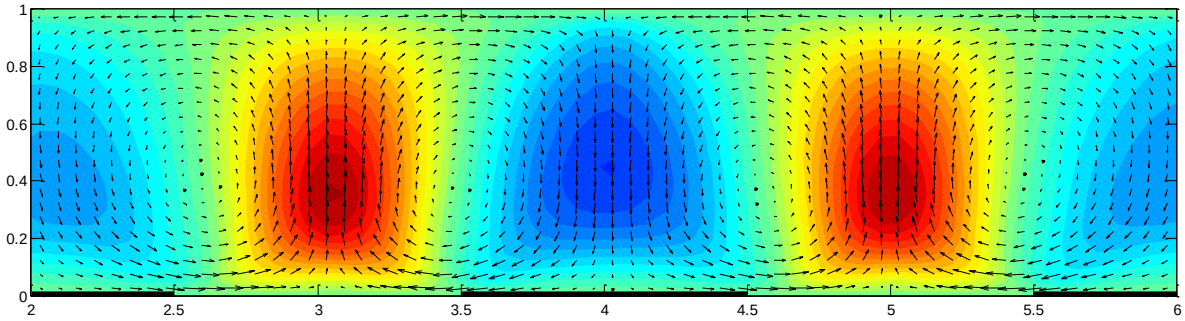
Modelling the anisotropy of turbulence with the SWASH model

Heterogeneous roughness conditions in open channel flows



Msc. Thesis
Tom Bogaard
Mei 2012

TU Delft Delft University of Technology

Modelling the anisotropy of turbulence with the SWASH model

Heterogeneous roughness conditions in open channel flows

Msc. Thesis: modelling the anisotropy of turbulence with the SWASH model – heterogeneous roughness conditions in open channel flows.

Author:	Tom Bogaard
Date:	Mei 15th 2012
Members of the committee:	Prof. dr. ir. W.S.J. Uijttewaal dr. ir. M. Zijlema (daily supervision) dr. ir. B. C. van Prooijen

Delft University of Technology,
Civil Engineering & Geosciences.

Abstract

In this study the focus is on modelling turbulence anisotropy in open channel flows with the SWASH model. Turbulence anisotropy significantly influences the flow features of: channel flows with heterogeneous roughness conditions, curved open channel flows, compound channel flows with different floodplain depths, etc.

The SWASH model is a non-hydrostatic wave-flow model, mainly used to predict the transformation of surface waves from offshore to the beach. For this study, adaptations were made to this SWASH model, in order to model turbulence anisotropy. Two different modelling approaches were used: RANS modelling and Large Eddy Simulation (LES). The SWASH model is extended with a non-linear k-ε closure to the RANS equations, since the standard linear closure does not take turbulence anisotropy into account. A 3D subgrid model is implemented to perform LES.

The performance of the LES code and the RANS model with the non-linear k-ε closure is tested on two flow geometries: an open channel flow with homogeneous bottom roughness conditions and an open channel flow with parallel smooth to rough bed sections.

Results of the RANS computations, for both horizontal homogeneous and non-homogeneous open channel flow, show good agreement with laboratory measurements of Muller and Studerus [13], Nezu and Rodi [17] and Wang and Cheng [32]. Although there is a number of closure constants involved with the non-linear k-ε model, additional tuning of these coefficients was not necessary for this study: both the homogeneous and non-homogenous test case were simulated successfully using the standard values proposed by Speziale [25]. With its low computational costs and robustness, the non-linear k-ε model appears to be a useful extension to the SWASH wave-flow model.

LES results for horizontal uniform flow are validated with DNS data of Moser, Kim and Mansour [12]. Especially near the bed the LES results deviate from the DNS data. The mean velocity, as well as the transverse and vertical turbulence intensities, is seriously underestimated. The deviation from the DNS data is related to the use of non- periodic boundary conditions, the coarse grid resolution, the size of the computational domain and the amount of numerical dissipation that is involved.

Since it is the bottom region where secondary currents are generated, the use of the present LES code for problems involving heterogeneous roughness is not appropriate.

Preface

The master thesis is the finishing part of the master program Hydraulic Engineering at TU Delft. This thesis is about modelling the anisotropy of turbulence and the last eight months I studied this interesting topic. During this period I have expanded my experience of programming and numerical modelling of turbulence flow. For the support during this learning process I would like to thank the members of the graduation committee. Specials thanks to Marcel Zijlema for his support with the SWASH model.

The last few years I spent a lot of time at the Royal library in The Haque, especially during examination periods. These days were a lot more fun in company of Remon, Christiaan and Erwin. In between studying, I have enjoyed the technical discussions and comfortable breaks with you guys.

After all I would like to thank my parents and Nicole for always creating the right practical boundary conditions and their enthusiasm when I was elaborating about modelling issues!

Enjoy reading!

Tom Bogaard.

Mei 2012.

Contents

Introduction

Features of open channel flows have been extensively studied by different authors in the previous decades. Physical scale models as well as in-situ measurements are moreover supplemented by numerical flow models since computational power increases rapidly. A channel geometry that is often studied is a straight horizontal uniform open channel with homogeneous bottom roughness conditions. However, in real rivers and waterways, the roughness conditions are often far from homogeneous. Flood plains for instance are in most cases characterized by relative high roughness conditions compared to the main channel. These heterogeneous roughness conditions lead to flow characteristics different from those for homogeneous channel flow.

Flow characteristics of an open channel flow with heterogeneous roughness conditions are already studied a long time ago in laboratory experiments by Muller and Studerus [13]. More recently, Wang and Cheng [32] published about open channel flows with longitudinal bed forms. Both authors investigated the generation of secondary currents above smooth to rough bed transitions. These secondary currents are influencing the mean primary velocity.

Prandtl distinguishes between two kinds of secondary currents. Secondary currents of Prantl's first kind are associated with non-uniformity of the flow in streamwise direction and are generated by vortex stretching. Secondary currents of Prandtl's second kind are driven by non-homogeneity and anisotropy of turbulence. The currents involved with heterogeneous bottom roughness are of Prandtl's second kind.

Since the secondary currents are induced by turbulence anisotropy and non-homogeneity, modelling the currents is not straightforward. For most practical engineering purposes RANS (Reynolds Averaged Navier-Stokes) models are used. Those flow models solve for the RANS equations in conjunction with a turbulence closure. When the turbulence model does not account for anisotropy of the Reynolds stresses the secondary currents are not represented. Turbulence models that are often used (standard k-ε model, algebraic model, constant eddy viscosity) does not account for turbulence anisotropy. Speziale [25] introduced a non-linear variant of the standard k-ε model. This non-linear closure model can be solved in conjunction with the RANS equations and accounts for anisotropy of turbulence.

Instead of solving the RANS equations, the spatially filtered Navier-Stokes equations can be solved by a 3D Large Eddy Simulation (LES). The aim of LES is to solve the large scale part of the energy spectrum on the numerical grid and model the small scale turbulence by a subgrid model. Consequently, anisotropy of the large scale motion is taken into account.

This thesis is about modelling the anisotropy of turbulence in open channel flows involving heterogeneous roughness conditions. For the numerical simulations the SWASH model [28] is used. The original code, mainly employed for nearshore wave

application, has been adapted first, since the present SWASH model does not account for turbulence anisotropy.

Objective and outline

Based on the introduction there are mainly two different topics that have to be addressed. The first research question arises from the adaption of the SWASH model:

1. In which way does SWASH need to be changed or extended, in order to model the anisotropy of turbulence?

Where the first question is related to the numerical issues involved with the adaptation of the SWASH model, the second question focuses on the physics of turbulence induced secondary currents:

2. How does turbulence anisotropy relate to the secondary currents, generated in open channel flows involving heterogeneous roughness conditions?

In the next sections the focus is on these two questions. In the first Chapter the relevant theoretical background of turbulent flow is addressed. The governing equations are treated as well as a well known solution for uniform homogeneous open channel flow. In Paragraph 1.4 the focus is on channel flow involving non-homogenous roughness conditions.
The second Chapter is on modelling turbulence. In Paragraph 2.2 the standard linear k-ε closure for the RANS equations is considered. Since this standard model does not take the anisotropy into account, the model is extended by Speziale (1987). This non-linear variant is highlighted in Paragraph 2.3. Large Eddy Simulation is discussed in section 2.4. The original model is briefly discussed in Paragraph 3.1. The implementation of the non-linear k-ε closure is subject of Paragraph 3.2. Implementation of the subgrid model used for LES in treated in section 3.4. The fortran coding of both models in provided in Appendix A and B.
Results of the model computations are presented in the last Chapter. A distinction is made between a model geometry with homogenous roughness conditions (Paragraph 4.1) and with non-homogeneous conditions (Paragraph 4.2). In each Paragraph results of the k-ε model are presented first, both with respect to the flow characteristics and the numerics. LES results are presented in 4.1.2 and 4.2.2. Special attention is paid to the discretization of the momentum equations, since LES results appear very sensitive to the numerical implementation.
Conclusions and recommendations are given in Chapter 5.

1. Theory of turbulence

In this first Chapter the governing equations for turbulent flow are presented in Paragraph 1.1. For stationary horizontal homogeneous open channel flow these equations can be simplified. Resulting relations are discussed in Paragraph 1.2. For more complex flow geometries turbulence anisotropy may influence the mean flow characteristics. Turbulence anisotropy is addressed by expanding the turbulent kinetic energy balance in Paragraph 1.3. Paragraph 1.4 focusus on secondary currents driven by turbulence anisotropy. In the last Paragraph the effects of non-homogeneous roughness conditions are discussed.

1.1 Governing equations

The governing equations for an incompressible flow are the continuity and the momentum equations. The flow models discussed in this thesis are all based on these basic equations. The complete incompressible Navier-Stokes equations consist of the momentum equations in three directions and the continuity relation. Including non-hydrostatic pressure they read (Pope [21]):

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{1.1.}$$

$$\frac{\partial u_i}{\partial t} + \frac{\partial u_i u_j}{\partial x_j} + \frac{\partial P/\rho}{\partial x_i} - \frac{\partial}{\partial x_j}\nu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) = \underbrace{f_i}_{\substack{other\ body\ forces\\ usually\ accounts\\ for\ gravity}} \tag{1.2.}$$

where u_i is the velocity in direction x_i, P the pressure, ρ the density and ν the kinematic viscosity.

Note: Throughout this thesis the Einstein summation convention holds for repeated Roman symbols. Greek symbols are used for noncontracted subscripts.

The gravity force can be eliminated by incorporating it into the pressure gradient:

$$f_i = \frac{\partial gz}{\partial x_i}$$

$$p = \underbrace{\frac{P}{\rho}}_{hydrodynamic\ pressure} + \underbrace{gz}_{gravity}$$

$$\frac{\partial p}{\partial x_i} = \frac{\partial P/\rho}{\partial x_i} + \frac{\partial gz}{\partial x_i}$$

where p is the non-hydrostatic pressure and g the gravitational acceleration.

The resulting momentum equations read:

$$\frac{\partial u_i}{\partial t} + \frac{\partial u_i u_j}{\partial x_j} + \frac{\partial p}{\partial x_i} - \frac{\partial}{\partial x_j}\nu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) = 0 \qquad 1.3.$$

Expanded in three different directions, $u(x,x,z), v(x,y,z)\ and\ w(x,y,z)$ respectively in streamwise, spanwise and vertical direction:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \qquad 1.4.$$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} + \frac{\partial p}{\partial x} - \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) = 0 \qquad 1.5.$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} + \frac{\partial p}{\partial y} - \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) = 0 \qquad 1.6.$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} + \frac{\partial p}{\partial z} - \nu\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) = 0 \qquad 1.7.$$

To obtain an equation describing the mean flow, the velocity is split up in two parts (Reynolds decomposition):

$$u_i = \bar{u}_i + u'_i \qquad 1.8.$$

Where, $\bar{u}_i$ represents the time averaged mean velocity and u'_i the instantaneous velocity fluctuation. Consequently $\overline{u'_\iota} = 0$.

After substitution of $u_i = \bar{u}_i + u'_i$ in equation (1.3) and averaging of the resulting equation, an equation for the mean flow is obtained. After some algebra (Pope [21]) the Reynolds Averaged Navier Stokes equation (RANS) reads:

$$\frac{\partial \bar{u}_\iota}{\partial t} + \frac{\partial \bar{u}_i \bar{u}_j}{\partial x_j} = -\frac{1}{\rho_0}\frac{\partial \bar{p}}{\partial x_i} + \nu\frac{\partial^2 \bar{u}_i}{\partial {x_j}^2} - \frac{\partial \overline{u_\iota' u_j'}}{\partial x_j} \qquad 1.9.$$

The momentum transfer due to small scale motions (last term on the RHS of equation (1.9)) is equivalent to a stress: the Reynolds stresses. Due to symmetry of the stress tensor there are three different shear stresses and three different normal stresses:

$$\overline{u_i{}' u_j{}'} = -\frac{\tau_{ij}}{\rho} = -\frac{\tau_{ji}}{\rho} \qquad 1.10.$$

$$\overline{u_i{}' u_i{}'} = -\frac{\tau_{ii}}{\rho} \qquad 1.11.$$

Substitution of the Reynolds stress expression in the momentum equations:

$$\frac{\partial \overline{u}_i}{\partial t} + \frac{\partial \overline{u}_i \overline{u}_j}{\partial x_j} = -\frac{1}{\rho_0}\frac{\partial \bar{p}}{\partial x_i} + \nu \frac{\partial^2 \overline{u}_i}{\partial {x_j}^2} + \frac{1}{\rho_0}\frac{\partial}{\partial x_j}\left(\tau_{ij}\right) \qquad 1.12.$$

1.2 Horizontal homogeneous open channel flow

The Navier-Stokes equations (1.1 and 1.2) introduced in Paragraph 1.1 can be simplified for stationary, homogeneous turbulent flow. The momentum equation (1.12) reduces to a balance between pressure and shear stress (Nieuwstadt [18]).

$$0 = \frac{1}{\rho_0}\frac{\partial \bar{p}}{\partial x} + \frac{\partial \tau_t}{\partial z} \qquad 1.13.$$

With:

$$\tau_t = -\overline{u'w'} + \nu \frac{\partial \overline{u}}{\partial z} \qquad 1.14.$$

$$0 = \frac{1}{\rho_0}\frac{\partial \bar{p}}{\partial z} - \frac{\partial \overline{v'^2}}{\partial z} \qquad 1.15.$$

To be able to integrate over depth, the friction velocity is introduced:

$$u_* = \sqrt{\frac{\tau_s}{\rho}} \qquad 1.16.$$

$${u_*}^2 = \frac{\tau_s}{\rho} = \tau_t(0) \qquad 1.17.$$

The friction velocity and pressure gradient are coupled by the momentum equation in x-direction. When integrated over depth we obtain:

$${u_*}^2 = -\frac{D}{\rho_0}\frac{\partial \bar{p}}{\partial x} \qquad 1.18.$$

Substitution of (1.18) in (1.13) shows that the shear stress vary linearly with depth:

$$\tau_t = -\overline{u'w'} + \nu\frac{\partial \bar{u}}{\partial z} = {u_*}^2(1 - \frac{z}{D}) \qquad 1.19.$$

In order to find an expression for the mean velocity a closure is needed. The Prandtl mixing length hypothesis (Nieuwstadt [18]) relates the velocity to the product of a mixing length and a velocity gradient (U and L are characteristic velocity and length scales):

$$U \sim L\left|\frac{\partial \bar{u}}{\partial z}\right| \qquad 1.20.$$

The eddy viscosity is estimated by the product of a characteristic velocity and length scale (Boussinesq hypothesis).

$$\nu_t \sim UL \qquad 1.21.$$

Substitution of 1.20 in 1.21:

$$\nu_t = L^2\left|\frac{\partial \bar{u}}{\partial z}\right| \qquad 1.22.$$

The turbulent shear stress is expressed as:

$$\tau_t = L^2\left|\frac{\partial \bar{u}}{\partial z}\right|\frac{\partial \bar{u}}{\partial z} \qquad 1.23.$$

Close to the wall the eddies are bound by the presence of the wall. The mixing length in the wall region is approximated by (κ represents the Von Karman constant):

$$L = \kappa z \qquad 1.24.$$

By substitution of 1.22 in 1.19 and 1.24 in 1.23 and the assumption of a constant shear stress, we obtain:

$${u_*}^2 = \frac{\tau_s}{\rho} = \tau_t(0) \qquad 1.25.$$

$${u_*}^2 = \kappa^2 z^2\left|\frac{\partial \bar{u}}{\partial z}\right|\frac{\partial \bar{u}}{\partial z} \qquad 1.26.$$

After integration, equation (1.26) results in the well-known logarithmic profile:

$$\bar{u} = \frac{u_*}{\kappa}\ln\frac{z}{z_0} \qquad 1.27.$$

The parameter z_0 is a constant resulting from the integration process. z_0 corresponds to: $z_0 = 0.11\ \frac{\nu}{|u_*|}$ for hydraulically smooth walls and to $z_0 = \frac{k_s}{30}$ for hydraulically rough conditions. k_s represents the Nikuradse roughness height.

Although expression (1.20) is strictly speaking only valid in the wall region, it estimates the mean velocity relatively correct also in the core region. Therefore the logarithmic profile is widely used as an expression for the mean velocity in turbulent open channel flows outside the viscous sublayer.

1.3 Turbulent Kinetic Energy budget

To have a closer look at the characteristics of turbulence fluctuations the turbulent kinetic energy balance for turbulence fluctuations is introduced. The turbulent kinetic energy (TKE) is defined as: $e = \frac{1}{2}\left(\overline{{u_\iota}'^2}\right)$. To obtain the TKE balance equation the RANS equations (1.9) are subtracted from the original NS equations (1.2). Since $u_i = \bar{u}_i + {u'}_i$ the resulting equation is an expression for the fluctuating velocity:

$$\frac{\partial {u_i}'}{\partial t} + \bar{u_j}\frac{\partial {u_i}'}{\partial x_j} + {u_j}'\frac{\partial \bar{u_\iota}}{\partial x_j} + \frac{\partial {u_i}'{u_j}'}{\partial x_j} - \frac{\partial \overline{{u_\iota}'{u_J}'}}{\partial x_j} = -\frac{1}{\rho_0}\frac{\partial p'}{\partial x_i} + \nu\frac{\partial^2 {u_i}'}{\partial {x_j}^2} \qquad 1.28.$$

Multiplication by ${u_i}'$, time averaging and substitution of $e = \frac{1}{2}\left(\overline{{u_\iota}'^2}\right)$ leads to the TKE balance for velocity fluctuations (Nieuwstadt [18]):

$$\frac{\partial e}{\partial t} + \bar{u}_j\frac{\partial e}{\partial x_j} = -\overline{{u_\iota}'{u_J}'}\frac{\partial \bar{u}_\iota}{\partial x_j} + \frac{\partial}{\partial x_j}\left(-\overline{{u_J}'e'} - \frac{1}{\rho_0}\overline{p'{u_J}'} + \nu\frac{\partial e}{\partial x_j}\right) - \nu\overline{\left(\frac{\partial {u_\iota}'}{\partial x_J}\right)^2} \qquad 1.29.$$

To analyze the anisotropy of the TKE distribution, the turbulent kinetic energy balance can be split up in three directions (transport by the vicous term is neglected). We define $e_\alpha = \frac{1}{2}\overline{{u_\alpha}'^2}$ (the Greek indices are used for noncontracted subscripts):

$$\frac{\partial e_\alpha}{\partial t} + \bar{u}_j\frac{\partial e_\alpha}{\partial x_j} = -\overline{{u_\alpha}'{u_J}'}\frac{\partial \bar{u_\alpha}}{\partial x_j} + \frac{\partial}{\partial x_j}\left(-\overline{{u_J}'e_\alpha'} - \frac{1}{\rho_0}\overline{p'{u_J}'} + \underbrace{\nu\frac{\partial e_\alpha}{\partial x_j}}_{\approx 0}\right) - \nu\overline{\left(\frac{\partial {u_\alpha}'}{\partial x_J}\right)^2} \qquad 1.30.$$

When stationary and horizontal homogenous turbulent flow is assumed the expression reduces to (the pressure term can be expanded using the Kronecker delta):

$$0 = -\overline{u_\alpha{}' u_J{}'}\frac{\partial \overline{u_\alpha}}{\partial x_j} + \frac{\partial}{\partial x_j}\left(-\overline{u_J{}' e_\alpha{}'} - \frac{1}{\rho_0}\overline{p' u_\alpha{}'}\delta_{\alpha j}\right) + \frac{1}{\rho_0}p'\frac{\partial \overline{u_\alpha{}'}}{\partial x_\alpha} - \nu\overline{\left(\frac{\partial u_\alpha{}'}{\partial x_J}\right)^2} \qquad 1.31.$$

The separate equations for the three different directions are displayed below and sum up to the total kinetic energy balance (1.22), (Nieuwstadt [18]).

$$0 = \underbrace{-\overline{u'w'}\frac{\partial \overline{u}}{\partial z}}_{production} \quad \underbrace{+\frac{1}{\rho_0}p'\frac{\partial \overline{u'}}{\partial x}}_{\substack{pressure\\ velocity\\ correlation}} \quad \underbrace{-\frac{\partial}{\partial z}\left(\frac{1}{2}\overline{u'^2 w'}\right)}_{transport} \quad \underbrace{-\nu\overline{\left(\frac{\partial u'}{\partial x_J}\right)^2}}_{dissipation}$$

$$0 = \quad +\frac{1}{\rho_0}p'\frac{\partial \overline{v'}}{\partial y} \quad -\frac{\partial}{\partial z}\left(\frac{1}{2}\overline{v'^2 w'}\right) \quad -\nu\overline{\left(\frac{\partial v'}{\partial x_J}\right)^2} \qquad 1.32.$$

$$0 = \quad +\frac{1}{\rho_0}p'\frac{\partial \overline{w'}}{\partial z} \quad -\frac{\partial}{\partial z}\left(\frac{1}{2}\overline{w'^3} + \frac{1}{\rho_0}\overline{p'w'}\right) \quad -\nu\overline{\left(\frac{\partial w'}{\partial x_J}\right)^2}$$

Production
The production term contains only one component. This implies turbulence anisotropy. The anisotropy is like the production processes related to the macro scales.

Dissipation
Viscous dissipation is related to the small Kolmogorov scales and for high Reynolds numbers the micro scales will be isotropic. The dissipation is equal for all components.

Pressure velocity correlation
Since the production at the macro scales is anisotropic and the dissipation on the micro scales is isotropic, the energy have to be redistributed (during the cascade process) over the three directions. This pressure-velocity term causes this redistribution.

The directional dependence of the production term is not the only source of turbulence anisotropy. Near closed walls and at the free surface anisotropy increases. Velocity components normal to these boundaries are limited since water particles are not able to penetrate the boundaries.

1.4 Turbulence driven secondary flow

The turbulence anisotropy, introduced in the previous Paragraph, lead according to Prandtl to the generation of secondary currents. Prandtl distinguished between two kinds of secondary flow. Skewness induced secondary flow, which is also known as Prandtl's first kind of secondary flow, is generally well understood. The generation is related to vortex stretching induced by non-uniformity of the flow in streamwise direction. The occurrence of secondary currents of Prandtl's second kind is related to the anisotropy of turbulence. Although measurement data on these currents is available since the work of Nikuradse in 1930, the generation mechanism is still a topic to discuss.

As mentioned in the previous Paragraph, the turbulent kinetic energy at the macro scale is in general not equally distributed over the horizontal and vertical components (turbulence anisotropy). This is partially related to the restriction of the production term of (1.25) to the streamwise direction. In addition the damping effect of the free surface and solid walls plays a role. Close to these boundaries the turbulent motion in the direction normal to the wall is restricted since the water particles can't penetrate the boundary. Consequently the turbulent motion parallel to the wall will be large compared to the perpendicular motion.

The non-homogeneity and anisotropy of turbulence is often linked to the generation of streamwise vorticity and implicitly to the generation of secondary currents. The streamwise vorticity balance is used to study the driving mechanism in detail.

Vorticity is defined as the rotation (curl) of the velocity field. In this sense vorticity is directly related to the rotational and dissipative character of turbulence.

$$\boldsymbol{\omega} = \nabla \times \boldsymbol{u} \tag{1.33.}$$

or;

$$\omega_i = \epsilon_{ijk} \frac{\partial u_k}{\partial x_j} \tag{1.34.}$$

Note: ϵ_{ijk} represents the permutation tensor.

Expresion (1.26) shows the relation between vorticity and the deformation rate of the velocity field:

$$\frac{\partial u_k}{\partial x_j} = \frac{1}{2} S_{ij} + \frac{1}{2} \Omega_{ij} = \frac{1}{2} \underbrace{\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right)}_{strain\ tensor} + \frac{1}{2} \underbrace{\left(\frac{\partial u_i}{\partial x_j} - \frac{\partial u_j}{\partial x_i} \right)}_{rotation\ tensor} \tag{1.35.}$$

The vorticity vector is only related to the skew-symmetric tensor:

$$\omega_i = \epsilon_{ijk} \frac{1}{2} \Omega_{ij} \tag{1.36.}$$

Taking the curl of the RANS equations (1.9) leads to the vorticity equations. The pressure term drops out since the curl of a gradient equals zero. Considering the streamwise vorticity equation for fully developed flow ($\frac{\partial}{\partial t} = 0, \frac{\partial}{\partial x} = 0$), we obtain:

$$\overline{u_j} \frac{\partial \overline{\omega_x}}{\partial x_j} = +\nu \frac{\partial^2 \overline{\omega_x}}{\partial {x_j}^2} - \frac{\partial \overline{\omega_x{}' u_j{}'}}{\partial x_j} \tag{1.37.}$$

The Reynolds stresses can be re-arranged to distinguish between non homogeneous and anisotropic terms (Nezu and Nakagawa [15]):

$$\overline{u_j} \frac{\partial \overline{\omega_x}}{\partial x_j} = +\nu \frac{\partial^2 \overline{\omega_x}}{\partial {x_j}^2} + \underbrace{\frac{\partial^2}{\partial y \partial z} (\overline{v'v'} - \overline{w'w'})}_{non-homogeneity} - \underbrace{\left(\frac{\partial^2}{\partial y^2} - \frac{\partial^2}{\partial z^2} \right) \overline{v'w'}}_{anisotropy} \tag{1.38.}$$

Most authors use this streamwise vorticity equation (1.31) for interpretation of their results, when it comes to secondary currents driven by turbulence anisotropy. The reference to the third and fourth term of (1.31) by the labels “non-homogeneity” and “anisotropy” respectively is rather vague. The “non-homogeneous” term would be zero when the Reynolds normal stresses are homogeneous or isotropic. Although this naming is a little arbitrary, it’s often used in literature and will therefore be used throughout this thesis.

Several measurements have been carried out on secondary currents of the second kind in straight open channel flows (i.e. Hinze [8], Gessner and Jones [7], Perkins [20]). Nezu and Nagakawa [15] showed that the non-homogenous term strengthen the secondary currents while the anisotropic term plays a suppressing role. The difference between the two terms should drive the turbulence induced secondary circulation. This statement is both experimentally and numerically verified by respectively Nezu and Nagakawa [16] and Naot and Rodi [14].

Since the standard k-ε model is not capable of modelling secondary currents because its assumption of an isotropic eddy viscosity (Chapter 2.2), numerical studies investigating secondary currents are either based on the non-linear k-ε model (Choi et. al. [4], Kimura and Hosoda [10]) or LES (Falcomer and Armenio [6]).

Very recently Sibel et. al (2011) carried out a LES of a compound channel flow with deep and shallow floodplain depth. They showed that the non-homogeneous term of the vorticity equation is an order of magnitude larger than the anisotropic term. They concluded that the anisotropy of the normal Reynolds stresses is dominant with respect to the origin of secondary currents.

The non-homogeneity of turbulence increases near solid walls and close to the free surface. Previous studies show in addition that there are other regions where streamwise vorticity is generated. Secondary currents are also generated when a transition between a smooth and rough bed is involved. The next Paragraph focusses on the features of secondary currents generated at smooth to rough bed transitions.

1.5 Non-homogeneous roughness conditions

In the previous section the streamwise vorticity balance is used to describe the generation of secondary currents. In particular the anisotropy of the Reynolds normal stresses seems to be responsible for their generation. Perkins [20] was the first who relates this non-homogeneous term to the friction velocity:

$$\frac{\partial^2}{\partial y \partial z}\left(\overline{v'v'} - \overline{w'w'}\right) = \frac{{u_*}^2}{l}\left(\frac{2}{u_*}\frac{\partial u_*}{\partial y} - \frac{1}{l}\frac{\partial l}{\partial y}\right) \qquad 1.39.$$

Where l is a measure for the thickness of the local boundary layer.

Since the work of Perkins H.J. [20] a number of researchers paid attention the non-homogeneity of turbulence induces by lateral varying bed shear stress (Mclean, Townsend, Gerard). Ikeda [9] solved equation (1.32) analytically using the assumption that u_* varies sinusoidal in transverse direction and the shear stress shows a linear distribution in vertical direction. His solution, useful to predict the right sign of the secondary currents rather than generating real values is showed in figure 2.

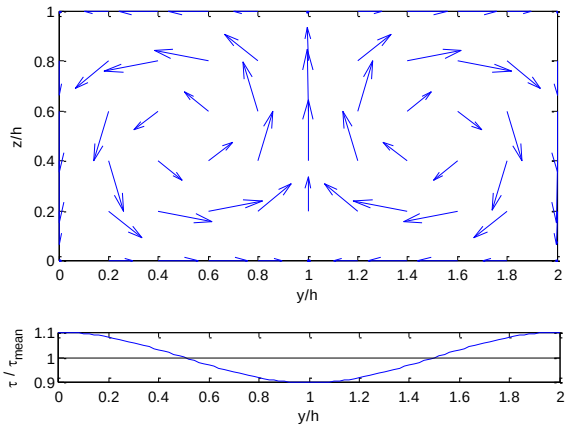


Fig. 1: Analytical solution to equation 1.39, v – w vector and spanwise distribution of bed shear stress, Ikeda [9].

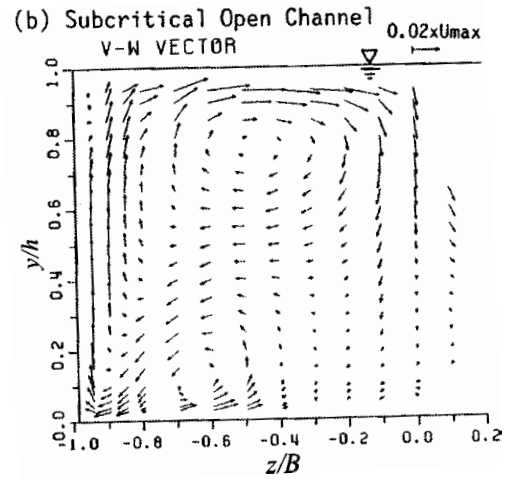


Fig. 2: Measurements of secondary currents in channel corner, v – w vector, Nezu and Rodi [17].

The velocity field shows clearly idealized secondary currents, ignoring side wall and free surface effects. Despite the simplicity of Ikeda's analytical model, its qualitative descriptions are in good agreement with physical experiments. Most interesting is the observed upward flow above the bottom part with relatively low bed shear stress and the downward flow at relatively rough sections.

In real open channel flows this idealized flow pattern is disturbed by free surface effects, the presence of side walls and non-sinusoidal varying bed roughness. Figure 2. shows secondary currents near a side wall measured by Nezu and Rodi [16] in an subcritical open channel flow.

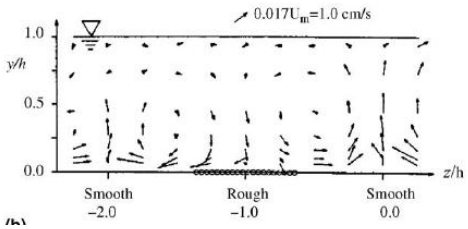


Fig. 3: Secondary currents on smooth to rough bed sections, v – w vector, Muller and Studerus [13].

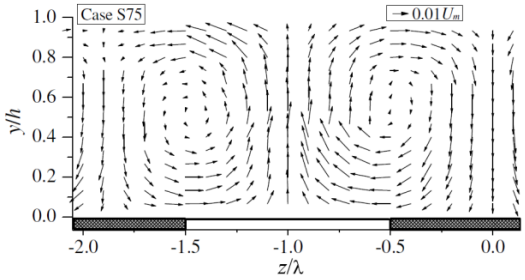


Fig. 4: Secondary currents on smooth to rough bed sections, v – w vector, Wang and Cheng [32].

Muller and Studerus [13] measured in 1979 secondary currents above parallel smooth to rough bed strips. The measurements show qualitatively the same results as the analytical model of Ikeda. Upward movement is measured above smooth strips and downward motion above rough bed strips. Furthermore, the measured flow velocities involved with the secondary motions are roughly 1% of the mean flow velocity. Where maxima of 2% are measured. The center of the secondary circulation is located just above the smooth bed strip.
In 2005 Wang and Cheng [32] carried out experiments similar to those performed by Muller and Studerus [13] and their results where comparable.

2. Turbulence modelling

In this Chapter the modelling of turbulent flow is discussed. After a general introduction RANS modelling with the standard k-ε closure is addressed in Paragraph 2.2. Extension of the linear k-ε closure with non-linear terms is discussed in Paragraph 2.3. Instead of solving the RANS equations, the spatially filtered Navier-Stokes equations can be solved. This Large Eddy Simulation approach is subject of Paragraph 2.4. Boundary conditions for both RANS models and LES are presented in section 2.5.

2.1 Introduction

The Navier-Stokes equations (1.1 and 1.2) describe in principle the flow quantities. To obtain these quantities the system of equations needs to be integrated in combination with proper initial and boundary conditions. The solution of the system needs to be well-posed which implicitly means that little variations of the boundary conditions results in nearly identical solutions of the system. This is the case for laminar flow. For turbulent flow the problem is ill-posed. Nearly identical boundary conditions results in totally different solutions to the system. The latter implies that the Navier-Stokes equations cannot be integrated analytically for turbulent flow (Nieuwstadt [18]).

Since the dynamic system is by definition deterministic, it must be possible to integrate numerically. Direct numerical integration of the Navier-Stokes equations, without making limiting assumptions, is called Direct Numerical Simulation (DNS). To solve the equations on a three dimensional numerical grid, the grid size must be small enough to cover the turbulent microstructure. The total number of grid points is therefore related to the ratio of the characteristic macro length scale and the length scale corresponding to turbulent energy dissipation: the Kolmogorov scale.

$$N_t = \left(\frac{L}{\eta}\right)^3$$

In which N_t is the number of grid points in the computational domain; L is the characteristic macro length scale and η is the Kolmogorov length scale.

The Kolmogorov length scale is related to the dissipation of turbulent kinetic energy (TKE) (Nieuwstadt [18]):

$$\eta = \left(\frac{\nu}{\varepsilon}\right)^{\frac{1}{2}}$$

$$\varepsilon \propto \frac{U^3}{L}$$

The small dissipative Kolmogorov scales are related to the Reynolds number based on the mean flow velocity.

$$Re = \frac{UL}{\nu}$$

Based on the relations above it turns out that the number of grid points is directly related to the Reynolds number:

$$N_t = Re^{\frac{9}{4}}$$

Even for relatively low Reynolds number flows DNS becomes too computational expensive.

There are a few other options instead of DNS to solve the Navier-Stokes equations. In all cases limiting assumptions have to be made. In the following Paragraphs the linear k-ε model, the non-linear k-ε model and Large Eddy Simulation will be introduced.

2.2 Linear k-ε model

Instead of solving the Navier-Stokes equations, the Reynolds Averaged Navier Stokes (RANS) equations (1.9) as introduced in the first Chapter can be solved. The RANS equations are describing the mean flow properties and contain the Reynolds stresses (the averaging operator is dropped in the remaining equations).

$$\frac{\partial \bar{u}_\iota}{\partial t} + \frac{\partial \bar{u}_\iota \bar{u}_j}{\partial x_j} = -\frac{1}{\rho_0}\frac{\partial \bar{p}}{\partial x_i} + \nu \frac{\partial^2 \bar{u}_i}{\partial {x_j}^2} - \underbrace{\frac{\partial \overline{u_\iota' u_j'}}{\partial x_j}}_{Reynolds\ stresses} \qquad 2.1.$$

By introducing the Reynolds stresses, extra unknown quantities have to be solved. This is done by the relating the Reynolds stresses to mean flow properties as follows (Launder and Spalding [11]):

$$-\overline{u_\iota' u_j'} = \nu_t S_{ij} - \frac{2}{3}k \qquad 2.2.$$

$$S_{ij} = \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \qquad 2.3.$$

The eddy viscosity term ν_t is related to the TKE and its dissipation:

$$\nu_t = C_\mu \frac{k^2}{\varepsilon} \qquad 2.4.$$

To close the set of equations, expressions for k and ε are needed. Expressions are found by deriving transport equations for both quantities.

The set of transport equations for production and dissipation of TKE reads (Pope [21]):

$$\frac{\partial k}{\partial t}+\frac{\partial k u_j}{\partial x_j}=-\overline{u_\iota u_j}\frac{\partial u_i}{\partial x_j}-\varepsilon+\frac{\partial}{\partial x_j}\left\{\left(\frac{\nu_t}{\sigma_k}+\nu\right)\frac{\partial k}{\partial x_j}\right\} \qquad 2.5.$$

$$\frac{\partial \varepsilon}{\partial t}+\frac{\partial \varepsilon u_j}{\partial x_j}=-C_{\varepsilon 1}\frac{\varepsilon}{k}\overline{u_\iota u_j}\frac{\partial u_i}{\partial x_j}-C_{\varepsilon 2}\frac{\varepsilon^2}{k}+\frac{\partial}{\partial x_j}\left\{\left(\frac{\nu_t}{\sigma_\varepsilon}+\nu\right)\frac{\partial \varepsilon}{\partial x_j}\right\} \qquad 2.6.$$

The transport equations require a number of closure constants. Common used values are: $\sigma_k=1.0,\ \sigma_\varepsilon=1.3,\ C_{\varepsilon 1}=1.44,\ C_{\varepsilon 1}=1.92, C_\mu=0.09.$

Since the constitutive relation (2.2) of the linear k-ε model does not take the anisotropy of the Reynolds stresses into account, the model is not capable of solving secondary currents of the second kind (Rodi [23]). By introducing non-linear terms in the constitutive relation the model will be applicable to flows induced by turbulence anisotropy. This nonlinear model will be explained in the next Paragraph.

2.3 Non-linear k-ε model

The well-known linear k-ε model consists of a linear constitutive relation (2.2). Since the relation is linear, anisotropy of the Reynolds stresses is not represented. In order to model flows induced by anisotropy, Speziale [25] added nonlinear terms to the constitutive relation.

$$\begin{aligned}-\overline{u_\iota' u_\jmath'}=\nu_t S_{ij}-\frac{2}{3}k\delta_{ij}&\\ -\frac{k}{\varepsilon}\nu_t\Big[C_{\tau 1}\Big(S_{i\alpha}S_{\alpha j}-\frac{1}{3}S_{\alpha\beta}S_{\alpha\beta}\delta_{ij}\Big)+C_{\tau 2}\big(\Omega_{i\alpha}S_{\alpha j}+\Omega_{j\alpha}S_{\alpha i}\big)&\\ +C_{\tau 3}\Big(\Omega_{i\alpha}\Omega_{j\alpha}-\frac{1}{3}\Omega_{\alpha\beta}\Omega_{\alpha\beta}\delta_{ij}\Big)\Big]&\end{aligned} \qquad 2.7.$$

In which:

$$\Omega_{ij}=\frac{\partial \overline{u}_\iota}{\partial x_j}-\frac{\partial \overline{u}_\jmath}{\partial x_i} \quad and \quad \mathrm{S}_{ij}=\frac{\partial \overline{u}_\iota}{\partial x_j}+\frac{\partial \overline{u}_\jmath}{\partial x_i}$$

and $C_{\tau 1}$, $C_{\tau 2}$, $C_{\tau 3}$ are closure constants.
This constitutive relation (2.7) is quadratic in the mean velocity gradients. This is the lowest order of the non-linear k-ε model. This quadratic relation covers the anisotropy of the Reynolds stresses. Relation (2.7) can also be extended with cubic terms in order

to be more sensitive to flows related to streamline curvature. For open channel flows involving secondary currents of Prandtl's second kind the quadratic relation seems to be sufficient.

Note: The Einstein summation convention holds for Greek symbols. The roman symbols are used for noncontracted subscripts.

According to Nissizima [19] the Speziale formulation can be written as follows:

$$-\overline{u_i u_j} = \nu_t S_{ij} - \frac{2}{3} k \delta_{ij} + \frac{k}{\varepsilon} \nu_t \sum_{\beta=1}^{3} C_\beta \left(S_{\beta ij} - \frac{1}{3} S_{\beta \alpha \alpha} \delta_{ij} \right) \qquad 2.8.$$

In which:

$$S_{1ij} = \frac{\partial u_i}{\partial x_\alpha} \frac{\partial u_j}{\partial x_\alpha}$$

$$S_{2ij} = \frac{1}{2} \left(\frac{\partial u_\alpha}{\partial x_i} \frac{\partial u_j}{\partial x_\alpha} + \frac{\partial u_\alpha}{\partial x_j} \frac{\partial u_i}{\partial x_\alpha} \right)$$

$$S_{3ij} = \frac{\partial u_\alpha}{\partial x_i} \frac{\partial u_\alpha}{\partial x_j}$$

and C_β is a closure constant.

For the coefficients C_1 , C_2, C_3 and $C_{\tau 1}$, $C_{\tau 2}$, $C_{\tau 3}$ the following relation holds:

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 0 & -2 \\ 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} C_{\tau 1} \\ C_{\tau 2} \\ C_{\tau 3} \end{pmatrix} \qquad 2.9.$$

The closure constants are tuned by different authors depending on their model application. Speziale introduced a set of constants based on model calibration with experimental data by Laufer (1951) on uniform homogeneous open channel flow. He suggested: $C_{\tau 1} = 0.1512, C_{\tau 2} = 0.1512$, $C_{\tau 3} = 0.0$, corresponding to $C_1 = 0.4536, C_2 = 0.3024$, $C_3 = -0.1512$.

The transport equations for k and ε remains the same, as well as the expression for the eddy viscosity. In Paragraph 3.3 the implementation of the constitutive relation is discussed and in Chapter 4 model results will be presented.

2.4 Large Eddy Simulation

Where the models discussed so far solves for the RANS equations, Large Eddy Simulation (LES) simulates the flow field by integrating the spatially filtered Navier-Stokes equations. As discussed in the introduction of this Chapter DNS is in most cases too computational expensive since the grid spacing is related to the small Kolmogorov scales. Since the focus of most practical flow problems is not on the energy dissipating scales but on anisotropic length scales on the order of the water depth, the equations are spatially filtered. The filtered equations solve for length scales larger than the filter width, smaller scales will be modelled by a so called subgrid model.

The filtering process is not discussed here (see Sagout [24]), but is usually done automatically by discretizing the equations on the computational grid. The filtered equations look nearly identical to the Navier-Stokes equations (1.1 and 1.2):

$$\frac{\partial \tilde{u}_i}{\partial x_i} = 0 \tag{2.10.}$$

$$\frac{\partial \tilde{u}_i}{\partial t} + \frac{\partial \tilde{u}_i \tilde{u}_j}{\partial x_i} = -\frac{\partial \tilde{P}/\rho}{\partial x_i} + \frac{\partial}{\partial x_j} \nu \left(\frac{\partial \tilde{u}_i}{\partial x_j} + \frac{\partial \tilde{u}_j}{\partial x_i} \right) - \underbrace{\frac{\partial \tilde{\tau}_{ij}^{SGS}}{\partial x_j}}_{subgrid\ term} \tag{2.11.}$$

Note: the tilde in 2.10 and 2.11 denotes that described quantities are spatially filtered. In the next Chapters this accent is not used anymore.

Equations 2.10 and 2.11 can now be integrated using a closure model for the subgrid term. In this thesis the standard Smagorinsky model is used. The subgrid stress is defined as the rate of strain of the mean flow:

$$\tau_{ij}^{SGS} = \nu_{SGS} \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) = \nu_{SGS} S_{ij} \tag{2.12.}$$

$$\nu_{SGS} = (c_s \Delta)^2 \sqrt{\frac{1}{2} S_{ij} S_{ij}} \tag{2.13.}$$

c_s is the Smagorinsky constant and Δ is representing the filter width. The Smagorinsky constant is has a theoretically based value of $c_s \approx 0.1$. For open channel flows a lower value of $c_s = 0.065$ suggested by Moin and Kim is often used.

2.5 Boundary conditions

The RANS equations as well as the filtered NS equations are numerically solved in conjunction with boundary conditions. In this section they are briefly discussed.

Inflow and outflow conditions
At the inflow boundary a velocity profile is imposed. For computation with the k-ε model a depth uniform velocity is used. During LES computations a logarithmic velocity profile in stream wise direction is imposed based on the logarithmic law. Random white noise fluctuations are added to the mean velocity to trigger the growth of large scale turbulent structures. The fluctuations are a certain percentage $\hat{r}$ of the ensemble averaged velocity $U_{in} = (u, v, w) = (u_0 \mp \hat{r}u_0,\ \hat{r}u_0,\ \hat{r}u_0)$,$\langle U_{in} \rangle = (\langle u \rangle, \langle v \rangle, \langle w \rangle) = (u_0,\ 0,\ 0)$. The random fluctuations are updated every time step or optionally kept on the same value for a certain time interval.

At the outflow boundary a weak reflective boundary is used as alternative for a fixed water level boundary. The latter gives rise to strong wave reflections and consequently long spin-up times.

Free surface
Since we consider a free surface flow an additional equation is needed to determine the water level. The free surface equation is obtained by integration of the continuity equation and substitution of the kinematic boundary conditions at the bottom and the free surface: $w|_{z=\zeta} = \frac{\partial \zeta}{\partial t} + u\frac{\partial \zeta}{\partial x} + v\frac{\partial \zeta}{\partial y}$ and $w|_{z=-d} = -u\frac{\partial d}{\partial x} - v\frac{\partial d}{\partial y}$. The non-hydrostatic pressure at the free surface is zero. For numerical implementation issues and excessive treatment of the free surface boundary conditions see Stelling and Zijlema [27].

Wall conditions
In order to solve the near wall dynamics directly the first grid point needs to be placed in the region $0 < z^+ = \frac{u_* z}{\nu} < 1$. When such a fine grid resolution is used the solid wall can be represented by a no-slip condition. Since a fine grid resolution is very computational expensive there are different wall models available to model near wall dynamics instead of solving them on the numerical grid. When a subgrid model for the wall shear stress is used, it is sufficient to place the first grid point within the logarithmic layer: $20 < z^+ < 200$. This partial slip condition is a choise not to solve the near wall stress on the computational grid.

In this thesis two different wall models are used. This depends on whether the bottom shear stress is known a priori or not. When the shear stress is known in advance (straight uniform open channel flow) the Schumann model [24] can be used. Schumann represents the wall shear stress as follows:

$$\tau_{xz} = \frac{\bar{u}_{x,y,z_1}}{\langle u_{x,y,z_1} \rangle} \langle \tau \rangle \qquad 2.14.$$

$$\tau_{yz} = \frac{\bar{v}_{x,y,z_1}}{\langle u_{x,y,z_1} \rangle} \langle \tau \rangle \qquad 2.15.$$

The ⟨ ⟩ brackets denote ensemble averaging of the described quantity. The shear stress is related to the friction velocity (2.16):

$$\langle \tau \rangle = \langle {u_*}^2 \rangle \qquad 2.16.$$

By means of the logarithmic velocity law (1.27) the friction velocity can be calculated. In case of hydraulic rough conditions the friction velocity is explicitly related to the equivalent roughness height. When conditions are hydraulically smooth the u_* has to be determined iteratively. When the bottom shear stress is known a priori this has to be done once where after the instantaneous bottom shear stress can be calculated by relations (2.14 and 2.15). When the bottom shear stress is not known a priori (for instance open channel flows with varying bottom roughness and side wall effects) an extension of the Schumann model has to be used which is known as the Grotzbach model. The ensemble averaging is now related to realized flow velocities from a number of previous time steps. The ensemble averaged friction velocity is calculated (iteratively in case of hydraulically smooth conditions) from:

$$\frac{\langle \bar{u}_{x,y,z_1} \rangle}{\langle u_* \rangle} = \frac{1}{\kappa} \ln \frac{z_1}{z_0} \qquad 2.17.$$

The instantaneous bottom shear stress is like the Schumann model calculated by equations 2.14 and 2.15. The Grotzbach model requires in contrast to the Schumann model repeated computation of the ensemble averaged friction velocity for every time step since its value can change over time.

The wall shear stress terms are in the end added as a source term to the horizontal momentum equations for the bottom layers. The bottom friction terms read: $\frac{\tau_{xz}}{\Delta z_1}$ for the streamwise momentum equation and $\frac{\tau_{yz}}{\Delta z_1}$ for the momentum equation in transverse direction.

When side walls are involved the wall shear stress and the bottom shear stress are calculated with the Grotzbach model. Except for the LES of homogeneous open channel flow where a free slip condition at the side walls is used.

3. The SWASH wave-flow model

In this Chapter the SWASH wave-flow model is introduced. The original model accounts for vertical mixing by means of the standard (linear) k-ε closure. Horizontal mixing is modelled by a depth averaged Smagorinsky model. The governing equations of the original model are presented in Paragraph 3.1, where 3.2. focuses on the numerical discretization.

Since the linear k-ε model does not account for turbulence anisotropy, secondary currents of the second kind cannot be solved. In order to model turbulence anisotropy adaptations were made to the original SWASH code: the non-linear k-ε closure of Speziale [25] is implemented. This extension of the linear k-ε model is described in Paragraph 3.3.

Although the original SWASH model accounts for horizontal subgrid stresses by means of the Smagorinsky model (HLES), a three dimensional subgrid model is not implemented in the original code. To run a 3D LES with the SWASH model a standard Smagorinsky model is implemented. The modified code is presented in Paragraph 3.4.

3.1 General model description

SWASH is a general-purpose numerical tool for simulating non-hydrostatic, free-surface, rotational flows. The code is based on the work of Stelling and Zijlema [27], Stelling and Duinmeijer [26] and Zijlema and Stelling [33], [34]. The model accounts besides other processes for bottom friction, subgrid turbulence and vertical mixing. The governing equations are (Zijlema, Stelling and Smit [35]):

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}+w\frac{\partial u}{\partial z}+\frac{g}{\rho_0}\frac{\partial \zeta}{\partial x}+\frac{1}{\rho_0}\frac{\partial q}{\partial x}+\underbrace{\frac{\tau_b}{h}} = \frac{1}{h}\left(\frac{\partial h\tau_{xx}}{\partial x}+\frac{\partial h\tau_{xy}}{\partial y}+\frac{\partial h\tau_{xz}}{\partial z}\right) \tag{3.1.}$$

$$\frac{\partial v}{\partial t}+u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}+w\frac{\partial v}{\partial z}+\frac{g}{\rho_0}\frac{\partial \zeta}{\partial y}+\frac{1}{\rho_0}\frac{\partial q}{\partial y}+\underbrace{\frac{\tau_b}{h}} = \frac{1}{h}\left(\frac{\partial h\tau_{yx}}{\partial x}+\frac{\partial h\tau_{yy}}{\partial y}+\frac{\partial h\tau_{yz}}{\partial z}\right) \tag{3.2.}$$

$$\frac{\partial w}{\partial t}+u\frac{\partial w}{\partial x}+v\frac{\partial w}{\partial y}+w\frac{\partial w}{\partial z}+\frac{1}{\rho_0}\frac{\partial q}{\partial z} = \frac{1}{h}\left(\frac{\partial h\tau_{zz}}{\partial z}\right) \tag{3.3.}$$

$$\frac{\partial \zeta}{\partial t}+\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}+\frac{\partial w}{\partial z} = 0 \tag{3.4.}$$

The equations are depth or layer averaged. Since the vertical grid spacing usually does not allow for solving the dynamics of the boundary layer, a bottom friction term (τ_b) is added to the near bottom layer. This friction term is added to the near bottom layer only. The horizontal turbulent stresses are neglected or optionally solved by a standard Smagorinsky model for horizontal mixing (HLES), whereas the vertical turbulent stresses are solved by the standard linear k-ε model.

3.2 Numerical discretization

Grid schematization

To discretize the governing equations on the computational grid, flow quantities are arranged on a staggered grid (Stelling and Duinmeijer [26]). In figure and table 1 the arrangement of unknowns is shown.

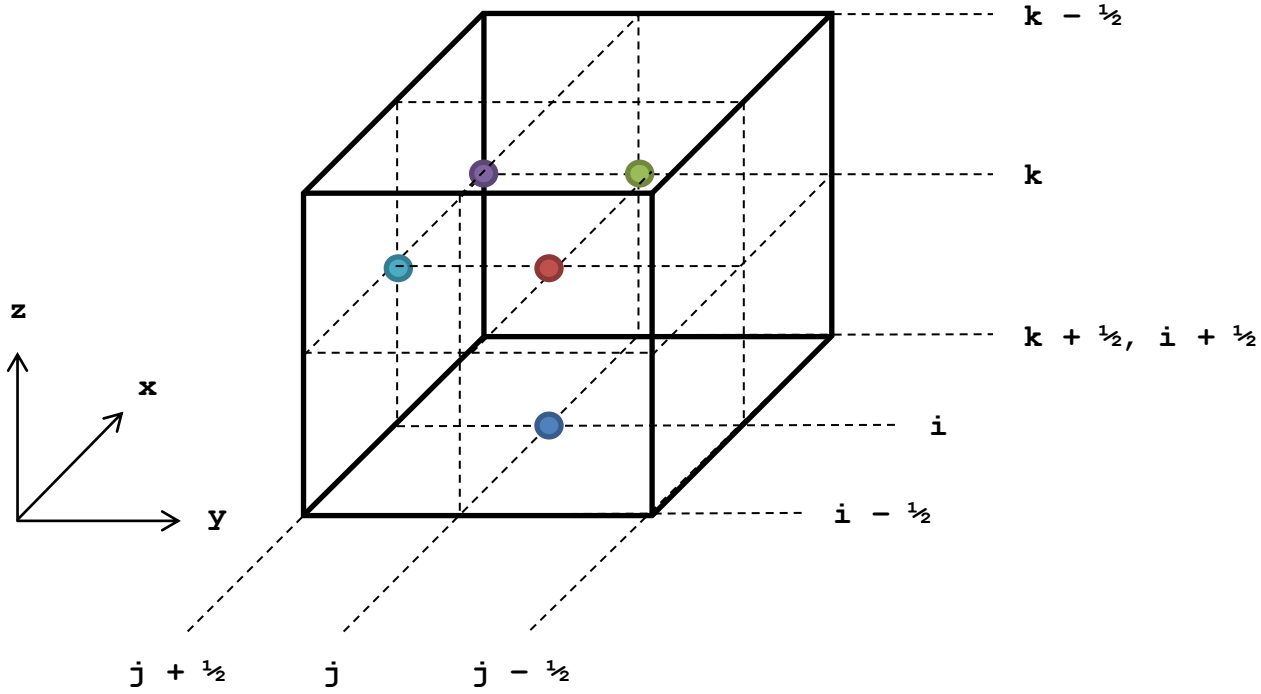


Fig. 5: Arrangement of unknowns on the computational grid.

	quantity	*Set of points on computational grid*
	U – velocity	(i + 1/2, j, k)
	V – velocity	(i, j + 1/2, k)
	W – velocity	(i , j, k + 1/2)
	P – non hydrostatic pressure	(i , j, k + 1/2)
	W – water level	(i , j, k)
	V – horizontal subgrid viscosity	(i + 1/2 , j + 1/2, k)

Table 1: Arrangement of unknowns on the computational grid

The vertical layers are numbered positively from the free surface up to the bottom, while the vertical velocity w is positive when acting in upward direction.

Numerical discretization

For the space discretization and time integration of the governing equations SWASH has a number of methods available. The most important options are discussed in this section. For a complete overview of all available methods, reference is made to the SWASH manual [28].

The time integration with respect to the horizontal advection terms is based on the prediction – correction method of MacCormack. Illustrated by the simplified u-momentum equation; $\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x}$:

Predictor step (˜ denotes predicted quantity) :

$$\widetilde{u^{n+1}_{i+\frac{1}{2},j,k}} = u^{n}_{i+\frac{1}{2},j,k} - u^{n}_{i+\frac{1}{2},j,k}\frac{\Delta t}{\Delta x}(u^{n}_{i+1\frac{1}{2},j,k} + u^{n}_{i+\frac{1}{2},j,k}) \qquad 3.5.$$

This prediction is made based on the first order forward differencing technique. The correction step makes use of the first order backward differencing technique:

$$u^{n+1}_{i+\frac{1}{2},j,k} = \frac{\widetilde{u^{n+1}_{i+\frac{1}{2},j,k}} + u^{n}_{i+\frac{1}{2},j,k}}{2} - u^{n}_{i+\frac{1}{2},j,k}\frac{\Delta t}{2\Delta x}(\widetilde{u^{n+1}_{i+\frac{1}{2},j,k}} + \widetilde{u^{n+1}_{i-\frac{1}{2},j,k}}) \qquad 3.6.$$

The resulting estimation is second order accurate in time and the resulting backward difference (BDF) scheme for the advection terms is also second order accurate in space.

The BDF scheme is the default scheme for the horizontal advection term in SWASH. There is a list of other schemes available which are numerically treated by the correction step of the MacCormack scheme. The space discretization of the vertical advection term is by default central differencing. Optionally there are different upwind methods available.

The space discretization of the non-hydrostatic pressure gradient can be carried out in two ways. The keller-box scheme as well as the standard central differencing approach is available.

The discretization of the vertical momentum equation is based upon the Keller-Box scheme (Zijlema and Stelling [34]). The horizontal advection terms are treated explicitly whereas the vertical terms are treated implicitly for stability reasons. There are a lot of options for their space discretization. The vertical gradient of the non-hydrostatic pressure is treated in the same way as in the horizontal momentum equations.

Introducing non–hydrostatic pressure to the momentum equations requires a solution of the global Poisson problem. The Poisson equation for pressure reads:

$$\frac{\partial^2 P}{\partial x^2}+\frac{\partial^2 P}{\partial y^2}+\frac{\partial^2 P}{\partial z^2}=\frac{1}{\Delta t}\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}+\frac{\partial w}{\partial z}\right) \qquad 3.7.$$

Solving this equation in conjunction with the momentum and continuity equation is not straightforward. There are mainly two ways to solve for the non-hydrostatic pressure. The first method is called the fractional step method. According to this method, the momentum equations are first solved without the pressure gradient, secondly the predicted velocities are substituted into the Poisson equation. After solving the Poisson equation, the predicted velocities are corrected with the pressure gradient. The second method is called the pressure correction technique. The momentum equations are solved with the non-hydrostatic term included based on the solution of Poisson equation from the previous time step. The predicted velocities of the current time step are substituted into the Poisson equation. The predicted velocities are corrected with the gradient of the difference between the non-hydrostatic pressure of the previous time step and the current time step. The pressure correction technique is second order accurate in time and this technique is used in the SWASH model.

For time integration of the pressure gradient the theta-box scheme can be used with $0.5 \leq \theta \leq 1$ for stability reasons. $\theta = 0.5$ corresponds with the second order Crank-Nicholson scheme and $\theta = 1.$ induces first order implicit Euler.

The continuity equation and water level gradient are integrated with the leap frog scheme.

3.3 Implementation of the non-linear k-ε model

In Paragraph 2.3 the non-linear k-ε model by Speziale is introduced. The non-linear model is in fact an extension of the linear constitutive relation from the standard k-ε model. The linear model is already a part of the Swash model. The transport equations for k and ε are the same for the linear model as well as for the non-linear model. Their space discretization is based upon first order upwind and the time integration is fully implicit (for details see the SWASH user manual). The implementation of the non-linear model is restricted to the implementation of the non-linear part of the constitutive relation.

$$-\overline{u_i u_j}=\underbrace{\nu_t S_{ij}-\frac{2}{3}k\delta_{ij}}_{linear\ part}+\underbrace{\frac{k}{\varepsilon}\nu_t\sum_{\beta=1}^{3}C_\beta\left(S_{\beta ij}-\frac{1}{3}S_{\beta\alpha\alpha}\delta_{ij}\right)}_{non-linear\ part} \qquad 3.8.$$

Whereas the linear part is treated explicitly except for the second derivative in vertical direction, the whole non-linear part is treated explicitly and added as a source term to the momentum equations. The non-linear part consists of a list of cross products of modified stress and strain tensors. In Appendix A the Fortran code of the non-linear relation is attached. To avoid a number of pages with multiplied derivatives, relation 3.8 is not expanded in the text. Both the linear and non-linear part are second order accurate in space (central differencing) and first order in time. The transport equations for k and ε (2.5 and 2.6) are first order in time and space (first order upwind scheme for space discretization and first order time spitting).

For the discretization and integration of the momentum equations reference is made to Chapter 3.2 and (Zijlema and Stelling [34]), since no changes have been made to the original SWASH code.

3.4 Implementation of Large Eddy Simulation

In Paragraph 3.4 the filtered Navier-Stokes equations are introduced. In order to perform a LES these equations need to be solved in conjunction with a subgrid model. To that end some adaptations have to be made to the governing equations of the SWASH model:

- Extension to fully 3D system of equations

The horizontal depth averaged subgrid stresses and the vertical subgrid stress needs to be replaced by a three dimensional subgrid model and subgrid stresses need to be added to each momentum equation. For this purpose the standard Smagorinsky model will be used.

- Modelling the subgrid scales

The Smagorinsky model requires for the calculation of a subgrid viscosity.

- Wall model

Bottom friction is taken into account by adding a shear stress to the horizontal momentum equations for the near bottom layer. In the SWASH code this shear stress is related to the depth averaged flow velocity by a user-defined friction coefficient. This wall model needs to be replaced by the Schumann or Grozbach model introduced in Section 2.5

In the next section the numerical treatment of the implemented terms is discussed. In addition, the Fortran code of the implemented modules is supplemented in Appendix B.

The subgrid stress terms are explicitly treated except for the terms where second order derivatives in vertical direction are involved. Space discretization is obviously based on central differencing since second order derivatives are involved.

The subgrid terms in the u-momentum equation reads:

$$\frac{\partial \tilde{\tau}_{xj}^{SGS}}{\partial x_j} = \frac{\partial \tilde{\tau}_{xx}^{SGS}}{\partial x} + \frac{\partial \tilde{\tau}_{xy}^{SGS}}{\partial y} + \frac{\partial \tilde{\tau}_{xz}^{SGS}}{\partial z} \tag{3.9.}$$

$$\begin{aligned}\frac{\partial \tilde{\tau}_{xj}^{SGS}}{\partial x_j} = \frac{\partial}{\partial x}\left\{\nu_{SGS}\left(\frac{\partial \tilde{u}}{\partial x} + \frac{\partial \tilde{u}}{\partial x}\right)\right\} + \frac{\partial}{\partial y}\left\{\nu_{SGS}\left(\frac{\partial \bar{v}}{\partial x} + \frac{\partial \bar{u}}{\partial y}\right)\right\} \\ + \frac{\partial}{\partial z}\left\{\nu_{SGS}\left(\frac{\partial \bar{w}}{\partial x} + \frac{\partial \bar{u}}{\partial z}\right)\right\}\end{aligned} \tag{3.10.}$$

The subgrid terms of the u-momentum equation are discretized around the u-point (i + 1/2, j, k) and are added to the RHS of the momentum equation:

$$\begin{aligned}&\frac{\partial}{\partial x}\left\{\nu_{SGS}\left(\frac{\partial \bar{u}}{\partial x} + \frac{\partial \bar{u}}{\partial x}\right)\right\} \\ &= \frac{1}{\Delta x}\left\{2\nu_{SGS}\left(\frac{u^{n-1}_{i+1\frac{1}{2},j,k} - u^{n-1}_{i+\frac{1}{2},j,k}}{\Delta x}\right)\right. \\ &\left. - 2\nu_{SGS}\left(\frac{u^{n-1}_{i+\frac{1}{2},j,k} - u^{n-1}_{i-\frac{1}{2},j,k}}{\Delta x}\right)\right\}\end{aligned} \tag{3.11.}$$

$$\begin{aligned}&\frac{\partial}{\partial y}\left\{\nu_{SGS}\left(\frac{\partial \bar{v}}{\partial x} + \frac{\partial \bar{u}}{\partial y}\right)\right\} \\ &= \frac{1}{\Delta y}\left\{\nu_{SGS}\left(\frac{v^{n-1}_{i+1,j+\frac{1}{2},k} - v^{n-1}_{i,j+\frac{1}{2},k}}{\Delta x} + \frac{u^{n-1}_{i+\frac{1}{2},j+1,k} - u^{n-1}_{i+\frac{1}{2},j,k}}{\Delta y}\right)\right. \\ &\left. - \nu_{SGS}\left(\frac{v^{n-1}_{i+1,j-\frac{1}{2},k} - v^{n-1}_{i,j-\frac{1}{2},k}}{\Delta x} + \frac{u^{n-1}_{i+\frac{1}{2},j,k} - u^{n-1}_{i+\frac{1}{2},j-1,k}}{\Delta y}\right)\right\}\end{aligned} \tag{3.12.}$$

$$\begin{aligned}&\frac{\partial}{\partial z}\left\{\nu_{SGS}\left(\frac{\partial \bar{w}}{\partial x}\right)\right\} \\ &= \frac{1}{\Delta z}\left\{\nu_{SGS}\left(\frac{w^{n-1}_{i+1,j,k-\frac{1}{2}} - w^{n-1}_{i,j,k-\frac{1}{2}}}{\Delta x}\right)\right. \\ &\left. - \nu_{SGS}\left(\frac{w^{n-1}_{i+1,j,k+\frac{1}{2}} - w^{n-1}_{i,j,k+\frac{1}{2}}}{\Delta x}\right)\right\}\end{aligned} \tag{3.13.}$$

The subgrid terms in the v-momentum equation reads:

$$\frac{\partial \tilde{\tau}_{yj}^{SGS}}{\partial x_j} = \frac{\partial \tilde{\tau}_{yx}^{SGS}}{\partial x} + \frac{\partial \tilde{\tau}_{yy}^{SGS}}{\partial y} + \frac{\partial \tilde{\tau}_{yz}^{SGS}}{\partial z} \tag{3.14.}$$

$$\begin{aligned}\frac{\partial \tilde{\tau}_{yj}^{SGS}}{\partial x_j} = &\frac{\partial}{\partial x}\left\{\nu_{SGS}\left(\frac{\partial \bar{u}}{\partial y} + \frac{\partial \bar{v}}{\partial x}\right)\right\} + \frac{\partial}{\partial y}\left\{\nu_{SGS}\left(\frac{\partial \bar{v}}{\partial y} + \frac{\partial \bar{v}}{\partial y}\right)\right\} \\ &+ \frac{\partial}{\partial z}\left\{\nu_{SGS}\left(\frac{\partial \bar{w}}{\partial y} + \frac{\partial \bar{v}}{\partial z}\right)\right\}\end{aligned} \tag{3.15.}$$

The terms of the v-momentum equation are discretized around the v-point (i, j + ½, k) and are added to the RHS of the momentum equation:

$$\begin{aligned}&\frac{\partial}{\partial x}\left\{\nu_{SGS}\left(\frac{\partial \bar{u}}{\partial y} + \frac{\partial \bar{v}}{\partial x}\right)\right\} \\ &= \frac{1}{\Delta x}\left\{\nu_{SGS}\left(\frac{u^{n-1}_{i+\frac{1}{2},j+1,k} - u^{n-1}_{i+\frac{1}{2},j,k}}{\Delta y} + \frac{v^{n-1}_{i+1,j+\frac{1}{2},k} - v^{n-1}_{i,j+\frac{1}{2},k}}{\Delta x}\right)\right. \\ &\left. - \nu_{SGS}\left(\frac{u^{n-1}_{i-\frac{1}{2},j+1,k} - u^{n-1}_{i-\frac{1}{2},j,k}}{\Delta y} + \frac{v^{n-1}_{i,j+\frac{1}{2},k} - v^{n-1}_{i-1,j+\frac{1}{2},k}}{\Delta x}\right)\right\}\end{aligned} \tag{3.16.}$$

$$\begin{aligned}&\frac{\partial}{\partial y}\left\{\nu_{SGS}\left(\frac{\partial \bar{v}}{\partial y} + \frac{\partial \bar{v}}{\partial y}\right)\right\} \\ &= \frac{1}{\Delta y}\left\{2\nu_{SGS}\left(\frac{v^{n-1}_{i,j+1\frac{1}{2},k} - v^{n-1}_{i,j+\frac{1}{2},k}}{\Delta y}\right)\right. \\ &\left. - 2\nu_{SGS}\left(\frac{v^{n-1}_{i,j+\frac{1}{2},k} - v^{n-1}_{i,j-\frac{1}{2},k}}{\Delta y}\right)\right\}\end{aligned} \tag{3.17.}$$

$$\begin{aligned}&\frac{\partial}{\partial z}\left\{\nu_{SGS}\left(\frac{\partial \bar{w}}{\partial y}\right)\right\} \\ &= \frac{1}{\Delta z}\left\{\nu_{SGS}\left(\frac{w^{n-1}_{i,j+1,k-\frac{1}{2}} - w^{n-1}_{i,j,k-\frac{1}{2}}}{\Delta y}\right)\right. \\ &\left. - \nu_{SGS}\left(\frac{w^{n-1}_{i,j+1,k+\frac{1}{2}} - w^{n-1}_{i,j,k+\frac{1}{2}}}{\Delta y}\right)\right\}\end{aligned} \tag{3.18.}$$

The second derivatives in vertical direction $\frac{\partial}{\partial z}\left\{\nu_{SGS}\left(\frac{\partial \bar{u}}{\partial z}\right)\right\}$ and $\frac{\partial}{\partial z}\left\{\nu_{SGS}\left(\frac{\partial \bar{v}}{\partial z}\right)\right\}$ in the u- and v- momentum equation respectively are implicitly treated to enhance the stability of the solution.

The subgrid terms are discretized around the w-velocity point (i, j, k + ½) and are treated explicitly.

$$\frac{\partial \tilde{\tau}_{zj}^{SGS}}{\partial x_j} = \frac{\partial}{\partial x}\left\{\nu_{SGS}\left(\frac{\partial \bar{u}}{\partial z}+\frac{\partial \bar{w}}{\partial x}\right)\right\}+\frac{\partial}{\partial y}\left\{\nu_{SGS}\left(\frac{\partial \bar{v}}{\partial z}+\frac{\partial \bar{w}}{\partial y}\right)\right\} + \frac{\partial}{\partial z}\left\{\nu_{SGS}\left(\frac{\partial \bar{w}}{\partial z}+\frac{\partial \bar{w}}{\partial z}\right)\right\} \tag{3.19.}$$

$$\frac{\partial}{\partial x}\left\{\nu_{SGS}\left(\frac{\partial \bar{u}}{\partial z}+\frac{\partial \bar{w}}{\partial x}\right)\right\} = \frac{1}{\Delta x}\left\{\nu_{SGS}\left(\frac{u^{n-1}_{i+\frac{1}{2},j,k}-u^{n-1}_{i+\frac{1}{2},j,k+1}}{\Delta z}+\frac{w^{n-1}_{i+1,j,k+\frac{1}{2}}-w^{n-1}_{i,j,k+\frac{1}{2}}}{\Delta x}\right) - \nu_{SGS}\left(\frac{u^{n-1}_{i-\frac{1}{2},j,k}-u^{n-1}_{i-\frac{1}{2},j,k+1}}{\Delta z}+\frac{w^{n-1}_{i,j,k+\frac{1}{2}}-w^{n-1}_{i-1,j,k+\frac{1}{2}}}{\Delta x}\right)\right\} \tag{3.20.}$$

$$\frac{\partial}{\partial y}\left\{\nu_{SGS}\left(\frac{\partial \bar{v}}{\partial z}+\frac{\partial \bar{w}}{\partial y}\right)\right\} = \frac{1}{\Delta x}\left\{\nu_{SGS}\left(\frac{v^{n-1}_{i,j+\frac{1}{2},k}-v^{n-1}_{i,j+\frac{1}{2},k+1}}{\Delta z}+\frac{w^{n-1}_{i,j+1,k+\frac{1}{2}}-w^{n-1}_{i,j,k+\frac{1}{2}}}{\Delta y}\right) - \nu_{SGS}\left(\frac{v^{n-1}_{i,j-\frac{1}{2},k}-v^{n-1}_{i,j-\frac{1}{2},k+1}}{\Delta z}+\frac{w^{n-1}_{i,j,k+\frac{1}{2}}-w^{n-1}_{i,j-1,k+\frac{1}{2}}}{\Delta y}\right)\right\} \tag{3.21.}$$

The vertical second derivative of the subgrid term $\frac{\partial}{\partial z}\left\{\nu_{SGS}\left(\frac{\partial \bar{w}}{\partial z}+\frac{\partial \bar{w}}{\partial z}\right)\right\}$ is treated implicitly.

Subgrid viscosity

The subgrid viscosity is based on the standard Smagorinsky model, introduced in Chapter 3.4.

$$\nu_{SGS} = (c_s \Delta)^2 \sqrt{\frac{1}{2} \bar{S}_{ij} \bar{S}_{ij}} \qquad 3.22.$$

The subgrid viscosity is discretised around the point (i + ½, j + ½, k) with use of velocities from the previous time step. The filter width Δ is estimated as $\sqrt[3]{\Delta x \Delta y \Delta z}$ since the filtering operation is done directly by discretizing the equations on the numerical grid. The Smagorinsky constant c_s near the bottom is reduced by a van Driest damping function (Pope [21]) in order to make sure c_s reaches zero at the bottom.

$$c_s = c_{s0} \left(1 - e^{-\frac{z_+}{A_+}} \right) \qquad 3.23.$$

$A_+ = 26$ is the Van Driest parameter. When necessary velocities are interpolated first and the derivative is based upon the interpolated velocities. This should be more accurate than the other way round.

The subgrid viscosity, discretised around the point (i + ½, j + ½, k), reads:

$$\nu_{SGS} = \left(c_s \sqrt[3]{\Delta x \Delta y \Delta z} \right)^2 \sqrt{\frac{1}{2} \bar{S}_{ij} \bar{S}_{ij}} \qquad 3.24.$$

with:

$$\frac{1}{2}\bar{S}_{ij}\bar{S}_{ij}$$

$$= \frac{1}{2}\left\{ 4\left[\frac{u^{n-1}_{i+1\frac{1}{2},j,k} + u^{n-1}_{i+1\frac{1}{2},j+1,k} - u^{n-1}_{i-\frac{1}{2},j,k} - u^{n-1}_{i-\frac{1}{2},j+1,k}}{4\Delta x}\right]^2 + 2\left[\frac{u^{n-1}_{i+\frac{1}{2},j+1,k} - u^{n-1}_{i+\frac{1}{2},j,k}}{\Delta y} + \frac{v^{n-1}_{i+1,j+\frac{1}{2},k} - v^{n-1}_{i,j+\frac{1}{2},k}}{\Delta x}\right]^2 \right.$$

$$+ 2\left[\frac{\left(u^{n-1}_{i+\frac{1}{2},j,k} + u^{n-1}_{i+\frac{1}{2},j+1,k} + u^{n-1}_{i+\frac{1}{2},j,k-1} + u^{n-1}_{i+\frac{1}{2},j+1,k-1}\right) - \left(u^{n-1}_{i+\frac{1}{2},j,k+1} + u^{n-1}_{i+\frac{1}{2},j+1,k+1} + u^{n-1}_{i+\frac{1}{2},j,k} + u^{n-1}_{i+\frac{1}{2},j+1,k}\right)}{4\Delta z}\right.$$

$$\left. + \frac{\left(w^{n-1}_{i+1,j,k+\frac{1}{2}} + w^{n-1}_{i+1,j+1,k+\frac{1}{2}} + w^{n-1}_{i+1,j,k-\frac{1}{2}} + w^{n-1}_{i+1,j+1,k-\frac{1}{2}}\right) - \left(w^{n-1}_{i,j,k+\frac{1}{2}} + w^{n-1}_{i,j+1,k+\frac{1}{2}} + w^{n-1}_{i,j,k-\frac{1}{2}} + w^{n-1}_{i,j+1,k-\frac{1}{2}}\right)}{4\Delta x}\right]^2$$

$$+ 4\left[\frac{\left(v^{n-1}_{i,j+\frac{1}{2},k} + v^{n-1}_{i+1,j+\frac{1}{2},k} + v^{n-1}_{i,j+1\frac{1}{2},k} + v^{n-1}_{i+1,j+1\frac{1}{2},k}\right) - \left(v^{n-1}_{i,j-\frac{1}{2},k} + v^{n-1}_{i+1,j-\frac{1}{2},k} + v^{n-1}_{i,j+\frac{1}{2},k} + v^{n-1}_{i+1,j+\frac{1}{2},k}\right)}{4\Delta y}\right]^2$$

$$+ 2\left[\frac{\left(v^{n-1}_{i,j+\frac{1}{2},k} + v^{n-1}_{i+1,j+\frac{1}{2},k} + v^{n-1}_{i,j+\frac{1}{2},k-1} + v^{n-1}_{i+1,j+\frac{1}{2},k-1}\right) - \left(v^{n-1}_{i,j+\frac{1}{2},k} + v^{n-1}_{i+1,j+\frac{1}{2},k} + v^{n-1}_{i,j+\frac{1}{2},k+1} + v^{n-1}_{i+1,j+\frac{1}{2},k+1}\right)}{4\Delta z}\right.$$

$$\left. + \frac{\left(w^{n-1}_{i,j+1,k+\frac{1}{2}} + w^{n-1}_{i+1,j+1,k+\frac{1}{2}} + w^{n-1}_{i,j+1,k-\frac{1}{2}} + w^{n-1}_{i+1,j+1,k-\frac{1}{2}}\right) - \left(w^{n-1}_{i,j,k+\frac{1}{2}} + w^{n-1}_{i+1,j,k+\frac{1}{2}} + w^{n-1}_{i,j,k-\frac{1}{2}} + w^{n-1}_{i+1,j,k-\frac{1}{2}}\right)}{4\Delta y}\right]^2$$

$$\left. + 4\left[\frac{\left(w^{n-1}_{i,j,k-\frac{1}{2}} + w^{n-1}_{i+1,j,k-\frac{1}{2}} + w^{n-1}_{i,j+1,k-\frac{1}{2}} + w^{n-1}_{i+1,j+1,k-\frac{1}{2}}\right) - \left(w^{n-1}_{i,j,k+\frac{1}{2}} + w^{n-1}_{i+1,j,k+\frac{1}{2}} + w^{n-1}_{i,j+1,k+\frac{1}{2}} + w^{n-1}_{i+1,j+1,k+\frac{1}{2}}\right)}{4\Delta z}\right]^2 \right\}$$

Gradients close to boundaries

When virtual points are needed to obtain derivatives close to boundaries, velocities are mirrored. At no slip boundaries this means zero velocities, at the bottom and the free surface this implies a von Neumann boundary condition.

The Schumann wall model is implemented to account for side wall and bottom friction. See Appendix B for the Fortran code.

4. Results and discussions

In this Chapter results of different model runs with respect to the flow characteristics will be presented, as well as the behavior of the implemented code with respect to the numerics. Since the original code is adapted and extended, a homogeneous open channel flow is investigated first (Paragraph 4.1). In addition calculations were made involving non homogeneous roughness conditions (Paragraph 4.2).

4.1 Homogeneous open channel flow

In Paragraph 4.1.1 an open channel flow with homogeneous bottom roughness is investigated with the linear and non-linear k-ε model. Numerical aspects of the k-ε model are treated in the same section. Paragraph 4.1.2 shows the results of a LES for homogeneous channel flow. Special attention is paid to the discretization of the momentum equations, since LES results appear very sensitive to the numerical implementation.

4.1.1 linear and non-linear k-ε model

Model set-up
A comparison is made between the standard linear model and the non-linear variant with coefficients specified by Speziale [25]. The flow geometry consist of a standard open channel with hydraulic smooth bottom and side wall conditions (fig. 6). The length and width of the domain are 120 and 8 times the water depth, respectively.

At the inflow boundary a uniform velocity profile is imposed: $\overline{U}_{in} = (\bar{u}, \bar{v}, \overline{w}) = (u_0,\ 0,\ 0)$. At the outflow boundary a weak reflective boundary is used as alternative for a fixed water level boundary. The latter gives rise to strong wave reflections and consequently long spin-up times. Initially the flow velocities were set to zero.

The Reynolds number based on the friction velocity is $Re_* = 623$ and we consider subcritical flow. The results are obtained from the stationary situation, when the wall shear stress balances the mean pressure gradient. Since the geometry of the channel is symmetric at the channel center ($y = 4H$) only the left half of the channel is plotted.

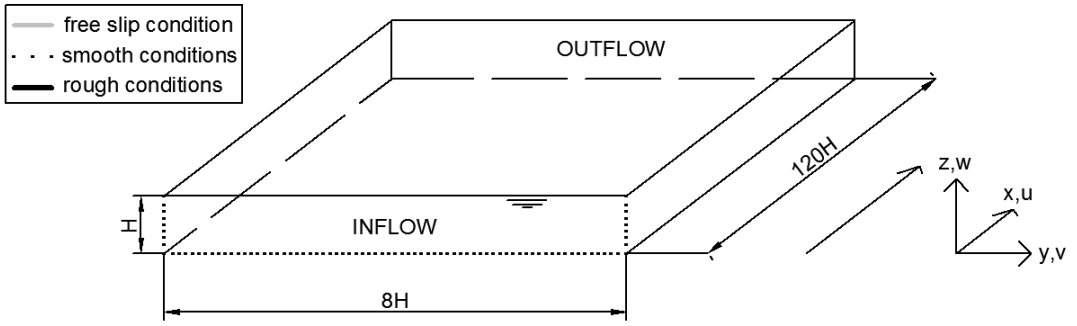


Fig. 6: Computational domain for RANS compuations with horizontal homogeneous roughness conditions.

Primary velocity distribution and secondary currents

The mean velocity profiles are plotted in figures 7 and 8. The black line shows in both figures the log law for hydraulically smooth conditions. The black dots represent the computational results in the channel center. The profiles are in good agreement with the experimental results of Nezu and Rodi [17] who states that, the mean velocity profile should not be influenced by the solid side walls after two times the water depth from the wall. Both the linear and non-linear model shows the same mean velocity profile at the channel center. Within in the core region ($z_+ > 200, \frac{z}{h} > 0.04$) the velocity deviates from the logarithmic law. This is expected since the law of the wake holds for the mean velocity in the core region.

Within one water depth from the wall the linear results deviates significantly from the non-linear results. The velocity dip at $\frac{z}{H} \approx \frac{3}{4}$ as described by Nezu and Nagakawa [15] is clearly visible in the model results from the non-linear model. The linear model, as expected, is not capable of reproducing the correct velocity profile near solid walls.

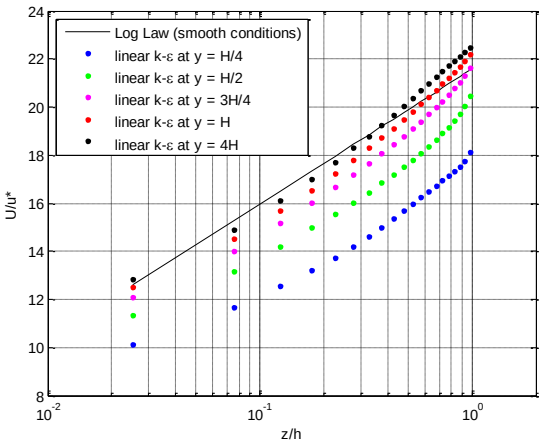


Fig. 7: Results linear k-ε model. Mean primary velocity profiles at the channel center and at different transversal locations close to the solid wall at H = 0.

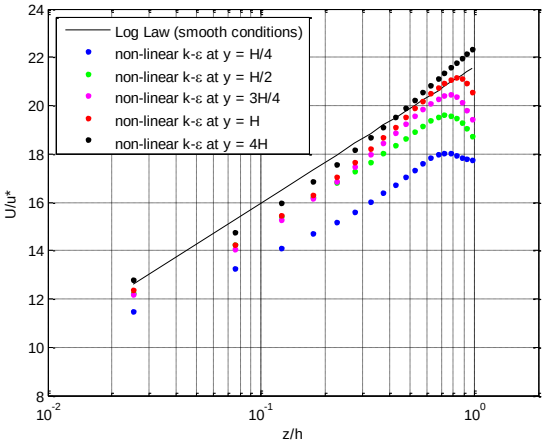


Fig. 8: Results non-linear k-ε model. Mean primary velocity profiles at the channel center and at different transversal locations close to the solid wall at H = 0.

The mean streamwise velocity distributions, corresponding to the velocity profiles in figure 7 and 8, are showed below. The relatively strong velocity dip in the measurements of Nezu and Rodi [17] compared to the present model results is most like due to the use of a different model geometry. Nezu and Rodi [17] carried out their measurements in an open channel where the channel width is as large as twice the water depth. The width of the secondary currents is therefore limited to half the channel width since the flow pattern is symmetric round the channel center. Measurements of Wang and Cheng [32] in a much wider open channel show a less dominant velocity dip (fig. 12).

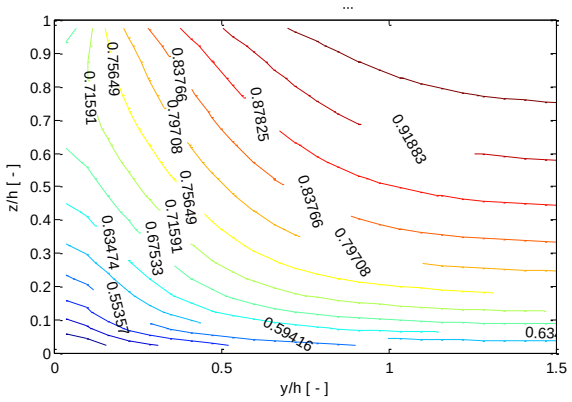


Fig. 9: Mean primary velocity distribution, percentage of maximum velocity, linear k-ε model.

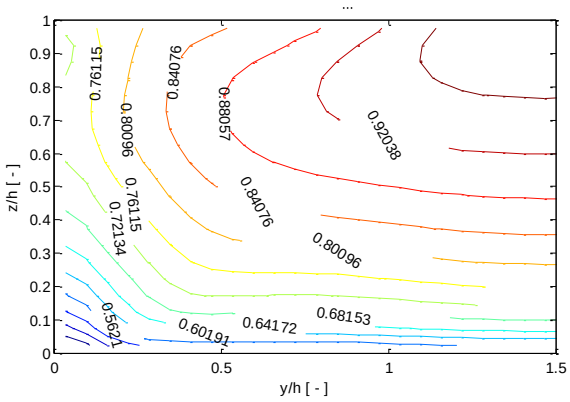


Fig. 10: Mean primary velocity distribution, percentage of maximum velocity, non-linear k-ε model.

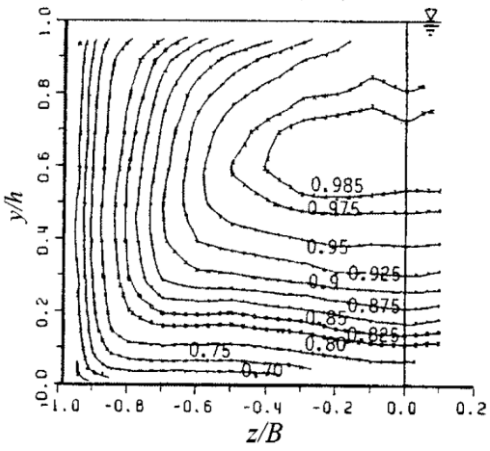


Fig. 1: Mean primary velocity distribution, percentage of maximum velocity, Nezu and Rodi [17].

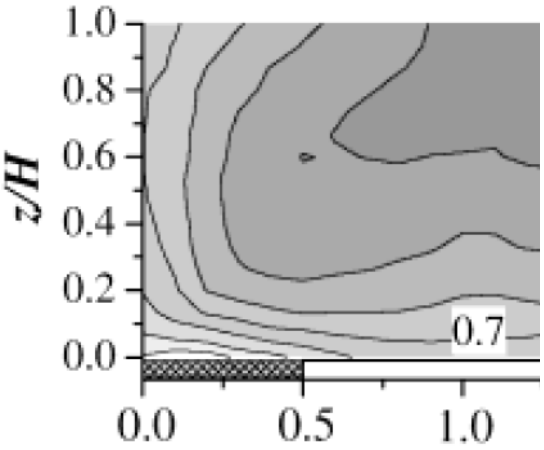


Fig. 12: Mean primary velocity distribution, percentage of maximum velocity, Wang and Cheng [32].

Vector plots of secondary currents are compared and presented in figure 2 and 3. The non-linear model does not show any secondary currents. This was expected since the non-linear does not solve for the turbulence anisotropy.

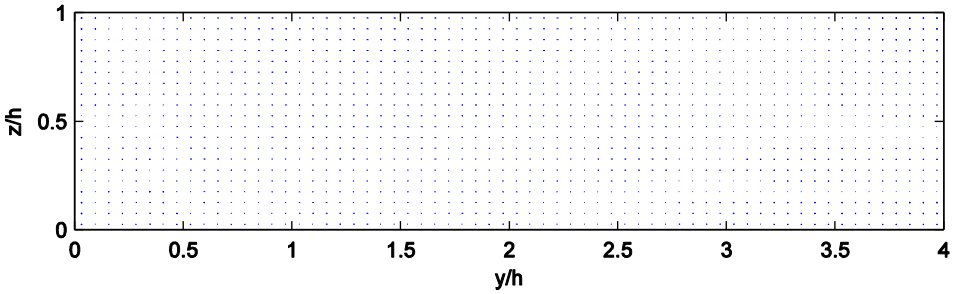


Fig. 2: Secondary currents, v-w vector, linear k-ε model.

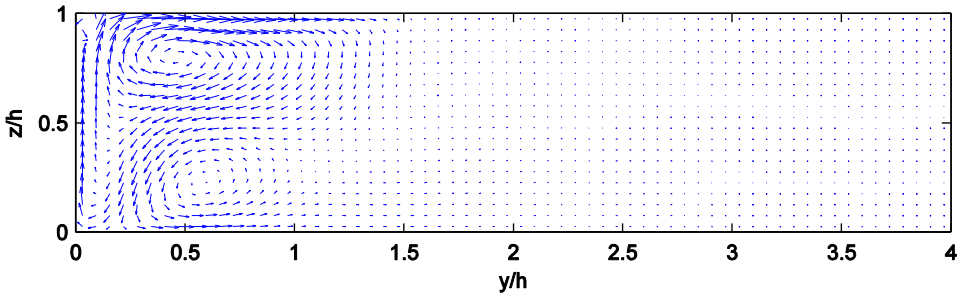


Fig. 3: Secondary currents, v-w vector, non-linear k-ε model.

The results of the non-linear model show good agreement with the measurements from Nezu and Rodi [17]. The bottom as well as the counter rotating cell at the free surface are somewhat extended in transverse direction compared to the measurements. This deviation between measurements and computational results is related to the difference in model geometry as described above.

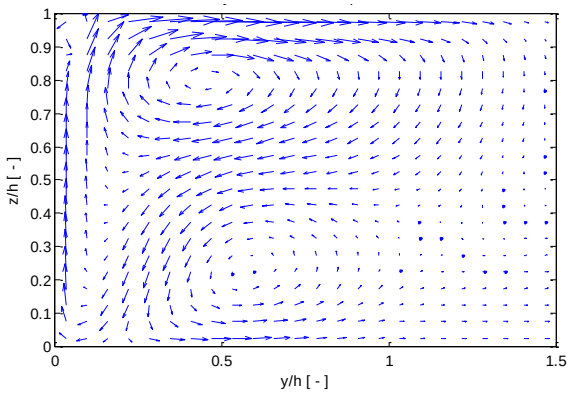


Fig. 4: Secondary currents in corner region, v-w vector, non-linear k-ε model.

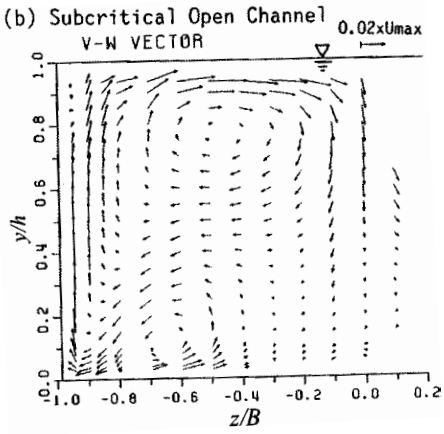


Fig. 5: Secondary currents in corner region, v-w vector, Nezu and Rodi [17].

The mean magnitude of the secondary currents is in order of 1% of the mean stream wise velocity. Maxima of 2% are found near the solid walls and at the free surface (fig. 6).

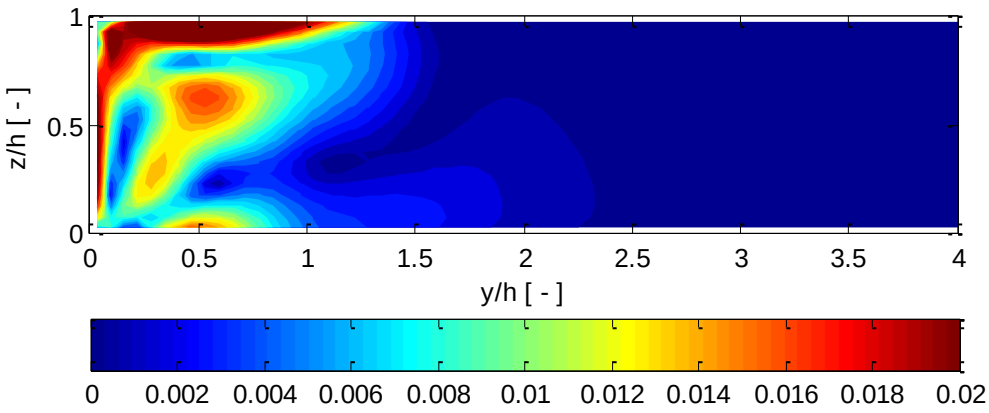


Fig. 6: Magnitude secondary currents, percentage of mean primary velocity, non-linear k-ε model.

Numerical discretization and computational costs

The linear k-ε closure as well as the non-linear closure are not very sensitive to the numerical discretization. Using default schemes (SWASH manual [28]) for time integration and space discretization gives stable results for both models. Since the non-linear terms of the Reynolds stresses are more dispersive than dissipative it enhances the numerical stability to choose an upwind scheme for the advection terms, when the non-linear closure is used. Using the first order upwind scheme instead of second order backward differencing shortens the spin-up time but does not influence the final stationary results. Time integration is of the second order using the default leap frog technique.

The vertical and transversal resolution is set to $\Delta z = \frac{H}{20}$ and $\Delta y = \frac{H}{16}$. Which is sufficient to solve for the secondary currents with a length scale $L \sim H$. RANS computations allow for a large $\frac{\Delta x}{\Delta z}$ ratio. Stable results are obtained with values up $\frac{\Delta x}{\Delta z} = 50$. In order to speed up the computations Δy was also set to $\frac{H}{8}$, resulting in the same flow pattern. To have a look at the origin of secondary currents the spatial distribution of different terms of the streamwise vorticity balance is analyzed. For this purpose a high resolution computation is made with $\Delta z = \frac{H}{40}$ and $\Delta y = \frac{H}{32}$.

For $\Delta z = \frac{H}{20}$ and $\Delta y = \frac{H}{16}$, a stationary situation was reached after 200 seconds of computational time which takes about four hours on one processor. The linear model with the same grid resolution converges a little faster. However, the linear model does not solve for the secondary currents and the fine grid resolution in transverse direction is of no importance in that sense.

4.1.2 Large Eddy Simulation

Since the original SWASH model has not been used before to perform a full 3D Large Eddy Simulation, the code has been adapted. This first test case with the adapted model consists of a standard open channel flow with homogeneous bed roughness. The aim of this test case is to predict the correct mean streamwise velocities and turbulence intensities. Model results are compared to the DNS results of Moser, Kim and Mansour [12]. Their model set-up consists of a closed channel with no slip conditions at both sides the domain. The LES results are compared with half their channel.

Model set-up
The flow geometry consists of a standard open channel with hydraulic smooth bottom conditions (fig. 18). The length and width of the domain are 120 and 8 times the water depth respectively. This geometry deviates from the geometry used for the RANS computations with the k-ε closure in the sense that no-slip conditions are used at both sides of the domain. The no-slip conditions are used to avoid the generation of secondary currents at the channel corners. At this stage these are not interesting since model results are compared to the DNS results of Moser, Kim and Mansour [12].

At the inflow boundary a logarithmic velocity profile is imposed. Random white noise fluctuations (see Paragraph 2.5) are added to trigger the growth of turbulence structures. The white noise fluctuations are 10% of the ensemble averaged velocity and are updated every 0.5 seconds. At the outflow boundary the water level is imposed by means of a weak reflective boundary. Initial flow velocities were set to zero.

The Reynolds number based on the friction velocity is $Re_* = 619$ and we consider subcritical flow. The results are obtained from the moment a turbulence situation has been established and are averaged over a period of 300 seconds.

The water depth is divided into 20 layers and the horizontal grid resolution is: $\Delta x = \Delta y = h/8$. This results in a total number of grid points of $N_{grid} = 1{,}228{,}800$.

The standard Smagoringsky subgrid model is used. The smagorsinky constant is set $c_{s0} = 0.065$ as suggested by Moin and Kim for open channel flow. Near the bottom c_{s0} is corrected by the van Driest damping function.

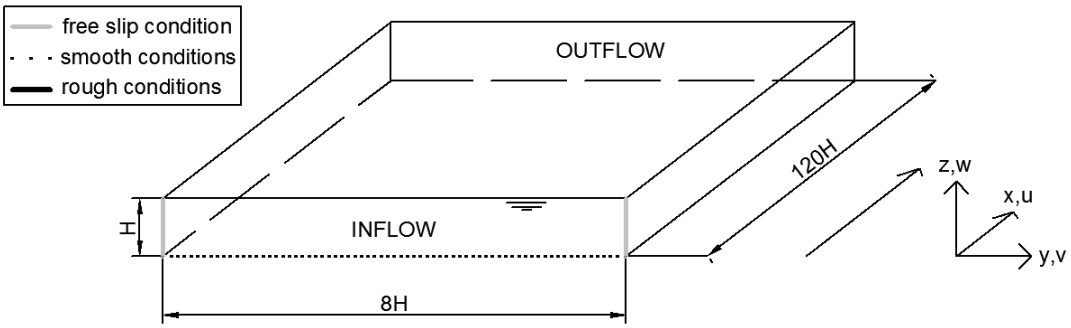


Fig. 18: Computational domain for LES with horizontal homogeneous roughness conditions.

Mean primary velocity and turbulence intensities

The mean primary velocity shows good agreement with the DNS data except for the layers closest to the bottom. Although the used vertical resolution is rather coarse and some deviation is expected, results are not satisfactory. Despite the relatively coarse vertical grid resolution used for LES by other authors (van Prooijen [22], van Balen [1]), their predictions of the mean velocity is rather good. There is a number of differences between the code used by van Prooijen en van Balen and the adapted SWASH model that can cause the different results. These numerical issues are discussed at the end of this Paragraph.

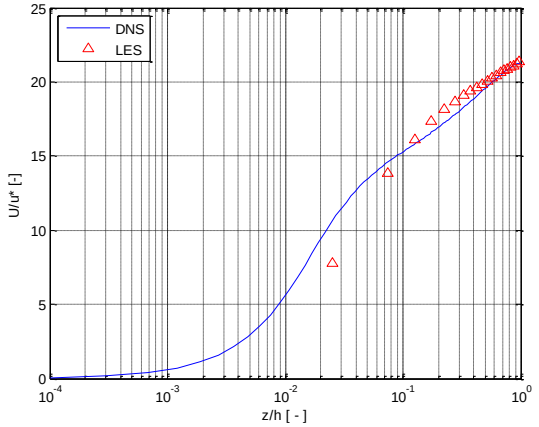


Fig. 19: Mean primary velocity, solid line: DNS of Moser, Kim and Mansour [12], triangles: LES with SWASH.

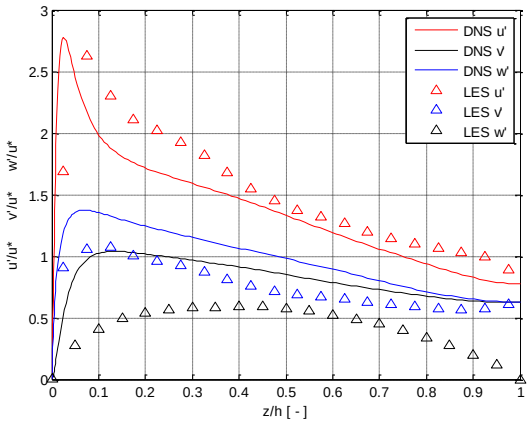


Fig. 20: Turbulence intensities, solid line: DNS of Moser, Kim and Mansour [12], triangles: LES with SWASH.

Beside the different codes both computations make use of different boundary conditions. These boundary conditions seem to be of large influence on the mean velocity profile and the obtained turbulence statistics. Both the DNS by Moser et. al. and the LES by van Prooijen uses periodic boundary conditions at the inflow, outflow and side walls. To start up the generation of turbulence structures their initial velocity field is disturbed by random white noise fluctuations. The use of periodic boundary conditions results after a certain spin-up time in a fully developed turbulence velocity

field at the inflow boundary. The velocity boundary used in the adapted SWASH model requires a certain part of the computational domain for turbulence structures to develop.

Talstra [30] investigated a shallow mixing layer geometry with a 3D LES and uses a comparable grid resolution to the present LES study. He found a large influence on the mixing layer development by imposing different velocity boundaries. Correct results are obtained when a fully developed turbulence velocity field is imposed. Random white noise fluctuations results in an under prediction of the turbulence intensities and mixing layer width in the near field. When no fluctuations are added and a stationary velocity is used, the mixing layer is absent. It is expected that the influence of upstream perturbations is even more present at the uniform channel flow modelled in this thesis because of the absence of a mean transverse velocity gradient.

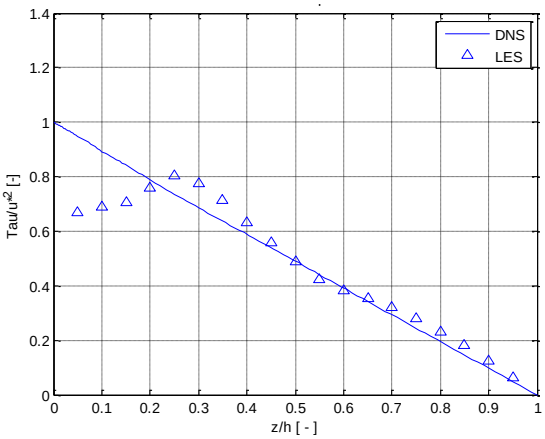


Fig. 21: Shear stress profile, $-\overline{u'w'} + \nu\frac{\partial \bar{u}}{\partial z}$, solid line: DNS of Moser, Kim and Mansour [12], triangles: LES with SWASH.

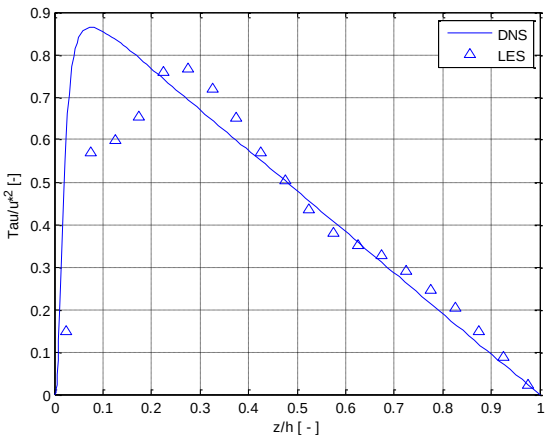


Fig. 22: primary Reynolds shear stress, $-\overline{u'w'}$, solid line: DNS of Moser, Kim and Mansour [12], triangles: LES with SWASH.

When an undisturbed velocity profile is used, there is no generation of turbulence structures and the velocity fluctuations are absent. With imposed fluctuations the generation of turbulence is triggered and the statistics of the turbulence intensities are presented in figure 20. Although there is some deviation from the DNS data the mean and transverse intensities are rather good. The vertical fluctuations however, are seriously underestimated. Besides the effects of the used inflow condition this seems to be related to some numerical dissipation, introduced by the use of upwind schemes (see next section). Near the free surface the vertical fluctuations are rapidly decreasing. This is expected since the model accounts for the free surface.

In figure 21 and 22 the mean primary shear stress profile and the primary Reynolds shear stress are plotted. These quantities show also significant deviation from the DNS data near the bottom. The main reasons for this deviation are, besides the use of non-

periodic boundary conditions, the coarse grid resolution and the amount of numerical dissipation. These topics are discussed below.

Numerical implementation
As described in Chapter 3 SWASH is a general purpose numerical tool. It can be used for problems where free-surface waves are involved as well as for open channel flows with hardly any free surface gradients. Different types of physical problems are dominated by different processes and corresponding terms in the governing equations. By discretization and integration of the equations there are a lot of options. Usually choices have to be made based on a balance between stability and accuracy. The numerical treatment of the governing equations is rather problem dependent. Consequently the wide range of physical problems SWASH can be used for asks for different discretization options.

Especially for computations using LES, sharp gradients are important since turbulence velocity fluctuations are rapidly varying. An important aspect in order to maintain the fluctuations is the space discretization of the advection terms. To this extend central differencing is most favourable. This numerical scheme introduces no numerical dissipation and maintains the velocity gradients. On the other hand it possibly leads to spurious oscillations when not enough physical dissipation is involved (turbulence and bottom friction are the main sources of physical dissipation). These non-physical oscillations can give rise to numerical instability.

In the table below the most favourable integration and discretization options are listed (provided that they are available in SWASH) for the use of LES.

Space Discretization	
Horizontal Advection terms	Central differences (2^{nd} order)
Vertical Advection terms	Central differences
Vertical pressure gradient	Central differences
Water depth	2^{nd} order flux-limiter
Time integration	
Pressure gradient	Crank-Nicholson (2^{nd} order)
Vertical advection terms	Crank-Nicholson

Table 1: preferable numerical discretization

In this preferable situation all terms are second order accurate in time as well as in space except for the subgrid stresses which are second order accurate in space and first order accurate in time.

Unfortunately, this discretization arrangement leads to unstable solutions for the homogeneous open channel flow geometry discussed above. The spurious oscillations introduced by central discretization of the advection terms are not sufficiently damped by bottom friction and subgrid turbulence. The Courant condition (4.1), restricting the

time step, was originally set at $Cr < 0.6$ but even for $Cr < 0.2$ the solution turns out to be unstable.

$$Cr = \Delta t \left(\sqrt{gh} + \sqrt{u^2 + v^2} \right) \sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}} \qquad 4.1.$$

An obvious choice to avoid the spurious oscillations involved with central differencing would be the use of upwind schemes for the discretization of the advection terms. Upwind schemes introduce numerical dissipation and do not produce spurious oscillations, which enhances the stability of the solution. Because the use of central differences is of great importance to maintain the turbulence velocity fluctuations, other options to add numerical diffusion will be discussed first.

Numerical dissipation

There are other options to introduce numerical dissipation, rather than the use of upwinding for the advection terms, in order to enhance the stability of the numerical solution. Numerical dissipation can be introduced by the:

- Time integration with respect to the vertical non-hydrostatic pressure gradient;
- Time integration of the vertical terms in the momentum equations and
- Discretization of water depth in the velocity points.

These options are favourable compared to the use of upwind schemes for the advection terms. The time integration with respect to the vertical non-hydrostatic pressure gradient seems to influence the stability most. Using a first order implicit Euler scheme instead of the second order accurate Crank-Nicholson scheme results in far more stable results, while turbulence fluctuations are unaffected.

Time integration of the vertical terms in the momentum equations can be done by the implicit Euler or the Crank-Nicholson scheme. For the discretization of the water depth first and higher order upwind schemes are available as well as flux-limiter schemes. Both integration of the vertical terms and the discretization of the water depth do not influence the stability significantly.

Although physical dissipation is present by means of bottom friction and the turbulence subgrid model and some numerical dissipation is added by using the implicit Euler scheme for the integration of the non-hydrostatic pressure gradient, the numerical solution is not stable. Therefore the use of upwind schemes is discussed next.

Instead of central differences the advection terms can be discretised using an upwind scheme. In this way numerical dissipation is implicitly added. This way of adding numerical diffusion is controversial since the effective filter is then very similar to the filter imposed by the Smagorinsky subgrid model (Sagout [24]). Even for seventh-order

accurate upwind schemes the total numerical dissipation is larger than the dissipation imposed by the subgrid model. Increasing the order of the upwind scheme does not necessarily increase the accuracy of the solution. Tafti [29] shows that when LES computations are carried out on a relatively coarse grid, results becomes worse with increasing order of the upwind scheme that have been used. Breuer [3] concluded that centred schemes perform best in comparison with schemes with a preferably direction. In addition to that he concluded that low numerical dissipation is more important that the formal accuracy of a certain scheme.

Despite the comments above, the effect of upwind schemes on the numerical solution is investigated. The first stable results were obtained with a horizontal grid spacing of $\Delta x = \Delta y = 2\Delta z = h/8$ and use of the second order accurate QUICK scheme for the horizontal advection terms. Upwind schemes introduce more diffusion when the grid spacing increases. For larger horizontal grid spacing the upwind schemes become too dissipative and no turbulence stuctures will develop. No significant difference is found when a third order CUI scheme is used.

For the simulation discussed in the first part of this Paragraph the discretization arrangement presented below is used.

Space Discretization	
Horizontal Advection terms	QUICK upwind scheme (second order)
Vertical Advection terms	Central differences (second order)
Vertical pressure gradient	Central differences
Water depth	2nd order flux-limiter
Time integration	
Pressure gradient	Implicit Euler (first order)
Vertical advection terms	Crank-Nicholson (second order)

Table 2: numerical discretization used for LES computations.

Another option to enhance the stability is to use the fraction step method instead of the pressure correction technique for the non-hydrostatic pressure term. Besides the additional adaptations that have to be made to the SWASH code the fraction step method requires use of the ILU preconditioner to be stable. The ILU preconditioner is computational very expensive for parallel computing and use of the much faster ILUD conditioner is preferred.

Preconditioning

Introducing non-hydrostatic pressure to the momentum equations requires a solution of the Poisson equation. The Poisson equation is iteratively solved with the linear BiCGSTAB solver in conjuction with a preconditioner. The RILU preconditioner is used which is a mixture of ILU and MILU (SWASH user manual). The mix of ILU and MILU is controlled by a weighting parameter $RILU = (1-\alpha)ILU + \alpha MILU$). It turns out that setting α to 1. (= MILU) improves the rate of convergence the most. For parallel computing the RILUD instead of the RILU preconditioner can be used. The first is

restricted to the diagonal of the matrix. For parallel runs this restriction reduces the number of iterations needed.

Solving the Poisson equation is an iterative process and a certain error needs to be accepted based upon efficiency and accuracy. The stopping criterion is based on the reduction of the residual (SWASH manual [28]). The iteration process is stopped when the ratio of the second norm of the residual ($\|r_m\|_2$) and the RHS of the initial residual ($\|b\|_2$) is less than a certain value: $\frac{\|r_m\|_2}{\|b\|_2} < \epsilon$. In general, if the accepted error decreases, the iteration process becomes more accurate. Unfortunately this requires more iteration steps and consequently more computational time. The accepted error is heavily problem dependent. For typical nearshore wave applications SWASH is used for the default value of $\epsilon = 10^{-2}$ results in a good balance between efficiency and accuracy. It turns out that for modelling turbulence fluctuations a significantly smaller error is accepted $\epsilon = \mathrm{O}(10^{-6})$. When ϵ is set at 10^{-2} the small velocity fluctuations imposed at the inflow boundary will be damped. Consequently there will be no growth of turbulence structures.

With respect to the turbulent kinetic energy balance (1.25) the underestimation of the wall normal fluctuations can be related to the lack of energy transfer from the axial fluctuations to the wall normal fluctuations. The energy transfer is essential for the correct prediction of the wall normal fluctuations since the production term of the TKE balance is limited to the axial direction. The transfer of energy is related to the pressure-velocity correlation ($\frac{1}{\rho_0} p' \frac{\partial \overline{u_i'}}{\partial x_i}$). The incorrect prediction of the turbulence fluctuations illustrates the importance of accurately predicting the pressure field.

Computational costs

A large number of simulations has been carried out to test the implementation of the subgrid model, boundary conditions and the use of different numerical discretization methods. In addition the total amount of time is limited. For practical reasons it was therefore decided that a single simulation has to be completed within a few days. This practical restriction limits the maximum numerical resolution that can be accomplished.

The computations run on a parallel computer cluster, using 24 processors for each simulation. Due to the restrictions above, the simulations used for the homogeneous channel flow are limited to a grid resolution of $\Delta x = \Delta y = \frac{h}{8} and\ \Delta z = \frac{h}{20}$ results in a total number of grid points of $N = 1{,}030{,}400$. The Courant number was set at $0.4 < C_r < 0.8$.

This resolution is relatively coarse and is expected to influence the vertical distribution of the mean flow properties and the turbulence intensities. Therefore a final run is carried out with a vertical resolution of $\Delta z = \frac{h}{32}$. Unfortunately the solution appears to be unstable. The instability is caused by the use of the previously discussed ILUD

preconditioner. When the ratio of $\frac{\Delta x}{\Delta z}$ increases the matrix becomes less diagonal dominant. Since the ILUD preconditioner is restricted to the diagonal of the matrix this dominancy is of great importance.

Alternatively the ILU preconditioner can be used. Compared to ILUD this preconditioner needs more iteration steps to meet the same accuracy. Furthermore an iteration step with ILU is more computational expensive. Finally, the intended simulation becomes too computational expensive.

As discussed in previous sections, most LES codes make use of periodic boundary conditions when simple geometries are modelled. There are a number of advantages involved with the use of periodic boundary conditions. From numerical point of view the most interesting one is that a Fourier solver can be used instead of the BiCGstab solver to solve the pressure Poisson equation. The BiCGstab solver is very time consuming compared to a Fourier solver. Since SWASH is a general purpose numerical model that is also used for complex geometries the more computational expensive BiCGstab solver is implemented.

4.1.3 Comparsion k-ε model and Large Eddy Simulation

In section 4.1.1 and 4.1.2 computational results are presented from the RANS model and LES, respectively. In this Section the behavior of both models is compared with respect to the flow features as well as the computational costs and robustness.

Flow features
In the region where solid side walls are of no importance, mean primary velocity profiles are represented very well by the RANS model, both with the linear and non-linear k-epsilon closure. LES, on the other hand, underestimates the mean primary velocity as well as the turbulence intensities in the near bottom region.
In the region close to solid side walls, two counter rotating secondary currents are expected. Secondary currents, represented by the non-linear k-ε model, are compared to measurements of Nezu and Rodi [17] and show good agreement. The disturbed primary velocity distribution as described by Nezu and Nagakawa [16] is also well represented. RANS computations with the linear k-ε closure neither show the secondary currents nor the disturbed velocity distribution.
Since the model set-up, used for the LES, consists of no-slip boundary conditions at the side walls, no comparison between the LES and the RANS computations can be made in the region close to the wall.

Computational costs and resolution
An important difference between LES and RANS computations is the grid schematization. RANS computations allow for a much larger $\frac{\Delta x}{\Delta z}$ ratio than LES. For the LES computations a maximum value of $\frac{\Delta x}{\Delta z} = 4$ is used, where RANS computations give

good results with values up to 50. Since the time step is related to the horizontal grid spacing by the Courant condition, the time step used for RANS computations can be much larger than the time step used for LES. Due to the coarse grid spacing in streamwise direction and the relatively large time step, the RANS computations are less computational expensive. Where LES runs for this study took at least two days on a parallel computer cluster with 20 active nodes, RANS computations were completed in four hours, using one single core with one active node. The linear k-ε closure converges somewhat faster than the non-linear closure.

Robustness

LES results are very sensitive to the discretization of the advection terms: central differencing give rise spurious oscillations, where upwinding introduces numerical dissipation. In contrast to LES (4.1.2), the RANS model is not very sensitive to the numerical discretization. Since the RANS equations reach for a stationary flow field, instead of the instantaneously changing flow regime of the filtered Navier-Stokes equations, this can be expected. When the non-linear closure is used for the RANS computations, it enhances the numerical stability to choose a first order upwind scheme for the advection terms. Using the first order upwind scheme instead of second order backward differencing shortens the spin-up time but does not influence the final stationary results.

4.2 Open channel flow with non-homogeneous roughness conditions

In Paragraph 4.1 an open channel flow was investigated with spatially uniform roughness conditions. In this section results are presented of an open channel with varying roughness conditions. Paragraph 4.2.1 and 4.2.2 show the results of the k-ε model and the LES respectively. In both Paragraphs the numerics are discussed as long as there are any differences between the non-homogeneous and the homogeneous situation.

4.2.1 non-linear k-ε model

In section 4.1.1 the linear and the non-linear k-ε model are used to simulate a homogenous open channel flow. Both models are capable of representing the correct mean velocity profile when the latter is not disturbed by the effects of turbulence anisotropy. When the presence of solid walls gives rise to turbulence anisotropy, the linear model fails and is not able to model the secondary currents induced by the anisotropy. Since the expected secondary currents involved with non-homogeneous roughness conditions are also induced by turbulence anisotropy, the computations were exclusively made with the non-linear model.

The model geometry consist of an open channel with hydraulically rough side wall conditions. The bottom is in transversal direction divided into a number of smooth and rough sections (fig. 23). The smooth and rough bed strips are both of the same size and equal to the water depth. At the smooth sections hydraulically smooth conditions are imposed. At the rough sections the log law for hydraulically rough conditions is used, the Nikuradse roughness height was set to 0.019 m. The results are obtained from the stationary situation, when the wall shear stress balances the mean pressure gradient. The Reynolds number based on the friction velocity is $Re_* = 856$ and we consider subcritical flow.

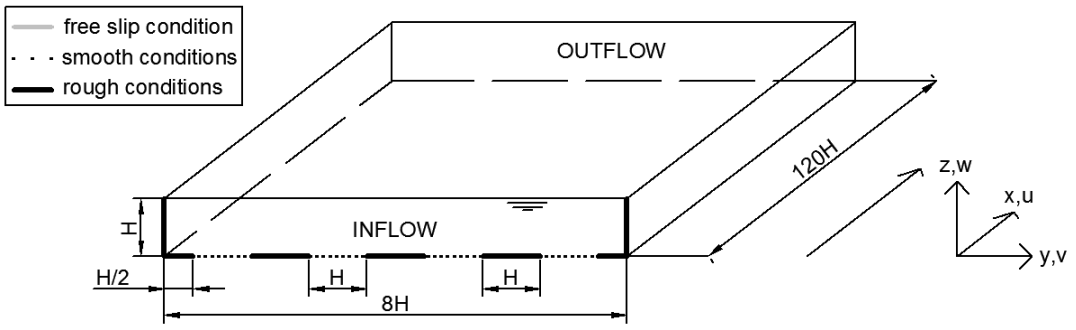


Fig. 23: Computational domain for RANS computations with non-homogeneous roughness conditions.

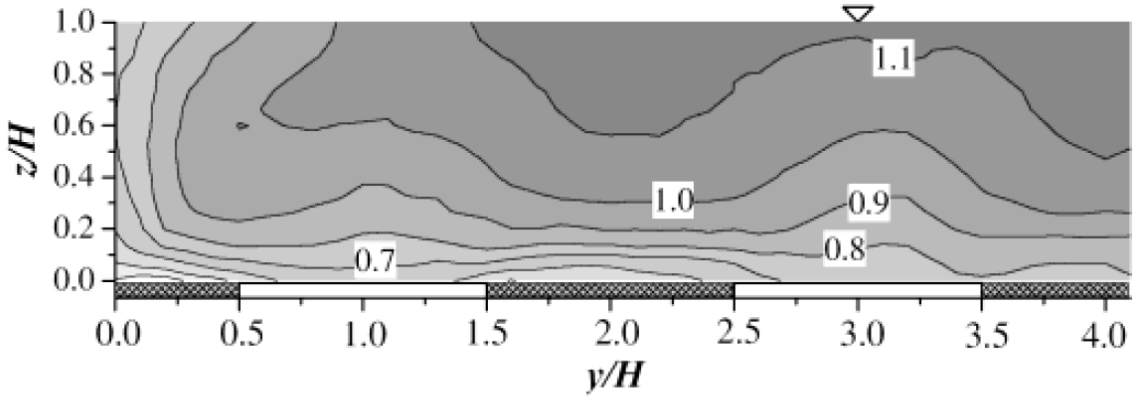


Fig. 24: Primary velocity distribution, velocity divided by spatially averaged velocity, Muller and Studerus [13].

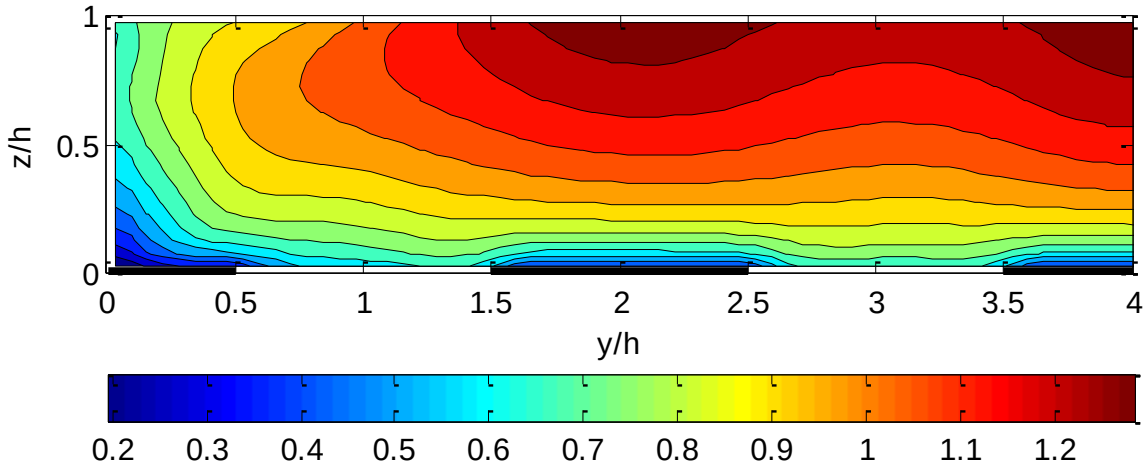


Fig. 25: Primary velocity distribution, velocity divided by spatially averaged velocity, non-linear k-ε model.

Figure 24 and 25 show the mean velocity distribution computed by the non-linear k-ε model as well as the measured distribution by Wang and Cheng [32]. Flow velocities near the bed are relatively low at rough sections and increases at the smooth sections due to the varying bed shear stress.

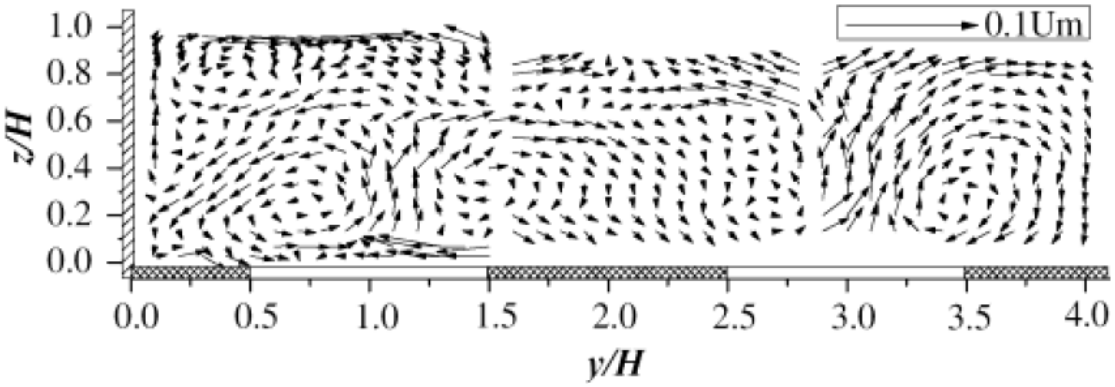


Fig. 26: Secondary currents on smooth and rough sections, v-w vector, Wang and Cheng [32].

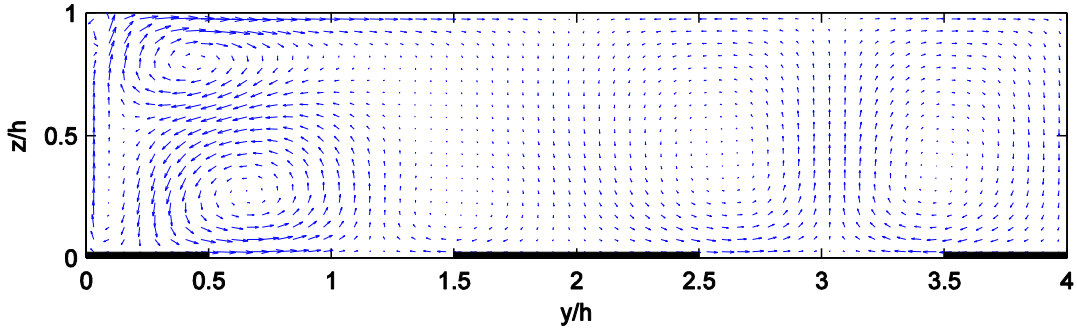


Fig. 27: Secondary currents on smooth and rough sections, v-w vector, non-linear k-ε model.

The secondary currents in the corner region are slightly stronger compared to the currents observed in homogeneous open channel flow. The use of different roughness conditions is expected to be responsible for the small difference. At the side wall and the bottom part most close to the wall hydraulically rough conditions imposed, whereas smooth conditions are imposed for the homogeneous channel computations. Naot and Rodi [14] investigated the effects of different roughness conditions on the magnitude of the secondary currents. The production of slightly stronger currents is found when hydraulically rough conditions are imposed but overall the effect on primary and secondary flow is very small.

At a certain distance ($y > 2h$) the presence of the side wall does not affect the primary and secondary flow anymore and the varying bed shear stress becomes important. In figure 30 the region in between $y > 2h$ and the channel center in showed. The upward motion is concentrated at the smooth sections and downward motion is observed at the rough sections. This direction of the secondary currents was expected from previous measurements and qualitative analytical descriptions by different authors made in the past (see Chapter 1). The maximum magnitude of the currents is just above 2% of the mean velocity which is in good agreement with the measurements of Wang and Cheng [32] (fig. 27). Nezu and Nagakawa [16] measured a slightly higher value of 2.5%.

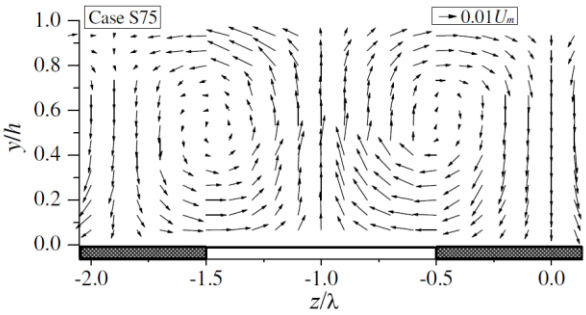


Fig. 27: Secondary currents, v-w vector, Wang and Cheng [32]

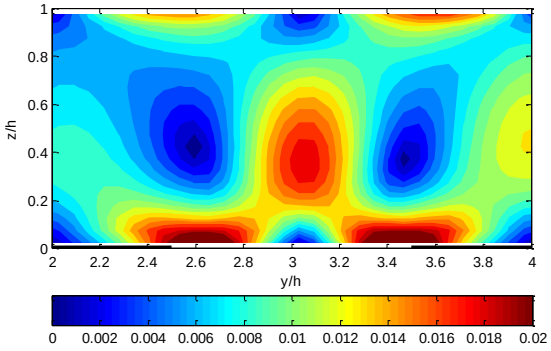


Fig. 28: Magnitude of secondary currents, percentage of mean primary velocity, non-linear k-ε model

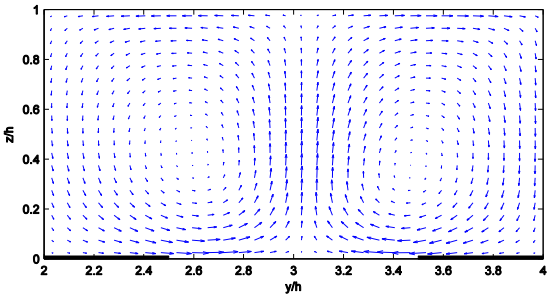


Fig. 30: Secondary currents, v-w vector, non-linear k-ε model.

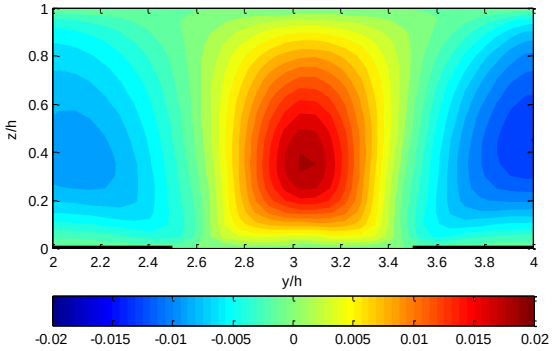


Fig. 31: Mean vertical velocity, percentage of mean primary velocity, non-linear k-ε model.

Reynolds normal stresses in streamwise ($\overline{u'u'}$) and spanwise direction ($\overline{u'v'}$) are compared with measurement results of Muller and Studerus [13]. The concentration of the normal stresses above the rough bed section is well represented. The primary shear stress shows negative and positive values at the left and right part of the rough bed section respectively. The values show good agreement with the measurements. The large primary shear stress concentration at the left interface between the rough and smooth section is also measured by Muller and Studerus [13]. The strong negative values at the right interface do not show up in the measurements. Model simulations by Choi et. al [4] with a non-linear k-ε model show a similar concentration of negative shear stress at this interface.

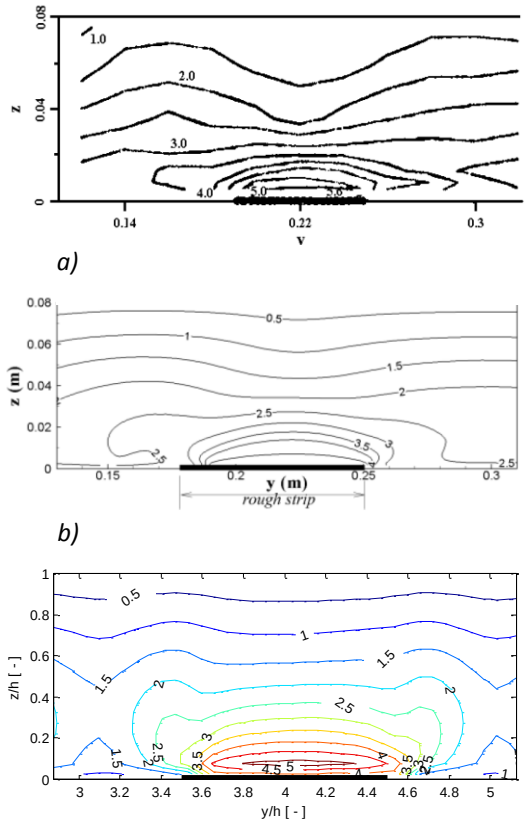


Fig. 32: dimensionless Reynolds primary normal stresses $\frac{u'u'}{{u_*}^2}$, a) measurements by Muller and Studerus [13], b) model results by Choi et al. [4] and results non-linear k-ε model.

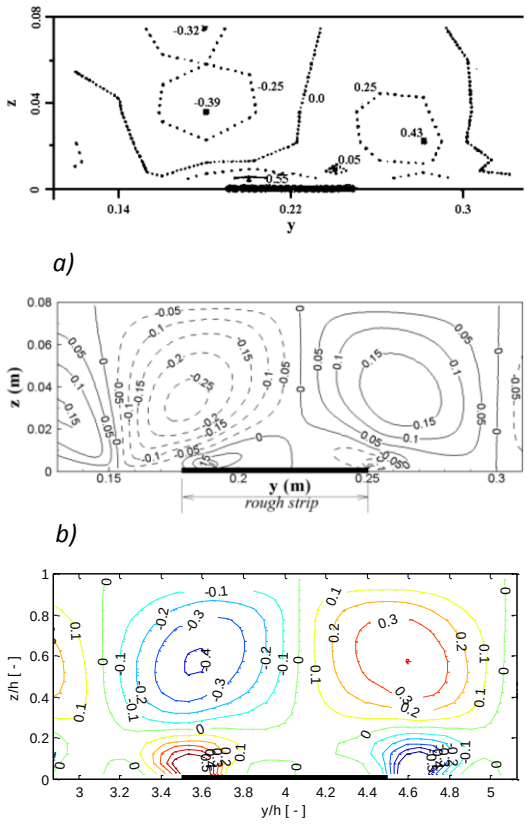


Fig. 33: dimensionless Reynolds cross plane shear stresses $\frac{u'v'}{{u_*}^2}$, a) measurements by Muller and Studerus [13], b) model results by Choi et al. [4] and results non-linear k-ε model.

The heterogeneity of the bed roughness influences the distribution of the primary Reynolds shear stress (fig. 34) and the effective friction that is encountered by the flow. As expected the shear stress shows large values at the rough section and relative low values at the smooth sections. Jarquin (2007) showed that the effective friction due to roughness heterogeneity increases with 20% with respect to the averaged value. With the non-linear k-ε model an increase of 15% is found.

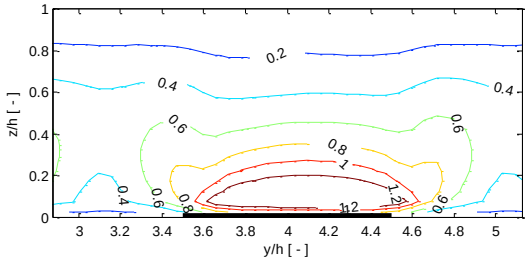


Fig. 34: dimensionless primary Reynolds shear stress ($-\frac{u'w'}{{u_*}^2}$) distribution, non-linear k-ε model.

As discussed in the first Chapter the observed secondary currents involved with spanwise varying bottom shear stress are secondary currents of Prandtl's second kind. Secondary currents of Prandtl's second kind are driven by the anisotropy of turbulence. The streamwise vorticity balance covers the non-homogeneity and anisotropy of the turbulent stresses and is often used to show the origin of these currents.

$$\overline{u_j}\frac{\partial\overline{\omega_x}}{\partial x_j} = +\nu\frac{\partial^2\overline{\omega_x}}{\partial {x_j}^2} + \underbrace{\frac{\partial^2}{\partial y\partial z}\left(\overline{v'v'} - \overline{w'w'}\right)}_{A} - \underbrace{\left(\frac{\partial^2}{\partial y^2} - \frac{\partial^2}{\partial z^2}\right)\overline{v'w'}}_{B} \qquad 4.2.$$

Term A is the secondary current generation term and is order of magnitude larger than term B. Figure 35 shows the strong positive and negative levels of this term at the interface between the smooth and rough sections. The large absolute values of this term at the interfaces suggest that the secondary currents originate at the interface between a smooth and rough section.

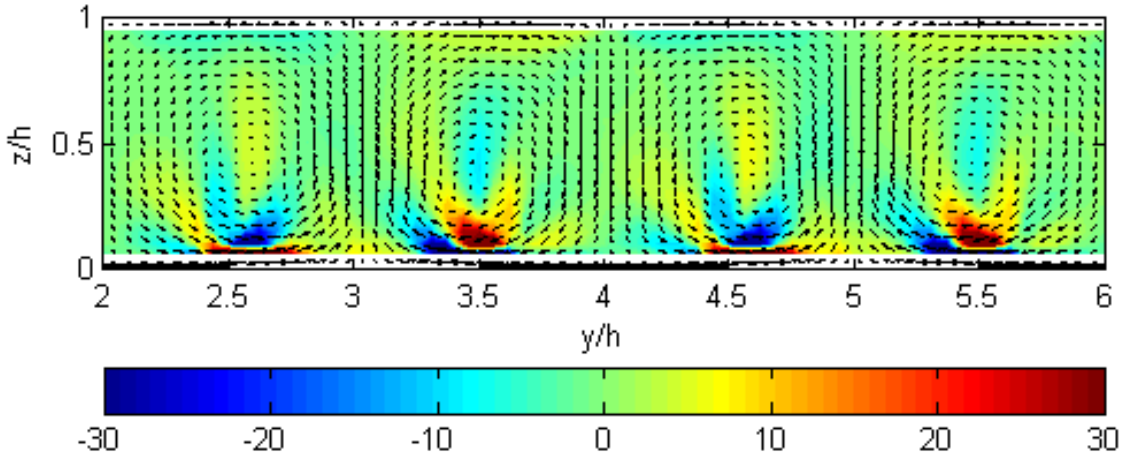


Fig. 35: Distribution of generation term of streamwise vorticity $\frac{\partial^2}{\partial y\partial z}\left(\overline{v'v'} - \overline{w'w'}\right)\frac{H^2}{{u_*}^2}$*, non-linear k-ε model*

Numerical discretization and computational costs

As described in section 4.1.1 the solution of the non-linear k-ε model is not very sensitive to the numerical discretization. The same discretization and integration methods are used as in the homogeneous case. The Courant number was set at $0.1 < C_r < 0.5$. A stationary flow was reached after 200 seconds. The total computational time for one single simulation was four hours.

4.2.2 Large Eddy Simulation

In section 4.1.2 the LES results of a homogeneous channel are presented. Deviations from DNS data were found especially in the region close to the bottom. The mean primary velocity as well as the turbulence intensities is seriously enderestimated. Reasons for the underestimation are a combination of the boundary conditions that are used, the amount of numerical dissipation that is involved and the maximum grid resolution that can be used with respect to the available amount of time. These issues have been addressed in section 4.1.2.

In this Paragraph the LES results are presented of an open channel flow with non-homogeneous roughness conditions. Following to the results of the RANS computation presented in the previous section and measurements by other authors, turbulence driven secondary currents are expected. These secondary currents have to be generated by the turbulence anisotropy near the bottom. Since the model set-up that was used for the LES of homogeneous channel flow does not show satisfactory results near the bottom, the use of this set up for the non-homogenous channel flow is rather arbitrary. However, due to limited amount of time that was availiable, no further sourcecode modifications were made.

Model set-up
The channel geometry is comparable to the geometry used for the LES with homogenious roughness conditions. The length and width of the domain are 120 and 8 times the water depth respectively. The bottom is in transversal direction divided into a number of smooth and rough sections (fig. 36). The smooth and rough bed strips are both of the same size and equal to the water depth. At rough sections, the Nikuradse roughness height was set to 0.019 m. Roughness conditions are imposed by the Schumann wall model (see Appendix B).

At the inflow boundary a logarithmic velocity profile is used with superimposed random white noise fluctuations. At the outflow boundary the water level is prescribed by means of a weak reflective boundary. Initial flow velocities were set to zero.

The water depth is divided into 16 layers and the horizontal grid resolution in streamwise and transverse direction is, respectively: $\Delta x = \frac{h}{8}\ and\ \Delta y = \frac{h}{10}$. This results in a total number of grid points of $N_{grid} = 1{,}228{,}800$.

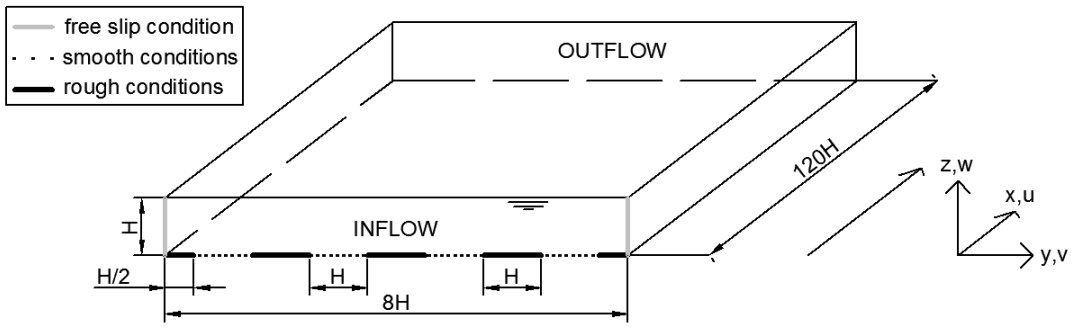


Fig. 36: Computational domain for LES with horizontal non-homogeneous roughness conditions.

The primary velocity distribution is showed in figure 37. Close to the bottom the velocities are relatively small at the rough sections compared to the smooth sections. This is expected since the bed shear stress acting on the near bottom layer is significantly higher at rough sections. However, due to the momentum exchange by the expected secondary currents the primary velocity near the free surface needs to be higher above the rough sections. The LES results do not represent this primary velocity distribution correctly. Some weak circulations are present but the expected currents related to roughness heterogeneity are absent (fig. 38).

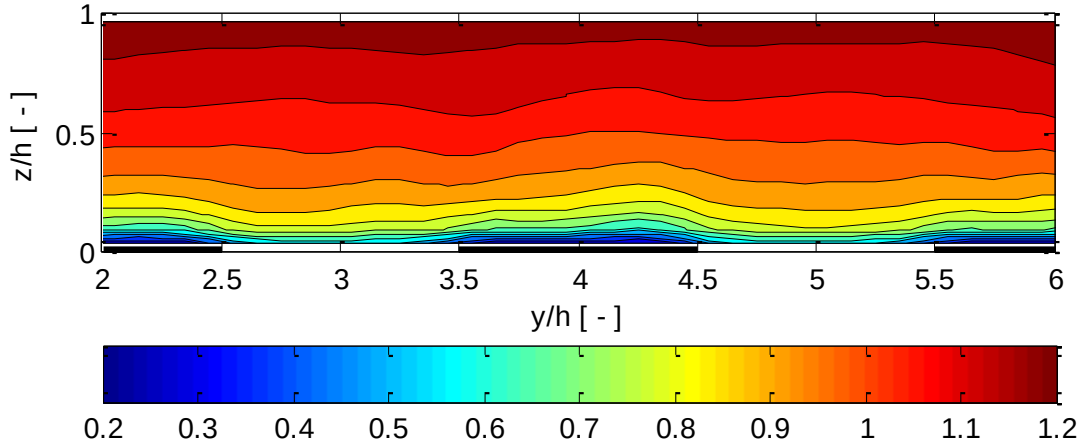


Fig. 37: Primary velocity distribution, velocity divided by spatially averaged velocity, LES.

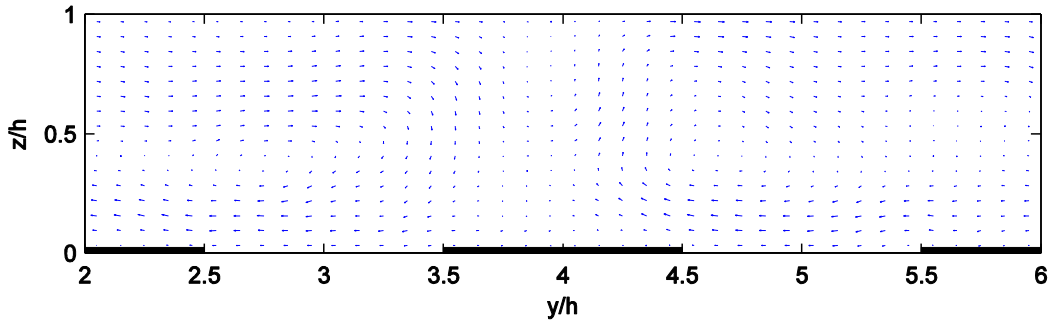


Fig. 38: Secondary currents, v-w vector, LES.

As concluded from laboratory experiments by Vermaas [31], the development of secondary currents can be scaled to the water depth. Within 80 times the water depth, the secondary currents will develop to full strength. Although the length of the model domain is sufficient in that sense (120 times the water depth), secondary currents were not generated. Since the secondary currents are generated by the strong turbulence anisotropy near the bottom, turbulence intensities need to be very well represented in that region. The absence of the circulations is related to the weak performance of the LES code with respect to tubulence statistics near the bottom.

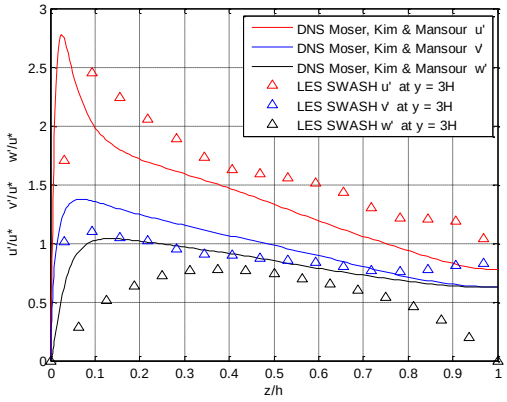


Fig. 39: Turbulence intensities at smooth section compared to DNS results of Moser, Kim and Mansour [12] of horizontal homogeneous channel flow. Solid line: DNS, triangles: LES results with SWASH.

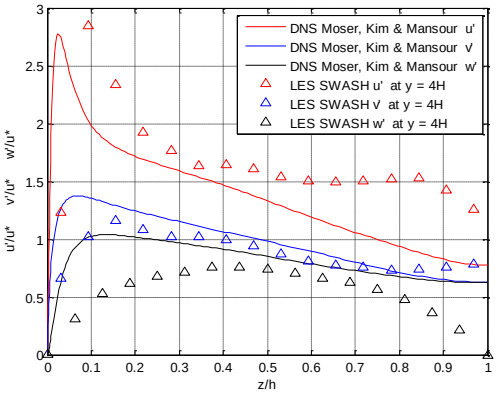


Fig. 40: Turbulence intensities at rough section compared to DNS results of Moser, Kim and Mansour [12] of horizontal homogeneous channel flow. Solid line: DNS, triangles: LES results with SWASH.

As discussed in previous sections secondary currents are generated in the near wall region. Since the used computational grid is rather coarse the near wall dynamics are not solved on the computational grid but modelled by a Schumann-Grötzbach wall model [24]. The wall model represents the instantaneous wall shear stress based on the logarithmic velocity law (Section 2.5). The use of the logarithmic wall stems from the assumption of a horizontal homogeneous flow (Section 1.2). Obviously, an error is introduced by applying the velocity law to geometries with horizontaly varying bed roughness. Although the logartitmic law is, strictly spoken, not valid for horizontal heterogeneous flow, it is not expected that the use of the Schumann-Grotzbach wall model is the main cause for the absence of secondary currents.

The performance of the LES code (near the bottom) depends on to the of use of non-periodic boundary conditions, the coarse grid resolution, the size of the computational domain and the amount of numerical dissipation that is involved. These issues are comparable for homogeneous and non-homogeneous channel flow and were discussed in section 4.1.2.

4.2.3 Comparison k-ε model and Large Eddy Simulation

In section 4.2.1 and 4.2.2 results are presented from the RANS model and LES, respectively. In this Section the behavior of both models is compared with respect to the flow features as well as the computational costs and robustness.

Flow features
The model geometry of the non-homogeneous bed case consists of an open channel flow with a number of parallel smooth to rough bed strips and hydraulically rough side wall conditions. Since the linear k-ε closure does not solve for turbulence anisotropy, simulations were restricted to the RANS model with the non-linear closure and LES. A disturbed primary velocity field and secondary currents are expected above the smooth to rough bed sections. In the corner regions, the flow is expected to behave similar to the homogeneous flow case (Section 4.1).

The RANS model with the non-linear k-ε closure represents the mean primary velocity distribution very well. Secondary currents show also good agreement with measurements by Wang and Cheng [32] and Muller and Studerus [13] and model simulations by Choi et al. [4]. Upward flow was concentrated at smooth section and downward flow at rough sections. The maximum magnitude of the currents is just above 2% of the mean velocity. Reynolds normal stresses and cross plane shear stresses are compared to model simulations by Choi et al. [4] and are well represented. In contrast to the RANS model, LES do neither show the correct primary velocity distribution, nor the expected secondary currents. The weak performance of the LES code is related to the amount of numerical dissipation, the not fully developed velocity field imposed at inflow condition and the coarse grid resolution.

Computational costs and Robustness
LES is relatively computational expensive compared to the RANS model. LES results are also very sensitive to the numerical discretization of the momentum equations. However, there a no differences between the homogeneous case discussed in Chapter 4.1 and the non-homogeneous case presented in this Chapter. For a comparison between the RANS model and LES with respect to the computational costs, grid resolution and robustness, see Section 4.1.3.

5. Conclusions and recommendations

The objective of this study is to model open channel flow, involving non-homogeneous roughness conditions, with the SWASH wave-flow model for free surface flow. Turbulence anisotropy plays a major role in non-homogenous channel flow, where turbulence induced secondary currents influencing the primary velocity distribution. The SWASH wave-flow model, mainly used to predict the transformation of surface waves from offshore to the beach, has been successfully adapted in order to model turbulence anisotropy. Simulations of open channel flows have been made in which secondary circulations driven by turbulence anisotropy are very well represented. With the implementation of the non-linear k-ε turbulence closure the range of applications SWASH can be used for, is extended with flow geometries where turbulence anisotropy cannot be neglected.

Before more detailed results on modeling turbulence anisotropy are presented, the two main research questions of this study are discussed. The first question relates to turbulence modelling:

- In which way does SWASH need to be changed or extended, in order to model the anisotropy of turbulence?

SWASH is a non-hydrostatic wave-flow model and open channel flows can be modelled with this code by solving the RANS equations in conjunction with a standard k-ε closure. However, the standard k-ε closure [11] does not take turbulence anisotropy into account. In the present study the SWASH model has been extended with the non-linear k-ε closure proposed by Speziale [25]. The non-linear closure is, applicable to flows where turbulence anisotropy influences the mean flow characteristics. In Section 5.1 conclusions on RANS modelling are drawn.

Since the anisotropic character of turbulence is related to the macro scales, LES can be used to model the turbulence driven secondary currents as well. In order to perform a LES with the SWASH model, a 3D subgrid model is implemented. Although the LES technique is in principle capable of solving turbulence induced secondary currents, no reliable results are obtained from the present LES study. In Section 5.2 conclusions on LES are presented. Section 5.3 consists of a comparison between RANS modelling and LES.

The second main question relates to the generation mechanism of secondary currents:

- How does turbulence anisotropy relate to the generation of secondary currents, in open channel flows involving heterogeneous roughness conditions?

Both secondary currents occurring at channel corners and at smooth to rough bed transitions are induced by anisotropy and non-homogeneity of turbulence. It is from the

isotropic character of the linear k-ε closure that RANS simulations using this turbulence closure do not show any secondary currents. Results of computations with the non-linear closure, on the other hand, show anisotropic and non-homogenous distributions of the Reynolds normal and shear stresses. Consequently, secondary currents are generated. To analyze the driving mechanism of the secondary currents at smooth to rough bed transitions, the streamwise vorticity balance is examined. Secondary currents seem to originate at the transition between smooth and rough bottom sections where the anisotropy of Reynolds normal stresses is dominant.

5.1 Conclusions on RANS modelling

RANS simulations are made with both the linear and the non-linear k-ε closure. The closure constants for the non-linear model were set to the suggested values by Speziale [25] for all model runs.

Usually, turbulence anisotropy is concentrated at the channel corners. Turbulence driven secondary currents (Prandtl's second kind) are generated and influence the mean primary velocity distribution in the corner region. The two counter rotating secondary currents in the corner region, as well as the primary velocity dip, described by Nezu and Nagakawa [15], are very well represented by the SWASH model with the non-linear k-ε closure. As expected, the standard linear closure represents neither the secondary currents, nor the correct primary velocity distribution.

In addition to the horizontal homogenous flow, an open channel flow was calculated with parallel smooth to rough bed strips and hydraulically rough side walls. Computational results of the SWASH model with the non-linear closure show good agreement with measurement results of Muller and Studerus [13], Nezu and Rodi [17] and Wang and Cheng [32]. Secondary currents with a magnitude of 2% of the mean primary velocity are generated. Upward flow was concentrated at smooth sections and downward flow at rough sections. Also Reynolds primary normal stresses and spanwise shear stresses distributions show good agreement to measurement results.

5.2 Conclusions on Large Eddy Simulation

In order to perform a LES with the SWASH model, the standard Smagorinsky subgrid model, in full three dimensions, is implemented. Since the used computational grid is rather coarse, the Schumann-Grötzbach wall model is implemented to model the instantaneous bed shear stress.

LES results for horizontal uniform flow are validated with DNS data of Moser, Kim and Mansour [12]. Especially near the bed, the LES results deviate from the DNS data. The mean velocity as well as the transverse and vertical turbulence intensities is seriously underestimated. The deviation from the DNS data appears to be related to the use of non-periodic boundary conditions, the coarse grid resolution, the size of the

computational domain and the amount of numerical dissipation. These issues are briefly discussed below.

For the present LES, non-periodic boundary conditions are used since periodic boundaries are at the moment not implemented in SWASH. At the inflow boundary a logarithmic velocity profile is used with superimposed random white noise fluctuations. At the outflow boundary the water level is prescribed by means of a weak reflective boundary. Since the computational domain is limited due to the available amount of computational time, this set of boundary conditions limits the growth of a well-developed turbulence velocity field.

Numerical dissipation is implicitly added by the use of upwind schemes for the discretization of the advection terms. The relatively coarse resolution enhances this amount of numerical dissipation. Central discretization of the advection terms is preferable for LES and most often used in combination with the Adams-Bashfort multistep method for time integration. However, the leap frog scheme for time integration, which is used by SWASH, turns out to be unstable in combination with central space discretization of the advection terms.

LES results deviate from DNS data for horizontal uniform open channel flow in the region close to the bottom. Since it is the bottom region where secondary currents are generated, the use of the present LES code for problems involving heterogeneous roughness is uncertain. In contrast to the RANS computations, no reliable results with respect to the generation of turbulence driven secondary currents are obtained from the present LES.

5.3 Comparison between RANS modelling and Large Eddy Simulation

For the horizontal homogeneous case, the mean primary velocity profiles are represented very well by the RANS model, both with the linear and non-linear k-ε closure. LES, on the other hand, underestimates the mean primary velocity as well as the turbulence intensities in the near bottom region.

Simulations of the non-homogeneous test case were restricted to the RANS model with the non-linear closure and LES. The RANS model with the non-linear k-ε closure represents both the disturbed mean primary velocity distribution and the secondary currents very well. Reynolds normal stresses and cross plane shear stresses are compared to model simulations by Choi et al. [4] and show also good agreement. In contrast to the RANS model, LES do neither show the correct primary velocity distribution, nor the expected secondary currents.

An important difference between LES and RANS computations is the grid schematization. RANS computations allow for a much larger $\frac{\Delta x}{\Delta z}$ ratio than LES. For the LES computations a maximum value of $\frac{\Delta x}{\Delta z} = 4$ is used, where RANS computations give

good results with values up to 50. Consequently, RANS computations are less computational expensive.

LES results are very sensitive to the discretization of the advection terms: central differencing give rise spurious oscillations, where upwinding introduces numerical dissipation. In contrast to LES, the RANS model is not very sensitive to the numerical discretization.

The RANS model with the non-linear k-ε closure show good results with respect to the mean velocity as well as the Reynolds shear and normal stresses distribution. Although there are a number of closure constants involved, additional tuning of these coefficients was not necessary for this study: both the homogenous and non-homogenous test case were simulated successfully using the standard values proposed by Speziale [25]. With its low computational costs and robustness, the non-linear k-ε model appears to be a useful extension to the SWASH wave-flow model.

5.4 Recommendations

During this study a number of questions arise and some of them are left unanswered. In this Section further research requirements are listed. Open channel flows were successfully simulated with the non-linear k-ε model and the first recommendation is related to validation of this k-ε model with other test cases. The other recommendations are related to LES: before the SWASH wave-flow can be used to run a successful 3D LES, further investigation on the discretization methods is needed and additional adaptations to the code have to be made.

- Validation of non-linear k-ε model with other test cases.

The results of the non-linear k-ε model for open channel flow with heterogeneous roughness conditions show good agreement with data provided by several laboratory experiments. It would be interesting to show its behavior in other flow geometries. For instance, compound channel flows of different flood plain depth or flows over sand ridges.

- Investigate the use of upwind schemes for LES.

It was not expected that space discretization and time integration of the filtered momentum equations are of such a great importance with respect to LES. Where most LES codes use central space discretization for the advection terms, the adapted SWASH model uses 2^{nd} order accurate upwind schemes for stability reasons. The use of upwind schemes is controversial since the effective filter is then very similar to the filter imposed by the Smagorinsky subgrid model (Sagout [24]). The numerical dissipation is expected to influence the turbulence intensities significantly.

- Implementation of periodic boundary conditions in the SWASH model

The use of non-periodic boundary conditions requires prescription of a (disturbed) velocity profile at the inflow boundary. Since it takes some time for a turbulent flow field to develop, a relatively large computational domain is needed in this case. When periodic boundary conditions are used the inflow conditions are better and consequently, the computational domain can be significantly smaller.

- Implementation of Fourier solver in the SWASH model.

In addition to the previous recommendation, the use of a Fourier solver for the Poisson problem is recommended. A Fourier solver is relatively fast compared to the time consuming BiCGStab solver. Since a Fourier solver can be used in conjunction with periodic boundary conditions only, these conditions need to be implemented first.

- Take non-equidistant grid spacing into account by discretization of the subgrid model.

LES computations were carried out using an equidistant grid in all three directions. In vertical direction however, it is likely to have a more fine resolution in the region close to the bottom. The discretization of the subgrid model needs to take the non-equidistance into account when velocity derivatives are computed.

Notation

Greek symbols

δ	Kronecker delta
Δt	time step
Δx	grid spacing in primary direction
Δy	grid spacing in transverse direction
Δz	grid spacing in vertical direction
ε	dissipation of turbulent kinetic energy
ϵ	permutation tensor
ζ	surface elevation
η	Kolmogorov length scale
κ	von Karman constant
ν	kinematic viscosity
ν_{SGS}	subgrid viscosity
ν_t	eddy viscosity
ρ	density
ρ_0	reference density
σ_i	closure constant for standard k-ε model
τ_s	wall shear stress
τ_t	primary shear stress
ω	vorticity
Ω_{ij}	rotation rate tensor

Roman symbols

C_i	closure constant for k-ε model (Nisizima)
$C_{\tau i}$	closure constant for k-ε model (Speziale)
c_s	Smagorinsky constant
D	depth
e	Turbulent Kinetic Energy (TKE)
g	gravity
k	turbulent kinetic energy
k_s	Nikuradse roughness height
p	non-hydrostatic pressure
P	pressure
Re	Reynolds number based on mean primary velocity and water depth.
Re_*	Reynolds number based on friction velocity and water depth.
S_{ij}	strain rate tensor
u	primary velocity
u_i	velocity in direction i.
u_*	friction velocity

v	transverse velocity
w	vertical velocity
x	primary coordinate
x_i	coordinate in direction i.
y	transverse coordinate
z	vertical coordinate

Operators

$\langle \dots \rangle$	ensemble averaging
$\overline{\dots}$	reynolds averaging
$\dots'$	fluctuation stemming from Reynolds or LES decomposition

The Einstein summation convention holds for roman symbols. Greek symbols are used for noncontracted subscripts.

Literature

[1] Balen, W. van. 2003. Curved open channel flows. Delft, PhD Thesis.

[2] Boersma, B.J. 2007. An introduction to DNS/LES simulation. www.adh.tudelft.nl/~bendiks/

[3] Breuer, M. 1998. Large Eddy Simulation of subcritical flow past a circular cylinder: numerical and modeling aspects. J. Numer. Meth. Fluids 6 (842-870).

[4] Choi, S.-U., Park, M., & Kang, H. 2008. Non-Linear k-ε Model for Open-Channel flows. J. of Coastal Researsh 52 (33-40).

[5] Demuren, A.O., & Rodi, W. Calculation of turbulence driven secondary motion in non-circular ducts. J. Fluid Mech. 172 (63-92).

[6] Falcomer, L., & Armenio, V. 2002. Large-eddy simulation of secondary flow over longitudinally ridged walls. J. of Turbulence 3 (8).

[7] Gessner, F.B., & Jones, J.B. 1965. On some aspects of fully-developed turbulent flow in rectangular channels. J. Fluid Mech. 23 (689-713).

[8] Hinze, J.O. 1967. Secondary Currents in Wall Turbulence. Phys. Fluids 10 (S122).

[9] Ikeda, S. 1981. Self-forced straight channels in sandy beds. J. Hydraulics Div. 107 (389-406).

[10] Kimura, I., & Hosoda, T. 2003. A non-linear k-ε model with realizability for Prediction of flows around bluff bodies. J. Numer. Meth. Fluids 42 (813-837).

[11] Launder, B.E., & Spalding, D.B., 1974. The numerical computation of turbulent flows. Computer Methods in Applied Mechanics and Engineering, 3(2): 269-289.

[12] Moser, R.D., Kim, J., & Mansour, N.N. 1999. Direct numerical Simulation of turbulent channel flow up to $Re_\tau = 590$. Physics of Fluids 11(4).

[13] Muller, A., & Studerus, X., 1979. Secondary flow in open channel. Proceedings of 18th IAHR congress, Cagliari, 3 (19-24).

[14] Naot, D., & Rodi, W. 1982. Calculation of secondary currents is channel flows. J. Hydraulics Div. 108 (948-968).

[15] Nezu, I., Nagakawa, H. 1993. Turbulence in open channel flow. Rotterdam, Balkema, A.A.

[16] Nezu, I., Nagakawa, H. 1994. Cellular secondary currents in straight conduit. J. Hydraulic Eng. 110 (173-193).

[17] Nezu, I., Rodi, W. 1985. Experimental study on secondary currents in open channel flow. Proc. Of 21st IAHR congress, Melbourne, vol 2. (115-119).

[18] Nieuwstadt, F.T.M. 2008. Turbulentie. Utrecht, Epsilon Uitgaven.

[19] Nisizima, S. 1990. A numerical Study of Turbulent Square-Duct Flow Using an Anisotropic k-ε Model. Theoretical and Computational Fluid Dynamics. Springer-Verlag.

[20] Perkins, H.J. 1970. The formation of streamwise vorticity in turbulent flow. J. Fluid Mech. 44 (721-740).

[21] Pope, S.B. 2000 Turbulent flows. Cambridge University press.

[22] Prooijen, B.C. van. 2004. Shallow mixing layers. PrintPartners Ipskamp B.V.

[23] Rodi, W. 1980. Turbulence models and their application in hydraulics. IAHR-monograph, Delft.

[24] Sagout. 1998. Large eddy simulation for incompressible flows. Scientific computation. Springer.

[25] Speziale, C.G. 1987. On non-linear k-l and k-ε models of turbulence. J. Fluid Mech. 178 (459-475).

[26] Stelling, G.S., & Duinmeijer, S.P.A. 2003. A staggered conservative scheme for every Froude number in rapidly varied shallow water flows. Num. Meth. Fluids 43 (1329-1354).

[27] Stelling, G.S., & Zijlema, M. 2003. An accurate and efficient finite-difference algorithm for non-hydrostatic free-surface flow with application to wave propagation. J. Numer. Meth. Fluids 44 (1-23).

[28] SWASH team, The. 2012. SWASH user manual. http://swash.sourgeforge.net

[29] Tafti, D. 1996. Comparison of some upwind-biased high-order formulations with a second order central difference scheme for time integration of the incompressible Navier-Stokes equations. Computers and fluids 25 (647-665).

[30] Talstra, H. 2011. Large-scale turbulence structures in shallow separating flows. PhD Thesis.

[31] Vermaas, D.A. 2007. Mixing layers in open channel flow with abrupt bed roughness changes. Wageningen, Msc. Thesis.

[32] Wang, Z.Q., & Cheng, N.S. 2006. Time-mean structure of secondary flows in open channels with longitudinal bedform. Advances in water resources 29 (1634-1949).

[33] Zijlema, M., & Stelling, G.S. 2005. Further experiences with computing non-Hydrostatic free-surface flows involving water waves. J. Numer. Meth. Fluids 48 (169-197).

[34] Zijlema, M., & Stelling, G.S. 2008. Efficient computation of surf zone waves using The nonlinear shallow water equations with non-hydrostatic pressure. Coastal Engineering 55 (780-790).

[35] Zijlema, M., Stelling, G.S., & Smit, P. 2011. SWASH: An operational public domain code for simulating wave fields and rapidly varied flows in coastal waters. Coastal Engineering 58 (992-1012).

Appendices

The original SWASH sourcecode is adapted and extended to perform RANS computations with the non-linear k-ε closure and to perform LES. The original code can be downloaded from http://swash.sourceforge.net/.

Appendix A consists of the modules that are added for the RANS computations. Modules that are needed to run LES are supplemented in Appendix B.

A. SWASH subroutines for RANS computations with the non-linear k-ε closure

In the new subroutine SwashKepsNonLin the Reynolds stress tensor is computed. The original module SwashKepsMod2DH is still needed to solve the transport equations for k and epsilon.

```
subroutine SwashKepsNonLin
!
!   --|-----------------------------------------------------------|--
!     | Delft University of Technology                            |
!     | Faculty of Civil Engineering                              |
!     | Environmental Fluid Mechanics Section                     |
!     | P.O. Box 5048, 2600 GA  Delft, The Netherlands            |
!     |                                                           |
!     | Programmers: The SWASH team                               |
!   --|-----------------------------------------------------------|--
!
!
!     SWASH (Simulating WAves till SHore); a non-hydrostatic wave-flow model
!     Copyright (C) 2010-2011  Delft University of Technology
!
!     This program is free software; you can redistribute it and/or
!     modify it under the terms of the GNU General Public License as
!     published by the Free Software Foundation; either version 2 of
!     the License, or (at your option) any later version.
!
!     This program is distributed in the hope that it will be useful,
!     but WITHOUT ANY WARRANTY; without even the implied warranty of
!     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
!     GNU General Public License for more details.
!
!     A copy of the GNU General Public License is available at
!     http://www.gnu.org/copyleft/gpl.html#SEC3
!     or by writing to the Free Software Foundation, Inc.,
!     59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
!
!
!   Authors
!
!    1.00: Tom Bogaard
!
!   Updates
!
!    1.00, January 2012: New subroutine
!
!   Purpose
!
!   Calculates reynolds stresses with non-linear k-epsilon model
!
!   Method
!
!   Modules used
!
    use ocpcomm4
    use SwashCommdata3
    use SwashFlowdata
    use m_genarr
    use m_parall
!
    implicit none
!
!   Parameter variables
!
    real, parameter :: c1  = 0.4536  ! closure constant for non-lin k-eps model
    real, parameter :: c2  = 0.3024  ! other closure constant for non-lin k-eps model
    real, parameter :: c3  = -0.1512 ! other closure constant for non-lin k-eps model
    real, parameter :: cmu = 0.09    ! other closure constant for standard k-eps model
!
!   Local variables
```

```
!
    integer, save                       :: ient = 0    ! number of entries in this subroutine
    integer                             :: m           ! loop counter in x-direction
    integer                             :: n           ! loop counter in y direction
    integer                             :: k           ! loop counter over vertical layers
    integer                             :: kk          ! loop counter over vertical layers (minimal value = 1)
    integer                             :: mu          ! index of point m+1
    integer                             :: md          ! index of point m-1
    integer                             :: mdd         ! index of point m-2
    integer                             :: muu         ! index of point m+2
    integer                             :: ndd         ! index of point n-2
    integer                             :: nu          ! index of point n+1
    integer                             :: nuu         ! index of point n+2
    integer                             :: nd          ! index of point n-1
    integer                             :: kd          ! index of layer k-1
    integer                             :: ku          ! index of layer k+1
    integer                             :: kdd         ! index of layer k-2
    integer                             :: kuu         ! index of layer k+2
    integer                             :: kwd         ! index of layer k-1 for w-velocity
    integer                             :: kwu         ! index of layer k+1 for w-velocity
    integer                             :: kwdd        ! index of layer k-2 for w-velocity
    integer                             :: kwuu        ! index of layer k+2 for w-velocity
    !
    integer                             :: nm          ! pointer to m   ,n
    integer                             :: num         ! pointer to m   ,n+1
    integer                             :: ndm         ! pointer to m   ,n-1
    integer                             :: nmu         ! pointer to m+1 ,n
    integer                             :: nmd         ! pointer to m-1 ,n
    integer                             :: numd        ! pointer to m-1 ,n+1
    integer                             :: numu        ! pointer to m+1 ,n+1
    integer                             :: ndmd        ! pointer to m+1 ,n-1
    integer                             :: ndmu        ! pointer to m-1 ,n+1
    integer                             :: nddm        ! pointer to m   ,n-2
    integer                             :: nddmu       ! pointer to m+1 ,n-2
    integer                             :: nmdd        ! pointer to m-2 ,n
    integer                             :: numdd       ! pointer to m-2 ,n+1
    integer                             :: nmuu        ! pointer to m+2 ,n
    integer                             :: ndmuu       ! pointer to m+2 ,n-1
    integer                             :: nuum        ! pointer to m   ,n+2
    integer                             :: nuumd       ! pointer to m-1 ,n+2
    !
    integer                             :: nlm         ! pointer to m+2 ,n
    integer                             :: nlum        ! pointer to m+2 ,n-1
    integer                             :: nfm         ! pointer to m   ,n+2
    integer                             :: nfum        ! pointer to m-1 ,n+2
    !
    real                                :: dudxu       ! velocity gradient in upward point
    real                                :: dudyu       ! velocity gradient in upward point
    real                                :: dudzu       ! velocity gradient in upward point
    real                                :: dvdxu       ! velocity gradient in upward point
    real                                :: dvdyu       ! velocity gradient in upward point
    real                                :: dvdzu       ! velocity gradient in upward point
    real                                :: dwdxu       ! velocity gradient in upward point
    real                                :: dwdyu       ! velocity gradient in upward point
    real                                :: dwdzu       ! velocity gradient in upward point
    real                                :: dudxd       ! velocity gradient in downward point
    real                                :: dudyd       ! velocity gradient in downward point
    real                                :: dudzd       ! velocity gradient in downward point
    real                                :: dvdxd       ! velocity gradient in downward point
    real                                :: dvdyd       ! velocity gradient in downward point
    real                                :: dvdzd       ! velocity gradient in downward point
    real                                :: dwdxd       ! velocity gradient in downward point
    real                                :: dwdyd       ! velocity gradient in downward point
    real                                :: dwdzd       ! velocity gradient in downward point
    !
    real                                :: sxxu        ! stress tensor in upward point
    real                                :: sxyu        ! stress tensor in upward point
    real                                :: sxzu        ! stress tensor in upward point
    real                                :: sxxd        ! stress tensor in downward point
    real                                :: sxyd        ! stress tensor in downward point
    real                                :: sxzd        ! stress tensor in downward point
```

```
real                            :: s1xxu       ! stress tensor in upward point
real                            :: s1xyu       ! stress tensor in upward point
real                            :: s1xzu       ! stress tensor in upward point
real                            :: s1xxd       ! stress tensor in downward point
real                            :: s1xyd       ! stress tensor in downward point
real                            :: s1xzd       ! stress tensor in downward point
real                            :: s2xxu       ! stress tensor in upward point
real                            :: s2xyu       ! stress tensor in upward point
real                            :: s2xzu       ! stress tensor in upward point
real                            :: s2xxd       ! stress tensor in downward point
real                            :: s2xyd       ! stress tensor in downward point
real                            :: s2xzd       ! stress tensor in downward point
real                            :: s3xxu       ! stress tensor in upward point
real                            :: s3xyu       ! stress tensor in upward point
real                            :: s3xzu       ! stress tensor in upward point
real                            :: s3xxd       ! stress tensor in downward point
real                            :: s3xyd       ! stress tensor in downward point
real                            :: s3xzd       ! stress tensor in downward point
!
real                            :: syxu        ! stress tensor in upward point
real                            :: syyu        ! stress tensor in upward point
real                            :: syzu        ! stress tensor in upward point
real                            :: syxd        ! stress tensor in downward point
real                            :: syyd        ! stress tensor in downward point
real                            :: syzd        ! stress tensor in downward point
real                            :: s1yxu       ! stress tensor in upward point
real                            :: s1yyu       ! stress tensor in upward point
real                            :: s1yzu       ! stress tensor in upward point
real                            :: s1yxd       ! stress tensor in downward point
real                            :: s1yyd       ! stress tensor in downward point
real                            :: s1yzd       ! stress tensor in downward point
real                            :: s2yxu       ! stress tensor in upward point
real                            :: s2yyu       ! stress tensor in upward point
real                            :: s2yzu       ! stress tensor in upward point
real                            :: s2yxd       ! stress tensor in downward point
real                            :: s2yyd       ! stress tensor in downward point
real                            :: s2yzd       ! stress tensor in downward point
real                            :: s3yxu       ! stress tensor in upward point
real                            :: s3yyu       ! stress tensor in upward point
real                            :: s3yzu       ! stress tensor in upward point
real                            :: s3yxd       ! stress tensor in downward point
real                            :: s3yyd       ! stress tensor in downward point
real                            :: s3yzd       ! stress tensor in downward point
!
real                            :: szxu        ! stress tensor in upward point
real                            :: szyu        ! stress tensor in upward point
real                            :: szzu        ! stress tensor in upward point
real                            :: szxd        ! stress tensor in downward point
real                            :: szyd        ! stress tensor in downward point
real                            :: szzd        ! stress tensor in downward point
real                            :: s1zxu       ! stress tensor in upward point
real                            :: s1zyu       ! stress tensor in upward point
real                            :: s1zzu       ! stress tensor in upward point
real                            :: s1zxd       ! stress tensor in downward point
real                            :: s1zyd       ! stress tensor in downward point
real                            :: s1zzd       ! stress tensor in downward point
real                            :: s2zxu       ! stress tensor in upward point
real                            :: s2zyu       ! stress tensor in upward point
real                            :: s2zzu       ! stress tensor in upward point
real                            :: s2zxd       ! stress tensor in downward point
real                            :: s2zyd       ! stress tensor in downward point
real                            :: s2zzd       ! stress tensor in downward point
real                            :: s3zxu       ! stress tensor in upward point
real                            :: s3zyu       ! stress tensor in upward point
real                            :: s3zzu       ! stress tensor in upward point
real                            :: s3zxd       ! stress tensor in downward point
real                            :: s3zyd       ! stress tensor in downward point
real                            :: s3zzd       ! stress tensor in downward point
!
real                            :: kepsku      ! TKE in upward point
real                            :: kepskd      ! TKE in downward point
```

```
    real                                :: kepseu      ! Dissipation in upward point
    real                                :: kepsed      ! Dissipation in downward point
    real                                :: visceu      ! eddy viscosity in upward point
    real                                :: visced      ! eddy viscosity in downward point
!
!   Structure
!
!   Description of the pseudo code
!
!   Source text
!
    if (ltrace) call strace (ient,'SwashKepsNonLin')
!
    do m = mf, ml
       !
       nfm  = kgrpnt(m,nf )
       nfum = kgrpnt(m,nfu)
       nlm  = kgrpnt(m,nl )
       nlum = kgrpnt(m,nlu)
       !
    enddo
!
! u-momemtun equation
!
       do k = 1, kmax
          !
          do n = nfu, nl
             !
             do m = mf+1, ml-1
                !
                md    = m - 1
                mu    = m + 1
                nd    = n - 1
                nu    = n + 1
                ndd   = n - 2
                muu   = m + 2
                !
                if ( n == nfu ) ndd = nd
                !
                nm    = kgrpnt(m ,n )
                nmd   = kgrpnt(md,n )
                nmu   = kgrpnt(mu,n )
                ndm   = kgrpnt(m ,nd)
                num   = kgrpnt(m ,nu)
                ndmd  = kgrpnt(md,nd)
                ndmu  = kgrpnt(mu,nd)
                numd  = kgrpnt(md,nu)
                numu  = kgrpnt(mu,nu)
                nddm  = kgrpnt(m,ndd)
                nddmu = kgrpnt(mu,ndd)
                nmuu  = kgrpnt(muu,n)
                ndmuu = kgrpnt(muu,nd)
                !
                kd  = max(k-1,1   )
                kdd = max(k-2,1   )
                ku  = min(k+1,kmax)
                kuu = min(k+2,kmax)
                kwd  = max(k-1,0   )
                kwdd = max(k-2,0   )
                kwu  = min(k+1,kmax)
                kwuu = min(k+2,kmax)
                !
                ! -u'u' / dx
                !
                ! compose parts of reysxxu
                !
                if ( m == ml-1) nmuu  = nmu
                if ( m == ml-1) ndmuu = ndmu
                !
                dudxu = ( u0(nmu,k) - u0(nm,k) ) / ( 0.5 * ( gvv(ndmu) + gvv(nmu) ) )
                dudyu = ( 0.25 * ( u0(numu,k) + u0(num,k) + u0(nmu,k) + u0(nm,k) ) &
```

```
          - 0.25 * ( u0(nmu,k) + u0(nm,k) + u0(ndmu,k) + u0(ndm,k) ) ) &
          / ( 0.5 * ( guu(nm) + guu(nmu) ) )
dudzu = ( 0.25 * ( u0(nm,k) + u0(nmu,k) + u0(nm,kd) + u0(nmu,kd) ) &
          - 0.25 * ( u0(nm,k) + u0(nmu,k) + u0(nm,ku) + u0(nmu,ku) ) ) / hks(nmu,k)
!
dvdxu = ( 0.25 * ( v0(nmuu,k) + v0(nmu,k) + v0(ndmuu,k) + v0(ndmu,k) ) &
          - 0.25 * ( v0(nmu,k) + v0(ndmu,k) + v0(nm,k) + v0(ndm,k) ) ) &
          / ( 0.5 * ( gvv(ndmu) + gvv(nmu) ) )
dvdyu = ( v0(nmu,k) - v0(ndmu,k) ) / ( 0.5 * ( guu(nm) + guu(nmu) ) )
dvdzu = ( 0.25 * ( v0(nmu,k) + v0(ndmu,k) + v0(nmu,kd) + v0(ndmu,kd) ) &
          - 0.25 * ( v0(nmu,k) + v0(ndmu,k) + v0(nmu,ku) + v0(ndmu,ku) ) ) / hks(nmu,k)
!
dwdxu = ( 0.25 * ( w0(nmuu,k) + w0(nmu,k) + w0(nmuu,k-1) + w0(nmu,k-1) ) &
          - 0.25 * ( w0(nmu,k) + w0(nm,k) + w0(nmu,k-1) + w0(nm,k-1) ) ) &
          / ( 0.5 * ( gvv(ndmu) + gvv(nmu) ) )
dwdyu = ( 0.25 * ( w0(nmu,k) + w0(numu,k) + w0(nmu,k-1) + w0(numu,k-1) ) &
          - 0.25 * ( w0(ndmu,k) + w0(nmu,k) + w0(ndmu,k-1) + w0(nmu,k-1) ) ) &
          / ( 0.5 * ( guu(nm) + guu(nmu) ) )
dwdzu = ( w0(nmu,k-1) - w0(nmu,k) ) / hks(nmu,k)
!
sxxu  = dudxu + dudxu
!
s1xxu = ( dudxu * dudxu ) + ( dudyu * dudyu ) + ( dudzu * dudzu )
s1yyu = ( dvdxu * dvdxu ) + ( dvdyu * dvdyu ) + ( dvdzu * dvdzu )
s1zzu = ( dwdxu * dwdxu ) + ( dwdyu * dwdyu ) + ( dwdzu * dwdzu )
!
s2xxu = 0.5 * ( ( dudxu * dudxu + dudxu * dudxu ) + ( dudyu * dvdxu + dudyu * dvdxu ) &
            + ( dudzu * dwdxu + dudzu * dwdxu ) )
s2yyu = 0.5 * ( ( dvdxu * dudyu + dvdxu * dudyu ) + ( dvdyu * dvdyu + dvdyu * dvdyu ) &
            + ( dvdzu * dwdyu + dvdzu * dwdyu ) )
s2zzu = 0.5 * ( ( dwdxu * dudzu + dwdxu * dudzu ) + ( dwdyu * dvdzu + dwdyu * dvdzu ) &
            + ( dwdzu * dwdzu + dwdzu * dwdzu ) )
!
s3xxu = ( dudxu * dudxu ) + ( dvdxu * dvdxu ) + ( dwdxu * dwdxu )
s3yyu = ( dudyu * dudyu ) + ( dvdyu * dvdyu ) + ( dwdyu * dwdyu )
s3zzu = ( dudzu * dudzu ) + ( dvdzu * dvdzu ) + ( dwdzu * dwdzu )
!
kepsku = 0.5 * ( rtur(nmu,k-1,1) + rtur(nmu,k,1) )
kepseu = 0.5 * ( rtur(nmu,k-1,2) + rtur(nmu,k,2) )
!
visceu = 0.5 * ( vnu3d(nmu,k-1) + vnu3d(nmu,k) )
!
! compose reysxxu (linear and non-linear part)
!
!
reysxxu(nm,k) = visceu * sxxu &
                - 1. * cmu * ( ( kepsku ** 3. ) / ( kepseu ** 2. ) ) * ( &
                       c1 * ( s1xxu - ( 1. / 3. ) * ( s1xxu + s1yyu + s1zzu ) ) + &
                       c2 * ( s2xxu - ( 1. / 3. ) * ( s2xxu + s2yyu + s2zzu ) ) + &
                       c3 * ( s3xxu - ( 1. / 3. ) * ( s3xxu + s3yyu + s3zzu ) ) )
!
! compose parts of reysxxd
!
dudxd = ( u0(nm,k) - u0(nmd,k) ) / ( 0.5 * ( gvv(ndm) + gvv(nm) ) )
dudyd = ( 0.25 * ( u0(num,k) + u0(numd,k) + u0(nm,k) + u0(nmd,k) ) &
          - 0.25 * ( u0(nm,k) + u0(nmd,k) + u0(ndm,k) + u0(ndmd,k) ) ) &
          / ( 0.5 * ( guu(nmd) + guu(nm) ) )
dudzd = ( 0.25 * ( u0(nmd,k) + u0(nm,k) + u0(nmd,kd) + u0(nm,kd) ) &
          - 0.25 * ( u0(nmd,k) + u0(nm,k) + u0(nmd,ku) + u0(nm,ku) ) ) / hks(nm,k)
!
dvdxd = ( 0.25 * ( v0(nmu,k) + v0(nm,k) + v0(ndmu,k) + v0(ndm,k) ) &
          - 0.25 * ( v0(nm,k) + v0(ndm,k) + v0(nmd,k) + v0(ndmd,k) ) ) &
          / ( 0.5 * ( gvv(ndm) + gvv(nm) ) )
dvdyd = ( v0(nm,k) - v0(ndm,k) ) / ( 0.5 * ( guu(nmd) + guu(nm) ) )
dvdzd = ( 0.25 * ( v0(nm,k) + v0(ndm,k) + v0(nm,kd) + v0(ndm,kd) ) &
          - 0.25 * ( v0(nm,k) + v0(ndm,k) + v0(nm,ku) + v0(ndm,ku) ) ) / hks(nm,k)
!
dwdxd = ( 0.25 * ( w0(nmu,k) + w0(nm,k) + w0(nmu,k-1) + w0(nm,k-1) ) &
          - 0.25 * ( w0(nm,k) + w0(nmd,k) + w0(nm,k-1) + w0(nmd,k-1) ) ) %
          / ( 0.5 * ( gvv(ndm) + gvv(nm) ) )
dwdyd = ( 0.25 * ( w0(nm,k) + w0(num,k) + w0(nm,k-1) + w0(num,k-1) ) &
```

```
          - 0.25 * ( w0(ndm,k) + w0(nm,k) + w0(ndm,k-1) + w0(nm,k-1) ) ) &
          / ( 0.5 * ( guu(nmd) + guu(nm) ) )
dwdzd = ( w0(nm,k-1) - w0(nm,k) ) / hks(nm,k)
!
sxxd  = dudxd + dudxd
!
s1xxd = ( dudxd * dudxd ) + ( dudyd * dudyd ) + ( dudzd * dudzd )
s1yyd = ( dvdxd * dvdxd ) + ( dvdyd * dvdyd ) + ( dvdzd * dvdzd )
s1zzd = ( dwdxd * dwdxd ) + ( dwdyd * dwdyd ) + ( dwdzd * dwdzd )
!
s2xxd = 0.5 * ( ( dudxd * dudxd + dudxd * dudxd ) + ( dudyd * dvdxd + dudyd * dvdxd ) &
            + ( dudzd * dwdxd + dudzd * dwdxd ) )
s2yyd = 0.5 * ( ( dvdxd * dudyd + dvdxd * dudyd ) + ( dvdyd * dvdyd + dvdyd * dvdyd ) &
            + ( dvdzd * dwdyd + dvdzd * dwdyd ) )
s2zzd = 0.5 * ( ( dwdxd * dudzd + dwdxd * dudzd ) + ( dwdyd * dvdzd + dwdyd * dvdzd ) &
            + ( dwdzd * dwdzd + dwdzd * dwdzd ) )
!
s3xxd = ( dudxd * dudxd ) + ( dvdxd * dvdxd ) + ( dwdxd * dwdxd )
s3yyd = ( dudyd * dudyd ) + ( dvdyd * dvdyd ) + ( dwdyd * dwdyd )
s3zzd = ( dudzd * dudzd ) + ( dvdzd * dvdzd ) + ( dwdzd * dwdzd )
!
kepskd = 0.5 * ( rtur(nm,k-1,1) + rtur(nm,k,1) )
kepsed = 0.5 * ( rtur(nm,k-1,2) + rtur(nm,k,2) )
!
visced = 0.5 * ( vnu3d(nm,k-1) + vnu3d(nm,k) )
!
! compose reysxxd (linear and non-linear part)
!
reysxxd(nm,k) = visced * sxxd &
                - 1. * cmu * ( ( kepskd ** 3. ) / ( kepsed ** 2. ) ) * ( &
                       c1 * ( s1xxd - ( 1. / 3. ) * ( s1xxd + s1yyd + s1zzd ) ) + &
                       c2 * ( s2xxd - ( 1. / 3. ) * ( s2xxd + s2yyd + s2zzd ) ) + &
                       c3 * ( s3xxd - ( 1. / 3. ) * ( s3xxd + s3yyd + s3zzd ) ) )
!
! -u'v' / dy
!
! compose parts of reysxyu
!
dudxu = ( 0.5 * ( u0(numu,k) + u0(nmu,k) ) - 0.5 * ( u0(numd,k) + u0(nmd,k) ) ) &
          / ( guu(nm) + guu(nmu) )
dudyu = ( u0(num,k) - u0(nm,k) ) / ( 0.5 * ( guu(nm) + guu(num) ) )
dudzu = ( 0.25 * ( u0(nm,k) + u0(num,k) + u0(nm,kd) + u0(num,kd) ) &
          - 0.25 * ( u0(nm,k) + u0(num,k) + u0(nm,ku) + u0(num,ku) ) ) &
          / ( 0.5 * ( hkum(nm,k) + hkum(num,k) ) )
!
dvdxu = ( v0(nmu,k) - v0(nm,k) ) / ( 0.5 * ( gvv(nm) + gvv(nmu) ) )
dvdyu = ( 0.5 * ( v0(numu,k) + v0(num,k) ) - 0.5 * ( v0(ndmu,k) + v0(ndm,k) ) ) &
          / ( guu(num) + guu(nm) )
dvdzu = ( 0.25 * ( v0(nm,k) + v0(nmu,k) + v0(nm,kd) + v0(nmu,kd) ) &
          - 0.25 * ( v0(nm,k) + v0(nmu,k) + v0(nm,ku) + v0(nmu,ku) ) ) &
          / ( 0.5 * ( hkum(nm,k) + hkum(num,k) ) )
!
dwdxu = ( 0.25 * ( w0(nmu,k) + w0(numu,k) + w0(nmu,k-1) + w0(numu,k-1) ) &
          - 0.25 * ( w0(nm,k) + w0(num,k) + w0(nm,k-1) + w0(num,k-1) ) ) &
          / ( 0.5 * ( gvv(nm) + gvv(nmu) ) )
dwdyu = ( 0.25 * ( w0(num,k) + w0(numu,k) + w0(num,k-1) + w0(numu,k-1) ) &
          - 0.25 * ( w0(nm,k) + w0(nmu,k) + w0(nm,k-1) + w0(nmu,k-1) ) ) &
          / ( 0.5 * ( guu(nm) + guu(num) ) )
!
sxyu  = dudyu + dvdxu
!
s1xyu = ( dudxu * dvdxu ) + ( dudyu * dvdyu ) + ( dudzu * dvdzu )
s2xyu = 0.5 * ( ( dudxu * dudyu + dvdxu * dudxu ) + ( dudyu * dvdyu + dvdyu * dvdxu ) &
            + ( dudzu * dwdyu + dvdzu * dwdxu ) )
s3xyu = ( dudxu * dudyu ) + ( dvdxu * dvdyu ) + ( dwdxu * dwdyu )
!
kepsku = 0.125 * ( rtur(nm,k,1) + rtur(nmu,k,1) + rtur(numu,k,1) + rtur(num,k,1) &
         + rtur(nm,k-1,1) + rtur(nmu,k-1,1) + rtur(numu,k-1,1) + rtur(num,k-1,1) )
kepseu = 0.125 * ( rtur(nm,k,2) + rtur(nmu,k,2) + rtur(numu,k,2) + rtur(num,k,2) &
         + rtur(nm,k-1,2) + rtur(nmu,k-1,2) + rtur(numu,k-1,2) + rtur(num,k-1,2) )
!
```

```
visceu = 0.125 * ( vnu3d(nm,k) + vnu3d(nmu,k) + vnu3d(numu,k) + vnu3d(num,k) &
         + vnu3d(nm,k-1) + vnu3d(nmu,k-1) + vnu3d(numu,k-1) + vnu3d(num,k-1) )
!
! compose reysxyu (linear and non-linear part)
!
reysxyu(nm,k) = visceu * sxyu &
                - 1. * cmu * ( ( kepsku ** 3. ) / ( kepseu ** 2. ) ) * ( &
                       c1 * ( s1xyu ) + &
                       c2 * ( s2xyu ) + &
                       c3 * ( s3xyu ) )
!
! compose parts of reysxyd
!
dudxd = ( 0.5 * ( u0(nmu,k) + u0(ndmu,k) ) - 0.5 * ( u0(nmd,k) + u0(ndmd,k) ) ) &
          / ( guu(ndm) + guu(ndmu) )
dudyd = ( u0(nm,k) - u0(ndm,k) ) / ( 0.5 * ( guu(ndm) + guu(nm) ) )
dudzd = ( 0.25 * ( u0(ndm,k) + u0(nm,k) + u0(ndm,kd) + u0(nm,kd) ) &
          - 0.25 * ( u0(ndm,k) + u0(nm,k) + u0(ndm,ku) + u0(nm,ku) ) ) &
          / ( 0.5 * ( hkum(ndm,k) + hkum(nm,k) ) )
!
dvdxd = ( v0(ndmu,k) - v0(ndm,k) ) / ( 0.5 * ( gvv(ndm) + gvv(ndmu) ) )
dvdyd = ( 0.5 * ( v0(nmu,k) + v0(nm,k) ) - 0.5 * ( v0(nddmu,k) + v0(nddm,k) ) ) &
          / ( guu(nm) + guu(ndm) )
dvdzd = ( 0.25 * ( v0(ndm,k) + v0(ndmu,k) + v0(ndm,kd) + v0(ndmu,kd) ) &
          - 0.25 * ( v0(ndm,k) + v0(ndmu,k) + v0(ndm,ku) + v0(ndmu,ku) ) ) &
          / ( 0.5 * ( hkum(ndm,k) + hkum(nm,k) ) )
!
dwdxd = ( 0.25 * ( w0(ndmu,k) + w0(nmu,k) + w0(ndmu,k-1) + w0(nmu,k-1) ) &
          - 0.25 * ( w0(ndm,k) + w0(nm,k) + w0(ndm,k-1) + w0(nm,k-1) ) ) &
          / ( 0.5 * ( gvv(ndm) + gvv(ndmu) ) )
dwdyd = ( 0.25 * ( w0(nm,k) + w0(nmu,k) + w0(nm,k-1) + w0(nmu,k-1) ) &
          - 0.25 * ( w0(ndm,k) + w0(ndmu,k) + w0(ndm,k-1) + w0(ndmu,k-1) ) ) &
          / ( 0.5 * ( guu(ndm) + guu(nm) ) )
!
sxyd  = dudyd + dvdxd
!
s1xyd = ( dudxd * dvdxd ) + ( dudyd * dvdyd ) + ( dudzd * dvdzd )
s2xyd = 0.5 * ( ( dudxd * dudyd + dvdxd * dudxd ) + ( dudyd * dvdyd + dvdyd * dvdxd ) &
        + ( dudzd * dwdyd + dvdzd * dwdxd ) )
s3xyd = ( dudxd * dudyd ) + ( dvdxd * dvdyd ) + ( dwdxd * dwdyd )
!
kepskd = 0.125 * ( rtur(nm,k,1) + rtur(nmu,k,1) + rtur(ndmu,k,1) + rtur(ndm,k,1) &
         + rtur(nm,k-1,1) + rtur(nmu,k-1,1) + rtur(ndmu,k-1,1) + rtur(ndm,k-1,1) )
kepsed = 0.125 * ( rtur(nm,k,2) + rtur(nmu,k,2) + rtur(ndmu,k,2) + rtur(ndm,k,2) &
         + rtur(nm,k-1,2) + rtur(nmu,k-1,2) + rtur(ndmu,k-1,2) + rtur(ndm,k-1,2) )
!
visced = 0.125 * ( vnu3d(nm,k) + vnu3d(nmu,k) + vnu3d(ndmu,k) + vnu3d(ndm,k) &
         + vnu3d(nm,k-1) + vnu3d(nmu,k-1) + vnu3d(ndmu,k-1) + vnu3d(ndm,k-1) )
!
! compose reysxyd (linear and non-linear part)
!
reysxyd(nm,k) = visced * sxyd &
                - 1. * cmu * ( ( kepskd ** 3. ) / ( kepsed ** 2. ) ) * ( &
                       c1 * ( s1xyd ) + &
                       c2 * ( s2xyd ) + &
                       c3 * ( s3xyd ) )
!
! -u'w' / dz
!
! compose parts of reysxzu
!
dudxu = ( 0.5 * ( u0(nmu,kd) + u0(nmu,k) ) - 0.5 * ( u0(nmd,kd) + u0(nmd,k) ) ) &
        / ( 2. * gvu(nm) )
dudyu = ( 0.25 * ( u0(nm,k) + u0(num,k) + u0(nm,kd) + u0(num,kd) ) &
          - 0.25 * ( u0(ndm,k) + u0(nm,k) + u0(ndm,kd) + u0(nm,kd) ) ) / gvv(nm)
dudzu = ( u0(nm,kd) - u0(nm,k) ) / ( 0.5 * ( hkum(nm,kd) + hkum(nm,k) ) )
!
dvdxu = ( 0.25 * ( v0(nmu,k) + v0(ndmu,k) + v0(nmu,kd) + v0(ndmu,kd) ) &
          - 0.25 * ( v0(nm,k) + v0(ndm,k) + v0(nm,kd) + v0(ndm,kd) ) ) / gvu(nm)
dvdzu = ( 0.25 * ( v0(nm,kd) + v0(nmu,kd) + v0(ndm,kd) + v0(ndmu,kd) ) &
          - 0.25 * ( v0(nm,k) + v0(nmu,k) + v0(ndm,k) + v0(ndmu,k) ) ) &
```

```
                / ( 0.5 * ( hkum(nm,kd) + hkum(nm,k) ) )
        !
        dwdxu = ( w0(nmu,kwd) - w0(nm,kwd) ) / gvu(nm)
        dwdyu = ( 0.25 * ( w0(nm,k-1) + w0(nmu,k-1) + w0(num,k-1) + w0(numu,k-1) ) &
                  - 0.25 * ( w0(ndm,k-1) + w0(ndmu,k-1) + w0(nm,k-1) + w0(nmu,k-1) ) ) / gvv(nm)
        dwdzu = ( ( 0.5 * ( w0(nmu,kwdd) + w0(nm,kwdd) ) ) - ( 0.5 * ( w0(nmu,k) + w0(nm,k) ) ) ) &
                  / ( hkum(nm,kd) + hkum(nm,k) )
        !
        sxzu  = dwdxu + dudzu
        !
        s1xzu = ( dudxu * dwdxu ) + ( dudyu * dwdyu ) + ( dudzu * dwdzu )
        s2xzu = 0.5 * ( ( dudxu * dudzu + dwdxu * dudxu ) + ( dudyu * dvdzu + dwdyu * dvdxu ) &
              + ( dudzu * dwdzu + dwdzu * dwdxu ) )
        s3xzu = ( dudxu * dudzu ) + ( dvdxu * dvdzu ) + ( dwdxu * dwdzu )
        !
        kepsku = 0.5 * ( rtur(nmu,k-1,1) + rtur(nm,k-1,1) )
        kepseu = 0.5 * ( rtur(nmu,k-1,2) + rtur(nm,k-1,2) )
        !
        visceu = 0.5 * ( vnu3d(nmu,k-1) + vnu3d(nm,k-1) )
        !
        ! compose reysxzu (linear and non-linear part)
        !
        reysxzu(nm,k) = visceu * sxzu &
                        - 1. * cmu * ( ( kepsku ** 3. ) / ( kepseu ** 2. ) ) * ( &
                               c1 * ( s1xzu ) + &
                               c2 * ( s2xzu ) + &
                               c3 * ( s3xzu ) )
        !
        ! compose parts of reysxzd
        !
        dudxd = ( 0.5 * ( u0(nmu,k) + u0(nmu,ku) ) - 0.5 * ( u0(nmd,k) + u0(nmd,ku) ) ) &
                  / ( 2. * gvu(nm) )
        dudyd = ( 0.25 * ( u0(nm,ku) + u0(num,ku) + u0(nm,k) + u0(num,k) ) &
                  - 0.25 * ( u0(ndm,ku) + u0(nm,ku) + u0(ndm,k) + u0(nm,k) ) ) / gvv(nm)
        dudzd = ( u0(nm,k) - u0(nm,ku) ) / ( 0.5 * ( hkum(nm,k) + hkum(nm,ku) ) )
        !
        dvdxd = ( 0.25 * ( v0(nmu,ku) + v0(ndmu,ku) + v0(nmu,k) + v0(ndmu,k) ) &
                  - 0.25 * ( v0(nm,ku) + v0(ndm,ku) + v0(nm,k) + v0(ndm,k) ) ) / gvu(nm)
        dvdzd = ( 0.25 * ( v0(nm,k) + v0(nmu,k) + v0(ndm,k) + v0(ndmu,k) ) &
                  - 0.25 * ( v0(nm,ku) + v0(nmu,ku) + v0(ndm,ku) + v0(ndmu,ku) ) ) &
                  / ( 0.5 * ( hkum(nm,k) + hkum(nm,ku) ) )
        !
        dwdxd = ( w0(nmu,k) - w0(nm,k) ) / gvu(nm)
        dwdyd = ( 0.25 * ( w0(nm,k) + w0(nmu,k) + w0(num,k) + w0(numu,k) ) &
                  - 0.25 * ( w0(ndm,k) + w0(ndmu,k) + w0(nm,k) + w0(nmu,k) ) ) / gvv(nm)
        dwdzd = ( ( 0.5 * ( w0(nmu,kwd) + w0(nm,kwd) ) ) - ( 0.5 * ( w0(nmu,ku) + w0(nm,ku) ) ) ) &
                  / ( hkum(nm,k) + hkum(nm,ku) )
        !
        sxzd  = dwdxd + dudzd
        !
        s1xzd = ( dudxd * dwdxd ) + ( dudyd * dwdyd ) + ( dudzd * dwdzd )
        s2xzd = 0.5 * ( ( dudxd * dudzd + dwdxd * dudxd ) + ( dudyd * dvdzd + dwdyd * dvdxd ) &
              + ( dudzd * dwdzd + dwdzd * dwdxd ) )
        s3xzd = ( dudxd * dudzd ) + ( dvdxd * dvdzd ) + ( dwdxd * dwdzd )
        !
        kepskd = 0.5 * ( rtur(nmu,k,1) + rtur(nm,k,1) )
        kepsed = 0.5 * ( rtur(nmu,k,2) + rtur(nm,k,2) )
        !
        visced = 0.5 * ( vnu3d(nmu,k) + vnu3d(nm,k) )
        !
        ! compose reysxzd (linear and non-linear part)
        !
        reysxzd(nm,k) = visced * sxzd &
                        - 1. * cmu * ( ( kepskd ** 3. ) / ( kepsed ** 2. ) ) * ( &
                               c1 * ( s1xzd ) + &
                               c2 * ( s2xzd ) + &
                               c3 * ( s3xzd ) )
        !
    enddo
    !
enddo
```

```
          !
       enddo
       !
!
! v-momemtun equation
!
       do k = 1, kmax
          !
          do n = nfu, nl-1
             !
             do m = mfu, ml
                !
                md    = m - 1
                mu    = m + 1
                nd    = n - 1
                nu    = n + 1
                mdd   = m - 2
                nuu   = n + 2
                !
                nm    = kgrpnt(m ,n )
                nmd   = kgrpnt(md,n )
                nmu   = kgrpnt(mu,n )
                ndm   = kgrpnt(m ,nd)
                num   = kgrpnt(m ,nu)
                ndmd  = kgrpnt(md,nd)
                ndmu  = kgrpnt(mu,nd)
                numd  = kgrpnt(md,nu)
                numu  = kgrpnt(mu,nu)
                numdd = kgrpnt(mdd,nu)
                nmdd  = kgrpnt(mdd,n)
                nuum  = kgrpnt(m,nuu)
                nuumd = kgrpnt(md,nuu)
                !
                kd  = max(k-1,1   )
                kdd = max(k-2,1   )
                ku  = min(k+1,kmax)
                kuu = min(k+2,kmax)
                kwd  = max(k-1,0   )
                kwdd = max(k-2,0   )
                kwu  = min(k+1,kmax)
                kwuu = min(k+2,kmax)
                !
                ! -v'u' / dx
                !
                ! compose parts of reysyxu
                !
                dudxu = ( ( 0.5 * ( u0(numu,k) + u0(nmu,k) ) ) - ( 0.5 * ( u0(numd,k) + u0(nmd,k) ) ) ) &
                          / ( gvv(nm) + gvv(nmu) )
                dudyu = ( u0(num,k) - u0(nm,k) ) / ( 0.5 * ( guu(num) + guu(nm) ) )
                dudzu = ( 0.25 * ( u0(nm,k) + u0(num,k) + u0(nm,kd) + u0(num,kd) ) &
                          - 0.25 * ( u0(nm,k) + u0(num,k) + u0(nm,ku) + u0(num,ku) ) ) &
                          / ( 0.5 * ( hkvm(nm,k) + hkvm(nmu,k) ) )
                !
                dvdxu = ( v0(nmu,k) - v0(nm,k) ) / ( 0.5 * ( gvv(nm) + gvv(nmu) ) )
                dvdyu = ( ( 0.5 * ( v0(numu,k) + v0(num,k) ) ) - ( 0.5 * ( v0(ndmu,k) + v0(ndm,k) ) ) ) &
                          / ( guu(nm) + guu(num) )
                dvdzu = ( 0.25 * ( v0(nm,k) + v0(nmu,k) + v0(nm,kd) + v0(nmu,kd) ) &
                          - 0.25 * ( v0(nm,k) + v0(nmu,k) + v0(nm,ku) + v0(nmu,ku) ) ) &
                          / ( 0.5 * ( hkvm(nm,k) + hkvm(nmu,k) ) )
                !
                dwdxu = ( 0.25 * ( w0(nmu,k) + w0(numu,k) + w0(nmu,kd) + w0(numu,kd) ) &
                          - 0.25 * ( w0(nm,k) + w0(num,k) + w0(nm,kd) + w0(num,kd) ) ) &
                          / ( 0.5 * ( gvv(nm) + gvv(nmu) ) )
                dwdyu = ( 0.25 * ( w0(numu,k) + w0(num,k) + w0(numu,kd) + w0(num,kd) ) &
                          - 0.25 * ( w0(nmu,k) + w0(nm,k) + w0(nmu,kd) + w0(nm,kd) ) ) &
                          / ( 0.5 * ( guu(num) + guu(nm) ) )
                !
                syxu  = dudyu + dvdxu
                !
                s1yxu = ( dvdxu * dudxu ) + ( dvdyu * dudyu ) + ( dvdzu * dudzu )
                s2yxu = 0.5 * ( ( dvdxu * dudxu + dudxu * dudyu ) + ( dvdyu * dvdxu + dudyu * dvdyu ) &
```

```
        + ( dvdzu * dwdxu + dudzu * dwdyu ) )
s3yxu = ( dudyu * dudxu ) + ( dvdyu * dvdxu ) + ( dwdyu * dwdxu )
!
kepsku = 0.125 * ( rtur(nm,k,1) + rtur(nmu,k,1) + rtur(numu,k,1) + rtur(num,k,1) &
         + rtur(nm,k-1,1) + rtur(nmu,k-1,1) + rtur(numu,k-1,1) + rtur(num,k-1,1) )
kepseu = 0.125 * ( rtur(nm,k,2) + rtur(nmu,k,2) + rtur(numu,k,2) + rtur(num,k,2) &
         + rtur(nm,k-1,2) + rtur(nmu,k-1,2) + rtur(numu,k-1,2) + rtur(num,k-1,2) )
!
visceu = 0.125 * ( vnu3d(nm,k) + vnu3d(nmu,k) + vnu3d(numu,k) + vnu3d(num,k) &
         + vnu3d(nm,k-1) + vnu3d(nmu,k-1) + vnu3d(numu,k-1) + vnu3d(num,k-1) )
!
! compose reysyxu (linear and non-linear part)
!
reysyxu(nm,k) = visceu * syxu &
                - 1. * cmu * ( ( kepsku ** 3. ) / ( kepseu ** 2. ) ) * ( &
                       c1 * ( s1yxu ) + &
                       c2 * ( s2yxu ) + &
                       c3 * ( s3yxu ) )
!
! compose parts of reysyxd
!
dudxd = ( ( 0.5 * ( u0(num,k) + u0(nm,k) ) ) - ( 0.5 * ( u0(numdd,k) + u0(nmdd,k) ) ) ) &
          / ( gvv(nmd) + gvv(nm) )
dudyd = ( u0(numd,k) - u0(nmd,k) ) / ( 0.5 * ( guu(numd) + guu(nmd) ) )
dudzd = ( 0.25 * ( u0(nmd,k) + u0(numd,k) + u0(nmd,kd) + u0(numd,kd) ) &
          - 0.25 * ( u0(nmd,k) + u0(numd,k) + u0(nmd,ku) + u0(numd,ku) ) ) &
          / ( 0.5 * ( hkvm(nmd,k) + hkvm(nm,k) ) )
!
dvdxd = ( v0(nm,k) - v0(nmd,k) ) / ( 0.5 * ( gvv(nmd) + gvv(nm) ) )
dvdyd = ( ( 0.5 * ( v0(num,k) + v0(numd,k) ) ) - ( 0.5 * ( v0(ndm,k) + v0(ndmd,k) ) ) ) &
          / ( guu(nmd) + guu(numd) )
dvdzd = ( 0.25 * ( v0(nmd,k) + v0(nm,k) + v0(nmd,kd) + v0(nm,kd) ) &
          - 0.25 * ( v0(nmd,k) + v0(nm,k) + v0(nmd,ku) + v0(nm,ku) ) ) &
          / ( 0.5 * ( hkvm(nmd,k) + hkvm(nm,k) ) )
!
dwdxd = ( 0.25 * ( w0(nm,k) + w0(num,k) + w0(nm,kd) + w0(num,kd) ) &
          - 0.25 * ( w0(nmd,k) + w0(numd,k) + w0(nmd,kd) + w0(numd,kd) ) ) &
          / ( 0.5 * ( gvv(nmd) + gvv(nm) ) )
dwdyd = ( 0.25 * ( w0(num,k) + w0(numd,k) + w0(num,kd) + w0(numd,kd) ) &
          - 0.25 * ( w0(nm,k) + w0(nmd,k) + w0(nm,kd) + w0(nmd,kd) ) ) &
          / ( 0.5 * ( guu(numd) + guu(nmd) ) )
!
syxd  = dudyd + dvdxd
!
s1yxd = ( dvdxd * dudxd ) + ( dvdyd * dudyd ) + ( dvdzd * dudzd )
s2yxd = 0.5 * ( ( dvdxd * dudxd + dudxd * dudyd ) + ( dvdyd * dvdxd + dudyd * dvdyd ) &
        + ( dvdzd * dwdxd + dudzd * dwdyd ) )
s3yxd = ( dudyd * dudxd ) + ( dvdyd * dvdxd ) + ( dwdyd * dwdxd )
!
kepskd = 0.125 * ( rtur(nm,k,1) + rtur(nmd,k,1) + rtur(numd,k,1) + rtur(num,k,1) &
         + rtur(nm,k-1,1) + rtur(nmd,k-1,1) + rtur(numd,k-1,1) + rtur(num,k-1,1) )
kepsed = 0.125 * ( rtur(nm,k,2) + rtur(nmd,k,2) + rtur(numd,k,2) + rtur(num,k,2) &
         + rtur(nm,k-1,2) + rtur(nmd,k-1,2) + rtur(numd,k-1,2) + rtur(num,k-1,2) )
!
visced = 0.125 * ( vnu3d(nm,k) + vnu3d(nmd,k) + vnu3d(numd,k) + vnu3d(num,k) &
         + vnu3d(nm,k-1) + vnu3d(nmd,k-1) + vnu3d(numd,k-1) + vnu3d(num,k-1) )
!
! compose reysyxd (linear and non-linear part)
!
reysyxd(nm,k) = visced * syxd &
                - 1. * cmu * ( ( kepskd ** 3. ) / ( kepsed ** 2. ) ) * ( &
                       c1 * ( s1yxd ) + &
                       c2 * ( s2yxd ) + &
                       c3 * ( s3yxd ) )
!
! -v'v' / dy
!
! compose parts of reysyyu
!
dudxu = ( u0(num,k) - u0(numd,k) ) / ( 0.5 * ( gvv(num) + gvv(nm) ) )
dudyu = ( 0.25 * ( u0(num,k) + u0(numd,k) + u0(nuum,k) + u0(nuumd,k) ) &
```

```
          - 0.25 * ( u0(nm,k) + u0(nmd,k) + u0(num,k) + u0(numd,k) ) ) &
          / ( 0.5 * ( guu(num) + guu(numd) ) )
dudzu = ( 0.25 * ( u0(num,k) + u0(numd,k) + u0(num,kd) + u0(numd,kd) ) &
          - 0.25 * ( u0(num,k) + u0(numd,k) + u0(num,ku) + u0(numd,ku) ) ) / hks(num,k)
!
dvdxu = ( 0.25 * ( v0(nm,k) + v0(nmu,k) + v0(num,k) + v0(numu,k) ) &
          - 0.25 * ( v0(nmd,k) + v0(nm,k) + v0(numd,k) + v0(num,k) ) ) &
          / ( 0.5 * ( gvv(num) + gvv(nm) ) )
dvdyu = ( v0(num,k) - v0(nm,k) ) / ( 0.5 * ( guu(num) + guu(numd) ) )
dvdzu = ( 0.25 * ( v0(nm,k) + v0(num,k) + v0(nm,kd) + v0(num,kd) ) &
          - 0.25 * ( v0(nm,k) + v0(num,k) + v0(nm,ku) + v0(num,ku) ) ) / hks(num,k)
!
dwdxu = ( 0.25 * ( w0(num,k) + w0(numu,k) + w0(num,kwd) + w0(numu,kwd) ) &
          - 0.25 * ( w0(numd,k) + w0(num,k) + w0(numd,kwd) + w0(num,kwd) ) ) &
          / ( 0.5 * ( gvv(num) + gvv(nm) ) )
dwdyu = ( 0.25 * ( w0(num,k) + w0(nuum,k) + w0(num,kwd) + w0(nuum,kwd) ) &
          - 0.25 * ( w0(nm,k) + w0(num,k) + w0(nm,kwd) + w0(num,kwd) ) ) &
          / ( 0.5 * ( guu(num) + guu(numd) ) )
dwdzu = ( w0(num,k-1) - w0(num,k) ) / hks(num,k)
!
syyu  = dvdyu + dvdyu
!
s1xxu = ( dudxu * dudxu ) + ( dudyu * dudyu ) + ( dudzu * dudzu )
s1yyu = ( dvdxu * dvdxu ) + ( dvdyu * dvdyu ) + ( dvdzu * dvdzu )
s1zzu = ( dwdxu * dwdxu ) + ( dwdyu * dwdyu ) + ( dwdzu * dwdzu )
!
s2xxu = 0.5 * ( ( dudxu * dudxu + dudxu * dudxu ) + ( dudyu * dvdxu + dudyu * dvdxu ) &
        + ( dudzu * dwdxu + dudzu * dwdxu ) )
s2yyu = 0.5 * ( ( dvdxu * dudyu + dvdxu * dudyu ) + ( dvdyu * dvdyu + dvdyu * dvdyu ) &
        + ( dvdzu * dwdyu + dvdzu * dwdyu ) )
s2zzu = 0.5 * ( ( dwdxu * dudzu + dwdxu * dudzu ) + ( dwdyu * dvdzu + dwdyu * dvdzu ) &
        + ( dwdzu * dwdzu + dwdzu * dwdzu ) )
!
s3xxu = ( dudxu * dudxu ) + ( dvdxu * dvdxu ) + ( dwdxu * dwdxu )
s3yyu = ( dudyu * dudyu ) + ( dvdyu * dvdyu ) + ( dwdyu * dwdyu )
s3zzu = ( dudzu * dudzu ) + ( dvdzu * dvdzu ) + ( dwdzu * dwdzu )
!
kepsku = 0.5 * ( rtur(num,k-1,1) + rtur(num,k,1) )
kepseu = 0.5 * ( rtur(num,k-1,2) + rtur(num,k,2) )
!
visceu = 0.5 * ( vnu3d(num,k-1) + vnu3d(num,k) )
!
! compose reysyyu (linear and non-linear part)
!
reysyyu(nm,k) = visceu * syyu &
                - 1. * cmu * ( ( kepsku ** 3. ) / ( kepseu ** 2. ) ) * ( &
                       c1 * ( s1yyu - ( 1. / 3. ) * ( s1xxu + s1yyu + s1zzu ) ) + &
                       c2 * ( s2yyu - ( 1. / 3. ) * ( s2xxu + s2yyu + s2zzu ) ) + &
                       c3 * ( s3yyu - ( 1. / 3. ) * ( s3xxu + s3yyu + s3zzu ) ) )
!
! compose parts of reysyyd
!
dudxd = ( u0(nm,k) - u0(nmd,k) ) / ( 0.5 * ( gvv(nm) + gvv(ndm) ) )
dudyd = ( 0.25 * ( u0(nm,k) + u0(nmd,k) + u0(num,k) + u0(numd,k) ) &
          - 0.25 * ( u0(ndm,k) + u0(ndmd,k) + u0(nm,k) + u0(nmd,k) ) ) &
          / ( 0.5 * ( guu(nm) + guu(nmd) ) )
dudzd = ( 0.25 * ( u0(nm,k) + u0(nmd,k) + u0(nm,kd) + u0(nmd,kd) ) &
          - 0.25 * ( u0(nm,k) + u0(nmd,k) + u0(nm,ku) + u0(nmd,ku) ) ) / hks(nm,k)
!
dvdxd = ( 0.25 * ( v0(ndm,k) + v0(ndmu,k) + v0(nm,k) + v0(nmu,k) ) &
          - 0.25 * ( v0(ndmd,k) + v0(ndm,k) + v0(nmd,k) + v0(nm,k) ) ) &
          / ( 0.5 * ( gvv(nm) + gvv(ndm) ) )
dvdyd = ( v0(nm,k) - v0(ndm,k) ) / ( 0.5 * ( guu(nm) + guu(nmd) ) )
dvdzd = ( 0.25 * ( v0(ndm,k) + v0(nm,k) + v0(ndm,kd) + v0(nm,kd) ) &
          - 0.25 * ( v0(ndm,k) + v0(nm,k) + v0(ndm,ku) + v0(nm,ku) ) ) / hks(nm,k)
!
dwdxd = ( 0.25 * ( w0(nm,k) + w0(nmu,k) + w0(nm,kwd) + w0(nmu,kwd) ) &
          - 0.25 * ( w0(nmd,k) + w0(nm,k) + w0(nmd,kwd) + w0(nm,kwd) ) ) &
          / ( 0.5 * ( gvv(nm) + gvv(ndm) ) )
dwdyd = ( 0.25 * ( w0(nm,k) + w0(num,k) + w0(nm,kwd) + w0(num,kwd) ) &
          - 0.25 * ( w0(ndm,k) + w0(nm,k) + w0(ndm,kwd) + w0(nm,kwd) ) ) &
```

```
          / ( 0.5 * ( guu(nm) + guu(nmd) ) )
dwdzd = ( w0(nm,k-1) - w0(nm,k) ) / hks(nm,k)
!
syyd  = dvdyd + dvdyd
!
s1xxd = ( dudxd * dudxd ) + ( dudyd * dudyd ) + ( dudzd * dudzd )
s1yyd = ( dvdxd * dvdxd ) + ( dvdyd * dvdyd ) + ( dvdzd * dvdzd )
s1zzd = ( dwdxd * dwdxd ) + ( dwdyd * dwdyd ) + ( dwdzd * dwdzd )
!
s2xxd = 0.5 * ( ( dudxd * dudxd + dudxd * dudxd ) + ( dudyd * dvdxd + dudyd * dvdxd ) &
        + ( dudzd * dwdxd + dudzd * dwdxd ) )
s2yyd = 0.5 * ( ( dvdxd * dudyd + dvdxd * dudyd ) + ( dvdyd * dvdyd + dvdyd * dvdyd ) &
        + ( dvdzd * dwdyd + dvdzd * dwdyd ) )
s2zzd = 0.5 * ( ( dwdxd * dudzd + dwdxd * dudzd ) + ( dwdyd * dvdzd + dwdyd * dvdzd ) &
        + ( dwdzd * dwdzd + dwdzd * dwdzd ) )
!
s3xxd = ( dudxd * dudxd ) + ( dvdxd * dvdxd ) + ( dwdxd * dwdxd )
s3yyd = ( dudyd * dudyd ) + ( dvdyd * dvdyd ) + ( dwdyd * dwdyd )
s3zzd = ( dudzd * dudzd ) + ( dvdzd * dvdzd ) + ( dwdzd * dwdzd )
!
kepskd = 0.5 * ( rtur(nm,k-1,1) + rtur(nm,k,1) )
kepsed = 0.5 * ( rtur(nm,k-1,2) + rtur(nm,k,2) )
!
visced = 0.5 * ( vnu3d(nm,k-1) + vnu3d(nm,k) )
!
! compose reysyyd (linear and non-linear part)
!
reysyyd(nm,k) = visced * syyd &
                - 1. * cmu * ( ( kepskd ** 3. ) / ( kepsed ** 2. ) ) * ( &
                       c1 * ( s1yyd - ( 1. / 3. ) * ( s1xxd + s1yyd + s1zzd ) ) + &
                       c2 * ( s2yyd - ( 1. / 3. ) * ( s2xxd + s2yyd + s2zzd ) ) + &
                       c3 * ( s3yyd - ( 1. / 3. ) * ( s3xxd + s3yyd + s3zzd ) ) )
!
! -v'w' / dz
!
! compose parts of reysyzu
!
!
dudyu = ( 0.25 * ( u0(num,k) + u0(numd,k) + u0(num,kd) + u0(numd,kd) ) &
          - 0.25 * ( u0(nm,k) + u0(nmd,k) + u0(nm,kd) + u0(nmd,kd) ) ) / guv(nm)
dudzu = ( 0.25 * ( u0(nm,kd) + u0(nmd,kd) + u0(num,kd) + u0(numd,kd) ) &
          - 0.25 * ( u0(nm,k) + u0(nmd,k) + u0(num,k) + u0(numd,k) ) ) &
          / ( 0.5 * ( hkvm(nm,kd) + hkvm(nm,k) ) )
!
dvdxu = ( 0.25 * ( v0(nm,k) + v0(nmu,k) + v0(nm,kd) + v0(nmu,kd) ) &
          - 0.25 * ( v0(nm,k) + v0(nmd,k) + v0(nm,kd) + v0(nmd,kd) ) ) / gvv(nm)
dvdyu = ( ( 0.5 * ( v0(num,kd) + v0(num,k) ) ) - ( 0.5 * ( v0(ndm,kd) + v0(ndm,k) ) ) ) &
          / ( 2. * guv(nm) )
dvdzu = ( v0(nm,kd) - v0(nm,k) ) / ( 0.5 * ( hkvm(nm,kd) + hkvm(nm,k) ) )
!
dwdxu = ( 0.25 * ( w0(nm,kwd) + w0(nmu,kwd) + w0(num,kwd) + w0(numu,kwd) ) &
          - 0.25 * ( w0(nmd,kwd) + w0(nm,kwd) + w0(numd,kwd) + w0(num,kwd) ) ) / gvv(nm)
dwdyu = ( w0(num,kwd) - w0(nm,kwd) ) / guv(nm)
dwdzu = ( ( 0.5 * ( w0(num,kwdd) + w0(nm,kwdd) ) ) - ( 0.5 * ( w0(num,k) + w0(nm,k) ) ) ) &
          / ( hkvm(nm,kd) + hkvm(nm,k) )
!
syzu  = dwdyu + dvdzu
!
s1yzu = ( dvdxu * dwdxu ) + ( dvdyu * dwdyu ) + ( dvdzu * dwdzu )
s2yzu = 0.5 * ( ( dvdxu * dudzu + dwdxu * dudyu ) + ( dvdyu * dvdzu + dwdyu * dvdyu ) &
        + ( dvdzu * dwdzu + dwdzu * dwdyu ) )
s3yzu = ( dudyu * dudzu ) + ( dvdyu * dvdzu ) + ( dwdyu * dwdzu )
!
kepsku = 0.5 * ( rtur(nm,k-1,1) + rtur(num,k-1,1) )
kepseu = 0.5 * ( rtur(nm,k-1,2) + rtur(num,k-1,2) )
!
visceu = 0.5 * ( vnu3d(nm,k-1) + vnu3d(num,k-1) )
!
! compose reysyzu (linear and non-linear part)
!
reysyzu(nm,k) = visceu * syzu &
```

```
                              - 1. * cmu * ( ( kepsku ** 3. ) / ( kepseu ** 2. ) ) * ( &
                                     c1 * ( s1yzu ) + &
                                     c2 * ( s2yzu ) + &
                                     c3 * ( s3yzu ) )
                  !
                  ! compose parts of reysyzd
                  !
                  dudyd = ( 0.25 * ( u0(num,ku) + u0(numd,ku) + u0(num,k) + u0(numd,k) ) &
                            - 0.25 * ( u0(nm,ku) + u0(nmd,ku) + u0(nm,k) + u0(nmd,k) ) ) / guv(nm)
                  dudzd = ( 0.25 * ( u0(nm,k) + u0(nmd,k) + u0(num,k) + u0(numd,k) ) &
                            - 0.25 * ( u0(nm,ku) + u0(nmd,ku) + u0(num,ku) + u0(numd,ku) ) ) &
                            / ( 0.5 * ( hkvm(nm,k) + hkvm(nm,ku) ) )
                  !
                  dvdxd = ( 0.25 * ( v0(nm,ku) + v0(nmu,ku) + v0(nm,k) + v0(nmu,k) ) &
                            - 0.25 * ( v0(nm,ku) + v0(nmd,ku) + v0(nm,k) + v0(nmd,k) ) ) / gvv(nm)
                  dvdyd = ( ( 0.5 * ( v0(num,k) + v0(num,ku) ) ) - ( 0.5 * ( v0(ndm,k) + v0(ndm,ku) ) ) ) &
                            / ( 2. * guv(nm) )
                  dvdzd = ( v0(nm,k) - v0(nm,ku) ) / ( 0.5 * ( hkvm(nm,k) + hkvm(nm,ku) ) )
                  !
                  dwdxd = ( 0.25 * ( w0(nm,k) + w0(nmu,k) + w0(num,k) + w0(numu,k) ) &
                            - 0.25 * ( w0(nmd,k) + w0(nm,k) + w0(numd,k) + w0(num,k) ) ) / gvv(nm)
                  dwdyd = ( w0(num,k) - w0(nm,k) ) / guv(nm)
                  dwdzd = ( ( 0.5 * ( w0(num,kwd) + w0(nm,kwd) ) ) - ( 0.5 * ( w0(num,ku) + w0(nm,ku) ) ) ) &
                            / ( hkvm(nm,k) + hkvm(nm,ku) )
                  !
                  syzd  = dwdyd + dvdzd
                  !
                  s1yzd = ( dvdxd * dwdxd ) + ( dvdyd * dwdyd ) + ( dvdzd * dwdzd )
                  s2yzd = 0.5 * ( ( dvdxd * dudzd + dwdxd * dudyd ) + ( dvdyd * dvdzd + dwdyd * dvdyd ) &
                          + ( dvdzd * dwdzd + dwdzd * dwdyd ) )
                  s3yzd = ( dudyd * dudzd ) + ( dvdyd * dvdzd ) + ( dwdyd * dwdzd )
                  !
                  kepskd = 0.5 * ( rtur(nm,k,1) + rtur(num,k,1) )
                  kepsed = 0.5 * ( rtur(nm,k,2) + rtur(num,k,2) )
                  !
                  visced = 0.5 * ( vnu3d(nm,k) + vnu3d(num,k) )
                  !
                  ! compose reysyzd (linear and non-linear part)
                  !
                  reysyzd(nm,k) = visced * syzd &
                                  - 1. * cmu * ( ( kepskd ** 3. ) / ( kepsed ** 2. ) ) * ( &
                                         c1 * ( s1yzd ) + &
                                         c2 * ( s2yzd ) + &
                                         c3 * ( s3yzd ) )
                  !
               enddo
               !
            enddo
            !
         enddo
!
! w-momemtum equation
!
         do k = 0, kmax - 1
            !
            do n = nfu, nl
               !
               do m = mfu, ml
                  !
                  md    = m - 1
                  mu    = m + 1
                  nd    = n - 1
                  nu    = n + 1
                  mdd   = m - 2
                  ndd   = n - 2
                  !
                  if ( n == nfu ) ndd = nd
                  !
                  nm    = kgrpnt(m ,n )
                  nmd   = kgrpnt(md,n )
                  nmu   = kgrpnt(mu,n )
```

```
ndm   = kgrpnt(m ,nd)
num   = kgrpnt(m ,nu)
ndmd  = kgrpnt(md,nd)
ndmu  = kgrpnt(mu,nd)
numd  = kgrpnt(md,nu)
numu  = kgrpnt(mu,nu)
nmdd  = kgrpnt(mdd,n)
nddm  = kgrpnt(m ,ndd)
!
kd  = max(k-1,1   )
kdd = max(k-2,1   )
ku  = min(k+1,kmax)
kuu = min(k+2,kmax)
kk  = max(k  ,1   )
kwd  = max(k-1,0   )
kwdd = max(k-2,0   )
kwu  = min(k+1,kmax)
kwuu = min(k+2,kmax)
!
! -w'u' / dx
!
! compose parts of reyszxu
!
dudxu = ( 0.5 * ( u0(nmu,kk) + u0(nmu,k+1 ) ) - 0.5 * ( u0(nmd,kk) + u0(nmd,k+1 ) ) ) &
          / ( 2. * gvu(nm) )
dudyu = ( 0.25 * ( u0(nm,kk) + u0(num,kk) + u0(nm,ku) + u0(num,ku) ) &
          - 0.25 * ( u0(ndm,kk) + u0(nm,kk) + u0(ndm,ku) + u0(nm,ku) ) ) / guu(nm)
dudzu = ( u0(nm,kk) - u0(nm,k+1) ) / ( 0.5 * ( hkum(nm,kk) + hkum(nm,k+1) ) )
!
dvdxu = ( 0.25 * ( v0(nmu,kk) + v0(ndmu,kk) + v0(nmu,ku) + v0(ndmu,ku) ) &
          - 0.25 * ( v0(nm,kk) + v0(ndm,kk) + v0(nm,ku) + v0(ndm,ku) ) ) / gvu(nm)
dvdzu = ( 0.25 * ( v0(ndm,kk) + v0(ndmu,kk) + v0(nm,kk) + v0(nmu,kk) ) &
          - 0.25 * ( v0(ndm,ku) + v0(ndmu,ku) + v0(nm,ku) + v0(nmu,ku) ) ) &
          / ( 0.5 * ( hkum(nm,kk) + hkum(nm,k+1) ) )
!
dwdxu = ( w0(nmu,k) - w0(nm,k) ) / gvu(nm)
dwdyu = ( 0.25 * ( w0(nm,k) + w0(nmu,k) + w0(num,k) + w0(numu,k) ) &
          - 0.25 * ( w0(ndm,k) + w0(ndmu,k) + w0(nm,k) + w0(nmu,k) ) ) / guu(nm)
dwdzu = ( 0.5 * ( w0(nmu,kwd) + w0(nm,kwd ) ) - 0.5 * ( w0(nmu,k+1) + w0(nm,k+1 ) ) ) &
          / ( hkum(nm,kk) + hkum(nm,k+1) )
!
szxu  = dudzu + dwdxu
!
s1zxu = ( dwdxu * dudxu ) + ( dwdyu * dudyu ) + ( dwdzu * dudzu )
s2zxu = 0.5 * ( ( dudxu * dudzu + dwdxu * dudxu ) + ( dudyu * dvdzu + dwdyu * dvdxu ) &
        + ( dudzu * dwdzu + dwdzu * dwdxu ) )
s3zxu = ( dudzu * dudxu ) + ( dvdzu * dvdxu ) + ( dwdzu * dwdxu )
!
kepsku = 0.5 * ( rtur(nmu,k,1) + rtur(nm,k,1) )
kepseu = 0.5 * ( rtur(nmu,k,2) + rtur(nm,k,2) )
!
visceu = 0.5 * ( vnu3d(nmu,k) + vnu3d(nm,k) )
!
! compose reyszxu (linear and non-linear part)
!
reyszxu(nm,k) = visceu * szxu &
                - 1. * cmu * ( ( kepsku ** 3. ) / ( kepseu ** 2. ) ) * ( &
                       c1 * ( s1zxu ) + &
                       c2 * ( s2zxu ) + &
                       c3 * ( s3zxu ) )
!
! compose parts of reyszxd
!
if ( m == mfu ) nmdd = nmd
!
dudxd = ( 0.5 * ( u0(nmu,kk) + u0(nmu,k+1 ) ) - 0.5 * ( u0(nmd,kk) + u0(nmd,k+1 ) ) ) &
          / ( 2. * gvu(nm) )
dudyd = ( 0.25 * ( u0(nm,kk) + u0(num,kk) + u0(nm,ku) + u0(num,ku) ) &
          - 0.25 * ( u0(ndm,kk) + u0(nm,kk) + u0(ndm,ku) + u0(nm,ku) ) ) / guu(nm)
dudzd = ( u0(nm,kk) - u0(nm,k+1) ) / ( 0.5 * ( hkum(nm,kk) + hkum(nm,k+1) ) )
!
```

```
dvdxd = ( 0.25 * ( v0(nmu,kk) + v0(ndmu,kk) + v0(nmu,ku) + v0(ndmu,ku) ) &
          - 0.25 * ( v0(nm,kk) + v0(ndm,kk) + v0(nm,ku) + v0(ndm,ku) ) ) / gvu(nm)
dvdzd = ( 0.25 * ( v0(ndm,kk) + v0(ndmu,kk) + v0(nm,kk) + v0(nmu,kk) ) &
          - 0.25 * ( v0(ndm,ku) + v0(ndmu,ku) + v0(nm,ku) + v0(nmu,ku) ) ) &
          / ( 0.5 * ( hkum(nm,kk) + hkum(nm,k+1) ) )
!
dwdxd = ( w0(nmu,k) - w0(nm,k) ) / gvu(nm)
dwdyd = ( 0.25 * ( w0(nm,k) + w0(nmu,k) + w0(num,k) + w0(numu,k) ) &
          - 0.25 * ( w0(ndm,k) + w0(ndmu,k) + w0(nm,k) + w0(nmu,k) ) ) / guu(nm)
dwdzd = ( 0.5 * ( w0(nmu,kwd) + w0(nm,kwd ) ) - 0.5 * ( w0(nmu,k+1) + w0(nm,k+1 ) ) ) &
          / ( hkum(nm,kk) + hkum(nm,k+1) )
!
szxd  = dudzd + dwdxd
!
s1zxd = ( dwdxd * dudxd ) + ( dwdyd * dudyd ) + ( dwdzd * dudzd )
s2zxd = 0.5 * ( ( dudxd * dudzd + dwdxd * dudxd ) + ( dudyd * dvdzd + dwdyd * dvdxd ) &
        + ( dudzd * dwdzd + dwdzd * dwdxd ) )
s3zxd = ( dudzd * dudxd ) + ( dvdzd * dvdxd ) + ( dwdzd * dwdxd )
!
kepskd = 0.5 * ( rtur(nmd,k,1) + rtur(nm,k,1) )
kepsed = 0.5 * ( rtur(nmd,k,2) + rtur(nm,k,2) )
!
visced = 0.5 * ( vnu3d(nmd,k) + vnu3d(nm,k) )
!
! compose reyszxd (linear and non-linear part)
!
reyszxd(nm,k) = visced * szxd &
                - 1. * cmu * ( ( kepskd ** 3. ) / ( kepsed ** 2. ) ) * ( &
                       c1 * ( s1zxd ) + &
                       c2 * ( s2zxd ) + &
                       c3 * ( s3zxd ) )
!
! -w'v' / dy
!
! compose parts of reyswyu
!
dudyu = ( 0.25 * ( u0(num,kk) + u0(numd,kk) + u0(num,ku) + u0(numd,ku) ) &
          - 0.25 * ( u0(nm,kk) + u0(nmd,kk) + u0(nm,ku) + u0(nmd,ku) ) ) / guv(nm)
dudzu = ( 0.25 * ( u0(nm,kk) + u0(nmd,kk) + u0(num,kk) + u0(numd,kk) ) &
          - 0.25 * ( u0(nm,ku) + u0(nmd,ku) + u0(num,ku) + u0(numd,ku) ) ) &
          / ( 0.5 * ( hkvm(nm,kk) + hkvm(nm,k+1) ) )
!
dvdxu = ( 0.25 * ( v0(nm,kk) + v0(nmu,kk) + v0(nm,ku) + v0(nmu,ku) ) &
          - 0.25 * ( v0(nmd,kk) + v0(nm,kk) + v0(nmd,ku) + v0(nm,ku) ) ) / gvv(nm)
dvdyu = ( 0.5 * ( v0(num,kk) + v0(num,k+1) ) - 0.5 * ( v0(ndm,kk) + v0(ndm,k+1) ) ) &
          / ( 2. * guv(nm) )
dvdzu = ( v0(nm,kk) - v0(nm,k+1) ) / ( 0.5 * ( hkvm(nm,kk) + hkvm(nm,k+1) ) )
!
dwdxu = ( 0.25 * ( w0(nm,k) + w0(nmu,k) + w0(num,k) + w0(numu,k) ) &
          - 0.25 * ( w0(nmd,k) + w0(nm,k) + w0(numd,k) + w0(num,k) ) ) / gvv(nm)
dwdyu = ( w0(num,k) - w0(nm,k) ) / guv(nm)
dwdzu = ( 0.5 * ( w0(nm,kwd) + w0(num,kwd) ) - 0.5 * ( w0(nm,k+1) + w0(num,k+1) ) ) &
          / ( hkvm(nm,kk) + hkvm(nm,k+1) )
!
szyu  = dwdyu + dvdzu
!
s1zyu = ( dwdxu * dvdxu ) + ( dwdyu * dvdyu ) + ( dwdzu * dvdzu )
s2zyu = 0.5 * ( ( dwdxu * dudyu + dvdxu * dudzu ) + ( dwdyu * dvdyu + dvdyu * dvdzu ) &
        + ( dwdzu * dwdyu + dvdzu * dwdzu ) )
s3zyu = ( dudyu * dudzu ) + ( dvdyu * dvdzu ) + ( dwdyu * dwdzu )
!
kepsku = 0.5 * ( rtur(nm,k,1) + rtur(num,k,1) )
kepseu = 0.5 * ( rtur(nm,k,2) + rtur(num,k,2) )
!
visceu = 0.5 * ( vnu3d(nm,k) + vnu3d(num,k) )
!
! compose reyswyu (linear and non-linear part)
!
reyszyu(nm,k) = visceu * szyu &
                - 1. * cmu * ( ( kepsku ** 3. ) / ( kepseu ** 2. ) ) * ( &
                       c1 * ( s1zyu ) + &
```

```
                      c2 * ( s2zyu ) + &
                      c3 * ( s3zyu ) )
!
! compose parts of reyswyd
!
dudyd = ( 0.25 * ( u0(nm,kk) + u0(nmd,kk) + u0(nm,ku) + u0(nmd,ku) ) &
          - 0.25 * ( u0(ndm,kk) + u0(ndmd,kk) + u0(ndm,ku) + u0(ndmd,ku) ) ) / guv(ndm)
dudzd = ( 0.25 * ( u0(ndm,kk) + u0(ndmd,kk) + u0(nm,kk) + u0(nmd,kk) ) &
          - 0.25 * ( u0(ndm,ku) + u0(ndmd,ku) + u0(nm,ku) + u0(nmd,ku) ) ) &
          / ( 0.5 * ( hkvm(ndm,kk) + hkvm(ndm,k+1) ) )
!
dvdxd = ( 0.25 * ( v0(ndm,kk) + v0(ndmu,kk) + v0(ndm,ku) + v0(ndmu,ku) ) &
          - 0.25 * ( v0(ndmd,kk) + v0(ndm,kk) + v0(ndmd,ku) + v0(ndm,ku) ) ) / gvv(ndm)
dvdyd = ( 0.5 * ( v0(nm,kk) + v0(nm,k+1) ) - 0.5 * ( v0(nddm,kk) + v0(nddm,k+1) ) ) &
          / ( 2. * guv(ndm) )
dvdzd = ( v0(ndm,kk) - v0(ndm,k+1) ) / ( 0.5 * ( hkvm(ndm,kk) + hkvm(ndm,k+1) ) )
!
dwdxd = ( 0.25 * ( w0(ndm,k) + w0(ndmu,k) + w0(nm,k) + w0(nmu,k) ) &
          - 0.25 * ( w0(ndmd,k) + w0(ndm,k) + w0(nmd,k) + w0(nm,k) ) ) / gvv(ndm)
dwdyd = ( w0(nm,k) - w0(ndm,k) ) / guv(ndm)
dwdzd = ( 0.5 * ( w0(ndm,kwd) + w0(nm,kwd) ) - 0.5 * ( w0(ndm,k+1) + w0(nm,k+1) ) ) &
          / ( hkvm(ndm,kk) + hkvm(ndm,k+1) )
!
szyd  = dwdyd + dvdzd
!
s1zyd = ( dwdxd * dvdxd ) + ( dwdyd * dvdyd ) + ( dwdzd * dvdzd )
s2zyd = 0.5 * ( ( dwdxd * dudyd + dvdxd * dudzd ) + ( dwdyd * dvdyd + dvdyd * dvdzd ) &
        + ( dwdzd * dwdyd + dvdzd * dwdzd ) )
s3zyd = ( dudyd * dudzd ) + ( dvdyd * dvdzd ) + ( dwdyd * dwdzd )
!
kepskd = 0.5 * ( rtur(nm,k,1) + rtur(ndm,k,1) )
kepsed = 0.5 * ( rtur(nm,k,2) + rtur(ndm,k,2) )
!
visced = 0.5 * ( vnu3d(nm,k) + vnu3d(ndm,k) )
!
! compose reyswyd (linear and non-linear part)
!
reyszyd(nm,k) = visced * szyd &
                - 1. * cmu * ( ( kepskd ** 3. ) / ( kepsed ** 2. ) ) * ( &
                      c1 * ( s1zyd ) + &
                      c2 * ( s2zyd ) + &
                      c3 * ( s3zyd ) )
!
! -w'w' / dz
!
! compose parts of reyszzu
!
dudxu = ( u0(nm,kk) - u0(nmd,kk) ) / ( 0.5 * ( gvv(nm) + gvv(ndm) ) )
dudyu = ( 0.25 * ( u0(nm,kk) + u0(nmd,kk) + u0(num,kk) + u0(numd,kk) ) &
          - 0.25 * ( u0(ndm,kk) + u0(ndmd,kk) + u0(nm,kk) + u0(nmd,kk) ) ) &
          / ( 0.5 * ( guu(nm) + guu(nmd) ) )
dudzu = ( 0.25 * ( u0(nm,kk) + u0(nmd,kk) + u0(nm,kd) + u0(nmd,kd) ) &
          - 0.25 * ( u0(nm,kk) + u0(nmd,kk) + u0(nm,ku) + u0(nmd,ku) ) ) / hks(nm,kk)
!
dvdxu = ( 0.25 * ( v0(nm,kk) + v0(nmu,kk) + v0(ndm,kk) + v0(ndmu,kk) ) &
          - 0.25 * ( v0(nmd,kk) + v0(nm,kk) + v0(ndmd,kk) + v0(ndm,kk) ) ) &
          / ( 0.5 * ( gvv(nm) + gvv(ndm) ) )
dvdyu = ( v0(nm,kk) - v0(ndm,kk) ) / ( 0.5 * ( guu(nm) + guu(nmd) ) )
dvdzu = ( 0.25 * ( v0(nm,kk) + v0(ndm,kk) + v0(nm,kd) + v0(ndm,kd) ) &
          - 0.25 * ( v0(nm,kk) + v0(ndm,kk) + v0(nm,ku) + v0(ndm,ku) ) ) / hks(nm,kk)
!
dwdxu = ( 0.25 * ( w0(nm,kk) + w0(nmu,kk) + w0(nm,kd) + w0(nmu,kd) ) &
          - 0.25 * ( w0(nmd,kk) + w0(nm,kk) + w0(nmd,kd) + w0(nm,kd) ) ) &
          / ( 0.5 * ( gvv(nm) + gvv(ndm) ) )
dwdyu = ( 0.25 * ( w0(nm,kk) + w0(num,kk) + w0(nm,kd) + w0(num,kd) ) &
          - 0.25 * ( w0(ndm,kk) + w0(nm,kk) + w0(ndm,kd) + w0(nm,kd) ) ) &
          / ( 0.5 * ( guu(nm) + guu(nmd) ) )
dwdzu = ( w0(nm,kwd) - w0(nm,k) ) / hks(nm,kk)
!
szzu  = dwdzu + dwdzu
!
```

```
s1xxu = ( dudxu * dudxu ) + ( dudyu * dudyu ) + ( dudzu * dudzu )
s1yyu = ( dvdxu * dvdxu ) + ( dvdyu * dvdyu ) + ( dvdzu * dvdzu )
s1zzu = ( dwdxu * dwdxu ) + ( dwdyu * dwdyu ) + ( dwdzu * dwdzu )
!
s2xxu = 0.5 * ( ( dudxu * dudxu + dudxu * dudxu ) + ( dudyu * dvdxu + dudyu * dvdxu ) &
        + ( dudzu * dwdxu + dudzu * dwdxu ) )
s2yyu = 0.5 * ( ( dvdxu * dudyu + dvdxu * dudyu ) + ( dvdyu * dvdyu + dvdyu * dvdyu ) &
        + ( dvdzu * dwdyu + dvdzu * dwdyu ) )
s2zzu = 0.5 * ( ( dwdxu * dudzu + dwdxu * dudzu ) + ( dwdyu * dvdzu + dwdyu * dvdzu ) &
        + ( dwdzu * dwdzu + dwdzu * dwdzu ) )
!
s3xxu = ( dudxu * dudxu ) + ( dvdxu * dvdxu ) + ( dwdxu * dwdxu )
s3yyu = ( dudyu * dudyu ) + ( dvdyu * dvdyu ) + ( dwdyu * dwdyu )
s3zzu = ( dudzu * dudzu ) + ( dvdzu * dvdzu ) + ( dwdzu * dwdzu )
!
kepsku = 0.5 * ( rtur(nm,k,1) + rtur(nm,kwd,1) )
kepseu = 0.5 * ( rtur(nm,k,2) + rtur(nm,kwd,2) )
!
visceu = 0.5 * ( vnu3d(nm,k) + vnu3d(nm,kwd) )
!
! compose reyszzu (linear and non-linear part)
!
reyszzu(nm,k) = visceu * szzu &
                - 1. * cmu * ( ( kepsku ** 3. ) / ( kepseu ** 2. ) ) * ( &
                       c1 * ( s1zzu - ( 1. / 3. ) * ( s1xxu + s1yyu + s1zzu ) ) + &
                       c2 * ( s2zzu - ( 1. / 3. ) * ( s2xxu + s2yyu + s2zzu ) ) + &
                       c3 * ( s3zzu - ( 1. / 3. ) * ( s3xxu + s3yyu + s3zzu ) ) )
!
! compose parts of reyszzd
!
dudxd = ( u0(nm,ku) - u0(nmd,ku) ) / ( 0.5 * ( gvv(nm) + gvv(ndm) ) )
dudyd = ( 0.25 * ( u0(nm,ku) + u0(nmd,ku) + u0(num,ku) + u0(numd,ku) ) &
          - 0.25 * ( u0(ndm,ku) + u0(ndmd,ku) + u0(nm,ku) + u0(nmd,ku) ) ) &
          / ( 0.5 * ( guu(nm) + guu(nmd) ) )
dudzd = ( 0.25 * ( u0(nm,ku) + u0(nmd,ku) + u0(nm,kk) + u0(nmd,kk) ) &
          - 0.25 * ( u0(nm,ku) + u0(nmd,ku) + u0(nm,kuu) + u0(nmd,kuu) ) ) / hks(nm,ku)
!
dvdxd = ( 0.25 * ( v0(nm,ku) + v0(nmu,ku) + v0(ndm,ku) + v0(ndmu,ku) ) &
          - 0.25 * ( v0(nmd,ku) + v0(nm,ku) + v0(ndmd,ku) + v0(ndm,ku) ) ) &
          / ( 0.5 * ( gvv(nm) + gvv(ndm) ) )
dvdyd = ( v0(nm,ku) - v0(ndm,ku) ) / ( 0.5 * ( guu(nm) + guu(nmd) ) )
dvdzd = ( 0.25 * ( v0(nm,ku) + v0(ndm,ku) + v0(nm,kk) + v0(ndm,kk) ) &
          - 0.25 * ( v0(nm,ku) + v0(ndm,ku) + v0(nm,kuu) + v0(ndm,kuu) ) ) / hks(nm,ku)
!
dwdxd = ( 0.25 * ( w0(nm,ku) + w0(nmu,ku) + w0(nm,kk) + w0(nmu,kk) ) &
          - 0.25 * ( w0(nmd,ku) + w0(nm,ku) + w0(nmd,kk) + w0(nm,kk) ) ) &
          / ( 0.5 * ( gvv(nm) + gvv(ndm) ) )
dwdyd = ( 0.25 * ( w0(nm,ku) + w0(num,ku) + w0(nm,kk) + w0(num,kk) ) &
          - 0.25 * ( w0(ndm,ku) + w0(nm,ku) + w0(ndm,kk) + w0(nm,kk) ) ) &
          / ( 0.5 * ( guu(nm) + guu(nmd) ) )
dwdzd = ( w0(nm,kk) - w0(nm,ku) ) / hks(nm,ku)
!
szzd  = dwdzd + dwdzd
!
s1xxd = ( dudxd * dudxd ) + ( dudyd * dudyd ) + ( dudzd * dudzd )
s1yyd = ( dvdxd * dvdxd ) + ( dvdyd * dvdyd ) + ( dvdzd * dvdzd )
s1zzd = ( dwdxd * dwdxd ) + ( dwdyd * dwdyd ) + ( dwdzd * dwdzd )
!
s2xxd = 0.5 * ( ( dudxd * dudxd + dudxd * dudxd ) + ( dudyd * dvdxd + dudyd * dvdxd ) &
        + ( dudzd * dwdxd + dudzd * dwdxd ) )
s2yyd = 0.5 * ( ( dvdxd * dudyd + dvdxd * dudyd ) + ( dvdyd * dvdyd + dvdyd * dvdyd ) &
        + ( dvdzd * dwdyd + dvdzd * dwdyd ) )
s2zzd = 0.5 * ( ( dwdxd * dudzd + dwdxd * dudzd ) + ( dwdyd * dvdzd + dwdyd * dvdzd ) &
        + ( dwdzd * dwdzd + dwdzd * dwdzd ) )
!
s3xxd = ( dudxd * dudxd ) + ( dvdxd * dvdxd ) + ( dwdxd * dwdxd )
s3yyd = ( dudyd * dudyd ) + ( dvdyd * dvdyd ) + ( dwdyd * dwdyd )
s3zzd = ( dudzd * dudzd ) + ( dvdzd * dvdzd ) + ( dwdzd * dwdzd )
!
kepskd = 0.5 * ( rtur(nm,k,1) + rtur(nm,k+1,1) )
kepsed = 0.5 * ( rtur(nm,k,2) + rtur(nm,k+1,2) )
```

```
                !
                visced = 0.5 * ( vnu3d(nm,k) + vnu3d(nm,k+1) )
                !
                ! compose reyszzd (linear and non-linear part)
                !
                reyszzd(nm,k) = visced * szzd &
                                - 1. * cmu * ( ( kepskd ** 3. ) / ( kepsed ** 2. ) ) * ( &
                                       c1 * ( s1zzd - ( 1. / 3. ) * ( s1xxd + s1yyd + s1zzd ) ) + &
                                       c2 * ( s2zzd - ( 1. / 3. ) * ( s2xxd + s2yyd + s2zzd ) ) + &
                                       c3 * ( s3zzd - ( 1. / 3. ) * ( s3xxd + s3yyd + s3zzd ) ) )
                !
             enddo
             !
          enddo
          !
       enddo
end subroutine SwashKepsNonLin
```

The components of the Reynolds stress tensor are explicitly added to the momentum equations in the module: SwashExpLay2DHflow.ftn90. Only the changed part of the module is printed.

```
 subroutine SwashExpLay2DHflow
    !
    if ( iturb == 1 ) then
       !
       do k = 1, kmax
          !
          do n = nfu, nl
             !
             do m = mf + 1, ml - 1
                !
                nm   = kgrpnt(m ,n )
                !
                if ( uwetp(nm) ) then
                   !
                   lmask = .true.
                   !
                   ! next, compute the divergence of the Reynolds stresses
                   !
                   rhsu(nm,k) = rhsu(nm,k) + ( reysxxu(nm,k) - reysxxd(nm,k) ) / gvu(nm) + &
                                             ( reysxyu(nm,k) - reysxyd(nm,k) ) / guu(nm) + &
                                             ( reysxzu(nm,k) - reysxzd(nm,k) ) / hkum(nm,k)
                   !
                endif
                !
             enddo
             !
          enddo
          !
       enddo
       !
    endif
    !
    if ( iturb == 1 ) then
       !
       do k = 1, kmax
          !
          do m = mfu, ml
             !
             do n = nfu, nl - 1
                !
                nm   = kgrpnt(m ,n )
                !
                if ( vwetp(nm) ) then
                   !
                   ! next, compute the divergence of the Reynolds stresses
                   !
                   rhsu(nm,k) = rhsu(nm,k) + ( reysyxu(nm,k) - reysyxd(nm,k) ) / gvv(nm) + &
                                             ( reysyyu(nm,k) - reysyyd(nm,k) ) / guv(nm) + &
```

```
                                        ( reysyzu(nm,k) - reysyzd(nm,k) ) / hkvm(nm,k)
               !
            endif
            !
         enddo
         !
      enddo
      !
   enddo
   !
endif
!
if ( iturb == 1 ) then
   !
   do k = 0, kmax-1
      !
      do m = mfu, ml
         !
         do n = nfu, nl
            !
            nd = n - 1
            md = m - 1
            !
            nm   = kgrpnt(m ,n )
            ndm  = kgrpnt(m ,nd)
            nmd  = kgrpnt(md,n )
            !
            kk  = max(k  ,1   )
            !
            if ( vwetp(nm) ) then
               !
               ! next, compute the divergence of the Reynolds stresses
               !
               rhsw(nm,k) = rhsw(nm,k) + ( reyszxu(nm,k) - reyszxd(nm,k) ) / &
                                         ( 0.5 * ( gvv(nm) + gvv(nmd) ) ) + &
                                         ( reyszyu(nm,k) - reyszyd(nm,k) ) / &
                                         ( 0.5 * ( guu(nm) + guu(ndm) ) ) + &
                                         ( reyszzu(nm,k) - reyszzd(nm,k) ) / &
                                         ( 0.5 * ( hks(nm,kk) + hks(nm,k+1) ) )
               !
            endif
            !
         enddo
         !
      enddo
      !
   enddo
   !
endif
```

By default the side walls are treated with no slip boundary conditions. Since the grid spacing usually does not allow for the use of no slip conditions a partial slip conditions is implemented. The wall shear stress is implicitly treated and added to the matrix. The wall shear stress is either based on hydraulically smooth or rough conditions.

```
   do m = mfu, ml
      do k = 1, kmax
         !
         nfum = kgrpnt(m,nfu)
         nlm  = kgrpnt(m,nl)
         !
         ! hydraulically rough or smooth conditions
         !
         if ( fricf(nfum,2) > 0 ) then
            !
            amatu(nfum,k,1) = amatu(nfum,k,1) + ( ( ( 0.09 ** 0.25 ) * sqrt( rtur(nfum,k,1) ) ) &
                              / ( log( 33.0 * 0.5 * guu(nfum) / fricf(nfum,2) ) / vonkar ) )  / guu(nfum)
         else
```

```
            !
            amatu(nfum,k,1) = amatu(nfum,k,1) + ( ( ( 0.09 ** 0.25 ) * sqrt( rtur(nfum,k,1) ) ) &
                              / ( log( 9.0 * ( ( ( ( 0.09 ** 0.25 ) * sqrt( rtur(nfum,k,1) ) ) &
                              * .5 * guu(nfum) ) / kinvis ) ) / vonkar ) )  / guu(nfum)
         endif
         !
         if ( fricf(nlm,2) > 0 ) then
            !
            amatu(nlm ,k,1) = amatu(nlm ,k,1) + ( ( ( 0.09 ** 0.25 ) * sqrt( rtur(nlm, k,1) ) ) &
                              / ( log( 33.0 * 0.5 * guu(nlm ) / fricf(nlm,2) ) / vonkar ) )  / guu(nlm)
            !
         else
            !
            amatu(nlm ,k,1) = amatu(nlm ,k,1) + ( ( ( 0.09 ** 0.25 ) * sqrt( rtur(nlm, k,1) ) ) &
                              / ( log( 9.0 * ( ( ( ( 0.09 ** 0.25 ) * sqrt( rtur(nlm, k,1) ) ) &
                              * .5 * guu(nlm)  ) / kinvis ) ) / vonkar ) )  / guu(nlm)
            !
         endif
         !
      enddo
   enddo
```

In addition the normal velocity gradients at the virtual cells are set to zero.

```
do m = mf, ml
   !
   nfm  = kgrpnt(m,nf )
   nfum = kgrpnt(m,nfu)
   nlm  = kgrpnt(m,nl )
   nlum = kgrpnt(m,nlu)
   !
   if ( ibb(m) == 1 .and. LMYF ) then
      !
      ! set to zero at closed boundary
      !
      u1(nfm,:) = u1(nfum,:)!0.
      !
   else if ( ibb(m) > 1 .and. LMYF ) then
      !
      ! zero normal gradient condition at open boundaries
      !
      u1(nfm,:) = u1(nfum,:)
      !
   endif
   !
   if ( ibt(m) == 1 .and. LMYL ) then
      !
      ! set to zero at closed boundary
      !
      u1(nlum,:) = u1(nlm,:) !0.
      !
   else if ( ibt(m) > 1 .and. LMYL ) then
      !
      ! zero normal gradient condition at open boundaries
      !
      u1(nlum,:) = u1(nlm,:)
      !
   endif
   !
enddo
```

B. SWASH subroutines for LES with a standard Smagorinsky closure

The new subroutine SwashSubVisc is used to calculate the subgrid viscosity with the standard Smagorinsky model. The subgrid viscosity is saved in the vnu3d(:,:) matrix which is originally used for the vertical eddy viscosity of the standard k-ε model.

```
subroutine SwashSubVisc
!
!   --|------------------------------------------------------------|--
!     | Delft University of Technology                             |
!     | Faculty of Civil Engineering                               |
!     | Environmental Fluid Mechanics Section                      |
!     | P.O. Box 5048, 2600 GA  Delft, The Netherlands             |
!     |                                                            |
!     | Programmers: The SWASH team                                |
!   --|------------------------------------------------------------|--
!
!
!     SWASH (Simulating WAves till SHore); a non-hydrostatic wave-flow model
!     Copyright (C) 2010-2011  Delft University of Technology
!
!     This program is free software; you can redistribute it and/or
!     modify it under the terms of the GNU General Public License as
!     published by the Free Software Foundation; either version 2 of
!     the License, or (at your option) any later version.
!
!     This program is distributed in the hope that it will be useful,
!     but WITHOUT ANY WARRANTY; without even the implied warranty of
!     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
!     GNU General Public License for more details.
!
!     A copy of the GNU General Public License is available at
!     http://www.gnu.org/copyleft/gpl.html#SEC3
!     or by writing to the Free Software Foundation, Inc.,
!     59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
!
!
!   Authors
!
!    1.00: Tom Bogaard
!
!   Updates
!
!    1.00, January 2012: New subroutine
!
!   Purpose
!
!   Calculates subgrid viscosity coefficient
!
!   Method
!
!   The Smagorinsky model is utilized to account for subgrid turbulent mixing
!
!   Note: if subgrid viscosity is larger than maximum, which is based on stability criterion, apply clipping
!
!   Modules used
!
    use ocpcomm4
    use SwashCommdata3
    use m_genarr
    use m_parall
    use SwashFlowdata
    use SwashTimeComm
!
    implicit none
!
!   Argument variables
```

```
!
!   Local variables
!
    integer             :: icistb       ! counter for number of instable points
    integer, save       :: ient = 0     ! number of entries in this subroutine
    integer             :: k            ! loop counter
    integer             :: kd           ! index of layer k-1
    integer             :: ku           ! index of layer k+1
    integer             :: kend         ! end index of loop over layers
    integer             :: ksta         ! start index of loop over layers
    integer             :: m            ! loop counter
    integer             :: n            ! loop counter
    integer             :: md           ! index of point m-1
    integer             :: mend         ! end index of loop over u-points
    integer             :: msta         ! start index of loop over u-points
    integer             :: mu           ! index of point m+1
    integer             :: nd           ! index of point n-1
    integer             :: ndm          ! pointer to m,n-1
    integer             :: ndmu         ! pointer to m+1,n-1
    integer             :: nend         ! end index of loop over v-points
    integer             :: nfm          ! pointer to m,nf
    integer             :: nfum         ! pointer to m,nfu
    integer             :: nldm         ! pointer to m,nl-1
    integer             :: nlm          ! pointer to m,nl
    integer             :: nm           ! pointer to m,n
    integer             :: nmd          ! pointer to m-1,n
    integer             :: nmf          ! pointer to mf,n
    integer             :: nmfu         ! pointer to mfu,n
    integer             :: nml          ! pointer to ml,n
    integer             :: nmld         ! pointer to ml-1,n
    integer             :: nmlu         ! pointer to mlu,n
    integer             :: nmu          ! pointer to m+1,n
    integer             :: nsta         ! start index of loop over v-points
    integer             :: nu           ! index of point n+1
    integer             :: num          ! pointer to m,n+1
    integer             :: numd         ! pointer to m-1,n+1
    integer             :: numu         ! pointer to m+1,n+1
    !
    real                :: dxl          ! local mesh size in x-direction
    real                :: dyl          ! local mesh size in y-direction
    real                :: dzl          ! local mesh size in z-direction
    !
    real                :: dudx         ! local velocity gradient dudx
    real                :: dudy         ! local velocity gradient dudy
    real                :: dudz         ! local velocity gradient dudz
    real                :: dvdx         ! local velocity gradient dvdx
    real                :: dvdy         ! local velocity gradient dvdy
    real                :: dvdz         ! local velocity gradient dvdz
    real                :: dwdx         ! local velocity gradient dwdx
    real                :: dwdy         ! local velocity gradient dwdy
    real                :: dwdz         ! local velocity gradient dwdz
    !
    real                :: zs           ! distance from bottom
    real                :: cs           ! Smagorinsky constant
    !
    real                :: rproc        ! auxiliary variable with percentage of instable points
    real                :: stabmx       ! auxiliary variable with maximum viscosity based stability criterion
    !
    character(80)       :: msgstr       ! string to pass message
!
!   Structure
!
!   Description of the pseudo code
!
!   Source text
!
    if (ltrace) call strace (ient,'SwashSubVisc')
    !
    icistb = 0
    !
    ! Smagorinsky model or mixing length model
```

```
!
! compute eddy viscosity coefficients at internal depth points only
!
! In case of a subdomain interface not coinciding with a real boundary, the begin point and/or
! end point of the loop becomes mf and ml, respectively, in order to save on communication
! (redundant calculation for vnu2d). The same for y-direction.
!
msta = mf + 1                ! first internal depth point in x-direction
if ( .not.LMXF ) msta = mf
mend = ml - 1                ! last  internal depth point in x-direction
if ( .not.LMXL ) mend = ml
nsta = nf + 1                ! first internal depth point in y-direction
if ( .not.LMYF ) nsta = nf
nend = nl - 1                ! last  internal depth point in y-direction
if ( .not.LMYL ) nend = nl
ksta = 1.
kend = kmax
!
do k = ksta, kend
   !
   do n = nsta, nend
      !
      do m = msta, mend
         !
         md = m - 1
         mu = m + 1
         nd = n - 1
         nu = n + 1
         !
         nm   = kgrpnt(m ,n )
         nmd  = kgrpnt(md,n )
         nmu  = kgrpnt(mu,n )
         ndm  = kgrpnt(m ,nd)
         num  = kgrpnt(m ,nu)
         ndmu = kgrpnt(mu,nd)
         numd = kgrpnt(md,nu)
         numu = kgrpnt(mu,nu)
         !
         ! for permanently dry neighbours, corresponding values will be mirrored
         !
         if ( nmd  == 1 ) nmd  = nm
         if ( nmu  == 1 ) nmu  = nm
         if ( ndm  == 1 ) ndm  = nm
         if ( num  == 1 ) num  = nm
         if ( ndmu == 1 ) ndmu = ndm
         if ( numd == 1 ) numd = nmd
         if ( numu == 1 ) numu = nmu
         !
         kd  = max(k-1,1   )
         ku  = min(k+1,kmax)
         !
         ! local mesh sizes
         !
         dxl = 0.5 * ( gvv(nm) + gvv(nmu) )
         dyl = 0.5 * ( guu(nm) + guu(num) )
         dzl = 0.25 * ( hks(nm,k) + hks(nmu,k) + hks(numu,k) + hks(num,k) )
         !
         ! local velocity gradients
         !
         dudx = ( u0(nmu,k) + u0(numu,k) - u0(nmd,k) - u0(numd,k) ) / ( 4. * dxl )
         dudy = ( u0(num,k) - u0(nm,k) ) / dyl
         dudz = ( ( 0.25 * ( u0(nm,kd) + u0(num,kd) + u0(nm,k) + u0(num,k) ) ) &
                - ( 0.25 * ( u0(nm,k) + u0(num,k) + u0(nm,ku) + u0(num,ku) ) ) ) / dzl
         !
         dvdx = ( v0(nmu,k) - v0(nm,k) ) / dxl
         dvdy = ( v0(num,k) + v0(numu,k) - v0(ndm,k) - v0(ndmu,k) ) / ( 4. * dyl )
         dvdz = ( ( 0.25 * ( v0(nm,k) + v0(nmu,k) + v0(nm,kd) + v0(nmu,kd) ) ) &
                - ( 0.25 * ( v0(nm,ku) + v0(nmu,ku) + v0(nm,k) + v0(nmu,k) ) ) ) / dzl
         !
         dwdx = ( 0.25 * ( w0(nmu,k) + w0(numu,k) + w0(nmu,k-1) - w0(numu,k-1) ) &
                - 0.25 * ( w0(nm,k) + w0(num,k) + w0(nm,k-1) - w0(num,k-1) ) ) /  dxl
```

```
         dwdy = ( 0.25 * ( w0(num,k) + w0(numu,k) + w0(num,k-1) - w0(numu,k-1) ) &
                - 0.25 * ( w0(nm,k) + w0(nmu,k) + w0(nm,k-1) - w0(nmu,k-1) ) ) /  dyl
         dwdz = ( ( 0.25 * ( w0(nm,k-1) + w0(nmu,k-1) + w0(numu,k-1) + w0(num,k-1) ) ) &
                -  ( 0.25 * ( w0(nm,k) + w0(nmu,k) + w0(numu,k) + w0(num,k) ) ) ) / dzl
         !
         ! van Driest damping function ( u* based on inflow boundary condition and smooth wall conditions )
         !
         zs = 0.5 * ( zks(nm,kmax-1) - zks(nm,kmax) ) * ( kmax - k + 1 ) * 2 - 0.5 &
              * ( zks(nm,kmax-1) - zks(nm,kmax) )
         !
         cs = 0.065 * ( 1 - exp( - ( ( ( 0.00536 * zs ) / 1E-06 ) / 26. ) ) )
         !
         ! compute subgrid viscosity
         !
         vnu3d(nm,k) = 1E-06 + cs * cs * ( dxl * dyl * dzl ) ** ( 2. / 3. ) &
                       * sqrt( 0.5 * ( 4. * ( dudx ) ** 2. + 2. * ( dudy + dvdx ) ** 2. + 2. &
                       * ( dudz + dwdx ) ** 2. + 4. * ( dvdy ) ** 2. + 2 * ( dvdz + dwdy ) ** 2. &
                       + 4 * ( dwdz ) ** 2. ) )
         !
         ! stability criterium based on horizontal mesh sizes
         !
         stabmx = 1./(2.*dt*(1./(dxl*dxl) + 1./(dyl*dyl)))
         !
         if ( .not. vnu3d(nm,k) < stabmx ) then
            vnu3d(nm,k) = stabmx
            icistb    = icistb + 1
         endif
         !
      enddo
      !
   enddo
   !
enddo
!
! no slip condition for subgrid viscosity at k = 0 and k = kmax + 1
!
do n = nsta, nend
   !
   do m = msta, mend
      !
      nm = kgrpnt(m ,n)
      !
      vnu3d(nm,0) = vnu3d(nm,1)
      !
      vnu3d(nm,kmax+1) = vnu3d(nm,kmax)
      !
   enddo
   !
enddo
!
! set to zero for permanently dry points
!
do k = 0, kmax + 1
   !
   vnu3d(1,k) = 0.
   !
enddo
!
! at real boundaries, copy them from internal points
!
do k = 0, kmax + 1
   !
   do n = nsta, nend
      !
      nmf  = kgrpnt(mf  ,n)
      nmfu = kgrpnt(mfu ,n)
      nmld = kgrpnt(ml-1,n)
      nml  = kgrpnt(ml  ,n)
      !
      if ( LMXF ) then
         !
```

```
            vnu3d(nmf,k) = vnu3d(nmfu,k)
            !
         endif
         !
         if ( LMXL ) then
            !
            vnu3d(nml,k) = vnu3d(nmld,k)
            !
         endif
         !
      enddo
      !
   enddo
   !
   do k = 0, kmax + 1
      !
      do m = mf, ml
         !
         nfm   = kgrpnt(m,nf  )
         nfum  = kgrpnt(m,nfu )
         nldm  = kgrpnt(m,nl-1)
         nlm   = kgrpnt(m,nl  )
         !
         if ( LMYF ) then
            !
            vnu3d(nfm,k) = vnu3d(nfum,k)
            !
         endif
         !
         if ( LMYL ) then
            !
            vnu3d(nlm,k)  = vnu3d(nldm,k)
            !
         endif
         !
      enddo
      !
   enddo
   !
   ! give warning for instable points
   !
   if ( icistb > 0 ) then
      !
      rproc = 100.*real(icistb)/ ( real(mcgrd) * kmax )
      !
      if ( .not. rproc < 1. ) then
         write (msgstr,'(a,f5.1)') &
         'percentage of instable points for computing horizontal eddy viscosity = ',rproc
         call msgerr (1, trim(msgstr) )
      endif
      !
   endif
   !
end subroutine SwashSubVisc
```

The new subroutine SwashLes computes the components subgrid stress tensor. The 2nd derivatives in vertical direction are implicitly treated and directly added to the matrix. They are computed in SwashExpLay2DHflow. All subgrid stresses are added to the momentum equations in the module SwashExpLay2DHflow.

```
subroutine SwashLes
!
!   --|------------------------------------------------------------|--
!     | Delft University of Technology                             |
!     | Faculty of Civil Engineering                               |
!     | Environmental Fluid Mechanics Section                      |
!     | P.O. Box 5048, 2600 GA  Delft, The Netherlands             |
!     |                                                            |
!     | Programmers: The SWASH team                                |
```

```
!   --|------------------------------------------------------------|--
!
!
!     SWASH (Simulating WAves till SHore); a non-hydrostatic wave-flow model
!     Copyright (C) 2010-2011  Delft University of Technology
!
!     This program is free software; you can redistribute it and/or
!     modify it under the terms of the GNU General Public License as
!     published by the Free Software Foundation; either version 2 of
!     the License, or (at your option) any later version.
!
!     This program is distributed in the hope that it will be useful,
!     but WITHOUT ANY WARRANTY; without even the implied warranty of
!     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
!     GNU General Public License for more details.
!
!     A copy of the GNU General Public License is available at
!     http://www.gnu.org/copyleft/gpl.html#SEC3
!     or by writing to the Free Software Foundation, Inc.,
!     59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
!
!
!   Authors
!
!    1.00: Tom Bogaard
!
!   Updates
!
!    1.00, January 2012: New subroutine
!
!   Purpose
!
!   Calculates reynolds stresses with non-linear k-epsilon model
!
!   Method
!
!   Modules used
!
    use ocpcomm4
    use SwashCommdata3
    use SwashFlowdata
    use m_genarr, only: kgrpnt, guu, guv, gvu, gvv, gsqs, gsqsu, gsqsv, work
    use m_parall
!
    implicit none
!
!   Local variables
!
    integer, save                     :: ient = 0    ! number of entries in this subroutine
    integer                           :: m           ! loop counter in x-direction
    integer                           :: n           ! loop counter in y direction
    integer                           :: k           ! loop counter over vertical layers
    integer                           :: kk          ! loop counter over vertical layers (minimal value = 1)
    integer                           :: mu          ! index of point m+1
    integer                           :: md          ! index of point m-1
    integer                           :: nu          ! index of point n+1
    integer                           :: nd          ! index of point n-1
    integer                           :: kd          ! index of layer k-1
    integer                           :: ku          ! index of layer k+1
    integer                           :: kwd         ! index of layer k-1 for w-velocity
    integer                           :: kwu         ! index of layer k+1 for w-velocity
    !
    integer                           :: nm          ! pointer to m   ,n
    integer                           :: num         ! pointer to m   ,n+1
    integer                           :: ndm         ! pointer to m   ,n-1
    integer                           :: nmu         ! pointer to m+1 ,n
    integer                           :: nmd         ! pointer to m-1 ,n
    integer                           :: numd        ! pointer to m-1 ,n+1
    integer                           :: numu        ! pointer to m+1 ,n+1
    integer                           :: ndmd        ! pointer to m+1 ,n-1
    integer                           :: ndmu        ! pointer to m-1 ,n+1
```

```
    !
    real                                :: dudxu       ! velocity gradient in upward point
    real                                :: dudyu       ! velocity gradient in upward point
    real                                :: dudzu       ! velocity gradient in upward point
    real                                :: dvdxu       ! velocity gradient in upward point
    real                                :: dvdyu       ! velocity gradient in upward point
    real                                :: dvdzu       ! velocity gradient in upward point
    real                                :: dwdxu       ! velocity gradient in upward point
    real                                :: dwdyu       ! velocity gradient in upward point
    real                                :: dwdzu       ! velocity gradient in upward point
    real                                :: dudxd       ! velocity gradient in downward point
    real                                :: dudyd       ! velocity gradient in downward point
    real                                :: dudzd       ! velocity gradient in downward point
    real                                :: dvdxd       ! velocity gradient in downward point
    real                                :: dvdyd       ! velocity gradient in downward point
    real                                :: dvdzd       ! velocity gradient in downward point
    real                                :: dwdxd       ! velocity gradient in downward point
    real                                :: dwdyd       ! velocity gradient in downward point
    real                                :: dwdzd       ! velocity gradient in downward point
    !
    real                                :: sxxu        ! stress tensor in upward point
    real                                :: sxyu        ! stress tensor in upward point
    real                                :: sxzu        ! stress tensor in upward point
    real                                :: sxxd        ! stress tensor in downward point
    real                                :: sxyd        ! stress tensor in downward point
    real                                :: sxzd        ! stress tensor in downward point
    !
    real                                :: syxu        ! stress tensor in upward point
    real                                :: syyu        ! stress tensor in upward point
    real                                :: syzu        ! stress tensor in upward point
    real                                :: syxd        ! stress tensor in downward point
    real                                :: syyd        ! stress tensor in downward point
    real                                :: syzd        ! stress tensor in downward point
    !
    real                                :: szxu        ! stress tensor in upward point
    real                                :: szyu        ! stress tensor in upward point
    real                                :: szzu        ! stress tensor in upward point
    real                                :: szxd        ! stress tensor in downward point
    real                                :: szyd        ! stress tensor in downward point
    real                                :: szzd        ! stress tensor in downward point
    !
    real                                :: visceu      ! eddy viscosity in upward point
    real                                :: visced      ! eddy viscosity in downward point
!
!   Structure
!
!   Description of the pseudo code
!
!   Source text
!
    if (ltrace) call strace (ient,'SwashLes')
!
! u-momemtun equation
!
       do k = 1, kmax
          !
          do n = nfu, nl
             !
             do m = mf+1, ml-1
                !
                md    = m - 1
                mu    = m + 1
                nd    = n - 1
                nu    = n + 1
                !
                nm    = kgrpnt(m ,n )
                nmd   = kgrpnt(md,n )
                nmu   = kgrpnt(mu,n )
                ndm   = kgrpnt(m ,nd)
                num   = kgrpnt(m ,nu)
                ndmd  = kgrpnt(md,nd)
```

```
ndmu  = kgrpnt(mu,nd)
numd  = kgrpnt(md,nu)
numu  = kgrpnt(mu,nu)
!
kd  = max(k-1,1   )
ku  = min(k+1,kmax)
kwd  = max(k-1,0   )
kwu  = min(k+1,kmax)
!
! -u'u' / dx
!
! compose parts of reysxxu
!
dudxu = ( u0(nmu,k) - u0(nm,k) ) / ( 0.5 * ( gvv(ndmu) + gvv(nmu) ) )
!
sxxu  = dudxu + dudxu
!
visceu = 0.25 * ( vnu3d(nm,k) + vnu3d(nmu,k) + vnu3d(ndmu,k) + vnu3d(ndm,k) )
!
! compose reysxxu (linear and non-linear part)
!
!
reysxxu(nm,k) = visceu * sxxu
!
! compose parts of reysxxd
!
dudxd = ( u0(nm,k) - u0(nmd,k) ) / ( 0.5 * ( gvv(ndm) + gvv(nm) ) )
!
sxxd  = dudxd + dudxd
!
visced = 0.25 * ( vnu3d(nm,k) + vnu3d(nmd,k) + vnu3d(ndmd,k) + vnu3d(ndm,k) )
!
! compose reysxxd (linear and non-linear part)
!
reysxxd(nm,k) = visced * sxxd
!
! -u'v' / dy
!
! compose parts of reysxyu
!
dudyu = ( u0(num,k) - u0(nm,k) ) / ( 0.5 * ( guu(nm) + guu(num) ) )
dvdxu = ( v0(nmu,k) - v0(nm,k) ) / ( 0.5 * ( gvv(nm) + gvv(nmu) ) )
!
sxyu  = dudyu + dvdxu
!
visceu = vnu3d(nm,k)
!
! compose reysxyu (linear and non-linear part)
!
reysxyu(nm,k) = visceu * sxyu
!
! compose parts of reysxyd
!
dudyd = ( u0(nm,k) - u0(ndm,k) ) / ( 0.5 * ( guu(ndm) + guu(nm) ) )
dvdxd = ( v0(ndmu,k) - v0(ndm,k) ) / ( 0.5 * ( gvv(ndm) + gvv(ndmu) ) )
!
sxyd  = dudyd + dvdxd
!
visced = vnu3d(ndm,k)
!
! compose reysxyd (linear and non-linear part)
!
reysxyd(nm,k) = visced * sxyd
!
! -u'w' / dz
!
! compose parts of reysxzu
!
dudzu = ( u0(nm,kd) - u0(nm,k) ) / ( 0.5 * ( hkum(nm,kd) + hkum(nm,k) ) )
dwdxu = ( w0(nmu,k-1) - w0(nm,k-1) ) / gvu(nm)
!
```

```
                sxzu  = dwdxu !+ dudzu
                !
                visceu = 0.25 * ( vnu3d(nm,k) + vnu3d(ndm,k) + vnu3d(nm,k-1) + vnu3d(ndm,k-1) )
                !
                ! compose reysxzu (linear and non-linear part)
                !
                reysxzu(nm,k) = visceu * sxzu
                !
                ! compose parts of reysxzd
                !
                dudzd = ( u0(nm,k) - u0(nm,ku) ) / ( 0.5 * ( hkum(nm,k) + hkum(nm,ku) ) )
                dwdxd = ( w0(nmu,k) - w0(nm,k) ) / gvu(nm)
                !
                sxzd  = dwdxd !+ dudzd
                !
                visced = 0.25 * ( vnu3d(nm,k+1) + vnu3d(ndm,k+1) + vnu3d(nm,k) + vnu3d(ndm,k) )
                !
                ! compose reysxzd (linear and non-linear part)
                !
                reysxzd(nm,k) = visced * sxzd
                !
             enddo
             !
          enddo
          !
       enddo
       !
!
! v-momemtun equation
!
       do k = 1, kmax
          !
          do n = nf+1, nl-1
             !
             do m = mfu, ml
                !
                md    = m - 1
                mu    = m + 1
                nd    = n - 1
                nu    = n + 1
                !
                nm    = kgrpnt(m ,n )
                nmd   = kgrpnt(md,n )
                nmu   = kgrpnt(mu,n )
                ndm   = kgrpnt(m ,nd)
                num   = kgrpnt(m ,nu)
                ndmd  = kgrpnt(md,nd)
                ndmu  = kgrpnt(mu,nd)
                numd  = kgrpnt(md,nu)
                numu  = kgrpnt(mu,nu)
                !
                kd  = max(k-1,1   )
                ku  = min(k+1,kmax)
                kwd  = max(k-1,0   )
                kwu  = min(k+1,kmax)
                !
                ! -v'u' / dx
                !
                ! compose parts of reysyxu
                !
                dvdxu = ( v0(nmu,k) - v0(nm,k) ) / ( 0.5 * ( gvv(nm) + gvv(nmu) ) )
                dudyu = ( u0(num,k) - u0(nm,k) ) / ( 0.5 * ( guu(num) + guu(nm) ) )
                !
                syxu  = dudyu + dvdxu
                !
                visceu = vnu3d(nm,k)
                !
                ! compose reysyxu (linear and non-linear part)
                !
                reysyxu(nm,k) = visceu * syxu
                !
```

```
      ! compose parts of reysyxd
      !
      dvdxd = ( v0(nm,k) - v0(nmd,k) ) / ( 0.5 * ( gvv(nmd) + gvv(nm) ) )
      dudyd = ( u0(num,k) - u0(nm,k) ) / ( 0.5 * ( guu(num) + guu(nm) ) )
      !
      syxd  = dudyd + dvdxd
      !
      visced = vnu3d(nmd,k)
      !
      ! compose reysyxd (linear and non-linear part)
      !
      reysyxd(nm,k) = visced * syxd
      !
      ! -v'v' / dy
      !
      ! compose parts of reysyyu
      !
      dvdyu = ( v0(num,k) - v0(nm,k) ) / ( 0.5 * ( guu(num) + guu(numd) ) )
      !
      syyu  = dvdyu + dvdyu
      !
      visceu = 0.25 * ( vnu3d(nm,k) + vnu3d(num,k) + vnu3d(numd,k) + vnu3d(nmd,k) )
      !
      ! compose reysyyu (linear and non-linear part)
      !
      reysyyu(nm,k) = visceu * syyu
      !
      ! compose parts of reysyyd
      !
      dvdyd = ( v0(nm,k) - v0(ndm,k) ) / ( 0.5 * ( guu(nm) + guu(nmd) ) )
      !
      syyd  = dvdyd + dvdyd
      !
      visced = 0.25 * ( vnu3d(nm,k) + vnu3d(ndm,k) + vnu3d(ndmd,k) + vnu3d(nmd,k) )
      !
      ! compose reysyyd (linear and non-linear part)
      !
      reysyyd(nm,k) = visced * syyd
      !
      ! -v'w' / dz
      !
      ! compose parts of reysyzu
      !
      dvdzu = ( v0(nm,kd) - v0(nm,k) ) / ( 0.5 * ( hkvm(nm,kd) + hkvm(nm,k) ) )
      dwdyu = ( w0(num,k-1) - w0(nm,k-1) ) / guv(nm)
      !
      syzu  = dwdyu !+ dvdzu
      !
      visceu = 0.25 * ( vnu3d(nm,k) + vnu3d(nmd,k) + vnu3d(nm,k-1) + vnu3d(nmd,k-1) )
      !
      ! compose reysyzu (linear and non-linear part)
      !
      reysyzu(nm,k) = visceu * syzu
      !
      ! compose parts of reysyzd
      !
      dvdzd = ( v0(nm,k) - v0(nm,ku) ) / ( 0.5 * ( hkvm(nm,k) + hkvm(nm,ku) ) )
      dwdyd = ( w0(num,k) - w0(nm,k) ) / guv(nm)
      !
      syzd  = dwdyd !+ dvdzd
      !
      visced = 0.25 * ( vnu3d(nm,k+1) + vnu3d(nmd,k+1) + vnu3d(nm,k) + vnu3d(nmd,k) )
      !
      ! compose reysyzd (linear and non-linear part)
      !
      reysyzd(nm,k) = visced * syzd
      !
   enddo
   !
enddo
!
```

```
       enddo
!
! w-momemtun equation
!
       do k = 0, kmax - 1
          !
          do n = nfu, nl
             !
             do m = mfu, ml
                !
                md    = m - 1
                mu    = m + 1
                nd    = n - 1
                nu    = n + 1
                !
                nm    = kgrpnt(m ,n )
                nmd   = kgrpnt(md,n )
                nmu   = kgrpnt(mu,n )
                ndm   = kgrpnt(m ,nd)
                num   = kgrpnt(m ,nu)
                ndmd  = kgrpnt(md,nd)
                ndmu  = kgrpnt(mu,nd)
                numd  = kgrpnt(md,nu)
                numu  = kgrpnt(mu,nu)
                !
                kd  = max(k-1,1   )
                ku  = min(k+1,kmax)
                kk  = max(k  ,1   )
                kwd  = max(k-1,0   )
                kwu  = min(k+1,kmax)
                !
                ! -w'u' / dx
                !
                ! compose parts of reyszxu
                !
                dwdxu = ( w0(nmu,k) - w0(nm,k) ) / gvu(nm)
                dudzu = ( u0(nm,kk) - u0(nm,ku) ) / ( 0.5 * ( hkum(nm,kk) + hkum(nm,k+1) ) )
                !
                szxu  = dudzu + dwdxu
                !
                visceu = 0.25 * ( vnu3d(nm,k) + vnu3d(ndm,k) + vnu3d(nm,k+1) + vnu3d(ndm,k+1) )
                !
                ! compose reyswxu (linear and non-linear part)
                !
                reyszxu(nm,k) = visceu * szxu
                !
                ! compose parts of reyszxd
                !
                dwdxd = ( w0(nm,k) - w0(nmd,k) ) / gvu(nmd)
                dudzd = ( u0(nmd,kk) - u0(nmd,ku) ) / ( 0.5 * ( hkum(nmd,kk) + hkum(nmd,k+1) ) )
                !
                szxd  = dudzd + dwdxd
                !
                visced = 0.25 * ( vnu3d(nmd,k) + vnu3d(ndmd,k) + vnu3d(nmd,k+1) + vnu3d(ndmd,k+1) )
                !
                ! compose reyszxd (linear and non-linear part)
                !
                reyszxd(nm,k) = visced * szxd
                !
                ! -w'v' / dy
                !
                ! compose parts of reyswyu
                !
                dwdyu = ( w0(num,k) - w0(nm,k) ) / guv(nm)
                dvdzu = ( v0(nm,kk) - v0(nm,ku) ) / ( 0.5 * ( hkvm(nm,kk) + hkvm(nm,k+1) ) )
                !
                szyu  = dwdyu + dvdzu
                !
                visceu = 0.25 * ( vnu3d(nm,k) + vnu3d(nmd,k) + vnu3d(nm,k+1) + vnu3d(nmd,k+1) )
                !
                ! compose reyswyu (linear and non-linear part)
```

```
                !
                reyszyu(nm,k) = visceu * szyu
                !
                ! compose parts of reyswyd
                !
                dwdyd = ( w0(nm,k) - w0(ndm,k) ) / guv(ndm)
                dvdzd = ( v0(ndm,kk) - v0(ndm,ku) ) / ( 0.5 * ( hkvm(ndm,kk) + hkvm(ndm,k+1) ) )
                !
                szyd  = dwdyd + dvdzd
                !
                visced = 0.25 * ( vnu3d(ndm,k) + vnu3d(ndmd,k) + vnu3d(ndm,k+1) + vnu3d(ndmd,k+1) )
                !
                ! compose reyswyd (linear and non-linear part)
                !
                reyszyd(nm,k) = visced * szyd
                !
                ! -w'w' / dz
                !
                ! compose parts of reyszzu
                !
                dwdzu = ( w0(nm,kwd) - w0(nm,k) ) / hks(nm,kk)
                !
                szzu  = 0. !dwdzu + dwdzu
                !
                visceu = 0.25 * ( vnu3d(nm,k) + vnu3d(nmd,k) + vnu3d(ndm,k) + vnu3d(ndmd,k) )
                !
                ! compose reyszzu (linear and non-linear part)
                !
                reyszzu(nm,k) = visceu * szzu
                !
                ! compose parts of reyszzd
                !
                dwdzd = ( w0(nm,k) - w0(nm,ku) ) / hks(nm,k+1)
                !
                szzd  = 0. !dwdzd + dwdzd
                !
                visced = 0.25 * ( vnu3d(nm,k+1) + vnu3d(nmd,k+1) + vnu3d(ndm,k+1) + vnu3d(ndmd,k+1) )
                !
                ! compose reyszzd (linear and non-linear part)
                !
                reyszzd(nm,k) = visced * szzd
                !
             enddo
             !
          enddo
          !
       enddo
end subroutine SwashLes
```

The components of the subgrid stress tensor are explicitly added to the momentum equations in the module: SwashExpLay2DHflow.ftn90. Only the changed part of the module is printed.

```
 subroutine SwashExpLay2DHflow
    !
    if ( iturb == 1 ) then
       !
       do k = 1, kmax
          !
          do n = nfu, nl
             !
             do m = mf + 1, ml - 1
                !
                nm   = kgrpnt(m ,n )
                !
                if ( uwetp(nm) ) then
                   !
                   lmask = .true.
                   !
                   ! next, compute the divergence of the Subgrid stresses
```

```
               !
               rhsu(nm,k) = rhsu(nm,k) + ( reysxxu(nm,k) - reysxxd(nm,k) ) / gvu(nm) + &
                                         ( reysxyu(nm,k) - reysxyd(nm,k) ) / guu(nm) + &
                                         ( reysxzu(nm,k) - reysxzd(nm,k) ) / hkum(nm,k)
               !
            endif
            !
         enddo
         !
      enddo
      !
   enddo
   !
endif
!
if ( iturb == 1 ) then
   !
   do k = 1, kmax
      !
      do m = mfu, ml
         !
         do n = nfu, nl - 1
            !
            nm   = kgrpnt(m ,n )
            !
            if ( vwetp(nm) ) then
               !
               ! next, compute the divergence of the Subgrid stresses
               !
               rhsu(nm,k) = rhsu(nm,k) + ( reysyxu(nm,k) - reysyxd(nm,k) ) / gvv(nm) + &
                                         ( reysyyu(nm,k) - reysyyd(nm,k) ) / guv(nm) + &
                                         ( reysyzu(nm,k) - reysyzd(nm,k) ) / hkvm(nm,k)
               !
            endif
            !
         enddo
         !
      enddo
      !
   enddo
   !
endif
!
if ( iturb == 1 ) then
   !
   do k = 0, kmax-1
      !
      do m = mfu, ml
         !
         do n = nfu, nl
            !
            nd = n - 1
            md = m - 1
            !
            nm   = kgrpnt(m ,n )
            ndm  = kgrpnt(m ,nd)
            nmd  = kgrpnt(md,n )
            !
            kk  = max(k  ,1   )
            !
            if ( vwetp(nm) ) then
               !
               ! next, compute the divergence of the Subgrid stresses
               !
               rhsw(nm,k) = rhsw(nm,k) + ( reyszxu(nm,k) - reyszxd(nm,k) ) / &
                                         ( 0.5 * ( gvv(nm) + gvv(nmd) ) ) + &
                                         ( reyszyu(nm,k) - reyszyd(nm,k) ) / &
                                         ( 0.5 * ( guu(nm) + guu(ndm) ) ) + &
                                         ( reyszzu(nm,k) - reyszzd(nm,k) ) / &
                                         ( 0.5 * ( hks(nm,kk) + hks(nm,k+1) ) )
               !
```

```
            endif
            !
         enddo
         !
      enddo
      !
   enddo
   !
endif
```

The Schumann wall model is used to account for bottom friction. SwashBotFrict is adapted to compute the instantaneous bottom shear stress. Only the adapted part of the module is printed. “irough == 5” is a new value of this indicator, a value of 5 implies that the Schumann model is used.

```
else if ( irough == 5 ) then
   !
   ! Nikuradse roughness height
   !
   if ( varfr ) then
      !
      do n = nfu, nl
         do m = mf, ml
            !
            nm = kgrpnt(m,n)
            !
            zs = 0.5 * ( zks(nm,kmax-1) - zks(nm,kmax) )
            !
            if ( wetu(nm) == 1 .and. zs > 0. ) then
               !
               if ( timco < 0.01 ) then
                  !
                  facu(nm)  = u0(nm,kmax)
                  if ( facu(nm) < 0.005 ) facu(nm) = 0.005
                  !
               else
                  !
                  facu(nm)  = 0.999 * facu(nm) + 0.001 * u0(nm,kmax)
                  if ( facu(nm) < 0.005 ) facu(nm) = 0.005
                  !
               endif
               !
               if ( fricf(nm,2) /= 0. ) then
                  !
                  logfrc(nm,1) = facu(nm) * ( ( vonkar / log( ( 30. * zs ) / fricf(nm,2) ) ) **2. )
                  logfrc(nm,2) = ( 1./vonkar ) * log( ( 30. * zs ) / fricf(nm,2) )
                  !
               else
                  !
                  r = 0.1 * facu(nm)
                  !
                  if ( r < 1. ) then
                     !
                     nit = 0
                     !
                     ! initial value for s
                     !
                     s    = r
                     sold = 0.
                     !
                     ! Newton-Raphson iteration
                     !
                     do
                        if ( abs(sold-s) < (eps*s) ) exit
                        !
                        nit           = nit + 1
                        sold          = s
                        s    = sold - ( ( facu(nm) / sold ) - 2.44 * log( ( sold * zs ) &
                               / kinvis ) - 5.29 ) / ( - facu(nm) * sold ** -2. - 2.44 / sold )
```

```
                    !
                    if ( .not. nit < maxnit ) then
                       call msgerr (1, 'no convergence in bottom friction computation')
                       s = r
                       exit
                    endif
                    !
                 enddo
                 !
              else
                 !
                 s = r
                 !
              endif
              !
              logfrc(nm,1)  = ( s ** 2. ) / facu(nm)
              logfrc(nm,2)  = facu(nm) / s
              !
           endif
           !
        endif
        !
     enddo
  enddo
```

The Schumann wall model can also be used at side walls. The shear stresses is directly computed in SwashExpLay2DHflow. The adapted part of the subroutine is printed.

```
do k = 1, kmax
  do m = mfu, ml
     !
     nfum = kgrpnt(m,nfu)
     nlm  = kgrpnt(m,nl)
     !
     if ( timco < 0.01 ) then
        !
        facuf(nfum,k)     = u0(nfum,k)
        if ( facuf(nfum,k) < 0.005 ) facuf(nfum,k) = 0.005
        !
        facul(nlm,k)      = u0(nlm,k )
        if ( facul(nlm,k ) < 0.005 ) facul(nlm ,k) = 0.005
        !
     else
        !
        facuf(nfum,k)     = 0.999 * facuf(nfum,k) + 0.001 * u0(nfum,k)
        if ( facuf(nfum,k) < 0.005 ) facuf(nfum,k) = 0.005
        !
        facul(nlm,k)      = 0.999 * facul(nlm,k ) + 0.001 * u0(nlm,k)
        if ( facul(nlm,k ) < 0.005 ) facul(nlm,k ) = 0.005
        !
     endif
     !
     if ( fricf(nfum, 2) > 0 ) then
        !
        amatu(nfum,k,1) = amatu(nfum,k,1) + ( facuf(nfum,k) * ( ( vonkar / log( ( 30. * 0.5 &
                          * guu(nfum) ) / fricf(nfum,2) ) ) **2. ) ) / ( guu(nfum) )
     else
        !
        rf = 0.1 * facuf(nfum)
        !
        if ( rf < 1. ) then
           !
           ! initial value for s
           !
           sf    = rf
           soldf = 0.
           !
           ! Newton-Raphson iteration
           !
```

```
            do
               if ( abs(soldf-sf) < (0.01*sf) ) exit
               !
               soldf          = sf
               sf    = soldf - ( ( facuf(nfum) / soldf ) - 2.44 * log( ( soldf * guu(nfum) ) &
                        / kinvis ) - 5.29 ) / ( - facuf(nfum) * soldf ** -2. - 2.44 / soldf )
               !
            enddo
            !
         else
            !
            sf = rf
            !
         endif
         !
         amatu(nfum,k,1) = amatu(nfum,k,1) + ( sf ** 2. / facuf(nfum) ) / ( guu(nfum) )
         !
      endif
      !
      if ( fricf(nlm, 2) > 0 ) then
         !
         amatu(nlm ,k,1) = amatu(nlm ,k,1) + ( facul(nlm,k ) * ( ( vonkar / log( ( 30. &
                           * 0.5 * guu(nlm ) ) / fricf(nlm,2) ) ) **2. ) ) / ( guu(nlm ) )
         !
      else
         !
         rl = 0.1 * facul(nlm)
         !
         if ( rl < 1. ) then
            !
            ! initial value for s
            !
            sl    = rl
            soldl = 0.
            !
            ! Newton-Raphson iteration
            !
            do
               if ( abs(soldl-sl) < (0.01*sl) ) exit
               !
               soldl          = sl
               sl    = soldl - ( ( facul(nlm) / soldl ) - 2.44 * log( ( soldl * guu(nlm) ) &
                        / kinvis ) - 5.29 ) / ( - facul(nlm) * soldl ** -2. - 2.44 / soldl )
               !
            enddo
            !
         else
            !
            sl = rl
            !
         endif
         !
         amatu(nlm ,k,1) = amatu(nlm ,k,1) + ( sl ** 2. / facul(nlm) ) / ( guu(nlm ) )
         !
      endif
      !
   enddo
enddo
```

In addition the normal velocity gradients at the virtual cells are set to zero.

```
do m = mf, ml
   !
   nfm  = kgrpnt(m,nf )
   nfum = kgrpnt(m,nfu)
   nlm  = kgrpnt(m,nl )
   nlum = kgrpnt(m,nlu)
   !
   if ( ibb(m) == 1 .and. LMYF ) then
      !
```

```
      ! set to zero at closed boundary
      !
      u1(nfm,:) = u1(nfum,:)!0.
      !
   else if ( ibb(m) > 1 .and. LMYF ) then
      !
      ! zero normal gradient condition at open boundaries
      !
      u1(nfm,:) = u1(nfum,:)
      !
   endif
   !
   if ( ibt(m) == 1 .and. LMYL ) then
      !
      ! set to zero at closed boundary
      !
      u1(nlum,:) = u1(nlm,:) !0.
      !
   else if ( ibt(m) > 1 .and. LMYL ) then
      !
      ! zero normal gradient condition at open boundaries
      !
      u1(nlum,:) = u1(nlm,:)
      !
   endif
   !
enddo
```

A logartimic velocity profile with superimposed random white noise fluctuations is imposed at the inflow boundary. The boundary conditions is implemented at the top of SwashExpLat2DHflow.

```
!
! fluctuating velocity boudary condition ( based on u* )
!
if ( LMXF ) then
!
do k = 1, kmax
   !
   do n = nfu, nl
      !
      indx = kgrpnt(mf,n)
      nfmf = kgrpnt(mf,nf)
      nlmf = kgrpnt(mf,nl)
      !
      v1(nfmf,:) = 0.
      v1(nlmf,:) = 0.
      !
      if ( timco < 0.01) then
         !
         fac3 = 0
         !
                         u1(indx,k)  = u0(indx,k)
         if ( n < nl )   v1(indx,k)  = v0(indx,k)
         if ( k < kmax ) w1(indx,k)  = w0(indx,k)
         !
      endif
      !
      if ( ( timco - fac3 ) > 0.5 ) then
         !
         zs = 0.5 * ( zks(indx,kmax-1) - zks(indx,kmax) ) * ( kmax - k + 1 ) &
              * 2 - 0.5 * ( zks(indx,kmax-1) - zks(indx,kmax) )
         !
         u0(indx,k) = ( ( 2.44 * log( ( zs * 0.0055 ) / kinvis ) ) + 5.29 )  * 0.0055
         !
         call RANDOM_NUMBER(fac)
         call RANDOM_NUMBER(fac1)
         call RANDOM_NUMBER(fac2)
         !
         if ( fac < 0.5) then
```

```
            !
            u0(indx,k) = u0(indx,k) + ( 0.1 * fac ) * u0(indx,k)
            !
         else
            !
            u0(indx,k) = u0(indx,k)  - ( 0.1 * ( 1. - fac ) ) * u0(indx,k)
            !
         endif
         !
         if ( fac1 < 0.5 .and. n < nl ) then
            !
            v0(indx,k) =  0 + ( 0.1 * fac1 ) * u0(indx,k)
            !
         elseif ( n < nl ) then
            !
            v0(indx,k) =  0 - ( 0.1 * ( 1. - fac1 ) ) * u0(indx,k)
            !
         endif
         !
         if ( fac2 < 0.5 .and. k < kmax ) then
            !
            w0(indx,k) = 0 + ( 0.1 * fac2 ) * u0(indx,k)
            w0(indx,0) = 0 + ( 0.1 * fac2 ) * u0(indx,k)
            !
         elseif ( k < kmax ) then
            !
            w0(indx,k) = 0 - ( 0.1 * ( 1. - fac2 ) ) * u0(indx,k)
            w0(indx,0) = 0 - ( 0.1 * ( 1. - fac2 ) ) * u0(indx,k)
            !
         endif
            !
         if ( k == kmax .and. indx == kgrpnt(mf,nl) ) fac3 = timco
         !
      else
         !
                         u0(indx,k) = u0(indx,k)
         if ( n < nl )   v0(indx,k) = v0(indx,k)
         if ( k < kmax ) w0(indx,k) = w0(indx,k)
                         w0(indx,0) = w0(indx,0)
         !
      endif
      !
                      u1(indx,k) = u0(indx,k)
      if ( n < nl )   v1(indx,k) = v0(indx,k)
      if ( k < kmax ) w1(indx,k) = w0(indx,k)
                      w1(indx,0) = w0(indx,0)
      !
   enddo
   !
enddo
!
endif
```