Modeling Turbulent Boundary Layers with Elliptic Relaxation

BY

Jørgen Myre

THESIS

for the degree of

MASTER OF SCIENCE

(Master i Anvendt matematikk og mekanikk)

Faculty of Mathematics and Natural Sciences
University of Oslo

May 2011

Det matematisk- naturvitenskapelige fakultet
Universitetet i Oslo

Modeling Turbulent Boundary Layers
with
Elliptic Relaxation

by

Jørgen Myre

THESIS
for the degree of

MASTER OF SCIENCE
(Master i Anvendt matematikk og mekanikk)

Faculty of Mathematics and Natural Sciences
University of Oslo

May 2011

Det matematisk- naturvitenskapelige fakultet
Universitetet i Oslo

Abstract

The theory behind the RANS equations and common DRSM modeling terms is presented. Most current DRSM turbulence models have models for the redistribution tensor which are developed for homogeneous flow. The Elliptic Relaxation model is presented, which is an attempt to make homogeneous redistribution models behave better close to walls. This model is then implemented in the CBC.RANS framework, which uses the Finite Element Method. Results are analyzed with a focus on stability. Further investigations and improvements are proposed based on successes and failures.

Acknowledgments

Firstly, I would thank my supervisor Bjørn Anders Reif, who provided the framework upon which this work stands.

Mikael Mortensen deserves more praise than there is space for here. Most of all for his neverending patience in the face of unreasonable requests and incomprehensible code, but also for his willingness to provide good advice and tips whenever I would interrupt his work day. Without his help, this thesis would have ended at chapter 3.

With this thesis I complete an education that has been my driving purpose for the past 8 years, and it would have been impossible without the support of my family. They have been helpful in countless ways, and I hope the work I've done lives up to their expectations.

But my most heartfelt thanks goes to my closest friends and our roleplaying games. Our travels through imaginary worlds have kept me sane through the trials of this all-too-real one. A special mention goes to Nils Ødegården, for his subsidy of my otherwise meager food budget and his level perspective on life during these past years.

For all these things, and more, I am deeply grateful.

Nomenclature

Notation and physical variables

$\mathbf{v}$	vector, vector notation
$\mathbf{e}_i$	standard basis vector, i'th direction (1 = x, 2 = y, 3 = z), vector notation
$\mathbf{A}$	second order tensor, vector notation
v_i	vector, index notation
A_{ij}	second order tensor, index notation
M_{ijkl}	fourth order tensor, index notation
a_{ij}	(Reynolds stress) Anisotropy tensor, index notation (dimensionless)
$\mathbf{A}$	(Reynolds stress) Anisotropy tensor, vector notation (dimensionless)
f_{ij}	Model tensor for redistribution, index notation (s^{-1})
$\mathbf{F}$	Model tensor for redistribution, vector notation (s^{-1})
k	Turbulent kinetic energy (m^2/s^2)
L	Model length scale (m)
$\tilde{P}$	Total pressure field ($kg/[ms^2]$)
P	Mean pressure field ($kg/[ms^2]$)
p	Fluctuating pressure field ($kg/[ms^2]$)
P_{ij}	Production of Reynolds stresses, index notation (m^2/s^3)
$\mathbf{P}$	Production of Reynolds stresses, vector notation (m^2/s^3)
$\mathbf{R}$	Reynolds stresses, vector notation (m^2/s^2)
S_{ij}	Rate of strain tensor, index notation (s^{-1})
$\mathbf{S}$	Rate of strain tensor, vector notation (s^{-1})
T	Model time scale (s)
T_{ij}	Turbulent transport of Reynolds stresses, index notation (m^2/s^3)
$\tilde{U_i}$	Total velocity field, index notation (m/s)
U_i	Mean velocity field, index notation (m/s)
U, V, W	Mean velocity field, individual components (m/s)
u_i	Fluctuating velocity field, index notation (m/s)
u, v, w	Fluctuating velocity field, individual components (m/s)
$\mathbf{u}$	Mean velocity field, vector notation (m/s)
$\overline{u_i u_j}$	Reynolds stresses, index notation (m^2/s^2)
$\overline{u^2}$, $\overline{v^2}$, $\overline{uv}$	Reynolds stresses, individual components (m^2/s^2)
W_{ij}	Rate of rotation tensor, index notation (s^{-1})
$\mathbf{W}$	Rate of rotation tensor, vector notation (s^{-1})
ϵ	Diffusion of turbulent kinetic energy (m^2/s^3)
ϵ_{ij}	Diffusion of Reynolds stresses (m^2/s^3)
ϕ_{ij}	Redistribution tensor, index notation (m^2/s^3)
ϕ^h_{ij}	Homogeneous model redistribution tensor, index notation (m^2/s^3)
ϕ^h	Homogeneous model redistribution tensor, vector notation (m^2/s^3)

Model Constants and numerical quantities

C_1	SSG, 3.4
C_1^*	SSG, 1.8
C_2	SSG, 4.2
C_3	SSG, 0.8
C_3^*	SSG, 1.3
C_4	SSG, 1.25
C_5	SSG, 0.4
C_L	Length scale, 0.25
C_P	LRR-IP production term, 0.6
C_R	LRR-IP, anisotropy term, 1.8
$C_{\epsilon 1}^*$	ϵ-equation production, 1.44
$C_{\epsilon 2}^*$	ϵ-equation dissipation, 1.9 or function
$C_{\epsilon d}$	Nonconstant $C_{\epsilon 1}^*$-equation, 0.045
C_η	Length scale, 80
C_μ	Turbulent transport, Daly-Harrow model, 0.22
C_ν	Stabilizing eddy viscosity ν_T-term, 0.09
e_d	ϵ-linearization in k-equation, $= 0.5$
σ_ϵ	ϵ-equation, scaling of Laplacian, 1.3
θ	Relaxation parameter for Picard Iteration
$\mathbf{E}$	Matrix in nonlinear system of equations
$\mathbf{E}^*$	Linearized matrix for nonlinear system
$\mathbf{f}$	Vector in nonlinear system of equations
$\mathbf{f}^*$	Linearized vector for nonlinear system
r	Residual of nonlinear system of equations
$\mathbf{x}$	Vector solution of nonlinear system of equations
$\mathbf{x}^n$	Vector holding numerical solution, n'th iteration

Contents

Chapter 1

Introduction

1.1 What is turbulence?

The Navier-Stokes equations are four equations governing the transport of momentum and mass in fluid flows. Both are derived by looking at arbitrary control volumes in a fluid. Together, they are all the equations required to describe the movement of incompressible viscous fluid flow regimes with constant temperature, and the flows we are to consider will fulfill these criteria.

Here are the equations:

$$\begin{aligned} \frac{d\mathbf{v}}{dt} + \mathbf{v} \cdot \nabla \mathbf{v} &= -\frac{1}{\rho} \nabla P + \nu \nabla^2 \mathbf{v} \\ \nabla \cdot \mathbf{v} &= 0 \end{aligned} \tag{1.1.1}$$

Of the flows described by Navier-Stokes, some are termed 'turbulent'. It is common to view turbulence as a property imposed on a laminar flow regime when the Reynolds number reaches a certain number, but this not nescessarily a correct assumption. Since the equations do not change, it is better to view turbulence as rather a property of all flows, but one neglible in flows where other forces, like viscosity, are more dominant. This idea is not a radical departure from the previous one, since the Reynolds number represents the ratio of inertia-to-viscous forces. In laminar flows the viscous forces works to 'damp' out disturbances in the flow field. This is because the viscous term in the equation is a Laplacian term, which is elliptic. Such elliptical terms, when dominant in the equation, will tend towards 'smooth' and steady-state solutions. The transition between laminar and turbulent flows is characterized by the fact that pertubations introduced in the flow are amplified downstream, that is to say that the inertial forces become more dominant than the viscous. It is common for theoretical purposes to divide a turbulent flow regime into two parts, the mean flow and the fluctuating flow (also referred to as 'pertubations'). The mean flow is stationary or varies very slowly in time while the fluctuating flow describes the rapid changes in the velocity field. Since the mean flow is in some respects mathematically similar to a laminar flow, this has lead to the 'imposed property'-view. Turbulent flow regimes are identifiable by the following:

- Vorticity: The flow exhibits strong vorticity (though these vortices might not be idealized vortices visible to the naked eye)
- Time-dependence: All turbulent flows display fluctuations in time, though they may be periodical. Some of these fluctuations can be very small and rapid compared to larger flow phenomena.
- 3-dimensional: All turbulent flows are 3-dimensional, displaying fluctuations in all three directions. This is in contrast to laminar flows which though they exist in the real world (and thus have at least some small change in all directions), can be regarded as purely 2-dimensional in some regions of the flow field.
- Diffusion: Rapid mixing of scalar properties like temperature and colored ink into the flow. Turbulent flows also transfer momentum from regions in the flow much faster than laminar flows.
- Dissipation of energy: A turbulent flow regime dissipates energy from the mean flow much faster than the viscosity observed in laminar flows.

These properties are ALWAYS present in turbulent flows, which means that a flow regime which does not exhibit all these properties is not turbulent. One other phenomenon in turbulent flow is the 'energy cascade'. This is a property with important computational consequences.

1.2 The Energy Cascade and modeling

A central physical phenomenon in turbulent flows is the presence of very small vortical structures or 'eddies', which are 'fed' energy from the larger structures through convection. In fact, it is common [7] to describe the turbulence as a spectrum of these eddies, from large scale-eddies at the mean flow level down to tiny eddies at a small-scale level. One can construct the scales for velocity, time and length of the large eddies from mean flow properties U and L. The smaller scales are more difficult, but were obtained through dimensional analysis by Andrey Kolmogorov in 1941. He proposed that the small scales should only be defined by 'local' properties at that point in space and be independent of larger properties. By defining the small scales as functions of the turbulent kinetic energy, dissipation and molecular viscosity, he devised such a set of scales. These 'Kolmogorov microscales' can be shown to be functions of the inverse Reynolds number [7], which means that as the Reynolds number increases, the scales become smaller. It is worth to point out that since the Reynolds number is a function of large-scale geometry, it is not strictly true that the microscales are independent of large scale properties. This means that as the Reynolds number increases, the smallest fluctuations in the flow will become smaller and smaller, as the energy is spread over a broader spectrum.

In experiments, these phenomena are easily observable. Initially, the transition from laminar flow to turbulent is characterized by oscillations and fluctuations which are easy to discern with the naked eye. As the Reynolds number is increased, the flow becomes more and more 'chaotic' and flow structures become more difficult to see. This is simply the effect of smaller and smaller fluctuations. This visual 'chaos' is also intensified by the fact that in addition to length scales there is a reduction in the time scale, so that the smallest fluctuations are becoming more and more rapid. A simple example is presented by the schematic in figure 1.1.

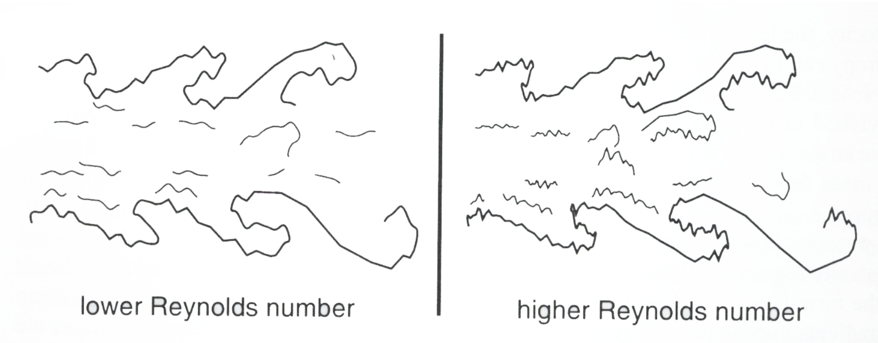


Figure 1.1: Schematic suggesting the increase in small scale structures in a free-shear layer as Reynolds number increases. Note that the large scales stay constant. Taken from [7]

A final aspect of the decrease in scales in an increase in computational difficulty. When a domain is discretized for numerical computation, the smallest grid discretization should be smaller than the smallest effects. When discretizing the Navier-Stokes equations, this means that the smallest grid division must be smaller than the Kolmogorov length scale to capture all turbulence effects. Since this is practically impossible for high Reynolds Numbers and complex geometries, we must develop models for the turbulence effects which are neglected when the grid is too coarse.

1.3 Boundary layers and their importance

In the fluid mechanics scientific community there seems to be a rather ambiguous use of the words 'boundary layer' and 'near-wall'. We will therefore attempt to explain how we interpret these terms, and how we will use them. The following account is based on a combination of [9], [18] and our education in the field of turbulence modeling, and as such we recognize it as somewhat biased.

In 1904 Ludwig Prandtl presented the concept of a 'boundary layer'. This was seen as a breakthrough in fluid dynamics, since it reconciled the very popular and mathematically elegant system of complex potential solutions and the observable (real) viscous phenomena in fluid flows. Prandtl posited that the flow around an object could be divided into two parts: A free stream (potential) flow, in which viscosity could be neglected, and a viscous boundary layer flow close to the object. When added together, these two solutions would describe the flow. The main reason for this need for two solutions was the fact that potential solutions did not predict zero velocity at the wall (the 'no-slip' condition), though this was observed in experiments. Several simplifications to the Navier-Stokes equations could also be applied in the boundary layer, further enabling easier computation. This breakthrough allowed for the evaluation of lift and drag on solid bodies, arguably moving fluid dynamics from the realm of theoretical mathematics into practical engineering applications. One important observable property was that the boundary layer area corresponded with the area of the flow where most of the activity governing heat transfer between the object and the flow and mixing of particles occurred. This gave credence to the theory, further cementing the idea of the boundary layer as a concept.

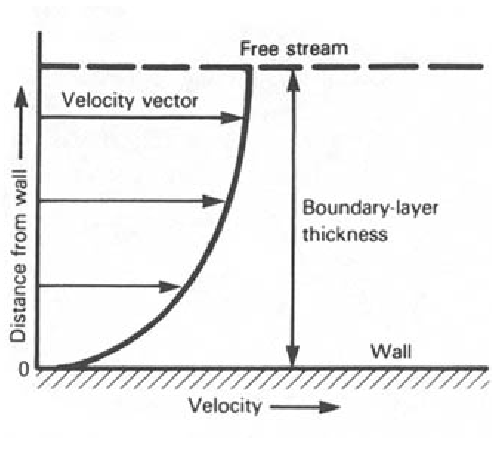


Figure 1.2: A simple schematic of the basic idea of the boundary layer. Taken from [4]

Later, when it was discovered that this approach could not adequately describe the phenomena observed at high Reynolds numbers, the idea of 'boundary layer solutions' was still attractive as the only way to model complicated phenomena, though crudely. This led to the theory that the regions of turbulent flow could be divided into regions where different solutions could be used, without loosing too much accuracy. In such a model, the area close to the wall became the 'near-wall' layer, which is smaller than but still conceptually similar to the boundary layer. Most prominent was the idea that the flow properties very close to a wall would be almost identical regardless of the larger geometry, beginning with the 'law-of-the-wall'-model (first presented by Theodore von Kàrmàn in 1930, see section 4.1 in [7]). This model assumes that the flow very close to the wall is a function of the pressure gradient parallel to the wall and fluid properties (density and viscosity) together with modeling parameters used for curve fitting. In other words, the near-wall flow is only a function of the Reynolds number. It is important to note that these variables could be found in experiments, while other properties like velocity profiles and pressure fluctuations were much more difficult to observe and measure in experiments during the first half of the 20th century.

However, our claim is that this approach of modeling separate areas of the flow is fundamentally wrong. It can be shown that a majority of the production of turbulent kinetic energy occurs close to the wall (we will get back to this in 2.6). If the production of turbulent kinetic energy (and thus, the impact of turbulence on the mean flow) and other important effects like mixing will occur in the area close to the wall, it seems counter-productive to simplify the area of the flow which arguably needs the most precision. It is therefore problematic to assume the near-wall flow is independent of larger flow properties, since the large scale properties being produced near the wall like turbulent kinetic energy are most certainly affected by large scale flow properties like geometry. This region should in fact be modeled with equations which are as close as possible to the 'real' equations. We will return to mathematical justifications for these claims in chapter 2 and 3.

Further, the near-wall modeling approach, like the boundary layer method, is not satisfactory if our goal is to create a general framework for computational fluid dynamics which can be applied to any geometry. It is further in the author's opinion that one model law should apply to the whole field, instead of a transfer from one law to another based simply on empirical data. This has the possible pitfall of creating a model which creates good results for one geometry because of curve-fitting, but is useless for all others.

Throughout the remainder of the text we will continue to use the term 'near-wall'. With this we mean the region close to the wall were the production of turbulent kinetic energy takes place, and common assumptions made to derive models for the redistribution of turbulent kinetic energy does not hold. This is a slightly vague concept, but it is important to keep in mind that for our model it is not necessary to know where the near-wall region begins and ends. A more precise definition would be 'the region close to a wall boundary where the assumptions of homogeneous turbulence for the modeling of the pressure-strain redistributions tensor does not hold'. Though this means that our 'near-wall region' is much larger than other definitions, we are not disinterested in effects very close to the wall, as the derivations in chapter 3 will show. But because our model seeks to bridge the gap between models that work away from wall and the ones which work very close, we will keep a more general definition. In the sections where we concern ourself with effects at or very close to the wall, the text itself will make this clear.

1.4 The Elliptic nature of pressure and near-wall effects

The pressure in a fluid is by nature elliptic. This will be shown in chapter 2, but what this means is that an effect altering the field at one point will alter the entire field, instantaneously. This effect is only instantaneous in incompressible flows, but all our flows will be incompressible and stationary, so this always holds for us. In other terms, we cannot assume that a small perturbation at one point will not create a noticeable change somewhere else in the flow field. Why this is important for our model will be properly explained in chapter 3. It will suffice to say here that this adds further credibility to our claims for the importance of near-wall effects and the need to model this area with great fidelity.

1.5 The Elliptic Relaxation model

Our goal is to present a model which does not include any near-wall modifications, but can still capture some of the near-wall phenomena in turbulent flows. We hope to attain this by creating a model which more accurately models the nature of the interaction between pressure fluctuations and velocity fluctuations in the area of strongly non-homogeneous flow. This will be presented in chapter 3, and implemented in code in chapter 4.

1.6 Computational Approach

We have implemented our Elliptic Relaxation model in the CBC.RANS framework, which uses the FENICS package for the Python programming language. FENICS translates more analytical expressions written with Finite Elements to efficient C++ code. CBC.RANS is a framework which uses FENICS to allow users to easily implement turbulence model into a segregated Navier-Stokes solver. We will explain all this in greater detail in chapter 4.

1.7 Notation

Due to the varying standards of notation, we felt it helpful to include a small section on notation.

1.7.1 Vector Notation

We will sometimes use vector notation. In this notation, a **boldface** lowercase letter denotes a vector, while a boldface uppercase letter denotes a second order tensor. We have tried to keep this notation consistent, and it should hold for the entire text in vector notation. For this text, all vectors and tensors exist in three dimensions, and the tensors are symmetric.

1.7.2 Outer products

In this text, we will drop the $\otimes$ from outer products of vectors and tensors. This means the following notation holds:

$$\mathbf{ab} \Longleftrightarrow \mathbf{a} \otimes \mathbf{b} \tag{1.7.1}$$

1.7.3 Index Notation

We will throughout chapter 2 and 3 use 'index notation', where vectors and tensors are written without basis vectors, with one line representing several lines for different basis vector directions. In this notation, we use the short hand:

$$\begin{aligned} v_i &= \sum v_i \mathbf{e}_i = \mathbf{v} \\ A_{ij} &= \sum\sum A_{ij} \mathbf{e}_i \mathbf{e}_j = \mathbf{A} \\ M_{ijkl} &= \sum\sum\sum\sum M_{ijkl} \mathbf{e}_i \mathbf{e}_j \mathbf{e}_k \mathbf{e}_l \end{aligned} \tag{1.7.2}$$

Where, for the remainder of this text, the sums are always over indices from 1 to 3. The second order tensors in the text will also be symmetric, that is:

$$A_{ij} = A_{ji} \quad \text{for} \quad i \neq j \tag{1.7.3}$$

That is, A_{ij} has 6 unique entries, not 9. As such, each equation represents either 3 (for the vectors) or 6 (for tensors) equations.

When writing out full equations, sometimes a product will include repeated indices. This is the result of a dot product, and signifies a sum over these indices for each individual index. Example:

$$A_{ik}A_{kj} = A_{ij}A_{kl}\mathbf{e}_i\mathbf{e}_j \cdot \mathbf{e}_k\mathbf{e}_l = A_{ik}A_{kj}\mathbf{e}_i\mathbf{e}_j = \mathbf{A} \cdot \mathbf{A} \tag{1.7.4}$$

(Here we replaced l with j, as the exact letter does not mean anything). In a similar vein, there will sometimes be two repeated indices. In such cases they correspond to a series of dot products or an inner product. In the case of an inner product, we have:

$$\mathbf{A} \colon \mathbf{A} = A_{ij} A_{kl} \mathbf{e}_i \mathbf{e}_j \colon \mathbf{e}_k \mathbf{e}_l = A_{kl} A_{kl} \tag{1.7.5}$$

For the remainder of the text, an attempt has been made to restrict summation indices to 'k' and 'l', while 'i' and 'j' are reserved for 'free' indices. In index notation it is common to use uppercase letters for quantities which exists as mean properties while keeping lowercase letters for fluctuating properties. However, this is not upheld at all when considering the bewildering array of scales, modeling constants and physical constants.

1.7.4 Derivatives

We will also use another shorthand for spatial and time derivatives in index notation:

$$\partial_i f = \frac{\partial f}{\partial x_i} \tag{1.7.6}$$

and a similar notation for time derivatives:

$$\partial_t f = \frac{df}{dt} \tag{1.7.7}$$

For repeated summation over indices, eg from a Laplacian term, we rewrite:

$$\partial^2_{ii} f = \frac{\partial^2 f}{\partial x_i^2} = \nabla^2 f \tag{1.7.8}$$

This is sometimes simplified for 1D cases to simply $\partial^2_{yy} f$. We will sometimes also write the total/convective derivative in the slightly more compact 'D'-form, that is:

$$\frac{Df}{Dt} = \partial_t f + U_k \partial_k f = \frac{df}{dt} + \mathbf{u} \cdot \nabla f \tag{1.7.9}$$

Note the last term uses the different names for the mean velocity field in each chapter (see start of chapter 4).

1.7.5 An Apology

We apologize for (what is in the author's mind) the unprofessional mixing of index and vector notation, but this work is situated between two fields of study and these follow two different conventions. Instead of engaging in the Sisyphean task of establishing a unique notation which would be unintelligible to both communities, we have in each chapter sought to use the notation commonly used when discussing the topic of that chapter. Therefore index notation is used in the theory chapters (chapters 2 and 3) and vector notation used in the chapters on implementation and results (chapter 4 and 5). Chapter 5, though, has some (probably confusing) mixing of the terms. In that chapter, vector notation refers to matters of computation and the code, while index notation concerns the theoretical concepts. There would simply be to much confusing when explaining implementation into FENICS if the algorithm was written in index notation when the FENICS-code is in quasi-vector notation, and concerning the theory much work in index notation simply sidesteps the questions of vector products.

1.8 The author's 'we'

The author has, based on preference, chosen to use the 'academic plural' for the first four chapters of this thesis. This is because the author finds this more aesthetically pleasing and easier to read than the repetition of phrases like 'one can then assume...' or 'it can be seen...' and the chapters are based on much theoretical work by others. However, this form is unsuitable for the last two chapters, which represents the author's opinions and results with little reference to other works. In this chapter a more personal form is adopted, with the occasional 'I'. The author feels this form is better than presenting the results and conclusions as stemming from some objective reality. As with the difference in notation, the author apologizes if this runs contrary to the reader's taste, but will in his defense refer to the complete lack of rules regarding the style to be used in a thesis at the Institute of Mathematics.

Chapter 2

Navier-Stokes and RANS/DRSM

2.1 The Navier-Stokes Equations

Here are the incompressible Navier-Stokes equations on index-form:

$$\begin{aligned} \partial_t \tilde{U}_i + \tilde{U}_k \partial_k \tilde{U}_i &= -\frac{1}{\rho} \partial_i \tilde{P} + \nu \partial_k \partial_k \tilde{U}_i \\ \partial_i \tilde{U}_i &= 0 \end{aligned} \tag{2.1.1}$$

Where $\tilde{U}_i$ is the total turbulent flow, and $\tilde{P}$ the total pressure field.

Our goals should now to be some way of simplifying these equations. As explained in the previous chapter, we will be unable to capture the precise movements of the fluctuations with computations, as these exist on very small scales. The approach we choose is to 'average' these equations to look at the long-term effects of the fluctuations on the mean flow, accepting that we will be unable to capture the exact movements at any one specific point in time.

2.2 Averages

We must then define what we mean by averages. Based our theories on experimental data, there are two ways to find averages: Ensemble average and time average. The ensemble average is found by performing several experiments and interpolating the average flow field from these. The time average is found by letting one experiment run for a very long time and doing the same with the flow at different times. Both require that a large number of samples have to be taken, and the time average requires that the samples must be taken over a time period longer than the largest time scale for the flow. It's then obvious that the time average will be unable to capture evolution of the flow in time, and we must only use this technique on problems in which we can assume the mean flow to be stationary. The ensemble average can capture such evolutions, but has the disadvantage of requiring a large number of experiments. In fact, to capture the 'true' average of a flow, we would require an infinite amount of experiments for the ensemble average and an infinite span of time for the time average.

Put in mathematical terms: Assume $x_i(t)$ to be set of values from one experiment amongst a set of N experiments and T to be some span of time longer than the largest time scale. We can then represent the average by $\overline{x}$, defined in the following way:
Ensemble Average:

$$\overline{x(t)} = \frac{1}{N}\sum_{i=1}^{N} x_i(t) \tag{2.2.1}$$

Time Average:

$$\overline{x} = \frac{1}{N}\sum_{j=1}^{N} x_i(t_j) \approx \frac{1}{T}\int_0^T x_i(t)dt \tag{2.2.2}$$

Note that it is normally assumed that the time average experiment can be sampled enough times that the sum of samples can be considered an integral. We observe that since the time average should not be a function of time, we require the following for any positive value of t_0:

$$\int_0^T x_i(t)dt = \int_{t_0}^{T+t_0} x_i(t)dt \tag{2.2.3}$$

We call any flow with this property 'statistically stationary' and for such flows the ensemble and time average should converge to the same 'true' average for large N and T. For the remainder of the text the word 'stationary' will be understood to mean both 'statistically stationary' and stationary in the sense of variables not being dependent on time, as they describe the same property as the statistical average approaches the 'true' average. Another simple observation is that for both these averages we have the following properties: $\overline{\partial_i f} = \partial_i \overline{f}$ and $\overline{\partial_t f} = \partial_t \overline{f}$.

2.3 Reynold-Averaged Navier-Stokes

Having defined what we mean by averages, we divide $\tilde{U}_i$ into a mean U_i and a fluctuating u_i component, ie $\tilde{U}_i = U_i + u_i$. The mean and the fluctuating parts are define by the following properties: $\overline{U_i} = U_i$ and $\overline{u_i} = 0$. We repeat the process for $\tilde{P} = P + p$. It important to note that although the following derivation assumes ensemble averages, the only difference obtained by using time averages is that all ∂_t-terms will disappear from the final equations. Although we will only consider stationary flows in this thesis, we assumed a more general framework based on ensemble averages would be better.

Another important item to note is that it is usual to follow this notation:

$$\begin{aligned} \{U_1, U_2, U_3\} &= \{U, V, W\} \\ \{u_1, u_2, u_3\} &= \{u, v, w\} \end{aligned} \tag{2.3.1}$$

And we will use this notation when referring to a specific entry in the vector or tensor, eg write 'v' instead of 'u_2', while 'U_i refers to all the entries (ie, the entire vector or tensor).
We now have the equations:

$$\begin{gathered} \partial_t(U_i + u_i) + (U_k + u_k)\partial_k(U_i + u_i) = -\frac{1}{\rho}\partial_i(P + p) + \nu\partial_k\partial_k(U_i + u_i) \\ \partial_i(U_i + u_i) = 0 \end{gathered} \tag{2.3.2}$$

If we average these equations, we get:

$$\begin{aligned} \partial_t U_i + U_k\partial_k U_i &= -\frac{1}{\rho}\partial_i P + \nu\partial_k\partial_k U_i - \partial_k\overline{u_k u_i} \\ \partial_i U_i &= 0 \end{aligned} \tag{2.3.3}$$

(Actually, we use a result of the second equation in 2.3.3, $\partial_i u_i = 0$, to obtain the first equation in 2.3.3.)

The equations 2.3.3 are generally known as the 'Reynolds-Averaged Navier-Stokes', or RANS, equations.

2.4 The Reynolds stresses

The last term in equations 2.3.3, $\overline{u_k u_i}$, is generally known as the 'Reynolds stresses'. The reason they are considered a form of stress is because the term has the dimensions of a stress term, but this name is misleading. The term $\partial_k\overline{u_k u_i}$ represent the average effect of the fluctuating flow field convecting the fluctuating momentum, more commonly referred to as 'the average effect of turbulent convection'. Assuming them to be flow properties, we will now attempt to find a transport equation for these 'stresses'.

By subtracting 2.3.3 from 2.3.2, we get the following equation:

$$\begin{aligned} \partial_t u_i + U_k\partial_k u_i + u_k\partial_k U_i + \partial_k(u_k u_i - \overline{u_k u_i}) &= -\frac{1}{\rho}\partial_i p + \nu\partial^2_{kk} u_i \\ \partial_i u_i &= 0 \end{aligned} \tag{2.4.1}$$

Now, we observe that the first line of 2.4.1 is a transport equation for u_i. If we rename this as $L(u_i)$, we can obtain a transport equation for $\overline{u_i u_j}$ by the normal product rule for differential operators:

$$L(\overline{u_i u_j}) = \overline{L(u_i u_j)} = \overline{u_i L(u_j) + u_j L(u_i)} \tag{2.4.2}$$

We now use 2.4.1 to obtain the following expression for $u_j L(u_i)$:

$$\begin{aligned} &\partial_t u_i u_j - u_i \partial_t u_j + u_j u_k \partial_k U_i + u_j U_k \partial_k u_i + u_j \partial_k (u_k u_i - \overline{u_k u_i}) \\ &\qquad = -\frac{1}{\rho} u_j \partial_i p + \nu u_j \partial_{kk}^2 u_i \end{aligned} \tag{2.4.3}$$

By adding 2.4.3 to a version of itself with reversed indices, we obtain the following:

$$\begin{aligned} \partial_t u_i u_j &+ u_j u_k \partial_k U_i + u_i u_k \partial_k U_j \\ &+ u_j U_k \partial_k u_i + u_i U_k \partial_k u_j \\ &+ u_j \partial_k (u_k u_i - \overline{u_k u_i}) + u_i \partial_k (u_k u_j - \overline{u_k u_j}) \\ &= -\frac{1}{\rho}(u_j \partial_i p + u_i \partial_j p) + \nu (u_i \partial_{kk}^2 u_j + u_j \partial_{kk}^2 u_i) \end{aligned} \tag{2.4.4}$$

We see that the individual terms can be rewritten like this:

$$\begin{aligned} u_j U_k \partial_k u_i + u_i U_k \partial_k u_j &= U_k (u_i \partial_k u_j + u_j \partial_k u_i) = U_k \partial_k u_i u_j \\ u_j \partial_k (u_k u_i - \overline{u_k u_i}) + u_i \partial_k (u_k u_j - \overline{u_k u_j}) &= \partial_k u_k u_i u_j - u_j \partial_k \overline{u_k u_i} - u_i \partial_k \overline{u_k u_j} \\ u_i \partial_k (\partial_k u_j) + u_j \partial_k (\partial_k u_i) &= \partial_k (u_j \partial_k u_i + u_i \partial_k u_j) - 2 \partial_k u_i \partial_k u_j \\ &= \partial_{kk}^2 u_i u_j - 2 \partial_k u_i \partial_k u_j \end{aligned} \tag{2.4.5}$$

We now assemble the terms and average (removing the $u_j \partial_k \overline{u_k u_i}$-terms). This gives us the final equation, which is known as the Reynold-stress Transport Equation:

$$\begin{aligned} \partial_t \overline{u_i u_j} + U_k \partial_k \overline{u_i u_j} = &\overbrace{-\frac{1}{\rho}(\overline{u_j \partial_i p} + \overline{u_i \partial_j p})}^{\phi_{ij}} \overbrace{- \overline{u_j u_k} \partial_k U_i - \overline{u_i u_k} \partial_k U_j}^{P_{ij}} \\ &- \underbrace{\partial_k \overline{u_k u_i u_j}}_{T_{ij}} - \underbrace{2\nu \overline{\partial_k u_i \partial_k u_j}}_{\epsilon_{ij}} + \nu \partial_{kk}^2 \overline{u_i u_j} \end{aligned} \tag{2.4.6}$$

Which is usually written as:

$$\frac{D\overline{u_i u_j}}{Dt} = \phi_{ij} + P_{ij} + T_{ij} - \epsilon_{ij} + \nu \partial_{kk}^2 \overline{u_i u_j} \tag{2.4.7}$$

The four new terms are usually called the following:

$$\begin{array}{ll} \phi_{ij} & \text{Pressure Redistribution} \\ P_{ij} & \text{Production} \\ \epsilon_{ij} & \text{Dissipation} \\ T_{ij} & \text{Turbulent transport} \end{array} \tag{2.4.8}$$

2.5 Modeling unknown terms

Looking at the equations presented in the previous section, we have a bewildering array of new unknowns, and an almost equal amount of new equations. This closure problem is the cause of the need for turbulence modeling. At the first 'iteration' we have the RANS equations, which together with continuity are four equations with 10 unknowns (pressure, the mean velocities and the Reynolds stresses). We then derive the Reynolds stress transport equation, but these six equations only add eighteen new unknowns ($\phi_{ij}, \epsilon_{ij}, T_{ij}$). This situation will only become worse, as each new set of equations inevitably only adds more and more unknowns. It therefore becomes necessary to model some of these unknown terms as functions of other terms.

As models go, there are two broad categories we will reference: Eddy-Viscosity models and Differential Reynolds Stress Models (DRSM). The former attempts to simplify the equations for the Reynolds stresses and model the effect of turbulence through adding a 'turbulent viscosity' to the Navier-Stokes equations. The latter, which we will concern ourselves with, attempt to model the Reynolds stresses directly through transport equations. These models attempt instead to create models for the unknowns in 2.4.7.

The first 'model term' is in fact no such thing, it is another way to view the physical quantities we already know. We define the 'turbulent kinetic energy' as:

$$k = \frac{1}{2}\overline{u_i u_i} \tag{2.5.1}$$

We can get a transport equation for k by setting $j = i$ in 2.4.6 and divide by 2:

$$\begin{aligned}\partial_t k + U_l \partial_l k = & \overbrace{-\frac{1}{\rho}\partial_i \overline{u_i p}}^{\frac{1}{2}\phi_{ii}} - \overbrace{\overline{u_i u_l}\partial_l U_i}^{\frac{1}{2}P_{ii}} \\ & - \frac{1}{2}\partial_l \overline{u_l u_i u_i} - \underbrace{\nu\overline{\partial_l u_i \partial_l u_i}}_{\epsilon=\frac{1}{2}\epsilon_{ii}} + \nu\partial_{ll}^2 k\end{aligned} \tag{2.5.2}$$

We have put braces on some of the terms, tying them in to terms in the Reynolds stress transport equations. Note the term $\overline{u_i u_l}\partial_l U_i$ (half the trace of the production tensor P_{ij}), which is the production of the kinetic energy. A common assumption is that ϕ_{ij} is traceless, removing the ϕ_{ii}-term from the equation. This assumption is only generally true for homogeneous flows (p.54, [7]).

The advantage of using k is that it adds no new terms, so we are at least not making the closure problem worse.

2.6 The near-wall production of kinetic energy

If we consider 2D channel flow with a coordinate system with $y = 0$ in the middle of a channel with height $2H$ and the flow driven by a pressure gradient of strength $\partial_x P$, we have the following from the RANS-equation in x-direction:

$$-\frac{\partial_x P}{\rho} + \nu \partial_{yy}^2 U - \partial_y \overline{uv} = 0 \tag{2.6.1}$$

Observing that this equation must be antisymmetric about $y = 0$, we have the following balance:

$$\nu \partial_y U - \overline{uv} = \frac{\partial_x P}{\rho} y \tag{2.6.2}$$

Now, what does this tell us? For one, in laminar flow where $\overline{uv} = 0$ we will have $U = -\frac{\partial_x P}{2\rho\nu} y^2 + \frac{\partial_x P H^2}{2\rho\nu}$. However, in turbulent flows, the order of U is tied to the order of $\overline{uv}$. This means (in most cases) that $\|\partial_y U\|$ will decrease much faster than the laminar case away from the wall. For a simple illustration, see figure 2.1.

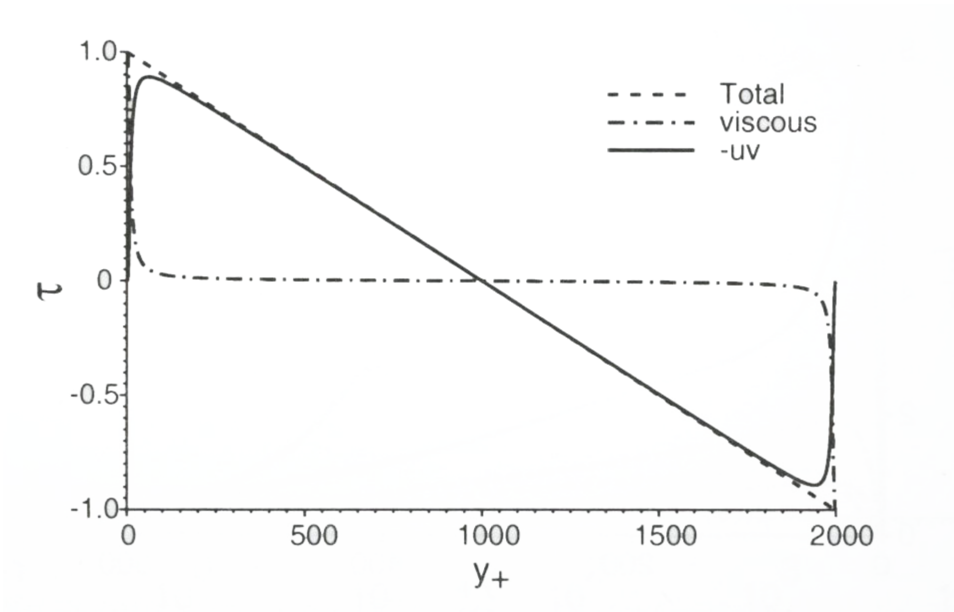


Figure 2.1: Schematic suggesting the balance between viscous stress ($\nu \partial_y U$) and Reynolds shear stress ($\overline{uv}$). Here $\tau = u_*^2$. Taken from [7]

It is also common to define a 'friction velocity' $u_* = \sqrt{-\frac{\partial_x P}{\rho}}$. We can then rewrite 2.6.2 as:

$$\nu \partial_y U - \overline{uv} = -u_*^2 y \tag{2.6.3}$$

This is a result which will be used in chapter 5.

Now, looking at the production of turbulent kinetic energy with the same restraints, we see that the only nonzero term in P_{ij} is:

$$P_{11} = -2\overline{uv}\partial_y U \tag{2.6.4}$$

The production must be nonnegative, as k cannot be negative. We assume opposite signs on $\overline{uv}$ and $\partial_y U$, and see that this makes P_{11} positive. If we look at the product of these two terms, we easily see that the production of turbulent kinetic energy k is largest close to the wall (exactly where varies with Re). A simple rule of thumb will also be that this point of highest production will be closer to the wall as the Reynolds number increases.

Now, what does this mean? It means that for the simplest flow, one of the most important factors for computing a correct velocity profile is decided in the near-wall area. The simplest argument we can make for near-wall modeling is then that if this area is important for such simple geometries, it must be important to model it correctly for more complex geometries.

It also means that all variables must be modeled with good precision, since we need to model the transfer of energy from k to $\overline{uv}$ properly. The argument goes thus: We want the most accurate U, which means we need a precise value for $\partial_y U$. But to get that, we need a precise value for $\overline{uv}$. But to get this, we need to model the transfer of kinetic energy from k to $\overline{uv}$, which means we need to model the redistribution tensor correctly. We also know from above that the area of most interest is the near-wall (because of production). Here we are getting slightly ahead of ourselves, but this is to foreshadow the assumptions in chapter 3.

Unrelated to our work, this also shows that the integral of turbulent velocity profile will always be smaller than the laminar case for a given C_P (ie, Reynolds number), which shows that turbulence always removes energy from the mean flow.

2.7 Modeling ϵ and turbulent transport

It is also common to model a transport equation for the dissipation trace ϵ and have some simplified model for the anisotropy of the dissipation tensor ϵ_{ij} as a function of other variables and ϵ (we will have to get back to our specific handling of this at the end of the chapter). It is important to remember that this equation is purely a model based on the mathematical form of the k-equation, and does not come from a physical law or set of equations with a physical basis. The model transport equation for ϵ we will use is:

$$\partial_t \epsilon + U_l \partial_l \epsilon = \frac{C^*_{\epsilon_1}\overline{u_i u_l}\partial_l U_i - C^*_{\epsilon_2}\epsilon}{T} + \frac{\nu}{\sigma_\epsilon}\partial^2_{ll}\epsilon + Q(\epsilon) \tag{2.7.1}$$

Where $C^*_{\epsilon_1}$, $C^*_{\epsilon_1}$ and σ_ϵ are model constants and Q is some model for the 'turbulent transport' of the 'scalar quantity' ϵ. In fact, we notice that most of our current equations (Reynolds stresses, k, ϵ) require a model for turbulent transport. For the remainder of the text, we have used the Daly& Harrow-model [5], since this is a commonly used model. It approximates turbulent transport as follows:

$$-\partial_k \overline{u_k u_i u_j} = \partial_k \left(C_\mu T \overline{u_k u_l} \partial_l \overline{u_i u_j} \right) \tag{2.7.2}$$

Where C_μ is a model constant and T is some timescale (we will get back to modeling scales in 2.8). It is usually pointed out that $C_\mu T \overline{u_k u_l}$ has the same units as viscosity and is usually written as ν_{kl}, but we dispense with this notation, as it simply obfuscates the true model. For the ϵ-equation, we have:

$$Q(\epsilon) = \partial_k \left(C_\mu T \overline{u_k u_l} \partial_l \epsilon \right) \tag{2.7.3}$$

But we see that if we are to use these two equations we have to have proper boundary conditions for no-slip walls. We observe that close to the wall the flow can be considered to be only a function of y. Note that we will use y and $y = 0$, but in this derivation y is the 'wall-distance coordinate', a coordinate in a new reference system aligned to the wall, not the Cartesian y. To avoid further confusion between these wall coordinates, we designate the normal and tangential components of our coordinate system relative to the wall as n, t_1 and t_2. We then have the following Taylor expansion:

$$\begin{aligned} u_i\big|_{y=0} &= 0 \\ &\Downarrow \\ u_{t_1}(y) &\simeq a_1 y + a_2 y^2 + a_3 y^3 + ... \Rightarrow u \simeq O(y) \\ u_n(y) &\simeq b_2 y^2 + b_3 y^3 + ... \Rightarrow v \simeq O(y^2) \\ u_{t_2}(y) &\simeq c_1 y + c_2 y^2 + c_3 y^3 + ... \Rightarrow w \simeq O(y) \end{aligned} \tag{2.7.4}$$

The order of u_n follows from continuity, as $\partial_y u_n = 0$ when we assume that the other components are almost constant in their respective directions very close to the wall. We then see that for k, we have the following

$$\begin{aligned} k\big|_{y=0} &= \frac{1}{2} u_i u_i \big|_{y=0} = 0 \\ k = \frac{1}{2} u_i u_i &\simeq O(y^2) \Rightarrow \partial_y k\big|_{y=0} = 0 \end{aligned} \tag{2.7.5}$$

We now have two different boundary conditions for k, which means we can now use our two new equations without any problems. However, when implementing these equations in computational systems, there are common difficulties with imposing two boundary conditions at one boundary on one variable. it is therefore necessary to derive a boundary condition on ϵ. We start by observing that by definition

$$\epsilon_w = \nu \overline{\partial_k u_i \partial_k u_i}\big|_{y=0} \geq 0 \tag{2.7.6}$$

If we assume

$$\partial_y u_{t_1}\big|_{y=0} \neq 0 \quad \text{and} \quad \partial_y u_{t_2}\big|_{y=0} \neq 0 \tag{2.7.7}$$

We get $\epsilon_w \simeq O(1)$. If we then look at the least possible order of the terms in 2.5.2, we have:

$$\overbrace{\partial_t k + U_l \partial_l k}^{O(y^2)} = \overbrace{-\overline{u_i u_l}\partial_l U_i - \frac{1}{2}\partial_l \overline{u_l u_i u_i}}^{\text{at least } O(y)} \overbrace{-\epsilon_w + \nu \partial^2_{ll} k}^{O(1)} \tag{2.7.8}$$

Which means that in the limit of $y \to 0$, we have

$$\begin{gathered} \epsilon_w = \nu \partial_y \partial_y k \\ \Downarrow \\ \lim_{y \to 0} k = A + By + \frac{\epsilon_w y^2}{2\nu} \end{gathered} \tag{2.7.9}$$

Now, from 2.7.5 we have $A = B = 0$, ending up with the boundary condition:

$$\epsilon_w = \lim_{y \to 0} \frac{2\nu k}{y^2} \tag{2.7.10}$$

Although this boundary condition looks rather more complicated to implement than the simpler ones in 2.7.5, it is just an expression of the same equation and not a model term, which means a system will behave in the same way with either of the two boundary conditions.

2.8 The tensors a_{ij}, S_{ij}, W_{ij} and scales

When modeling unknowns, it is sometimes valuable to see if one can create nondimensional variables and scales (one-dimensional quantities derived from known variables). These can be used to easily identify the terms which will be necessary in a modeled equation.

Looking at our variables, we see that we can create a nondimensional version of $\overline{u_i u_j}$ like this:

$$a_{ij} = \frac{\overline{u_i u_j}}{2k} - \frac{1}{3}\delta_{ij} \tag{2.8.1}$$

This nondimensional tensor is called the Reynolds stress anisotropy tensor, or simply the anisotropy tensor in this text.

We can also create the following scales [6]:

$$\begin{aligned} T &= \max\left\{\frac{k}{\epsilon}, 6\sqrt{\frac{\nu}{\epsilon}}\right\} \\ L &= C_L \max\left\{\frac{k^{\frac{3}{2}}}{\epsilon}, C_\eta \left(\frac{\nu^3}{\epsilon}\right)^{\frac{1}{4}}\right\} \end{aligned} \tag{2.8.2}$$

Where the different C's are determined from experiments. We will come back to the values we will use at the end of Chapter 4. It is important to note that these are maximum-expressions because scales should not be allowed to become zero. An expert reader will note that the first expressions are the Kolmogorov scales, while the second are combinations of ν and ϵ obtained using the pi-theorem.

In addition we can construct the following two tensors from the mean flow field:

$$\begin{aligned} S_{ij} &= \frac{1}{2}\left(\partial_i U_j + \partial_j U_i\right) \\ W_{ij} &= \frac{1}{2}\left(\partial_i U_j - \partial_j U_i\right) \end{aligned} \tag{2.8.3}$$

These two tensors are called the rate of strain and rate of rotation tensor, respectively. They are commonly used in modeling, as they are symmetric and antisymmetric rewritings of $\partial_i U_j$.

All of these five variables will be referred to later in the text, they were collected here instead of introducing them 'on the fly' in the text.

2.9 The magical mystical ϕ_{ij}

Reviewing our equations and variables, the only unknown term we have not addressed is ϕ_{ij}. Before we can evaluate any models, it is important to examine the physical interpretation of ϕ_{ij}. Our first step is to derive an equation for the fluctuating pressure p. NOTE: The derivation which follows is usually done for a slightly different pressure-strain redistribution tensor, but the justifications also work for our (somewhat modified) tensor, and we have tried to keep the description generic to focus on the general assumptions. For example, the same approach is used (with regard to JUSTIFICATION, not the end result) for the pressure-strain in [15] and the redistribution tensor in [7]. Firstly, we observe that the basis of modeling any term in 2.4.6 is that the term can be approximated by function of the known variables, that is:

$$\phi_{ij} = f_{ij}(\overline{u_i u_j}, k, \epsilon, \partial_l U_k, \delta_{ij}) \tag{2.9.1}$$

Since this modeling can be done for any of the terms, one usually lumps several terms into one unknown, like the common expression

$$\phi_{ij} - \epsilon_{ij} = \left(\phi_{ij} + \frac{2}{3}\epsilon\delta_{ij} - \epsilon_{ij}\right) - \frac{2}{3}\epsilon\delta_{ij} = \phi_{ij}^{*} - \frac{2}{3}\epsilon\delta_{ij} \tag{2.9.2}$$

and we will get back our approach to this in the next chapter.

To the endless grief of the student of turbulence modeling, there is no solid naming convention for different combinations of tensors, and one must often read carefully what quantities are included in a modeled term (see differences in notation from [15] and [7]).

We observe that if we take the divergence of the first Navier-Stokes equation, we get the following:

$$\nabla^2 \tilde{P} = -\rho \partial_k \tilde{U_l} \partial_l \tilde{U_k} \tag{2.9.3}$$

By averaging 2.9.3 and substracting the result from the original equation, we get the following equation:

$$\nabla^2 p = -\rho(\partial_k u_l \partial_l u_k - \overline{\partial_k u_l \partial_l u_k}) - 2\rho \partial_k U_l \partial_l u_k \tag{2.9.4}$$

The first two terms of the righthand side of 2.9.4 is commonly called the 'slow' part and the third term the 'rapid' part, because the latter changes immediately as the mean flow changes.

If we disregard the 'slow' part and considering a homogeneous (unbounded) flow, we have a formal solution:

$$p(\mathbf{x}) = \frac{1}{4\pi} \iiint_{-\infty}^{\infty} \frac{2\rho \partial_k U_l \partial_l' u_k(\mathbf{x}')}{\|\mathbf{x} - \mathbf{x}'\|} d^3\mathbf{x}' \tag{2.9.5}$$

If we remember that:

$$\phi_{ij} = -\frac{1}{\rho}\left(\overline{u_j \partial_i p} + \overline{u_i \partial_j p}\right)$$

By derivating 2.9.5, integrating by parts and using the fact that $\partial_l U_k$ is constant in homogeneous flow (details in [7], p 154), we arrive at

$$\rho \phi_{ij}^h = \frac{\rho \partial_l U_k}{2\pi} \iiint_{-\infty}^{\infty} \left(\overline{u_j(\mathbf{x}) \partial_i' \partial_k' u_l(\mathbf{x}')} + \overline{u_i(\mathbf{x}) \partial_j' \partial_k' u_l(\mathbf{x}')}\right) \frac{1}{\|\mathbf{x} - \mathbf{x}'\|} d^3\mathbf{x}' \tag{2.9.6}$$

Where the 'h' signifies that this result only holds for homogeneous turbulent flows. This is now used as justification for modeling ϕ_{ij}^h as:

$$\rho \phi_{ij}^h = M_{ijkl} \partial_l U_k \tag{2.9.7}$$

However, we must remember that our ϕ_{ij}^h does not model only the pressure-redistribution, so in addition to the 'rapid' term, we need a general 'slow' term which is not a function of $\partial_l U_k$. Summed up

$$\rho \phi_{ij}^h = A_{ij} + M_{ijkl} \partial_l U_k \tag{2.9.8}$$

Where both A_{ij} and M_{ijkl} are functions of $\overline{u_i u_j}, k, \epsilon, \partial_l U_k$ andδ_{ij}. The next step is then constructing these two tensors in such a way that the numerical calculations approximate real-life (observed) phenomena. We will not go into much detail on these (homogeneous!) models, except to assume that they are correct for (homogeneous!) flow.

All this leads to the question: How accurate is the type of models described above for flows near walls? The answer is 'pretty accurate, but they fail very close to the wall'. As shown in 2.6, $\partial_y U$ is anything but constant near the wall, and a basic assumption of these models is that $\partial_y U$ is at least close to constant. However, some of these models are very accurate away from the near-wall, and it would be nice if one could still use these models for near-wall modeling. But as we argued in 2.6, the near-wall area is very important and we would like to model it with the best precision possible. One would then ask 'is it possible to modify these homogeneous models so that they work in the near-wall region?' We believe it is, and that the answer is Elliptic Relaxation.

Chapter 3

Elliptic Relaxation

3.1 The non-local nature of pressure effects

As mentioned in previous chapters, the nature of pressure is elliptic. For justification, if we look to 2.9.3 we can see that this equation (which governs the total pressure field) is elliptic, because it is governed by a Laplacian operator. This means, as mentioned earlier, that a change at one point in the pressure field can affect the entire field. As such, it would be important to create a model for ϕ_{ij} which can reproduce this elliptic nature. This is not present in most models which are developed from assumptions of homogeneous flow, since these model terms tend to be a simple product of other flow properties. The simplest way to do this is to create a model equation for ϕ_{ij} which is elliptic. Such an equation would then impose this behavior. Our goals with the Elliptic Relaxation model is to do just that.

3.2 Elliptic pressure and near-wall effects

If we consider the effects discussed in 2.6, we can assert the following:

1. The large area in the middle of the stream where $\partial_l U_k$ is near zero fits well with the basic assumption of homogeneous models for ϕ_{ij} ($\partial_l U_k$ is almost constant)

2. The near-wall region where $\partial_l U_k$ is rapidly changing does not fit the basic assumption

From the first assertion we can say that a model for ϕ_{ij} using homogeneous assumptions will not be too wrong in the middle of the flow field. But the second tells us that the opposite should be true close to the wall: A model for homogeneous flow would by it's very nature be ill-suited for this area. Our assumption with the Elliptic Relaxation model is that the most critical property missing from a homogeneous model when modeling a nonhomogeneous flow is the elliptic nature of the pressure.

As mentioned in 2.6, we know that for channel flow the redistribution of energy from $\overline{uu}$ to $\overline{uv}$ is important if we are to model $\partial_y U$ correctly. Since we know that the production of kinetic energy and the greatest region of change in $\partial_y U$ is the near-wall region, this adds further motivation for us to create a very accurate redistribution model for this region. We can infer from this that redistribution should be important close to walls in general.

Now, as mentioned in the introduction, our hope is that the Elliptic Relaxation model will model near-wall effects better. But as a perceptive reader might notice, out model does not concern itself with near-wall effects per se. This is because that our theory is that by better modeling the effects of elliptic pressure fluctuations we will have a model which is more accurate for *both* the mean flow and the near-wall.

Lastly, it might be conceptually helpful to point out that the Elliptic Relaxation model is not a model in itself, but a modification of an existing model (for homogeneous flow). We have for simplicity's sake called it a model, as this is simpler than referring to it as 'model modification'. But we will occasionally refer to it as a 'modification' or 'method', based on context. In general, when referring to the 'ER model' we consider the entire set of equations we are using, while when we use the name ER 'modification' or 'method', we are referring to the difference between our model and a normal homogeneous model (there will be slight elaboration on this later).

3.3 Derivation of the Elliptic Relaxation Model

This section is mostly a reproduction of [8], so we will not cite that article throughout the text, but the reader would be aware that they are very similar. We remember from the last chapter, that we had found an equation for the 'rapidly' fluctuating pressure, 2.9.5, reproduced here:

$$p(\mathbf{x}) = \frac{1}{4\pi} \iiint_{-\infty}^{\infty} \frac{\overbrace{2\rho \partial_k U_l \partial_l' u_k(\mathbf{x}')}^{S(\mathbf{x}')}}{|\mathbf{x} - \mathbf{x}'|} d^3\mathbf{x}' \tag{3.3.1}$$

Which leads to the following equation for the 'rapid' parts of ϕ_{ij}:

$$\overline{u_i \partial_j p}(\mathbf{x}) = \frac{1}{4\pi} \iiint_{-\infty}^{\infty} \frac{\overline{u_i(\mathbf{x}) \frac{\partial S(\mathbf{x}')}{\partial x_j}}}{|\mathbf{x} - \mathbf{x}'|} d^3\mathbf{x}' \tag{3.3.2}$$

If we now assume we can rewrite $\overline{u_i(\mathbf{x}) \frac{\partial S(\mathbf{x}')}{\partial x_j}}$ as:

$$\overline{u_i(\mathbf{x}) \frac{\partial S(\mathbf{x}')}{\partial x_j}} = R_{ij} e^{-|\mathbf{x} - \mathbf{x}'|/L} \tag{3.3.3}$$

(OBS: R_{ij} is not symmetrical!) We obtain an equation on the form

$$\overline{u_i \partial_j p}(\mathbf{x}) = \iiint\limits_{-\infty}^{\infty} R_{ij} \frac{e^{-|\mathbf{x}-\mathbf{x}'|/L}}{4\pi|\mathbf{x}-\mathbf{x}'|} d^3\mathbf{x}' \tag{3.3.4}$$

This equation is the Green's function solution for a modified Helmholtz-equation on the form:

$$\nabla^2 \overline{u_i \partial_j p}(\mathbf{x}) - \frac{1}{L^2} \overline{u_i \partial_j p}(\mathbf{x}) = R_{ij} \tag{3.3.5}$$

This can be used to get the following equation:

$$\nabla^2 \phi_{ij} - \frac{1}{L^2} \phi_{ij} = -\frac{R_{ij} + R_{ji}}{\rho} \tag{3.3.6}$$

We can then use this equation to get create the following model equation:

$$L^2 \nabla^2 f_{ij} - f_{ij} = -\frac{\phi^h_{ij}}{k} \tag{3.3.7}$$

where $kf_{ij} = \phi_{ij}$. The factor of k also enforces the proper behavior $\phi_{ij} = 0$ at walls. It is then important to note that the core of the Elliptic Relaxation is Laplacian term $L^2\nabla^2 f_{ij}$. Thus when we refer to the 'ER modification' in the mathematical sense, we are referring to this term. We see that without this term we recover the homogeneous model. This is indeed what happens when in an area of homogeneous flow, as spatial derivatives (the Laplacian) is zero.

This might look like a small modification, but note that before ϕ_{ij} was just a product of other variables, and was for this reason easily computed. With Elliptic Relaxation we need to solve the 6 equations for the entries in f_{ij}, adding to the computational complexity of the system.

3.4 Further modifications

As mentioned in the last part of the previous chapter, it is common to include parts of other unknowns into a modeled unknown like our f_{ij}, and our model is no different. We have until now not mentioned modeling of the term ϵ_{ij}, an this is the reason. Since our model includes a separate equation for ϵ, we make the following modification to our model:

$$\phi^*_{ij} = \phi_{ij} - \epsilon_{ij} + \epsilon \frac{\overline{u_i u_j}}{k} \tag{3.4.1}$$

We would now reason that a modification of the equation for f_{ij} is in order, and insert the following:

$$L^2 \nabla^2 f_{ij} - f_{ij} = -\frac{\phi^h_{ij}}{k} - \frac{2a_{ij}}{T} \tag{3.4.2}$$

Looking at 2.4.7, we now have:

$$\frac{D\overline{u_iu_j}}{Dt} + \epsilon\frac{\overline{u_iu_j}}{k} = \phi^*_{ij} + P_{ij} - T_{ij} + \nu\partial^2_{kk}\overline{u_iu_j} \tag{3.4.3}$$

We immediately see that in the homogeneous limit we get

$$\phi^*_{ij} \to \phi^h_{ij} + \epsilon\frac{\overline{u_iu_j}}{k} - \epsilon\frac{2}{3}\delta_{ij} \tag{3.4.4}$$

which means that our model reverts to the common homogeneous model. To avoid further confusion, from this point onwards we will drop the ϕ^*_{ij}-notation.

The reasons for moving the new term to the left side is simply convenience, as it makes no simplifications. As such, we feel it is perfectly reasonable, but we will note that this might mean that models for ϕ^h_{ij} which do not make this assumption might not fit that well with the Elliptic Relaxation model outlined in this chapter. However, it should be simple to see how our model could be rewritten to accommodate such differences in the ϕ^h_{ij}-model.

3.5 Boundary Conditions

Since 3.3.7 is elliptic, we require boundary conditions to solve it. To obtain these, we do a similar analysis of terms as in 2.7.8, but need to be a little more careful, as f_{ij} is a tensor, not a scalar. If we assume $\overline{u_iu_j} \sim O(y^q)$, we have $q = 2$ for $\overline{u_tu_t}$, 3 for $\overline{u_tu_n}$ and 4 for $\overline{u_tu_t}$, irrespective of specific tangential component. We now have the following

$$\begin{aligned} \overbrace{\partial_t\overline{u_iu_j}}^{O(y^q)} + \overbrace{U_k\partial_k\overline{u_iu_j}}^{O(y^{q+1})} = \overbrace{\phi_{ij}}^{O(?)} &\overbrace{-\overline{u_iu_k}\partial_kU_j - \overline{u_ju_k}\partial_kU_i}^{\text{at least } O(y^3)} + \overbrace{\nu\partial^2_{kk}\overline{u_iu_j}}^{O(y^{q-2})} \\ - \underbrace{\frac{\overline{u_iu_j}}{k}}_{O(y^{q-2})}\epsilon &+ \underbrace{\partial_l\left(C_\mu T\overline{u_lu_m}\partial_m\overline{u_iu_j}\right)}_{O(y^{q+4})} \end{aligned} \tag{3.5.1}$$

We note that this means ϕ_{ij} must be $O(y^{q-2})$ or higher close to the wall. This an interesting fact, which should be considered. If we assume the pressure fluctuations to be $\sim O(1)$, the order of ϕ_{ij} is only higher for $q \neq 2$, where it is $\sim O(y)$. We will in any case see that this analysis has problems for $q \neq 2$. What this means is that as one get very close to the wall, for the components nn and nt the principal effects are viscous stresses, dissipation and redistribution. This should mean that redistribution is important for modeling $\overline{u_nu_n}$, $\overline{u_nu_{t_1}}$ and $\overline{u_nu_{t_2}}$ very close to the wall.

To continue, we also observe that:

$$\partial^2_{kk}\overline{u_iu_j} \simeq q(q-1)\frac{\overline{u_iu_j}}{y^2} \tag{3.5.2}$$

and as $y \to 0$, we have from 2.7.10:

$$y^2 = \frac{2\nu k}{\epsilon} \tag{3.5.3}$$

Looking at only terms of $O(y^{q-2})$ or less, we have:

$$\begin{aligned}
\lim_{y\to 0} \phi_{ij} &= \lim_{y\to 0} \frac{\overline{u_i u_j}}{k}\epsilon - \nu \partial^2_{kk}\overline{u_i u_j} \\
&\Downarrow \\
\lim_{y\to 0} f_{ij} &= \lim_{y\to 0} \frac{\overline{u_i u_j}}{k^2}\epsilon - \frac{q(q-1)}{2}\frac{\overline{u_i u_j}}{k^2}\epsilon \\
&= \lim_{y\to 0} \left(1 - \frac{q(q-1)}{2}\right)\frac{\overline{u_i u_j}}{k^2}\epsilon
\end{aligned} \tag{3.5.4}$$

If we designate the normal and tangential components relative to the wall as n, t_1 and t_2, we gain the following boundary conditions on f_{ij}:

$$\begin{aligned}
f_{nn} &= -5 \lim_{y\to 0} \frac{\overline{u_n u_n}}{k^2}\epsilon \\
f_{nt_1} &= -2 \lim_{y\to 0} \frac{\overline{u_n u_{t_1}}}{k^2}\epsilon \\
f_{nt_2} &= -2 \lim_{y\to 0} \frac{\overline{u_n u_{t_2}}}{k^2}\epsilon \\
f_{t_1 t_1} &= f_{t_1 t_2} = f_{t_2 t_2} = 0
\end{aligned} \tag{3.5.5}$$

The last three conditions are rather problematic as they do not uphold the condition $f_{ii} = 0$ close to the wall. Instead Demuren and Wilson [8] used the following condition:

$$f_{t_1 t_1} = f_{t_2 t_2} = -\frac{1}{2} f_{nn} \tag{3.5.6}$$

And we intend to use this condition.

Another problem is the nonlinearity with respects to k in the boundary condition. This might cause instability in the system. However, there is a rather simple fix for this. If we can find a value for the wall distance y, we can replace $\frac{\epsilon}{k^2}$ with $\frac{4\nu^2}{\epsilon y^4}$. This condition is then still nonlinear because of the product $\frac{\overline{u_i u_j}}{\epsilon}$, but should be more stable than before.

We have now assembled all the parts of our Elliptic Relaxation model, and will move on to implementation. The next page will sum up our equations

3.6 The final model

We have now arrived at the final model to be used for the rest of this thesis:

$$\frac{D\overline{u_iu_j}}{Dt} = kf_{ij} - \overline{u_iu_l}\partial_l U_j - \overline{u_ju_l}\partial_l U_i + \nu\partial_l\partial_l\overline{u_iu_j} - \frac{\overline{u_iu_j}}{k}\epsilon + \partial_l\left(C_\mu T\overline{u_lu_m}\partial_m\overline{u_iu_j}\right)$$

$$\frac{Dk}{Dt} = -\overline{u_iu_l}\partial_l U_i - \epsilon + \nu\partial_l\partial_l k + \partial_l\left(C_\mu T\overline{u_lu_m}\partial_m k\right)$$

$$\frac{D\epsilon}{Dt} = -\frac{C^*_{\epsilon_1}\overline{u_iu_l}\partial_l U_i + C^*_{\epsilon_2}\epsilon}{T} + \nu\partial_l\partial_l\epsilon + \partial_l\left(\frac{C_\mu T}{\sigma_\epsilon}\overline{u_lu_m}\partial_m\epsilon\right)$$

$$L^2\nabla^2 f_{ij} - f_{ij} = \frac{\phi^h_{ij}}{k} - \frac{2a_{ij}}{T}$$

$$T = \max\left\{\frac{k}{\epsilon}, 6\sqrt{\frac{\nu}{\epsilon}}\right\},\quad L = C_L\max\left\{\frac{k^{\frac{3}{2}}}{\epsilon}, C_\mu\left(\frac{\nu^3}{\epsilon}\right)^{\frac{1}{4}}\right\}$$

With the boundary equations on walls (y is the wall normal coordinate):

$$\overline{u_iu_j} = 0 \qquad k = 0 \qquad \epsilon = \lim_{y\to 0}\frac{2\nu k}{y^2}$$

$$f_{nn} = -5\lim_{y\to 0}\frac{\overline{u_nu_n}}{k^2}\epsilon \qquad f_{nt_1} = -2\lim_{y\to 0}\frac{\overline{u_nu_{t_1}}}{k^2}\epsilon \qquad f_{nt_2} = -2\lim_{y\to 0}\frac{\overline{u_nu_{t_2}}}{k^2}\epsilon$$

$$f_{t_1t_1} = f_{t_1t_2} = f_{t_2t_2} = 0$$

Chapter 4

Implementation

4.1 The Finite Element Method

We assume that the reader is familiar with the basic concepts of the Finite Element Method (FEM), more specifically the Galerkin method and common bases for test/trial-functions. Fortunately, we will not be using very complicated FEM-methods or venture much further into the workings of the FEniCS compiler. As such, a reader unfamiliar with FEM should be able to follow most of the arguments in this chapter.

It is important to note that there is a reluctance in some parts of the scientific community to use FEM with Computational Fluid Mechanics (CFD). We are not aware of the nuances of the arguments for or against this, though there will be some discussion of possible problems in chapter 5. It is important to point out that the systems of equations which are derived from naive discretization with Finite Differences, the much lauded (in the CFD community) Finite Volume Method and the Finite Elements Method are surprisingly similar, in some cases identical. As such we cannot see large problems with using FEM, and for further justification see [12].

4.2 A note on FEniCS

From the FEniCS web page [1]:

> The FEniCS Project is a collection of free software aimed at automated, efficient solution of differential equations. The project provides tools for working with computational meshes, finite element variational formulations of PDEs, ODE solvers and linear algebra.

FEniCS is an attempt to create a package for the Python programming language which will allow the solving of differential equations with FEM in a manner very similar to analytical work with those equations. The goal is to reduce the time used by the user on setting up numerical solutions and removing the need for complicated code for what could be considered 'trivial tasks' like computing the entries in a matrix. The flip side of this is that as more thing are done in a 'black box' manner, the user looses control of some details, which can lead to mistakes (one example of this is presented in chapter 5).

We will not delve into the mysteries of FEniCS, noting only that it allows us to write code for FEM which is almost identical to the mathematical model we create. This means that the time used writing code is reduced, and that even to someone not familiar with Python or FEniCS, the code should be somewhat readable. For a reader interested in the 'nuts and bolts' of FEniCS, we refer to [1] and [12].

4.3 CBC.RANS

From the CBC.RANS page on launchpad [2]:

> CBC.RANS is a FEniCS-based programming framework for modeling turbulent flows by the Reynolds-Averaged Navier-Stokes Equations. CBC.RANS is primarily being developed as a joint effort between the Norwegian Defense Research Establishment (FFI), Kjeller, Norway and Simula Research Laboratory in Oslo, Norway. The solvers are developed in Python and provide a simple interface, where new turbulence models can easily be added. The Navier-Stokes solvers can be used for both laminar and turbulent flows, steady state or transient. Currently implemented models contain the standard k-epsilon model, the Spalart Allmaras model and the V2F model. CBC.RANS is distributed freely in the hope that it will be useful, but without any warranty.

CBC.RANS is an attempt to create a programming framework which can handle a large number of turbulence models with the least possible amount of work done implementing each model. Through clever use of object oriented programming, the user needs only write small pieces of code to implement new models as classes. These models can be further modified or elaborated upon by creating a subclass. In the end this allows for incredibly rapid change in the mathematical model without having to rewrite large pieces of code. In essence, CBC.RANS is a further refinement of the FEniCS goal of creating a simple mathematical framework, focused on turbulence modeling.

Now that our computational tools have been introduced, we turn to modifying our mathematical system so that it can be used in code.

4.4 Test- and trial-functions

We will for a great part of this text avoid mentioning test- and trial-functions. Because our focus is on stability and convergence and our solver is rather simple, there is little to be gained by using complicated test- and trial-functions. As such, we use Continuous Galerkin elements of order 1 for all our variables. In the case of vectors and tensor, FEniCS handles the rearranging of entries as separate equations. We simply define a test function on the function space for the tensor variable.

There are also little advanced use of test functions. They are always from the same function space as the equation they are testing, and only modified in the case of the Laplacian terms. The latter case is covered below in 4.5.

The following equations will not follow the test/trial-function notation common in FEM. There are two reasons for this: Firstly, this serves to keep the chapter as general as possible, as the inclusion of test functions and integrals would clutter up the equations, making them difficult to read for someone not well-versed in FEM. This will also keep the text accessible for someone looking to implement the same in FDM/FVM. Secondly, FEniCS diverges from this notation so there is little to be gained in using it. Instead we encourage the reader to consult the appendices when done with this chapter, as the use of test/trial-forms there should be easy to read.

4.5 Rewriting ∇^2-terms

One basic relation which will be used when computing the inner products in the code and used in the reasoning behind linearization is the following: Assuming f and g to be functions in space, we have:

$$\iint_\Omega f\nabla^2 g \,\mathrm{d}\Omega = -\iint_\Omega \nabla f \cdot \nabla g \,\mathrm{d}\Omega + \int_{\partial\Omega} f(\mathbf{n}\cdot\nabla g)\,\partial\Omega \tag{4.5.1}$$

similarly for vector or tensor functions:

$$\iint_\Omega \mathbf{V} : \,(\nabla\cdot\mathbf{A})\,\mathrm{d}\Omega = -\iint_\Omega (\nabla\mathbf{V}) : \mathbf{A}\,\mathrm{d}\Omega + \int_{\partial\Omega} \mathbf{V} : \,(\mathbf{n}\cdot\mathbf{A})\,\partial\Omega \tag{4.5.2}$$

(In fact, we see the scalar equation as a special case of the general tensor equation, but not all readers might see it that way, so we included both. Note also that $\mathbf{A}$ is one order higher than $\mathbf{V}$.)

Now we notice that the last term in both equations is only evaluated at the edges of the domains. In fact, these terms are the common way to implement the boundary conditions on the system. However, since FEniCS implements boundary conditions in a 'black box'-manner, it is not included in the equations inside the boundary. We treat our boundary conditions in a slight different manner, but we will still not use this formulation. Thus we will disregard this boundary integral term in the following derivations.

As such, the viscous Laplacian and the turbulent transport terms are the ones that require rewriting, while the rest are simply the original term multiplied by a test function and integrated over the cell. If we assume v and $\mathbf{V}$ are appropriate test functions, they are rewritten:

$$
\begin{aligned}
v\nabla^2 f &= -\nabla v : \nabla f \\
\mathbf{V} : \nabla^2 \mathbf{A} &= -\nabla \mathbf{V} : \nabla \mathbf{A}
\end{aligned}
\tag{4.5.3}
$$

Although we will not rewrite our equations in this chapter, this is important for the justification behind our linearization, as will be seen. We apologize if this is somewhat confusing, but since the rewriting modifies the test function and we do not include test functions in the equations in this chapter, we felt it best not to rewrite the equations.

4.6 Our system of equations on vector form

Since CBC.RANS uses a lower case u for the velocity vector, we return to a proper vector notation (lowercase for a vector). It is then important to note that $\mathbf{u}$ in this chapter is not the fluctuations, but the mean velocity (U_i in chapter 2). We rewrite our variables from chapters 2 and 3 in the following manner (summation over indices is implied):

$$
\mathbf{u} = U_i \mathbf{e}_i
$$

$$
\begin{aligned}
\mathbf{R} &= \overline{u_i u_j}\, \mathbf{e}_i \mathbf{e}_j & \mathbf{F} &= f_{ij}\, \mathbf{e}_i \mathbf{e}_j \\
\mathbf{S} &= S_{ij}\, \mathbf{e}_i \mathbf{e}_j & \mathbf{W} &= W_{ij}\, \mathbf{e}_i \mathbf{e}_j \\
\mathbf{I} &= \delta_{ij}\, \mathbf{e}_i \mathbf{e}_j & \mathbf{A} &= \tfrac{1}{2k}\mathbf{R} - \tfrac{1}{3}\mathbf{I}
\end{aligned}
$$

$$
\mathbf{P} = -\mathbf{R} \cdot \nabla \mathbf{u} - (\mathbf{R} \cdot \nabla \mathbf{u})^T
$$

$$
\begin{aligned}
\nu \nabla^2 \mathbf{R} - \mathbf{u} \cdot \nabla \mathbf{R} + k\mathbf{F} + \mathbf{P} - \frac{\mathbf{R}}{k}\epsilon + \nabla \cdot (C_\mu T \mathbf{R} \cdot \nabla \mathbf{R}) &= 0 \\
\nu \nabla^2 k - \mathbf{u} \cdot \nabla k + \frac{1}{2}\mathrm{tr}(\mathbf{P}) - \epsilon + \nabla \cdot (C_\mu T \mathbf{R} \cdot \nabla k) &= 0 \\
\nu \nabla^2 \epsilon - \mathbf{u} \cdot \nabla \epsilon + \frac{C^*_{\epsilon_1} \frac{1}{2}\mathrm{tr}(\mathbf{P}) - C^*_{\epsilon_2}\epsilon}{T} + \nabla \cdot \left(\frac{C_\mu T}{\sigma_\epsilon} \mathbf{R} \cdot \nabla \epsilon \right) &= 0 \\
L^2 \nabla^2 \mathbf{F} - \mathbf{F} + \frac{\phi^h}{k} + \frac{2}{T}\mathbf{A} &= 0
\end{aligned}
\tag{4.6.1}
$$

Here all terms have been rearranged on the form $f(x) = 0$, since this formulation is useful when using FEM. We have also removed the time derivatives, as we are investigating stationary flows.

4.7 Method for approximating the solution

As seen above, we have a set of nonlinear partial differential equations. When discretized with the Finite Element Method, we will need an algorithm to solve a (now discrete) set of equations for a large number of unknowns. Since our goals is to investigate the stability of several differently coupled systems, it seems logical to choose a rather simple and robust solver.
In vector notation, our system could be written like this:

$$\mathbf{E}(\mathbf{x}) \cdot \mathbf{x} = \mathbf{f}(\mathbf{x}) \tag{4.7.1}$$

where $\mathbf{E}$ is a matrix which describes the equations for the set of unknowns $\mathbf{x}$ and $\mathbf{f}$ is vector function of $\mathbf{x}$. Finding the solution for our set of equations would then be the same as finding the $\mathbf{x}$ which solves 4.7.1.

Solving such a nonlinear system of equations for $\mathbf{x}$ is rather difficult. Several methods exist, but these are sometimes difficult to implement or only usable for particular cases. Our approach is to solve the system by a simple relaxed Picard iteration, which is to say that we linearize the equations in the following way: We start at an initial guess for the set of unknowns $\mathbf{x}^0$. Then we rewrite the system as follows:

$$\begin{aligned} \mathbf{E}^*(\mathbf{x}^n) \cdot \mathbf{x}^* &= \mathbf{f}^*(\mathbf{x}^n) \\ \mathbf{x}^{n+1} &= \theta \mathbf{x}^* + (1-\theta)\mathbf{x}^n \end{aligned} \tag{4.7.2}$$

Where $\mathbf{E}^*$ is a constant matrix and $\mathbf{f}^*$ a constant vector, both computed from the values stored in $\mathbf{x}^n$ and θ is a relaxation parameter. The values stored in $\mathbf{x}^n$ are generally called 'old', since they are from the previous iteration. Solving for $\mathbf{x}^{n+1}$, the hope is that as one continues to iterate, $\mathbf{x}^{n+1}$ will converge towards the $\mathbf{x}$ which will solve 4.7.1. Although mathematically simple, the method of Picard iteration usually has problem with convergence, but our hope is that a relaxed method will work better.

The most important part of this method is choosing how to derive the linearized forms for $\mathbf{E}^*$ and $\mathbf{f}^*$ from $\mathbf{E}$ and $\mathbf{f}$. Which terms to evaluate at $\mathbf{x}^*$ and which can be calculated from $\mathbf{x}^n$ might be difficult to judge at first glance. Choosing a good linearization will ensure stability and can speed up convergence, while a bad choice will lead to instability and little chance of convergence. Any terms which include a $\mathbf{x}^*$ are inserted into $\mathbf{E}^*$, while the rest of the terms are inserted into $\mathbf{f}^*$. This is commonly referred to as the left-hand and right-hand side of 4.7.2 respectively.

When solving for a large system with several different unknown functions, it is common to split the system into into several parts with each part composing one or more (but not all) of the unknowns, and solve for each part in turn. Since we only solve some of the unknowns, we use old values from the last iteration step of their respective solvers for the other unknowns. We use the term 'uncoupled system' or 'segregated system' for these kinds of systems, whereas an undivided system is termed 'fully coupled'. Uncoupled systems are inherently less stable, but the payoff is a reduced time cost to the calculations.

It is important to note that no matter the coupling of systems, they will converge towards the same solution to the equations $\mathbf{E}$, as we assume there is only one solution. The difference between them lies in the time used to compute each iteration and the stability.

How to construct $\mathbf{E}^*$ and $\mathbf{f}^*$ will be covered now.

4.8 How CBC.RANS handles turbulence models

Although [11] gives a more in-depth overview of how CBC.RANS handles solving systems of differential equations, we will attempt to give a brief explanation, to give the reader a general understanding. Any in-depth explanation would probably become a lower quality reproduction of [11]. We have however tried to keep the rest of this chapter more general, only commenting on specifics in the code when necessary.

CBC.RANS decouples the equations governing the mean flow field and pressure from the turbulence model and creates a separate system for solving these equations. The base class for this solver is `NSSolver` (stored in `NSSolver.py`). This is so that models developed for efficient solving of the Navier-Stokes equations can be applied to this solver. These schemes are created as subclasses of `NSSolver`. The turbulence model interacts with the Navier-Stokes solver through manipulating variables used by it, for example viscosity (for an eddy-viscosity model) or the body forces (for a DRSM). Our model does both, for reasons we will explain later. In this way, the `NSSolver` is solving equations which are mathematically similar to the Navier-Stokes equations.

The turbulence model is managed in a similar way, under a base class called `TurbSolver`. There is naturally a greater variation in the subclasses of `TurbSolver` than it's counterpart `NSSolver`. These subclasses are free to create any number of variables and quantities, allowing them to be very simple (like a no-frills k-ϵ model) or very complicated (our ER model). In general a turbulence model has a parent class with simply the model name (eg, `StandardKE`), and then have subclasses holding different linearization schemes. Because of this division, we will not cover the `NSSolver` in any detail except for a small section below. We concern ourself almost exclusively with the linearization and solving of our turbulence model.

To set up a problem, CBC.RANS uses a class which includes the geometry, called `TurbProblem`. A given geometry is then a subclass of `TurbProblem`. The subclasses of `TurbProblem` we will concern ourself with are `channel`, `diffusor` and `apbl`. We not go into much detail about the `TurbProblem` class except to note peculiarities which affect our code in some sections.

4.9 Source terms, scales and linearization

Looking at our equations, we see that most are on the form:

$$\nabla^2\text{-term} + \text{source terms} - \text{convective term} + \text{turbulent transport} = 0 \quad (4.9.1)$$

These names are meant for descriptive purposes only, to better explain our linearization. Though the names are based on physical effect, these play little part in the following section. Why do we distinguish between such terms? Because we must try to avoid a system like 4.7.2 with zeroes on the diagonal of the matrix. Since these terms will be linearized differently, it is important to make a distinction between them. Systems with zeroes on the diagonal are much more likely to result in unstable systems, as there is an increased chance that the matrix $\mathbf{E}^*$ will be singular. Now, it is important to note that as we rewrite the Laplacian and turbulent transport-terms in the equation, these equations change sign (see 4.5).

Note that is common to use the words 'explicit' for a term which uses old values and 'implicit' for terms which uses new values. We will continue to use the term 'old' and 'new' for variables, as we think these terms are clearer for an uninitiated reader. We must also note that the rule of thumb is that using implicit or 'new' values is best for stability, but that we have to use caution concerning the diagonal, as noted above.

We define 'source terms' as terms which physically add or remove energy and which work only at the point. It is true that the convective and turbulent terms also add or remove energy from one point, but this energy should theoretically be added at some other point and these terms are always dependent on nearby points through space derivatives. In our case this means terms involving the production tensor $\mathbf{P}$ and the ϵ-terms in the case of equations for k, ϵ and $\mathbf{R}$. We see that in the case of the $\mathbf{R}$-equations, contributions from $\mathbf{P}$ will always lie on the diagonal. Assuming a positive value for the production, we then have a system where all terms except the production will have equal sign. This means that without the production term, chances are good for getting a non-zero diagonal. With it on the diagonal, however, we have a real risk of entries with the value of zero. As such, it is very important for stability to move the production to the right-hand side of the equation, that is $\mathbf{f}^*$ in 4.7.2. We must then only use old values for $\mathbf{P}$.

Looking at the ϵ-terms, the opposite applies. As these terms will be negative, we would like to have them on the diagonal. In the equation for $\mathbf{R}$ this is done by simply using old values for k and ϵ. For the k-equation, this is usually done by using an old value for ϵ and multiplying by the new k, then dividing by the old k. Further, Mikael Mortensen has found that there in an increase in stability if we discretize the ϵ-part in a 'half-implicit, half-explicit'-manner. We then rewrite:

$$
\begin{gathered}
\nu\nabla^2 k - \underline{\mathbf{u}} \cdot \nabla k + \frac{1}{2}\mathrm{tr}(\underline{\mathbf{P}}) - \epsilon + \nabla \cdot (C_\mu T \underline{\mathbf{R}} \cdot \nabla k) = 0 \\
\Downarrow \\
\nu\nabla^2 k - \underline{\mathbf{u}} \cdot \nabla k + \frac{1}{2}\mathrm{tr}(\underline{\mathbf{P}}) - \left(e_d \epsilon + (1 - e_d)\frac{k}{\underline{k}}\underline{\epsilon} \right) + \nabla \cdot (C_\mu T \underline{\mathbf{R}} \cdot \nabla k) = 0
\end{gathered} \tag{4.9.2}
$$

Where e_d is a constant, 0.5 in all our cases. In this case we get some stability from the avoidance of a zero diagonal AND using new values for ϵ.

The convective term has a rather common scheme, with a very good physical basis. Convecting velocity is always the velocity at the last iteration step, making the term linear. This is justified by the fact that to make sure that the convection does not remove or add energy, the convecting velocity field must be divergence-free. Since the second of the Navier-Stokes equations makes sure that any velocity found from a system of these equations is divergence free, we use a convecting velocity from an old iteration. For more details, see article on convection in [12]. This is of course a moot point, as we must always use an 'old' value for the velocity, since the solver for the velocity field is uncoupled from our solver.

Concerning the scales T and L, we see that to evaluate them as functions of new values would add a large amount of nonlinearity to the system. Looking at the equations, there seems to be no term where the scaling constants should be more important than some other variable. This, coupled with the fact that these constants are based on modeling and not on any physical phenomena, would seem to make it reasonable to evaluate them using values from the previous iteration.

Looking at the turbulent transport, we can see that it seems most appropriate to use the new value of the property being 'transported' while using old values for the 'turbulent convection' (ie, the Reynolds Stresses in the term).

Concerning the $k\mathbf{F}$-term, which does not fit our earlier categories, it is obvious that this is important for our model. We see that when this term is using new values, it will never be on the main diagonal. In this case, we don't need to regard it with this demand in mind. How this term is linearized is only a problem when $k - \epsilon$ is coupled with $\mathbf{F}$. In this case it seems more proper to use an old value for k, and solve for $\mathbf{F}$. In all other cases we have chosen to use new values, since we see no immediate problem with this.

4.10 Equations on vector form and linearized

Following our conclusions from the previous sections, we have the following discretizations, with an underline denoting an 'old' value:

$$
\begin{aligned}
\nu\nabla^2\mathbf{R} - \underline{\mathbf{u}}\cdot\nabla\mathbf{R} + \overbrace{k\mathbf{F}}^{\text{see below}} + \underline{\mathbf{P}} - \frac{\epsilon}{\underline{k}}\mathbf{R} + \nabla\cdot(C_\mu T\underline{\mathbf{R}}\cdot\nabla\mathbf{R}) &= 0\\
\nu\nabla^2 k - \underline{\mathbf{u}}\cdot\nabla k + \frac{1}{2}\mathrm{tr}(\underline{\mathbf{P}}) - \left(e_d\epsilon + (1-e_d)\frac{k}{\underline{k}}\underline{\epsilon}\right) + \nabla\cdot(C_\mu T\underline{\mathbf{R}}\cdot\nabla k) &= 0\\
\nu\nabla^2\epsilon - \underline{\mathbf{u}}\cdot\nabla\epsilon + \frac{C^*_{\epsilon_1}\mathrm{tr}(\underline{\mathbf{P}})}{2T} - \frac{C^*_{\epsilon_2}\epsilon}{T} + \nabla\cdot\left(\frac{C_\mu T}{\sigma_\epsilon}\underline{\mathbf{R}}\cdot\nabla\epsilon\right) &= 0\\
L^2\nabla^2\mathbf{F} - \mathbf{F} + \frac{\phi^h}{\underline{k}} + \frac{2}{T}\underline{\mathbf{A}} &= 0
\end{aligned}
\tag{4.10.1}
$$

where

$$
\underline{\mathbf{P}} = -\underline{\mathbf{R}}\cdot\nabla\underline{\mathbf{u}} - (\underline{\mathbf{R}}\cdot\nabla\underline{\mathbf{u}})^T \tag{4.10.2}
$$

As mentioned previously, the $k\mathbf{F}$-term is dependent on how the system is coupled. It will either be $\underline{k}\mathbf{F}$ if $\mathbf{F}$ is coupled with $\mathbf{R}$, or $\underline{k}\,\underline{\mathbf{F}}$ if they are uncoupled.

4.11 Different schemes for ϕ^h_{ij}

Up until this point, we have not addressed the specific form of ϕ^h_{ij}. This is because this term is not strictly a part of the Elliptic Relaxation model, and we wanted to keep the model as general as possible during the derivation. But for testing, we need a model ϕ^h_{ij} to test with. We have decided to use the LRR-IP and SSG models (described in [13] and [15] respectively). The LRR-IP is a rather simple model, where:

$$
\phi^h_{ij} = -C_R\frac{2k}{T}a_{ij} - C_P\left(P_{ij} - \frac{P_{ll}}{3}\delta_{ij}\right) \tag{4.11.1}
$$

which when linearized and in vector form becomes:

$$
\phi^h = -C_R\frac{2\underline{k}}{T}\mathbf{A} - C_P\left(\underline{\mathbf{P}} - \frac{\mathrm{tr}(\underline{\mathbf{P}})}{3}\mathbf{I}\right) \tag{4.11.2}
$$

The first term here is implicit, as we see that it will not add to the diagonal, so using new values should increase stability.

Another model, which is much more accurate, is the SSG model:

$$
\begin{aligned}
\phi_{ij}^{h} &= -(C_1\epsilon - C_1^{*}\mathrm{tr}(P_{ij}))a_{ij} + C_2\epsilon(a_{ik}a_{kj} - \frac{1}{3}a_{mn}a_{mn}\delta_{ij}) \\
&+(C_3 - C_3^{*}(a_{mn}a_{mn})^{\frac{1}{2}})kS_{ij} + C_4k(a_{ik}S_{jk} + a_{jk}S_{ik} - \frac{2}{3}a_{mn}S_{mn}\delta_{ij}) \\
&\qquad +C_5k(W_{ik}a_{kj} + W_{jk}a_{ki})
\end{aligned}
\tag{4.11.3}
$$

This equation can be written on vector form as:

$$
\begin{aligned}
\phi^{h} &= -\left(C_1\epsilon - C_1^{*}\mathrm{tr}(\underline{\mathbf{P}})\right)\underline{\mathbf{A}} + C_2\epsilon\left(\underline{\mathbf{A}}\cdot\underline{\mathbf{A}} - \frac{1}{3}(\underline{\mathbf{A}}\colon\underline{\mathbf{A}})\mathbf{I}\right) \\
&+\left(C_3 - C_3^{*}(\underline{\mathbf{A}}\colon\underline{\mathbf{A}})^{\frac{1}{2}}\right)k\underline{\mathbf{S}} + C_4k\left(\underline{\mathbf{A}}\cdot\underline{\mathbf{S}} + \underline{\mathbf{S}}\cdot\underline{\mathbf{A}} - \frac{2}{3}(\underline{\mathbf{A}}\colon\underline{\mathbf{S}})\mathbf{I}\right) \\
&\qquad +C_5k\left(\underline{\mathbf{W}}\cdot\underline{\mathbf{A}} - \underline{\mathbf{A}}\cdot\underline{\mathbf{W}}\right)
\end{aligned}
\tag{4.11.4}
$$

Lacking a reference for implementation of this term, we had to experiment with the terms with regards to linearization. The results of this is discussed in chapter 5.

4.12 2D-simplification and symmetric tensors

FEniCS has one major disadvantage with respects to our system: It can only support 2D meshes. As such, we will be unable to test for 3D flows. However, this is no big problem, as the Elliptic Relaxation model is sufficiently untested in FEM to keep 2D results interesting. Another problem we encountered, which has not been resolved by the FEniCS developers, is a bug stopping us from creating coupled function spaces with symmetric tensors, which are need for a system of coupled **R** and **F**. As we are unable to address this directly, our code must solve with four entries in **R** and **F**, and set the off-diagonal entries to be equal after each iteration. This way we keep the symmetry after each iteration, but it is important to note that the solver could assign different values and this could be a cause of instability. However, this is a moot point as a mixed symmetric function space is impossible. It will also use more time and require more memory, but this is of less consequence for our work.

4.13 Implementation of boundary conditions

CBC.RANS already includes code for implementing the boundary condition on ϵ, so we will not cover that. To set up our boundary conditions for F_{ij}, we consider the following system:

$$N = \text{span}\{\mathbf{t},\mathbf{n}\} \quad \text{and} \quad S = \text{span}\{\mathbf{e}_1,\mathbf{e}_2\} \tag{4.13.1}$$

Where $\mathbf{n}$ and $\mathbf{t}$ are unit vectors respectively normal and tangential to the wall. We then have the following transformation rule:

$$\mathbf{A}_N = [\mathbf{t},\mathbf{n}]_S^{-1}\,\mathbf{A}_S\,[\mathbf{t},\mathbf{n}]_S \tag{4.13.2}$$

Where the $\mathbf{A}_N$ denotes a generic matrix/tensor in the N coordinate system with entries a_{ij}. If we write this out fully for 2D, we have:

$$\mathbf{A}_N = \begin{bmatrix} n_2 & -n_1 \\ n1 & n_2 \end{bmatrix}\begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix}\begin{bmatrix} n_2 & n_1 \\ -n1 & n_2 \end{bmatrix}$$

$$= \begin{bmatrix} n_2^2 a_{11} + n_1^2 a_{22} - n_1 n_2(a_{12}+a_{21}) & n_1 n_2(a_{11}-a_{22}) + n_2^2 a_{12} - n_1^2 a_{21} \\ n_1 n_2(a_{11}-a_{22}) + n_2^2 a_{21} - n_1^2 a_{12} & n_2^2 a_{11} + n_1^2 a_{22} + n_1 n_2(a_{12}+a_{21}) \end{bmatrix} \tag{4.13.3}$$

We have the following boundary condition:

$$\mathbf{F}_N = \begin{bmatrix} 10C\overline{u_n u_n} & -8C\overline{u_t u_n} \\ -8C\overline{u_t u_n} & -20C\overline{u_n u_n} \end{bmatrix} \tag{4.13.4}$$

Where we adopt the notation $C = \frac{\nu^2}{\epsilon y^4}$ for brevity. This now gives us:

$$\begin{bmatrix} n_2 & -n_1 \\ n1 & n_2 \end{bmatrix}\begin{bmatrix} f_{11} & f_{12} \\ f_{12} & f_{22} \end{bmatrix}\begin{bmatrix} n_2 & n_1 \\ -n1 & n_2 \end{bmatrix} = \begin{bmatrix} 10CR_{nn} & -8CR_{tn} \\ -8CR_{tn} & -20CR_{nn} \end{bmatrix} \tag{4.13.5}$$

$$\Downarrow$$

$$\begin{bmatrix} n_2^2 f_{11} + n_1^2 f_{22} - n_1 n_2(f_{12}+f_{21}) & n_1 n_2(f_{11}-f_{22}) + n_2^2 f_{12} - n_1^2 f_{21} \\ n_1 n_2(f_{11}-f_{22}) + n_2^2 f_{21} - n_1^2 f_{12} & n_2^2 f_{11} + n_1^2 f_{22} + n_1 n_2(f_{12}+f_{21}) \end{bmatrix} =$$
$$\begin{bmatrix} 10C\left(n_2^2 R_{11} + n_1^2 R_{22} + n_1 n_2(R_{12}+R_{21})\right) & -8C\left(n_1 n_2(R_{11}-R_{22}) + n_2^2 R_{12} - n_1^2 R_{21}\right) \\ -8C\left(n_1 n_2(R_{11}-R_{22}) + n_2^2 R_{21} - n_1^2 R_{12}\right) & -20C\left(n_2^2 R_{11} + n_1^2 R_{22} + n_1 n_2(R_{12}+R_{21})\right) \end{bmatrix} \tag{4.13.6}$$

Though it seems complicated, this equation is simple to construct with inner products, and has been implemented in the different codes in `Wall.py` in CBC.RANS, provided in the last appendix.

One important thing to notice is that we have chosen to use the 'y-definition' of the f_{ij}-boundary condition, for the reasons explained in chapter 3. Since the wall distance is constant for each node, we compute it at the start of each run by using the Eikonal equation [17].

4.14 The different schemes for coupling

As mentioned in the section on CBC.RANS, we will be unable to construct a fully coupled system, as CBC.RANS always uncouples the Navier-Stokes equations from the turbulent model equations. However, there are still several possible schemes for coupling our equations. They are as follows:

(1)	`ER_FullyCoupled`	k-ϵ, Reynolds stresses and f_{ij} fully coupled
(2)	`ER_2Coupled`	k-ϵ coupled, Reynolds stresses and f_{ij} coupled
(3)	`ER_3Coupled`	k-ϵ coupled, Reynolds stresses and f_{ij} uncoupled

We have implemented these three schemes in CBC.RANS and the names for them follow the conventions for naming turbulence solvers in CBC.RANS. They are all subclasses of the class `ER`, which holds the basic variables used by all versions of our system.

The reason that k and ϵ are always coupled is because of the stability of the boundary condition for ϵ. This boundary condition will create a large amount of instability when it is based on old values of k. Since we concern ourselves with Reynolds stresses and ϕ_{ij}, we see no reason to argue with this approach. It is also no perceivable benefit to solve with $\mathbf{F}$ coupled with k and ϵ with $\mathbf{R}$ alone (this reasoning will prove to be justified in the next chapter).

Further, if we assume the unknowns are stored in a vector on the form $[k\,\epsilon\,\mathbf{R}\,\mathbf{F}]^T$, a quick look at the equations show that the matrix for a fully coupled system will take the following form:

$$\begin{bmatrix} KE & KE_R & KE_F \\ R_{KE} & R & R_F \\ F_{KE} & F_R & F \end{bmatrix} = \begin{bmatrix} KE & 0 & 0 \\ 0 & R & R_F \\ 0 & F_R & F \end{bmatrix} \tag{4.14.1}$$

The righthand part of this equation stems from the fact that we can see that there are no implicit contributions from the equations for k and ϵ in the equations for $\mathbf{R}$ and $\mathbf{F}$, and vice versa. This is because in the linearization process we put priority on terms involving the terms the equations was supposed to model. As such, there is little lost when decoupling with the scheme `ER_2Coupled`. We think it is a good assumption that the precision lost in `ER_2Coupled` is more than made up by the faster iterations, so in the end `ER_2Coupled` will use less time in reaching a solution than `ER_FullyCoupled` and be practically just as stable.

Based on the fact that a fully coupled scheme is the most stable, scheme 1 should be the most stable, followed by 2, and 3 should be fastest (but possibly very unstable).

How the matrices and vectors in these schemes will look should be apparent from the previous sections. Instead of writing all three here, an interested reader should consult the appendices as they should be easy to read in the code. They can be found at the end of each of the subclasses.

4.15 Avoidance of negative values

As we know from previous chapters, some of our unknowns cannot have negative values. This holds for k, ϵ, $\overline{u^2}$ and $\overline{v^2}$. However, sometimes an iteration will set these to negative values, and this must not be allowed to happen. Negative values of these variables will quickly create instabilities and might create convergence towards an unphysical solution. To fix this we have included a check which sets all values for these variables which are negative equal to 10^{-12}, that is to say slightly above zero. This check is done after each iteration step, in the `update`-method.

4.16 The function $C_{\epsilon 1}$

CBC.RANS uses a common rewriting of the model constant $C_{\epsilon 1}$, in which it is a nondimensional function:

$$C_{\epsilon 1} = 1.4 \left(1 + C_{\epsilon d} \sqrt{\frac{k}{\max(10^{-10}, \underline{\mathbf{R}} \colon \mathbf{nn})}}\right) \tag{4.16.1}$$

Where $\mathbf{n}$ is the normal vector of the closest wall. Note that this means that the tensor $\mathbf{nn}$ is constant except where there is an equal distance to the closest wall, where it is zero. However, we have not encountered large problems with this, as this will almost never happen with an even number of nodes. The maximum is there to prevent the divisor from becoming zero. We see that for channel flow, this gives us:

$$C_{\epsilon 1} = 1.4 \left(1 + C_{\epsilon d} \sqrt{\frac{k}{\max(10^{-10}, \overline{v^2})}}\right) \tag{4.16.2}$$

Note that since $C_{\epsilon d}$ is very small, this only has an effect when k is much larger than $\overline{v^2}$ (ie, the near-wall) and works to increase the production of ϵ in this region.

4.17 Interaction with `NSSolver`

We have up until now not mentioned how our `ER`-schemes interact with the `NSSolver`. As mentioned previously, CBC.RANS allows manipulation of the viscosity and body forces. The modified version of the first Navier-Stokes equation used by CBC.RANS reads:

$$\frac{D\mathbf{u}}{Dt} = -\frac{1}{\rho}\nabla p + \nabla \cdot \left(\left[\nu + \nu_T\right]\right) \nabla \mathbf{u}) + \mathbf{f} \tag{4.17.1}$$

Where ν_T is some eddy viscosity, and $\mathbf{f}$ are the body forces, generally zero. Now, we generally want to have an eddy viscosity as it adds to stability (following the same reasoning as in our linearization). **f** has little impact on stability, as it in an explicit term on the right-hand side. As such removing the eddy viscosity might be counterproductive. We have instead modified our **f** in the following way:

$$\mathbf{f}^* = \mathbf{f} - \nabla \cdot (\nu_T \nabla \underline{\mathbf{u}} - \underline{\mathbf{R}}) \tag{4.17.2}$$

Mikael Mortensen proposed the eddy viscosity

$$\nu_T = \frac{C_\nu}{T} \underline{\mathbf{R}} \colon \mathbf{nn} \tag{4.17.3}$$

Note that in a channel, this means that:

$$\nu_T = \frac{C_\mu}{T} \overline{v^2} \tag{4.17.4}$$

And it is an attempt to capture this behavior for a general geometry that lead to Mortensen's proposed model. We will not mention this model in the results but we will note here that instabilities never start in the `NSSolver`, which means that at that in our work this modification has not been a problem.

4.18 Model constants and Re_τ

Up until this point we have not given specific values to our model constants. This is simply because we want to keep our explanations as flexible as possible. However, for our computation the values are all taken from their respective articles, and we have reproduced them in the code and in the nomenclature list.

There is also the matter of the dynamic similitude of our computations. All CBC.RANS turbulence problems takes the turbulent Reynolds number Re_τ as input. This number is the ratio between half channel width times friction velocity u_* (from 2.6) and the viscosity. This is rather unimportant for our computational stability since it justs sets values to fit a certain ratio. The reader should consult chapter 4 in [7] for more information on Re_τ. In CBC.RANS the friction velocity is arbitrarily set to 0.05 and the channel width is alway 2, so for our computations this means:

$$Re_\tau = \frac{u_* H}{\nu} \Rightarrow \nu = \frac{0.05}{Re_\tau} \tag{4.18.1}$$

These numbers are important for interpreting the values on the axes of plots in chapter 5. One should note that the value for u_* is used to compute the viscosity and decide the profile for channel flow. In more complex geometries the parameter Re_τ is only used for setting the viscosity and inflow/outflow profile.

It is important to note that all this means that a given model might not give the same bulk velocity Re for a given Re_τ as another model, so comparison between different models should keep this in mind.

4.19 Mesh resolution near walls

It should be clear at this point that an increase in mesh resolution close to walls would be desirable with our current model. Indeed, this is almost always the case in turbulence modeling, and CBC.RANS accounts for this by using a node distribution which sets the distance between nodes to be much shorter close to a wall. The basic mesh in CBC.RANS is the FEniCS `Rectangle`-domain, usually set to be a rectangle $[0,1] \times [-1,1]$ with walls along $y = -1$ and $y = 1$. The nodes are then moved closer to the wall as an 'arctan-distribution'. If y_i is the y-coordinate for a given node, it is given a new value y_i^* following this formula:

$$y_i^* = \frac{\arctan \pi y_i}{\arctan \pi} \tag{4.19.1}$$

This means that there will be a significantly finer mesh resolution near walls. For more complicated meshes, CBC.RANS takes this new rectangle mesh and performs a second transformation to turn it into the problem geometry.

4.20 Initial guesses

Our system is now almost complete, there is just more part missing: Which values to use as the values of $\mathbf{x}^0$, what we call the 'initial guess'. This is very important, as an initial guess close to the solution will use few iterations to reach it, and has a smaller chance of instabilities caused by strange values (like negative values for nonnegative variables). The initial guess is handled in CBC.RANS by the `TurbProblem` without input, though in a rough manner. All complicated geometries begin by setting the entire domain in the main flow direction to the profile for a 2D channel, squeezed or stretched to fit the given cross section. This profile is obtained from a saved file created after finding the solution to the channel geometry in `channel.py`. `channel.py` sets the initial guess as $\mathbf{x}^0 = \mathbf{0}$ and this is a very bad initial guess. However, we will be able to get results, as the next chapter will show.

4.21 Error estimates

To see if our iterations are approaching the solution, we need an error estimate. This is done in the normal way by looking at the residual of 4.7.1. As such we can se that the error/residual r for a given iteration is given as:

$$r(\mathbf{x}^n) = \|\mathbf{E}(\mathbf{x}^n) \cdot \mathbf{x}^n - \mathbf{f}(\mathbf{x}^n)\| \tag{4.21.1}$$

We can see that if $\mathbf{x}^n$ approaches the solution to 4.7.1, $r(\mathbf{x}^n)$ approaches zero.

4.22 Running our code

Up until now, we have not mentioned HOW to run CBC.RANS. CBC.RANS works best when used in the `Ipython` environment [3]. When in this environment and the `/turbproblems/`-folder, a simple run is done by writing:

```
run test --p channel --vd 2 --n ER_2Coupled --m LRR-IP --Nx 6
    --Ny 120 --max_iter 100 --wu 0.6 --wt 0.4 --Ret 395.
```

This will compute the flow field for a 2D channel with 6 nodes in the x (flow-vise) direction and 120 in the y-direction. It will run to 100 iterations, and θ is 0.6 for the `NSSolver` and 0.4 for `ER_2Coupled`. The pressure gradient and viscosity is computed to fit with a turbulent Reynolds Number $Re_\tau = 395$.

4.23 Reading the code

When writing code, it is desirable to be able to be able to discern the nature (scalar, vector or tensor) of a variable by simply looking at it's name. Unfortunately, this is very difficult to accomplish when working with simple text editors. As such, we have not tried to include a notation on vectors, as the only vector is the unchangeable `u`. We have however tried to distinguish between scalars and tensors by adding the suffix `-ij` to them. This explains why $\mathbf{R}$ is referred to in the code as `Rij`. Note also that ϵ has been renamed `e`, for obvious reasons.

We have included the code we have written as appendices, but due to the large number of indents in some of the lines and the restrictions on color pages, we will not advice the reader to consult these pages. Instead, they should access the CBC.RANS Launchpad site [2] and view the code online (go to the 'Code' tab, then 'Browse the source code'). The basic color formatting used by Launchpad makes the code very readable online, much more so than the appendices in this text. In addition the reader will be able to access the other parts of the framework, which we cannot include as appendices (as this would take up 50+ pages).

Chapter 5

Results

5.1 Goals and limitations

Before discussing my results, it should be pointed out that the major goal of this work was implementation and stability. As such, most of the time has been spent trying to find stable solutions for complicated geometries than comparing channel flow results to DNS data. This has in some ways led to failure as much time has been used trying to get convergence without success. Therefore, the goal here cannot be much more than to describe the results qualitatively. As such, I am more concerned with the shape of curves than their exact values, insofar these values make physical sense (as it will be seen, sometimes they do not). This is also the reason for the somewhat lackluster graphs and plots, as I have not had the time to learn how to manipulate the `.vtk` files produced by FEniCS.

5.2 Comparison with DNS data

I have compared my results with the DNS data in [14]. The results in that book are for a bulk velocity Reynolds number of 13,750. Since the code computes the mean velocity, the results do not have the same bulk velocity Reynolds number for each solution. But if an average velocity across the channel of around 0.8 times the maximum value of U is assumed, one gets a Reynolds number between 11,000 and 15,000 (these are rough estimates, it is realistic to expect a Reynolds number around 13,000). Although this means one cannot directly compare the results with the data in [14], the Reynolds numbers are close enough to be able to use them for a qualitative analysis. The variables have therefore not been scaled or changed. This might mean that the results are difficult to compare with other results, and I apologize for this.

The y-axis has also been marked in 'y plus' wall units for plots of sections of the flow near the wall. This is to make the plots more readable to students of turbulence modeling. Since this is a very minor use of wall units, it felt unnecessary to include a section on wall units and refer instead to [7]. For reference, $y^+ = 40$ equals $y = -0.9$ and $y^+ = 0$ equals $y = -1$ in these plots.

5.3 Lack of convergence with ER_3Coupled

Of the three schemes outlined at the end of the previous chapter, ER_3Coupled is unusable. The scheme is highly unstable, to the extent that it seems impossible to find a solution. Even with a very small ($\sim$0.05) relaxation parameter θ, the solver eventually becomes unstable and never converges. I found that a small θ would avoid some of the instabilities (by observing at which iteration number the solver diverges), but not all. It is possible that the scheme is stable for an incredible small θ or very good initial values, but this is a moot point as one of the more coupled solvers will converge faster.

The most probable cause for this instability is the boundary conditions for $\mathbf{F}$ for the same reasons that a system with decoupled k and ϵ will be unstable, as the boundary condition is mathematically very similar. Since $\mathbf{F}$ and $\mathbf{R}$ have a set of 3 such boundary conditions there is good reason to believe that this is the case.

For the remainder of the text, the results will be with ER_2Coupled and ER_FullyCoupled. These will give the same solution, but differences in convergence will be noted where necessary.

5.4 Results with LRR-IP

(This section distinguishes between LRR-IP with and without Elliptic Relaxation. For all later sections, any reference to LRR-IP implies the ER version.)

Of the models for the homogeneous redistribution tensor, I easily got results with LRR-IP in channel flow. Although good results in channel flow have already been achieved with several models and even with Elliptic Relaxation [8], these have not been with FEM. These result are also a nice justification for the ER modification of homogeneous model, as will be shown shortly. Since the SSG model is much more complicated, I will try to show simple properties of ER with LRR-IP.

To compare with a homogeneous version, I simply removed the Laplacian term from the equation for $\mathbf{F}$. The other terms in that equation are kept, as this restores the original system before ER modification, though with the ϕ^*_{ij}-modification described in chapter 3. The only modification from the standard LRR-IP model is then the use of the boundary conditions for $\mathbf{F}$, but as these are developed without any relation to Elliptic Relaxation they should not pose a problem in the physical sense. A version with Dirichlet boundary conditions gives an almost identical solution, which backs up this statement. But because they are potentially inappropriate for the LRR-IP model, it is important to note that one should not consider these results as those the LRR-IP model is expected give. That is to say, this is a examination of what the Laplacian term in the ER-modification does, not a direct comparison between LRR-IP and LRR-IP with ER.

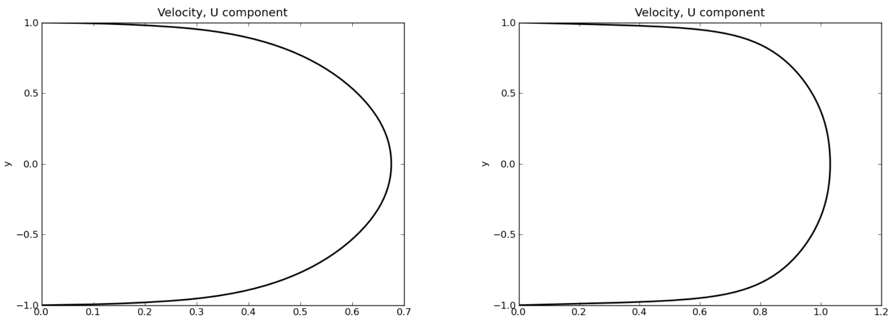


Figure 5.1: Plots of LRR-IP velocity without (left) and with ER (right)

Initially, one can see from the results that both versions produce results which seem reasonable. The values of k and ϵ around $y = 0$ are very similar, which is to be expected as the ER version approaches the homogeneous model. The unmodified LRR-IP produces a mean velocity profile very similar to the standard $k - \epsilon$ implemented in CBC.RANS.

The simplest difference between the two is the mean flow profile. Without ER the profile is much closer to the y^2-curve of the laminar solution, while the ER solution is much closer to the desired 'slug' shape as seen in DNS and experiments. This effect is tied in with the greatly increased production of k and $\overline{u^2}$ close to the wall, since $\partial_y U$ is much steeper with ER. This is helped by the fact that the ER modification reduces the negative f_{11} close to the wall, decreasing the magnitude of the other stresses near the wall since less energy is transfered from $\overline{u^2}$ to $\overline{uv}$ and $\overline{v^2}$. These factors combined lead to the almost double maximum value of $\overline{u^2}$ and the 50% increase in maximum k. These higher values decrease much faster away from the wall, leading to similar values in the middle of the stream for both versions (with some differences, see below). From this it seems reasonable to assume that the ER modification works fairly well, as the mean velocity profile seems appropriate and the general distribution of the Reynolds stresses roughly fit with DNS data [14]. There are however three very interesting differences, which concern ϵ near the wall, the difference between $\overline{u^2}$ and $\overline{v^2}$ and the value of $\overline{uv}$. The first of these is encouraging, the two latter are more troubling.

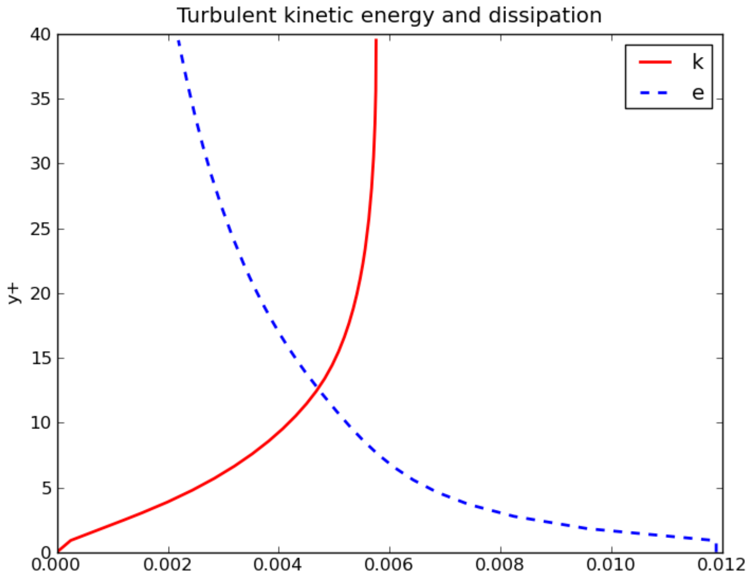


Figure 5.2: k (m^2/s^2) and ϵ (m^2/s^3) for LRR-IP without ER

Concerning the first, it is seen that without ER the curve of ϵ is simple and strongly increasing close to the wall (see fig. 5.2). This is a common behavior in many models, but not exactly correct. It is possible to observe that very close to the wall, one in fact gets an area where ϵ decreases before it again grows (see fig. 5.3). What is remarkable is that that by adding the ER modification to the LRR-IP model (which has no such behaviors of ϵ)this behavior occurs! This is even more encouraging as the ER model or any other part of the model has not developed with approximating this behavior in mind. That this behavior then appears in the ER version would seem to indicate that the ER modification is in some way modeling near-wall flow more correctly. It is a very simple proof of concept for the initial assumption that making the redistribution tensor elliptic would better model the physical effects in the near-wall region.

As to the difference between $\overline{u^2}$ and $\overline{v^2}$ (se fig. 5.4 and 5.5), I must make an educated guess as to the source of the difference. Whereas $\overline{u^2}$ is significantly higher than $\overline{v^2}$ in the unmodified version, the ER version predicts a $\overline{u^2}$ which is smaller than $\overline{v^2}$. The former is understandable as there is significant production of $\overline{u^2}$ away from the wall due to $\partial_y U$ not being zero. The latter ER version behavior is more difficult to explain. One reason is clearly the reduction in production away from the wall. A reduced production will naturally make the equations for $\overline{u^2}$ and $\overline{v^2}$ much more similar. With ER

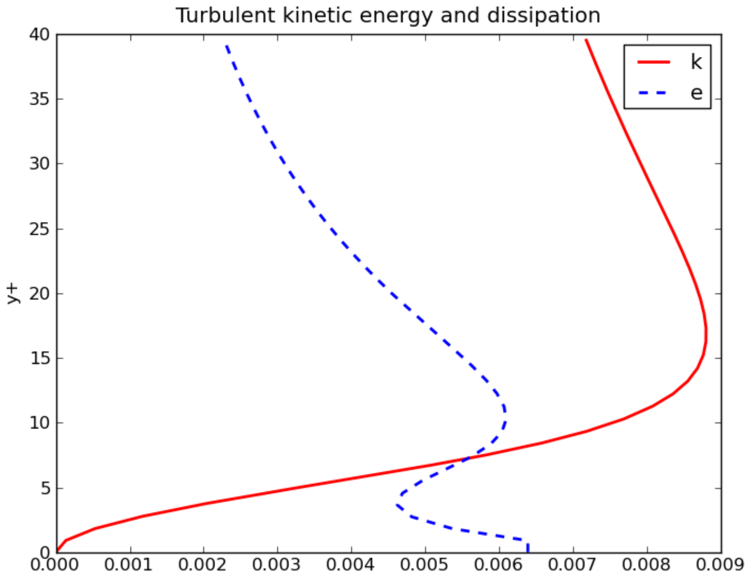


Figure 5.3: k (m^2/s^2) and ϵ (m^2/s^3) for LRR-IP with ER

the model predicts a larger ϵ in the area between the near-wall and the middle of the flow, which should also work towards reducing the Reynolds stresses away from the wall, making them more equal. However, this does not explain why $\overline{u^2}$ becomes smaller than $\overline{v^2}$ Regardless of cause, I found it interesting that whereas the homogeneous LRR-IP overpredicts $\overline{u^2}$ in the middle of the flow, when modified with ER it underpredicts it.

If the $\overline{u^2}$ difference was disconcerting, the $\overline{uv}$ difference is very disturbing. Before this can be discussed fully, one of the results from 2.6 must be repeated. If one assumes $\partial_y U$, it was found in 2.6.3 that the behavior $\overline{uv} = u_*^2 y$ should be seen away from the wall. In this case, it means the behavior $\overline{uv} = 0.0025y$. It is clearly seen that the unmodified LRR-IP version has this behavior. However, the modified version does not. In fact, it seems that it has the behavior $\overline{uv} = \frac{1}{2}u_*^2 y$.

This is very confusing. Since both versions use the same code, an error in constants should have shown up in the non-ER version too. Further, as the results with SSG will show, there does not seem to be a general problem with code. If this was a problem with any other variable than $\overline{uv}$ I could have easily blamed this on a bad model and moved on, but that Reynolds stress should follow 2.6.3 exactly. Saying the system has converged to an unphysical solution also seems somewhat dubious. The other variables in the system take reasonable, if not correct values. Also, the velocity profile

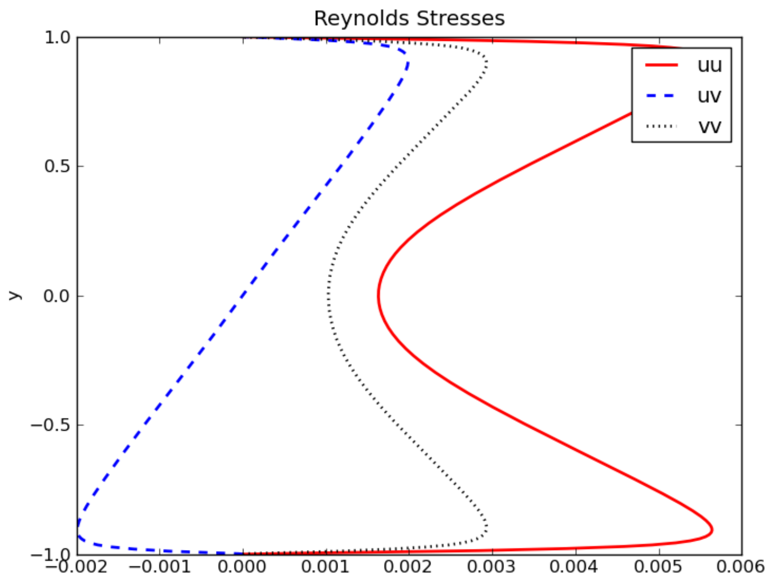


Figure 5.4: Reynolds Stresses (m^2/s^2) for LRR-IP without ER

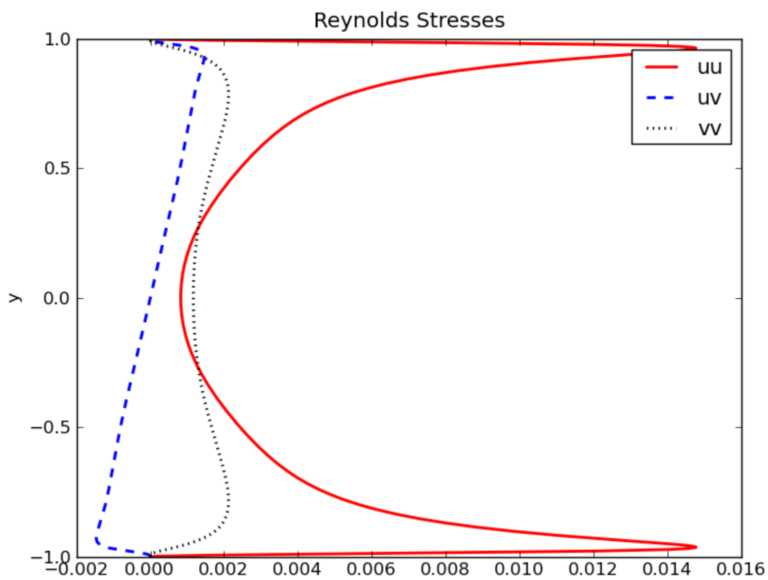


Figure 5.5: Reynolds Stresses (m^2/s^2) for LRR-IP with ER

is almost the same as for the SSG model. A very wrong $\overline{uv}$ should have given a different velocity profile. I was unable to find the error responsible for this, and the lack of explanation for this behavior must be left as the biggest failure of this work.

There is one last point to note about computation of these results. Both are computed with Ny = 400 and the ER_2Coupled scheme, and relaxation parameter 0.6 for the Navier-Stokes solver and 0.4 for the turbulence solver. The non-ER model converged in 79 iterations (1424s), while the ER model used 112 iterations (2350s). It seems reasonable to assume that the difference in number of iterations is based on the suitability of the initial guess and that more iterations are needed to get the exact shape near the wall with ER.

5.5 Results with SSG

I was unable to get any results with the SSG model without good initial values, as both schemes proved highly unstable even with very small (~0.01) relaxation parameters. However, I observed that a smaller relaxation parameter (for both the N-S and turbulence solver) would delay divergence and even show signs of convergence for a number of iterations before diverging. Here the problem is not the ER modification, as runs without the Laplacian term proved equally unstable. There are several possible reasons for numerical stability, but they will wait until after discussing the results which I got with a more cautious approach.

To remove the problem with initial guesses being bad, the result of the LRR-IP model was used as an initial guess, with very small (0.05) relaxation parameters. This yielded results, but only after a great amount of time: The resulting run converged after 1205 with ER_2Coupled (8h, 40mins). With a larger relaxation parameter it did not converge. From this, I feel confident in concluding that the SSG model is highly sensitive to initial guesses and relaxation parameter.

Looking at the Reynolds stresses in figure 5.6, the SSG solution seems to fix most of the problems with LRR-IP. In the near-wall region the SSG solution has very similar results. There is a difference in the magnitude of $\overline{u^2}$ and k with an increase at the maximum close to the wall of roughly 15%. The SSG model also produces the near-wall change in ϵ. There is no region where $\overline{v^2}$ is larger than $\overline{u^2}$, but the two Reynolds stresses are almost equal at $y = 0$. It can be seen that the solution clearly has the behavior $\overline{uv} = u_*^2 y$ away from the wall. This is all very encouraging. Another thing to note is that $\max(\overline{u^2}) \simeq 7u_*^2$, which is roughly the desired result.

Summing up, the SSG solution fits very well with the DNS data in [14]. I would have liked to compare my results with those in [8], but the authors have there made a plot of ϕ_{ij}, not f_{ij}, which makes it difficult to compare

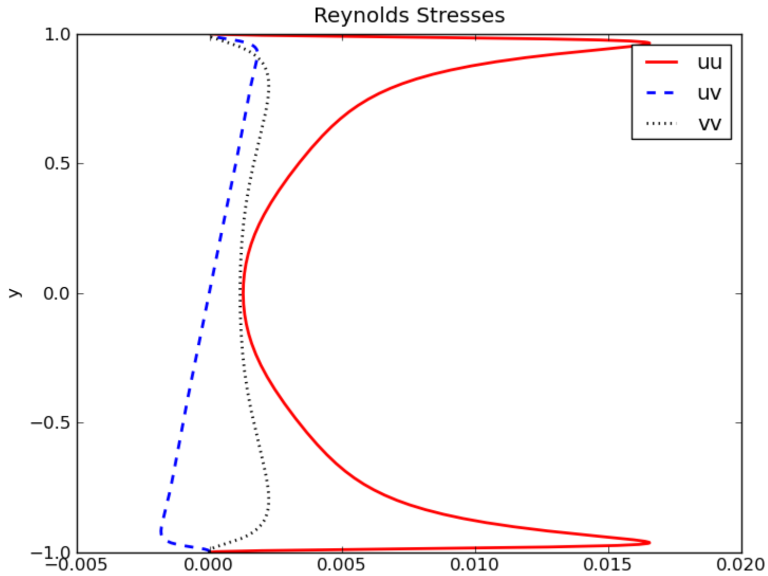


Figure 5.6: Reynolds Stresses (m^2/s^2) for SSG

different ER models. My values for $k\mathbf{F}$ look very similar, but it would be much more interesting to examine the values of f_{ij} directly, as they would show potential differences much clearer. I have included several near-wall plots of both SSG and LRR-IP in appendix A, so that the reader might consult them for further investigation.

5.6 SSG instabilities

My theory is that the instabilities experienced with SSG stem from the fact that ϕ^h in the SSG model is almost wholly explicit. Luckily, this is one of the strengths of the CBC.RANS-framework, since linearization is easily changed. However, several simple variants with implicit linearizations did not bear fruit. The only conclusion I could reach was that the instability was not caused by the last ('C_5') term, as setting all C's equal to zero except for C_5 resulted in a stable system.

As to the exact causes for the instabilit

- Linearization
- Initial guess
- Mesh

· Solver algorithm

· FEM

Of these, linearization and initial guesses have already been mentioned. I do not believe that the mesh should be a problem with a reasonable resolution (Ny > 100), but it is possible that the ER modification of SSG creates results which are unattainable with a too coarse mesh, so I do not disregard it outright.

Concerning the solver algorithm, I believe that a more complicated algorithm might work better. However, I am unable to change this facet of CBC.RANS and do not have sufficient competence to devise a better solution. There is a possibility that a more advanced solving algorithm than Picard Iteration would be more stable, though in this regard the boundary conditions for ϵ and f_{ij} create problems. For example, the Newton-Raphson method is very easy to implement in FEniCS, but does not work with these variable boundary conditions. Another might be the fact that if the system was fully coupled with the velocity-solver, many of the terms in the SSG model could be set as implicit with regards to the velocity.

I am then left with the possibility that the instabilities stem from some problem with using a system designed for FVM or FDM in a FEM framework. As previously mentioned, there is much skepticism in the CFD community regarding the use of FEM. I do not disregard the possibility that there are effects in a FDM/FVM-system which are not properly handled by naively implementing it in FEM. However, it seems too simple to disregard FEM out of hand. These FEM effects could probably be handled by advanced methods developed for FEM. As before, I must leave this question open due to lack of knowledge.

5.7 Results with the diffusor geometry

I have made several attempts at getting my code to work with the diffusor geometry, after Mikael Mortensen got convergence with a coarse mesh using LRR-IP [10]. In the week before finishing this thesis, I was successful, because I detected an error which had previously led to instabilities. Two figures have been provided of the solution, but I have not been able to compare it with other data, and as such all I can say is that the solution does not look unphysical. I will now try to explain the error which prevented convergence.

As mentioned before, the Reynolds stresses in the RANS equations appear in the term:

$$-\partial_k \overline{u_k u_i} = -\nabla \cdot \mathbf{R} \tag{5.7.1}$$

Which in channel flow means:

$$-\partial_y \overline{vu} \tag{5.7.2}$$

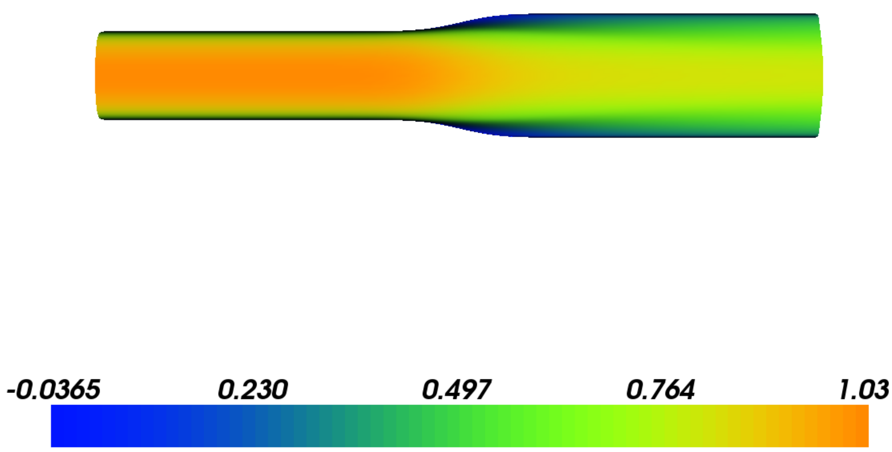


Figure 5.7: Velocity component U (m/s) for LRR-IP in diffusor

With symmetric tensors, this is the same as $\partial_y\overline{uv}$, but the code is not working with symmetric tensors (see 4.12). Note also that the equation in 4.11.1, with our linearization, allows for a solution in which $\mathbf{R}$ is asymmetric. I assumed the most correct solution to this problem was setting $\overline{uv} = \overline{vu}$, since the latter was the most important term. But this was wrong. It seems that to instead set $\overline{vu} = \overline{uv}$ makes the system more stable. This was not a problem for channel flow (though changing it did reduce the number of iterations needed for convergence). Although I can offer no definitive explanation, I think the problem lies in the way FEniCS handles the `div`-operator. In common notation we have

$$\nabla \cdot \mathbf{R} = \partial_k R_{ki} \tag{5.7.3}$$

Whereas the assumption in FEniCS is that the operator works on the latter index, that is:

$$\texttt{div}(\mathbf{R}) = \partial_k R_{ik} \tag{5.7.4}$$

If this is correct, then $\overline{uv}$ is the more important variable. This is just an error in implementation, which was unfortunately fixed to late. However, it should not be a reason for instabilities with SSG, as the set of equations for

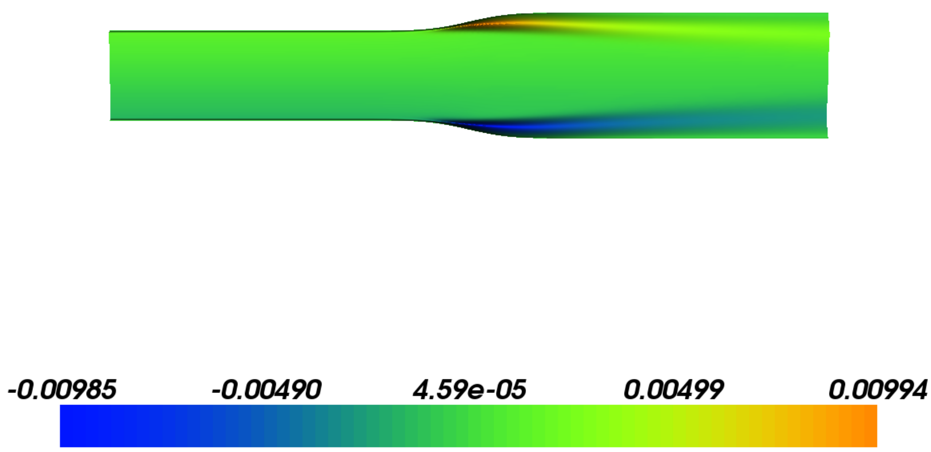


Figure 5.8: $\overline{uv}$ (m^2/s^2) for LRR-IP in diffusor

SSG must be symmetric since ϕ^h is symmetric (since it only uses old values, which are set to be symmetric).

It is important to note that this problem is one which will disappear as soon as mixed function spaces of symmetric tensors become available in FEniCS (which should not be that far away). My problem was specifically caused by the fact that there is very little mention in the FEniCS documentation about tensors, so while they are implemented, there is little documentation of how the package works with tensors. This then causes problems when the user assumes one notation and FEniCS is in fact using another.

In any case this is a proof that the code works, but the solution is probably not usable for comparison with DNS data, since it is reasonable to assume that the results will have the same problems as those in channel flow. This proof of convergence is important, however, as it proves that there is nothing fundamentally wrong with the code.

It is important to note that with this geometry the initial values for the first iteration is the channel profile stretched to fit a given cross section, and this puts a limitation on how coarse the mesh can be. Solutions for channel flow with Ny < 60 are not physical solutions. Or rather, they do not capture the correct near-wall behavior, and converge towards a solution which does not resemble DNS data. These solution tend to result in a flow profile which has the correct 'slug' shape but with a a significantly lower maximum U. As such, one must consider the ER model to be unsuitable for a mesh with fewer than 100 nodes for a given cross section (in most cases this means Ny > 100). Here 100 is an arbitrary limit, as I did not try to find the exact limit for this behavior. Convergence is possible with a coarser mesh, but I am erring on the side of caution. I also observed that at Ny $= 100$ the channel flow profile is mostly correct. It is in my opinion necessary with Ny > 120 for good results, but at least the solution resembles DNS data for Ny $= 100$. There is of course the possibility that the problem stems from the fact that the channel flow profile is a bad initial guess, but my only other choice would be to compute Reynolds Stresses from a eddy-viscosity model. There is little reason to believe that this initial guess will not also be far from the solution sought by the solver.

It is possible that some of the issues with convergence for SSG model are relevant with convergence on this mesh, but again my lack of sufficient training in numerical analysis stops further investigation. As such, I must leave this as something of a mystery for future researchers to solve.

5.8 The apbl geometry

It was the original goal of this thesis to compute results for the apbl geometry [16] with SSG. However, as the preceding part of this chapter has shown, I met many problems with much simpler geometries. My few attempts at computing the flow field in apbl have failed, probably for the same reasons as with the diffusor. It is my belief that a scheme which removes the instabilities found in SSG with the diffusor will work on the apbl domain.

The next chapter will discuss what conclusions can be drawn from these results.

Chapter 6

Conclusions

6.1 Successes

The previous chapter might give the impression that I have not achieved much in my work with implementing ER, so I would like to reiterate the following results which I feel are noteworthy:

- I have implemented Elliptic Relaxation for a general 2D framework, which can be easily extended to 3D when that becomes available in FEniCS. This is proof that it is possible to implement complicated DRSM turbulence models in FEM.
- This implementation properly captures effects close to the wall (the ϵ-squiggle), reinforcing the justification and assumption at the heart of the Elliptic Relaxation method.
- I have shown that the CBC.RANS-framework supports complicated DRSM-models, which reinforces the claim that CBC.RANS is a flexible and highly useful framework for numerical research into turbulence modeling.
- The LRR-IP code converged even with a very bad initial guess after very short time, though the solution had several problematic issues. But if nothing else, this can serve as a cheap (in terms of time) and 'almost correct' initial guess for other ER models.
- The LRR-IP code converged for the more complicated diffusor geometry, proving that the code works for a general geometry.
- The SSG code converged to a seemingly good solution for channel flow, which serves as a proof of concept.
- I have investigated the stability of the different coupled schemes possible, identified several problems with my approach and found room for improvement.

Though the current implementation in CBC.RANS is unable to find solutions for complex problem geometries, convergence with channel flow give an indication that implementing Elliptic Relaxation in FEM should be possible. It is important to remember that the current implementation is of the most naive sort, and many of the problems might have surprisingly simple fixes. For example, changing C_{ϵ_1} from a constant to a function greatly increases the stability of the LRR-IP model, and this modification adds close to nothing in time cost. I think it is quite possible that a collection of simple fixes might be all that is needed to make the system converge.

For the remainder of this short chapter, I think it is most useful for me to ponder the failures which were touched upon in chapter 5, as they are clear indicators of possible areas of improvement.

6.2 Failures

The failure to get convergence for complex geometries and the errors in the solution for LRR-IP leave much to be desired of the code. It seems reasonable to think that as these equations have been implemented successfully in FDM, a FEM code should work. This must then be recognized as a failure on my part. I also recognize the fact that I am not that knowledgeable when it comes to experimental/DNS data, but have assumed this to be a symptom of lack of experience in the field of turbulence modeling.

6.3 Lack of numerical analysis

At the beginning of my work on this thesis, I had no experience in implementing systems more complicated than the wave equation in 2D. Coming from the last class to receive basic programming courses at the university in Java, I had to acquaint myself with Python, Finite Elements and numerical methods for Navier-Stokes. This was by far the biggest challenges in this work, as I had a much stronger background with regard to theoretical turbulence modeling. Understanding the FEniCS package proved initially difficult because the documentation was written for a readership with a different background and with a different focus than what I was used to. As such, the biggest part of the work done to complete this thesis is entirely hidden from the reader, since the CBC.RANS code is deceptively short. But to arrive at that short code, significant understanding of FEM and how FEniCS uses it was required. CBC.RANS represents, however, a very robust and useful framework when understood correctly, and once properly understood the implementation process was surprisingly fast (after, of course, some debugging).

It seems clear to me that a stronger background in numerical analysis would be required to investigate the convergence problems with the SSG model and the complicated geometries, and maybe the possibility of finding more exact stability estimates. It is also quite probable that other numerical methods which I am unaware of could be used to speed up the convergence rate.

6.4 The critique of FEM

I do not intend to throw myself into the debate over the usefulness of FEM for CFD except to again note that this work should prove encouraging, even if it was not greatly successful, for further implementation of DRSMs in FEM. It seems to me, however, that the articles I have read on turbulence modeling ([8], [7], [15]) tend to omit matters of implementation, which could be an indication that the authors do not consider such things to be important. That the same community then disregards FEM as useful for CFD is in my opinion somewhat perplexing. But at the same time the arguments presented in [12] seem to sidestep entirely the problem of turbulence, having little to no mention of turbulence modeling in a work on numerical methods for fluid flow.

The polarization of this debate and the clear partition of research seems to me to be detrimental to the CFD community. Both approaches clearly produce results, and an approach which combines the methods could possibly lead to greater productivity. This work should show that to solve complicated computational problems like modeling Elliptic Relaxation, both knowledge of numerical analysis and turbulence modeling is needed. And for a user well-versed in FEM and turbulence modeling, FEniCS and CBC.RANS represent powerful tools for numerical experimentation with complex problems.

6.5 The Matter of Uniqueness of Solution

One nagging problem brought to light by the convergence with coarse meshes and the $\overline{uv}$-behavior with LRR-IP is the possibility of multiple solutions to the equations. This poses a twofold problem. Firstly, it is difficult to know if the solution is unphysical or just a correct bad approximation at first glance. This is not a critical problem, but it means that all results must be closely scrutinized before accepting them, requiring that one must have results to compare with. The second more serious issue is: Assuming one of the possible solutions represents the wanted physical solution, one would need an initial guess as close as possible to that desired solution. Both parts of this problem create a sort of 'chicken and egg' situation. To be sure the solution is the correct one, the researcher needs access to either a very close initial guess or data of the wanted solution. This is well illustrated with the LRR-IP and the SSG results. If the LRR-IP result is an unphysical solution, then it seems reasonable to assume that to get the right solution an initial guess closer is needed, like the SSG solution. But the SSG solution was obtained using the LRR-IP solution! And we could only assert that the SSG solution was correct by comparing it with DNS data. This twofold problem must be addressed before the model can be used to predict results, as it currently must be compared to a known solution.

I understand that this is more or less an unsolvable problem, as the proof of uniqueness of solution for the Elliptic Relaxation system is unlikely to be found before the same for the Navier-Stokes equations.

6.6 Future work

The preceding sections have hopefully outlined possible areas of future investigations. It is my hope that my work and results can be used as a stepping stone for more substantial results regarding implementation of DRSMs in FEM.

Bibliography

[1] http://fenicsproject.org/.

[2] https://launchpad.net/cbc.rans.

[3] http://ipython.scipy.org/moin/.

[4] *NASA EP-89*. 1971. http://history.nasa.gov/SP-4103/p529.jpg.

[5] Daly, B., and Harrow, F. Transport equations of turbulence. *Phys. Fluids 13* (1970), 2634–2649.

[6] Durbin, P. A. Near-wall closure modelling without 'damping functions'. *Theoretical Computational Fluid Dynamics 3* (1991), 1–13.

[7] Durbin, P. A., and Reif, B. A. P. *Statistical Theory and Modeling for Turbulent Flows*. John Wiley & Sons, 2001.

[8] Durbin, P. A., and Reif, B. A. P. *Closure Stategies for Turbulent and Transitional Flows*. Cambridge University Press, 2002, ch. 4, pp. 127–152.

[9] Hoffman, J., and Johnson, C. *Computational Turbulent Incompressible Flow*. Springer, 2007.

[10] Langtangen, H. P., Mortensen, M., and Myre, J. CBC.RANS a new flexible, programmable software framework for computational fluid dynamics. In *Konferanse i beregningsorientert mekanikk (Mekit11)* (2011).

[11] Langtangen, H. P., Mortensen, M., and Wells, G. N. A FEniCS-based programming framework for modeling turbulent flow by the reynolds-averaged navier-stokes equations. *Advances in Water Resources* (2011).

[12] Mardal, Logg, and Wells, Eds. *Automated Scientific Computing*. Springer, 2010. https://launchpad.net/fenics-book.

[13] P.Jones, W., and E.Launder, B. The prediction of laminarization with a two-equation model of turbulence. *International Journal of Heat Mass Transfer 15* (1972), 301–314.

[14] Pope, S. B. *Turbulent Flows.* Cambridge University Press, 2000.

[15] Speziale, C. G., Sarkar, S., and Gatski, T. Modelling the pressure-strain correlation of turbulence: an invariant dynamical systems approach. *Journal of Fluid Mechanics 227* (1991), 245–272.

[16] Stanislas, M., Foucaut, J. M., and Kostas, J. Investigation of near wall turbulence structure of an APG TBL using double SPIV. Laboratoire de Mecanique de Lille.

[17] Tucker, P. G. Differential equation-based wall distance computation for DES and RANS. *Journal of Computational Physics 190* (2003), 229–248.

[18] White, F. M. *Viscous Fluid Flow.* McGraw Hill, 2006.

Appendix A

Near-wall graphs

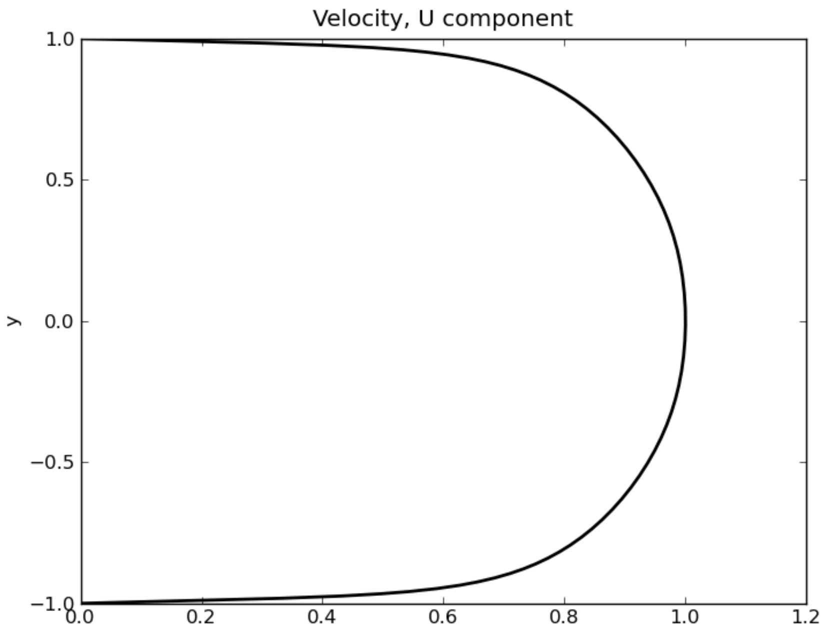


Figure A.1: Velocity (m/s) for SSG

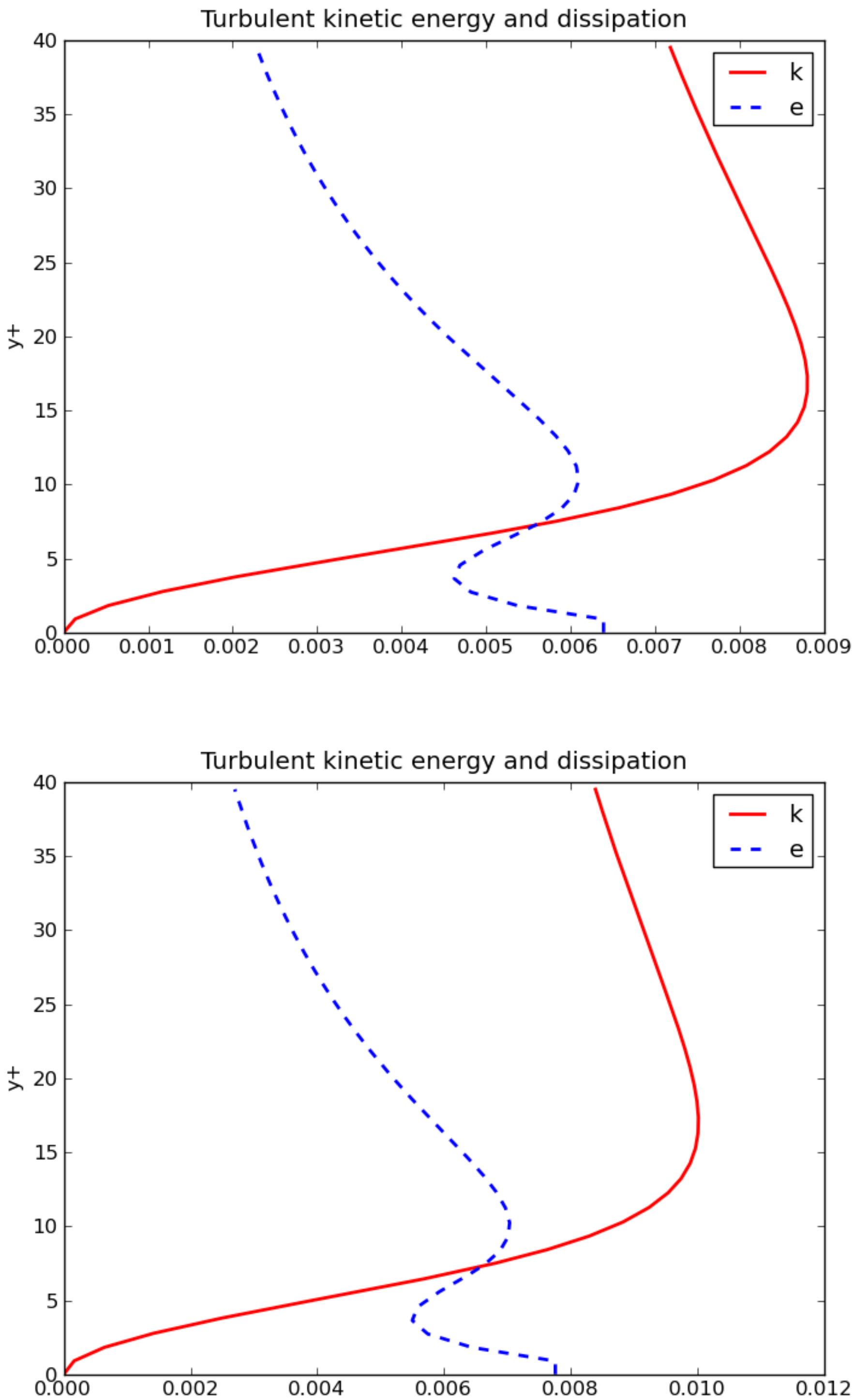
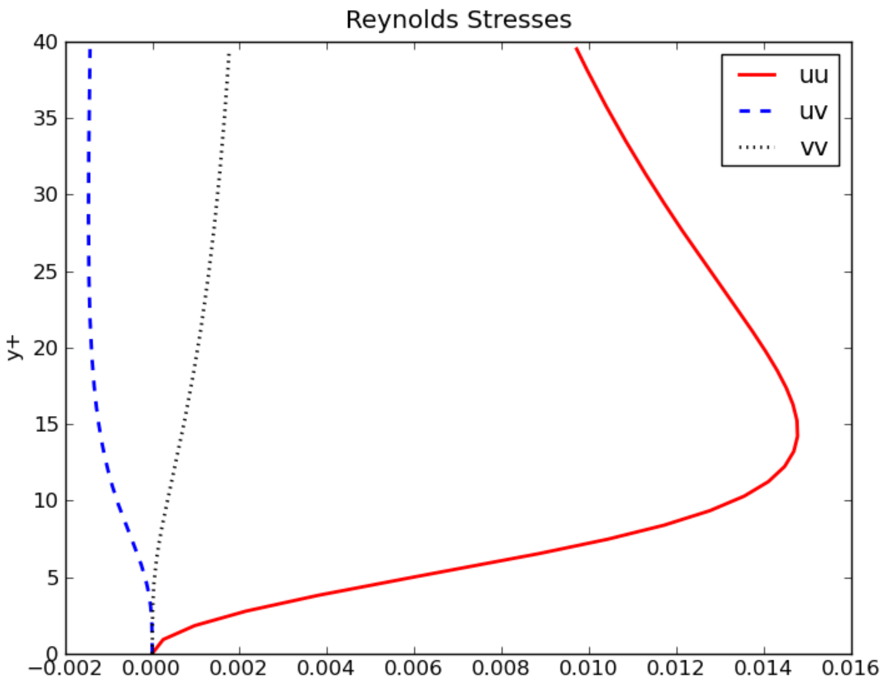


Figure A.2: k (m^2/s^2) and ϵ (m^2/s^3) for LRR (top) and SSG (bottom)



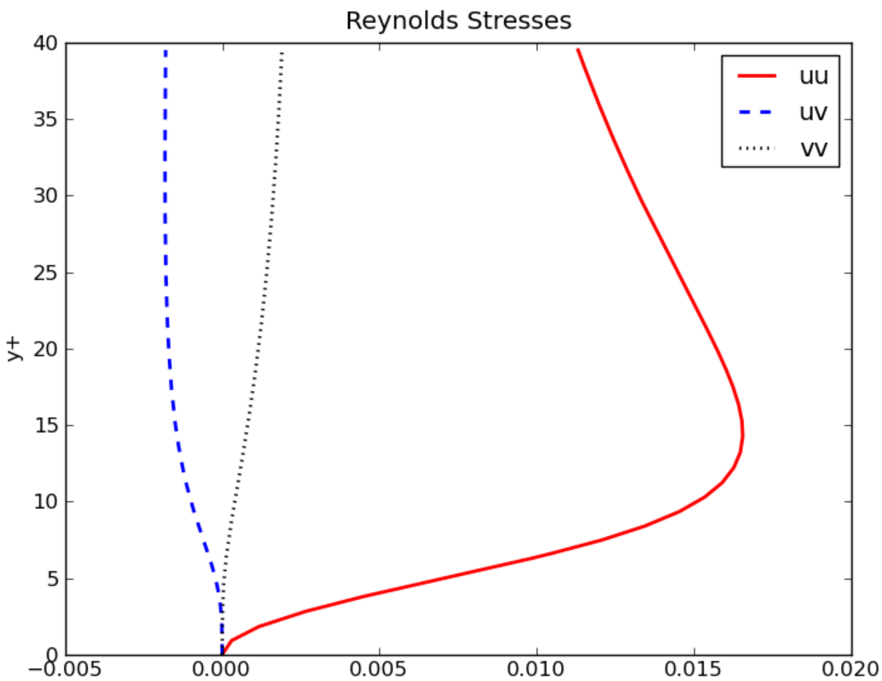


Figure A.3: Reynolds Stresses (m^2/s^2) for LRR (top) and SSG (bottom)

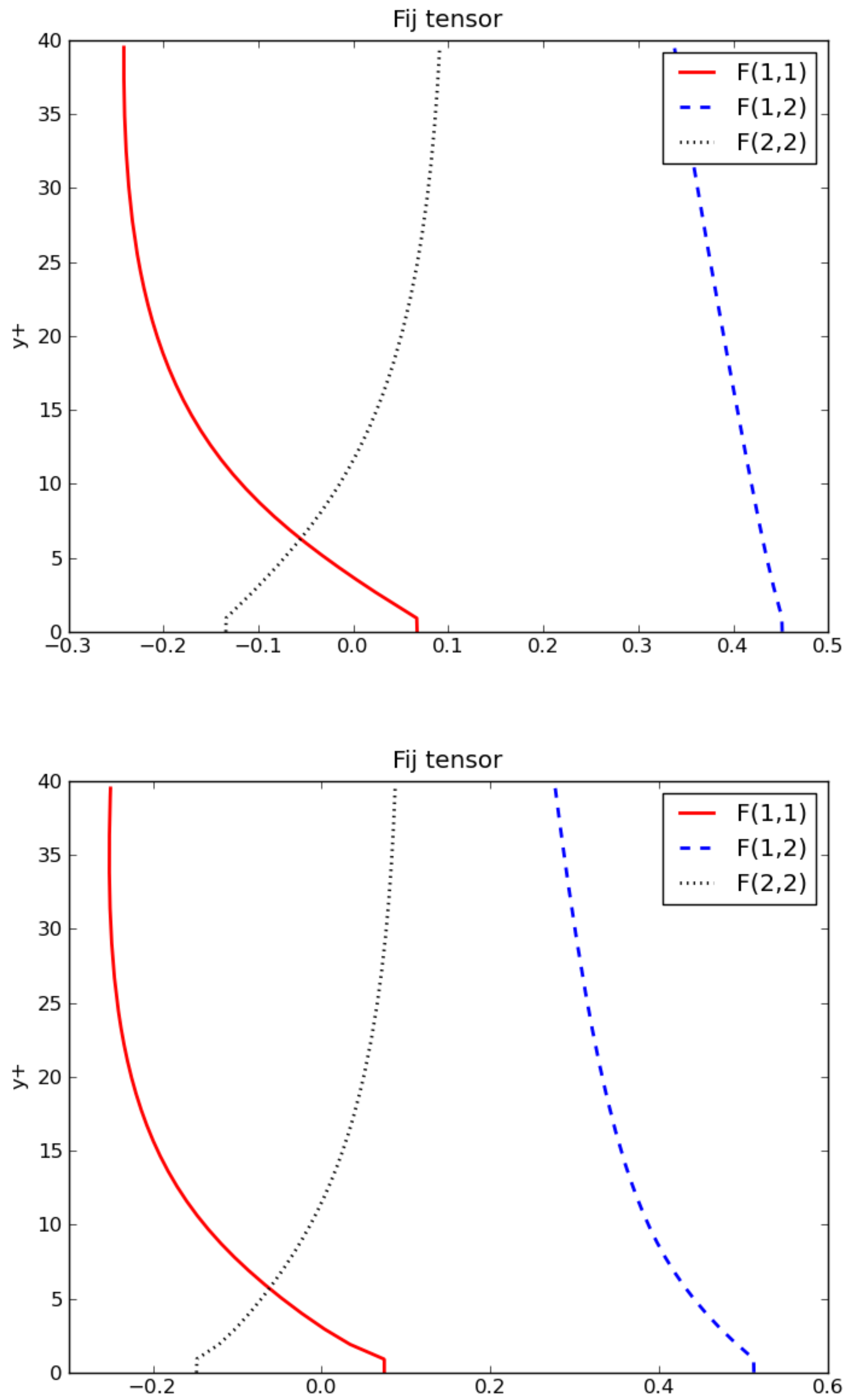


Figure A.4: $\mathbf{F}$ (s^{-1}) for LRR-IP (top) and SSG (bottom)

Appendix B

CBC.RANS-code

B.1 Turbsolver-subclass ER

This is the subclass of Turbsolver which holds all common scales, variables and derived properties for an ER system.

```
__author__ = "Jorgen Myre <jorgenmy@math.uio.no>"
__date__ = "2011-05-14"
__copyright__ = "Copyright (C) 2011 " + __author__
__license__  = "GNU GPL version 3 or any later version?"
"""

    K-Epsilon/Reynolds-stress turbulence models

"""
from TurbSolver import *
from Eikonal import Eikonal
from cbc.rans.common.Wall import QWall #Need to make new wall-
    function to match system

class ER(TurbSolver):
    """
    Base class for ER turbulence models
    NOTE:  Rij and Fij should be stored as variables, not DQs,
        unlike in normal model
    """

    def __init__(self, system_composition, problem, parameters):
        # A segregated system of two coupled systems:
        parameters['space']['Rij'] = TensorFunctionSpace
        parameters['space']['Fij'] = TensorFunctionSpace
        self.dim = problem.NS_problem.mesh.geometry().dim()
        # When symmetric tensors is possible:
        #parameters['symmetry']['Rij'] = dict(((i,j), (j,i))
        #    for i in range(self.dim) for j in range(self.dim)
            if i > j )
        #parameters['symmetry']['Fij'] = dict(((i,j), (j,i))
        #    for i in range(self.dim) for j in range(self.dim)
```

```
        if i > j )
    TurbSolver.__init__(self,
                system_composition=system_composition,
                problem=problem,
                parameters=parameters)

def define(self):
    """define derived quantities for ER model."""
    V, NS = self.V['dq'], self.Turb_problem.NS_solver #
        Short forms
    DQ, DQ_NoBC = DerivedQuantity, DerivedQuantity_NoBC
    NS.V['SS'] = TensorFunctionSpace(self.Turb_problem.
        NS_problem.mesh,
                                    self.prm['family']['Rij
                                        '], self.prm['degree
                                        ']['Rij'])

    NS.schemes['derived quantities'] = [
        DQ_NoBC(NS, 'Sij_', NS.S, "epsilon(u_)", dict(u_=NS.
            u_), bounded=False, apply='project'),
        DQ_NoBC(NS, 'Wij_', NS.V['SS'], "0.5*(grad(u_) -
            grad(u_).T)", dict(u_=NS.u_),
                bounded=False)]
    self.Sij_ = NS.Sij_
    self.Wij_ = NS.Wij_
    self.i, self.j, self.m, self.l = i, j, Index(2), l
    self.dim = V.cell().d
    self.Aij = self.Rij*(0.5/self.k_) - 1./3.*self.dij
    ns = vars(self)
    # A33 = - (A11 + A33) = - tr(Aij), AijAji = inner(Aij,
        Aij) + A33**2 = inner(Aij, Aij) + trace(Aij)**2
    self.schemes['derived quantities'] = [
        DQ_NoBC(self, 'T_' , V, "max_(k_*(1./e_), 6.*sqrt(nu
            *(1./e_)))", ns),
        DQ_NoBC(self, 'L_' , V, "CL*max_(Ceta*(nu**3/e_)
            **(0.25), k_**(1.5)*(1./e_))", ns),
        DQ_NoBC(self, 'Aij_', NS.S, "Rij_*(0.5/k_) - 1./3.*
            dij", ns, bounded=False),
        DQ_NoBC(self, 'Pij_', self.V['Rij'], "- dot(Rij_,
            grad(u_).T) - dot(grad(u_), Rij_.T)", ns,
                bounded=False),
        #DQ_NoBC(self, 'Ce1_', V, "1.3 + 0.25/(1. + (0.15*y/
            L_)**2)**4", ns, bounded=True),
        DQ_NoBC(self, 'Ce1_', V, "1.4*(1. + Ced*sqrt(k_/max_
            (1.e-10, inner(Rij_, outer(ni, ni)))))", ns,
            bounded=True),
        DQ (self, 'nut_', V, 'Cmu*(inner(Rij_, outer(ni, ni)
            ))*T_', ns)
    ]
    if self.Turb_problem.prm['Model'] == 'LRR-IP':
        self.schemes['derived quantities'] += [
            DQ_NoBC(self, 'PHIij_', self.V['Rij'], "-CR*(1./
                T_)*2.*Aij*k_ \
                                    - C2*(Pij_ - 1./3.*tr(Pij_)*
```

```
                                        dij)", ns, bounded=False)
                                        ]

    elif self.Turb_problem.prm['Model'] == 'SSG':
        self.schemes['derived quantities'] += [
            DQ_NoBC(self, 'PHIij_', self.V['Rij'], "-( Cp1*
                e_ + Cp1s*0.5*tr(Pij_) )*Aij \
                                + Cp2*e_*(dot(Aij_, Aij_) -
                                    1./3*(inner(Aij, Aij_)+
                                    tr(Aij_)**2)*dij ) \
                                + ( Cp3 - Cp3s*sqrt(inner(
                                    Aij_, Aij_) + tr(Aij_)
                                    **2) )*k_*Sij_ \
                                + Cp4*k_*(dot(Aij_, Sij_) +
                                    dot(Sij_,Aij_) - 2./3*
                                    inner(Aij_, Sij_)*dij) \
                                + Cp5*k_*(dot(Wij_, Aij_) -
                                    dot(Aij_, Wij_) )", ns,
                                    bounded=False)]

    # insert Rij and "anti-stabilization"-term into f,
        similar to setting nu = nu + nut
    #NS.f = self.Turb_problem.NS_problem.body_force() - div(
        self.Rij_ - 2.*self.nut_*self.Sij_)
    NS.correction = self.Rij_ - 2.*self.nut_*self.Sij_ # For
         testing using integration by parts. Used with
        Steady_Coupled_5

    TurbSolver.define(self)

def model_parameters(self):
    """Parameters for the ER model."""
    model = self.Turb_problem.prm['Model']
    info('Setting parameters for %s ER model ' %(model))
    for dq in ['T_', 'L_', 'nut_']:
        # Specify projection as default
        # (remaining DQs are use_formula by default)
        self.prm['apply'][dq] = self.prm['apply'].get(dq, '
            project')

    self.model_prm = dict(
        Cmu_nut = Constant(0.09),
        Ce1 = Constant(1.44),
        Ced = Constant(0.045),
        Ce2 = Constant(1.9),
        sigma_e = Constant(1.3),
        sigma_k = Constant(1.0),
        Cp1  = Constant(3.4),
        Cp1s = Constant(1.8),
        Cp2  = Constant(4.2),
        Cp3  = Constant(0.8),
        Cp3s = Constant(1.30),
        Cp4  = Constant(1.25),
        Cp5 = Constant(0.4),
```

```
        Ceta = Constant(80.0),
        CL = Constant(0.25),
        Cmu = Constant(0.22),
        e_d = Constant(0.5),
        CR = Constant(1.8),
        C2 = Constant(3./5.)
    )
    self.dij = Identity(self.V['dq'].cell().d)
    self.__dict__.update(self.model_prm)

def create_BCs(self, bcs):
    # Compute distance to nearest wall
    self.distance = Eikonal(self.V['dq'], self.boundaries)
    self.y = self.distance.y
    DerivedQuantity_NoBC(self, 'ni', VectorFunctionSpace(
        self.Turb_problem.NS_problem.mesh,
                             self.prm['family']['dq'], self.prm
                                 ['degree']['dq']),
                             "grad(y)/sqrt(inner(grad(y), grad(y
                                 )))",
                             dict(y=self.y), bounded=False,
                                 apply='project')
    self.ti = Function(VectorFunctionSpace(self.Turb_problem
        .NS_problem.mesh,
                             self.prm['family']['dq'], self.prm
                                 ['degree']['dq']))
    N = self.ti.vector().size()/2
    self.ti.vector()[:N] = self.ni.vector()[N:]
    self.ti.vector()[N:] = -self.ni.vector()[:N]

    self.attach_boundary_functions(bcs)
    bcu = {}
    for name in self.system_names:
        bcu[name] = []

    for bc in bcs:
        for name in self.system_names:
            V = self.V[name]
            if bc.type() in ('VelocityInlet', 'Wall'):
                if hasattr(bc, 'func'):
                    if isinstance(bc.func, dict):
                        add_BC(bcu[name], V, bc, bc.func[
                            name])
                    else:
                        add_BC(bcu[name], V, bc, bc.func)
                else:
                    if bc.type() == 'Wall': # Default is
                        zero on walls
                        if isinstance(V, FunctionSpace):
                            func = Constant(1e-12)
                        elif isinstance(V, (
                            MixedFunctionSpace,
                                              VectorFunctionSpace
                                                  )):
```

```
                              n = 0
                              for i in xrange(0, (V.
                                  num_sub_spaces() or 1)):
                                      n = n + (V.sub(i).
                                          num_sub_spaces() or
                                          1)
                              """
                              func = Constant((1e-12, )*V.cell
                                  ().d)
                              if V.num_sub_spaces() > 2:
                                  #func = Constant((1e-12, )
                                      *8)
                                  #func = Constant((1e-12, )*V
                                      .num_sub_spaces()*V.sub
                                      (0).num_sub_spaces())
                              """
                              func = Constant((1e-12, )*n)

                          elif isinstance(V,
                              TensorFunctionSpace):
                              func = Expression((('1.e-12', )*
                                  V.cell().d, )*
                                                  V.cell().d)
                          else:
                              raise NotImplementedError
                          add_BC(bcu[name], V, bc, func)
                      elif bc.type() == 'VelocityInlet':
                          raise TypeError('expected func for
                              VelocityInlet')
              elif bc.type() in ('ConstantPressure', 'Outlet')
                  :
                  # This bc could be weakly enforced
                  bcu[name].append(bc)
              elif bc.type() == 'Periodic':
                  add_BC(bcu[name], V, bc, None)
              else:
                  info("No assigned boundary condition for %s
                      -- skipping..."
                       %(bc.__class__.__name__))
      return bcu
```

B.2 ER-subclasses

These are the subclasses of `ER` which holds the solver scheme and linearizations.

```
__author__ = "Jorgen Myre <jorgenmy@math.uio.no>"
__date__ = "2011-02-22"
__copyright__ = "Copyright (C) 2011 " + __author__
__license__  = "GNU GPL version 3 or any later version?"
"""
```

```
    ER turbulence model
    The two systems (k and epsilon) and (Rij and Fij) are
        individually solved

"""
from ER import *
from cbc.rans.common.Wall import QWall

class ER_FullyCoupled(ER):

    def __init__(self, problem, parameters):

        ER.__init__(self,
                    system_composition=[['k', 'e', 'Rij', 'Fij
                        ']],
                    problem=problem,
                    parameters=parameters)

    def define(self):
        ER.define(self)
        self.schemes['keRijFij']  =  eval(self.prm['
            time_integration'] + '_keRijFij_' +
                                    str(self.prm['scheme']['
                                        keRijFij']))(self,
                                        self.system_composition
                                            [0])

    def create_BCs(self, bcs):
        # Set regular boundary conditions
        bcu = ER.create_BCs(self, bcs)
        # Set wall functions
        for bc in bcs:
            if bc.type() == 'Wall':
                bcu['keRijFij'].append(QWall['Fij_3'](bc, self.y
                    , self.nu(0), self.keRijFij_, self.ni))
                bcu['keRijFij'][-1].type = bc.type
        return bcu

    # Updates stored variables before solving next part of the
        system
    def solve_inner(self, max_iter=1, max_err=1e-7, update=
        lambda: None,
                   logging=True):
        total_error = ""
        for name in self.system_names:
            err, j = solve_nonlinear([self.schemes[name]],
                                     max_iter=max_iter, max_err=
                                         max_err,
                                     update=update, logging=logging)
            self.solve_derived_quantities()
            total_error += err
        self.total_number_iters += j
        return total_error
```

```
class KERIJFIJBase(TurbModel):
    def update(self):
        N = self.V.sub(0).dim()
        dim = 2
        xa = self.x.array()
        """Make k and e positive"""
        xa[0:2*N] = maximum(1.e-12, xa[0:2*N])
        """uu, vv and ww are >= 0. Off diagonals are not."""
        start = 2*N
        for i in range(dim):
            stop = start + N
            xa[start:stop] = maximum(1.e-12, xa[start:stop])
            start = stop + N*dim # unsymmetric

        xa[3*N:4*N] = xa[4*N:5*N]
        xa[7*N:8*N] = xa[8*N:9*N]

        self.x.set_local(xa)

class Steady_keRijFij_1(KERIJFIJBase):
    def form(self, k, e, v_k, v_e, k_, e_, Pij_, nu, u_, Ce1,
        Ce1_, Ce2,
                  Cmu, e_d, sigma_e, nut_,Rij, Rij_, v_Rij,
                  Fij, Fij_, v_Fij, Aij_, Aij, PHIij_, T_, L_,
                      **kwargs):
        Fk = nu*inner( grad(k) , grad(v_k) )*dx \
             + inner( dot( u_, grad(k) ) , v_k )*dx \
             - inner( 0.5*tr(Pij_) , v_k)*dx \
             + (e*e_d + k*(1./k_)*e_*(1. - e_d))*v_k*dx \
             + inner( Cmu*T_*dot(Rij_,grad(k)) , grad(v_k) )*dx
             #+ nut_*inner(grad(v_k), grad(k))*dx

        Fe = nu*inner( grad(e) , grad(v_e) )*dx \
             + inner( dot( u_, grad(e) ) , v_e )*dx \
             - inner( 0.5*Ce1_*(1./T_)*tr(Pij_) , v_e)*dx \
             + Ce2*(1./T_)*e*v_e*dx \
             + inner( Cmu*T_*(1./sigma_e)*dot(Rij_,grad(e)) ,
                 grad(v_e) )*dx
             #+ nut_*(1./sigma_e)*inner(grad(v_e), grad(e))*dx

        Fr = nu*inner(grad(Rij), grad(v_Rij))*dx \
             + inner( dot(grad(Rij), u_) , v_Rij )*dx \
             - inner( k_*Fij , v_Rij )*dx \
             - inner( Pij_ , v_Rij )*dx \
             + inner( Rij*e_*(1./k_) , v_Rij)*dx \
             + inner( Cmu*T_*dot(grad(Rij), Rij_) , grad(v_Rij)
                )*dx
             #+ nut_*inner(grad(Rij) , grad(v_Rij) )*dx

        Ff = inner( grad(Fij), grad(L_**2*v_Fij) )*dx \
             + inner( Fij , v_Fij )*dx \
             - (1./k_)*inner( PHIij_ , v_Fij )*dx \
             - (2./T_)*inner( Aij_ , v_Fij )*dx
```

```
        return Fk + Fe + Fr +Ff
```

```
__author__ = "Jorgen Myre <jorgenmy@math.uio.no>"
__date__ = "2011-02-22"
__copyright__ = "Copyright (C) 2011 " + __author__
__license__  = "GNU GPL version 3 or any later version?"
"""

    ER turbulence model
    The two systems (k and epsilon) and (Rij and Fij) are
        individually solved

"""
from ER import *
from cbc.rans.common.Wall import QWall

class ER_2Coupled(ER):

    def __init__(self, problem, parameters):

        ER.__init__(self,
                    system_composition=[['k', 'e'], ['Rij', '
                        Fij']],
                    problem=problem,
                    parameters=parameters)

    def define(self):
        ER.define(self)
        self.schemes['ke']  =  eval(self.prm['time_integration']
             + '_ke_' +
                                str(self.prm['scheme']['ke
                                    ']))(self,
                                    self.system_composition
                                        [0])
        self.schemes['RijFij'] =  eval(self.prm['
            time_integration'] + '_RijFij_' +
                                  str(self.prm['scheme']['
                                      RijFij']))(self,
                                  self.system_composition
                                      [1])

    def create_BCs(self, bcs):
        # Set regular boundary conditions
        bcu = ER.create_BCs(self, bcs)
        # Set wall functions
        for bc in bcs:
            if bc.type() == 'Wall':
                bcu['ke'].append(QWall['ke'](bc, self.y, self.nu
                    (0)))
                bcu['ke'][-1].type = bc.type
                bcu['RijFij'].append(QWall['Fij_2'](bc, self.y,
                    self.nu(0), self.ke_, self.ni))
                #bcu['RijFij'].append(QWall['Fij_2'](bc, self.y,
                     self.nu(0), self.ke_))
```

```
                bcu['RijFij'][-1].type = bc.type
        return bcu

    # Updates stored variables before solving next part of the
        system
    def solve_inner(self, max_iter=1, max_err=1e-7, update=
        lambda: None,
                    logging=True):
        total_error = ""
        for name in self.system_names:
            err, j = solve_nonlinear([self.schemes[name]],
                                     max_iter=max_iter, max_err=
                                         max_err,
                                     update=update, logging=logging)
            self.solve_derived_quantities()
            total_error += err
        self.total_number_iters += j
        return total_error

class KEBase(TurbModel):
    def update(self):
        """This makes k=1e-10 on walls."""
        bound(self.x, minf=1e-10)

class RIJFIJBase(TurbModel):
    def update(self):
        """uu, vv and ww are >= 0. Off diagonals are not."""
        N = self.V.sub(0).sub(0).dim()
        dim = 2
        xa = self.x.array()
        start = 0
        for i in range(dim):
            stop = start + N
            xa[start:stop] = maximum(1.e-12, xa[start:stop])
            start = stop + N*dim # unsymmetric

        xa[N:2*N] = xa[2*N:3*N]
        xa[5*N:6*N] = xa[6*N:7*N]

        self.x.set_local(xa)

class Steady_ke_1(KEBase):
    def form(self, k, e, v_k, v_e, k_, e_, Rij_, Pij_, nu, u_,
        Ce1, Ce1_, T_, Ce2,
                  Cmu, e_d, sigma_e, nut_, **kwargs):
        Fk = nu*inner( grad(k) , grad(v_k) )*dx \
             + inner( dot( u_, grad(k) ) , v_k )*dx \
             - inner( 0.5*tr(Pij_) , v_k)*dx \
             + (e*e_d + k*(1./k_)*e_*(1. - e_d))*v_k*dx \
             + inner( Cmu*T_*dot(Rij_,grad(k)) , grad(v_k) )*dx
             #+ nut_*inner(grad(v_k), grad(k))*dx

        Fe = nu*inner( grad(e) , grad(v_e) )*dx \
             + inner( dot( u_, grad(e) ) , v_e )*dx \
```

```
            - inner( 0.5*Ce1_*(1./T_)*tr(Pij_) , v_e)*dx \
            + Ce2*(1./T_)*e*v_e*dx \
            + inner( Cmu*T_*(1./sigma_e)*dot(Rij_,grad(e)) ,
                grad(v_e) )*dx
            #+ nut_*(1./sigma_e)*inner(grad(v_e), grad(e))*dx

        return Fk + Fe

class Steady_RijFij_1(RIJFIJBase):
    def form(self, Rij, Rij_, v_Rij, k_, e_, Pij_, nu, u_, nut_,
             Fij, Fij_, v_Fij, Aij_, Aij, PHIij_, Cmu, T_, L_,
                 **kwargs):
        Fr = nu*inner(grad(Rij), grad(v_Rij))*dx \
            + inner( dot(grad(Rij), u_) , v_Rij )*dx \
            - inner( k_*Fij , v_Rij )*dx \
            - inner( Pij_ , v_Rij )*dx \
            + inner( Rij*e_*(1./k_) , v_Rij)*dx \
            + inner( Cmu*T_*dot(grad(Rij), Rij_) , grad(v_Rij)
                )*dx
            #+ nut_*inner(grad(Rij) , grad(v_Rij) )*dx

        Ff = inner( grad(Fij), grad(L_**2*v_Fij) )*dx \
            + inner( Fij , v_Fij )*dx \
            - (1./k_)*inner( PHIij_ , v_Fij )*dx \
            - (2./T_)*inner( Aij_ , v_Fij )*dx

        return Fr + Ff
```

```
__author__ = "Jorgen Myre <jorgenmy@math.uio.no>"
__date__ = "2011-05-14"
__copyright__ = "Copyright (C) 2011 " + __author__
__license__  = "GNU GPL version 3 or any later version?"
"""

    ER turbulence model
    The three systems (k and epsilon), (Rij) and (Fij) are
        individually solved

"""
from ER import *
from cbc.rans.common.Wall import QWall

class ER_3Coupled(ER):

    def __init__(self, problem, parameters):
        # A segregated system of three coupled systems:
        parameters['space']['Rij'] = TensorFunctionSpace
        parameters['space']['Fij'] = TensorFunctionSpace
        self.dim = problem.NS_problem.mesh.geometry().dim()
        parameters['symmetry']['Rij'] = dict(((i,j), (j,i))
            for i in range(self.dim) for j in range(self.dim) if
                i > j )
        parameters['symmetry']['Fij'] = dict(((i,j), (j,i))
```

```
            for i in range(self.dim) for j in range(self.dim) if
                i > j )

        ER.__init__(self,
                    system_composition=[['k', 'e'], ['Rij'], ['
                        Fij']],
                    problem=problem,
                    parameters=parameters)

    def define(self):
        ER.define(self)
        self.schemes['ke']  =  eval(self.prm['time_integration']
            + '_ke_' +
                                    str(self.prm['scheme']['ke'])
                                        )(self,
                                        self.system_composition
                                            [0])
        self.schemes['Rij'] =  eval(self.prm['time_integration']
            + '_Rij_' +
                                    str(self.prm['scheme']['Rij
                                        ']))(self,
                                        self.system_composition
                                            [1])
        self.schemes['Fij'] =  eval(self.prm['time_integration']
            + '_Fij_' +
                                    str(self.prm['scheme']['Fij
                                        ']))(self,
                                        self.system_composition
                                            [2])


    def create_BCs(self, bcs):
        # Set regular boundary conditions
        bcu = ER.create_BCs(self, bcs)
        # Set wall functions
        for bc in bcs:
            if bc.type() == 'Wall':
                bcu['ke'].append(QWall['ke'](bc, self.y, self.nu
                    (0)))
                bcu['ke'][-1].type = bc.type
                bcu['Fij'].append(QWall['Fij'](bc, self.y, self.
                    nu(0), self.ke_, self.Rij_))
                bcu['Fij'][-1].type = bc.type
        return bcu

    # Updates stored variables before solving next part of the
        system
    def solve_inner(self, max_iter=1, max_err=1e-7, update=
        lambda: None,
                    logging=True):
        total_error = ""
        for name in self.system_names:
            err, j = solve_nonlinear([self.schemes[name]],
```

```
                                    max_iter=max_iter, max_err=
                                        max_err,
                                    update=update, logging=logging)
            self.solve_derived_quantities()
            total_error += err
        self.total_number_iters += j
        return total_error

class KEBase(TurbModel):
    def update(self):
        """This makes k=1e-10 on walls."""
        bound(self.x, minf=1e-10)

class RIJBase(TurbModel):
    def update(self):
        """uu, vv and ww are >= 0. Off diagonals are not."""
        N = self.V.sub(0).dim()
        dim = self.V.cell().d
        xa = self.x.array()
        start = 0
        for i in range(dim):
            stop = start + N
            xa[start:stop] = maximum(1.e-12, xa[start:stop])
            start = stop + N*(dim - 1 + i) # Symmetric
        self.x.set_local(xa)

class FIJBase(TurbModel):
    def update(self):
        """Fij should not be bounded anywhere, but kept here as
            placeholder."""
        pass


class Steady_ke_1(KEBase):
    def form(self, k, e, v_k, v_e, k_, e_, Rij_, Pij_, nu, u_,
        Ce1, T_, Ce2,
                Cmu, e_d, **kwargs):
        Fk = nu*inner( grad(k) , grad(v_k) )*dx \
             + inner( dot( u_, grad(k) ) , v_k )*dx \
             - inner( tr(Pij_) , v_k)*dx \
             + (k_*e*e_d + k*e_*(1. - e_d))*(1./k_)*v_k*dx \
             + inner( Cmu*T_*dot(Rij_,grad(k)) , grad(v_k) )*dx

        Fe = nu*inner( grad(e) , grad(v_e) )*dx \
             + inner( dot( u_, grad(e) ) , v_e )*dx \
             - inner( Ce1*(1./T_)*tr(Pij_) , v_e)*dx \
             + Ce2*(1./T_)*e*v_e*dx \
             + inner( Cmu*T_*dot(Rij_,grad(e)) , grad(v_e) )*dx

        return Fk + Fe

class Steady_Rij_1(RIJBase):
    def form(self, Rij, Rij_, v_Rij, k_, e_, Fij_, Pij_, nu, u_,
         T_, Cmu, **kwargs):
```

```
        i, j, l, m = indices(4)
        Fr = nu*inner(grad(Rij),grad(v_Rij))*dx \
             + inner( dot(grad(Rij),u_) , v_Rij )*dx \
             - inner( k_*Fij_ , v_Rij )*dx \
             - inner( Pij_ , v_Rij )*dx \
             + inner( Rij*e_*(1./k_) , v_Rij)*dx \
             + inner( Cmu*T_*dot(grad(Rij),Rij_) , grad(v_Rij) )
                 *dx

        return Fr

class Steady_Fij_1(FIJBase):

    def form(self, Fij, v_Fij, k_, Aij_, PHIij_, T_, L_, Cmu, **
        kwargs):
        Ff = (L_**2)*inner( grad(Fij), grad(v_Fij) )*dx \
             + inner( Fij , v_Fij )*dx \
             - (1./k_)*inner( PHIij_ , v_Fij )*dx \
             - 2.*(1./T_)*inner( Aij_ , v_Fij )*dx

        return Ff
```

B.3 Implementation of boundary conditions

This is the classes for boundary conditions for the different systems, which are normally found in the `Wall.py` file in the CBC.RANS package.

```
class FIJWall_1(Wallfunction):
    """Wall-BC for uncoupled Fij"""
    """Set F11 = -0.5*F22 implicitly, F22 = -20*(v**2/y**4)*vv/e
        and F12 = -8*(v**2/y**4)*uv/e explicitly."""
    def __init__(self, bc, y, nu, ke, Rij):
        Wallfunction.__init__(self, y.function_space(), bc)
        self.y = y.vector()
        self.N = len(self.y)
        self.nu = nu
        self.ke = ke.vector()
        self.Rij = Rij.vector()
        if not (len(self.ke) == 2*self.N):
            info('Warning! Only works when functionspace of
                Eikonal is equal to epsilon')

    def apply(self, *args):
        """Apply boundary condition to tensors."""
        aro = array(list(self.vertices_on_boundary),'I')
        ari = array(list(self.vertices_inside_boundary),'I')
        for var in args:
            if(isinstance(var, Matrix)):
                var.ident(aro)
                var.ident(ari)
                var.ident(aro + self.N)
                var.ident(ari + self.N)
```

```
            var.ident(aro + 2*self.N)
            var.ident(ari + 2*self.N)
            # Set F11 = -0.5*F22 approaching boundaries
            for i in self.vertices_inside_boundary:
                col = array([i + 2*self.N], 'I')
                val = array([0.5])
                var.setrow(i, col, val)
                var.apply('')
            #var[i, i + 2*self.N] = 0.5
            for j in self.vertices_on_boundary:
                i = self.bnd_to_in[j]
                col = array([i + 2*self.N], 'I')
                val = array([0.5])
                var.setrow(j, col, val)
                var.apply('')
                #var[j, i + 2*self.N] = 0.5

        if(isinstance(var, Vector)):
            var[aro] = 0.
            var[ari] = 0.
            #Set Fij = - (8 or 10)*nu**2/y**4*Rij
    #for v2f sets eps = max(1e-3,e), maybe do that here too?
    for j in self.vertices_on_boundary:
        i = self.bnd_to_in[j]
        var[j + self.N] = -8.*(self.nu**2/self.y[i]**4)*self
            .Rij[i+self.N]*(1./self.ke[i +self.N])
        var[j + 2*self.N] = -20.*(self.nu**2/self.y[i]**4)*
            self.Rij[i+2*self.N]*(1./self.ke[i +self.N])
        for i in self.vertices_inside_boundary:
            var[i + self.N] = -8.*(self.nu**2/self.y[i]**4)*
                self.Rij[i+self.N]*(1./self.ke[i +self.N])
            var[i + 2*self.N] = -20.*(self.nu**2/self.y[i
                ]**4)*self.Rij[i+2*self.N]*(1./self.ke[i +
                self.N])

class FIJWall_2_UNSYMMETRIC2(Wallfunction):
    """Wall-BC for Rij and Fij coupled"""
    """Set F11 = -0.5*F22 implicitly, F22 = -20*(v**2/y**4)*vv/e
        and F12 = -8*(v**2/y**4)*uv/e explicitly."""
    def __init__(self, bc, y, nu, ke, ni):
        Wallfunction.__init__(self, y.function_space(), bc)
        self.y = y.vector()
        self.N = len(self.y)
        self.nu = nu
        self.ke = ke.vector()
        self.ni = ni.vector()
        if not (len(self.ke) == 2*self.N):
            info('Warning! Only works when functionspace of
                Eikonal is equal to epsilon')

    def apply(self, *args):
        """Apply boundary condition to tensors."""
        aro = array(list(self.vertices_on_boundary),'I')
        ari = array(list(self.vertices_inside_boundary),'I')
```

```
N = self.N
nn = self.ni
for var in args:
    if(isinstance(var, Matrix)):
        # Just keep these because they set everything to
            zero except
        # the diagonal that is overloaded anyway
        var.ident(aro + 4*N)
        var.ident(ari + 4*N)
        var.ident(aro + 5*N)
        var.ident(ari + 5*N)
        var.ident(aro + 6*N)
        var.ident(ari + 6*N)
        var.ident(aro + 7*N)
        var.ident(ari + 7*N)

        for i in self.vertices_inside_boundary:
            colF = array([i + 4*N, i + 5*N, i + 6*N, i +
                7*N], 'I')
            colR = array([i, i + N, i + 2*N, i + 3*N], '
                I')
            n1 = nn[i]
            n2 = nn[i + N]
            t1 = n2
            t2 = -n1
            valn =  array([n1*n1, n1*n2, n2*n1, n2*n2])
                # nn[0,0], nn[0,1], nn[1,0], nn[1,1]
            valt =  array([t1*t1, t1*t2, t2*t1, t2*t2])
                # tt[0,0], tt[0,1], tt[1,0], tt[1,1]
            valnt = array([n1*t1, n2*t1, n1*t2, n2*t2])
                # nt[0,0], nt[0,1], nt[1,0], nt[1,1]
            valtn = array([t1*n1, t2*n1, t1*n2, t2*n2])
                # tn[0,0], tn[0,1], tn[1,0], tn[1,1]

            #print 'valn  ', i, valn
            #print 'valt  ', i, valt
            #print 'valnt ', i, valnt
            #print 'valtn ', i, valtn

            # Ftt = -0.5*Fnn
            var.setrow(i + 4*self.N, colF, valt + 0.5*
                valn)

            # Fnt = -8*(nu**2/y**4)*Rnt/e
            vv = 8.*(self.nu**2/self.y[i]**4)*(1./self.
                ke[i +self.N])
            var.setrow(i + 5*self.N, colR, vv*valnt)
            var.setrow(i + 5*self.N, colF, valnt)
            # Ftn = -8*(nu**2/y**4)*Rtn/e
            var.setrow(i + 6*self.N, colR, vv*valtn)
            var.setrow(i + 6*self.N, colF, valtn)

            # Fnn = -20*(nu**2/y**4)*Rnn/e
            vv = 20.*(self.nu**2/self.y[i]**4)*(1./self.
```

```
                ke[i +self.N])
            var.setrow(i + 7*self.N, colR, vv*valn)
            var.setrow(i + 7*self.N, colF, valn)
            var.apply('')

        for j in self.vertices_on_boundary:
            i = self.bnd_to_in[j]
            colF = array([j + 4*N, j + 5*N, j + 6*N, j +
                 7*N], 'I')
            colR = array([i, i + N, i + 2*N, i + 3*N], '
                I')
            n1 = nn[i]
            n2 = nn[i + N]
            t1 = -n2
            t2 = n1
            valn =  array([n1*n1, n1*n2, n2*n1, n2*n2])
                # nn[0,0], nn[0,1], nn[1,0], nn[1,1]
            valt =  array([t1*t1, t1*t2, t2*t1, t2*t2])
                # tt[0,0], tt[0,1], tt[1,0], tt[1,1]
            valnt = array([n1*t1, n2*t1, n1*t2, n2*t2])
                # nt[0,0], nt[0,1], nt[1,0], nt[1,1]
            valtn = array([t1*n1, t2*n1, t1*n2, t2*n2])
                # tn[0,0], tn[0,1], tn[1,0], tn[1,1]

            # Ftt = -0.5*Fnn
            var.setrow(j + 4*self.N, colF, valt + 0.5*
                valn)

            # F12 = -8*(nu**2/y**4)*Rnt/e
            vv = 8.*(self.nu**2/self.y[i]**4)*(1./self.
                ke[i + N])
            var.setrow(j + 5*N, colR, vv*valnt)
            var.setrow(j + 5*N, colF, valnt)
            var.setrow(j + 6*N, colR, vv*valtn)
            var.setrow(j + 6*N, colF, valtn)

            # Fnn = -20*(nu**2/y**4)*Rnn/e
            vv = 20.*(self.nu**2/self.y[i]**4)*(1./self.
                ke[i +N])
            var.setrow(j + 7*N, colR, vv*valn)
            var.setrow(j + 7*N, colF, valn)
            var.apply('')

    if(isinstance(var, Vector)):
        var[aro + 4*N] = 0.
        var[ari + 4*N] = 0.
        var[aro + 5*N] = 0.
        var[ari + 5*N] = 0.
        var[aro + 6*N] = 0.
        var[ari + 6*N] = 0.
        var[aro + 7*N] = 0.
        var[ari + 7*N] = 0.

class FIJWall_3_UNSYMMETRIC(Wallfunction):
```

```
"""Wall-BC for fully coupled ER system"""
"""Wall-BC for e, Rij and Fij coupled"""
"""Set F11 = -0.5*F22 implicitly, F22 = -20*(v**2/y**4)*vv/e
    and F12 = -8*(v**2/y**4)*uv/e explicitly."""
def __init__(self, bc, y, nu, keRijFij, ni):
    Wallfunction.__init__(self, y.function_space(), bc)
    self.y = y.vector()
    self.N = len(self.y)
    self.nu = nu
    self.ke = keRijFij.vector()
    self.ni = ni.vector()
    if not (len(self.ke) == 2*self.N):
        info('Warning! Only works when functionspace of
            Eikonal is equal to epsilon')

def apply(self, *args):
    """Apply boundary condition to tensors."""
    aro = array(list(self.vertices_on_boundary),'I')
    ari = array(list(self.vertices_inside_boundary),'I')
    N = self.N
    nn = self.ni
    for var in args:
        if(isinstance(var, Matrix)):
            # Just keep these because they set everything to
                zero except
            # the diagonal that is overloaded anyway
            var.ident(aro + N)
            var.ident(ari + N)
            var.ident(aro + 6*N)
            var.ident(ari + 6*N)
            var.ident(aro + 7*N)
            var.ident(ari + 7*N)
            var.ident(aro + 8*N)
            var.ident(ari + 8*N)
            var.ident(aro + 9*N)
            var.ident(ari + 9*N)

            for i in self.vertices_inside_boundary:

                colE = array([i], 'I')
                valE = array([-2.*self.nu/self.y[i]**2])
                var.setrow(i + self.N, colE, valE)

                colF = array([i + 6*N, i + 7*N, i + 8*N, i +
                    9*N], 'I')
                colR = array([i + 2*N, i + 3*N, i + 4*N, i +
                    5*N], 'I')
                n1 = nn[i]
                n2 = nn[i + N]
                t1 = n2
                t2 = -n1
                valn =  array([n1*n1, n1*n2, n2*n1, n2*n2])
                    # nn[0,0], nn[0,1], nn[1,0], nn[1,1]
                valt =  array([t1*t1, t1*t2, t2*t1, t2*t2])
```

```
        # tt[0,0], tt[0,1], tt[1,0], tt[1,1]
    valnt = array([n1*t1, n2*t1, n1*t2, n2*t2])
        # nt[0,0], nt[0,1], nt[1,0], nt[1,1]
    valtn = array([t1*n1, t2*n1, t1*n2, t2*n2])
        # tn[0,0], tn[0,1], tn[1,0], tn[1,1]

    #print 'valn  ', i, valn
    #print 'valt  ', i, valt
    #print 'valnt ', i, valnt
    #print 'valtn ', i, valtn

    # Ftt = -0.5*Fnn
    var.setrow(i + 6*self.N, colF, valt + 0.5*
        valn)

    # Fnt = -8*(nu**2/y**4)*Rnt/e
    vv = 8.*(self.nu**2/self.y[i]**4)*(1./self.
        ke[i +self.N])
    var.setrow(i + 7*self.N, colR, vv*valnt)
    var.setrow(i + 7*self.N, colF, valnt)
    # Ftn = -8*(nu**2/y**4)*Rtn/e
    var.setrow(i + 8*self.N, colR, vv*valtn)
    var.setrow(i + 8*self.N, colF, valtn)

    # Fnn = -20*(nu**2/y**4)*Rnn/e
    vv = 20.*(self.nu**2/self.y[i]**4)*(1./self.
        ke[i +self.N])
    var.setrow(i + 9*self.N, colR, vv*valn)
    var.setrow(i + 9*self.N, colF, valn)
    var.apply('')

for j in self.vertices_on_boundary:
    i = self.bnd_to_in[j]

    colE = array([i], 'I')
    valE = array([-2.*self.nu/self.y[i]**2])
    var.setrow(j + self.N, colE, valE)

    colF = array([j + 6*N, j + 7*N, j + 8*N, j +
         9*N], 'I')
    colR = array([i + 2*N, i + 3*N, i + 4*N, i +
         5*N], 'I')
    n1 = nn[i]
    n2 = nn[i + N]
    t1 = -n2
    t2 = n1
    valn =  array([n1*n1, n1*n2, n2*n1, n2*n2])
        # nn[0,0], nn[0,1], nn[1,0], nn[1,1]
    valt =  array([t1*t1, t1*t2, t2*t1, t2*t2])
        # tt[0,0], tt[0,1], tt[1,0], tt[1,1]
    valnt = array([n1*t1, n2*t1, n1*t2, n2*t2])
        # nt[0,0], nt[0,1], nt[1,0], nt[1,1]
    valtn = array([t1*n1, t2*n1, t1*n2, t2*n2])
        # tn[0,0], tn[0,1], tn[1,0], tn[1,1]
```

```
            # Ftt = -0.5*Fnn
            var.setrow(j + 6*self.N, colF, valt + 0.5*
                valn)

            # F12 = -8*(nu**2/y**4)*Rnt/e
            vv = 8.*(self.nu**2/self.y[i]**4)*(1./self.
                ke[i + N])
            var.setrow(j + 7*N, colR, vv*valnt)
            var.setrow(j + 7*N, colF, valnt)
            var.setrow(j + 8*N, colR, vv*valtn)
            var.setrow(j + 8*N, colF, valtn)

            # Fnn = -20*(nu**2/y**4)*Rnn/e
            vv = 20.*(self.nu**2/self.y[i]**4)*(1./self.
                ke[i +N])
            var.setrow(j + 9*N, colR, vv*valn)
            var.setrow(j + 9*N, colF, valn)
            var.apply('')


    if(isinstance(var, Vector)):
        var[aro + self.N] = 0.
        var[ari + self.N] = 0.
        var[aro + 6*N] = 0.
        var[ari + 6*N] = 0.
        var[aro + 7*N] = 0.
        var[ari + 7*N] = 0.
        var[aro + 8*N] = 0.
        var[ari + 8*N] = 0.
        var[aro + 9*N] = 0.
        var[ari + 9*N] = 0.
```