Cite as: Segersson, D.: A tutorial to urban wind flow using OpenFOAM. 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

A tutorial to urban wind flow using OpenFOAM

Developed for OpenFOAM-plus v.1706
Requires: python with matplotlib

Author:
David Segersson
Stockholm University & Swedish Meteorological and Hydrological Institute
david.segersson@smhi.com

Peer reviewed by:
Elin Olsson
Mohammad Arabnejad

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

December 21, 2017

Learning outcomes

The reader will learn:

How to use it:

- How to set-up boundary conditions for simulations of wind in an urban environment.
- How to apply and evaluate rough wall-functions for the atmospheric boundary layer.

The theory of it:

- How to model ground roughness consistently with inlet boundary conditions.
- How to represent tree canopy using a porosity model.

How it is implemented:

- How run-time selectable source-terms can be added using the fvOptions framework.
- How to implement a custom run-time selectable option for adding source-terms to the momentum and turbulence equations.
- How to set a varying roughness length within a wall patch.

Prerequisites

The reader is expected to know the following in order to get maximum benefit out of this report:

- Basic boundary layer meteorology.
- Fundamentals of Computational Fluid Dynamics.
- Basic C++
- Run standard tutorials, e.g. the damBreak tutorial.

Suggested litterature for further reading:

- Franke J., Hellsten A., Schlunzen H., Carissimo B. Best practice guideline for the CFD simulation of flows in the urban environment. Cost action 732. 1 May 2007.
- Hargreaves D.M., Wright N.G., On the use of the k-epsilon model in commercial CFD software to model the neutral atmospheric boundary layer. J. Wind Eng. Ind. Aerodyn. 95 (2007) 355-369.
- Dalpé B., Masson C., Nunmerical Simulation of wind flow near a forest edge. J. Wind Eng. Ind. Aerodyn. 97 (2009) 228-241.

Contents

1 Introduction **5**

2 Theory **7**
2.1 Boundary conditions . . . 7
2.1.1 Inlet boundary conditions . . . 7
2.1.2 Rough wall functions for the ABL . . . 8
2.1.3 Top boundary condition . . . 8
2.2 Influence of tree canopy . . . 8

3 Implementation **10**
3.1 The fvOption framework . . . 10
3.1.1 The options and optionList classes . . . 10
3.1.2 The option class . . . 11
3.2 Creating a new option . . . 12
3.2.1 Object oriented design . . . 12
3.2.2 The landuseClass class . . . 12
3.2.3 The Raster class . . . 14
3.2.4 The canopySource base class . . . 14
3.2.5 Initializing the canopySource option . . . 18
3.2.6 Calculating patch distance . . . 19
3.2.7 Defining a canopySource . . . 19
3.2.8 Adding source terms . . . 20
3.2.9 Implementation of non-uniform roughness . . . 22
3.3 Compile library and solver . . . 22

4 Tutorial **24**
4.1 Pre-processing . . . 24
4.1.1 Getting started . . . 24
4.1.2 Meshing . . . 24
4.1.3 Boundary and initial conditions . . . 25
4.1.4 Discretization and solver settings . . . 26
4.1.5 Monitoring residuals . . . 26
4.1.6 Sampling . . . 27
4.2 Running the case . . . 28
4.3 Verifying horizontal homogenity . . . 28
4.4 Modeling the effect of canopy . . . 30
4.4.1 Applying the custom option . . . 30
4.4.2 Running the case . . . 30
4.4.3 Checking the result . . . 30
4.4.4 Comparison with measurements . . . 31
4.4.5 Non-uniform canopy height . . . 32

5 References **34**

6 Study questions **35**

Chapter 1

Introduction

This tutorial describes how to simulate a neutral ABL (Atmospheric Boundary Layer) in an urban environment using the steady state incompressible solver simpleFoam. Wind flow in an urban environment is typically affected by the buildings, varying topography, vegetation, varying surface roughness and moving vehicles. We will here focus on how vegetation and surface roughness can be described.

Several commercial CFD codes (e.g. CFX, Fluent) have been shown unable to maintain standard atmospheric wind speed and turbulence profiles over a flat terrain with homogenious roughness (using the standard k-Epsilon model and wall-functions (Hargreaves & Wright, 2006). This is due to inconsistent formulations of the inlet boundary conditions and the wall functions used. A distance of a few hundred meters upstream of the studied geometry is normally included in the computational domain, which is enough for the original inlet profile to change and diverge from the intended before reaching the center of the domain. A simple remedy for this is to use a shorter distance upstream of the ostacle. However, for larger obstacles or domains with varying topografy this often not possible. This can be a problem for example when comparing with measurements and trying to apply specific approaching wind conditions. Also, it makes it difficult to separate changes in the flow caused by the studied geometry, e.g. a building, from changes caused by this imbalance in boundary conditions. OpenFOAM includes wall functions based on Hargreaves & Wright (2007) which should be consistent with the inlet boundary conditions for atmospheric flow of Richards and Hoxey (1993), that are also available in OpenFOAM. As part of the tutorial, we evaluate the ability of these boundary conditions in OpenFOAM to maintain the inlet profiles over a longer distance.

The tutorial also contains a description of how flow through tree canopy can be described by adding source-terms to momentum and turbulence transport equations. A complete implementation of run-time selectable source-terms is presented and it is shown how they can be applied to represent varying surface roughness and canopy source-terms in a flexible way.

For the evaluation of tree canopy source-terms, the results are compared to field measurements by Irvine & Gardiner (1997). The measurements are made at a forest edge, with four meteorological masts, carrying three anemometers each. For each mast, the anemometers are placed at heights 0.5 h, 1.0 h and 2.0 h, where h is the average tree height (h=7.5m). The masts are placed at x=-6.1h, at the forest edge (x=0 h), x=3.6h and x=14.5h. Statistical profiles are provided for wind speed and it's standard deviation in the main horizontal wind direction, σ_u, and vertically σ_w. The profiles are normalized by the friction velocity, $u*$ at mast 1 on a height of 2h. There are no measurements of standard deviation in the cross-wind direction. To allow calculation of turbulence intensity for comparison with modelling results, a relation for neutral atmosphere, $\sigma_v = 2.1u*$ is used following Dalpé and Masson (2007).

The model geometry consists of a 1 km long and 500 m high rectangular 2D domain with flat ground. In figure 1.1, the domain is presented together with the meteorological masts.

The value of the roughness length, z_0, is estimated from statistical wind profiles given in Irvine & Gardiner (1997). The statistical profiles represent 3 experiment runs, all close to neutral conditions and with wind direction close to perpendicular to the forest edge. The roughness length is estimated

to 0.06 m at mast 1, using the measurements at a height of 2h.

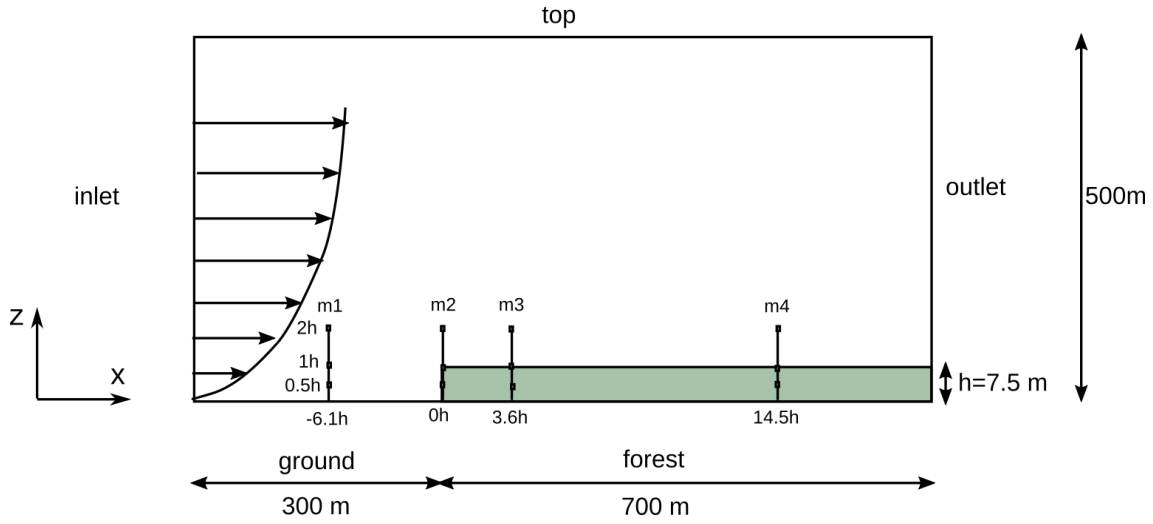

Figure 1.1: Geometry of the Irvine test case for tree canopy flow. The masts are marked m1-m4. The green block represents the forest canopy.

Chapter 2

Theory

In general, Large Eddy Simulation (LES) has been proven superior to RANS when describing the ABL, but for many p^s, RANS still remains more common. For some applications, such as pedestrian wind comfort, RANS has been proven to capture the main flow features of interest, making the considerably larger computational effort of LES hard to justify. In this tutorial, RANS and the $k-\epsilon$ turbulence model are applied. The standard $k-\epsilon$ is still the most popular turbulence model for describing the neutral ABL using RANS. The otherwise popular k-Omega SST model has not been widely adopted for atmospheric flows. A reason might be that one of the main benefits of the k-Omega model is the ability to describe the low Reynolds number flow very close to walls. The ground surface in atmospheric flows is usually very rough, making it impossible to resolve the flow in the near-wall region anyway.

The size of the modelling domain when studying wind flow in urban environments is typically in the order of 1 km^2 horizontally and has a height a few hundred meters up to 1 km. At this scale it is impossible to resolve individual trees and small obstacles in the mesh, making it necessary to parameterize their effect on the flow. The ground surface roughness is usually handled using wall functions adapted for rough walls. Larger obstacles that are not resolved by the mesh, such as trees and moving vehicles, are usually described as a porous media.

2.1 Boundary conditions

2.1.1 Inlet boundary conditions

The approaching wind profile for a neutral ABL is often modelled using boundary conditions suggested by Richards and Hoxey (1993). In the formulation of their boundary conditions Richards and Hoxey assume

1. zero vertical velocity.
2. pressure is constant in vertical and streamwise directions.
3. constant shear stress in the boundary layer.
4. the turbulent kinetic energy, k and dissipation rate, ϵ, satisfy their transport equations.

The velocity and turbulence profiles are

$$u = \frac{u_*}{\kappa} ln\left(\frac{z+z_0}{z_0}\right) \tag{2.1}$$

$$k = \frac{u_*^2}{\sqrt{C_\mu}} \tag{2.2}$$

$$\epsilon = \frac{u_*^3}{\kappa\,(z + z_0)} \tag{2.3}$$

where κ is von Karmans constant, $u*$ is friction velocity [ms[1]] and z_0 is the roughness length [m]. The k and ϵ conservation equations, under the assumptions listed above, are satisfied when

$$\sigma_\epsilon = \frac{\kappa^2}{(C_{\epsilon 2} - C_{\epsilon 1})\sqrt{C_\mu}}$$

where σ_ϵ, $C_{\epsilon 2}$, $C_{\epsilon 1}$ and C_μ are coefficients of the $k-\epsilon$ model. Using the standard values of the k-ϵ model, apart from σ_ϵ, which is given the value 1.11 according to Hargreaves & Wright (2007).

2.1.2 Rough wall functions for the ABL

Wall-functions adapted for rough walls are usually applied to represent different types of ground surfaces. There are a couple of different wall functions available for rough walls in OpenFOAM. The rough wall functions often used in industrial applications describes the roughness elements using the equivalent sandgrain roughness, k_s (Nikuradse, 1933), which refers to the diameter of sand-grains resulting in the same effect on flow as the real roughness. This wall-function formulation is developed for pipe-flow, with small homogenious roughness elements. To describe the much larger roughness elements encountered in atmospheric flow, it can be necessary to use a k_s of more than 2 meters (it can be shown that $k_s = 20z_0$). Since the wall-function formulation requires the distance from the wall to the first cell-center to be larger than k_s, this makes it impossible to resolve flow at pedestrian level. For atmospheric flows, the roughness is instead parameterized using the z_0 roughness length, which is a parameter of the standard logarithmic wind profile (WMO, 2008). The roughness length is the height at which the logathimic wind profile reaches zero. It is a length scale that is related to the roughness height by approximately a factor 0.1.

A difference in the formulation is that the boundary condition is applied at at height z0 above ground and not exactly at the surface (the ground distance is given by z0 + z), making it possible to represent larger roughness elements.

In OpenFOAM, rough wall-functions for the ABL are available using the nutkAtmRoughWall-Function for ν_t (tubulent viscosity), kqRWallFunction for k and epsilonWallFunction for ϵ. The main reference of the implementation is Hargreaves & Wright (2007).

It should be mentioned that the y^+ range of 30-300 that is usually required for a successful usage of wall-functions does not hold when simulating the ABL using this methodology. According to best practice guidelines for describing flow in the built environment (Franke et al. 2007) the height of the first cell should instead be a few decimeters (0.2 m).

2.1.3 Top boundary condition

In order to maintain the inlet profiles over a longer distance, a fixed shear stress should be applied at the top boundary (Richards & Hoxey, 1993). There is a fixedShear boundary condition available in OpenFOAM, but since this mainly affect the profile at the top of the domain, which is not of primary interest in this tutorial, a slip boundary condition is used at the top boundary.

2.2 Influence of tree canopy

The effect of vegetation on the flow is usually modelled by treating the canopy by adding source-terms to momentum and turbulence equations. There are several models available for this purpose, e.g. Dalpé and Masson (2007) and Svensson & Häggkvist (1990). The model by Dalpé and Masson (2007) is presented in equations 2.4-2.6:

$$S_u = -\rho C_d \alpha |U| U \tag{2.4}$$

$$S_k = \rho C_d \alpha \big(\beta_p |U|^3 - \beta_d k |U|\big) \tag{2.5}$$

$$S_\epsilon = \rho C_d \alpha \frac{\epsilon}{k} \big(C_{\epsilon 4} \beta_p |U|^3 - C_{\epsilon 5} \beta_d k |U|\big) \tag{2.6}$$

where S_u, S_k and S_ϵ are source-terms on the right-hand side of transport-equations for momentum, k and ϵ respectively, α is LAD (Leaf Area Density) and C_d is the tree canopy drag coefficient. The β_p (=1.0), β_d (=5.03), $C_{\epsilon 4}$ (=0.78) and $C_{\epsilon 5}$ (=0.78) are empirical coefficients. In the incompressible solver simpleFoam, the density has been eliminated from the equations (the dynamic pressure p/ρ is used). In this case density is removed from the source-terms.

LAD [m^{-1}] represents the total leaf area per m^3 and varies within the tree canopy. Typically, different tree species have different vertical profiles of LAD. It can be difficult to find values on LAD for a specific type of tree. A more common value is LAI (Leaf Area Index), which is the integral of LAD from ground to the canopy height. The drag coefficient of a tree also varies between species. Dalpé & Masson (2007) uses a value of 0.2, which is applied here also.

Chapter 3

Implementation

3.1 The fvOption framework

Source-terms, constraints and corrections can be added to solvers in OpenFOAM without customising and recompiling the solver. This is made using the fvOptions framework. This framework will be used to create a custom option for canopy related source-terms. Options are activated and configured through the dictionary `fvOptions` which is found in the constant directory of the case. The fvOtions framework was introduced in OpenFOAM 2.2. Before, almost every solver of OpenFOAM used to have multiple versions to allow for example MRF (Moving Reference Frames) and porous flow. There are a number of options included with OpenFOAM, a few examples are:

- `semiImplicitSource`
- `actuationDiscSource`
- `meanVelocitySource`
- `explicitPorositySource`
- `buoyancyForce`

3.1.1 The options and optionList classes

Examples of run-time selectable options available in OpenFOAM are found in

`$FOAM_SRC/fvOptions/sources.`

To understand how options are created it is valuable to study the base classes `options`, and `optionList`. The `options` and `optionList` classes take care of the run-time selection and instatiation of the different options. The classes are found in

`$FOAM_SRC/finiteVolume/cfdTools/general/fvOptions`

Each solver using the fvOptions framework includes the file `createFvOptions.H` where an object named `fvOptions` is created as

```
fv::options& fvOptions(fv::options::New(mesh));
```

The `New` function is a static member of the class `options`, defined in

`$FOAM_SRC/finiteVolume/cfdTools/general/fvOptions/fvOptions.C`

From this function, the constructor of the options class is called, which in turn calls the constructor of the base class `optionList`. In the `optionList` constructor, the member function named `reset` is called. In this function the individual options are instantiated by calling the static member-function `New` of the `option` class, which identifies and calls the constructors of the different options defined in the `constant/fvOptions` dictionary. The order in which the functions are called is shown in figure 3.1. The inheritance graph of the options class is given in figure 3.2.

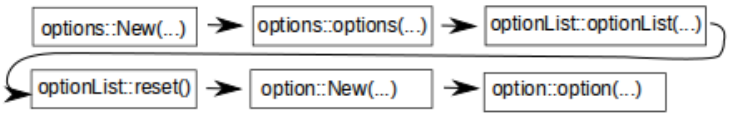

Figure 3.1: Base class functions called when creating the fvOptions object.

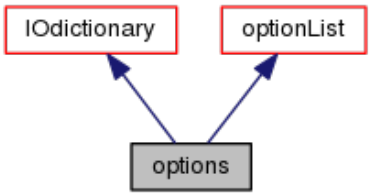

Figure 3.2: Inheritance graph of the options class.

3.1.2 The option class

The option class is an abstract base class with virtual functions for adding source-terms, constraints and corrections. This class defines how solvers and turbulence models should call the option to add source-terms or apply constraints and corrections. The inheritance graph of the options class is given in figure 3.3. As seen from this graph, the option class is the base for a few of the options included in OpenFOAM. For options that need to be applied for a cell-set instead of all cells or only for a given duration, the class `cellSetOption` that inherits the `option` class serves as the base class.

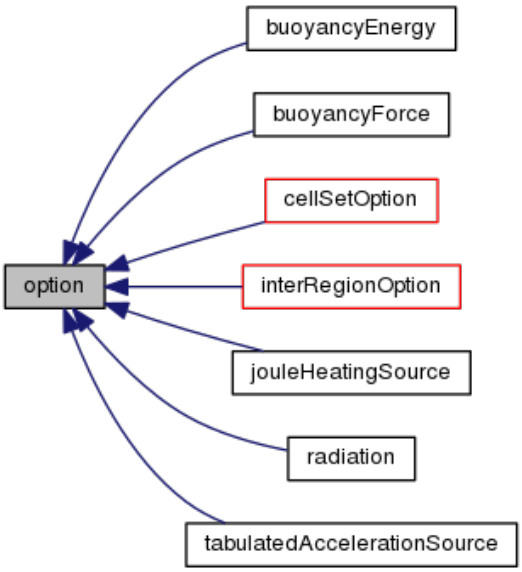

Figure 3.3: Inheritance graph of the option class.

3.2 Creating a new option

Tree canopy is described using a porosity concept. There are different porosity models for industrial applications implemented into OpenFOAM, but the models used for tree canopy are quite different from these and there is little to gain from basing an implementation on the existing porosity models. Instead we choose to implement a completely new option type.

The desired features of the new option are:

- allow a spatially varying landuse[1]
- each type of landuse should allow specifying a roughness length and a vertical LAD-profile
- each type of landuse should be applicable for varying tree canopy height
- allow specifying landuse patch-wise or via a separate GIS raster file
- allow specifying tree canopy-height per landuse or via a separate GIS raster file
- allow writing fields to disk for visualization
- allow parallel calculations

3.2.1 Object oriented design

The following classes are defined:

- `canopySource` is defined to handle initialization of fields and variables required to describe the landuse. This class inherits the `option` base class.
- `landuseClass` is defined to represent properties of a landuse type and to provide some member functions to process and access the different landuse parameters.
- `Raster` is provided to handle the reading of landuse type and canopy height from the ESRI ascii grid raster format.
- `groundDist` is provided for estimation of distance to specified patches.
- `dalpeMassonCanopySource` inherits the canopySource class and defines source-terms for turbulence according to Dalpé & Masson (2009).

By creating a separate class containing only the source-term definitions, different models can easily be added. All source-code is provided together with this report. The different classes are also described below.

3.2.2 The landuseClass class

A separate class named `landuseClass` is implemented to represent a specific type of landuse. The header file of the class is given on the following page.

[1] Landuse refers to the land cover, e.g. crops, forest, water or similar.

```
#ifndef landuseClass_H
#define landuseClass_H

#include "label.H"
#include "scalar.H"
#include "dimensionedScalar.H"
#include "dimensionSet.H"

namespace Foam
{
/*---------------------------------------------------------------------------*\
                           Class landuseClass Declaration
\*---------------------------------------------------------------------------*/
class landuseClass {

  private:
    label code_;    // unique id of the landuse class
    word name_;     // name of the landuse class
    scalar Cd_;     // drag coefficient
    scalar LAI_;    // Leaf Area Index (used if LADmax is not specified)
    scalar z0_;     // roughness length
    scalar height_; // canopy height
    scalar LADmax_; // max of Leaf Area Density
    scalarList LADProfile_; // normalized profile of LAD

  public:
    // Constructors
    landuseClass();
    landuseClass(const dictionary &dict, word name);

    // Destructor
    ~landuseClass();

    // member functions
    const scalarList& LADProfile() {
      return LADProfile_;
    }

    scalar LAD(scalar z, scalar treeHeight);
    void LADmaxFromLAI();
    scalar integrateLAD();

    // access
    const word& name() { return name_; }
    const label& code() { return code_; }
    const scalar& Cd() { return Cd_; }
    const scalar& LAI() { return LAI_; }
    const scalar& z0() { return z0_; }
    const scalar& height() { return height_; }
    const scalar& LADmax() { return LADmax_; }
};
}
#endif /*landuseClass_H*/
```

The `code` attribute is an scalar (an integer) unique for each landuse class. This is used later to allow specifying landuse classes in a `scalarField`. The class also contains member functions to calculate LAI by integrating the vertical LAD profile and to set LADmax based on LAI. This is included since often only LAI is known for a type of vegetation, allowing LADmax to be calculated if a normalized vertical LAD profile is assumed. There is also a function to calculate LAD for a given height, using the normalized LAD profile and the LADmax value. To facilitate instatiation of landuse classes from settings in the fvOptions dict, a constructor is implemented that reads required parameters from a sub-dictionary with a given name:

```
landuseClass::landuseClass(const dictionary& dict, word name) {

    dictionary landuseClassDict(dict.subDict(name));
    landuseClassDict.lookup("code") >> code_;
    name_ = name;
    Cd_ = landuseClassDict.lookupOrDefault("Cd", 0.2, false, false);
    height_ = landuseClassDict.lookupOrDefault("height", 0.0, false, false);
    z0_ = landuseClassDict.lookupOrDefault("z0", 0.001, false, false);
    LAI_ = landuseClassDict.lookupOrDefault("LAI", 0.0, false, false);
    LADmax_ = landuseClassDict.lookupOrDefault("LADmax", -1.0, false, false);

    if (landuseClassDict.found("LADProfile"))
      landuseClassDict.lookup("LADProfile", false, false) >> LADProfile_;

    if (LADmax_ == -1.0) {
      LADmaxFromLAI();
    }
  }
```

At the end of the constructor, LADmax is estimated from LAI if it has not been set (has the initial value -1). This is made by incrementing LADmax gradually, until the integral of the vertical profile matches the provided LAI.

The function `scalar LAD(scalar z, scalar treeHeight)` scales the provided LAD-profile to match the specified tree height and LADmax and then extracts LAD at the height z.

3.2.3 The Raster class

To enable reading landuse and canopy height from GIS rasters a separate class representing the ESRI Ascii raster format is provided. The implementation does not make use of the OpenFOAM libraries and the implementation is therefore not described in this tutorial.

3.2.4 The canopySource base class

The `canopySource` class will serve as a base class for various implementations of tree canopy source terms, but can also be used by itself to only include source terms for momentum. These are similar for most canopy models and are therefore implemented in the base class. For solvers not including density the source term is given below

```
void Foam::fv::canopySource::addSup
(
    fvMatrix<vector>& eqn,
    const label fieldi
)
{
    const volVectorField& U = eqn.psi();
    const volScalarField& canopy = canopy_;
```

```
    fvMatrix<vector> S_canopy
    (
       fvm::Sp(canopy * mag(U), U)
     );
    eqn -=  S_canopy;
}
```

For turbulence source terms, there are several different variants that have been published and two of them are implemented here: Svensson & Häggkvist (1990) and Dalpé & Masson (2007). Since these inherit the `canopySource` base class, the only functionality they contain is the implementation of the source-terms. The Svensson & Häggkvist (1990) model is only provided to demonstrate the extendability of the design, this model is not further described in this tutorial.

As a starting-point for our custom option, the `tabulatedAccelerationSource` was copied and renamed to

`$WM_PROJECT_USER_DIR/applications/src/fvOptions/sources/canopySource`

The files in the directory were renamed to `canopySource.H` and `canopySource.C`. The content of `canopySource.H` is given on the next two pages.

```
#ifndef canopySource_H
#define canopySource_H

#include "fvOption.H"
#include "dimensionedTypes.H"
#include "DynamicList.H"
#include "landuseClass.H"
#include "Raster.H"
#include "nutkAtmRoughWallFunctionFvPatchScalarField.H"
#include "groundDist.H"
#include "wallFvPatch.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
namespace Foam
{
namespace fv
{
/*---------------------------------------------------------------------------*\
                Class canopySource Declaration
\*---------------------------------------------------------------------------*/
class canopySource
:
    public option
{
protected:
        // member data for landuse
        wordList sourcePatches_;
        autoPtr<volScalarField> canopy_;
        HashTable<landuseClass, label> landuseTable_;
        HashTable<landuseClass, label> patchLanduseTable_;

        // read landuse, z0 and LAD from disk if they are present
        Switch readFromDisk_;

        // read landuse from raster instead of specifying per patch
        Switch readLanduseFromRaster_;
        Raster landuseRaster_;

        // read canopy height from raster instead of specifying per landuse class
        Switch readCanopyHeightFromRaster_;
        Raster canopyHeightRaster_;

        // translation vector from coordiantes in raster to coordinates in mesh
        vector translateRaster;

        Switch writeFields_;

        // if variables are present on disk, they are read instead of calculated
        // this allows setting the data with external programs
        bool LAD_from_disk_;
        bool landuse_from_disk_;
        bool z0_from_disk_;

        //- Source terms to momentum equation
        // (for solvers with and without explicit density)
```

```
        void addSup
        (
            const volScalarField& rho,
            fvMatrix<vector>& eqn,
            const label fieldi
        );
        void addSup
        (
            fvMatrix<vector>& eqn,
            const label fieldi
        );

        //- Disallow default bitwise copy construct
        canopySource(const canopySource&);
        //- Disallow default bitwise assignment
        void operator=(const canopySource&);

        void checkData() const;

        // Member Functions
        Raster readRaster(fileName rasterPath);
        void calculatePatchDistance(label patch, volScalarField &d);
        void setPatchLanduse(label patch, volScalarField &landuse,
                             volScalarField &LAD, volScalarField &z0,
                             volScalarField &nut, volScalarField &d);
        void calculateCanopy();
        void readLanduseClasses();
public:
       //- Runtime type information
       TypeName("canopySource");
       // Constructors
       canopySource
       (
          const word& name,
          const word& modelType,
          const dictionary& dict,
          const fvMesh& mesh
       );
       //- Destructor
       virtual ~canopySource()
       {}
       // Member Functions
       //- Read dictionary
       bool read(const dictionary& dict);
};
}
}
#endif
```

The definition of canopySource is more than 500 lines, making it too long to include as a whole in this document. Instead, only the most interesting parts are described and the rest of the code is provided separately.

3.2.5 Initializing the canopySource option

The run-time selection mechanism used for fvOptions end up calling the constructor of the selected option type. In the initialization of the canopySource, the constructor of the option base class is called, which initializes some basic attributes. One important example is the active_ attribute, which is read from the fvOptions dictionary. This attribute, which can be used to switch the effect of tree canopy on or off, is accessed through the active() member function and determines if the source term is active or not. Before reading further data, this is checked in the constructor. Thereafter, the member variable fieldNames_ of the option base class, that specifies for which fields the option should be applied, is set. The constructor is given below.

```
Foam::fv::canopySource::canopySource
(
    const word& name,
    const word& modelType,
    const dictionary& dict,
    const fvMesh& mesh
)
:
  option(name, modelType, dict, mesh)
{
  if (active()) {
    fieldNames_.setSize(3);
    fieldNames_[0] = word("U");
    fieldNames_[1] = word("k");
    fieldNames_[2] = word("epsilon");
    applied_.setSize(fieldNames_.size(), false);
    read(dict);
  }
}
```

At the end of the constructor, the read function is called. This function contains more specific input and processing of data. The complete source code of the read function is too long to present here but is included in the source code included with this tutorial. The main steps are

1. read list of source patches for which landuse (tree canopy and roughness length) will be defined
2. determine if landuse and canopy height should be specified patch wise in the dictionary or read from a separate GIS raster file.
3. the function `readLanduseClasses` is run. This function reads all landuse classes defined in `constant/fvOptions` and stores them in a hash table with the landuse code as hash key.
4. If reading from rasters is chosen, raster objects for landuse and/or canopy height are instantiated and stored as member variables of the `canopySource` object.
5. if not reading from rasters, landuse should be specified for each source patch. These are stored in a hash table to facilitate quick lookups.
6. read all landuse classes defined in the `fvOptions` dictionary
7. check data validity
8. run `calculateCanopy`, which creates and sets fields used to describe the landuse.

The `calculateCanopy` member function creates the volScalarFields `landuse`, `LAD` and `z0`. If the fields already exist, they are read from disk. A reference to the `nut` field is also retrieved from the object registry. The landuse field contains landuse codes corresponding to the landuse class in each grid cell. For patches that are not specified as source patches (non-ground patches) and for internal cells above the tree canopy, the value is initialized to -1. The `LAD` field contains the leaf area density within the tree canopy and zero otherwise. The `z0` field contains the roughness length for the source patches and -1 otherwise. It should be remembered that the `z0` field is only created to allow visualization. After the fields have been created, the member functions `calculatePatchDistance` and `setPatchLanduse` are called for each of the source patches defined in `constant\fvOptions`. The member function `calculatePatchDistance` is described in the next section. The `setPatchLanduse` member function sets the values for fields `landuse`, `z0` and `LAD` at a patch and for the internal cells up to canopy height of the landuse applied at each patch face. The roughness length that is used in the calculations is set on the `z0` parameter of the `nutkAtmRoughWallFunction` patch field. As a last step, the member variable `canopy_` is calculated as the product of `LAD` and `Cd` of the corresponding landuse class. Since this product is constant, calculating it beforehand saves some computations.

3.2.6 Calculating patch distance

The distance to a specific patch is calculated using a so called `meshWave`. The algorithms for calculating distances are part of the OpenFOAM core library and can be found in directories

```
$FOAM_SRC/finiteVolume/fvMesh/wallDist/wallDist
$FOAM_SRC/finiteVolume/fvMesh/wallDist/patchDistMethods/meshWave
```

The provided code (`grounddistance.H` and `groundDistance.C`) has been modified from an earlier version of OpenFOAM and the implementation is not described in this tutorial.

3.2.7 Defining a canopySource

An example of the dictionary defining a `canopySource` is given below.

```
canopy
{
    type                        canopySource;
    active                      on;
    writeFields                 on;         // write to disk
    readLanduseFromRaster       on;         // read from raster
    readCanopyHeightFromRaster on;
    translateRaster             (0 0 0);         // translate raster
    sourcePatches               (ground forest); // patches to process

    // if landuse is not set from raster
    patchLanduse                (0 1);           // landuse code per patch

    LADProfile (0.05 0.1 0.15 0.35 1.1 0.9 0.5 0.2 0.15 0.05 0.01);  // default

    landuse
    {

        low_birch                               // name of landuse class
        {
            code 1;                             // id of landuse class
            Cd 0.2;                             // drag coefficient
            LADmax 1.2;                         // Maximum Leaf Area Density [m^-1]
            z0 0.06;                            // roughness length [m]
```

```
            // used if LADmax is not specified
            LAI 2.15;                           // Leaf Area Index

            // if not read from raster
            height 7.5;                         // tree canopy height

            // Vertical profile of Leaf Area Density
            // first value is closest to ground
            // each value represents an equal share of the tree height
            // e.g. for a 4 m tree and 4 values, each value will represent 1 m
            // values are scaled so that the highest will correspond to LADMax
            LADProfile ( 0.05 0.1 0.15 0.35 1.1 0.9 0.5 0.2 0.15 0.05 0.01 );
        };

        grass
        {
            code 0;
            Cd 0.2;
            LAI 0;
            z0 0.06;
            height 0;
        };

    };
}
```

3.2.8 Adding source terms

The option base class implements a few virtual functions defining the interface with solvers making use of the option. For source terms, the virtual functions all have the name `addSup`.

Source terms can either be implicit or explicit. Implicit means that they are added to the coefficient matrix on the left-hand side of the linearized equation system, while explicit means they are added to the right-hand side. For improved convergence it beneficial to add an as large part as possible implicitly. However, when the source-term is linearized as $S = S_C + S_P T_P$, where S_C and S_P are constants and T_P is the variable that is solved for, the rule is that S_P should always be less than or equal to zero or stability will be reduced (Patankar, 1980).

In OpenFOAM an implicit source-term is added using the fvm::Sp( ) function. There is also a function fvm::SuSp that tries to determine if source term on the right-hand side can be linearized with a negative S_P. If so, the source term is added to the diagonal of the coefficient matrix and otherwise it's directly added to the right-hand side as an explicit source-term.

The member function `addSup` exists in a few different flavors to allow being called for incompressible and compressible solvers (with or without rho (density) among the arguments) as well as for different types of fields (`fvMatrix$<scalar>$`, `fvMatrix<vector>`, etc.). In this case we implement `addSup` for incompressible and compressible flow and for both vector fields and scalar fields following Dalpé and Masson (2007). The momentum source term is defined the same way for most canopy models and is therefore added to the `canopySource` base class. The mathematical formulation is given in 2.1. Since the product before U is negative, we can safely use `fvm::Sp` to add the source-term implicitly.

As mentioned in the description of the `calculateCanopy` member function, the product $C_d\alpha$ always appear together in the calculations. Therefore, this product is calculated once and then stored as a member variable of the option with the name `canopy_` with units [m^{-1}]. Also, for the incompressible version, `rho` is excluded.

`void Foam::fv::dalpeMassonCanopySource::addSup`

```
(
    fvMatrix<vector>& eqn,
    const label fieldi
)
{
    const volVectorField& U = eqn.psi();
    const volScalarField& canopy = canopy_;

    fvMatrix<vector> S_canopy
    (
       fvm::Sp(canopy * mag(U), U)
     );

    eqn -=  S_canopy;
}
```

The mathematical formulation of source-terms for k and ϵ are given in 2.2 and 2.3. Both source-terms are implemented in the same `addSup` function and the choice of which term to apply is made depending on the name of the variable that is solved for.
For the `k` equation source-term, the first part should be added explicitly. For the second part, the coefficient before the `k` will allways be negative, allowing us to add this part implicitly.

For the `epsilon` equation, the sign of the expression within the bracket depends on the magnitude of `U` and `k` and it is therefore not evident if the source term should be added implicitly or explicitly. However, testing reveals that stability is significantly improved when the term is added implicitly. The implementation is given below.

```
void Foam::fv::dalpeMassonCanopySource::addSup
(
    fvMatrix<scalar>& eqn,
    const label fieldi
)
{
  const volScalarField& canopy = canopy_;

  if (eqn.psi().name() == word("k")) {

    const volScalarField& k = eqn.psi();
    const volVectorField& U = mesh_.lookupObject<volVectorField>("U");

    fvMatrix<scalar> Sk
    (
     betaP_*canopy*pow(mag(U),3) - fvm::Sp(betaD_*canopy*mag(U), k)
    );

    eqn +=  Sk;
  }
  else if (eqn.psi().name() == word("epsilon")) {

    const volScalarField& epsilon = eqn.psi();
    const volScalarField& k = mesh_.lookupObject<volScalarField>("k");
    const volVectorField& U = mesh_.lookupObject<volVectorField>("U");

    fvMatrix<scalar> Sepsilon
    (
     fvm::Sp(canopy/k*(C4_*betaP_*pow(mag(U),3) - C5_*betaD_*k*mag(U)), epsilon)
```

```
    );

    eqn += Sepsilon;
  }
}
```

3.2.9 Implementation of non-uniform roughness

When specifying the `nutkAtmRoughWallFunction` boundary condition that is applied at the ground patch of the nut field, the roughness length `z0` is a required parameter. The type of this parameter in the `nutkAtmRoughWallFunction` class is a `scalarField`, meaning that different values can be applied for the cell faces of a patch. However, there is no way to specify a scalarField for a parameter in the boundary condition subdictionary using the standard utilities. An intuitive way to allow non-uniform roughness to be applied at the ground patch, would be to create a new wall boundary condition with customized functionality to read roughness from a separate file. However, in order to keep all landuse properties at the same place and apply them using the same framework, the non-uniform roughness length is instead set from the custom option.

In the function `setPatchLanduse`, the fields `landuse`, `LAD`, `z0`, are set for a specific patch (see description under section "Initializing the canopySource option"). Landuse classes are either specified per patch, using the `patchLanduse` entry in the `fvOptions` dictionary, or read from a separate georeferenced raster file that is specified using the `landuseRasterFileName` entry in the same dictionary. There is also a switch called `readLanduseFromRaster` to determine which method to use. Reading from raster is supported since this is a commonly used format for landuse data. A class named `Raster` to describe the raster format is provided but not described in this tutorial.

In the function `setPatchLanduse`, the `z0` parameter (scalarField) of the nut field at the ground patch is accessed by:

```
Foam::nutkAtmRoughWallFunctionFvPatchScalarField& wallNut =
  refCast<Foam::nutkAtmRoughWallFunctionFvPatchScalarField>(nut.boundaryFieldRef()[patch]);
scalarField& nutZ0 = wallNut.z0();  // creating a reference to the z0 parameter
```

The `nut.boundaryFieldRef()[patch]` returns the base class `fvPatchScalarField`. Therefore, to be able to access the specific attributes of the `nutkAtmRoughWallFunctionFvPatchScalarField`, a cast is necessary. The scalarField `nutZ0` can then be modified freely and the parameter value will be written to disk when the `nut` field is written.

3.3 Compile library and solver

The same directory structure as for derived fvOptions in the OpenFOAM source code is used. The directory layout and source-files are

```
$WM_PROJECT_USER_DIR/applications/src
└── fvOptions
    └── sources
        ├── canopySource
        │   ├── canopySource.H
        │   ├── canopySource.C
        │   ├── landuseClass.H
        │   ├── landuseClass.C
        │   ├── Raster.H
        │   ├── Raster.C
        │   ├── groundDist.H
        │   └── groundDist.C
        └── dalpeMassonCanopySource
```

```
        dalpeMassonCanopySource.H
    dalpeMassonCanopySource.C
      senssonHaggkvistCanopySource
        svenssonHaggvistCanopySource.H
        svenssonHaggkvistCanopySource.C
    Make
      options
        files
```

The user src-directory is here placed under the applications directory to allow having all the user source code in the same repository for version-control. To compile the library:

```
cd $WM_PROJECT_USER_DIR/applications/src/fvOptions
wmake
```

The library is named libABLFvOptions, meaning that for the solvers to find it, the row

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

should be added to the controlDict.

Chapter 4

Tutorial

4.1 Pre-processing

As a basis for the test-case we will use the tutorial turbineSiting. We will then modify solvers, schemes and boundary conditions to comply with best practice guidelines for urban wind flow by Franke et al.(2007). The turbineSiting tutorial is a simple setup for incompressible RANS on a small hill with a turbine. We will replace the geometry with a 2D domain with flat ground.

4.1.1 Getting started

Copy the turbineSiting tutorial to the run directory and name it irvineForestEdge (the name is refering to the measurement data used to evaluate the tree canopy influence).

```
cp -r  $FOAM_TUTORIALS/incompressible/simpleFoam/turbineSiting \
       $FOAM_RUN/irvineForestEdge
cd $FOAM_RUN/irvineForestEdge
```

4.1.2 Meshing

The mesh is generated using blockMesh. The blockMeshDict is not included in this report but can be found in the system directory of the included test-case. Since it's a 2D domain, the back and front of the domain are set as empty in the blockMeshDict. The coordinate system will be set to x=0 at the forest edge. This means that the domain inlet will be at x=-300 and the outlet at x=700. The ground will be at z=0.

After generating the mesh using blockMesh, we will split the bottom patch to create a separate area named forest (see figure 1.1). This is done using two standard utilities of OpenFOAM: `topoSet` and `createPatch`. The `topoSet` utility is used to select the faces that will be part of the new forest patch and the `createPatch` utility defines a new patch from the set. Each of the utilities requires settings in dictionaries with the corresponding names (`topoSetDict` and `createPatchDict`). The `topoSet` utility allows the creation and manipulation of cellSets or faceSets using different operators. Here the operator `boxToFace` is used that selects cell faces using a bounding box. In order to select all faces of the ground patch (z<0.01) where there is forest (0<x<700), the `topoSetDict` should contain

```
actions
(
    {
        name    forest;     // name of the set to be created
        type    faceSet;    // type of set
        action  new;        // create a new set
        source  boxToFace;  // type of method to select faces
        sourceInfo          // bounding box within which faces will be selected
        {
          box  (0 -100 -100) (1000 100 0.01);
        }
    }

);
```

and the createPatchDict should contain

```
pointSync false;
patches
(
    {
        name forest;         // name of the new patch
        patchInfo
        {
            type wall;       // type of the new patch
        }
        constructFrom set;   // how the patch should be constructed
        set forest;          // name of the set to construct the patch from
    }
);
```

All three steps are run by issuing:

```
blockMesh
topoSet
createPatch -overwrite
```

4.1.3 Boundary and initial conditions

The boundary conditions of the original turbineSiting tutorial must be modified for the new geometry. The original boundary conditions are copied to a 0 time directory:

```
cp -r 0.orig 0
```

The boundary conditions from the original turbineSiting case are manually modified for each variable by:

1. Copying the bc entry for the terrain patch to the ground patch.
2. Renaming the terrain patch to forest.

The inlet boundary conditions should now be set according to table **??**. The side and top boundaries are the same for all fields and are therefore also included from a separate file: 0/include/sideAndTopPatches. The outlet conditions are set to inletOutlet for `k`, `epsilon` and `U`. For `p`, the outlet condition is `uniformFixedValue (0)`. The boundary condition for the ground and forest patches for the different fields are given in 4.2.

Table 4.1: Inlet boundary conditions.

Field	Boundary condition
U	atmBoundaryLayerInletVelocity
epsilon	atmBoundaryLayerInletEpsilon
k	atmBoundaryLayerInletK
p	zeroGradient

Table 4.2: Ground boundary conditions.

Field	Boundary condition
U	atmBoundaryLayerInletVelocity
epsilon	atmBoundaryLayerInletEpsilon
k	atmBoundaryLayerInletK
p	zeroGradient
nut	nutkAtmRoughWallFunction

In the turbineTutorial, to make sure parameters are set consistently for the different fields, the parameters are specified separately in `0/include/ABLconditions` and then included for relevant patches. The parameters are set according to data provided by Irvine & Gardiner (1997). Instead of using the roughness length reported it is calculated from the measured wind profiles a mast 1, on a height of 2h. Only the second and third run is used, since they are closer to neutral conditions.

```
z0       0.06;     // roughness length [m]
Uref     6.17;     // reference wind speed [m/s]
zRef     15;       // height of reference wind speed [m]
zGround  0;        // ground height [m]
flowDir  (1 0 0);  // direction vector of wind velocity
zDir     (0 0 1);  // vertical direction vector
```

Finally, the `constant/fvOptions` dictionary should be edited to remove the entries for `actuationDiskSource` named disk1 and disk2. In this file we will later add out own custom option for canopy.

4.1.4 Discretization and solver settings

The system/controlDict is modified to perform 2500 iterations, writing results every 500 iterations and cleaning away previous timesteps. The system/fvSchemes is modified to apply the second-order limitedLinear divergence schemes. Tolerance for solvers are reduced to 1e-12 and the relative tolerance to 0.01, to ensure that the solver does not finish before sufficient convergence is reached.

4.1.5 Monitoring residuals

To activate monitoring of the residuals for the case:

```
cp -r $FOAM_ETC/caseDicts/postProcessing/numerical/residuals system
```

Then edit the 'fields' entry in the file to also display residuals for k and ϵ and add the following entry to system/controlDict:

```
functions {
  #includeFunc residuals
};
```

4.1.6 Sampling

To compare with measurements from Irvine et al (1997), a function object is used to sample data from the converged solution. A starting point for the dictionary is copied by:

```
cp -r $FOAM_ETC/caseDicts/postProcessing/graphs/singleGraph system/verticalProfiles
```

The file is then edited as below:

```
h 7.5;  // tree height
z1 0;   // vertical profile start z
z2 500; // vertical profile end z
y 0.5;

// mast locations relative to the forest edge
mast1_dist #calc "-6.1*$h";
mast2_dist 0;
mast3_dist #calc "3.6*$h";
mast4_dist #calc "14.5*$h";

setConfig
{
    type    uniform;
    axis    distance;
    nPoints 2000;
}
fields  (U p k epsilon);

interpolationScheme cellPoint;
setFormat       raw;
type            sets;
libs            ("libsampling.so");
writeControl    writeTime;

sets
(
    inlet_profile
    {
        $setConfig;
        start (-295 $y $z1);
        end   (-295 $y $z2);
    }
    outlet_profile
    {
        $setConfig;
        start (690 $y $z1);
        end   (690 $y $z2);
    }
    mast1
    {
        $setConfig;
        start ($mast1_dist $y $z1);
        end   ($mast1_dist $y $z2);
    }
    mast2
    {
```

```
        $setConfig;
        start ($mast2_dist $y $z1);
        end   ($mast2_dist $y $z2);
    }
    mast3
    {
        $setConfig;
        start ($mast3_dist $y $z1);
        end   ($mast3_dist $y $z2);
    }
    mast4
    {
        $setConfig;
        start ($mast4_dist $y $z1);
        end   ($mast4_dist $y $z2);
    }
);
```

4.2 Running the case

The case is decomposed into 4 domains and calculations are then started in the background.

```
decomposePar
foamJob -parallel simpleFoam >& log &
```

Residuals are monitored using (see figure 4.1):

```
foamMonitor -l -r 2 postProcessing/residuals/0/residuals.dat
```

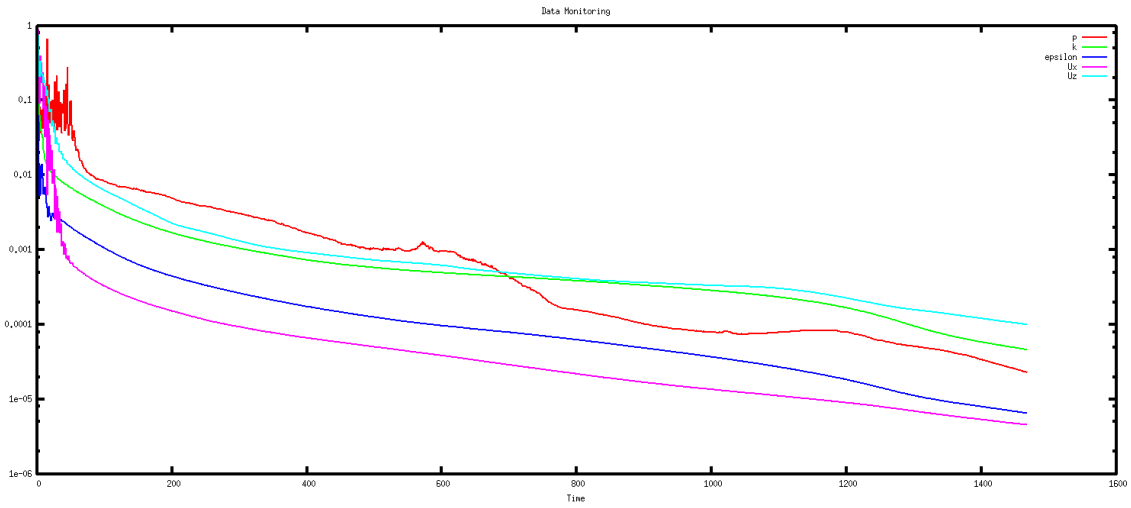

Figure 4.1: Monitoring residuals.

After calculations are finished, the case is reconstructed:

```
reconstructPar -latestTime
```

4.3 Verifying horizontal homogenity

An evaluation is performed to check if the inlet, ground and top boundary conditions are consistent. A simulation is run through the empty domain and the vertical profile at the inlet is then compared

to the vertical profile at the outlet. Vertical profiles are extracted by executing the functionObject verticalProfiles as:

```
simpleFoam -postProcess -func verticalProfiles -latestTime
```

A simple plotting script called `plot_horizontal_homogenity.py` is provided together with the tutorial. The plotting program is executed directly in the terminal and automatically reads the vertical profiles at inlet and outlet from the postProcessing directory. The profiles plotted are shown in figure 4.2. For wind speed and turbulence dissipation rate, the inlet profiles are preserved well throughout the domain. However, for `k`, the inlet profile seems over-estimated in comparison to the profile from Richards & Hoxey (1993). Also, the profiles changes significantly along the domain. Hargreaves & Wright (1993) shows similar deviations using standard wall-functions in Fluent and CFX. However, given that the implementation in OpenFOAM is based on their findings, it is surprising that the inlet profiles for k are not better preserved in OpenFOAM. More investigations are required to identify the exact reason for this.

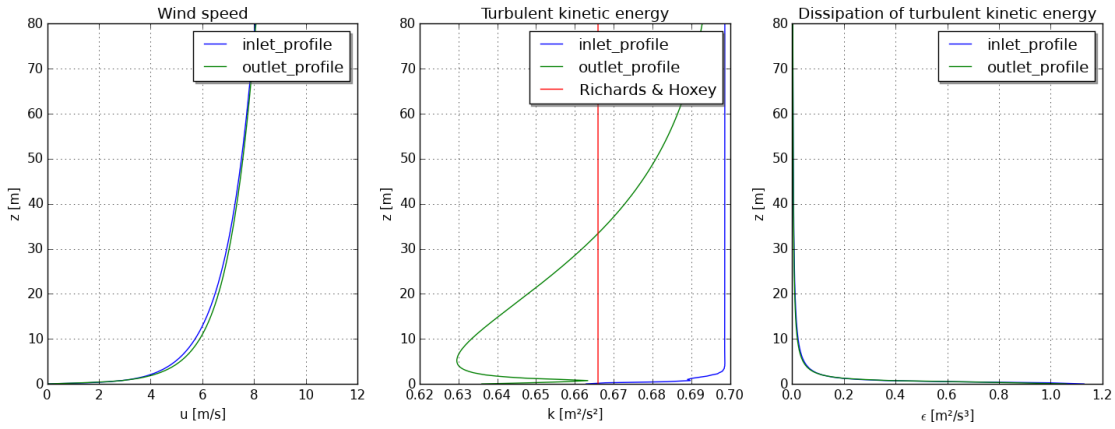

Figure 4.2: Evolution of inlet profiles through the 1km domain.

4.4 Modeling the effect of canopy

4.4.1 Applying the custom option

The dictionary constant/fvOptions should be created with the following content:

```
canopy
{
    type            dalpeMassonCanopySource;
    active          off;
    writeFields     on;
    readLanduseFromRaster  off;
    readCanopyHeightFromRaster  off;
    canopyHeightRasterFileName  "constant/canopy_height.asc";
    translateRaster (0 0 0);
    sourcePatches (ground forest);
    patchLanduse (0 1);

    landuse
      {
        low_birch
          {
            code 1;
            Cd 0.2;
            LAI 2.15;
            z0 0.06;
            height 7.5;
            LADProfile ( 0.05 0.1 0.15 0.35 1.1 0.9 0.5 0.2 0.15 0.05 0.01 );
          };

        grass
          {
            code 0;
            Cd 0.2;
            LAI 0;
            z0 0.06;
            height 0;
          };

      };
}
```

4.4.2 Running the case

Remove the results from the previous run and re-run the case the same way as before. After the run has finished, the case is reconstructed and the sampling function object is executed to generate new data for the vertical profiles.

4.4.3 Checking the result

Since writeFields was set to "on" for the canopy source in `constant/fvOptions`, the `landuse`, `z0` and `LAD` fields are written to disk when they are created. They only get written once, so they will be located in the 0-directory.

Visualization of `LAD` using paraView shows that it has been successfully set (see figure 4.3). The `U` field is shown in figure 4.4 and indicates a reduction in speed downstream of the forest edge. For

k, a clear increase is instead seen (see figure 4.5).

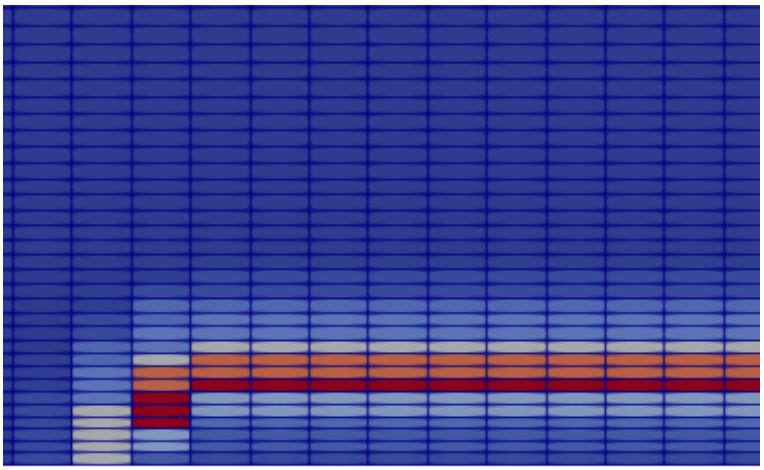

Figure 4.3: The field for Leaf Area Density at the forest edge seen from the south.

Figure 4.4: The magnitude of U along the 2D-domain.

4.4.4 Comparison with measurements

To compare the results with measurements, the plotting script `plot_vertical_profiles.py` is provided in the tutorial case directory. This script is executed without any arguments. Besides the vertical profiles from the simulations it also reads a csv-file with statistical profiles for wind speed and standard deviations from Irvine & Gardiner (1997). The statistical profiles are made dimensionless by dividing with friction velocity calculated at mast 1, on a height of 2 h. They are provided in `constant/measurements_irvine_1997.csv`.

The result from the plotting is shown in figure 4.6. It can bee seen in the figure that the velocity profile is well described. The largest deviation is seen at the first mast, indicating that the inlet

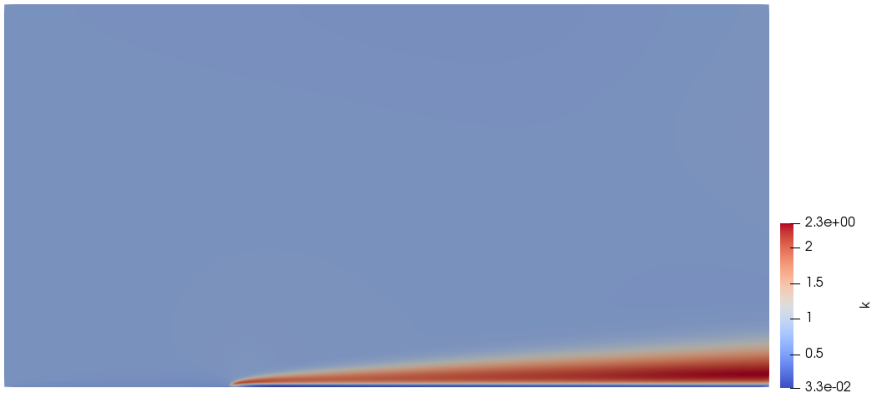

Figure 4.5: The turbulent kinetic energy along the 2D domain.

profile does not fully match the measured flow at mast 1. To some extent, this might be caused by the problems to maintain the inlet profiles as demonstrated in figure 4.2. Also, the meteorological conditions do not perfectly fulfill the assumption on which the applied inlet profiles are based.

4.4.5 Non-uniform canopy height

Irvine & Gardiner (1997) provides a height profile of the forest starting from the edge. This profile has been formatted as a raster file, allowing LAD to follow the variations. To make use of the raster, the entry for the canopy source in `constant/fvOptions` dictionary is modified to include:

```
readCanopyHeightFromRaster  on;
canopyHeightRasterFileName  "constant/canopy_height.asc";
```

By removing the `LAD`, `landuse` and `z0` fields from the `0`-directory and re-running simpleFoam for a few time-steps, the fields are rewritten using the raster as input. The varying LAD-profile is shown in figure 4.7.

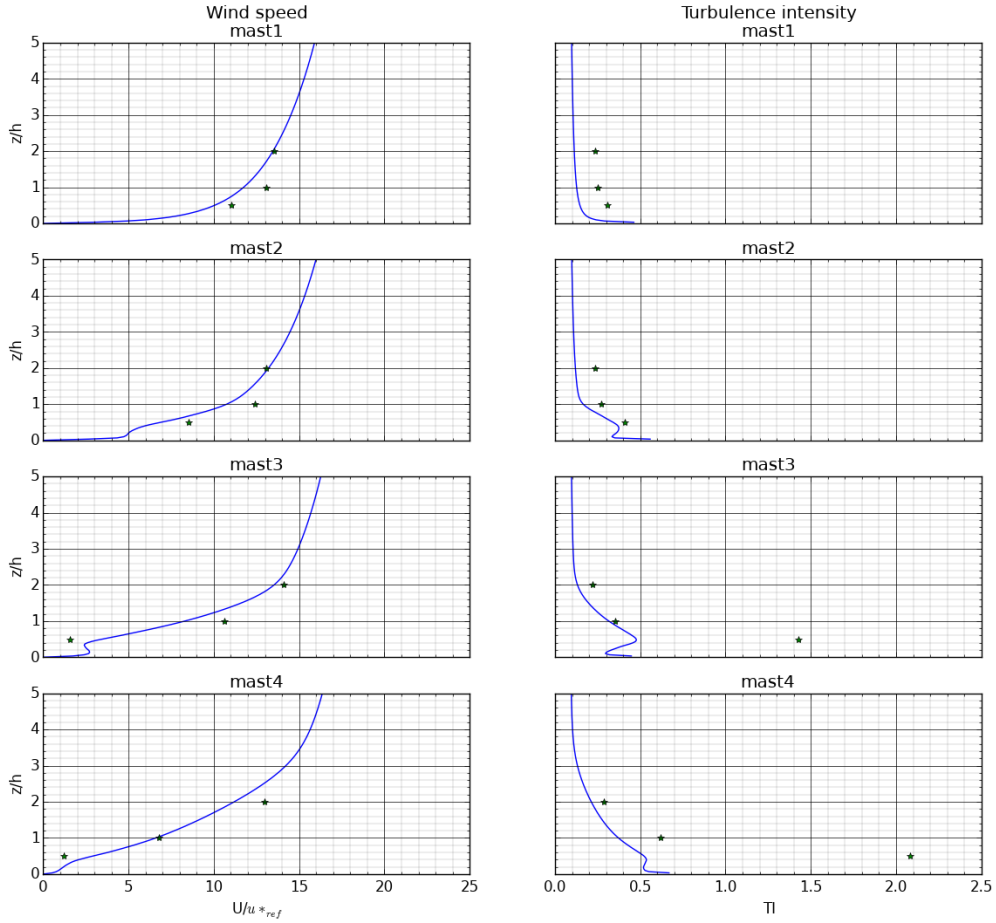

Figure 4.6: Comparison of calculated (lines) and measured (markers) profiles of wind speed and turbulence intensity.

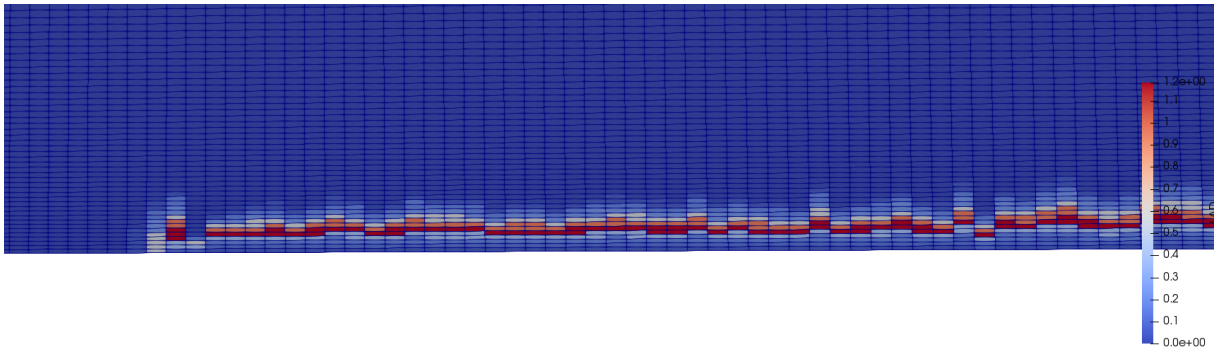

Figure 4.7: Varying height of LAD at the forest edge.

Chapter 5

References

- Dalpé B., Masson C., Numerical simulation of wind flow near a forest edge, J. Wind Eng. Ind. Aerodyn. 97 (2009) 228-241.
- Franke J., Hellsten A., Schlunzen H., Carissimo B. Best practice guideline for the CFD simulation of flows in the urban environment. Cost action 732. 1 May 2007.
- Hargreaves D.M., Wright N.G., On the use of the k-epsilon model in commercial CFD software to model the neutral atmospheric boundary layer. J. Wind Eng. Ind. Aerodyn. 95 (2007) 355-369.
- Irvine, M.R., Gardiner, B.A., Hill, M.K., 1997. The evolution of turbulence across a forest edge. Boundary-Layer Meteorology 84 (3), 467-496.
- WMO Guide to Meteorological Instruments and Methods of Observation WMO-No. 8. 2008.
- Nikuradse, J. (1933), Strömungsgesetze in rauhen Rohren, Forsch. Arb. Ing., 361.
- Patankar V. S., Numerical Heat Transfer and Fluid Flow. Hemisphere Publishing Corporation. 1980.
- Svensson, U., Häggkvist, K. A two-equation turbulence model for canopy flows. J. Wind Eng. Ind. Aerodyn. 35 (1990) 201-211.

Chapter 6

Study questions

1. Why can it be difficult to apply common rough wall functions using the Nikuradse sand-grain roughness for atmospheric flow?
2. How can the effect from tree canopy on the flow be modelled without resolving the tree canopy in detail?
3. How can any OpenFOAM solver add source-terms from a run-time selectable option without knowing any details of the implementation?
4. Which functions does OpenFOAM provide to linearize source-terms?