPatchVectorField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    //face normal vector
    vectorField nHat = this->patch().nf();

    // Du/dn
    vectorField gradient = this->snGrad();

    // only tangential components
    gradient = transform(I - sqr(nHat), gradient);

    // gradient Direction (since the pow function doesn't take vectors, we later have
  to multiply the magnitude of the gradient with the gradient direction)
    vectorField gradientDirection = gradient / (mag(gradient) + SMALL);

    // slip velocity of the last iteration
    vectorField u_wallslip_lastIteration = (*this);

    const label patchI = patch().index();
    scalarField nuw = 1e-6*mag(patch().nf());

    if (db().found("turbulenceModel"))
    {
        const incompressible::turbulenceModel& turbModel =
            db().lookupObject<incompressible::turbulenceModel>
            (
                "turbulenceModel"
            );

        //Info<< "\nTurbulence found\n" << endl;
        nuw = rho_*turbModel.nu()().boundaryField()[patchI];
    }
    else
    {
        nuw = mag(patch().nf());
        //Info<< "\nTurbulence NOT found\n" << endl;
    }
    //Info<<"\n nuw_val = "<< nuw <<endl;

    // slip velocity in the current iteration
    vectorField u_wallslip = ka1_*(Foam::log(1 -
  ka2_*nuw*mag(gradient)))*gradientDirection;
179
180    //Info <<"\n u_wallslip = "<< u_wallslip << endl;
181
182    //Calculate and set u_wallslip
183
    vectorField::operator=(relaxationFactor_*u_wallslip_lastIteration+(1.0-relaxationFactor_)*u_wallslip);
184    Info <<"u_wallslip=" << u_wallslip[50]<<endl;
185    fixedValueFvPatchVectorField::updateCoeffs();
186 }
187
188
189 // Write
190 void asymSlipGenNewtonianRelaxationFvPatchVectorField::write(Ostream& os) const
191 {
192    fvPatchVectorField::write(os);
193    os.writeKeyword("rho") << rho_ << token::END_STATEMENT << nl;
194    os.writeKeyword("ka2") << ka2_ << token::END_STATEMENT << nl;
195    os.writeKeyword("ka1") << ka1_ << token::END_STATEMENT << nl;
196    os.writeKeyword("relaxationFactor") << relaxationFactor_ << token::END_STATEMENT <<
    nl;
197    writeEntry("value", os);
198 }
199
200
201 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
202
203 makePatchTypeField(fvPatchVectorField,
    asymSlipGenNewtonianRelaxationFvPatchVectorField);
204
205 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
206
207 } // End namespace Foam
208
209 // ************************************************************************* //
```

Lines prior to lin178 remain the same as the Navier/ Hatzikiriakos slip model.

- line 178: The updating formula is changed to suit the transcendental equation 1.5
- line 183: Relaxation is performed as done for the Navier model (still explicit formulation is used and relaxation is important)
- line 192-197: To write the value of the boundary condition computed parameter and the new model parameters to the file os.

### 1.2.5 Implementation of semi-implicit formulation

The aim of this section is to highlight the strategy used to implement a semi-implicit solver for non-linear boundary condition laws: non-linear Navier slip, Hatzikiriakos and Asymtotic laws. There is a fundamental difference between how the explicit and semi-implicit boundary conditions are applied to a boundary patch. In the semi-implicit formulation, the 'while' Boolean operator is used (will be shown in the excerpts from the code in the current section). This is used because in the semi-implicit method, the values need to be designated in each iteration of the bisection method loop conditionally based on what value each face of the patch assumes.

This brings the situation to the possibility that some of the faces in the patch satisfying the Boolean and others not. Therefore, a uniform application of boundary condition to the patch is not possible. For this purpose the faces of the boundary patch has to be looped over and the individual faces are checked if it satisfies the Boolean. The wall slip velocity variable is updated at individual faces of the patch unlike in the explicit method where they were updated for a patch at once for a given iteration (note, this is not the iteration of the bisection method but that of the steady state simpleFOAM solver.)

## Starting from explicit formulations

As discusses about the two practical challenges about the implementation of the semi-implicit formulation which are: looping over the all the individual faces of the patch and explicitly defining the gradient with the current iteration value of the wall slip velocity. Therefore a step by step approach is adopted where we ensure both the modifications do not provide any deviation from the results expected. In order to have a validation to this, we do this for the existing explicit formulation. First we loop over all the faces to apply the explicit boundary condition face by face and later apply the gradient boundary condition explicitly by mentioning the wall slip velocity of the previous iteration.

Regarding the first strategy to loop over all faces, the following excerpt of code could be pasted in the explicit formulation of Hatzikiriakos model and could be verified for results. The same results could be observed which proves that the technique used to apply the boundary condition on individual face is correct. This has to be done in the main file, hatziSlipGenNewtonianRelaxationFvPatchField.C in the directory corresponding to the explicit Hatzikiriakos model.

```
forAll(cPatch,faceI)
   {
   vector nHat_face = nHat[faceI];
   //Info << "nHat_face" << nHat_face <<endl;
   vector gradient_face = gradient[faceI];
   //gradient_face = transform(I -sqr(nHat_face),gradient_face);
   vector gradientDirection_face = gradientDirection[faceI];
   scalar nuw_face = nuw[faceI];
   u_wallslip[faceI] = kh1_*(Foam::sinh(-
       kh2_*nuw_face*mag(gradient_face)))*gradientDirection_face;
   //Info << "nuw_face" << nuw_face << "walllslip" << u_wallslip[faceI] <<endl;

  }
  Info << "walllslip" << u_wallslip[50] <<endl;
```

The part of the code in the explicit formulation of the Hatzikiriakos model that has to be removed is,

```
// * * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * * * * //



   // slip velocity in the current iteration
   vectorField u_wallslip =
 kh1_*(Foam::sinh(-kh2_*nuw*mag(gradient)))*gradientDirection;

   //Info <<"\n u_wallslip = "<< u_wallslip << endl;
```

In the directory pertaining to the explicit formulation of the Hatzikriakos model, the final case files are made available. For both the alternatives described, compile the boundary condition class using wmake command and run the case using the simpleFoam command. It could be observed that at individual faces of the patch, both the codes generates the same wall slip velocity.

It is of interest to note that 'faceI' is a running variable and it is correct for us to indicate the corresponding face positions to the quantities from the patch level descriptions as the index of the faces on the patch are numbered with linear incrimination This could be verified with the number in the boundary file in the constant/polyMesh directory with the startFace and the endFace index.

For example, if for the given configuration we run blockMesh and check the boundary file, it says startFace as 4875 and nFaces as 100. If we try to print from boundary condition class, the startFace and endFace, we would get 4875 and 4975 respectively.

The next step is to define the gradient of the velocity normal to the wall. In the explicit formulation, this is performed using the snGrad().

```
 vectoField snGrad = ( *this - pif)*this->patch().deltaCoeffs();
```

This activity can be performed by copying the following code into the member function in the main file of any particular model. It could be ensured that the snGrad() function and the explicitly defined equivalent gives the same gradient on a particular face of the boundary patch.

```
       // check grad
    vectorField check_grad = (*this - pif )*this->patch().deltaCoeffs();

    Info <<"check_grad" << check_grad[50] << "grad" <<gradient[50]<<endl;
```

It is very important to note that the snGrad() function could be replaced in the explicit formulation by the equivalent mentioned. The validity of this statement is crucial for the implementation of the semi-implicit formulation as the *this in the formula would be replaced by the guess value and the entire framework holds validity to be interchangeably used.

### Moving on to semi implicit formulations

At first, implement the semi-implicit formulation for non-linear Navier slip law, and then extend the same for Hatzikiriakos and Asymtotic laws.

### Navier Slip law - semi-implicit formulation

### Procedure to set up the boundary condition

Follow the set of steps to develop a Navier semi implicit boundary condition from Navier explicit boundary condition.

- `cd madhavan_files`
- `cp -r navier  navier_semi`
- `cd navier_semi/`
- mv myNLSlipGenNewtonianRelaxationV2 myNLSlipGenNewtoniansemiV2
- cd myNLSlipGenNewtoniansemiV2/
- wclean
- cd Make/

The options file in the Make directory remains the same as in the explicit formulations. The files file in the semi-implicit Navier slip is written as the following

```
nonLinNavSlipGenNewtoniansemiFvPatchField.C

LIB = $(FOAM_USER_LIBBIN)/libmyNonLinNavSlipGenNewtoniansemi
```

- cd ..

- mv nonLinNavSlipGenNewtonianRelaxationFvPatchField.C nonLinNavSlipGenNewtoniansemiFv-PatchField.C [6]
- mv nonLinNavSlipGenNewtonianRelaxationFvPatchField.H nonLinNavSlipGenNewtoniansemiFv-PatchField.H [7]
- replace nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField by nonLinNavSlipGenNew-toniansemiFvPatchVectorField in both the header and the main files. [8]
- Change the name of the header file in the include statements in the main file
- Change the typeName in the header file manually.

### Directory Structure

The Directory structure and the changes in the files file are very similar to the explicit case. The directory structure could be seen in the following. Since large parts of codes have to copied into the member functions of the main file, the readers are suggested to use the final files to test the code, nonLinNavSlipGenNewtoniansemiFvPatchField.C and nonLinNavSlipGenNewtoniansemiFv-PatchField.H, which are provided.

```
    navier_semi
    |___ mynonLinNavSlipGenNewtoniansemiV2
        |___ Make
        |   |___ files
        |   |___options
        |___ nonLinNavSlipGenNewtoniansemiFvPatchField.C
        |___ nonLinNavSlipGenNewtoniansemiFvPatchField.H

3 directories, 4 files
```

### Constructors

The member data that are to be constructed are the same for both explicit and semi-implicit formulations. This means that the same .H files could be used for a particular formulation irrespective of whether semi-implicit or explicit formulation is used. However one should change the typeName, this would be the name by which the newly created semi-implicit boundary condition class will be saved in the hash table.

### Member Functions

The major change in the boundary condition implementation comes from the member function definition and it is explained in the following lines in nonLinNavSlipGenNewtoniansemiFvPatchField.C. One could replace the member functions in the explicit solver's member function in the main file by the following to have the semi implicit solver implemented.

```
// * * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * * * * //

void nonLinNavSlipGenNewtoniansemiFvPatchVectorField::updateCoeffs()
{
    if (updated())
    {
```

[6]Remove the hyphen after semiFv
[7]Remove the hyphen after semiFv
[8]Remove the hyphen after GenNew

```
        return;
    }
    // obatin the viscosity
    const label patchI = patch().index();
    scalarField nuw = 1e-6*mag(patch().nf());

    if (db().found("turbulenceModel"))
    {
        const incompressible::turbulenceModel& turbModel =
            db().lookupObject<incompressible::turbulenceModel>
            (
                "turbulenceModel"
            );

        //Info<< "\nTurbulence found\n" << endl;
        nuw = rho_*turbModel.nu()().boundaryField()[patchI];
    }
    else
    {
        nuw = mag(patch().nf());
        //Info<< "\nTurbulence NOT found\n" << endl;
    }
    //Info<<"\n nuw_val = "<< nuw <<endl;

   //-----------definition of viscosity in the previous iteration finsihes above-- we could use it as it is------------

   //----------definition for patch level properties-----

    //face normal vector
    vectorField nHat = this->patch().nf();
    //Info << "nHat" << nHat;
    // cell centre value
    vectorField pif = this->patchInternalField();
    // gradient at the patch level
    vectorField gradient = this->snGrad();
    //Info << "grdient" << gradient[50] << "g.x"<<gradient[50].component(0) << "g.y" <<gradient[50].component(1) <<"g.z" << gradient[50].component(2) << endl;
    //Info << "gradient =" << gradient[50] <<endl;
    vectorField gradient1 = this->snGrad(); //--- used for definition at the face level
    // only tangential components
    gradient = transform(I - sqr(nHat), gradient);
    // gradient Direction (since the pow function doesn't take vectors, we later have to multiply the magnitude of the gradient with the gradient direction)
     vectorField gradientDirection = gradient / (mag(gradient) + SMALL);
    // obtain 1/del_y for the patch
     scalarField invdelypatch = this->patch().deltaCoeffs();
    //Info << "dely" <<invdelypatch<< endl;
    // defining the wall slip velocity in the previous iteration
     vectorField u_wallslip_prev = *this;
     vectorField u_wallslip = *this; // --- it is an initialization which would change over loop of faces
     //Info <<"IF" << pif[50] << "boun" << u_wallslip_prev[50] <<endl;
    const labelList cells = patch().faceCells();
    //Info<< "cellcenre"<<pif[50]<<"nuwface"<<nuw[50]<<"hfcell"<<invdelypatch[50];
    //-------------------------------looping over the facecells starts here-------------------------------------
     const fvMesh& mesh   = patch().boundaryMesh().mesh();
    const faceList& faces = mesh.faces();
```

```
const label& startFace = patch().patch().start();
const label   endFace = startFace + size() - 1 ;
label patchID = mesh.boundaryMesh().findPatchID("patchI");
const polyPatch& cPatch = mesh.boundaryMesh()[patchI];
label faceId_start = cPatch.start();
forAll(cPatch,facei)
{
   //obtain the nuw for the face from the patch
   scalar nuw_face = nuw[facei];

   //getting the 1/del_y for the individual face
   scalar invdely = invdelypatch[facei];

   //gradient of velocity in cell in previous iteration
   vector gradient_prev_face = gradient1[facei];
   //obtain the normal for the face
   vector nHat_face = nHat[facei];
   //gradient of velocty in the prev iteration---- only tangential
   gradient_prev_face = transform(I -sqr(nHat_face),gradient_prev_face);
   //initialise the guess limits for the bisection method
   vector a(0,0,0);
   vector b = pif[facei];
   //introduce the guess function
   vector guess_value = 0.5*(a+b);
   //Info << "gv" << guess_value<<endl;
   //defining the internal field of the cell of the face in the previous iteration
   vector pif_face = pif[facei];
   //defining the new gradient with the guess value and the internal field of the cell
   vector gradient_face = (guess_value - pif_face)*invdely;
   //obtaining only the tangential components of the gradient
   gradient_face = transform(I -sqr(nHat_face),gradient_face);
   //obtaining the direction of the gradient
   vector gradient_cellDirection = gradient_face / (mag(gradient_face) + SMALL);
   //obtain the residual in the transcedent equation
   vector u_wallslip_func = guess_value + slipFactor_*(Foam::pow(nuw_face, n_))*mag(Foam::pow(mag(gradient_face), n_))*gradient_cellDirection;
   //set the guess value as the root if not needed to go into the loop
   u_wallslip[facei] = guess_value;
   // ---------start the boolean here to apply bisection methood -------------------
   scalar j = 0;
   scalar threshold = (b.component(0) - a.component(0))/(Foam::pow(2,j));
   while(threshold > 1e-9)
   {
   //introduce the guess function
    guess_value = 0.5*(a+b);

    //Info << "guess_value" << guess_value<<endl;
   //defining the internal field of the cell of the patch
    pif_face = pif[facei];
   //defining the new gradient with the guess value and the internal field of the cell
    gradient_face = (guess_value - pif_face)*invdely;
   //obtain the normal for the face
    nHat_face = nHat[facei];
   //obtaining only the tangential components of the gradient
    gradient_face = transform(I -sqr(nHat_face),gradient_face);
    gradient_cellDirection = gradient_face / (mag(gradient_face) + SMALL);
   //Info << "direction" << gradient_cellDirection;
```

```
      u_wallslip_func = guess_value + slipFactor_*(Foam::pow(nuw_face, n_))*mag(Foam::pow(mag(gradient_face), n_))*gradient_cellDirection;
      //obtain only the tangential components of the residual from the trascedent equation
       scalar tan_uws_func = u_wallslip_func.component(0);
      //apply the conditions of bosection- method to transcedent equation to find the root

       if (tan_uws_func >0)
       {
          b = guess_value;
       }
       else if (tan_uws_func <0)
       {
          a = guess_value;
       }

       vector u_wallslip_face = guess_value;
       //tan_uws_func =mag(u_wallslip_cell.component(0));
       //Info << "tan_vel =" << tan_uws_func;
       u_wallslip[facei] = u_wallslip_face;
       j = j +1;
       //Info << "a="<< a <<" ; "<< "b=" << b<< endl;
       threshold = (b.component(0) -a.component(0))/(Foam::pow(2,j));
       //Info <<"threshold_loop" << threshold;
      }
// i = i +1;
  //Info << "wallslip" << u_wallslip[facei];
 }
   Info<< "u_walllslip= " <<u_wallslip[50]<<endl;
   //Calculate and set u_wallslip
vectorField::operator=(relaxationFactor_*u_wallslip_prev+(1.0-relaxationFactor_)*u_wallslip);
   fixedValueFvPatchVectorField::updateCoeffs();
}



// Write
void nonLinNavSlipGenNewtoniansemiFvPatchVectorField::write(Ostream& os) const
{
   fvPatchVectorField::write(os);
   os.writeKeyword("rho") << rho_ << token::END_STATEMENT << nl;
   os.writeKeyword("n") << n_ << token::END_STATEMENT << nl;
   os.writeKeyword("slipFactor") << slipFactor_ << token::END_STATEMENT << nl;
   os.writeKeyword("relaxationFactor") << relaxationFactor_ << token::END_STATEMENT << nl;
   writeEntry("value", os);
}


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

makePatchTypeField(fvPatchVectorField, nonLinNavSlipGenNewtoniansemiFvPatchVectorField);

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace Foam
```

- lines 134-139: To ensure if the values are up-to-date before the boundary condition is called for the next time.
- lines 140-159: return the value of the Dynamic viscosity based on the turbulence model used. Since the value of the mean velocity chosen is so small, the calculations will pertain to Laminar model for computing the Dynamic viscosity (will be elaborated in the 'case set up')
- line167: the normal vector to all the faces of the boundary patch is defined as an array.
- line170: The internal fields, the cell center velocity of the cells adjacent to the boundary patch is defined as an array- These are the values from the previous iteration.
- line 175: The gradient of velocity on the faces of the patch is computed.These are the values from the previous iteration.
- line 177: From all the three components of the gradient, the tangential gradient of all three velocity components are derived using the transform function. This is the wall normal gradient.
- line 179: The normalized vector of the wall normal gradient is obtained. This is again an array of values containing values for all the faces on the boundary patch.
- line 181: The half cell length is obtained for all the cells attached to the boundary patches.
- lines 190-197: The mesh properties are defined and the looping is started over all the faces of the loop. facei is used as the running index to loop over the faces.
- line 200: The dynamic viscosity is derived from the patch level to the face level
- line 203: The half cell length, i.e the distance between the cell centre and the face of cell is defined for the faces adjacent to the patch from the properties of the patch.
- line 208: The surface normal vector of the particular face is obtained from the patch property.
- line 210: The gradient of the face in the previous iteration is derived from the patch
- line 212-213: The guess vectors are initialized for the non-linear Navier slip law.
- line 215: The guess value is estimated from the range
- line 218: The cell center velocity of the particular face is obtained from the patch
- line 220: The gradient of the face is computed for the current iteration with the guess value
- line 222: The tangential component is obtained from the gradient using the transform function
- line 226: The residual of the guess vector is estimated in the transcendental equation for slip for non-linear Navier slip.
- line 231: The threshold is defined for the bisection method
- lines 253-260: The guess limits of the bisection method are adjusted to reach the threshold defined.
- The update value of the wall slip velocity is stored in the data member u$_wallslipasavectorField.$

**Hatzikiriakos slip law - semi-implicit formulation**

**Procedure to set up the boundary condition**

Follow the set of steps to develop a semi-implicit formulation based Hatzikiriakos model. The set of steps are directed to make the modifications to start from the semi-implicit formulation based Navier slip model.

- 
- `cd madhavan_files`
- `cp -r hatzikiriakos hatzikiriakos_semi`
- `cd hatzikiriakos_semi`
- mv myhatziSlipGenNewtonianRelaxationV2/ myhatziSlipGenNewtoniansemiV2/
- cd myhatziSlipGenNewtoniansemiV2/
- OF22x
- mv hatziSlipGenNewtonianRelaxationFvPatchField.C hatziSlipGenNewtoniansemiFvPatchField.C
- mv hatziSlipGenNewtonianRelaxationFvPatchField.H hatziSlipGenNewtoniansemiFvPatchField.H
- Replace nonLinNavSlipGenNewtonianRelaxationFvPatchVectorField by hatziSlipGenNewtoniansemiFvPatchVectorField using the sed command in both the header and the main file.
- Replace the name of the header file in the include statement in the main file in the main file.
- cd Make/

The options file in the Make directory remains the same as in the explicit formulations. The files file in the semi-implicit Hatzikiriakos slip is written as the following

```
hatziSlipGenNewtoniansemiFvPatchField.C

LIB = $(FOAM_USER_LIBBIN)/libmyhatziSlipGenNewtoniansemi
```

Now the code is ready for changes in the member functions which distinguishes it from the Navier slip model.

**Directory Structure**

The Directory structure and the changes in the files file are very similar to the explicit case. The directory structure could be seen in the following.

```
   hatzi_semi
   |___ myhatziSlipGenNewtoniansemiV2
       |___ Make
       |   |___ files
       |   |___options
       |___ hatziSlipGenNewtoniansemiFvPatchField.C
       |___ hatziSlipGenNewtoniansemiFvPatchField.H

3 directories, 4 files
```

### Constructors

The constructors are the same as used for the explicit formulation of the Hatzikiriakos slip law. The member data used in the compilation of the boundary conditions are density rho, relaxationFactor, $k_{h1}$ and $k_{h2}$. Similarly, the typeName is changed to be referenced uniquely through the hash table.

### Member Functions

It is of our interest to note the changes involved in the implementation of the Hatzikiriakos semi-implicit model. From the analytical solution it has been proved that the solution guess range for the bisection need to be as large as in the Navier slip model but could be operated in a smaller one to have lesser number of iteration by the bisection method.

The initial guess range for the Hatzikiriakos semi-implicit model has the following limits according to the analytical proofs.

$$\Big[a; \quad b\Big] = \Big[\tfrac{k_{H1}k_{H2}\mu(\gamma)^{i-1}/\Delta y}{k_{H1}k_{H2}\mu(\gamma)^{i-1}/\Delta y+1}; \quad {u_p}^{i-1}\Big] \tag{1.11}$$

The following is excerpt from the code in the main file, hatziSlipGenNewtoniansemiFvPatchField.C where the member functions are defined. The excerpt shows the piece of the code where the conditions are changed in the updateCoeffs(), it does not show the write(). It is to be taken care that $k_{h1}$ and $k_{h2}$ are written consistent to the model.

```
    forAll(cPatch,facei)
    {
       //obtain the nuw for the face from the patch
       scalar nuw_face = nuw[facei];

       //getting the 1/del_y for the individual face
       scalar invdely = invdelypatch[facei];

       //gradient of velocity in cell in previous iteration
       vector gradient_prev_face = gradient1[facei];
       //obtain the normal for the face
       vector nHat_face = nHat[facei];
       //gradient of velocty in the prev iteration---- only tangential
       gradient_prev_face = transform(I -sqr(nHat_face),gradient_prev_face);
       //initialise the guess limits for the bisection method
       //scalar num = kh1_*kh2_*nuw_cell*invdely;
       //num = num/(num + 1);
       //vector a(num,num,num);
       vector a(0,0,0);
       vector b = pif[facei];
       //introduce the guess function
       vector guess_value = 0.5*(a+b);
       //Info << "gv" << guess_value<<endl;
       //defining the internal field of the cell of the face in the previous iteration
       vector pif_face = pif[facei];
       //defining the new gradient with the guess value and the internal field of the
cell
       vector gradient_face = (guess_value - pif_face)*invdely;
       //obtaining only the tangential components of the gradient
       gradient_face = transform(I -sqr(nHat_face),gradient_face);
       //obtaining the direction of the gradient
       vector gradient_cellDirection = gradient_face / (mag(gradient_face) + SMALL);
       //obtain the residual in the transcedent equation
       vector u_wallslip_func = guess_value -
kh1_*(Foam::sinh(-kh2_*nuw_face*mag(gradient_face)))*gradient_cellDirection;
       //set the guess value as the root if not needed to go into the loop
```

```
      u_wallslip[facei] = guess_value;
      // ---------start the boolean here to apply bisection methood -------------------
      scalar j = 0;
      scalar threshold = (b.component(0) - a.component(0))/(Foam::pow(2,j));
      while(threshold > 1e-9)
      {
      //introduce the guess function
       guess_value = 0.5*(a+b);

       //Info << "guess_value" << guess_value<<endl;
      //defining the internal field of the cell of the patch
       pif_face = pif[facei];
      //defining the new gradient with the guess value and the internal field of the cell
       gradient_face = (guess_value - pif_face)*invdely;
      //obtain the normal for the face
       nHat_face = nHat[facei];
      //obtaining only the tangential components of the gradient
       gradient_face = transform(I -sqr(nHat_face),gradient_face);
       gradient_cellDirection = gradient_face / (mag(gradient_face) + SMALL);
      //Info << "direction" << gradient_cellDirection;
       vector u_wallslip_func = guess_value - kh1_*(Foam::sinh(-kh2_*nuw_face*mag(gradient_face)))*gradient_cellDirection;
      //obtain only the tangential components of the residual from the trascedent equation
       scalar tan_uws_func = u_wallslip_func.component(0);
      //apply the conditions of bosection- method to transcedent equation to find the root

       if (tan_uws_func >0)
       {
           b = guess_value;
       }
       else if (tan_uws_func <0)
       {
           a = guess_value;
       }

       vector u_wallslip_face = guess_value;
       //tan_uws_func =mag(u_wallslip_cell.component(0));
       //Info << "tan_vel =" << tan_uws_func;
       u_wallslip[facei] = u_wallslip_face;
       j = j +1;
       //Info << "a="<< a <<" ; "<< "b=" << b<< endl;
       threshold = (b.component(0) -a.component(0))/(Foam::pow(2,j));
       //Info <<"threshold_loop" << threshold;
      }
// i = i +1;
  //Info << "wallslip" << u_wallslip[facei];
 }
   Info<< "u_walllslip= " <<u_wallslip[50]<<endl;
   //Calculate and set u_wallslip
vectorField::operator=(relaxationFactor_*u_wallslip_prev+(1.0-relaxationFactor_)*u_wallslip);
   fixedValueFvPatchVectorField::updateCoeffs();
}

```

lines 211-216: new limits for the bisection method has been set up the and the guess vector will be based on these values. But when the case was set up, these limits did not work out to good results. So as for Navier slip law, the same guess limits of $[0, u_p^{i-1}]$ is used.

line 229:The residual is computed from the altered transcendental equation.

line 251: The residual is computed in the bisection based on the altered transcendental equation for the Hatzikiriakos slip law

### Asymtotic slip law - semi-implicit formulation

### Procedure to set up the boundary condition

The following set of procedure described is to organize a directory structure similar to the explicit forumation of Navier slip model and change the necessary class and class member names.

- 
- `cd madhavan_files`
- `cp -r Asymtotic Asymtotic_semi`
- `cd Asymtotic_semi`
- mv asymSlipGenNewtonianRelaxationV2/ myasymSlipGenNewtoniansemiV2/
- cd myasymSlipGenNewtoniansemiV2/
- rm comentario
- mv asymSlipGenNewtonianRelaxationFvPatchField.C asymSlipGenNewtoniansemiFvPatchField.C
- mv asymSlipGenNewtonianRelaxationFvPatchField.H asymSlipGenNewtoniansemiFvPatchField.H
- Replace asymSlipGenNewtonianRelaxationFvPatchVectorField by asymSlipGenNewtoniansemiFvPatchVectorFieldin both the header and the main file using the sed command.
- Replace the name of the header file in the include statement in the main file.
- cd Make/

The options file in the Make directory remains the same as in the explicit formulations. The files file in the semi-implicit Asymtotic slip is written as the following

```
nonasymSlipGenNewtoniansemiFvPatchField.C

LIB = $(FOAM_USER_LIBBIN)/libmyasymSlipGenNewtoniansemi
```

### Directory Structure

The Directory structure is very similar to the explicit case and as changes within the formulations in the explicit method, the files file will change. The directory structure could be visualized in the following.

```
asym_semi
|___ myasymSlipGenNewtoniansemiV2
    |___ Make
    |   |___ files
    |   |___options
```

```
        |___ asymSlipGenNewtoniansemiFvPatchField.C
        |___ asymSlipGenNewtoniansemiFvPatchField.H

3 directories, 4 files
```

## Constructors

The constructors are the same as used for the explicit formulation of the Asymtotic slip law. The member data used in the compilation of the boundary conditions are density rho, relaxationFactor, $k_{a1}$ and $k_{a2}$. Similarly, the typeName is changed to be referenced uniquely through the hash table.

## Member functions

It is of our interest to note the differences involved to implement a semi-implicit formulation of Asymtotic law. From analytical solutions it is derived and proved that the limits of the solution or the root of the transcendental equation (wall slip velocity) is always within the given range of values as described in the equation 1.12.

$$\begin{cases} [a,b] = [0, u_p^{i-1}] & k_{a1} \geq 1 \\ [a,b] = [0, \frac{k_{a1}k_{a2}\mu(\gamma)^{i-1} + k_{a1}\Delta y_f}{k_{a1}k_{a2}\mu(\gamma)^{i-1} + \Delta y_f} u_p^{i-1}] & k_{a1} < 1 \end{cases} \tag{1.12}$$

The guess vector which is used in the formulation is derived from these limits and is used in gradient calculation for the current iterations.

The transcendental equation has been changed again as in the case of the explicit formulation. The residual has been calculated with respect to this transcendental equation. This is illustrated in the excerpt from the member from the code.

```
   forAll(cPatch,facei)
   {
      //obtain the nuw for the face from the patch
      scalar nuw_face = nuw[facei];

      //getting the 1/del_y for the individual face
      scalar invdely = invdelypatch[facei];

      //gradient of velocity in cell in previous iteration
      vector gradient_prev_face = gradient1[facei];
      //obtain the normal for the face
      vector nHat_face = nHat[facei];
      //gradient of velocty in the prev iteration---- only tangential
      gradient_prev_face = transform(I -sqr(nHat_face),gradient_prev_face);
      //initialise the guess limits for the bisection method
      scalar num = (ka1_*ka2_*nuw_face + (ka1_/invdely))/(ka2_*ka1_*nuw_face + (1/invdely));
      vector a (0,0,0);
      //vector b = pif[facei];
      vector b = num*pif[facei];
      //introduce the guess function
      vector guess_value = 0.5*(a+b);
      //Info << "gv" << guess_value<<endl;
      //defining the internal field of the cell of the face in the previous iteration
      vector pif_face = pif[facei];
      //defining the new gradient with the guess value and the internal field of the cell
      vector gradient_face = (guess_value - pif_face)*invdely;
      //obtaining only the tangential components of the gradient
      gradient_face = transform(I -sqr(nHat_face),gradient_face);
      //obtaining the direction of the gradient
      vector gradient_cellDirection = gradient_face / (mag(gradient_face) + SMALL);
      //obtain the residual in the transcedent equation
      vector u_wallslip_func = guess_value - ka1_*(Foam::log(1-ka2_*nuw_face*mag(gradient_face)))*gradient_cellDirection;
      //set the guess value as the root if not needed to go into the loop
      u_wallslip[facei] = guess_value;
      // ---------start the boolean here to apply bisection method -------------------
      scalar j = 0;
      scalar threshold = (b.component(0) - a.component(0))/(Foam::pow(2,j));
      while(threshold > 1e-9)
      {
      //introduce the guess function
       guess_value = 0.5*(a+b);

       //Info << "guess_value" << guess_value<<endl;
      //defining the internal field of the cell of the patch
       pif_face = pif[facei];
      //defining the new gradient with the guess value and the internal field of the cell
       gradient_face = (guess_value - pif_face)*invdely;
      //obtain the normal for the face
       nHat_face = nHat[facei];
      //obtaining only the tangential components of the gradient
       gradient_face = transform(I -sqr(nHat_face),gradient_face);
       gradient_cellDirection = gradient_face / (mag(gradient_face) + SMALL);
      //Info << "direction" << gradient_cellDirection;
       vector u_wallslip_func = guess_value - ka1_*(Foam::log(1-ka2_*nuw_face*mag(gradient_face)))*gradient_cellDirection;
      //obtain only the tangential components of the residual from the trascedent equation
       scalar tan_uws_func = u_wallslip_func.component(0);
      //apply the conditions of bosection- method to transcedent equation to find the root

       if (tan_uws_func >0)
       {
           b = guess_value;
       }
       else if (tan_uws_func <0)
       {
           a = guess_value;
       }

       vector u_wallslip_face = guess_value;
       //tan_uws_func =mag(u_wallslip_cell.component(0));
       //Info << "tan_vel =" << tan_uws_func;
       u_wallslip[facei] = u_wallslip_face;
       j = j +1;
       //Info << "a="<< a <<" ; "<< "b=" << b<< endl;
       threshold = (b.component(0) -a.component(0))/(Foam::pow(2,j));
       //Info <<"threshold_loop" << threshold;
      }
// i = i +1;
  //Info << "wallslip" << u_wallslip[facei];
 }
   Info<< "u_walllslip= " <<u_wallslip[50]<<endl;
   //Calculate and set u_wallslip

```

```
    vectorField::operator=(relaxationFactor_*u_wallslip_prev+(1.0-relaxationFactor_)*u_wallslip);
273    fixedValueFvPatchVectorField::updateCoeffs();
```

lines 206-209: New limits for the bisection method has been set up and the guess vector will be based on these values

line 222: The residual is computed from the altered transcendental equation.

line 244: The residual is computed in the bisection based on the altered transcendental equation for the Asymtotic slip law.

## 1.3 Setting up the case

The next step is to test the implementation made by setting up the case. Open the terminal and follow the following steps

- OF22x
- cp -r /chalmers/sw/unsup64/OpenFOAM/OpenFOAM-2.2.x/tutorials/incompressible/simpleFoam/pitzDaily .
- mv pitzDaily slipFlow
- cd slipFlow
- cd 0

Change the U file to the following

Note the custom boundary condition applied to the top and the bottom boundary walls. The case is typically solved as a 2D problem. Therefore empty condition is applied to the front and back. As mentioned in the previous section about the constructor initialization, the values are looked up here in the file. The rho, n,slipFactor and relaxationfactor are user-defined values. It is also important to note the changes made on the names of the patches as we only have 5 patches with the names described below, unlike the pitzdaily case.

```
dimensions     [0 1 -1 0 0 0 0];

internalField  uniform (0 0 0);

boundaryField
{
    topWall // upperwall
    {
        type               nonLinNavSlipGenNewtonianRelaxation; // could be any custom BC
        //type             uniform(0,0,0) now removed for the custom BC
        rho                1000;
        slipFactor         0.01;
                n                  0.5;
        relaxationFactor   0.9;
        value              (0 0 0);
    }
// n, slipFactor, relaxationFactor and n introduced for the custom boundary
// These values would be looked-up for
// Introduce kh1, kh2 for hatzikriakos
// Introduce ka1, ka2 here for Asymtotic
    bottomWall //lowerwall
    {
        type               nonLinNavSlipGenNewtonianRelaxation; // could be any custom BC
        //type             uniform(0,0,0) now removed for the custom BC
        rho                 1000;
        slipFactor          0.01;
                n                   0.5;
        relaxationFactor    0.9;
        value               (0 0 0);
    }
// n, slipFactor, relaxationFactor and n introduced for the custom boundary
// These values would be looked-up for
// Introduce kh1, kh2 for hatzikriakos
// Introduce ka1, ka2 here for Asymtotic
    inlet
    {
        type            fixedValue;
        value           uniform (0.001 0 0); // uniform(10, 0, 0); value changed to suit the
            case
         }

    outlet
    {
                type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

Change the p file to the following.

The names of the patches have been changed accordingly as in the U file and the outlet has been set to a constant pressure of 0. There are no special changes made to the boundary conditions in p and the again the empty boundary condition on front and back reflects the 2D nature of the problem.

Since in the case we have, we have a very low value of the mean flow, technically no turbulence is of interest. Laminar model is solved and therefore all the initialization of other turbulent properties are removed from the 0 directory.

- rm k
- rm nut
- rm nuTilda
- cd ..
- cd constant/polyMesh

change the blockMeshDict to the following The length of the channel is 0.2m, the height of the channel is 0.02m and the depth (1 cell-in the z direction) is 0.0002m. Based on this the vertices are defined. Hexahedral cells are chosen with no refinement of the mesh size along any of the directions. Then vertices pertaining to particular patches are grouped together.

```
convertToMeters 0.1;

vertices
(
    (0 0 0)
    (0.2 0 0)
    (0.2 0.02 0)
    (0 0.02 0)
    (0 0 0.0002)
    (0.2 0 0.0002)
    (0.2 0.02 0.0002)
    (0 0.02 0.0002)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (100 25 1) simpleGrading (1 1 1)
);

edges
(
);

patches
(
    wall topWall
    (
        (3 7 6 2)
    )
         wall bottomWall
         (
                  (1 5 4 0)
         )
    wall inlet
    (
        (0 4 7 3)
         )
         wall outlet
         (
        (2 6 5 1)
    )
    empty frontAndBack
    (
        (0 3 2 1)
        (4 5 6 7)
    )
);

mergePatchPairs
(
);
```

---

- Make the flow Laminar in RASProperties.
- switch off the turbulence
- switch off printCoeffs
- change the value of Dynamic viscosity to 2e-6 in transportProperties; other parameters could be commented or removed
- cd ..
- cd system

change the fvSolution file to the following Keep only the p and U sub-dictionary. All the others could be eliminated. The resulting file would look like the following

```
solvers
{
    p
    {
        solver         PCG;
        preconditioner DIC;
        tolerance      1e-07;
        relTol         0.01;
    }

    U
    {
        solver         PBiCG;
        preconditioner DILU;
        tolerance      1e-06;
        relTol         0.1;
    }

//     k
//    {
//        solver         PBiCG;
//        preconditioner DILU;
//        tolerance      1e-05;
//        relTol         0.1;
//    }

//    epsilon
//    {
//       solver         PBiCG;
//        preconditioner DILU;
//        tolerance      1e-05;
//        relTol         0.1;
//    }

//    R
//    {
//        solver         PBiCG;
//        preconditioner DILU;
//        tolerance      1e-05;
//        relTol         0.1;
//    }

//    nuTilda
//    {
//        solver         PBiCG;
//        preconditioner DILU;
//        tolerance      1e-05;
//        relTol         0.1;
 //   }




}

SIMPLE
{
    nNonOrthogonalCorrectors 0;
    residualControl
    {
        p               1e-12;
        U               1e-12;
//    "(k|epsilon|omega)" 1e-3;

    }
}

relaxationFactors
{
    fields
    {
        p               0.3;
    }
    equations
    {
        U               0.7;
//        k               0.7;
//        epsilon         0.7;
//        R               0.7;
//        nuTilda         0.7;

    }
}
```

change the fvSchemes file to the following

Keep the files related to U, p and phi. One could remove all the other fields pertaining to the turbulent properties. The resulting file would like the following

```
ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         Gauss linear;
    grad(p)         Gauss linear;
    grad(U)         Gauss linear;
}

divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss linearUpwind grad(U);
    div((nuEff*dev(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default         none;
    laplacian(nuEff,U) Gauss linear corrected;
    laplacian((1|A(U)),p) Gauss linear corrected;
    laplacian(DkEff,k) Gauss linear corrected;
    laplacian(DepsilonEff,epsilon) Gauss linear corrected;
    laplacian(DREff,R) Gauss linear corrected;
    laplacian(DnuTildaEff,nuTilda) Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
    interpolate(U) linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    p               ;
}
```

change the controlDict file to the following

endTime is adjusted based on the number of iterations the simulations take to converge. The default functions in the controlDict are removed and in the library to access, the typeName of the boundary condition class (specified in the header files) is mentioned. This is where the typeName is uniquely identified from the hash table when the case is setup.

```
application simpleFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         2000;

deltaT          1;

writeControl    timeStep;

writeInterval   100;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;

libs
(
"libmyNonLinNavSlipGenNewtonianRelaxation.so"
);
```

The final directory structure before the compilation of the case should be as the following.

```
.
|___ slipFactor0.01
    |__ 0
    |   |__ p
    |   |__ U
    |__ constant
    |   |__ polyMesh
    |   |__ RASProperties
    |   |__ transportProperties
    |__ system
        |__ controlDict
        |__ decomposeParDict
        |__ fvSchemes
        |__ fvSolution
        |__ sampleDict

5 directories, 9 files
```

- cd ..
- blockMesh
- simpleFoam

## 1.4 Results and Discussion

It could be identified from the transcendental equations that one of the two model parameters in all the three models in linear co-efficient and the other is a non-linear co-efficient. It was considered to be interesting to vary the non-linear model parameter n in the case of Navier slip model, $k_{h2}$ in the case of Hatzikiriakos model and $k_{a2}$ in the case of Asymtotic model.

The next important observation to be made the accuracy of the semi-implicit solution with respect to the explicit solution. For the Navier slip model, a constant value of 0.01 for the slipFactor and 3 different values of 1,3 and 0.5 were assumed for the value of n. The plot is made for the half-channel length and we could see a good agreement of results between the explicit and semi-implicit formulation.. These plots can be seen in Figure 1.1

Similarly for the Hatzikiriakos model, the non-linear parameter $k_{h2}$ was varied for different values of 1,3 and 0.5 for a constant value of $k_{h1}$ of 0.01. A good agreement of results could be seen from the plot in Figure 1.2.

For the Asymtotic model, a similar comparison was made between the explicit and the semi-implicit model is performed using the non-linear parameter $k_{a2}$-. 1,3 and 0.5 are the values chosen for the non linear parameter for a constant value of $k_{a1}$ of 0.01. The variation could be seen in Figure 1.3.

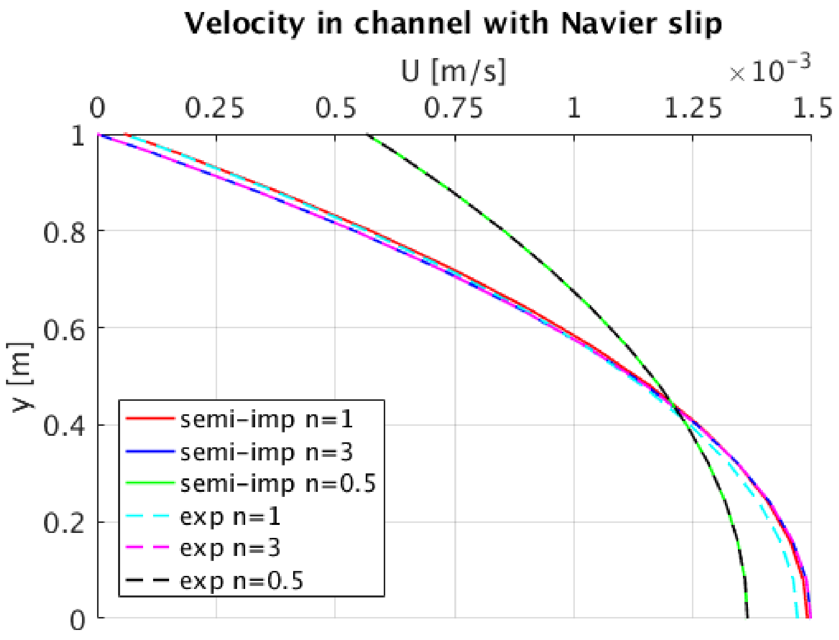

Figure 1.1: Comparison of formulations for Navier slip

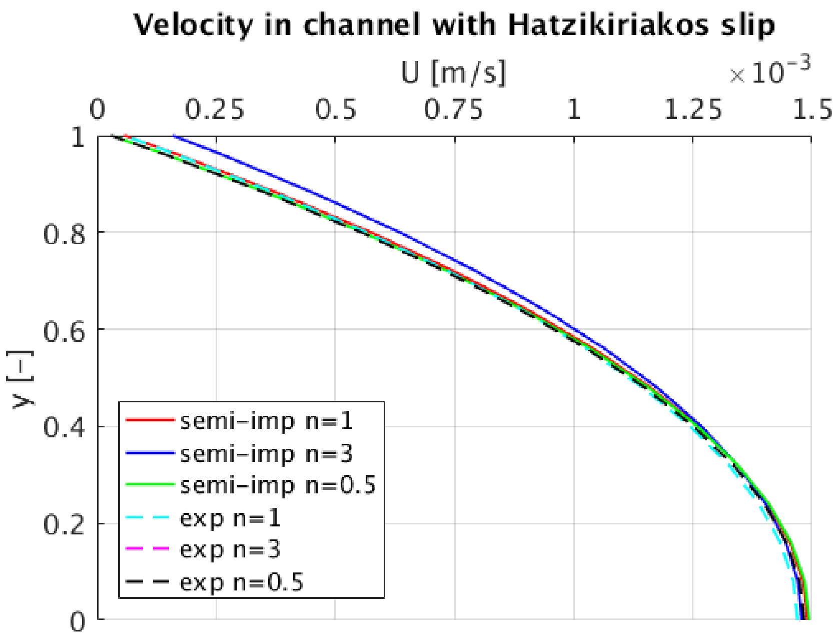

Figure 1.2: Comparison of formulations of Hatzikiriakos slip

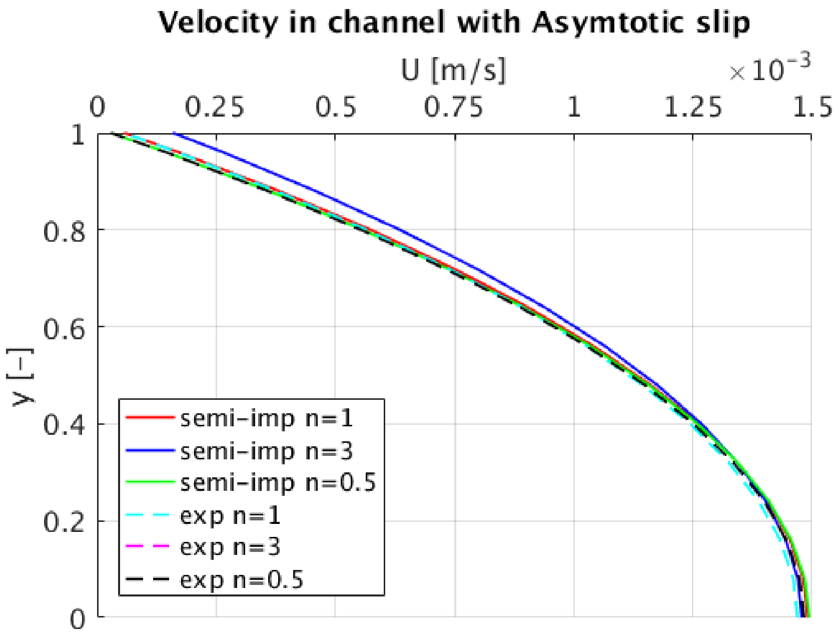

Figure 1.3: Comparison of formulations of Asymtotic slip

## 1.5 Scope for future work

The boundary condition applied through the semi-implicit formulation has been implemented which suits a very specific application with considerable assumptions. The major assumption is that the channel flow is oriented along the x-axis. If the channel is inclined at an angle, the realistic case involves more involved calculations for the implementation of bisection method using the transformation of co-ordinates. This is because, in the semi-implicit formulations, the wall normal gradient has to be found out and the components of gradients of snGrad() do not naturally align with the physical tangential and normal directions of the flow.

More advanced studies could also be made by the assumption of complex rheological models for the material flowing through the channel. As mentioned earlier, these slip boundary conditions are very relevant in the polymer extrusion industries and consideration of material model could be a possible step advancing from the current project.

## 1.6 Study Questions

- State the formulational difference between explicit Navier slip model and semi-implicit Navier slip model
- How to avoid compilation of boundary condition class for variation with model parameters and perform it from case setup ?
- What are the possible flow situations that could lead to slip flows ?
- What is the necessity to apply the boundary conditions to the individual faces of the boundary patch and not to the patch at once ?
- Why is the explicit formulation performed with applying the boundary condition to the boundary patch. Does applying applying to the individual faces produce the same result ?
- What is the requirement of using Boolean operator in the semi-implicit formulation ?
- What does the snGrad() calculate ? And what change should be made to suit for the semi-implicit formulation ?
- What is the advantage of having an altered guess limits in the semi implicit formulation for Hatzikiriakos and Asymtotic model ?
- Explain why the member data are modified for different models of same formulation but kept the same for different formulations of the same model ?

## 1.7 References

- L.L. Ferras, J.M. Nobrega and F.T. pinho. Analytical solution for Newtonian and inelastic non-newtonian fluids with wall slip. 2012
- L.L. Ferras, J.M. Nobrega and F.T. pinho. Implementation of slip boundary conditions in the finite volume method: new techniques. 2013