NTNU – Trondheim
Norwegian University of
Science and Technology

Programming turbulence models in FORTRAN

Eirik Helno Herø

Master of Science in Mechanical Engineering
Submission date: June 2015
Supervisor: Reidar Kristoffersen, EPT

Norwegian University of Science and Technology
Department of Energy and Process Engineering

NTNU
Norwegian University
of Science and Technology

Department of Energy
and Process Engineering

EPT-M-2015-36

MASTER THESIS

for

Student Eirik Helno Herø

Spring 2015

Programming turbulence models in FORTRAN

Programmering av turbulensmodeller i FORTRAN

Background and objective

FORTRAN is a common programming language in the scientific community, and in computational fluid dynamics in particular. It is also common in the fluid dynamics department at NTNU and therefore a logical choice for this project. The purpose of this project is for the student to expand his knowledge of FORTRAN and explore in depth some of the theory presented through the courses at NTNU. This project expands on the program created in TEP4540, Project Work, Autumn 2014, by the same student.

The following tasks are to be considered:

- Implement and investigate different turbulence models like k-epsilon, k-omega and k-omega-SST.
- Investigate the gain of using multiple processors.
- If the time allows, any further expansion that covers more physical problems like free surface modelling would be interesting.

-- ” --

Within 14 days of receiving the written text on the master thesis, the candidate shall submit a research plan for his project to the department.

When the thesis is evaluated, emphasis is put on processing of the results, and that they are presented in tabular and/or graphic form in a clear manner, and that they are analyzed carefully.

The thesis should be formulated as a research report with summary both in English and Norwegian, conclusion, literature references, table of contents etc. During the preparation of the text, the candidate should make an effort to produce a well-structured and easily readable report. In order to ease the evaluation of the thesis, it is important that the cross-references are correct. In the making of the report, strong emphasis should be placed on both a thorough discussion of the results and an orderly presentation.

The candidate is requested to initiate and keep close contact with his/her academic supervisor(s) throughout the working period. The candidate must follow the rules and regulations of NTNU as well as passive directions given by the Department of Energy and Process Engineering.

Risk assessment of the candidate's work shall be carried out according to the department's procedures. The risk assessment must be documented and included as part of the final report. Events related to the candidate's work adversely affecting the health, safety or security, must be documented and included as part of the final report. If the documentation on risk assessment represents a large number of pages, the full version is to be submitted electronically to the supervisor and an excerpt is included in the report.

Pursuant to "Regulations concerning the supplementary provisions to the technology study program/Master of Science" at NTNU §20, the Department reserves the permission to utilize all the results and data for teaching and research purposes as well as in future publications.

The final report is to be submitted digitally in DAIM. An executive summary of the thesis including title, student's name, supervisor's name, year, department name, and NTNU's logo and name, shall be submitted to the department as a separate pdf file. Based on an agreement with the supervisor, the final report and other material and documents may be given to the supervisor in digital format.

☐ Work to be done in lab (Water power lab, Fluids engineering lab, Thermal engineering lab)
☐ Field work

Department of Energy and Process Engineering, 14. January 2015

Olav Bolland
Department Head

Reidar Kristoffersen
Academic Supervisor

Preface

This report is written at the Department of Energy and Process Engineering, NTNU, spring semester 2015 as my master thesis in Mechanical Engineering. It is designed to give me an insight into programming in Fortran and turbulence models.

My deepest thanks to Reidar Kristoffersen for the continued opportunities to work on the exiting field that is computational fluid dynamics. Your expertise and knowledge is always available and your motivation is a big inspiration.

Eirik Herø
Trondheim, June 2015

Abstract

The two-equation turbulence models Wilcox k-omega and Menter k-omega SST are programmed in FORTRAN and tested on the cases channel flow and backward-facing step. A short time step and a good initial field is required to obtain a solution. Results are adequate for most engineering purposes with a 15 % error in predicted reattachment length. Gain from parallel programming is only found on the elliptic equation solver.

Sammendrag

To-ligning turbulens modellene Wilcox k-omega og Menter k-omega SST er programmert i FORTRAN og testet på channel flow og backward-facing step. For å få en løsning er det nødvendig med et kort tidssteg og et godt initialfelt. Resultatene er tilfredsstillende for de fleste ingeniør bruksområdene med 15 % feilestimat på reattachment length. Tidsbesparing fra parallell programmering ble bare funnet på elliptisk ligning løseren.

Contents

Chapter 1

Introduction

With computational power and memory increasing, computational fluid dynamics, CFD, is increasingly used in engineering. This demand is supplied with software where turbulence models, especially two-equation models, are an already integrated part. All the user needs to do is check a box and the turbulence is simulated, but are the models really that simple? With turbulence being an advanced three dimensional transient phenomena, it appears a large amount of faith is placed in the models and the implementation into the software.

Working with CFD without commercial CFD software requires a large understanding of fluid dynamics and programming, an uncommon and challenging combination. Without the understanding of the physics the results are hard to interpret correctly, and the large amount of programming makes it a challenge for beginners. Although programming is integrated into most engineering educations, it is nowhere near the level needed to create a complete CFD program.

The goal of the thesis is to learn about two-equation turbulence modeling, as well as programming with FORTRAN. To accomplish this the 2D laminar program from the project work will be expanded to include turbulence models and tested on the backward-facing step case. The elliptic pressure equation solver BiCGSTAB is kept and the grid will be uniform for ease of programming. There will also be a 1D program solving channel flow to test the models on a simple problem and an investigation into possible multiprocessor programming to shorten simulation time. The Reynolds numbers covered are low to medium, allowing to compare with perfectly smooth walls from DNS results. The two-equation turbulence models covered in this paper is:

- Wilcox k-omega
- Menter k-omega SST
- k-epsilon

1.1 Hardware and Software

The simulations are run on a ASUS G75VW with Ubuntu 14.04.1 LTS, 16 GB of memory and a Intel Core i7-3630QM processor at 2.40 GHz. The programs are written in gedit Text Editor 3.10.4 and compiled by NAG Fortran Compiler 6.0. Double precision, the

NAG working precision, must be set on all real variables used by the NAG Fortran Library Mark 24 (64-bit) for Linux, FLL6A24D9L. For chapter 3 only quadruple precision is used.

Chapter 2

Background Theory

2.1 The Projection Method

The projection method, as presented by Kristoffersen[1], solves the Navier-Stokes Equation in three steps:

$$\frac{u_i^* - u_i^n}{\Delta t} + u_j^n \frac{\partial u_i^n}{\partial x_j} = -\beta \frac{1}{\rho} \frac{\partial p^n}{\partial x_i} + \frac{\partial}{\partial x_j}[(\nu + \nu_t^n)\frac{\partial u_i^n}{\partial x_j}] \tag{2.1}$$

$$\frac{\partial^2}{\partial x_j \partial x_j} \frac{p^{n+1} - \beta p^n}{\rho} = \frac{1}{\Delta t} \frac{\partial u_i^*}{\partial x_i} \tag{2.2}$$

$$\frac{u_i^{n+1} - u_i^*}{\Delta t} = -\frac{1}{\rho} \frac{\partial}{\partial x_i}(p^{n+1} - \beta p^n) \tag{2.3}$$

Here u^* is a tentative velocity and the constant β is a switch between zero and one. When β is set to one, more information from p^n is transfered to U^*, while for β set to zero, p^n is not used in the tentative flow-field. In this paper β set to zero, that way the old pressure field provides an excellent guess for the solution of equation 2.2.

2.2 Law of the Wall

The law of the wall describes the average velocity in the boundary layer of a turbulent flow. For y^+ values below 5, the viscous sublayer, the u^+ is equal to y^+. The log layer is named after the logarithmic relation between y^+ and u^+;

$$u^+ = \frac{1}{\kappa} \ln y^+ + C^+ \tag{2.4}$$

where, for a smooth wall, $C^+ \approx 5.0$ and $\kappa \approx 0.41$. At $5 < y^+ < 30$, the buffer layer, neither of the equations hold.

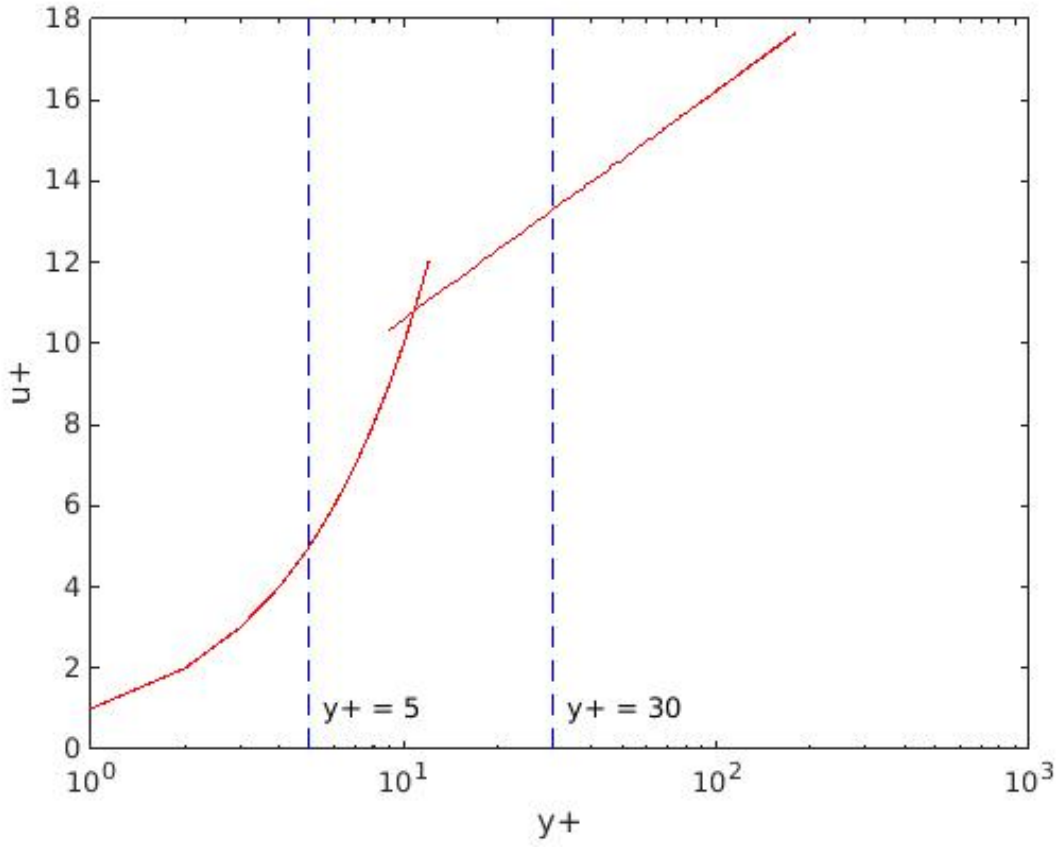

Figure 2.1: Law of the wall

2.3 The Eddy Viscosity Hypothesis

The eddy viscosity concept was introduced in 1887 by Boussinesq, and is widely used when modeling turbulence. It states that the turbulent Reynolds stresses are proportional to the gradients of the mean strain-rate tensor [2]:

$$\tau_{ij} = \mu_t(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij}) - \frac{2}{3}\rho k\delta_{ij} \tag{2.5}$$

This eddy viscosity is not a physical property, but varies with local flow conditions and geometry. Although the eddy viscosity hypothesis is not completely correct, it can provide an adequate estimate of turbulent flows[3]. In its simplest description the eddy viscosity can be characterized by a velocity q, which is based on k, and a length scale L, often based on the dissipation and production ratio of k;

$$\mu_t = C_\mu \rho q L \tag{2.6}$$

the modeling of q and L then decides the final expression of μ_t.

2.4 Two-Equation Turbulence Models

The two-equation turbulence models are derived from the long time averaged Navier-Stokes equation using the eddy viscosity theorem. The only change to the Navier-Stokes equation is the addition of the eddy viscosity to the molecular viscosity. Two other turbulent quantities are introduced, the mean turbulent kinetic energy, k, and the quantity describing the dissipation of the turbulent kinetic energy. The dissipation is often symbolized by ϵ and the specific dissipation by ω. The turbulent kinetic energy is defined as:

$$k = \frac{1}{2}\overline{u_j' u_j'} \tag{2.7}$$

Both k and ω, or ϵ, have their own transport equations and, although they change from model to model, contain terms for rate of change in a fluid element in mean motion, production, diffusion and dissipation. Other terms may be included to account for specific physical and geometrical mechanisms.

2.4.1 Wilcox K-Omega

The Wilcox k-omega model, introduced by Wilcox in the late 1980s, uses the specific dissipation and is one of the few models that are able to resolve the boundary layer without damping functions[4]. According to Menter[4], the Wilcox k-omega of 1988 is as accurate as any other two-equation model in predicting the mean flow and have better numerical stability. The model have been modified somewhat over the years, and the current standard is the 2006 version[5]. However, the 1988 model is valued for its simplicity:

$$\frac{D\rho k}{Dt} = \tau_{ij}\frac{\partial u_i}{\partial x_j} - \beta^*\rho\omega k + \frac{\partial}{\partial x_j}[(\mu + \sigma_k\mu_t)\frac{\partial k}{\partial x_j}] \tag{2.8}$$

$$\frac{D\rho\omega}{Dt} = \frac{\gamma\omega}{k}\tau_{ij}\frac{\partial u_i}{\partial x_j} - \beta\rho\omega^2 + \frac{\partial}{\partial x_j}[(\mu + \sigma_\omega\mu_t)\frac{\partial \omega}{\partial x_j}] \tag{2.9}$$

$$\mu_t = \frac{\rho k}{\omega} \tag{2.10}$$

The boundary conditions at the wall are $\mu_t|_{wall} = 0$ $k_{wall} = 0$ and $\omega_{wall} = \frac{60\nu}{\beta d^2}$, where d is the distance to the first cell center. Wilcox initially suggests a different boundary condition for ω, $\omega_{wall} \rightarrow \frac{6\nu_{wall}}{\beta d^2}$ as $d \rightarrow 0$, but Menter later claims the one used to be superior[4]. The constants in the model are:

$$\begin{array}{ccc} \sigma_k = 0.5 & \sigma_\omega = 0.5 & \beta^* = 0.09 \\ \gamma = \frac{5}{9} & \beta = \frac{3}{40} & \end{array}$$

2.4.2 Menter K-Omega SST

In a 1994 paper, reference [4], Menter presented one of the pillars in two-equation turbulence modeling by combining the original k-epsilon and Wilcox k-omega models. In broad strokes, the model uses Wilcox k-omega in the boundary layer and k-epsilon in the outer region and free shear flows. The Menter k-omega Shear-Stress Transport was empirically derived from the two others and uses several functions.

$$\frac{D\rho k}{Dt} = \tau_{ij}\frac{\partial u_i}{\partial x_j} - \beta^*\rho\omega k + \frac{\partial}{\partial x_j}[(\mu + \sigma_k\mu_t)\frac{\partial k}{\partial x_j}] \tag{2.11}$$

$$\begin{aligned}\frac{D\rho\omega}{Dt} = \frac{\gamma}{\mu_t}\tau_{ij}\frac{\partial u_i}{\partial x_j} - \beta\rho\omega^2 + \frac{\partial}{\partial x_j}[(\mu + \sigma_\omega\mu_t)\frac{\partial \omega}{\partial x_j}] \\ + 2(1 - F_1)\sigma_{\omega 2}\rho\frac{1}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j}\end{aligned} \tag{2.12}$$

$$\mu_t = \frac{a_1\rho k}{max(a_1\omega, \Omega F_2)} \tag{2.13}$$

where Ω is the magnitude of the vorticity. The boundary conditions are the same as for Wilcox k-omega and the constants are a combination of constants related to the original models, defined as:

$$\Phi = F_1\Phi_1 + (1 - F_1)\Phi_2 \tag{2.14}$$

The functions and constants are:

$$F_1 = than(arg_1^4) \tag{2.15}$$

$$arg_1 = min[max(\frac{\sqrt{k}}{0.09\omega d}; \frac{500\nu}{d^2\omega}), \frac{4\sigma_{\omega 2}k}{CD_{k\omega}d^2}] \tag{2.16}$$

$$CD_{k\omega} = max(2\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j}, 10^{-20}) \tag{2.17}$$

$$F_2 = tanh(arg_2^2) \tag{2.18}$$

$$arg_2 = max(2\frac{\sqrt{k}}{0.09\omega d}; \frac{500\nu}{d^2\omega}) \tag{2.19}$$

where d is the distance to the closest surface.

$$\begin{array}{ccc}\sigma_{k1} = 0.85 & \sigma_{\omega 1} = 0.85 & \beta_1 = 0.075 \\ a_1 = 0.31 & \beta^* = 0.09 & \kappa = 0.41 \\ \sigma_{k2} = 1.0 & \sigma_{\omega 2} = 0.856 & \beta_2 = 0.0828\end{array}$$

$$\gamma_1 = \frac{\beta_1}{\beta^*} - \frac{\sigma_{\omega 1}\kappa^2}{\sqrt{(\beta^*)}}$$

$$\gamma_2 = \frac{\beta_2}{\beta^*} - \frac{\sigma_{\omega 2}\kappa^2}{\sqrt{(\beta^*)}}$$

2.5 Discretization and Boundary Conditions

The problems are discretized with forward Euler scheme in time and second order central differencing for all spacial derivatives. In Figure 2.2 the local positioning used in both this report and the programs is shown.

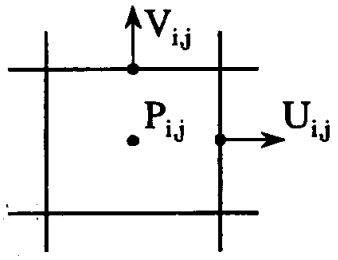

Figure 2.2: MAC grid cell, picture from [1]

For the velocity the boundary conditions are handled with ghost cells, as shown in Figure 2.3, they are zero at all walls. The inlet values are given, while the outlet values are derivatives equal to zero. Boundary conditions for the pressure are derivatives equal to zero at walls, and given or derivatives equal to zero at inlet and outlet.

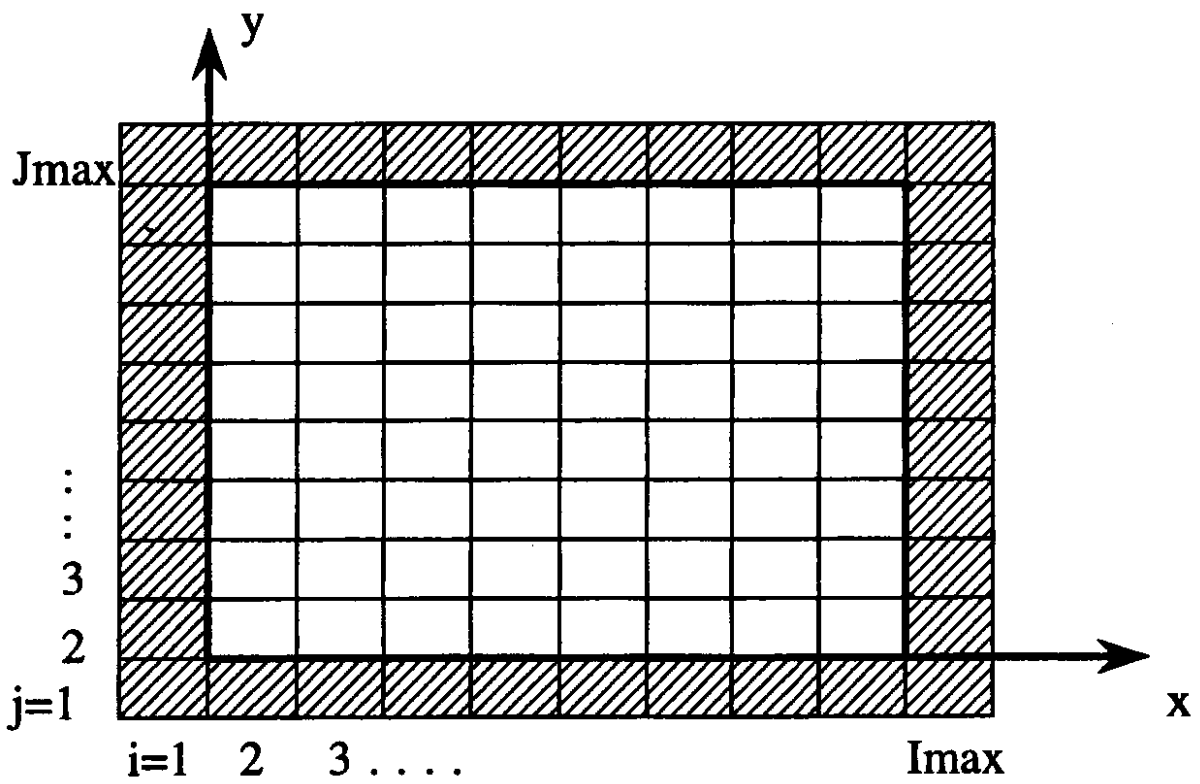

Figure 2.3: Global grid, picture from [1]

Using the described discretization and conservative form for equation 2.1, the equations 2.1, 2.2 and 2.3 gives the following equations:

$$\begin{aligned}\frac{u^*_{i,j}-u^n_{i,j}}{\Delta t} = &-\frac{(u^n_{i+1,j}-u^n_{i,j})^2-(u^n_{i,j}-u^n_{i-1,j})^2}{4\cdot\Delta x}\\ &-\frac{(u^n_{i,j+1}+u^n_{i,j})\cdot(v^n_{i+1,j}+v^n_{i,j})-(u^n_{i,j}+u^n_{i,j-1})\cdot(v^n_{i+1,j-1}+v^n_{i,j-1})}{4\cdot\Delta y}\\ &-\beta\frac{p^n_{i+1,j}-p^n_{i,j}}{\Delta x}+\\ &\frac{1}{Re}\left(\frac{u^n_{i+1,j}-2\cdot u^n_{i,j}+u^n_{i-1,j}}{(\Delta x)^2}+\frac{u^n_{i,j+1}-2\cdot u^n_{i,j}+u^n_{i,j-1}}{(\Delta y)^2}\right)\end{aligned} \tag{2.20}$$

$$\begin{aligned}\frac{v^*_{i,j}-v^n_{i,j}}{\Delta t} = &-\frac{(v^n_{i,j+1}-v^n_{i,j})^2-(v^n_{i,j}-v^n_{i,j-1})^2}{4\cdot\Delta y}\\ &-\frac{(u^n_{i,j+1}+u^n_{i,j})\cdot(v^n_{i+1,j}+v^n_{i,j})-(u^n_{i-1,j+1}+u^n_{i-1,j})\cdot(v^n_{i-1,j}+v^n_{i,j})}{4\cdot\Delta x}\\ &-\beta\frac{p^n_{i,j+1}-p^n_{i,j}}{\Delta y}+\\ &\frac{1}{Re}\left(\frac{v^n_{i+1,j}-2\cdot v^n_{i,j}+v^n_{i-1,j}}{(\Delta x)^2}+\frac{v^n_{i,j+1}-2\cdot v^n_{i,j}+v^n_{i,j-1}}{(\Delta y)^2}\right)\end{aligned} \tag{2.21}$$

Writing $p^{n+1}-\beta p^n$ as Φ for discretization of 2.2 and 2.3:

$$\begin{aligned}\frac{\Phi_{i+1,j}-2\cdot\Phi_{i,j}+\Phi_{i-1,j}}{(\Delta x)^2}+\frac{\Phi_{i,j+1}-2\cdot\Phi_{i,j}+\Phi_{i,j-1}}{(\Delta y)^2} =\\ \frac{1}{\Delta t}\cdot\left(\frac{u^*_{i,j}-u^*_{i-1,j}}{\Delta x}+\frac{v^*_{i,j}-v^*_{i,j-1}}{\Delta y}\right)\end{aligned} \tag{2.22}$$

$$\frac{u^{n+1}_{i,j}-u^*_{i,j}}{\Delta x} = -\frac{\Phi_{i+1,j}-\Phi_{i,j}}{\Delta x} \tag{2.23}$$

$$\frac{v^{n+1}_{i,j}-v^*_{i,j}}{\Delta y} = -\frac{\Phi_{i,j+1}-\Phi_{i,j}}{\Delta y} \tag{2.24}$$

The boundary conditions for the tentative velocity field is difficult to interpret physically, also affecting the boundary conditions of Φ in equation 2.22. It can be shown, as by R. Kristoffersen[1], that the solution of U^{n+1} at the boundaries are independent of the boundary values of U^*. R. Kristoffersen further shows that if these are set as Dirichlet conditions, the boundary conditions for Φ is Neumann conditions. Therefore the boundary condition for U^* is the same as for U, while Φ has the same as P, coded directly for the numerical solver.

Chapter 3

Multi -processor and -thread Programming

3.1 Introduction to MP

To investigate the gain of multiprocessor, or multi thread programs, compared to single thread programs in FORTRAN the sum

$$\sum_{n=0}^{\infty} \frac{1}{n^2} = 2 \tag{3.1}$$

is considered.

CPU time is a good measure of the workload required to run a program. It is available as a function or subroutine, intrinsically, as well as in most libraries. On one thread it is possibly the best way to measure time spent, as it returns the actual time spent on the program by the processor. For several processors it returns the sum of all the individual times, resulting in a far larger number than the real time, and is therefore not a good measure on the efficiency of a multiprocessor program. Therefore the real time is used to test the efficiency, with the system_clock intrinsic subroutine.

3.2 OpenMP

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs[6]. As OpenMP 3.1 and 3.0 is supported by NAG FORTRAN Compiler 6.0, its simplicity and availability makes it a natural choice for this report. It is simply implemented by commands written with the prefix !$ and, in NAG FORTRAN Compiler, the option -openmp when compiling. The command !$PARALLEL is used to declare a parallel region, while the next command define how the work should be shared between the threads. The relevant commands for this project is shown in the following list.

- !$DO
- !$SECTIONS
- !$WORKSHARE

3.3 Results and Discussion for MP

With quadruple precision floating point variables in equation 3.1, n equal to 150 or higher gives underflow, and so the sum is run from 0 to 149. It is also run from 149 to 0, to ensure no information is lost when adding a small number to a large number. Both loops return the same sum for the same value of n, and the correct sum of 2.0 for n $\gtrsim 94$. This may seem like a contradiction, but is simply due to the fact that numbers and arithmetics on a computer is never accurate. This may further be shown by the intrinsic functions tiny(x), returning the smallest positive number representable by the same kind as x, and epsilon(x), returning the smallest positive number of the same kind as x that may be added to 1 and give an answer larger than 1. For quadruple precision the ratio between the two numbers are; $\frac{tiny(x)}{epsilon(x)} \simeq 8.12987 \cdot 10^{-261}$.

The first test was to execute the two sums on a single tread compared to executing them on two threads, one thread for each sum. It was also placed inside a loop for computing the sums 15 or $25 \cdot 10^5$ times. All programs executed 15 times with the results, mean and variance of time, shown in table 3.1. From this it is visible that parallelization comes with a cost of extra computations. Without the loop the task is not measurable at millisecond precision, but sheared on two threads it is just visible. Comparing the results from the longest loop, the total increase in computational time amounts to approximately 15% and a substantial increase in variance, 220 times higher.

	Nr of loops	**Mean [ms]**	**Variance [ms^2]**
Single thread	1	0	0
	15	0.4667	0.2667
	$25 \cdot 10^5$	3593	26.43
Two threads	1	6.6	7.114
	15	7.818	6.442
	$25 \cdot 10^5$	4131	5825

Table 3.1: Table of real time spent during program execution

Splitting the $25 \cdot 10^5$ iteration loop in two and allowing it to run on two different threads gives a time of 2704 ms, resulting in a gain of 25%. However, the variance is increased to 180 times higher at 4685 ms^2, further showing the uncertain overhead computations added by parallelization.

Table 3.2 shows the effect of entering and leaving the parallel region on every computation compared to only create and leave the parallel region once. The effect of entering and leaving the parallel region more than once amounts to increasing the computational time to 11 times higher, a result underlining the difficulties one can meet during parallel programming.

	Nr of loops	**Mean [ms]**	**Variance [ms^2]**
Parallel command outside loop	15	7.818	6.442
	$25 \cdot 10^5$	4131	5825
Parallel command inside loop	15	32.87	228.1
	$25 \cdot 10^5$	45540	2023000

Table 3.2: Table of real time spent during program execution depending on position of !$OMP PARALLEL

The final test was to utilize all processors on the loop using to !$OMP DO construct, resulting in a time of 1319, a gain of 63%. This construct is very easy to use and a good choice when possible.

3.4 Conclusion for MP

To use several threads efficiently it is important to identify a large amount of computations that may be carried out in parallel. If the workload is too low, there is no gain and possibly a loss in parallelization of the code. Yet, with OpenMP, parallel programming may be both beneficial and easy to implement. The biggest challenges is identifying possible parallel regions and the optimal parallelization method.

Chapter 4

1D Channel Flow

4.1 Introduction to 1D Channel Flow

The steady state of the simple problem of a pressure driven incompressible flow between two plates provides an excellent study case for turbulence models. Its simplicity and the large number of DNS data available from the simulations of Kim, Moin and Mansour[7],hereafter KMM, makes for easy implementation and comparison.

4.2 Problem Setup for 1D Channel Flow

The domain of 1x1 is discretized on a 1x600 grid which is simplified by a symmetry plane to 1x300. All values are saved in the cell center, except the pressure, which is prescribed at the inlet and outlet. The streamwise boundary conditions are the derivative equal to zero and the boundary condition at the wall shown in table 4.1.

Variable	BC at wall
u	u=0
k	k=0
μ_t	$\mu_t = 0$
ω	$\omega = 10\frac{6\nu}{\beta_1(\Delta y_1)^2}$

Table 4.1: The boundary conditions at the wall. Δy_1 is the distance to the closest surface.

Second order central differencing is used on all spacial derivatives and first order Euler scheme for time, using the time as an iterative dimension. With:

$$\frac{\partial u_j^n}{\partial y} = \frac{u_{j+1} - u_{j-1}}{2\Delta y} \tag{4.1}$$

and for any A and Φ;

$$\begin{aligned}\frac{\partial}{\partial y}[(\mu + A\mu_t)\frac{\partial \Phi_j}{\partial y}] = \frac{1}{(\Delta y)^2}[&A_{j+0.5}(\mu + \mu_{tj+0.5})(\Phi_{j+1} - \Phi_j) \\ &- A_{j-0.5}(\mu + \mu_{tj-0.5})(\Phi_j - \Phi_{j-1})]\end{aligned} \tag{4.2}$$

$$A_{j+0.5} = 0.5 \cdot (A_{j+1} + A_j) \tag{4.3}$$

Then by defining the pressure at the inlet to be $\frac{P}{\rho} = 1$ and 0 at the outlet, $\frac{\partial P}{\partial x} = -1$, the Navier-Stokes equation reduces to:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = 1 + \frac{1}{\partial y}[(\nu + \nu_t^n)\frac{\partial u_j^n}{\partial y}] \tag{4.4}$$

The system is run with $\Delta t = 1E-6$ and considered converged when the second norm of the vector $u(t) - u(t-1)$ is less than $1E-4$.

As the shear stress is known by balance of the forces, the shear velocity in the system is the square root of 0.5. Then, with Re_τ defined by half the channel height and u_τ, as in KMM, the problem formulation allows Re_τ to be chosen by choosing the kinematic viscosity.

4.2.1 Turbulence Models

The equations in the different turbulence models are greatly simplified by the boundary conditions and discretization, as shown below:

Wilcox K-Omega

Turbulent kinetic energy:

$$\rho\frac{k_j^{n+1} - k_j^n}{\Delta t} = \mu_t^n(\frac{\partial u_j^n}{\partial y})^2 - \beta^*\rho\omega_j^n k_j^n + \frac{\partial}{\partial y}[(\mu + \sigma_k\mu_t^n)\frac{\partial k_j^n}{\partial y}] \tag{4.5}$$

Specific dissipation:

$$\rho\frac{\omega_j^{n+1} - \omega_j^n}{\Delta t} = \rho\gamma(\frac{\partial u_j^n}{\partial y})^2 - \beta\rho(\omega_j^n)^2 + \frac{\partial}{\partial y}[(\mu + \sigma_\omega\mu_t^n)\frac{\partial \omega_j^n}{\partial y}] \tag{4.6}$$

Menter K-Omega SST

Turbulent kinetic energy:

$$\rho\frac{k_j^{n+1} - k_j^n}{\Delta t} = \mu_t^n(\frac{\partial u_j^n}{\partial y})^2 - \beta^*\rho\omega_j^n k_j^n + \frac{\partial}{\partial y}[(\mu + \sigma_k\mu_t^n)\frac{\partial k_j^n}{\partial y}] \tag{4.7}$$

Specific dissipation:

$$\begin{aligned}\rho\frac{\omega_j^{n+1} - \omega_j^n}{\Delta t} = \gamma\rho(\frac{\partial u_j^n}{\partial y})^2 - \beta\rho(\omega_j^n)^2 + \frac{\partial}{\partial y}[(\mu + \sigma_\omega\mu_t^n)\frac{\partial \omega_j^n}{\partial y}] \\ + 2(1 - F_1)\sigma_{\omega 2}\rho\frac{1}{\omega_j^n}\frac{\partial k_j^n}{\partial y}\frac{\partial \omega_j^n}{\partial y}\end{aligned} \tag{4.8}$$

$$\frac{1}{\omega_j}\frac{\partial k_j}{\partial y}\frac{\partial \omega_j}{\partial y} = \frac{1}{\omega_j}\frac{(k_{j+1} - k_{j-1})(\omega_{j+1} - \omega_{j-1})}{4(\Delta y)^2} \tag{4.9}$$

4.3 Results and Discussion for 1D Channel Flow

Two Re_τ numbers were investigated. To obtain $Re_\tau = 590$, corresponding to $Re_{umean} \approx 13400$, ν was set to $6 \cdot 10^{-4}$, and for a less turbulent case, $Re_\tau = 180$ and $Re_{umean} \approx 3300$, ν was set to $2 \cdot 10^{-3}$. The same results were obtained on a twice as fine grid, implying grid independence.

Although this should be a straight forward 1D case the models appear to be numerically unstable, demanding a time step in the order of one percent of the laminar case. They also need an adequate initial guess for the turbulent variables, in this case obtainable from the DNS data.

The complex Menter k-omega SST uses significantly more time compared to the simpler Wilcox k-omega. Keeping in mind the simplification of the Navier-Stokes equation, it may not be noticeable in more advanced cases, but it is apparent in this simple case.

4.3.1 Results $Re_\tau = 180$

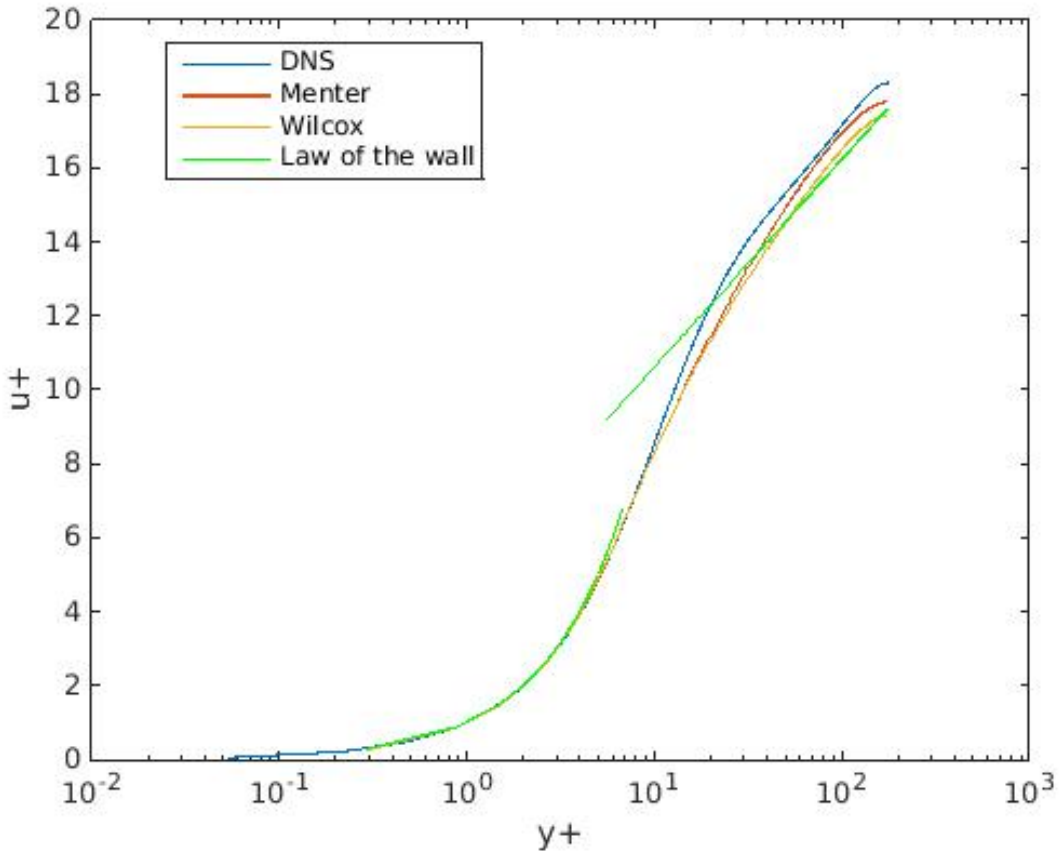

Figure 4.1: Logarithmic plot of velocity versus y+

The mean velocity, as shown in Figure 4.1, is adequately predicted for most purposes with both models, compared to both law of the wall and the DNS solution of KMM. While Menter k-omega SST predicts close to the DNS values, revealing the empirical approach

	U_c/U_{cDNS}	U_c/U_{cLotW}
Menter k-omega SST	0.972	1.010
Wilcox k-omega	0.952	0.989

Table 4.2: Comparison of center channel velocities

when designing the model, Wilcox k-omega predicts closer to the law of the wall, an important part in the deduction of the model.

Table 4.2 shows a comparison of the center channel velocities from Figure 4.1. As the law of the wall is technically inapplicable to the center of the channel the DNS solution is the most interesting comparison, meaning the error is in the 3.0-5.0 % area.

Both models fail to predict the sharp gradient after the sublayer, resulting in an error none of the models manage to compensate for. As can be seen from both table 4.2 and figure 4.1, Menter k-omega SST provides the velocity field closest to the solution.

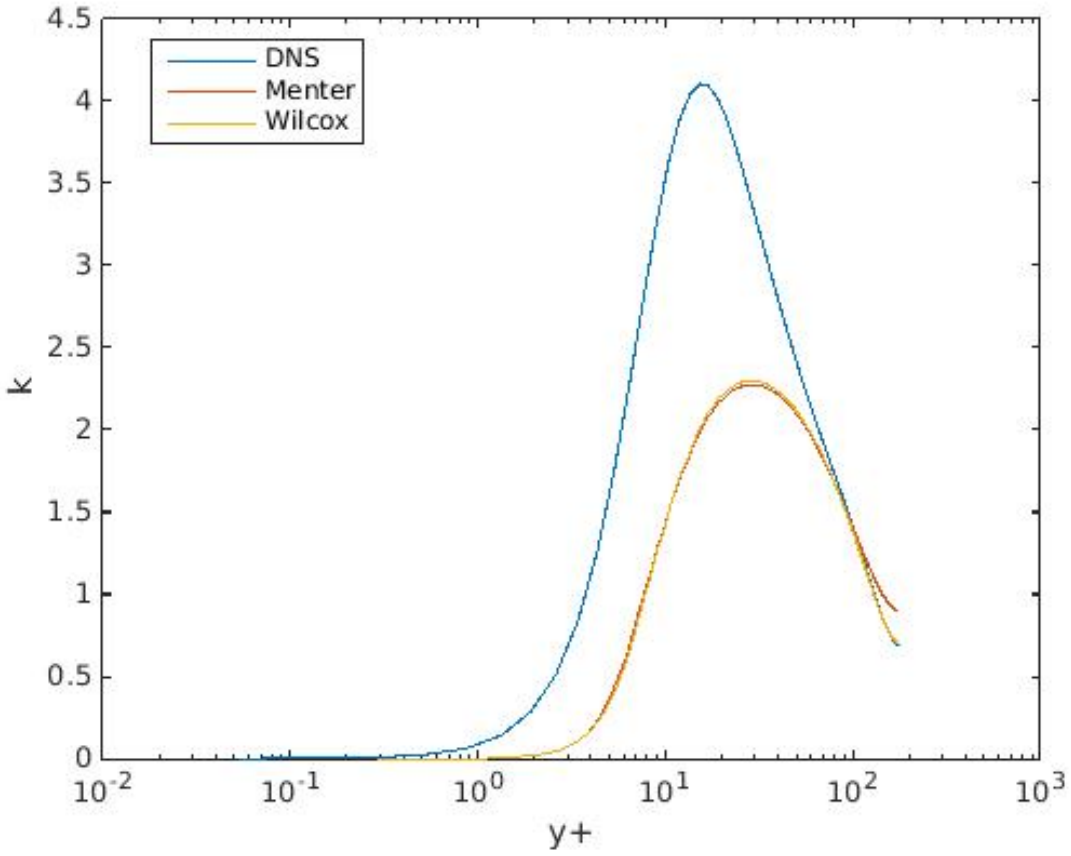

Figure 4.2: Logarithmic plot of turbulent kinetic energy versus y+

Figure 4.2 shows the turbulent kinetic energy distribution in the channel. The value close to the wall is severely off the DNS solution, as expected[4], but the two models provides practically the same solution. For more than the upper half of the channel the values are similar to the DNS solution.

4.3.2 Results $Re_\tau = 590$

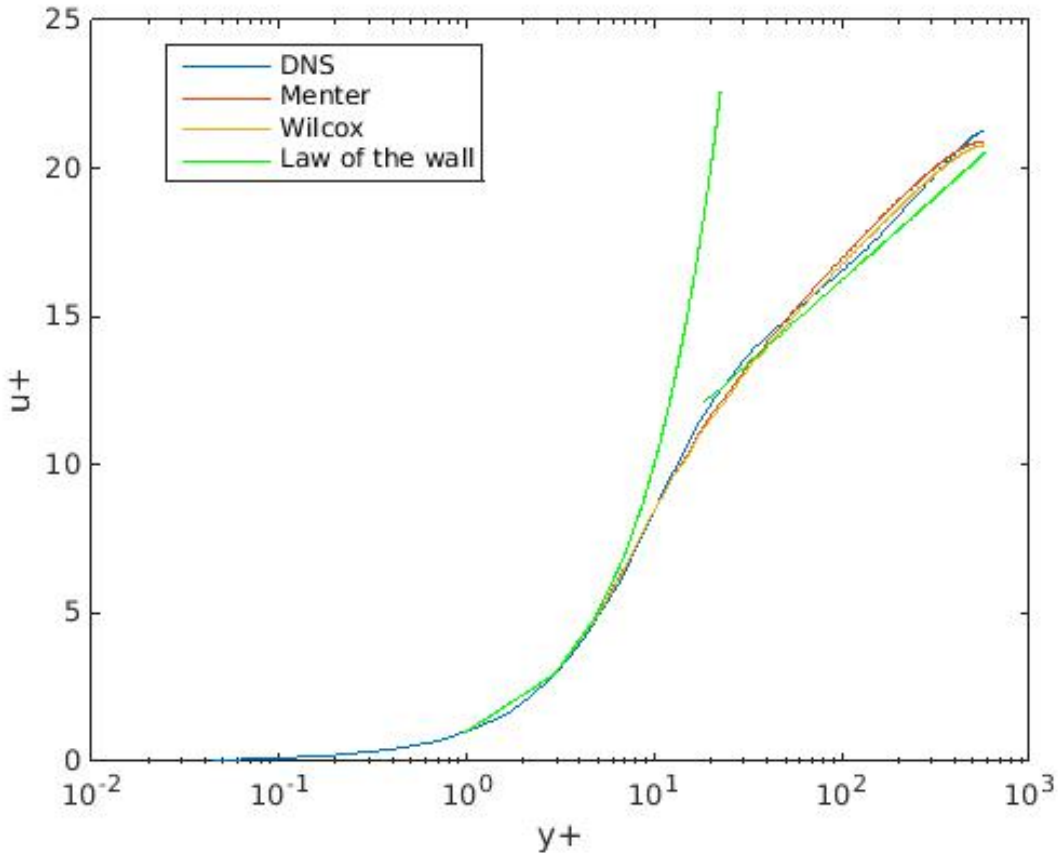

Figure 4.3: Logarithmic plot of velocity versus y+

The models predict better values for this higher Reynolds number, with the error in the range 1.5-2.5 %, see table 4.3. This is expected, as higher Reynolds numbers are easier to predict. Further, as with lower Reynolds number, Menter k-omega SST has the closest prediction for the center channel velocity and the average velocity.

The turbulent kinetic energy, figure 4.4, has the same tendency as the $Re_\tau = 180$.

	U_c/U_{cDNS}	U_c/U_{cLotW}
Menter k-omega SST	0.984	1.018
Wilcox k-omega	0.977	1.012

Table 4.3: Comparison of center channel velocities

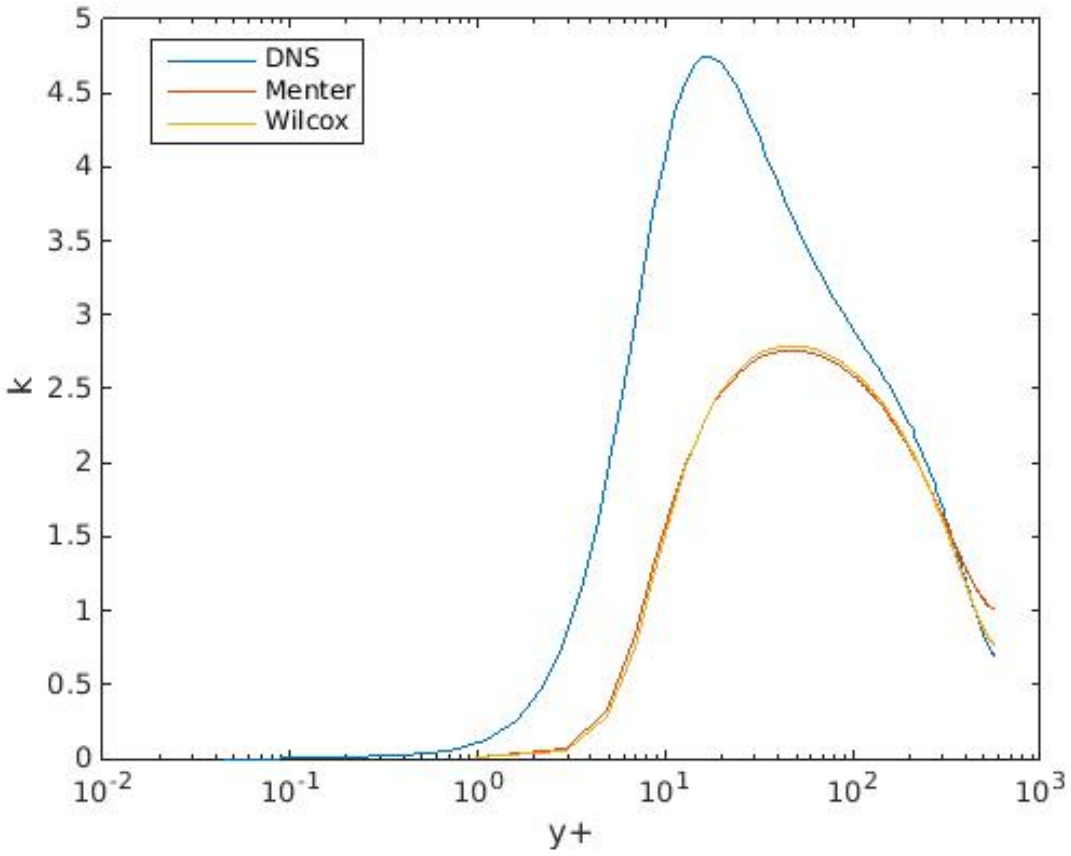

Figure 4.4: Logarithmic plot of turbulent kinetic energy versus y+

4.3.3 Eddy Viscosity

The eddy viscosity is an important parameter as it shows the effect of the different models on the mean flow. It is the only turbulent parameter directly effecting the equations for the mean velocities, and subsequently it is the only parameter the turbulence models need to predict correctly in order to produce the correct velocity field. When comparing the models one could expect that the average difference in the eddy viscosity would be directly related to the average difference in the velocity, but as table 4.4 implies this is not the case. Instead the figures 4.5 and 4.6 suggests that the difference in the cells close to the wall effects this more as figure 4.5 shows a bigger difference in the first 150 cells compared to figure 4.6. This is an interesting point, especially considering the description of the eddy viscosity as "varying with local flow conditions and geometry" and the extra effort to model this property made in the Menter k-omega SST.

	$Norm_2(\frac{u_W^+ - u_M^+}{max(u_M^+)})$	$Norm_2(\frac{\nu_{tW} - \nu_{tM}}{max(\nu_{tM})})$
Re180	0.366	1.61
Re590	0.171	1.88

Table 4.4: Numerical comparison of velocity and ν_t field

Another point towards advanced modeling of the eddy viscosity is the strange shape of the Wilcox k-omega model in the center of the channel. While the Menter k-omega SST model behaves as expected with a maximum in the middle of the channel, the Wilcox k-omega predicts a local minimum. Although strange and unexpected, perhaps even unphysical, this does not seem to have a large effect on the solution of the problem.

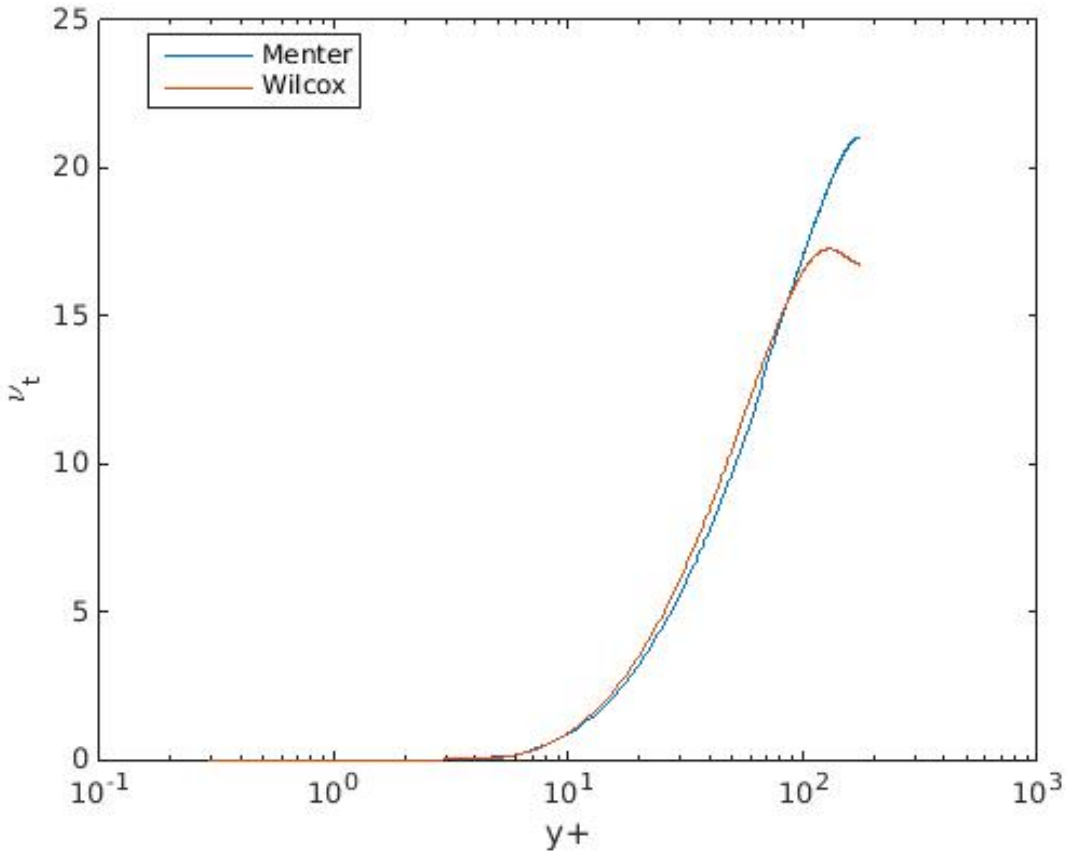

Figure 4.5: $Re_{u\tau} = 180$, Logarithmic plot of eddy viscosity versus y+

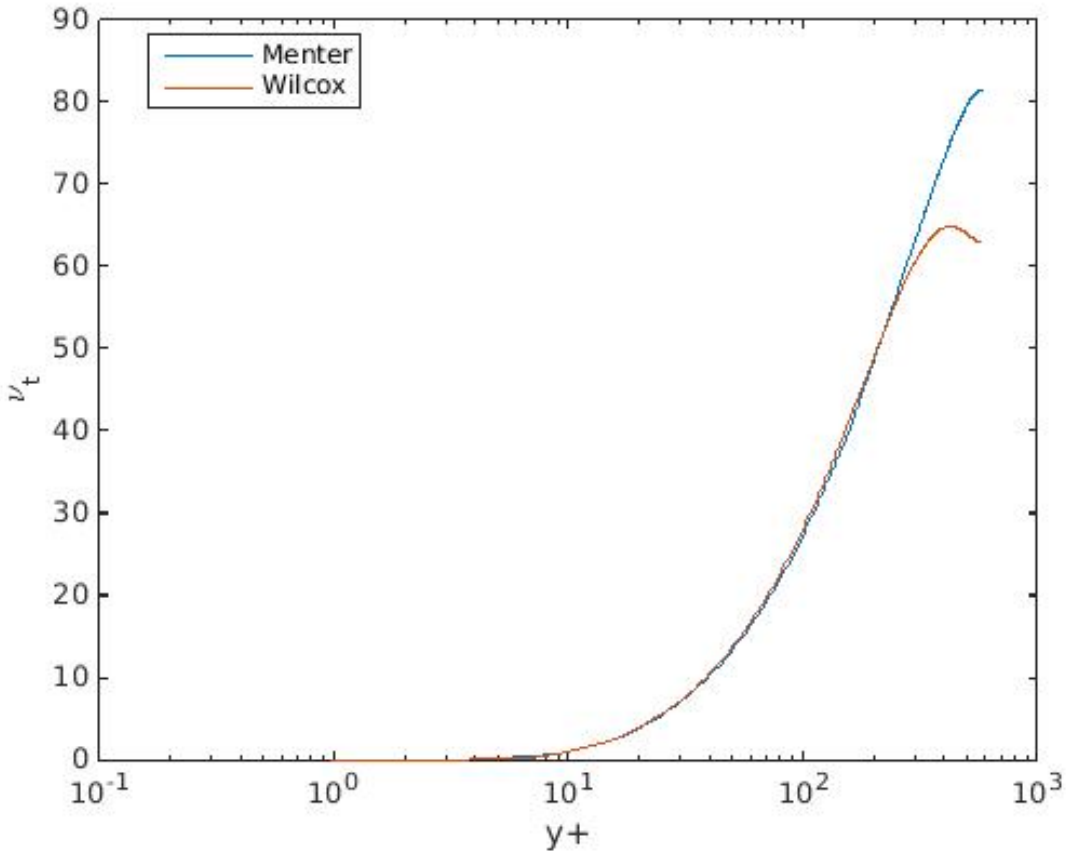

Figure 4.6: $Re_{u\tau} = 590$, Logarithmic plot of eddy viscosity versus y+

4.3.4 Attempt to use MATLAB's bvp-solver

A natural way to solve this case would be as a boundary value problem and it was attempted to do this for the Menter k-omega SST model in MATLAB using the inherent and powerful bvp4c function. The results were unsuccessful for grids and initial guesses later found to yield a solution with the method described in the problem setup of this chapter.

The online documentation for MATLAB states:

bvp4c is a finite difference code that implements the three-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a C1-continuous solution that is fourth-order accurate uniformly in [a,b]. Mesh selection and error control are based on the residual of the continuous solution.[8]

The following could be possible reasons to why the bvp4c fails:

1. The system is too stiff or otherwise numerically unstable.

2. The bvp4c routine changes the grid continuously and the BC for ω is dependent on the first grid size.

3. The derivative of the viscosity is needed, but it is not defined in the model. The treatment of this may be incorrect.

The Equation System

The reduced equations for velocity, turbulent kinetic energy and dissipation in the Menter k-omega SST gives the following system:

$$u_1' = u_2 = \frac{\partial u}{\partial y} \tag{4.10}$$

$$u_2' = \frac{\partial^2 u}{\partial y^2} = \frac{-1}{\nu + \nu_t}(1 + \frac{\partial \nu_t}{\partial y}\frac{\partial u}{\partial y}) \tag{4.11}$$

$$k_1' = k_2 = \frac{\partial k}{\partial y} \tag{4.12}$$

$$k_2' = (\frac{-1}{\nu + \sigma_k \nu_t})(\nu_t(\frac{\partial u}{\partial y})^2 - \beta\omega k + \sigma_k \frac{\partial \nu_t}{\partial y}\frac{\partial k}{\partial y}) \tag{4.13}$$

$$\omega_1' = \omega_2 = \frac{\partial \omega}{\partial y} \tag{4.14}$$

$$\omega_2' = (\frac{-1}{\nu + \sigma_\omega \nu_t})(\frac{\gamma}{\nu_t}(\frac{\partial u}{\partial y})^2 - \beta\omega^2 + \sigma_\omega \frac{\partial \nu_t}{\partial y}\frac{\partial \omega}{\partial y}) + 2(1 - F_1)\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial k}{\partial y}\frac{\partial \omega}{\partial y} \tag{4.15}$$

$$\nu_t = \frac{a_1 k}{max(a_1\omega; \Omega F_2)} \tag{4.16}$$

and $\frac{\partial \nu_t}{\partial y}$ is defined using the standard rule $(\frac{f}{g})' = \frac{fg' - f'g}{g^2}$, where $f = a_1 k$ and $g = max(a_1\omega; \Omega F_2)$

4.4 Conclusions for 1D Channel Flow

Simulating the mean flow in a simple turbulent case is possible with both Menter k-omega SST and Wilcox k-omega as the models are presented in their respective papers. For both low and medium Reynolds numbers the results are adequate for most engineering purposes. The parameter effecting the mean flow, the eddy viscosity, appears to be best with the treatment of Menter k-omega SST, suggesting the extra computations are well worth it. It also suggests the treatment of this parameter as a relation between turbulent kinetic energy and dissipation only may not be correct for certain flow patterns or geometry. In addition the results imply that if the wall interaction is of little importance, very good mean flow may be obtained with wall functions, effectively skipping the sublayer where the models are furthest from predicting the correct solution.

The encountered challenges when simulating are mainly related to running the simulation due to the numerical instability of the models. With the small time step needed, even this simple problem takes time solving. There is also the issue of obtaining a good initial field before the simulation even starts.

The turbulent kinetic energy is an important part of the models, it is the only turbulent variable that has an obvious physical interpretation and definition. Yet the models fail entirely to predict its value near the wall, while the other values are adequate at best. From this three questions arise; "does it matter?", "is the turbulent kinetic energy in the model the same as the theoretical turbulent kinetic energy?" and "how does this effect the eddy viscosity?".

Chapter 5

K-Epsilon on 1D-Channel Flow $Re_\tau = 590$

K-Epsilon is perhaps the most famous two-equation model and widely used in the industry. Because it is known to fail in the lower regions of the boundary layer most implementations of the model use wall functions and demand the first cell to be at $y^+ > 30$, effectively neglecting the most interesting part of this problem. As it also is known to work poorly with adverse pressure gradients, it was decided to compute it as a worst case scenario of the case in chapter 4, using it exactly as Wilcox k-omega and Menter k-omega SST. This version of the model is taken from Ferziger[3];

$$P_k = (\mu_t(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}) - \frac{2}{3}\rho\delta_{ij}k)\frac{\partial u_i}{\partial x_j} \tag{5.1}$$

$$\frac{D\rho k}{Dt} = P_k - \rho\epsilon + \frac{\partial}{\partial x_j}[(\mu + \frac{\mu_t}{\sigma_k})\frac{\partial k}{\partial x_j}] \tag{5.2}$$

$$\frac{D\rho\epsilon}{Dt} = C_{\epsilon 1}\frac{\epsilon}{k}P_k - \rho C_{\epsilon 2}\frac{\epsilon^2}{k} + \frac{\partial}{\partial x_j}(\frac{\mu_t}{\sigma_\epsilon}\frac{\partial \epsilon}{\partial x_j}) \tag{5.3}$$

$$\mu_t = \rho C_\mu \frac{k^2}{\epsilon} \tag{5.4}$$

Boundary conditions: $\epsilon_{wall} = \nu\frac{\partial^2 k}{\partial y^2}|_{wall}$ and $k_{wall} = 0$

$$\begin{array}{lll} \sigma_k = 1.0 & \sigma_\epsilon = 1.3 & C_\mu = 0.09 \\ C_{\epsilon 1} = 1.44 & C_{\epsilon 2} = 1.92 & \end{array}$$

The discretized equations then becomes;

$$P_k = \mu_{tj}(\frac{\partial u_j^n}{\partial y})^2 \tag{5.5}$$

$$\rho\frac{k_j^{n+1} - k_j^n}{\Delta t} = P_k - \rho\epsilon_j + \frac{\partial}{\partial y}[(\mu + \frac{\mu_{tj}}{\sigma_k})\frac{\partial k}{\partial y}] \tag{5.6}$$

$$\rho\frac{\epsilon_j^{n+1} - \epsilon_j^n}{\Delta t} = C_{\epsilon 1}\frac{\epsilon_j}{k_j}P_k - \rho C_{\epsilon 2}\frac{\epsilon_j^2}{k_j} + \frac{\partial}{\partial y}(\frac{\mu_{tj}}{\sigma_{\epsilon_j}}\frac{\partial \epsilon_j}{\partial y}) \tag{5.7}$$

$$\mu_{tj} = \rho C_\mu \frac{k_j^2}{\epsilon_j} \tag{5.8}$$

$$\epsilon_{wall} = \nu \frac{2k_{j=1}}{(0.5\Delta y)^2} \tag{5.9}$$

5.1 Results, Discussion and Conclusion for K-Epsilon

Attempts to use the solution from chapter 4, with $\epsilon = \omega$, failed at first. However, using $\epsilon = \omega \cdot k$ was enough to get a solution.

As expected, the k-epsilon fails to predict the mean flow completely, as seen in figure 5.1. Furthermore it seems to predict an adequate gradient from $y+ \approx 30$, suggesting it is an adequate model if used with a wall function and first cell in this range.

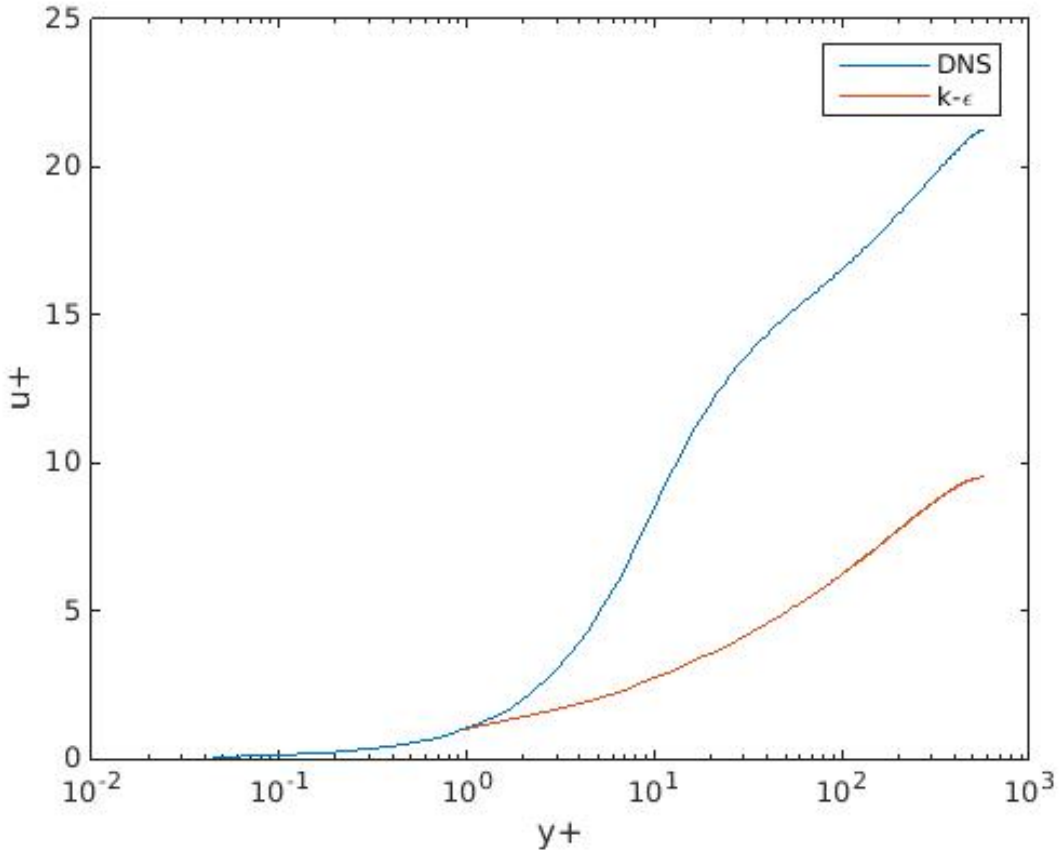

Figure 5.1: Comparison of velocity

This case demonstrates the dangers misusing turbulence models, underlining the need of understanding the theory behind.

Chapter 6

Backward-Facing Step

6.1 Introduction to Backward-Facing Step

The backward-facing step case is a natural choice for testing the models as it is a simple geometry and it has an adverse pressure gradient, which some two-equation models are known to struggle with. In addition the reattachment length is a great case parameter. A DNS by Le, Moin and Kim[9] is used to compare with other results.

6.2 Problem Setup for Backward-Facing Step

A domain of 30h x 6h consists of a inlet section of 10h x 5h which expands in the y direction to a 20h x 6h section. The whole domain is discretized as explained in chapter 2 on a 900x300 grid, leaving a 10h x h section of unused cells. The pressure solver is coded to skip these cells, while the other equations solves every cell and later updates the unused cells. The inlet values are taken from a 1D case with $Re_h = \frac{U_0 h}{\nu} = 5100$, where U_0 is maximum inlet velocity and h is the step height, and $\frac{\partial P}{\partial x} = 0$. The outlet values are derivatives equal to zero and $P = 0$. The 1D case is also used as an initial field of all values.

Due to long simulation time steady state is considered reached when the second norm of change in the horizontal velocity at $x = 23h$ over $1E-2$ seconds is less then $1E-6$.

6.3 Results for Backward-Facing Step

As for the 1D channel flow both models predict largely the same solution, with an average difference in the stream function at 0.0018. Therefore the solution from the Wilcox k-omega will mainly be used for comparison with the DNS. Figures 6.1 and 6.2 show the streamlines, from the simulation and from the DNS respectively. While similar, the two-equation models fail to predict the second small vortex close to the step. They also predict a substantially longer reattachment length, as seen in table 6.1.

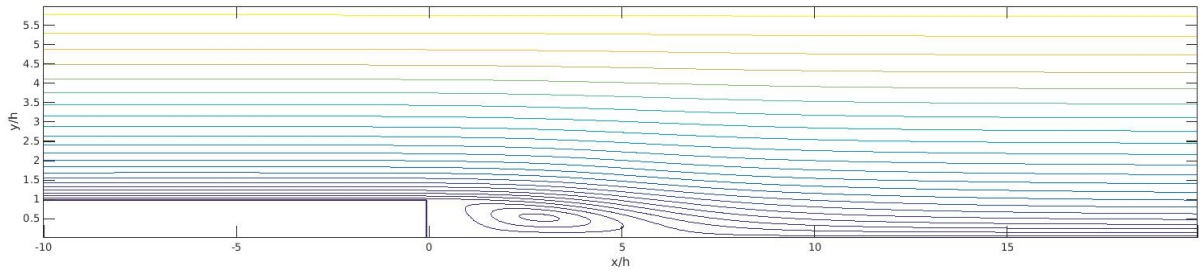

Figure 6.1: Streamline plot from Wilcox k-omega model

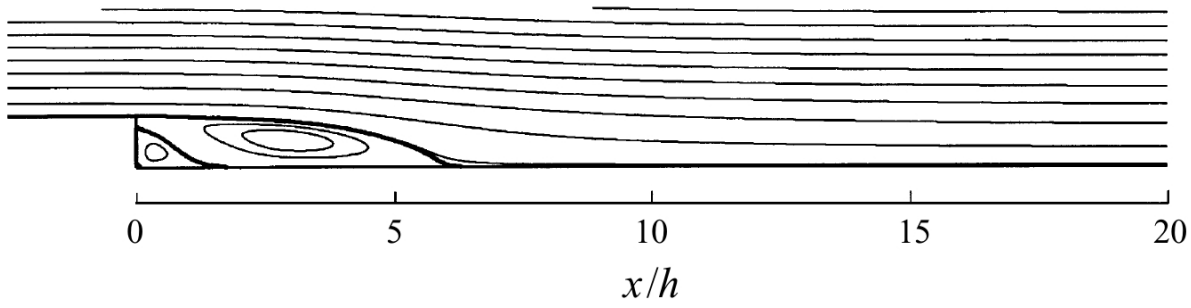

Figure 6.2: Streamline plot from DNS[9]

	Reattachment length [h]	$\frac{h}{h_{DNS}}$
DNS	6.28	1.0
Wilcox	7.2	1.147
Menter	7.23	1.151

Table 6.1: Comparison of predicted reattachment length

Although figures 6.3 and 6.4 show the pressure fields are significantly different, this probably is due to the error in the velocity field. The figures are included to give a better understanding of the accuracy of the solution. The reference pressure P_0 is at $x/h = -5.0$.

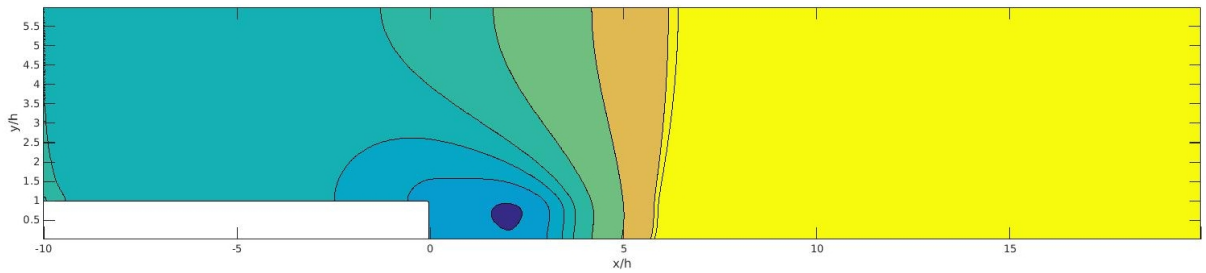

Figure 6.3: Contours of pressure field from Wilcox model, $\frac{P-P_0}{\rho U_0^2}$

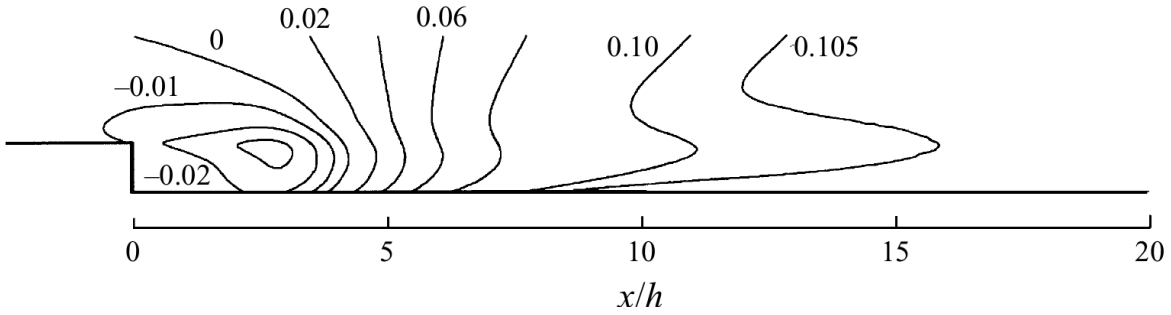

Figure 6.4: Contours of pressure field from DNS[9], $\frac{P-P_0}{\rho U_0^2}$

6.4 Conclusion for Backward-Facing Step

Due to long solution time these results might not be fully converged. Still, the solution was regularly checked during simulation and it does not appear to be far from the converged solution. While some care must be taken using this solution, at least a 10 % error must be expected on the reattachment length. The solution time of this case presents an interesting problem, while the models demand a short time step, the pressure equation takes considerable time solving. As the field approaches steady state the time step may be larger and the pressure equation is quicker, but if steady state is almost found there is no point simulating. If one can bypass this problem, a larger grid could be an interesting test.

Chapter 7

Investigation of Functions and Switches in Menter SST

The Menter k-omega SST model switches between the Wilcox k-omega and k-epsilon models based on flow conditions and geometry. As this is an important part of the behavior of the model, some graphics of F_1, F_2 and ν_t is discussed. While in the 1D case the model always uses Wilcox, the 2D case is more interesting.

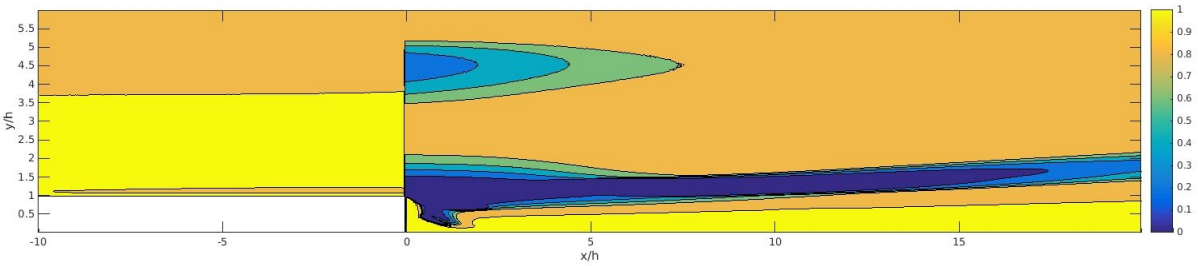

Figure 7.1: Contours of F_1

Figure 7.1 shows the values taken by F_1 across the domain, largely being one until the step. Which means it uses the Wilcox k-omega, consistent with the results from 1D. After this, in a region with much turbulence, F_1 becomes very small, $F_1 \approx 0.01$, using k-epsilon. The regions with low turbulence, like close to the wall or close to the top, still uses Wilcox. A strange region quite high above the step also has a lower F_1, about $F_1 = 0.5$, but this seems to have little effect on the flow and ν_t. The fact that the model switches this much is a bit surprising, keeping in mind the results in chapter 6 are very similar, one would expect seeing more Wilcox in this case. Another interpretation is that both Wilcox and k-epsilon are two-equation models, and switching between them does not greatly alter the solution.

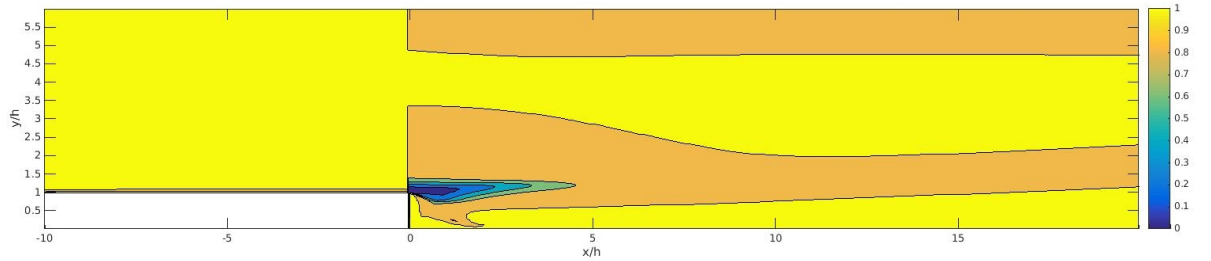

Figure 7.2: Contours of F_2

While the effect of F_2 may be a little difficult to interpret due to being inside a max() condition, when it is zero, the eddy viscosity is defined exactly like in Wilcox. Figure 7.2 shows that in the most turbulent and interesting region, just off the step, F_2 has the value zero. This implies that the advanced eddy viscosity definition is designed to change the effect of less turbulent regions on the mean flow.

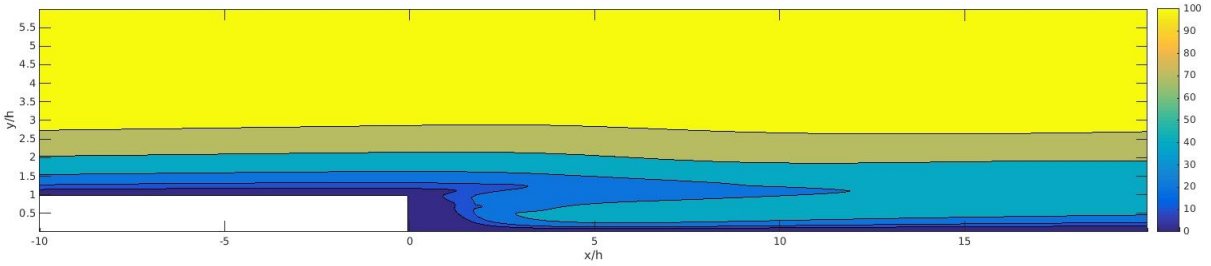

Figure 7.3: Contours of ν_t field

Finally, figure 7.3 is included to show that the distribution is even and as expected from the results in chapter 6. The region of low F_2 seems to coincide with a region of low ν_t.

Chapter 8

Thesis Summary

8.1 Process

In the beginning, chapter 3 was quite quickly done, as openMP is easy to use. Turning out be a dead end, focus was moved on to the 1D turbulent case, which was thought to be a relatively simple task. The combination of high numerical instability, need of adequate initial field and low FORTRAN knowledge proved it to be very difficult. After a long period of trial and error, a solution was obtained for both low and medium Reynolds number. Instead of the natural choice of a high Reynolds number case it was decided to move on to the backward-facing step case, as the deadline was approaching.

A 2D repeating boundary conditions channel flow program was made to confirm the 2D implementation of the models, before creating the backward-facing step program. It took some time to create the sparse matrices needed for the pressure solver, but most of the time went to simulate the first backward-facing step solution. After a failed attempt at a too coarse grid the simulation on the final grid went on for days.

As it is common, the model k-epsilon was put to the test to show some dangers when using turbulence modeling. With solution field and comparable data available, this was readily done.

8.2 Conclusions

It was intended to shorten the simulation time with the use of parallel programming, but the algorithm needs to be run sequentially. The only sequence where gain is possible is the elliptic pressure equation, but the library routine is already running in parallel.

Wilcox k-omega and Menter k-omega SST provides very similar solutions results, although Wilcox k-omega appears a little better at the backward-facing step case while Menter k-omega SST appears better at the channel flow case. With an error in the 5 % region in a 1D case, the error in the 10-15 % region is both expected and acceptable. It is well within acceptable limits for most engineering purposes for both cases.

Looking into the near-wall treatment of the models shows a discrepancy between the predicted turbulent kinetic energy and the DNS value. While there is no direct connection between the turbulent kinetic energy and the mean flow, this region does coincide with the region where the gradient of the mean flow is under predicted. This may warrant further investigation. For larger cases, where the near-wall flow is of secondary interest,

wall functions may be a better choice then resolving the complete boundary layer. With this in mind, k-epsilon may have some usefulness.

A big surprise was to find all the models to be very numerically unstable and require a initial field close to the solution. It would be very interesting to know how this is solved in CFD software!

In light of the goals to learn more about the nature of two-equation models and FORTRAN programming, this thesis must be seen as a success. Although having been difficult and leaving some questions unanswered, a thesis like this is an excellent way to dive into theory only glanced at through NTNU.

8.3 Further Work

There are some natural ways to continue this work;

1. High Reynolds number channel flow and high Reynolds number backward-facing step.
2. Investigate wall functions for k, ω and ϵ.
3. Investigate different cases.
4. Investigate effect of different modeling of the equations in near-wall regions.
5. If not using a library function to solve the pressure equation, investigate multiprocessor programming with MPI or openMP.

Despite this, the author would suggest nobody continues this work. The systems are touchy and simulation takes a very long time. Deeper investigation of the theory of models may be to advanced for a regular masters degree. It is then imperative that the student is strongly familiar with the programming language and at least have programmed a laminar CFD program.

Appendix A

Functions Used

This appendix aims to give an overview of the Fortran functions used in the project.

A.1 Fortran Intrinsic Functions

A.1.1 Shape

Syntax:
result = shape(source)
Description:
Returns the shape of an array. Result is an integer array of rank one with as many elements as source has dimensions.

A.1.2 Reshape

Syntax:
result = reshape(array,shape)
Description:
Changes the shape of an array to correspond to the shape argument
Hint:
Shape may be the shape function with a valid argument

A.1.3 Norm2

Syntax:
result = norm2(array)
Description:
Returns the second norm of the array. Result = $||array||_2$

A.2 NAG Library Functions

For more information see the online manual at:
http://www.nag.com/numeric/fl/nagdoc_fl24/html/FRONTMATTER/manconts.html

A.2.1 X05AAF

Syntax:
x05aaf(result)
Description:
Returns result as an integer array with seven entries. Result = [year, month(1-12), day(1-31), hour(0-23), minute(0-59), second(0-59), millisecond(0-999)]

A.2.2 F11DAF

Syntax:
f11daf(n, nnz, a, la, irow, icol, lfill, dtol, pstrat, milu, ipivp, ipivq, istr, idiag, nnzc, npivm, iwork, liwork, ifail)
Description:
Setup and optional preconditioner of sparse matrix. Compresses the matrix and performs incomplete LU-decomposition.
Variables:
integer: n, nnz, la, irow(la), icol(la), lfill, ipivp(n), ipivq(n), istr(n+1), idiag(n), nnzc, npivm, iwork(liwork), liwork, ifail
real(kind=nag_wp): a(la), dtol
character(1): pstrat, milu

A.2.3 F11BDF

Syntax:
f11bdf(method, precon, norm, weight, iterm, n, m, tol, maxitn, anorm, sigmax, monit, lwreq, work, lwork, ifail)
Description:
Setup for f11bef for the iterative solution of a real sparse general system of simultaneous linear equations. Method defines the method used, here BiCGSTAB, while tol and norm decides the termination.
Variables:
integer: iterm, n, m, maxitn, monit, lwreq, lwork, ifail
real(kind=nag_wp): tol, anorm, sigmax, work(lwork)
character: method, precon, norm, weight

A.2.4 F11BEF

Syntax:
f11bef(irevcm, u, v, wgt, work, lwork, ifail)
Description:
Iterative solver for real sparse general system of simultaneous linear equations, Au=v, which calls for other functions to compute matrix/vector-operations.
Variables:
integer: irevcm, lwork, ifail

real(kind=nag_wp): u(*), v(*), wgt(*), work(lwork)

A.2.5 F11XAF

Syntax:
f11xaf(trans, n, nnz, a, irow, icol, check, x, y, ifail)
Description:
Computes a matrix-vector or transposed matrix-vector product. Both used, but no checking.
Variables:
integer: n, nnz, irow(nnz), icol(nnz), ifail
real(kind=nag_wp): a(nnz), x(n), y(n)
character(1): trans, check

A.2.6 F11DBF

Syntax:
f11dbf(trans, n, a, la, irow, icol, ipivp, ipivq, istr, idiag, check, y, x, ifail)
Description:
Solves a system of linear equations, Mx = y, involving the incomplete LU preconditioning matrix generated from calling f11daf. Never used transposed or checked.
Variables:
integer: n, la, irow(la), icol(la), ipivp(n), ipivq(n), istr(n+1), idiag(n), ifail
real(kind=nag_wp): a(la), y(n), x(n)
character(1): trans, check

A.2.7 Key NAG Variables

To get the wanted method, the variable for F11BDF must in this case be set as Method = 'BiCGSTAB'. It is also important that precon is set as precon = 'P' when preconditioning is used. To obtain the termination criteria used in this paper, variables norm and iterm must be set as norm = 'I' and iterm = 1.

Appendix B

FORTRAN 95 Code for 2D Backward-Facing Step With Wilcox K-Omega Model

```
MODULE field
IMPLICIT NONE
INTEGER    :: iter,itmax,i,j
INTEGER    :: ibar,jbar,ip,jp,imax,jmax,im,jm,ist,jst,istm,jstm,istp,jstp
INTEGER    :: starttime(7),endtime(7),timespent(7)
REAL(KIND=2)   :: delt,t,tmax,delx,dely,epsi,relax,xmax,ymax,rdx,rdy,beta
    ,rdt,rdxx,rdyy,ny
REAL(KIND=2), ALLOCATABLE, DIMENSION (:,:) :: u,v,p,dp,uo,vo,po
END MODULE field

MODULE turbf
IMPLICIT NONE
REAL(KIND=2) :: sigk,bettas,sigom,betta,gamm
REAL(KIND=2), ALLOCATABLE, DIMENSION (:,:) :: d1,nyt,k,om
END MODULE turbf

MODULE nag_values
IMPLICIT NONE
INTEGER                :: iterm, n, m, maxitn, monit, lwreq, lwork, ifail
    , irevcm, ifail1, &
             nnz,la,lfill,nnzc,npivm,liwork
INTEGER, ALLOCATABLE   :: irow(:),icol(:),ipivp(:),ipivq(:),istr(:),idiag
    (:),iwork(:)
REAL(KIND=2)                 :: tol, anorm, sigmax, dtol
REAL(KIND=2), ALLOCATABLE    :: work(:),x(:),b(:),wgt(:) ,a(:)
CHARACTER(8)           :: method
CHARACTER(1)           :: precon, norm, weigth,pstrat,milu
END MODULE nag_values

PROGRAM sola2d
!-----------------------------------------------------------------
!     Solves the navier-stokes equations on the backward-facing
!   step case with Wilcox k-omega turbulence model
!     Version June 2015, Eirik Heroe
!-----------------------------------------------------------------
USE field
```

```
34 IMPLICIT NONE
35 INTEGER    ::  ii,tid,iii,itercount,itermax
36 REAL(KIND=2)     ::  checkValue=1.0E-6,norm
37 REAL(KIND=2), ALLOCATABLE, DIMENSION (:) :: ucheck
38
39 timespent = 0
40 CALL x05aaf(starttime) !Start time
41 CALL setvelval       !Define several important parameters
42 CALL precon_pressure   !Ready the pressure solver
43 CALL init          !Define initial velocity and pressure fields.
44 CALL bcvel           !Boundary conditions
45 CALL setturbval
46 CALL inputfromlastitr
47 !CALL inputfrom1D
48
49 !     ==== start of global loop ====
50 WRITE(*,*) ' Checkvalue = ', checkValue
51 tid = (INT(((1.0)/delt)))
52 itermax = 20
53
54 itercount = 0
55 ucheck = u(700,2:jp)
56 DO ii = 1,tid
57    !Velocity iterations
58    CALL tentvel
59    CALL bcvel
60    CALL congrad_nag_pres
61    CALL calvel
62    CALL bcvel
63    !Turbulence iterations
64    CALL turbitr
65    t = t + delt
66 END DO
67 WRITE(*,*) ' t = ',t
68 norm = (NORM2(u(700,2:jp)-ucheck))
69 WRITE(*,*) ' Norm = ',norm
70 CALL output
71 itercount = itercount + 1
72 WRITE(*,*) ' Iteration ',itercount, ' of ',itermax
73
74 DO ii = 1,(itermax-1)
75    ucheck = u(700,2:jp)
76    DO iii = 1,tid
77       !Velocity iterations
78       CALL tentvel
79       CALL bcvel
80       CALL congrad_nag_pres
81       CALL calvel
82       CALL bcvel
83       !Turbulence iterations
84       CALL turbitr
85       t = t + delt
86   END DO
87   WRITE(*,*) ' t = ',t
88   norm = (NORM2(u(700,2:jp)-ucheck))
89   WRITE(*,*) ' Norm = ',norm
```

```
  CALL output
  itercount = itercount + 1
  WRITE(*,*) ' Iteration ',itercount, ' of ',itermax
   IF (norm<=checkValue) THEN
      WRITE(*,*)' Found steady state'
      EXIT
   END IF
END DO

!     ==== end of global loop ====

CALL x05aaf(endtime) !Time after iterations
timespent = endtime - starttime
CALL output

WRITE(*,*)'simulation finished.'
END PROGRAM sola2d
!-------------------------------------------------------------
SUBROUTINE setvelval
!     purpose:
!     to define most of the parameters
USE field
IMPLICIT NONE
REAL(KIND=2)  :: h

h = 0.1
xmax = 30.0*h
ymax = 6.0*h

ibar = 900
jbar = 300
ip=ibar+1
jp=jbar+1
imax=ibar+2
jmax=jbar+2
im = ibar -1
jm = jbar -1

ist = 300
jst = 50
istp = ist + 1
jstp = jst + 1
istm = ist - 1
jstm = jst - 1

delx = REAL(xmax)/REAL(ibar)
dely = REAL(ymax)/REAL(jbar)

t = 0.0
delt = 1.0E-5

beta  = 0.00

rdx  = 1.0/delx
rdxx = rdx**2.0
rdy  = 1.0/dely
```

```
rdyy = rdy**2.0
rdt  = 1.0/delt

ny = 3.1175E-4

ALLOCATE(u(ibar+2,jbar+2),v(ibar+2,jbar+2),p(ibar+2,jbar+2),dp(ibar+2,
    jbar+2),uo(ibar+2,jbar+2),vo(ibar+2,jbar+2),po(ibar+2,jbar+2))

RETURN
END SUBROUTINE setvelval
!------------------------------------------------------------
SUBROUTINE bcvel
!     purpose:
!     to give boundary conditions to the velocities around the domain
USE field
IMPLICIT NONE

u(ip,1:jmax) = u(ibar,1:jmax)   ! Outlet
v(imax,2:jp) = v(ip,2:jp)       ! Outlet

u(ist,2:jst) = 0.0        ! Vertical "stepwall"
v(ist,1:jstm) = -v(istp,1:jstm) ! Vertical "stepwall"

v(1:imax,jp) = 0.0          ! Symmetry wall
u(1:ip,jmax) = u(1:ip,jp)     ! Symmetry wall

v(istp:imax,1) = 0.0      ! Lower wall
u(istp:ip,1)   = -u(istp:ip,2)  !lower wall

u(2:istm,jst) = -u(2:istm,jstp) ! Horizontal "stepwall"
v(2:ist,jst) = 0.0        ! Horizontal "stepwall"

RETURN
END SUBROUTINE
!------------------------------------------------------------
SUBROUTINE setturbval
!     purpose:
!     to set turbulent values
USE field
USE turbf
IMPLICIT NONE

ALLOCATE(nyt(ibar+2,jbar+2),k(ibar+2,jbar+2),om(ibar+2,jbar+2))

!Usually overwritten:
om = 0.01
nyt = 1.0E-7
k = 0.01



!Verdier, "k-omega"
bettas = 0.09
sigk  = 0.5
sigom = 0.5
```

```
201 betta = 0.075
202 gamm = 5.0/9.0
203
204 END SUBROUTINE
205 !-------------------------------------------------------------
206 SUBROUTINE init
207 !     purpose:
208 !     to initiate the velocity and pressure fields, usually
209 !     overwritten
210 USE field
211 IMPLICIT NONE
212
213 u=0.01
214 v=0.0
215 p=0.0
216 dp = 0.0
217
218 RETURN
219 END SUBROUTINE
220 !-------------------------------------------------------------
221 SUBROUTINE tentvel
222 !     purpose:
223 !     to compute the tentative velocity fields
224 USE field
225 USE turbf
226 IMPLICIT NONE
227 REAL(KIND=2),DIMENSION(im,jbar)   :: visux,visuy
228 REAL(KIND=2),DIMENSION(ibar,jm)   :: visvx,visvy
229
230 uo = u
231 vo = v
232
233 visux = rdxx*((ny+nyt(3:ip,2:jp))*(uo(3:ip,2:jp)-uo(2:ibar,2:jp))-(ny+
      nyt(2:ibar,2:jp))*(uo(2:ibar,2:jp)-uo(1:im,2:jp)))
234 visvy = rdyy*((ny+nyt(2:ip,3:jp))*(vo(2:ip,3:jp)-vo(2:ip,2:jbar))-(ny+
      nyt(2:ip,2:jbar))*(vo(2:ip,2:jbar)-vo(2:ip,1:jm)))
235
236 visuy = rdyy*(     &
237     &    (uo(2:ibar,3:jmax)-uo(2:ibar,2:jp)) * &
238     &    (ny+0.25*(nyt(3:ip,3:jmax)+nyt(3:ip,2:jp)+nyt(2:ibar,3:jmax)+nyt
      (2:ibar,2:jp))) &
239     &    - (uo(2:ibar,2:jp)-uo(2:ibar,1:jbar)) * &
240     &    (ny+0.25*(nyt(3:ip,2:jp)+nyt(3:ip,1:jbar)+nyt(2:ibar,2:jp)+nyt(2:
      ibar,1:jbar))) &
241     &     )
242
243 visvx = rdxx * (     &
244     &    (vo(3:imax,2:jbar)-vo(2:ip,2:jbar)) * &
245     &    (ny+0.25*(nyt(3:imax,3:jp)+nyt(3:imax,2:jbar)+nyt(2:ip,3:jp)+nyt
      (2:ip,2:jbar))) &
246     &    - (vo(2:ip,2:jbar)-vo(1:ibar,2:jbar)) * &
247     &    (ny+0.25*(nyt(2:ip,3:jp)+nyt(2:ip,2:jbar)+nyt(1:ibar,3:jp)+nyt(1:
      ibar,2:jbar))) &
248     &     )
249
250 u(2:ibar,2:jp) = uo(2:ibar,2:jp) + delt*( &
```

```
251     & rdx*beta*(p(2:ibar,2:jp)-p(3:ip,2:jp)) &
252     & -0.25*rdx*((uo(2:ibar,2:jp)+uo(3:ip,2:jp))**2.0-(uo(2:ibar,2:jp)+uo
        (1:im,2:jp))**2.0) &
253     & -0.25*rdy*(&
254     &    (vo(2:ibar,2:jp )+vo(3:ip,2:jp ))*(uo(2:ibar,2:jp) +uo(2:ibar,3:
        jmax))- &
255     &    (vo(2:ibar,1:jbar)+vo(3:ip,1:jbar))*(uo(2:ibar,1:jbar)+uo(2:ibar
        ,2:jp )) )&
256     & + visux + visuy                    &
257     &  )
258
259 v(2:ip,2:jbar) = vo(2:ip,2:jbar) + delt*( &
260     & rdy*beta*(p(2:ip,2:jbar)-p(2:ip,3:jp)) &
261     & -0.25*rdy*((vo(2:ip,2:jbar)+vo(2:ip,3:jp))**2.0-(vo(2:ip,1:jm)+vo
        (2:ip,2:jbar))**2.0) &
262     & -0.25*rdx*(&
263     &    (uo(2:ip,2:jbar) +uo(2:ip,3:jp) )*(vo(2:ip,2:jbar) +vo(3:imax,2:
        jbar))- &
264     &    (uo(1:ibar,2:jbar)+uo(1:ibar,3:jp))*(vo(1:ibar,2:jbar)+vo(2:ip,2:
        jbar )) )&
265     & + visvy + visvx                    &
266     &  )
267
268 u(1:istm,1:jstm) = 0.0
269 v(1:istm,1:jstm) = 0.0
270
271 RETURN
272 END SUBROUTINE
273 !------------------------------------------------------------
274 SUBROUTINE calvel
275 !     purpose:
276 !     to update the tentative velocity field to the correct field
277 USE field
278 IMPLICIT NONE
279
280 u(2:ibar,2:jp) = u(2:ibar,2:jp) + delt*rdx*(dp(2:ibar,2:jp)-dp(3:ip,2:jp
        ))
281 v(2:ip,2:jbar) = v(2:ip,2:jbar) + delt*rdy*(dp(2:ip,2:jbar)-dp(2:ip,3:jp
        ))
282
283 RETURN
284 END SUBROUTINE
285 !------------------------------------------------------------
286 SUBROUTINE turbitr
287 USE field
288 USE turbf
289 IMPLICIT NONE
290 REAL(KIND=2),DIMENSION(ibar,jbar)   :: prod,dudy,dudx,dvdx,dvdy,viskx,
        visky,visox,visoy,velk,velom,dkdx,dkdy,domdx,domdy
291
292
293 dudx = (u(2:ip,2:jp)-u(1:ibar,2:jp))*rdx
294 dudy = ((u(2:ip,3:jmax)+u(1:ibar,3:jmax))-(u(2:ip,1:jbar)+u(1:ibar,1:
        jbar)))*0.5*(0.5*rdy)
295
```

```
296 dvdx = ((v(3:imax,2:jp)+v(3:imax,1:jbar))-(v(1:ibar,2:jp)+v(1:ibar,1:
      jbar)))*0.5*(0.5*rdx)
297 dvdy = (v(2:ip,2:jp)-v(2:ip,1:jbar))*rdy
298
299 dkdx = (k(3:imax,2:jp)-k(1:ibar,2:jp))*0.5*rdx
300 dkdy = (k(2:ip,3:jmax)-k(2:ip,1:jbar))*0.5*rdy
301
302 domdx = (om(3:imax,2:jp)-om(1:ibar,2:jp))*0.5*rdx
303 domdy = (om(2:ip,3:jmax)-om(2:ip,1:jbar))*0.5*rdy
304
305 velk = dudx*k(2:ip,2:jp) + dkdx*(u(2:ip,2:jp)+u(1:ibar,2:jp))*0.5 &
306   + dvdy*k(2:ip,2:jp) + dkdy*(v(2:ip,2:jp)+v(2:ip,1:jbar))*0.5
307
308 velom = dudx*om(2:ip,2:jp) + domdx*(u(2:ip,2:jp)+u(1:ibar,2:jp))*0.5 &
309   + dvdy*om(2:ip,2:jp) + domdy*(v(2:ip,2:jp)+v(2:ip,1:jbar))*0.5
310
311 prod =    &
312   & dudx*(nyt(2:ip,2:jp)*(2.0*dudx-(2.0/3.0)*(dudx+dvdy))-(2.0/3.0)*k(2:
      ip,2:jp)) +& !    11
313   & dvdy*(nyt(2:ip,2:jp)*(2.0*dvdy-(2.0/3.0)*(dudx+dvdy))-(2.0/3.0)*k(2:
      ip,2:jp)) +& !    22
314   & dvdx*nyt(2:ip,2:jp)*(dvdx+dudy) + & !    21
315   & dudy*nyt(2:ip,2:jp)*(dudy+dvdx)       !    12
316
317 viskx =   &
318   & ny*rdxx*(k(3:imax,2:jp)+k(1:ibar,2:jp)-2.0*k(2:ip,2:jp))    &   !ny
      ddk/ddx
319   & + 0.5*rdxx*sigk*(                &           !d/dx(sig nyt(dk/dx)) (
320   &   (nyt(3:imax,2:jp)+nyt(2:ip,2:jp))*(k(3:imax,2:jp)-k(2:ip,2:jp)) &
      ! ---
321   &  -(nyt(2:ip,2:jp)+nyt(1:ibar,2:jp))*(k(2:ip,2:jp)-k(1:ibar,2:jp)) &
      ! ---
322   & )                ! )
323
324 visky =   &
325   & ny*rdyy*(k(2:ip,3:jmax)+k(2:ip,1:jbar)-2.0*k(2:ip,2:jp))    &   !ny
      ddk/ddy
326   & + 0.5*rdyy*sigk*(                &           !d/dy(sig nyt(dk/dy)) (
327   &   (nyt(2:ip,3:jmax)+nyt(2:ip,2:jp))*(k(2:ip,3:jmax)-k(2:ip,2:jp)) &
      ! ---
328   &  -(nyt(2:ip,2:jp)+nyt(2:ip,1:jbar))*(k(2:ip,2:jp)-k(2:ip,1:jbar)) &
      ! ---
329   & )                ! )
330
331 visox =   &
332   & ny*rdxx*(om(3:imax,2:jp)+om(1:ibar,2:jp)-2.0*om(2:ip,2:jp))   & !ny
      ddom/ddx
333   & + 0.5*rdxx*sigom*(               &           !d/dx(sig nyt(dom/dx)) (
334   &   (nyt(3:imax,2:jp)+nyt(2:ip,2:jp))*(om(3:imax,2:jp)-om(2:ip,2:jp))
      &! ---
335   &  -(nyt(2:ip,2:jp)+nyt(1:ibar,2:jp))*(om(2:ip,2:jp)-om(1:ibar,2:jp))
      &! ---
336   & )                ! )
337
338 visoy =   &
```

```
339   & ny*rdyy*(om(2:ip,3:jmax)+om(2:ip,1:jbar)-2.0*om(2:ip,2:jp))    & !ny
       ddom/ddy
340   & + 0.5*rdyy*sigom*(                &            !d/dy(sig nyt(dom/dy)) (
341   &    (nyt(2:ip,3:jmax)+nyt(2:ip,2:jp))*(om(2:ip,3:jmax)-om(2:ip,2:jp))
       &! ---
342   &   -(nyt(2:ip,2:jp)+nyt(2:ip,1:jbar))*(om(2:ip,2:jp)-om(2:ip,1:jbar))
       &! ---
343   & )                 ! )
344
345 k(2:ip,2:jp) = k(2:ip,2:jp) + delt*(     &
346   & - velk            &
347   & + prod            &
348   & -bettas*om(2:ip,2:jp)*k(2:ip,2:jp)     &
349   & + viskx + visky      )
350
351 om(2:ip,2:jp) = om(2:ip,2:jp) + delt*(     &
352   & - velom             &
353   & + (gamm/(nyt(2:ip,2:jp)))*prod     &
354   & - betta*om(2:ip,2:jp)**2.0       &
355   & + visox + visoy    )
356
357
358 !Outlets
359 k(imax,2:jp) = k(ip,2:jp)
360 om(imax,2:jp) = om(ip,2:jp)
361
362 !Symmetry wall
363 k(2:ip,jmax)  = k(2:ip,jp)
364 om(2:ip,jmax) = om(2:ip,jp)
365
366 !Vertical "stepwall"
367 k(ist,2:jst) = -k(istp,2:jst)
368 om(ist,2:jst) = -om(istp,2:jst) + 60.0*ny/(betta*(delx/2.0)**2.0)
369
370 !Horizontal "stepwall"
371 k(2:istm,jst) = -k(2:istm,jstp)
372 om(2:istm,jst) = -om(2:istm,jstp) + 60.0*ny/(betta*(dely/2.0)**2.0)
373
374 !Lower wall
375 k(istp:ip,1)  = -k(istp:ip,2)
376 om(istp:ip,1)  = -om(istp:ip,2) + 60.0*ny/(betta*(dely/2.0)**2.0)
377
378 ! nyt
379 nyt(2:ist,jstp:jp)  = k(2:ist,jstp:jp)/(om(2:ist,jstp:jp))  !Part1
380 nyt(istp:ip,2:jp)  = k(istp:ip,2:jp)/(om(istp:ip,2:jp))   !Part2
381
382 nyt(2:ip,jmax) = nyt(2:ip,jp)   !Symmetry wall
383 nyt(2:istm,jst) = -nyt(2:istm,jstp) !Horizontal "stepwall"
384 nyt(istp:ip,1)   = -nyt(istp:ip,2)  !Lower wall
385 nyt(ist,2:jst)  = -nyt(istp,2:jst)  !Vertical "stepwall"
386 nyt(imax,2:jp) = nyt(ip,2:jp)     !Outlet
387
388 k(1:istm,1:jstm) = 0.0
389 om(1:istm,1:jstm) = 1.0
390
391 END SUBROUTINE
```

```
392 !-----------------------------------------------
393 SUBROUTINE congrad_nag_pres
394 !     purpose:
395 !     solve the pressure equation
396 USE field
397 USE nag_values
398 IMPLICIT NONE
399 INTEGER       :: nm,np
400
401 nm = (ip-ist)*(jstm)
402 np = nm +1
403 ifail = 0
404 CALL f11bdf(method,precon,norm,weigth,iterm,n,m,tol,maxitn,anorm,sigmax,
    monit,lwreq,work,lwork,ifail)
405 lwreq = lwork
406
407 b(1:nm) = RESHAPE(&
408    ((u(istp:ip,2:jst)-u(ist:ibar,2:jst))*rdx+((v(istp:ip,2:jst)-v(istp:
    ip,1:jstm))*rdy))&
409    ,(/nm/))
410 b(np:n) = RESHAPE(&
411    ((u(2:ip,jstp:jp)-u(1:ibar,jstp:jp))*rdx+((v(2:ip,jstp:jp)-v(2:ip,jst
    :jbar))*rdy))&
412    ,(/n-nm/))
413
414 irevcm = 0
415 ifail = 1
416
417 loop: DO
418    CALL f11bef(irevcm,x,b,wgt,work,lwreq,ifail)
419    IF (irevcm/=4) THEN
420       ifail1 = 0
421       SELECT CASE (irevcm)
422             CASE (-1)
423                CALL f11xaf('T',n,nnz,a,irow,icol,'N',x,b,ifail1)
424             CASE (1)
425                CALL f11xaf('N',n,nnz,a,irow,icol,'N',x,b,ifail1)
426             CASE (2)
427                CALL f11dbf('N',n,a,la,irow,icol,ipivp,ipivq,istr, &
428                     idiag,'N',x,b,ifail1)
429          END SELECT
430          IF (ifail1/=0) THEN
431             irevcm = 6
432          END IF
433    ELSE IF (ifail/=0) THEN
434       EXIT loop
435    ELSE
436       EXIT loop
437    END IF
438 END DO loop
439
440 dp(istp:ip,2:jst) = RESHAPE(x(1:nm),(/(ip-istp),(jstm)/))
441 dp(2:ip,jstp:jp) = RESHAPE(x(np:n),(/ibar,(jp-jstp)/))
442
443 ! BC at ends
444 dp(1,2:jp)  =  dp(2,2:jp)
```

```
445 dp(imax,2:jp) =  0.0!p(2,2:jp)
446
447 !BC at top/bottom
448 dp(2:ip,jmax) = dp(2:ip,jp)
449
450 dp(2:istm,jst) = dp(2:istm,jstp)! Horizontal step
451 dp(istp:ip,1)  = dp(istp:ip,2)  ! Lower wall
452 dp(ist,2:jst)  = dp(istp,2:jst) ! Vertical step
453
454 po = p
455 p = po*beta + dp
456
457 ! BC at ends
458 p(1,2:jp)   =  p(2,2:jp)
459 p(imax,2:jp)  =  0.0
460
461 !BC at top/bottom
462 p(2:ip,jmax) = p(2:ip,jp)
463
464 p(2:istm,jst) = p(2:istm,jstp)! Horizontal step
465 p(istp:ip,1)  = p(istp:ip,2)  ! Lower wall
466 p(ist,2:jst)  = p(istp,2:jst) ! Vertical step
467
468
469 RETURN
470 END SUBROUTINE congrad_nag_pres
471 !-----------------------------------------------
472 SUBROUTINE output
473 USE field
474 USE turbf
475 IMPLICIT NONE
476 INTEGER,PARAMETER   :: uout=13,vout=14,pout=15,kout=16,omout=17,nytout
      =18,tout=19,u2out=20
477 REAL(KIND=2)    :: totaltime
478 CHARACTER(LEN=12) :: frmt
479
480 !'hours spent, minutes spent, seconds spent, millisecnds spent'
481 !timespent(4),timespent(5),timespent(6),timespent(7)
482 totaltime = timespent(4)*60 + timespent(5)*60 + timespent(6)! +
      timespent(7)/1000
483
484 WRITE(frmt,'(A1,I3,A8)') '(',imax,'E15.5E2)'
485 PRINT*,'frmt = ',frmt
486
487 OPEN (UNIT=tout,FILE='time.dat',STATUS='unknown',FORM='formatted')
488 WRITE(tout,'(F15.5)') totaltime
489 CLOSE(tout)
490
491 OPEN (UNIT=uout,FILE='u.dat',STATUS='unknown',FORM='formatted')
492 WRITE(uout,frmt) u(:,:)
493 CLOSE(uout)
494
495 OPEN (UNIT=vout,FILE='v.dat',STATUS='unknown',FORM='formatted')
496 WRITE(vout,frmt) v(:,:)
497 CLOSE(vout)
498
```

```
499 OPEN (UNIT=pout,FILE='p.dat',STATUS='unknown',FORM='formatted')
500 WRITE(pout,frmt) p(:,:)
501 CLOSE(pout)
502
503 OPEN (UNIT=kout,FILE='k.dat',STATUS='unknown',FORM='formatted')
504 WRITE(kout,frmt) k(:,:)
505 CLOSE(kout)
506
507 OPEN (UNIT=omout,FILE='om.dat',STATUS='unknown',FORM='formatted')
508 WRITE(omout,frmt) om(:,:)
509 CLOSE(omout)
510
511 OPEN (UNIT=nytout,FILE='nyt.dat',STATUS='unknown',FORM='formatted')
512 WRITE(nytout,frmt) nyt(:,:)
513 CLOSE(nytout)
514
515 OPEN (UNIT=u2out,FILE='u2.dat',STATUS='unknown',FORM='formatted')
516 WRITE(u2out,'(E15.5)') u(51,:)
517 CLOSE(u2out)
518
519 END SUBROUTINE output
520 !
    ------------------------------------------------------------------------

521 SUBROUTINE inputfromlastitr
522 USE field
523 USE turbf
524 IMPLICIT NONE
525 INTEGER,PARAMETER    :: uin=99,vin=98,pin=97,kin=96,omin=95,nytin=94
526 CHARACTER(LEN=12) :: frmt
527
528 WRITE(frmt,'(A1,I3,A8)') '(',imax,'E15.5E2)'
529 PRINT*,'frmt = ',frmt
530
531 OPEN (UNIT=uin,FILE='u.dat',STATUS='unknown',FORM='formatted')
532 READ(uin,frmt) u(:,:)
533 CLOSE(uin)
534
535 OPEN (UNIT=vin,FILE='v.dat',STATUS='unknown',FORM='formatted')
536 READ(vin,frmt) v(:,:)
537 CLOSE(vin)
538
539 OPEN (UNIT=pin,FILE='p.dat',STATUS='unknown',FORM='formatted')
540 READ(pin,frmt) p(:,:)
541 CLOSE(pin)
542
543 OPEN (UNIT=kin,FILE='k.dat',STATUS='unknown',FORM='formatted')
544 READ(kin,frmt) k(:,:)
545 CLOSE(kin)
546
547 OPEN (UNIT=omin,FILE='om.dat',STATUS='unknown',FORM='formatted')
548 READ(omin,frmt) om(:,:)
549 CLOSE(omin)
550
551 OPEN (UNIT=nytin,FILE='nyt.dat',STATUS='unknown',FORM='formatted')
552 READ(nytin,frmt) nyt(:,:)
```

```
553 CLOSE(nytin)
554
555 WRITE(*,*) ' Loaded last iteration '
556
557 END SUBROUTINE inputfromlastitr
558 !
    ------------------------------------------------------------------------

559 SUBROUTINE inputfrom1D
560 USE field
561 USE turbf
562 IMPLICIT NONE
563 INTEGER,PARAMETER     :: uin1=89,kin1=88,omin1=87,nytin1=86,uin2=85,kin2
    =84,omin2=83,nytin2=82
564 INTEGER        :: ij
565 REAL(KIND=2),DIMENSION(250)    :: u250,k250,om250,nyt250
566 REAL(KIND=2),DIMENSION(300) :: u300,k300,om300,nyt300
567 CHARACTER(LEN=12)    :: frmt='(E15.5)'
568
569 !------- 250
570 OPEN (UNIT=uin1,FILE='u250.inp',STATUS='unknown',FORM='formatted')
571 READ(uin1,frmt) u250(:)
572 CLOSE(uin1)
573
574 OPEN (UNIT=kin1,FILE='k250.inp',STATUS='unknown',FORM='formatted')
575 READ(kin1,frmt) k250(:)
576 CLOSE(kin1)
577
578 OPEN (UNIT=omin1,FILE='om250.inp',STATUS='unknown',FORM='formatted')
579 READ(omin1,frmt) om250(:)
580 CLOSE(omin1)
581
582 OPEN (UNIT=nytin1,FILE='nyt250.inp',STATUS='unknown',FORM='formatted')
583 READ(nytin1,frmt) nyt250(:)
584 CLOSE(nytin1)
585
586 DO ij = 1,ist
587   u(ij,jstp:jbar)    = u250(:)
588   u(ij,jp) =    u(ij,jbar)
589   k(ij,jstp:jbar)    = k250(:)
590   k(ij,jp) =    k(ij,jbar)
591   om(ij,jstp:jbar)   = om250(:)
592   om(ij,jp) =    om(ij,jbar)
593   nyt(ij,jstp:jbar) = nyt250(:)
594   nyt(ij,jp) =    nyt(ij,jbar)
595 END DO
596
597 !--------  300
598 OPEN (UNIT=uin2,FILE='u300.inp',STATUS='unknown',FORM='formatted')
599 READ(uin2,frmt) u300(:)
600 CLOSE(uin2)
601
602 OPEN (UNIT=kin2,FILE='k300.inp',STATUS='unknown',FORM='formatted')!!!
    This is bad
603 READ(kin2,frmt) k300(:)
604 CLOSE(kin2)
```

```
605
606 OPEN (UNIT=omin2,FILE='om300.inp',STATUS='unknown',FORM='formatted')!!
      This is bad
607 READ(omin2,frmt) om300(:)
608 CLOSE(omin2)
609
610 OPEN (UNIT=nytin2,FILE='nyt300.inp',STATUS='unknown',FORM='formatted')!!
       This is probably bad
611 READ(nytin2,frmt) nyt300(:)
612 CLOSE(nytin2)
613
614 DO ij = istp,imax
615   u(ij,2:jp)   = u300(:)
616   k(ij,2:jp)   = k300(:)
617   om(ij,2:jp)  = om300(:)
618   nyt(ij,2:jp) = nyt300(:)
619 END DO
620
621 !------- BC
622 u(ip,1:jmax) = u(ibar,1:jmax) ! Outlet
623 v(imax,1:jmax) = v(ip,2:jmax) ! Outlet
624
625 u(ist,2:jst) = 0.0 ! Vertical "stepwall"
626 v(ist,1:jstm) = -v(istp,1:jstm) ! Vertical "stepwall"
627
628 v(1:imax,jp) = 0.0 ! Symmetry wall
629 u(1:ip,jmax) = u(1:ip,jp) ! Symmetry wall
630
631 v(istp:imax,1) = 0.0 ! Lower wall
632 u(istp:ip,1)   = -u(istp:ip,2) !lower wall
633
634 u(2:istm,jst) = -u(2:istm,jstp) ! Horizontal "stepwall"
635 v(2:ist,jst) = 0.0    ! Horizontal "stepwall"
636
637 !Outlets
638 k(imax,2:jp) = k(ip,2:jp)
639 om(imax,2:jp) = om(ip,2:jp)
640
641 !Symmetry wall
642 k(2:ip,jmax)  = k(2:ip,jp)
643 om(2:ip,jmax) = om(2:ip,jp)
644
645 !Vertical "stepwall"
646 k(ist,2:jst) = -k(istp,2:jst)
647 om(ist,2:jst) = -om(istp,2:jst) + 60.0*ny/(betta*(delx/2.0)**2.0)
648
649 !Horizontal "stepwall"
650 k(2:istm,jst) = -k(2:istm,jstp)
651 om(2:istm,jst) = -om(2:istm,jstp) + 60.0*ny/(betta*(dely/2.0)**2.0)
652
653 !Lower wall
654 k(istp:ip,1)  = -k(istp:ip,2)
655 om(istp:ip,1)  = -om(istp:ip,2) + 60.0*ny/(betta*(dely/2.0)**2.0)
656
657 ! nyt
658 nyt(2:ist,jstp:jp)  = k(2:ist,jstp:jp)/(om(2:ist,jstp:jp))!Part1
```

```
659 nyt(istp:ip,2:jp)  = k(istp:ip,2:jp)/(om(istp:ip,2:jp)) !Part2
660
661 nyt(2:ip,jmax) = nyt(2:ip,jp)!Symmetry wall
662 nyt(2:istm,jst) = -nyt(2:istm,jstp) !Horizontal "stepwall"
663 nyt(istp:ip,1)   = -nyt(istp:ip,2)  !Lower wall
664 nyt(ist,2:jst)  = -nyt(istp,2:jst)  !Vertical "stepwall"
665 nyt(imax,2:jp) = nyt(ip,2:jp) ! Outlet
666
667
668 WRITE(*,*) ' Read input from 1D solution '
669
670 END SUBROUTINE inputfrom1D
671 !----------------------------------------
672 SUBROUTINE precon_pressure
673 !     purpose:
674 !     set values connected to the NAG library and precondition the
      matrix
675 USE field
676 USE nag_values
677 IMPLICIT NONE
678 INTEGER   :: row_leng, row_leng_step, prev_leng, runde
679 REAL(KIND=2)    :: ap,apc,apv,aph
680
681 ap=-2.0*(rdxx+rdyy)
682 apc=-(rdxx+rdyy)
683 apv=-(rdxx+2.0*rdyy)
684 aph=-(2.0*rdxx+rdyy)
685 ! Parameters for f11daf:
686 n = (istm)*(jbar-jstm)+(ibar-istm)*(jbar)
687 nnz = 5*n - 2*(ibar+jbar)
688 la = 3*nnz
689 ALLOCATE(a(la),irow(la),icol(la))
690 a = 0.0
691 !======== Create a ============
692 row_leng = 5*(ibar-2) + 8
693 row_leng_step = 5*(ibar-istm-2) + 8
694 !First row
695 a(1) = apc
696 irow(1) = 1
697 icol(1) = 1
698 a(2)     = rdxx
699 irow(2)  = 1
700 icol(2)  = 2
701 a(3)     = rdyy
702 irow(3)  = 1
703 icol(3)  = 1 + (ibar-istm)
704
705 runde = 1
706 DO i = 5,(ibar-istm-2)*4+3,4
707    runde  = runde + 1
708    a(i)       = aph
709    irow(i)    = runde
710    icol(i)    = runde
711    a(i-1)     = rdxx
712    irow(i-1)  = irow(i)
713    icol(i-1)  = icol(i)-1
```

```
714    a(i+1)      = rdxx
715    irow(i+1)   = irow(i)
716    icol(i+1)   = icol(i)+1
717    a(i+2)      = rdyy
718    irow(i+2)   = irow(i)
719    icol(i+2)   = icol(i) + (ibar-istm)
720 END DO
721 i=4*(ibar-2-istm)+3+2
722 a(i)     = aph
723 irow(i) = (ibar-istm)
724 icol(i) = (ibar-istm)
725 a(i-1)     = rdxx
726 irow(i-1) = irow(i)
727 icol(i-1) = icol(i)-1
728 a(i+1)       = rdyy
729 irow(i+1) = irow(i)
730 icol(i+1) = icol(i) + (ibar-istm)
731
732 !Row 2 to stepheight-1
733 DO i = 2,(jstm-1)
734    runde = 0
735    prev_leng = row_leng_step*(i-2)+4*(ibar-istm-2)+6
736
737    a(2+prev_leng)     = apv
738    irow(2+prev_leng) = (ibar-istm)*(i-1)+1
739    icol(2+prev_leng) = (ibar-istm)*(i-1)+1
740
741    a(1+prev_leng)     = rdyy
742    irow(1+prev_leng) = irow(prev_leng+2)
743    icol(1+prev_leng) = icol(prev_leng+2) - (ibar-istm)
744
745    a(3+prev_leng)     = rdxx
746    irow(3+prev_leng) = irow(prev_leng+2)
747    icol(3+prev_leng) = irow(prev_leng+2) + 1
748
749    a(4+prev_leng)     = rdyy
750    irow(4+prev_leng) = irow(prev_leng+2)
751    icol(4+prev_leng) = icol(prev_leng+2) + (ibar-istm)
752
753    DO j = 7,(ibar-2-istm)*5+3,5
754       runde = runde + 1
755       a(prev_leng+j)     = ap
756       irow(prev_leng+j) = runde + (ibar-istm)*(i-1)+1
757       icol(prev_leng+j) = runde + (ibar-istm)*(i-1)+1
758
759       a(prev_leng+j-1)     = rdxx
760       irow(prev_leng+j-1) = irow(prev_leng+j)
761       icol(prev_leng+j-1) = icol(prev_leng+j)-1
762
763       a(prev_leng+j-2)     = rdyy
764       irow(prev_leng+j-2) = irow(prev_leng+j)
765       icol(prev_leng+j-2) = icol(prev_leng+j) - (ibar-istm)
766
767       a(prev_leng+j+1)     = rdxx
768       irow(prev_leng+j+1) = irow(prev_leng+j)
769       icol(prev_leng+j+1) = icol(prev_leng+j)+1
```

```
770
771       a(prev_leng+j+2)     = rdyy
772       irow(prev_leng+j+2) = irow(prev_leng+j)
773       icol(prev_leng+j+2) = icol(prev_leng+j) + (ibar-istm)
774     END DO
775     a(prev_leng+row_leng_step-1)     = ap
776     irow(prev_leng+row_leng_step-1) = runde + (ibar-istm)*(i-1)+2
777     icol(prev_leng+row_leng_step-1) = runde + (ibar-istm)*(i-1)+2
778
779     a(prev_leng+row_leng_step)       = rdyy
780     irow(prev_leng+row_leng_step)    = irow(prev_leng+row_leng_step-1)
781     icol(prev_leng+row_leng_step)    = icol(prev_leng+row_leng_step-1)+(
      ibar-istm)
782
783     a(prev_leng+row_leng_step-2)     = rdxx
784     irow(prev_leng+row_leng_step-2) = irow(prev_leng+row_leng_step-1)
785     icol(prev_leng+row_leng_step-2) = icol(prev_leng+row_leng_step-1)-1
786
787     a(prev_leng+row_leng_step-3)     = rdyy
788     irow(prev_leng+row_leng_step-3) = irow(prev_leng+row_leng_step-1)
789     icol(prev_leng+row_leng_step-3) = icol(prev_leng+row_leng_step-1) -(
      ibar-istm)
790 END DO
791
792 ! row stepheight
793 prev_leng = row_leng_step*(jstm-1)+4*(ibar-istm-2)+6
794
795 a(2+prev_leng)     = apv
796 irow(2+prev_leng) = (ibar-istm)*(jstm-1)+1
797 icol(2+prev_leng) = (ibar-istm)*(jstm-1)+1
798
799 a(1+prev_leng)     = rdyy
800 irow(1+prev_leng) = irow(prev_leng+2)
801 icol(1+prev_leng) = icol(prev_leng+2) - (ibar-istm)
802
803 a(3+prev_leng)     = rdxx
804 irow(3+prev_leng) = irow(prev_leng+2)
805 icol(3+prev_leng) = irow(prev_leng+2) + 1
806
807 a(4+prev_leng)     = rdyy
808 irow(4+prev_leng) = irow(prev_leng+2)
809 icol(4+prev_leng) = icol(prev_leng+2) + (ibar)
810
811 runde = 0
812 DO j = 7,(ibar-2-istm)*5+3,5
813   runde = runde + 1
814   a(prev_leng+j)     = ap
815   irow(prev_leng+j) = runde + (ibar-istm)*(jstm-1)+1
816   icol(prev_leng+j) = runde + (ibar-istm)*(jstm-1)+1
817
818   a(prev_leng+j-1)     = rdxx
819   irow(prev_leng+j-1) = irow(prev_leng+j)
820   icol(prev_leng+j-1) = icol(prev_leng+j)-1
821
822   a(prev_leng+j-2)     = rdyy
823   irow(prev_leng+j-2) = irow(prev_leng+j)
```

```
824   icol(prev_leng+j-2) = icol(prev_leng+j) - (ibar-istm)
825
826   a(prev_leng+j+1)    = rdxx
827   irow(prev_leng+j+1) = irow(prev_leng+j)
828   icol(prev_leng+j+1) = icol(prev_leng+j)+1
829
830   a(prev_leng+j+2)    = rdyy
831   irow(prev_leng+j+2) = irow(prev_leng+j)
832   icol(prev_leng+j+2) = icol(prev_leng+j) + ibar
833 END DO
834
835 a(prev_leng+row_leng_step-1)    = ap
836 irow(prev_leng+row_leng_step-1) = runde + (ibar-istm)*(jstm-1)+2
837 icol(prev_leng+row_leng_step-1) = runde + (ibar-istm)*(jstm-1)+2
838
839 a(prev_leng+row_leng_step)      = rdyy
840 irow(prev_leng+row_leng_step)   = irow(prev_leng+row_leng_step-1)
841 icol(prev_leng+row_leng_step)   = icol(prev_leng+row_leng_step-1) + ibar
842
843 a(prev_leng+row_leng_step-2)    = rdxx
844 irow(prev_leng+row_leng_step-2) = irow(prev_leng+row_leng_step-1)
845 icol(prev_leng+row_leng_step-2) = icol(prev_leng+row_leng_step-1)-1
846
847 a(prev_leng+row_leng_step-3)    = rdyy
848 irow(prev_leng+row_leng_step-3) = irow(prev_leng+row_leng_step-1)
849 icol(prev_leng+row_leng_step-3) = icol(prev_leng+row_leng_step-1) -(ibar
      -istm)
850
851 ! row stepheight
852 prev_leng = row_leng_step*(jstm-2)+4*(ibar-istm-2)+6
853
854 a(2+prev_leng)    = apv
855 irow(2+prev_leng) = (ibar-istm)*(jstm-1)+1
856 icol(2+prev_leng) = (ibar-istm)*(jstm-1)+1
857
858 a(1+prev_leng)    = rdyy
859 irow(1+prev_leng) = irow(prev_leng+2)
860 icol(1+prev_leng) = icol(prev_leng+2) - (ibar-istm)
861
862 a(3+prev_leng)    = rdxx
863 irow(3+prev_leng) = irow(prev_leng+2)
864 icol(3+prev_leng) = irow(prev_leng+2) + 1
865
866 a(4+prev_leng)    = rdyy
867 irow(4+prev_leng) = irow(prev_leng+2)
868 icol(4+prev_leng) = icol(prev_leng+2) + (ibar)
869
870 runde = 0
871 DO j = 7,(ibar-2-istm)*5+3,5
872   runde = runde + 1
873   a(prev_leng+j)    = ap
874   irow(prev_leng+j) = runde + (ibar-istm)*(jstm-1)+1
875   icol(prev_leng+j) = runde + (ibar-istm)*(jstm-1)+1
876
877   a(prev_leng+j-1)    = rdxx
878   irow(prev_leng+j-1) = irow(prev_leng+j)
```

```
879   icol(prev_leng+j-1) = icol(prev_leng+j)-1
880
881   a(prev_leng+j-2)    = rdyy
882   irow(prev_leng+j-2) = irow(prev_leng+j)
883   icol(prev_leng+j-2) = icol(prev_leng+j) - (ibar-istm)
884
885   a(prev_leng+j+1)    = rdxx
886   irow(prev_leng+j+1) = irow(prev_leng+j)
887   icol(prev_leng+j+1) = icol(prev_leng+j)+1
888
889   a(prev_leng+j+2)    = rdyy
890   irow(prev_leng+j+2) = irow(prev_leng+j)
891   icol(prev_leng+j+2) = icol(prev_leng+j) + ibar
892 END DO
893
894 a(prev_leng+row_leng_step-1)    = ap
895 irow(prev_leng+row_leng_step-1) = runde + (ibar-istm)*(jstm-1)+2
896 icol(prev_leng+row_leng_step-1) = runde + (ibar-istm)*(jstm-1)+2
897
898 a(prev_leng+row_leng_step)      = rdyy
899 irow(prev_leng+row_leng_step)   = irow(prev_leng+row_leng_step-1)
900 icol(prev_leng+row_leng_step)   = icol(prev_leng+row_leng_step-1) + ibar
901
902 a(prev_leng+row_leng_step-2)    = rdxx
903 irow(prev_leng+row_leng_step-2) = irow(prev_leng+row_leng_step-1)
904 icol(prev_leng+row_leng_step-2) = icol(prev_leng+row_leng_step-1)-1
905
906 a(prev_leng+row_leng_step-3)    = rdyy
907 irow(prev_leng+row_leng_step-3) = irow(prev_leng+row_leng_step-1)
908 icol(prev_leng+row_leng_step-3) = icol(prev_leng+row_leng_step-1) -(ibar
      -istm)
909
910 ! row stepheight+1
911 prev_leng = row_leng_step*(jstm-1)+4*(ibar-istm-2)+6
912
913 a(prev_leng+1)    = apc
914 irow(prev_leng+1) = (jstm)*(ibar-istm) + 1
915 icol(prev_leng+1) = (jstm)*(ibar-istm) + 1
916
917 a(prev_leng+2)     = rdxx
918 irow(prev_leng+2)  = irow(prev_leng+1)
919 icol(prev_leng+2)  = icol(prev_leng+1)+1
920
921 a(prev_leng+3)     = rdyy
922 irow(prev_leng+3)  = irow(prev_leng+1)
923 icol(prev_leng+3)  = icol(prev_leng+1) + ibar
924
925 runde = 1
926 DO i = 5,(istm-1)*4+3,4
927    runde  = runde + 1
928    a(prev_leng+i)       = aph
929    irow(prev_leng+i)    = (jstm)*(ibar-istm) + runde
930    icol(prev_leng+i)    = (jstm)*(ibar-istm) + runde
931
932    a(prev_leng+i-1)     = rdxx
933    irow(prev_leng+i-1)  = irow(prev_leng+i)
```

```
934     icol(prev_leng+i-1)   = icol(prev_leng+i)-1
935
936     a(prev_leng+i+1)      = rdxx
937     irow(prev_leng+i+1)   = irow(prev_leng+i)
938     icol(prev_leng+i+1)   = icol(prev_leng+i)+1
939
940     a(prev_leng+i+2)      = rdyy
941     irow(prev_leng+i+2)   = irow(prev_leng+i)
942     icol(prev_leng+i+2)   = icol(prev_leng+i) + ibar
943 END DO
944 DO i = 3+(istm-1)*4+3,3+(istm-1)*4+(ibar-istm-2)*5+3,5
945   runde = runde + 1
946   a(prev_leng+i)     = ap
947   irow(prev_leng+i) = (jstm)*(ibar-istm) + runde
948   icol(prev_leng+i) = (jstm)*(ibar-istm) + runde
949
950   a(prev_leng+i-1)     = rdxx
951   irow(prev_leng+i-1) = irow(prev_leng+i)
952   icol(prev_leng+i-1) = icol(prev_leng+i)-1
953
954   a(prev_leng+i-2)     = rdyy
955   irow(prev_leng+i-2) = irow(prev_leng+i)
956   icol(prev_leng+i-2) = icol(prev_leng+i) - ibar
957
958   a(prev_leng+i+1)     = rdxx
959   irow(prev_leng+i+1) = irow(prev_leng+i)
960   icol(prev_leng+i+1) = icol(prev_leng+i)+1
961
962   a(prev_leng+i+2)     = rdyy
963   irow(prev_leng+i+2) = irow(prev_leng+i)
964   icol(prev_leng+i+2) = icol(prev_leng+i) + ibar
965 END DO
966 prev_leng = (ibar-istm-2)*4 + 6 + row_leng_step*(jstm-1) + (istm-1)*4 +
      3 + (ibar-istm-1)*5
967 a(prev_leng+3)     = ap
968 irow(prev_leng+3) = runde + (jstm)*(ibar-istm) + 1
969 icol(prev_leng+3) = runde + (jstm)*(ibar-istm) + 1
970
971 a(prev_leng+4)     = rdyy
972 irow(prev_leng+4) = irow(prev_leng+3)
973 icol(prev_leng+4) = icol(prev_leng+3) + ibar
974
975 a(prev_leng+2)     = rdxx
976 irow(prev_leng+2) = irow(prev_leng+3)
977 icol(prev_leng+2) = icol(prev_leng+3) - 1
978
979 a(prev_leng+1)     = rdyy
980 irow(prev_leng+1) = irow(prev_leng+3)
981 icol(prev_leng+1) = icol(prev_leng+3) - ibar
982 ! row stepheight+2 to row jbar-1
983 DO i = (jstm+2),(jbar-1)
984     runde = 0
985 !    prev_leng = (i-jstm-2)*row_leng + ((ibar-istm-2)*4 + 6) + (
      row_leng_step*(jstm-1)) + ((istm-1)*4 + 7 + (ibar-istm-1)*5)
986     prev_leng = (ibar-istm-2)*4 + 6 + row_leng_step*(jstm-1) + (istm-1)*4
       + 7 + (ibar-istm-1)*5 + (i-(jstm+2))*row_leng
```

```
987    a(2+prev_leng)    = apv
988    irow(2+prev_leng) = jstm*(ibar-istm) + ibar + ibar*(i-jstm-2) + 1
989    icol(2+prev_leng) = irow(2+prev_leng)
990
991    a(1+prev_leng)    = rdyy
992    irow(1+prev_leng) = irow(prev_leng+2)
993    icol(1+prev_leng) = icol(prev_leng+2) - ibar
994
995    a(3+prev_leng)    = rdxx
996    irow(3+prev_leng) = irow(prev_leng+2)
997    icol(3+prev_leng) = irow(prev_leng+2) + 1
998
999    a(4+prev_leng)    = rdyy
1000   irow(4+prev_leng) = irow(prev_leng+2)
1001   icol(4+prev_leng) = icol(prev_leng+2) + ibar
1002
1003   DO j = 7,(ibar-2)*5+4,5
1004      runde = runde + 1
1005      a(prev_leng+j)    = ap
1006      irow(prev_leng+j) = runde + irow(prev_leng+2)
1007      icol(prev_leng+j) = irow(prev_leng+j)
1008
1009      a(prev_leng+j-1)    = rdxx
1010      irow(prev_leng+j-1) = irow(prev_leng+j)
1011      icol(prev_leng+j-1) = icol(prev_leng+j)-1
1012
1013      a(prev_leng+j-2)    = rdyy
1014      irow(prev_leng+j-2) = irow(prev_leng+j)
1015      icol(prev_leng+j-2) = icol(prev_leng+j) - ibar
1016
1017      a(prev_leng+j+1)    = rdxx
1018      irow(prev_leng+j+1) = irow(prev_leng+j)
1019      icol(prev_leng+j+1) = icol(prev_leng+j)+1
1020
1021      a(prev_leng+j+2)    = rdyy
1022      irow(prev_leng+j+2) = irow(prev_leng+j)
1023      icol(prev_leng+j+2) = icol(prev_leng+j) + ibar
1024   END DO
1025   a(prev_leng+row_leng-1)    = ap
1026   irow(prev_leng+row_leng-1) = runde + irow(2+prev_leng)+1
1027   icol(prev_leng+row_leng-1) = irow(prev_leng+row_leng-1)
1028
1029   a(prev_leng+row_leng)      = rdyy
1030   irow(prev_leng+row_leng)   = irow(prev_leng+row_leng-1)
1031   icol(prev_leng+row_leng)   = icol(prev_leng+row_leng-1) + ibar
1032
1033   a(prev_leng+row_leng-2)    = rdxx
1034   irow(prev_leng+row_leng-2) = irow(prev_leng+row_leng-1)
1035   icol(prev_leng+row_leng-2) = icol(prev_leng+row_leng-1)-1
1036
1037   a(prev_leng+row_leng-3)    = rdyy
1038   irow(prev_leng+row_leng-3) = irow(prev_leng+row_leng-1)
1039   icol(prev_leng+row_leng-3) = icol(prev_leng+row_leng-1) - ibar
1040 END DO
1041
1042 !row jbar, last row
```

```
1043 prev_leng = (jbar-(jstm+2))*row_leng + ((ibar-istm-2)*4 + 6) + (
       row_leng_step*(jstm-1)) + ((istm-1)*4 + 7 + (ibar-istm-1)*5)
1044
1045 a(prev_leng+2)     = apc
1046 irow(prev_leng+2) = (ibar-istm)*jstm + ibar*(jbar-jstm-1)+1
1047 icol(prev_leng+2) = irow(prev_leng+2)
1048
1049 a(prev_leng+1)     = rdyy
1050 irow(prev_leng+1) = irow(prev_leng+2)
1051 icol(prev_leng+1) = icol(prev_leng+2)-ibar
1052
1053 a(prev_leng+3)     = rdxx
1054 irow(prev_leng+3) = irow(prev_leng+2)
1055 icol(prev_leng+3) = icol(prev_leng+2) + 1
1056
1057 runde = 0
1058 DO i = 6,(ibar-2)*4+3,4
1059    runde = runde +1
1060    a(prev_leng+i)     = aph
1061    irow(prev_leng+i) = (ibar-istm)*jstm + ibar*(jbar-jstm-1)+1+runde
1062    icol(prev_leng+i) = irow(prev_leng+i)
1063
1064    a(prev_leng+i-1)     = rdxx
1065    irow(prev_leng+i-1) = irow(prev_leng+i)
1066    icol(prev_leng+i-1) = icol(prev_leng+i)-1
1067
1068    a(prev_leng+i+1)     = rdxx
1069    irow(prev_leng+i+1) = irow(prev_leng+i)
1070    icol(prev_leng+i+1) = icol(prev_leng+i)+1
1071
1072    a(prev_leng+i-2)     = rdyy
1073    irow(prev_leng+i-2) = irow(prev_leng+i)
1074    icol(prev_leng+i-2) = icol(prev_leng+i) - ibar
1075 END DO
1076
1077 i=prev_leng + 4*(ibar-2) +3+3
1078
1079 a(i)       = aph
1080 irow(i)    = (ibar-istm)*jstm + ibar*(jbar-jstm-1)+runde+2
1081 icol(i)    = irow(i)
1082 a(i-1)     = rdxx
1083 irow(i-1) = irow(i)
1084 icol(i-1) = icol(i) - 1
1085 a(i-2)     = rdyy
1086 irow(i-2) = irow(i)
1087 icol(i-2) = icol(i) - ibar
1088
1089
1090 a=a*delt
1091
1092 ! Set values for NAG functions
1093 lfill = 0
1094 dtol = 0.0
1095 pstrat = 'C'
1096 milu = 'N'
1097 ALLOCATE(ipivp(n),ipivq(n))
```

```
!ipivp for pstrat = 'U'
!ipivq for pstrat = 'U'
ALLOCATE(istr(n+1),idiag(n))
!nnzc on exit
!npivm on exit
liwork = 7*n+2
ALLOCATE(iwork(liwork))
! Done parameters for f11daf

! Parameters for f11bdf
method = 'BICGSTAB'
precon = 'P'
norm = 'I'
weigth = 'N'
iterm = 1
m = 1!not ref for Method CGS
tol = 1.0E-8!10*EPSILON(dtol)
maxitn = 3000
anorm = -1 !To be fixed by f11bef
!sigmax not ref for iterm = 1
monit=maxitn !No monitoring
lwork = 100+(2*n+m)*(m+2)+0
ALLOCATE(work(lwork))
! Done parametere for f11bdf

ifail = 0
CALL f11daf(n,nnz,a,la,irow,icol,lfill,dtol,pstrat,milu,ipivp,ipivq,istr
    ,idiag,nnzc,npivm,iwork,liwork,ifail)

ALLOCATE(x(n),b(n),wgt(n))

RETURN
END SUBROUTINE precon_pressure
```

Bibliography

[1] Kristoffersen, R. A Navier-Stokes Solver Using The Multigrid Method. Trondheim: NTNU; 1994. MTF-Report 1994:106(A).

[2] Wilcox, D.C. Reassessment of the Scale-Determining Equation for Advanced Turbulence Models. AIAA Journal. 1988;26(11):1299-1310

[3] Ferziger, J.H., Perić, M. Computational Methods for Fluid Dynamics. 3rd ed. Berlin: Springer; 2002

[4] Menter F.R. Two-Equation Eddy-Viscosity Turbulence Models for Engineering Applications. AIAA Journal. 1994;32(8):1598-1605

[5] Rumsey, C. The Wilcox k-omega Turbulence Model[Internet]. [Langley Research Center: NASA]; [updated: 4. August 2014; cited: 18. February 2015] Available from: http://turbmodels.larc.nasa.gov/wilcox.html

[6] OpenMP ARB Company. FAQ OpenMP [Internet]. [place unknown: publisher unknown]; [updated: 12. November 2013; cited: 20. January 2015] Available from: http://www.openmp.org

[7] Moser M.D., Kim J., Mansour N.N. Direct numerical simulation of turbulent channel flow up to Re_τ=590. Physics of Fluids. 1999;11(4):943-945

[8] MATLAB. Solve boundary value problems for ordinary differential equations - MATLAB bvp4c. Place Unknown: Publisher Unknown; Date Unknown[Updated Unknown; cited 4. June 2015], Available from: http://se.mathworks.com/help/matlab/ref/bvp4c.html

[9] Le, H., Moin, P., Kim, J. Direct Numerical Simulation of Turbulent Flow Over a Backward-Facing Step. J. Fluid Mech. 1997;330:349-374