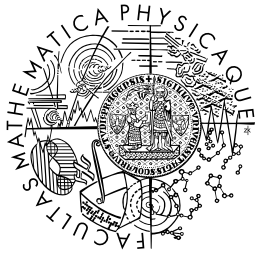

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

David Jiříček

Application of spectral element method in computations of incompressible turbulent flow

Mathematical Institute of Charles University

Supervisor of the master thesis:	RNDr. Jan Pech, Ph.D.
Study programme:	Physics
Study branch:	Mathematical Modelling in Physics

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Název: Použití metody spektrálních elementů ve výpočtech nestlačitelného turbulentního proudění

Autor: David Jiříček

Katedra: Matematický ústav UK

Vedoucí diplomové práce: RNDr. Jan Pech, Ph.D., Ústav termomechaniky Akademie věd ČR

Abstrakt: Práce se zabývá modelováním nestlačitelného turbulentního proudění pomocí metody spektrálních elementů (SEM, spectral element method). Jsou zde představeny principy této metody, kterou můžeme chápat jako kombinaci spektrální metody a metody konečných prvků. Turbulentní proudění je popsáno dvou rovnicovým k-ω modelem turbulence, konkrétně Kolmogorovovým modelem a Wilcoxovým modelem. V práci jsou objasněny důvody této volby. Oba modely ve verzi pro 2D výpočty jsou implementovány do Nektar++, již existujícího C++ frameworku pro řešení rovnic metodou spektrálních elementů. Byla nalezena analytická řešení Navier-Stokesových rovnic s proměnnou viskozitou. Tato řešení byla následně použita pro otestování implementace. Simulovali jsme turbulentní proudění v rovinném kanálu a porovnali jsme výsledky s přímou numerickou simulací.

Klíčová slova: metoda spektrálních elementů, modely turbulence, nestlačitelné proudění

Title: Application of spectral element method in computations of incompressible turbulent flow

Author: David Jiříček

Department: Mathematical Institute of Charles University

Supervisor: RNDr. Jan Pech, Ph.D., Institute of Thermomechanics of the CAS

Abstract: In the thesis we study incompressible turbulent flow using spectral element method (SEM). We present fundamentals of SEM which can be seen as a combination of spectral method and finite element method. Turbulent flow is described with the help of two-equation k-ω turbulence models. We give reasons for the choice of Kolmogorov's and Wilcox model and implement them in 2D into an existing SEM C++ framework, Nektar++. Analytical solutions of Navier-Stokes equation were found and used as test cases for the implementation of the models. We simulated turbulent flow in channel and compared the results with direct numerical simulation.

Keywords: spectral element method, turbulence models, incompressible flow

Dedication: I dedicate this work to Kateřina and my parents. Thank you for your love and support.

I would also like to thank my supervisor Jan Pech, for his guidance and important advice, prof. Příhoda, who show me some aspects of turbulent models, and prof. Málek, for providing me with useful information.

Contents

Introduction

Spectral element method (SEM) emerged as a combination of two approaches to solving partial differential equations - global spectral method (see monograph by [Gottlieb, 1977]) and finite element method (see [Oden, 2011]). Global spectral methods offer high order precision within limited geometries, whereas finite elements always brings geometrical flexibility. Original idea of combining them and getting the best of both worlds comes from [Patera, 1984], who came up with multi-domain spectral methods, and from [Babuska et al., 1981], who used finite elements with polynomials of high degree p. The basic idea stays the same - divide the computational domain into multiple elements (multiple domains), but within these elements, keep a high degree polynomial (or Fourier) basis.

In the recent years, SEM has become more popular. In engineering, there is higher need for precise time-dependent solution over long time periods, for example in the field of electromagnetics in aerospace design [Karniadakis, 2005]. Computational framework, including open-source projects, is more advanced and accessible than ever before. Researchers can therefore rely on existing code, instead of developing everything from scratch.

Also, historically, spectral methods were limited to simple domains, nowadays they are used even for complex geometries [Canuto et al., 2007].

In fluid dynamics, spectral elements are often the method of choice for performing direct numerical simulation (DNS) of turbulent flow. Turbulent flow is characterized by high Reynolds number and it is notoriously difficult to simulate. Precision of SEM helps to resolve all scales of turbulent motion.

When simulating turbulent flow with finite elements of low polynomial order, one usually has to resort to *turbulent models*. Turbulent models do not simulate small scale velocity fluctuations, but approximate their influence on the properties of main, averaged flow.

The aim of the thesis is to simulate turbulence with SEM in the novel way. Instead of performing DNS, we will employ turbulence model. These models are often quite complex and high sensitivity of SEM can give us greater insight into what is really happening with the solution. It brings us closer to the mathematical analysis of the model, which can be difficult or even impossible. Nevertheless, one of the starting points for our model consideration was in work of [Bulíček and Málek, 2016], where long-time and large-data existence of a suitable weak solution to Kolmogorov's two-equation model was established.

We will take advantage of existing SEM computation framework, Nektar++, and implement existing turbulence model as a new type of solver there. As a part of the open-source framework, it is expected to be available also to other researches.

Structure of the thesis is following. In chapter 1, we introduce the turbulence model. Our aim was to choose model with some mathematical background.

Basic building blocks of spectral element method are described in chapter 2.

Chapter 3 gives brief summary of *general linear methods* - abstract framework for time discretization of ordinary and partial differential equations. This framework forms the basis of time-stepping in Nektar++, so it is necessary to understand it, before implementation of the model.

Chapter 4 contains details of model implementation. Tests of the implementation and numerical experiments are in the chapter 5.

1. Model description

In the current version of Nektar++, the only option to simulate turbulent flow is to use unmodified Navier-Stokes equations and perform DNS - direct numerical simulation.

In many cases, it is computationally expensive and its advantageous to use turbulence model. Generally speaking, turbulence models modify Navier-Stokes equations to count for turbulence effects without resolving all scales of the motion. Turbulence modelling is a broad subject, for introduction we refer reader to [Lars, 2017], [Wilcox, 2006] or [Příhoda, 2007] (in Czech language).

Because of its precision, spectral element method can help us to get very close to analytical solution of PDE system. The method is also more sensitive to regularity of solution, domain and proper initial and boundary data. Thus, it is natural to examine turbulence model, that was rigorously mathematically analyzed.

Kolmogorov's two-equation model, see [Kolmogorov, 1941] [1] , was analyzed in [Bulíček and Málek, 2016], who described it by following equations

$$\frac{\partial v_i}{\partial x_i} = 0 \tag{1.1}$$

$$\frac{\partial v_i}{\partial t} + v_j \frac{\partial v_i}{\partial x_j} = -\frac{\partial p}{\partial x_i} + 2\nu_0 \frac{\partial}{\partial x_j}\left(\frac{b}{\omega} S_{ij}\right) \tag{1.2}$$

$$\frac{\partial b}{\partial t} + v_j \frac{\partial b}{\partial x_j} = -b\omega + \kappa_4 \frac{b}{\omega} S_{ij} S_{ij} + \kappa_3 \frac{\partial}{\partial x_j}\left[\frac{b}{\omega}\left(\frac{\partial b}{\partial x_j}\right)\right] \tag{1.3}$$

$$\frac{\partial \omega}{\partial t} + v_j \frac{\partial \omega}{\partial x_j} = -\kappa_2 \omega^2 + \kappa_1 \frac{\partial}{\partial x_j}\left[\frac{b}{\omega}\left(\frac{\partial \omega}{\partial x_j}\right)\right] \tag{1.4}$$

where S_{ij} stands for symmetric velocity gradient, b is $\frac{3}{2}$ of turbulent kinetic energy k (measure of turbulence intensity or intensity of velocity fluctuation), ω is specific dissipation rate and ν_0, κ_1, κ_2, κ_3, κ_4 are positive constants.

Analysis of [Bulíček and Málek, 2016] established long-term and large data existence of suitable weak solution. Model is *complete*, it can be used with no prior knowledge of turbulent structure of the flow. When we compare it with other two-equation models, Kolmogorov's model is also remarkably simple and elegant.

From the implementation point of view, however, Kolmogorov's model has some shortcomings.

Firstly, Kolmogorov did not specify the value of constants.

Secondly, the diffusion term in the velocity equation linearly depends on b and we miss molecular diffusion term. That means the model is useful only for flows with high Reynolds number Re, with large turbulence intensity. We cannot use model near the wall, where b is negligible and viscous effects dominate. It is necessary to use wall function - their implementation would be difficult.

Lastly, Kolmogorov's model is obsolete and a lot of improvement has been made since its invention.

[1] English translation can be found in [Tikhomirov, 1991].

Fortunately, Kolmogorov's model is very similar to modern k-ω models. We decided to implement one of the most popular k-ω models, proposed by Wilcox in his book [Wilcox, 2006].

$$\frac{\partial v_i}{\partial x_i} = 0 \tag{1.5}$$

$$\frac{\partial v_i}{\partial t} + v_j \frac{\partial v_i}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left[2\left(\nu + \nu_T\right) S_{ij}\right] \tag{1.6}$$

$$\frac{\partial k}{\partial t} + v_j \frac{\partial k}{\partial x_j} = \tau_{ij}\frac{\partial v_i}{\partial x_j} - \beta^\star k\omega + \frac{\partial}{\partial x_j}\left[\left(\nu + \sigma^\star \frac{k}{\omega}\right)\left(\frac{\partial k}{\partial x_j}\right)\right] \tag{1.7}$$

$$\frac{\partial \omega}{\partial t} + v_j \frac{\partial \omega}{\partial x_j} = \alpha\frac{\omega}{k}\tau_{ij}\frac{\partial v_i}{\partial x_j} - \beta\omega^2 + \frac{\sigma_d}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j} + \frac{\partial}{\partial x_j}\left[\left(\nu + \sigma\frac{k}{\omega}\right)\left(\frac{\partial \omega}{\partial x_j}\right)\right] \tag{1.8}$$

Wilcox model overcomes all shortcomings we mentioned. All constants are specified, see section 1.1. It includes molecular diffusion (term with constant non-turbulent viscosity ν), so it can be integrated through the viscous sub-layer up to wall. It is quite new and thanks to some additional terms, it gives superior results.

Although Wilcox model is more sophisticated, it is strongly related to Kolmogorov's model. All it takes to transform it is to change values of model constants. That way, we can use mathematic background, provided by analysis of Kolmogorov's model.

1.1 Overview of Wilcox k-ω model

Our model is a version of k-ω model, suitable for incompressible flows. It is taken exactly as it stands from [Wilcox, 2006] [2] The only modification being additional constant $\sigma_\nu = 1$, that appears in the definition of ν_T.

$$\nu_T = \sigma_\nu \frac{k}{\tilde{\omega}} \tag{1.9}$$

This constant does not have any effect on the model, but we have added it to simplify reduction of Wilcox model to Kolmogorov one.

Wilcox model uses *eddy viscosity* ν_T to compute turbulence effects. Eddy viscosity is determined by two variables, kinetic energy of turbulent fluctuations k and rate of dissipation of energy ω. Both variables satisfy their own evolution equation, so the model belongs to the class of two-equation models.

Together with Reynolds averaged Navier-Stokes equations it can be summarized as follows:

- Unknown variables (fields)

$$\begin{aligned} &v_i(x_j,t) \ldots \textit{velocity}\\ &p(x_j,t) \ldots \textit{pressure}\\ &k(x_j,t) \ldots \textit{turbulence kinetic energy}\\ &\omega(x_j,t) \ldots \textit{specific dissipation rate} \end{aligned}$$

[2] In the book, the cited model is referred to as the "Wilcox (2006) k-ω model".

- Incompressibility condition

$$\frac{\partial v_i}{\partial x_i} = 0 \tag{1.10}$$

- Reynolds averaged Navier-Stokes equations with eddy viscosity ν_T

$$\frac{\partial v_i}{\partial t} + v_j \frac{\partial v_i}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left[2\left(\nu + \nu_T\right) S_{ij}\right] \tag{1.11}$$

- Evolution equations for turbulent variables k and ω

$$\frac{\partial k}{\partial t} + v_j \frac{\partial k}{\partial x_j} = \tau_{ij}\frac{\partial v_i}{\partial x_j} - \beta^\star k\omega + \frac{\partial}{\partial x_j}\left[\left(\nu + \sigma^\star \frac{k}{\omega}\right)\left(\frac{\partial k}{\partial x_j}\right)\right] \tag{1.12}$$

$$\frac{\partial \omega}{\partial t} + v_j \frac{\partial \omega}{\partial x_j} = \alpha\frac{\omega}{k}\tau_{ij}\frac{\partial v_i}{\partial x_j} - \beta\omega^2 + \frac{\sigma_d}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j} + \frac{\partial}{\partial x_j}\left[\left(\nu + \sigma \frac{k}{\omega}\right)\left(\frac{\partial \omega}{\partial x_j}\right)\right] \tag{1.13}$$

- Definition of eddy viscosity

$$\nu_T = \sigma_\nu \frac{k}{\tilde{\omega}}, \quad \tilde{\omega} = \max\left\{\omega, C_{lim}\sqrt{\frac{2S_{ij}S_{ij}}{\beta^\star}}\right\}, \quad C_{lim} = \frac{7}{8}, \quad \sigma_\nu = 1 \tag{1.14}$$

- Numerical value of model constants

$$\alpha = \frac{13}{25}, \quad \beta = \beta_0 f_\beta, \quad \beta^\star = \frac{9}{100}, \quad \sigma = \frac{1}{2}, \quad \sigma^\star = \frac{3}{5}, \quad \sigma_{do} = \frac{1}{8},$$

$$\sigma_d = \begin{cases} 0, & \frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j} \le 0 \\ \sigma^{do}, & \frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j} > 0 \end{cases}$$

$$\beta_0 = 0.0708, \qquad f_\beta = \frac{1 + 85\chi_\omega}{1 + 100\chi_\omega}, \qquad \chi_\omega = \left\|\frac{\Omega_{ij}\Omega_{jk}S_{ki}}{\left(\beta^\star\omega\right)^3}\right\|$$

- Definition of Reynolds stress tensor τ_{ij}, symmetric velocity gradient S_{ij} and skew-symmetric matrix Ω_{ij}.

$$\Omega_{ij} = \frac{1}{2}\left(\frac{\partial v_i}{\partial x_j} - \frac{\partial v_j}{\partial x_i}\right), \quad S_{ij} = \frac{1}{2}\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right), \quad \tau_{ij} = 2\nu_T S_{ij} - \frac{2}{3}k\delta_{ij}$$

- Formulas for computing dissipation ϵ and length scale l.

$$\epsilon = \beta^\star \omega k, \qquad l = \frac{\sqrt{k}}{\omega}$$

Model is defined on domain $\Omega \times [0, T]$. Fields $\boldsymbol{v}$, k and ω satisfy initial and boundary conditions, prescribed by user.

1.2 Reduction to Kolmogorov model

In contrast to Kolmogorov's model ((1.1) - (1.4)), Wilcox model ((1.5) - (1.8)) uses different variable for turbulent energy, k instead of b. Relation between them is

$$k = \frac{2}{3}b \tag{1.15}$$

It also contain some new terms:

Molecular diffusion (= terms containing constant non-turbulent viscosity ν):

$$2\nu\frac{\partial}{\partial x_j}S_{ij}, \qquad \nu\frac{\partial k}{\partial x_j \partial x_j}, \qquad \nu\frac{\partial \omega}{\partial x_j \partial x_j},$$

present in all three evolution equations.

Production of ω (term with α) and **cross-diffusion** (term with σ_d)

$$\alpha\frac{\omega}{k}\tau_{ij}\frac{\partial v_i}{\partial x_j}, \qquad \frac{\sigma_d}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j},$$

appearing in evolution equation for ω (1.8).

Other features introduced in Wilcox model is **stress-limiter**, i.e. usage of $\tilde{\omega}$ instead of plain ω in calculation of ν_T.

$$\nu_T = \sigma_\nu\frac{k}{\tilde{\omega}}, \qquad \tilde{\omega} = \max\left\{\omega, C_{lim}\sqrt{\frac{2S_{ij}S_{ij}}{\beta^\star}}\right\}$$

and also modification accounting for **vortex stretching** (proposed by [Pope, 1978]), in the form of function f_β.

We can "turn-off" all these features, if we change values of related constants ν, α, σ_{do}, C_{lim} to zero and f_β to one.

Before proceeding further, we derive auxiliary formula

$$S_{ij}S_{ij} = \frac{1}{2}S_{ij}\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right) = \frac{1}{2}S_{ij}\frac{\partial v_i}{\partial x_j} + \frac{1}{2}S_{ji}\frac{\partial v_j}{\partial x_i} = S_{ij}\frac{\partial v_i}{\partial x_j} \tag{1.16}$$

from definition of S_{ij} and its symmetry.

For production term in (1.7) we have

$$\tau_{ij}\frac{\partial v_i}{\partial x_j} = 2\nu_T S_{ij}\frac{\partial v_i}{\partial x_j} - \frac{2}{3}k\delta_{ij}\frac{\partial v_i}{\partial x_j} = 2\nu_T S_{ij}S_{ij} \tag{1.17}$$

where we used definition of τ_{ij}, incompressibility condition and formula (1.16).

To sum up, taking Wilcox model, setting

$$k = \frac{2}{3}b, \qquad \nu = 0, \qquad \alpha = 0, \qquad \sigma_{do} = 0, \qquad C_{lim} = 0, \qquad f_\beta = 1$$

and rewriting k production term as shown in (1.17) yields

$$\frac{\partial v_i}{\partial x_i} = 0 \tag{1.18}$$

$$\frac{\partial v_i}{\partial t} + v_j \frac{\partial v_i}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{4}{3}\sigma_\nu \frac{\partial}{\partial x_j}\left[\frac{b}{\omega} S_{ij}\right] \tag{1.19}$$

$$\frac{\partial b}{\partial t} + v_j \frac{\partial b}{\partial x_j} = -\beta^\star b\omega + 2\sigma_\nu \frac{b}{\omega} S_{ij} S_{ij} + \frac{2}{3}\sigma^\star \frac{\partial}{\partial x_j}\left[\frac{b}{\omega}\left(\frac{\partial b}{\partial x_j}\right)\right] \tag{1.20}$$

$$\frac{\partial \omega}{\partial t} + v_j \frac{\partial \omega}{\partial x_j} = -\beta\omega^2 + \sigma \frac{2}{3}\frac{\partial}{\partial x_j}\left[\frac{k}{\omega}\left(\frac{\partial \omega}{\partial x_j}\right)\right] \tag{1.21}$$

This system strongly reminds of Kolmogorov's model, it remains only to identify constants.

$$\nu_0 = \frac{2}{3}\sigma_\nu, \qquad \kappa_1 = \frac{2}{3}\sigma, \qquad \kappa_2 = \beta, \qquad \kappa_3 = \frac{2}{3}\sigma^\star, \qquad \kappa_4 = 2\sigma_\nu \qquad \beta^\star = 1$$

All constants are positive and satisfies requirements of analysis. There is one difference, constants ν_0 and κ_4 are no longer independent.

2. Spatial discretization using SEM

Spectral element method (SEM) is a numerical method used for solving partial differential equations. It was developed by community around global spectral methods, but it coincides generally with hp-finite elements.

In contrast to low order finite element method (FEM), the spectral-hp elements enables us to reach much higher precision.

In this chapter, we explain its fundamental concepts, advantages, disadvantages and also mention similarities and differences to the related methods.

Our introduction is restricted only to classic Galerkin (CG) form of SEM, with conforming elements, because we chose it for the implementation of the turbulent model. Spectral elements, however, can be successfully implemented even for discontinous Galerkin (DG).

2.1 General formulation

Let $\boldsymbol{D}$ be a (not necessarily linear) differential operator on the function space V. Functions from V are defined in the spatial domain $\Omega \subset \mathbb{R}^d$, where $d \in \{1, 2, 3\}$.

Let us suppose that there exists $u \in V$ that solves equation

$$\boldsymbol{D}(u) = 0 \tag{2.1}$$

supplemented with appropriate boundary conditions. Our goal is to find the approximation u_δ of exact solution u.

To find the best approximation using SEM, we start with continuous Galerkin discretization. We define finite-dimensional space $V_\delta \subset V$, with dimension N_{gm} and basis Φ_k of C^0 continuous functions. All Φ_k satisfy Dirichlet boundary condition for u.

Then we derive the variational (weak) formulation of equation (2.1) and set both *test* and *trial* space equal to V_δ. Boundary condition are dealt with by lifting the solution. The (homogeneous) solution u_δ and test function v_δ have similar form

$$u_\delta = \sum_{k=1}^{N_{gm}} \hat{u}_k \Phi_k(x) \qquad \Phi_k \in V_\delta \tag{2.2}$$

$$v_\delta = \sum_{k=1}^{N_{gm}} \hat{v}_k \Phi_k(x) \qquad \Phi_k \in V_\delta \tag{2.3}$$

where $\hat{u}_k$, $\hat{v}_k$ are coefficients and N_{gm} is the number of degrees of freedom.

Inserting (2.2) and (2.3) into variational formulation of (2.1) produces N_{gm} (non-linear) equations for N_{gm} unknowns. All these procedures are conducted in a standard way, reader can find example for simple Galerkin projection in section 2.9, or for non-linear operator, in [Rannacher, 1999]. The difference between SEM and other methods stems from the choice of trial space V_δ.

Trial space V_δ is defined by its basis Φ_k. In SEM, Φ_k are called *global expansion modes*. To construct them, let us divide the domain Ω into smaller,

non-overlapping sections, called elements, in similar manner as in FEM. In two dimensions, these elements are usually triangles or quadrilaterals, in three dimensions they are polyhedrons.

In each element Ω_e, we define *local expansion modes* ϕ_i^e, where $i \in \{1, ..., N_{lm}\}$. These modes are C^0 in Ω and have support only in given element Ω_e. Additionally, they are C^∞ in the Ω_e and create a basis of dimension N_{lm} there.

In most cases, we choose ϕ_i^e to be polynomials, because of easy integration, differentiation and manipulation in general. Nevertheless, for periodic problems, Fourier basis is more suitable. We will stick with polynomials in this chapter. We will denote maximal polynomial order of the basis as P.

Global modes Φ_k are linear combination of local modes ϕ_i^e.

$$\Phi_k(x) = \sum_{e=1}^{N_{el}} \sum_{i=1}^{N_{lm}} a_{i,k}^e \phi_i^e(x) \tag{2.4}$$

where N_{el} is number of elements, N_{lm} number of local modes and $a_{i,k}^e$ are assembly coefficients (coefficients of linear combination).

Assembly coefficients $a_{i,k}^e$ are chosen so that Φ_k satisfy C^0 continuity and are nonzero only in small number of neighboring elements. The process of going from local modes to global ones is called *global assembly* and we will cover it in more detail in section 2.4.

Therefore, we search for u_δ in the space of piece-wise polynomials, based on elemental discretization. So far SEM appears similar to FEM. The main and fundamental difference is, that the polynomial order P of local basis ϕ_i^e is higher in spectral elements than in finite ones.

Because of that, SEM requires efficient manipulation with high-order polynomials. We need to choose "the right" local expansion basis (see section 2.1) and set up robust framework to work with it.

2.2 Spectral accuracy and sensitivity of SEM

In FEM, we usually work only with linear or quadratic polynomials. Convergence to exact solution is achieved by reducing the size of elements, denoted h. Error of numerical solution usually behaves like

$$||u - u_\delta||_{L^2} \sim h^k \tag{2.5}$$

where k is finite constant, depending on regularity of solution.

When we use spectral elements, we are not restricted only to changing the size of elements, we can also increase polynomial order P of the local expansion basis ϕ_i^e.

That is why the spectral element method coincides with hp-FEM. We have two options to reach convergence: *h-type*, reducing size of elements and *p-type* increasing polynomial order.

By increasing P, we can for smooth functions achieve even exponential convergence to exact solution. This is called *spectral* convergence or accuracy and it is one of main advantages of SEM.

Let us clarify the term exponential convergence. We say numerical solution u_δ has exponential convergence in P to exact solution u, if

$$||u - u_\delta||_{L^2} < Ce^{-qP^r} \tag{2.6}$$

where C, q and r are positive constants.

In other words, L^2 error of numerical solution decreases faster, than P^{-k} for any finite order k, because

$$\lim_{k\to\infty} e^{-qP^r} P^k = 0 \tag{2.7}$$

We can expect spectral convergence when the exact solution u is infinitely differentiable. This means, SEM is highly sensitive on regularity of solution and proper boundary condition.

When the solution is not regular, it may be better to reduce h and get smaller elements Ω_e, where we can hope solution will behave better.

For more insight on spectral convergence, we refer reader to [Boyd, 2001].

2.3 Standard element and parametric mapping

As we have learned in previous section, approximate solution u_δ is linear combination of global modes - high-order piece-wise polynomials, which are in turn linear combination of local modes, defined only in the given element Ω_e. Elements can have different sizes and shapes, thus the local modes (local basis) can be different for each Ω_e.

Let us introduce one common *standard* (or reference) *element* Ω_{st}, and *parametric mapping* $\chi_e(\xi)$ unique for each Ω_e. Mapping $\chi_e(\xi)$ provides transformation between standard coordinate ξ and global coordinate x and between Ω_{st} and Ω_e.

Figure 2.1 shows example in two dimensions. Now, standard element is quadrilateral (in fact square), defined as

$$\Omega_{st} = \{-1 \leq \xi_1, \xi_2 \leq 1\} \tag{2.8}$$

Parametric mapping for given element has the form

$$x_i = \chi_e^i(\xi) = x_i^A \frac{-\xi_1 + 1}{2}\frac{-\xi_2 + 1}{2} + x_i^B \frac{\xi_1 + 1}{2}\frac{-\xi_2 + 1}{2} + x_i^C \frac{\xi_1 + 1}{2}\frac{\xi_2 + 1}{2} + x_i^D \frac{-\xi_1 + 1}{2}\frac{\xi_2 + 1}{2}$$

Previous example is simple, because Ω_e has straight sides, but parametric mapping can be also used for expressing more general deformation of elements, such as curved domains. Importantly, parametric mapping is always polynomial in ξ.

Mapping $\chi_e(\xi)$ transforms standard element to given local element

$$\chi_e(\Omega_{st}) = \Omega_e \tag{2.9}$$

and because $\phi_i^e(x)$ is non-zero only in Ω_e, following holds

$$\phi_i^e(\chi_f(\xi)) = \begin{cases} \phi_i(\xi), & e = f \\ 0, & e \neq f \end{cases} \qquad \xi \in \Omega_{st} \tag{2.10}$$

Function $\phi_i(\xi)$ is called *standard* expansion mode [1]. It is defined in Ω_{st}. Using $\chi_e(\xi)$, we can transform $\phi_i(\xi)$ to all elements. Therefore it suffices to construct only *one* standard polynomial basis $\phi_i(\xi)$, with $i \in \{1, ..., N_{lm}\}$, dimension N_{lm}, and use mappings to get all local bases.

Table 2.1 shows definitions of various standard elements. Some of them are easier to work with, because their variables have *independent* bounds. They are segment, quadrilateral and hexahedron and we will call them *simple* elements.

The rest of standard elements have at least one *dependent* bound, where the value of bound of one variable is influenced by others. These are *hybrid* elements.

Hybrid elements can be mapped to simple ones (of the same dimension) using another set of coordinates η and mapping $\eta(\xi)$.

For example, **triangle** (coordinates ξ) is mapped to quadrilateral (coordinates η) with

$$\eta_1 = 2\frac{1+\xi_1}{1-\xi_2} - 1, \qquad \eta_2 = \xi_2 \tag{2.11}$$

and **pyramid** region is mapped to hexahedron with

$$\eta_1 = 2\frac{1+\xi_1}{1-\xi_3} - 1, \qquad \eta_2 = 2\frac{1+\xi_2}{1-\xi_3} - 1, \qquad \eta_3 = \xi_3 \tag{2.12}$$

Mapping $\eta(\xi)$ and its inverse $\xi(\eta)$ will prove useful for integration and differentiation in Ω_{st}.

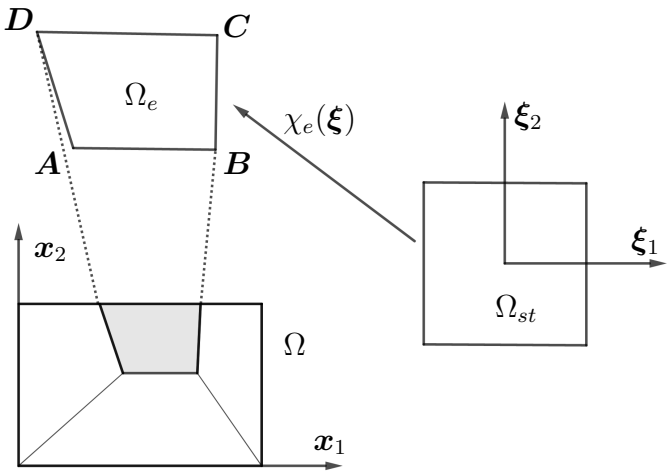

Figure 2.1: Example of relation between domain Ω, local element Ω_e and standard element Ω_{st}.

[1] In literature, one can encounter term "local modes" used for function defined on standard element (i.e. function of variable ξ). However, we decided to reserve term local for functions defined in Ω_e (with global variable x, but non-zero only in Ω_e) and use term "standard" for function on Ω_{st}.

Table 2.1: Standard element in various dimensions

Dimension	Shape	Definition
$1D$	Segment	$\{\xi_1 \mid -1 \leq \xi_1 \leq 1\}$
$2D$	Quadrilateral	$\{(\xi_1, \xi_2) \mid -1 \leq \xi_1, \xi_2 \leq 1\}$
	Triangle	$\{(\xi_1, \xi_2) \mid -1 \leq \xi_1, \xi_2;\ \xi_1 + \xi_2 \leq 0\}$
$3D$	Hexahedron	$\{(\xi_1, \xi_2, \xi_3) \mid -1 \leq \xi_1, \xi_2, \xi_3 \leq 1\}$
	Tetrahedron	$\{(\xi_1, \xi_2, \xi_3) \mid -1 \leq \xi_1, \xi_2, \xi_3;\ \xi_1 + \xi_2 + \xi_3 \leq -1\}$
	Pyramid	$\{(\xi_1, \xi_2, \xi_3) \mid -1 \leq \xi_1, \xi_2, \xi_3;\ \xi_1 + \xi_3, \xi_2 + \xi_3 \leq 0\}$
	Prism	$\{(\xi_1, \xi_2, \xi_3) \mid -1 \leq \xi_1, \xi_2, \xi_3;\ \xi_1 + \xi_2 \leq 0;\ \xi_3 \leq 1\}$

2.4 Global assembly

As we mentioned in section 2.1, global modes are linear combination of local modes, with coefficients $a^e_{i,k}$. Combining (2.2) and (2.4) we get

$$\begin{aligned} u_\delta &= \sum_{k=1}^{N_{gm}} \hat{u}_k \Phi_k(x) = \sum_{k=1}^{N_{gm}} \hat{u}_k \sum_{e=1}^{N_{el}} \sum_{i=1}^{N_{lm}} a^e_{i,k} \phi^e_i(x) \\ &= \sum_{e=1}^{N_{el}} \sum_{i=1}^{N_{lm}} \left(\sum_{k=1}^{N_{gm}} a^e_{i,k} \hat{u}_k \right) \phi^e_i(x) = \sum_{e=1}^{N_{el}} \sum_{i=1}^{N_{lm}} \tilde{u}^e_i \phi^e_i(x) \end{aligned} \tag{2.13}$$

In the last equality, we defined *local* coefficients $\tilde{u}^e_i$

$$\sum_{k=1}^{N_{gm}} a^e_{i,k} \hat{u}_k = \tilde{u}^e_i \tag{2.14}$$

The solution can be expressed using either global coefficients $\hat{u}_k$ or local coefficients $\tilde{u}^e_i$.

Indices e and i can be replaced by new index j, where j numbers every possible combination of e and i, so $j \in \{1, ..., N_{el} \cdot N_{lm}\}$. New index can be expressed as

$$j = N_{lm} \cdot (e - 1) + i \tag{2.15}$$

Global and local coefficients can be written as a vectors

$$\hat{\boldsymbol{u}}_g = \left[\hat{u}_1, ..., \hat{u}_k, ..., \hat{u}_{N_{gm}}\right]^T \tag{2.16}$$

$$\tilde{\boldsymbol{u}}_l = \left[\tilde{u}_1, ..., \tilde{u}_j, ..., \tilde{u}_{N_{el} \cdot N_{lm}}\right]^T = \left[\tilde{u}^1_1, ..., \tilde{u}^1_{N_{lm}}, ..., \tilde{u}^e_i, ..., \tilde{u}^{N_{el}}_1, ..., \tilde{u}^{N_{el}}_{N_{lm}}\right]^T \tag{2.17}$$

where subscripts g and l stands for global and local.

Assembly coefficients are treated in the same way and arranged in a form of so called *assembly matrix* $\boldsymbol{A}$ with entries

$$A_{jk} = A_{(N_{lm} \cdot (e-1) + i), k} = a^e_{i,k} \tag{2.18}$$

Then the matrix form of (2.14) is

$$\boldsymbol{A} \hat{\boldsymbol{u}}_g = \tilde{\boldsymbol{u}}_l \tag{2.19}$$

Operation $\boldsymbol{A}$ *scatters* global coefficients into local.

Assembly coefficients satisfy several conditions. For conforming elements, any local mode contribute to *exactly one* global mode. Thus any row of $\boldsymbol{A}$ contains *only* one nonzero element.

Every column of $\boldsymbol{A}$ contains *at least* one nonzero element. It is because columns are coefficients of linear combination of given global mode $\Phi_k(x)$ (recall (2.4)). That makes $\boldsymbol{A}$ a very sparse matrix, but with full column rank (as $\Phi_k(x)$ are linearly independent).

Having full column rank, $\boldsymbol{A}$ has at least as many rows as columns, but usually more. Equivalently the number of local coefficients $\tilde{\boldsymbol{u}}_l$ is greater or equal to the number of global $\hat{\boldsymbol{u}}_g$. However, we do not have any additional local degrees of freedom, because local coefficients are constrained to ensure the C^0 continuity of global modes. This constrains are enforced by $\boldsymbol{A}$.

In general, C^0 continuity of given $\Phi_k(x)$ is guaranteed when adjacent local modes (contributing to $\Phi_k(x)$ and sharing common boundary) have the same boundary values. We are interested only in modes that are non-zero on the boundary, otherwise the continuity is satisfied trivially.

For non-zero case, it is necessary to match local coefficients of adjacent modes. With right choice of standard basis $\phi_i(\xi)$ (more on that later), no scaling between coefficients is needed and all constrains look like

$$\tilde{u}_i^e = \pm\tilde{u}_j^{e+1} = \hat{u}_k \tag{2.20}$$

where ϕ_i^e and ϕ_j^{e+1} are adjacent local modes with common boundary. Sign depends on parametric mappings $\chi_e(\xi)$ and $\chi_{e+1}(\xi)$ of both elements, because common boundary can mapped with opposite sign (in opposite direction).

With all constrains looking like (2.20), $\boldsymbol{A}$ has entries -1 and 1.

Given properties of $\boldsymbol{A}$, it is not effective to use it as a matrix operator. Instead $\boldsymbol{A}$ can be thought as a form of mapping. Every local coefficient $\tilde{u}_i^e$ is mapped to one global coefficient $\hat{u}_k$ and to sign 1 or -1. This is very efficiently implemented as map m_a between indices

$$m_a : (i, e) \to (k, s) \tag{2.21}$$

where i, e enumerate local modes, k enumerates global modes and $s = \pm 1$. Pair (i, e) forms a key and pair (k, s) is a value.

With knowledge of global coefficients and assembly matrix $\boldsymbol{A}$ (or map m_a) we can get value of any local coefficient. To perform inverse operation, we have to construct $\boldsymbol{A}^{-1}$. As $\boldsymbol{A}$ is generally a non-square matrix with more rows than columns and a full column rank, it has left inverses, one of them is

$$\boldsymbol{A}_{left}^{-1} = \left(\boldsymbol{A}^T\boldsymbol{A}\right)^{-1}\boldsymbol{A}^T, \tag{2.22}$$

and no right inverse.

Inverse mapping to m_a can be also constructed easily. For given index k_0 we just search through values (k, s), stored in m_a, until we find pair containing k_0 and denote it (k_0, s_{found}) Then we take any of the keys (i, e) pointing to found pair and denote it (i_{found}, e_{found}). This is new record for in inverse mapping m_a^{-1} in the form

$$k_0 \to (s_{found}, i_{found}, e_{found}) \tag{2.23}$$

In m_a^{-1}, index k of global mode is a key and tuple (s, i, e) is a value.

Assembly matrix $\boldsymbol{A}$ (or mapping m_a) is necessary for computing integrals over Ω.

2.5 Global integration and differentiation

During computation, we often need to integrate or differentiate functions that reside in approximation space V_δ, but sometimes also arbitrary ones.

For example, Galerkin method requires to perform integration of a function u_δ, expressed as a linear combination global modes, with arbitrary function f.

$$\int_\Omega u_\delta(x)f(x)dx = \sum_{k=1}^{N_{gm}} \hat{u}_k \int_\Omega \Phi_k(x)f(x) = \sum_{k=1}^{N_{gm}} \hat{u}_k \hat{I}_k \tag{2.24}$$

Using (2.4), (2.9) and substitution $x = \chi_e(\xi)$, we get

$$\hat{I}_k = \sum_{e=1}^{N_{el}} \sum_{i=1}^{N_{lm}} \int_{\Omega_e} a_{i,k}^e \phi_i^e(x)f(x) = \sum_{e=1}^{N_{el}} \sum_{i=1}^{N_{lm}} a_{i,k}^e \int_{\Omega_{st}} \phi_i(\xi) f(\chi_e(\xi)) \frac{d\chi_e(\xi)}{d\xi} d\xi \tag{2.25}$$

Putting (2.24), (2.25) and definition of local coefficients (2.14) together brings us formula of global integration

$$\int_\Omega u_\delta(x)f(x)dx = \sum_{e=1}^{N_{el}} \sum_{i=1}^{N_{lm}} \tilde{u}_i^e \int_{\Omega_{st}} \phi_i(\xi) f(\chi_e(\xi)) \frac{d\chi_e(\xi)}{d\xi} d\xi \tag{2.26}$$

useful, when we know local coefficients.

Last integral, denoted $\tilde{I}_i^e$, does not depend on k or the shape of global mode. It is computed within the standard element. Indexes e and i are again changed to j (see (2.15)) and we use (2.18) to get

$$\hat{I}_k = \sum_{e=1}^{N_{el}} \sum_{i=1}^{N_{lm}} a_{i,k}^e \tilde{I}_i^e = \sum_{j=1}^{N_{el} \cdot N_{lm}} A_{jk} \tilde{I}_j \tag{2.27}$$

or, in matrix form

$$\hat{\boldsymbol{I}}_g = \boldsymbol{A}^T \tilde{\boldsymbol{I}}_l \tag{2.28}$$

where $\tilde{\boldsymbol{I}}_l = [\tilde{I}_j]^T$ and $\hat{\boldsymbol{I}}_g = [\hat{I}_k]^T$ are vectors. Operation $\boldsymbol{A}^T$ is called *global assembly* and it is transpose of assembly matrix $\boldsymbol{A}$. It enables us to compute all global integrals within standard element.

Differentiation, for $u_\delta \in V_\delta$, is also transformed from Ω to Ω_{st}. Chain rule and (2.14) yields

$$\frac{du_\delta}{dx} = \sum_{k=1}^{N_{gm}} \hat{u}_k \frac{d}{dx} \Phi_k(x) = \sum_{e=1}^{N_{el}} \sum_{i=1}^{N_{lm}} \tilde{u}_i^e \frac{d\phi_i(\xi)}{d\xi} \frac{d\xi}{dx} \tag{2.29}$$

where $\frac{d\phi_i(\xi)}{d\xi}$ is derivative of i-th standard expansion mode, defined in Ω_{st}, and $\frac{d\xi}{dx} = \left(\frac{d\chi_e}{d\xi}\right)^{-1}$.

For $f \notin V_\delta$, we can use numerical differentiation, defined in section 2.7.

2.6 Integration within standard element

One of the purposes of framework of global, local and standard modes is to transform all integrals evaluated in Ω to sum of integrals evaluated in Ω_{st}. After successful transformation, how would we actually compute them?

Answer lies in numerical quadrature. We take set of Q distinct points $\xi_i \in \Omega_{st}$ and Q weights $w_i \in \mathbb{R}$, where $1 \leq i \leq Q$. Then we can approximate integral of arbitrary function f by finite sum

$$\int_{\Omega_{st}} f(\xi) d\xi \approx \sum_{i=1}^{Q} w_i f(\xi_i) \tag{2.30}$$

Numerical quadratures differ in the choice of quadrature points ξ_i and weights w_i.

In SEM, quadrature points are often equal to roots of orthogonal polynomials. Their characteristic property is high precision when integrating polynomials and unequal distribution across Ω_{st}. They are denser towards the edges of Ω_{st}.

For example, on $\Omega_{st} = [-1, 1]$, we can use Legendre polynomials $L_n(\xi)$, which form orthonormal basis in $[-1, 1]$.

$$(L_n, L_m)_{[-1,1]} = \int_{-1}^{1} L_n(\xi) L_m(\xi) d\xi = \delta_{nm} \tag{2.31}$$

with $L_0(\xi) = 1$. If we then take Q roots of $L_Q(\xi)$ as quadrature points ξ_i and define weights as

$$w_i = \frac{2}{1 - (\xi)^2} \left[\frac{d}{d\xi} (L_Q(\xi)) \Big|_{\xi=\xi_i} \right]^{-2} \tag{2.32}$$

we get famous Gaussian quadrature, which is exact for polynomials of order up to $2Q - 1$. For simple proof, see [Cook, 2008].

Other choices can be Gauss-Radau-Legendre, Gauss-Lobatto-Legendre or Chebyshev quadratures. For more information about quadratures, we refer reader to [Karniadakis, 2005].

When Ω_{st} is quadrilateral or hexahedron, we use Fubini theorem to transform integral over Ω_{st} to one dimensional integrals. For example, when $\Omega_{st} = [-1, 1] \times [-1, 1]$ (with coordinates ξ_1 and ξ_2) Fubini theorem with (2.30) yields

$$\int_{\Omega_{st}} f(\xi_1, \xi_2) = \int_{-1}^{1} \int_{-1}^{1} f(\xi_1, \xi_2) d\xi_1 d\xi_2 = \sum_{i=1}^{Q_1} \sum_{j=1}^{Q_2} w_i w_j f((\xi_1)_i, (\xi_2)_j) \tag{2.33}$$

where $(\xi_1)_i$, $(\xi_2)_j$ are 1D quadrature points and w_i, w_j are weights; $i \in \{1, ..., Q_1\}$ and $j \in \{1, ..., Q_2\}$.

From (2.33) we see that quadrature points in 2D are defined as tensor product of two 1D quadratures. In 3D, situation is analogous, so for quadrature points in quadrilateral and hexahedron we have

$$\xi_{ij} = ((\xi_1)_i, (\xi_2)_j) \tag{2.34}$$

$$\xi_{ijk} = ((\xi_1)_i, (\xi_2)_j, (\xi_3)_k) \tag{2.35}$$

and they form an orthogonal grid with $Q_1 \cdot Q_2$ (or $Q_1 \cdot Q_2 \cdot Q_3$) nodes.

For *hybrid* elements, after employing Fubini theorem, it is necessary to substitute $\xi(\eta)$ (see section 2.3) to transform hybrid element into simple one and get integral with constant bounds. Quadrature points are then defined on coordinates η, using tensor product, as before. So for 2D and 3D *hybrid* elements we have following distribution of quadrature points

$$\xi_{ij} = [\xi_1((\eta_1)_i, (\eta_2)_j)), \xi_2((\eta_1)_i, (\eta_2)_j))] \tag{2.36}$$
$$\xi_{ijk} = [\xi_1((\eta_1)_i, (\eta_2)_j, (\eta_3)_k)), \xi_2((\eta_1)_i, (\eta_2)_j, (\eta_3)_k)), \xi_3((\eta_1)_i, (\eta_2)_j, (\eta_3)_k))] \tag{2.37}$$

where $(\eta_1)_i$, $(\eta_2)_j$, $(\eta_3)_k$ are 1D quadrature points distributions.

For example, when Ω_{st} is triangle, we get

$$\int_{\Omega_{st}} f(\xi_1, \xi_2) = \int_{-1}^{1}\int_{-1}^{-\xi_2} f(\xi_1, \xi_2) d\xi_1 d\xi_2 = \int_{-1}^{1}\int_{-1}^{1} f(\xi_1(\eta), \xi_2(\eta)) \left|\frac{d\xi}{d\eta}\right| d\eta_1 d\eta_2 \tag{2.38}$$

where $\xi(\eta)$ is inverse of $\eta(\xi)$, which equals to (2.11). One has to be careful as $\xi(\eta)$ transforms $(\eta_1, \eta_2) = (\eta_1, 1)$ to one point $(\xi_1, \xi_2) = (-1, 1)$. Thus, quadrature points in coordinate η_2 should not be equal to 1.

2.7 Differentiation within standard element

Quadrature points ξ_i, defined in section 2.6, are helpful not only for integration, but also for numerical differentiation.

Let $l_j(\xi)$ be Lagrange interpolation polynomial with property

$$l_j(\xi_i) = \delta_{ij} \tag{2.39}$$

These polynomials form basis for numerical differentiation on quadrature points. Let $f(\xi)$ be arbitrary function defined in Ω_{st}, and $f_i = f(\xi_i)$. Lagrange interpolation of f yields

$$f(\xi) \approx \sum_{i=1}^{Q} f_i l_i(\xi) \tag{2.40}$$

where Q is total number of quadrature points in Ω_{st}.

If we differentiate (2.40), we get

$$\frac{df(\xi)}{d\xi} \approx \sum_{i=1}^{Q} f_i \frac{dl_i(\xi)}{d\xi} \tag{2.41}$$

which is the formula for numerical differentiation in Ω_{st}, that uses only values at quadrature points ξ_i. Values $\frac{dl_i(\xi)}{d\xi}$ remain constant during computations, so we can just calculate them in advance.

In one dimension, $l_j(\xi_i)$ has order $Q-1$ (recall Q is number of quadrature points) and its defined as

$$l_j(\xi) = \frac{\prod_{i=1, i\neq j}^{Q}(\xi - \xi_i)}{\prod_{i=1, i\neq j}^{Q}(\xi_j - \xi_i)} \tag{2.42}$$

In multiple dimensions, we again need to separate simple and hybrid regions.

For *simple* regions, we suppose quadrature points fits definition (2.34), so they form orthogonal grid with size $Q = Q_1 \cdot Q_2$ in 2D, or size $Q = Q_1 \cdot Q_2 \cdot Q_3$ in 3D.

Lagrange polynomials are then just tensor product of 1D Lagrange polynomials.

$$l_{ij}(\xi_1, \xi_2) = l_i(\xi_1) l_j(\xi_2) \tag{2.43}$$

$$l_{ijk}(\xi_1, \xi_2, \xi_3) = l_i(\xi_1) l_j(\xi_2) l_k(\xi_3) \tag{2.44}$$

For *hybrid* regions, we suppose quadrature points fit definition (2.36). They also form orthogonal grid with the size as before, but in *different* coordinates η. Therefore, it is useful to define Lagrange polynomials also in coordinate η and then transform them into ξ.

How precise is numerical differentiation using (2.41)? Lagrangian polynomials are linearly independent and they form basis of space $\text{span}\{l_1(\xi), ..., l_Q(\xi)\}$.

If function $f \in \text{span}\{l_1(\xi), ..., l_Q(\xi)\}$, then the interpolation and the differentiation are exact. That is a big advantage, because basis function $\phi_i(\xi)$ are usually polynomials, so with Q large enough, we can differentiate them exactly.

The takeaway message is that using only values of f at quadrature points ξ_i, we are able to numerically differentiate it in the same set of points. Furthermore, differentiation can be exact for polynomial f.

2.8 Physical values and coefficients

Quadrature points ξ_j in Ω_{st} can be mapped to local element Ω_e via $\chi_e(\xi)$. If we perform this mapping for all elements of domain Ω, we get set of collocation/nodal points x_i. Each member of this set is defined as

$$x_i = x_{[Q\cdot(e-1)+j]} = \chi_e(\xi_j) \tag{2.45}$$

where Q is number of quadrature points in Ω_{st} and N_{el} is number of elements in Ω. The total number of nodal points x_i is $N_Q = Q \cdot N_{el}$.

Let $\boldsymbol{u}$ be a vector of values of u_δ at nodal points x_i

$$u_i = u_\delta(x_i) \tag{2.46}$$

It has length N_Q and we would sometimes call u_i as *physical values.* [2] .

Vector of physical values $\boldsymbol{u}$ serves as an alternative description of u_δ, the other one being global $\hat{\boldsymbol{u}}$ (or local $\tilde{\boldsymbol{u}}$) coefficients. Transformation from $\boldsymbol{u}$ to $\tilde{\boldsymbol{u}}$ is called *forward transformation.* Opposite process is called *backward transformation.*

Backward transformation is easy, we just evaluate u_δ in nodal points. Supposing $x_j \in \Omega_{e^\star}$ (recall elements do not overlap)

$$u_j = u_\delta(x_j) = \sum_{e=1}^{N_{el}} \sum_{i=1}^{N_{lm}} \tilde{u}_i^e \phi_i^e(x_j) = \sum_{i=1}^{N_{lm}} \tilde{u}_i^{e^\star} \phi_i(\xi_{j-Q\cdot(e^\star-1)}) \tag{2.47}$$

Forward transformation is more difficult. It is in fact Galerkin projection of function u into V_δ. Integrals are evaluated using numerical quadrature - on points x_i, where we know values u_i, so the problem is well-defined.

[2] In Nektar++, corresponding object is called *m_phys*

2.9 Standard expansion basis

Choice of standard expansion basis $\phi_i(\xi)$, with $i \in \{1, ..., N_{lm}\}$ is motivated, besides its completeness and approximation properties, by sparsity and conditioning of the matrix arising from Galerkin discretisation of a given problem.

is motivated, besides its completeness and approximation properties, by sparsity and conditioning of the matrix arising from Galerkin discretisation of a given problem.

One-dimensional basis is created by modifying Jacobi polynomials $J_i^{1,1}$ (see Appendix A)

$$\phi_i^A(\xi) = \begin{cases} \frac{1}{2}(\xi+1), & i = 1 \\ \frac{1}{4}(\xi+1)(-\xi+1)J_{i-2}^{1,1}(\xi), & 1 < i < P+1 \\ \frac{1}{2}(-\xi+1), & i = P+1 \end{cases} \tag{2.48}$$

In Nektar++, this basis is called *Modified_A* and has dimension $N_{lm} = P + 1$, where P is maximal polynomial order of the basis. Only two members, $\phi_1^A(\xi)$ and $\phi_{P+1}^A(\xi)$, have non-zero boundary values. These are called *boundary* modes, all other members are *interior* modes. This greatly simplifies matrix from Galerkin discretization. For more about the reasons to choose this basis, see section 2.9.1.

In multidimensional setting, one constructs basis using tensor product of one dimensional bases $\phi_i^A(\xi)$. For example, tensorial basis for $\Omega_{st} = [-1, 1] \times [-1, 1] \times [-1, 1]$ is

$$\phi_i^{cube}(\xi_1, \xi_2, \xi_3) = \phi_p^A(\xi_1)\phi_q^A(\xi_2)\phi_r^A(\xi_3), \tag{2.49}$$

where $1 \leq p, q, r \leq P + 1$ and

$$i = (p-1)(P+1)^2 + (q-1)(P+1) + r \tag{2.50}$$

2.9.1 Construction of the basis

Let us consider simple case of Galerkin projection. We have $f \in V = L^2$ and we want to find projection of f to space V_δ. This is equivalent to finding the solution u_δ of variational problem

$$(v_\delta, u_\delta) = (v_\delta, f) \tag{2.51}$$

where $v_\delta \in V_\delta$ is test function, $u_\delta \in V_\delta$ is trial function and $(\cdot, \cdot)$ is $L^2(\Omega)$ inner product.

$$(f, g) = \int_\Omega f(x)g(x)dx \tag{2.52}$$

Trial and test functions can be expressed using global modes and coefficients $\hat{v}_k$ and $\hat{u}_k$ as in the section 2.1. Linearity of integral gives us

$$\sum_{l=1}^{N_{gm}} \sum_{k=1}^{N_{gm}} \hat{v}_l(\Phi_l, \Phi_k)\hat{u}_k = \sum_{l=1}^{N_{gm}} \hat{v}_l(\Phi_l, f) \tag{2.53}$$

Coefficients $\hat{u}_k$ (and consequently u_δ) are solution of the variational problem if and only if equation 2.53 holds for any values of $\hat{v}_l$. This means, we have to solve system of N_{gm} linear equations

$$\sum_{k=1}^{N_{gm}} (\Phi_l, \Phi_k)\hat{u}_k = (\Phi_l, f) \qquad\qquad l \in \{1, ..., N_{gm}\} \tag{2.54}$$

Result (2.25) from previous section, with respect to mode Φ_l, yields

$$(\Phi_l, \Phi_k) = \sum_{e=1}^{N_{el}} \sum_{i=1}^{N_{lm}} a_{i,l}^e \int_{\Omega_{st}} \phi_i(\xi) \Phi_k(\chi_e(\xi)) \frac{d\chi_e(\xi)}{d\xi} d\xi \tag{2.55}$$

$$(\Phi_l, f) = \sum_{e=1}^{N_{el}} \sum_{i=1}^{N_{lm}} a_{i,l}^e \int_{\Omega_{st}} \phi_i(\xi) f(\chi_e(\xi)) \frac{d\chi_e(\xi)}{d\xi} d\xi \tag{2.56}$$

First integral can be simplified, because Φ_k is also made of local modes.

$$\int_{\Omega_{st}} \phi_i(\xi) \Phi_k(\chi_e(\xi)) \frac{d\chi_e(\xi)}{d\xi} d\xi = \int_{\Omega_{st}} \phi_i(\xi) \sum_{f=1}^{N_{el}} \sum_{j=1}^{N_{lm}} a_{j,k}^f \phi_j^f(\chi_e(\xi)) \frac{d\chi_e(\xi)}{d\xi} d\xi \tag{2.57}$$

From relation (2.9) yields $e = f$ and we get

$$(\Phi_l, \Phi_k) = \sum_{e=1}^{N_{el}} \sum_{i=1}^{N_{lm}} \sum_{j=1}^{N_{lm}} a_{i,l}^e a_{j,k}^e \int_{\Omega_{st}} \phi_i(\xi) \phi_j(\xi) \frac{d\chi_e(\xi)}{d\xi} d\xi \tag{2.58}$$

Index e numbers individual elements, indices i, j numbers standard basis.

There is not much to do with the second integral, as we do not know anything about the function f.

Integrals (Φ_l, Φ_k) form entries of *mass* matrix $\boldsymbol{M}$. This is the matrix we ought to simplify and our tool is the right choice of standard basis $\phi_i(\xi)$. Basis $\phi_i(\xi)$ influences not only values of integrals in standard element, but also the assembly matrix $\boldsymbol{A}$, because entries in $\boldsymbol{A}$ express continuity constrains of global modes.

Unfortunately, it is difficult to say something about $\boldsymbol{M}$, because we do not know anything specific about domain Ω or about number and shape of individual elements.

We have to content with some general ideas. First of them is *orthogonality*. Integral appearing in (2.58) is very similar to L^2 inner product on Ω_{st}, only Jacobian $\frac{d\chi_e(\xi)}{d\xi}$ is left over. If we choose $\phi_i(\xi)$ to be orthogonal polynomial basis in Ω_{st} then L^2 inner product

$$(\phi_i(\xi), \phi_j(\xi)) = \int_{\Omega_{st}} \phi_i(\xi) \phi_j(\xi) d\xi = C_{ij} \delta_{ij} \tag{2.59}$$

where C_{ij} is constant. Even with Jacobian $\frac{d\chi_e(\xi)}{d\xi}$ the result is not much worse. For Ω_e with the same shape (not necessarily size) as Ω_{st} it is just constant and we are fine. For deformed elements, it is polynomial in ξ, with polynomial order J_P. If $\mid i - j \mid > J_P$ then the resulting integral is still zero. Thus many of the entries (Φ_l, Φ_k) of $\boldsymbol{M}$ can be zero too.

On a first sight, this pure orthogonal basis seems to be very useful. We take a closer look on an example in one dimension, with $\Omega_{st} = [-1, 1]$. Orthogonal polynomials here are Legendre polynomials L_i so we take $\phi_i(\xi) = L_{i-1}(\xi)$, with $i \in \{1, ..., P+1\}$ to get polynomial basis with order P. Explicit representation (see Appendix A) of the Legendre polynomials is

$$L_i(\xi) = \frac{1}{2^i} \sum_{k=0}^{i} \binom{i}{k}^2 (\xi - 1)^{i-k} (\xi + 1)^k \tag{2.60}$$

Domain Ω_{st} and basis $L_{i-1}(\xi)$ is transformed to all elements Ω_e and local modes $\phi_i^e(x)$ via mappings $\chi_e(\xi)$. The orthogonality is preserved for constant Jacobians.

Supposing that Jacobian $\frac{d\chi_e(\xi)}{d\xi}$ is constant in every element, we derive following result about orthogonality of global modes.

First, we recall that every local mode contributes to exactly one global. When mode $\phi_i^e(x)$ is a part of global mode Φ_k, then no other global mode can contain it. Thus Φ_k is orthogonal to all other global modes in Ω_e. This holds for all elements, so global modes are mutually orthogonal in whole Ω. Matrix $\boldsymbol{M}$ is then diagonal.

With polynomial Jacobian, orthogonality can be at least partially preserved and $\boldsymbol{M}$ can be multi-diagonal matrix. Word "can" depends on polynomial order of basis and Jacobian. Despite of nice looking matrix $\boldsymbol{M}$, Lagrange basis has significant problem - with boundaries.

Boundary of Ω_{st} is $\{-1, 1\}$. After inserting these values into (2.60) we see that L_i is non-zero on both boundaries for any i. We also notice

$$L_i(-1) = (-1)^i L_i(1) \qquad \forall i \tag{2.61}$$

Domain Ω_{st} and basis $L_{i-1}(\xi)$ is transformed to all elements Ω_e and local modes $\phi_i^e(x)$ via mappings $\chi_e(\xi)$. Their boundary values are preserved. To ensure the C^0 continuity of global modes, adjacent local modes have to match their boundary values.

Let us consider two adjacent elements Ω_e and Ω_f, sharing common point x_{ef} (in one dimension, it is the only possibility). Point x_{ef} is "right" boundary of Ω_e and "left" boundary for Ω_f.

From (2.61) we derive

$$\phi_i^e(x_{ef}) = L_i(1) = (-1)^i L_i(-1) = (-1)^i \phi_i^f(x_{ef}) \tag{2.62}$$

which is the matching condition or continuity constrain. Any global mode that contains ϕ_i^e also have to contain ϕ_i^f with appropriate sign. Translated into language of assembly coefficients

$$a_{i,k}^e = (-1)^i a_{i,k}^f \tag{2.63}$$

Condition (2.63) can be "chained" through all adjacent elements. When global mode $\Phi_k(x)$ contains local mode of order i in one element Ω_e, then it has to contain mode of order i in all elements. Consequently, each global mode spans through whole domain Ω.

These "giant" modes are contrary to the idea of dividing Ω into smaller sections. Additionally, it shrinks the space of global modes V_δ, so we have less global degrees of freedom, but still a lot of elements to care about. Although mass matrix $\boldsymbol{M}$ is nice, $\Phi_k(x)$ do not make up good approximation space.

We showed choice of standard basis $\phi_i(\xi) = L_i(\xi)$ leads to bad space V_δ. The example was only one dimensional, but it demonstrate the another general idea that having a lots of local modes with non-zero boundaries leads to trouble.

What is the solution? We do not want to get rid of orthogonality completely. Maybe we can alter polynomials $L_i(\xi)$ to zero boundary values. Easiest way to do this is just to multiply $L_i(\xi)$ with linear functions that are zero on given

boundary. Let us create a new standard element basis of polynomial order P

$$\phi_i(\xi) = \phi_i^L(\xi) = \begin{cases} \frac{1}{2}(\xi+1), & i=1 \\ \frac{1}{4}(\xi+1)(-\xi+1)L_{i-2}(\xi), & 1<i<P+1 \\ \frac{1}{2}(-\xi+1), & i=P+1 \end{cases} \tag{2.64}$$

Basis functions $\phi_i^L(\xi)$ for $1 < i < P+1$ are Legendre polynomials altered to have zero boundary, called *interior* modes. Two remaining basis functions take care of connecting elements and we call them *boundary* modes. For convenience, we denote this particular basis with superscript L. Standard element is still the same $\Omega_{st} = [-1, 1]$

Continuity constrains will be much easier to construct, because we need to connect only modes $\phi_1^L(\xi)$ and $\phi_{P+1}^L(\xi)$. Using notation from previous example

$$\phi_1^e(x_{ef}) = \phi_1^L(1) = \phi_{P+1}^L(-1) = \phi_{P+1}^f(x_{ef}) \tag{2.65}$$

$$a_{1,k}^e = a_{P+1,k}^f \tag{2.66}$$

These conditions do not "chain" across whole domain Ω. Global modes consists of either only one interior local mode or two adjacent boundary local modes (satisfying continuity constrains). We retained large number of global degrees of freedom.

Let us have a look at orthogonal properties of this basis. Without loss of generality, let us take $\phi_i^L(\xi)$ and $\phi_j^L(\xi)$, where $1 \leq i \leq j \leq P$. Considering L^2 inner product on Ω_{st}

$$\int_{\Omega_{st}} \phi_i^L(\xi)\phi_j^L(\xi)d\xi = \begin{cases} \int_{\Omega_{st}} \left(\frac{1}{2}(\xi+1)\right)^2 d\xi, & i=j=1 \\ \int_{\Omega_{st}} \left(\frac{1}{2}(\xi+1)\right)^2 \frac{1}{2}(-\xi+1)L_{j-2}(\xi), & i=1<j\leq P \\ \int_{\Omega_{st}} \left(\frac{1}{4}(\xi+1)(-\xi+1)\right)^2 L_{i-2}(\xi)L_{j-2}(\xi), & 1<i,j\leq P \end{cases} \tag{2.67}$$

we see that it splits into several cases. We omitted cases with $\phi_{P+1}^L(\xi)$, because they are similar to $\phi_1^L(\xi)$.

Legendre polynomial L_{j-2} is orthogonal to polynomials with order $j-3$ or lower. For first case it does not matter. Second case is zero, when $j-2 > 3$ so $j > 5$. Integral in last case is zero for $j-2 > 4+i-2$ so $j > 4+i$.

Second case represents coupling of interior modes $\phi_j^L(\xi)$ with boundary mode $\phi_1^L(\xi)$. We want to suppress the coupling, because it connects interior modes of one element with interior modes via linear boundary modes.

Can we have even better properties? Answer is positive. We can keep the structure of the basis $\phi_i^L(\xi)$ and use better set of polynomials instead of L_i. We denote new basis $\phi_i^P(\xi)$ and it is defined by (2.64) with not yet specified polynomials P instead of L.

Orthogonal polynomials are generally defined with respect to L^2 inner product with weight function $w(x)$

$$(f,g)_w = \int_{\Omega_{st}} f(x)g(x)w(x)dx \tag{2.68}$$

L_i are derived with respect to $w(x) = 1$. We chose them before, because with $\phi_i(\xi) = L_{i-1}(\xi)$ inner product on Ω_{st} contains no other term besides function L_i and L_j themselves.

With base $\phi_i^P(\xi)$, the situation is different. In all cases [3] of (2.67), there are additional terms in integrals besides Legendre polynomials L_i. All are positive in Ω_{st}, so we consider them as candidates for weight function

$$w_1 = (\xi+1)(-\xi+1) \tag{2.69}$$
$$w_2 = (\xi+1)^2(-\xi+1) \tag{2.70}$$
$$w_3 = (\xi+1)(-\xi+1)^2 \tag{2.71}$$
$$w_4 = (\xi+1)^2(-\xi+1)^2 \tag{2.72}$$
$$\tag{2.73}$$

Function w_3 was not mentioned in (2.67), but it would appear in inner products with $\phi_{P+1}^P(\xi)$.

General form of the weight function is

$$w = (\xi+1)^\alpha(-\xi+1)^\beta \tag{2.74}$$

and associated orthogonal polynomials are Jacobi polynomials $J_i^{\alpha,\beta}(\xi)$ (see Appendix A).

All considered set of orthogonal polynomials are just special case of Jacobi polynomials. For example, Legendre polynomials $L_i = J_i^{0,0}$.

Polynomials $J_i^{1,1}(\xi)$ associated with w_1 are better than L_i in all cases of (2.67). For example, integral in second case is

$$\int_{\Omega_{st}} \left(\frac{1}{2}(\xi+1)\right)^2 \frac{1}{2}(-\xi+1)J_{j-2}^{1,1}(\xi) = \int_{\Omega_{st}} \frac{1}{8}(\xi+1)w_i(\xi)J_{j-2}^{1,1}(\xi) \tag{2.75}$$

and there is only linear function left over. For $j > 3$, this case would be zero. Integral in last case is zero for $j > 2+i$.

With $J_i^{2,2}(\xi)$ and weight function w_4, third case has ideal properties, so interior modes are orthogonal. Unfortunately, function w_4 is never contained in second case of (2.67). We completely lost orthogonality between interior and boundary modes and they are coupled. As we mentioned before, coupling between them is something we want to avoid.

Function w_2 and w_3 are non-symmetric and they also cause coupling with boundary modes. In the case of w_2, all modes are coupled with $\phi_{P+1}^P(\xi)$. In the case of w_3 with mode $\phi_1^P(\xi)$.

We conclude that the best candidate is w_1 and polynomials $J_i^{1,1}(\xi)$. Although not ideal for interior modes, they are better than L_i and they do not cause too much coupling.

Putting all together we get basis

$$\phi_i^A(\xi) = \begin{cases} \frac{1}{2}(\xi+1), & i = 1 \\ \frac{1}{4}(\xi+1)(-\xi+1)J_{i-2}^{1,1}(\xi), & 1 < i < P+1 \\ \frac{1}{2}(-\xi+1), & i = P+1 \end{cases} \tag{2.76}$$

that we wanted to derive.

[3] In following, when we talk about *all cases* of L^2 weighted product, we mean only cases where orthogonal polynomial are present. Clearly, choice of orthogonal polynomials can not influence other cases.

3. Temporal discretization

Temporal discretization is implemented in the framework of *General Linear Methods* (or *GLM*). GLM was first introduced by [Butcher, 1996]. The goal is to unify wide range of time-stepping methods (for example, both multi-step and multi-stage methods can be described as GLM) and give them common interface, in the form of coefficients matrices. Then, just by changing values and shape of these matrices, one can effortlessly switch between different time-stepping methods.

Nektar++ implements advanced version of GLM, modified for time-dependent PDEs, as presented in [Vos et al., 2011]. It also extends original concept of GLM to incorporate implicit-explicit (*IMEX*) schemes. We will explain this version in following example.

Let us have partial differential equation for scalar function $u(\boldsymbol{x}, t)$ defined on domain $\Omega \times [0, T)$

$$\begin{aligned} \frac{\partial u}{\partial t} &= g(u) + f(u) && \text{in } \Omega \times [0, T) \\ \frac{\partial u}{\partial n}(\boldsymbol{x}, t) &= u_N(\boldsymbol{x}, t) && \text{on } \partial\Omega_N \times [0, T) \\ u(\boldsymbol{x}, t) &= u_D(\boldsymbol{x}, t) && \text{on } \partial\Omega_D \times [0, T) \\ u(\boldsymbol{x}, 0) &= u_0(\boldsymbol{x}) && \text{in } \Omega \end{aligned}$$

where g and f are functions or differential operators, g is usually linear and it is called *implicit term*, but f can be non-linear and is called *explicit term*. Sets $\partial\Omega_D$ and $\partial\Omega_N$ are parts of boundary of Ω, where we prescribed Dirichlet or Neumann boundary conditions.

Let us suppose we spatially discretized function u with the help of spectral or finite elements and got approximation $u_\delta \in V_\delta$, determined by vector of concatenated local coefficients $\tilde{\boldsymbol{u}} = [\tilde{u}_j(t)]^T$ (see definition (2.17)), where $\tilde{u}_j = \tilde{u}_{k+e \cdot N_{lm}} = \tilde{u}_k^e$ (we omitted symbol δ in vector $\tilde{\boldsymbol{u}}$ to avoid too many indices).

Using backward transformation, we obtained $\boldsymbol{u}$ - vector of values of u_δ at nodal points x_i (see definition (2.45)).

In Nektar++ implementation of GLM, we work in "physical" space, with vector $\boldsymbol{u}$, but we are not losing any information. We recall it is possible to switch between physical values $\boldsymbol{u}$ and coefficients $\tilde{\boldsymbol{u}}$ using forward and backward transformation (implemented in class *Explist*), so either physical values or coefficients uniquely determine function u_δ.

To introduce GLM, we need additional notation.

N_Q	total number of nodal points x_i in Ω	(3.1)
N_s	number of stages	(3.2)
N_r	number of steps	(3.3)
$\boldsymbol{u}^n$	values of u_δ at nodal points (vector $\boldsymbol{u}$) at time t_n	(3.4)
$\tilde{\boldsymbol{u}}^n$	local coeffients determining u_δ at time t_n	(3.5)
$\boldsymbol{u}^{[n]}$	solution vector at time t_n	(3.6)
$\boldsymbol{t}^{[n]}$	time vector belonging to sol. vector $u^{[n]}$	(3.7)
$\boldsymbol{\theta}_j$	values of j-th pre-stage at nodal points at time t_n	(3.8)
$\boldsymbol{s}_j$	values of j-th stage at nodal points	(3.9)
τ_j	time of j-th stage	(3.10)
$\boldsymbol{g}(\boldsymbol{s}_j)$	implicit stage derivative vector at time t_n	(3.11)
$\boldsymbol{f}(\boldsymbol{s}_j)$	explicit stage derivative vector at time t_n	(3.12)
$\boldsymbol{A}^{IM}$	$N_s \times N_s$ stage coeff. matrix for $\boldsymbol{g}(\boldsymbol{s}_j)$; $a_{ij}^{IM} = 0 \quad j > i$	(3.13)
$\boldsymbol{A}^{EX}$	$N_s \times N_s$ stage coeff. matrix for $\boldsymbol{f}(\boldsymbol{s}_j)$; $a_{ij}^{EX} = 0 \quad j \geq i$	(3.14)
$\boldsymbol{B}^{IM}$	$N_r \times N_s$ output coeff. matrix for $\boldsymbol{g}(\boldsymbol{s}_j)$	(3.15)
$\boldsymbol{B}^{EX}$	$N_r \times N_s$ output coeff. matrix for $\boldsymbol{f}(\boldsymbol{s}_j)$	(3.16)
$\boldsymbol{U}$	$N_s \times N_r$ stage coeff. matrix for input sol. vector $\boldsymbol{u}^{[n-1]}$	(3.17)
$\boldsymbol{V}$	$N_r \times N_r$ output coeff. matrix for input sol. vector $\boldsymbol{u}^{[n-1]}$	(3.18)

Solution vector $\boldsymbol{u}^{[n]}$ is a 2D structure, "supervector", its elements are other vectors. We do not use term "matrix", because it does not make sense to work with rows of $\boldsymbol{u}^{[n]}$.

$$\boldsymbol{u}^{[n]} = \left[\boldsymbol{u}_1^{[n]}, \boldsymbol{u}_2^{[n]}, ..., \boldsymbol{u}_{N_r}^{[n]}\right] \tag{3.19}$$

Each element $\boldsymbol{u}_i^{[n]}$ is vector with length equal to N_Q. First element $\boldsymbol{u}_1^{[n]}$ is always equal to $\boldsymbol{u}^{n-1}$, solution from previous time level. The rest of components are auxiliary vectors, specific for the given time integration scheme.

Time vector $\boldsymbol{t}^{[n]}$ has length N_r and its elements $t_i^{[n]}$ are real numbers, representing time instances or time intervals. First element $t_1^{[n]}$ is always equal to t_n.

Pre-stage value vector $\boldsymbol{\theta}_j$ and stage value vector $\boldsymbol{s}_j$ also have length N_Q, their components $(\theta_j)_i$ and $(s_j)_i$ are equal to value of j-th (pre-)stage at point x_i.

Stage derivative vectors $\boldsymbol{g}(\boldsymbol{s}_j)$ and $\boldsymbol{f}(\boldsymbol{s}_j)$ are defined as

$$\boldsymbol{g}(\boldsymbol{s}_j) = \left[(g(\boldsymbol{s}_j))_i\right]^T \tag{3.20}$$

$$\boldsymbol{f}(\boldsymbol{s}_j) = \left[(f(\boldsymbol{s}_j))_i\right]^T \tag{3.21}$$

where i goes from 1 to N_Q. It is necessary to explain, what is meant by $(g(\boldsymbol{s}_j))_i$.

It is value of g in nodal point x_i. When g is function, that it has clear meaning

$$(g(\boldsymbol{s}_j))_i = g((s_j)_i) \tag{3.22}$$

because g is evaluated point-wise, argument of g is real number.

When g is differential operator, then the argument is function. So it should be provided with entire stage vector $\boldsymbol{s}_j$, to be able to numerically differentiate it.

3.1 GLM algorithm summary for scalar case

Goal of the algorithm is to obtain vectors $\boldsymbol{u}^{[n+1]}$ and $\boldsymbol{t}^{[n+1]}$ from old vectors $\boldsymbol{u}^{[n]}$ and $\boldsymbol{t}^{[n]}$.

Again, we emphasize that the algorithm is carried out in physical space, all vectors (apart from $\boldsymbol{t}^{[n]}$) are connected with nodal points and have length N_Q.

The algorithm consists of two main parts:

1. **Calculating stages**

   For i from 1 to number of stages N_s compute

   (a) **Pre-stage value $\boldsymbol{\theta}_i$**

   $$\boldsymbol{\theta}_i = \Delta t \sum_{j=1}^{i-1} a_{ij}^{IM} \boldsymbol{g}(\boldsymbol{s}_j) + \Delta t \sum_{j=1}^{i-1} a_{ij}^{EX} \boldsymbol{f}(\boldsymbol{s}_j) + \sum_{j=1}^{N_r} u_{ij} \boldsymbol{u}_j^{[n]} \tag{3.23}$$

   For given i, all terms in the right hand side are already known, so this step is purely explicit.

   (b) **Stage time τ_i**

   $$\tau_i = \Delta t \sum_{j=1}^{i-1} a_{ij}^{EX} + \sum_{j=1}^{N_r} u_{ij} t_j^{[n]} \tag{3.24}$$

   (c) **Stage value $\boldsymbol{s}_i$**

   To get value $\boldsymbol{s}_i$, we need to solve PDE

   $$s_i(\boldsymbol{x}) - \left(\Delta t \cdot a_{ii}^{IM}\right) g(s_i(\boldsymbol{x})) = \theta_i(\boldsymbol{x}) \qquad \text{in } \Omega \tag{3.25}$$

   $$\frac{\partial s_i}{\partial n}(\boldsymbol{x}) = u_N(\boldsymbol{x}, \tau_i) \qquad \text{on } \partial\Omega_N \tag{3.26}$$

   $$s_i(\boldsymbol{x}) = u_D(\boldsymbol{x}, \tau_i) \qquad \text{on } \partial\Omega_D \tag{3.27}$$

   We do not know pre-stage function θ_i entirely, we only have its values at x_i - pre-stage vector $\boldsymbol{\theta}_i$. Thus, this PDE should be discretized in same space V_δ as the original equation (3.1), with the *same* set of nodal points. Then we can use $\boldsymbol{\theta}_i$ for evaluating integrals numerically and do not worry about rest of the function θ_i.

   Stage value vector $\boldsymbol{s}_i$ is solution s_i evaluated at points x_i.

   It is one of the two steps in entire algorithm, where boundary conditions of original function u and all the heavy machinery of spectral (finite) elements comes into play.

   (d) **Stage derivatives $\boldsymbol{g}(\boldsymbol{s}_i)$ and $\boldsymbol{f}(\boldsymbol{s}_i)$**

   For implicit stage derivative $\boldsymbol{g}(\boldsymbol{s}_i)$, we have formula

   $$\boldsymbol{g}(\boldsymbol{s}_i) = \frac{\boldsymbol{s}_i - \boldsymbol{\theta}_i}{\Delta t \cdot a_{ii}^{IM}} \tag{3.28}$$

   If $a_{ii}^{IM} = 0$, then the scheme is purely explicit and we need only explicit stage derivative (function f); implicit stage derivative (function g) is omitted.

Explicit stage derivative $\boldsymbol{f}(\boldsymbol{s}_i)$ is computed according to definition (3.21).

2. **Assembling new solution vector $\boldsymbol{u}^{[n+1]}$ and time vector $\boldsymbol{t}^{[n+1]}$**

   In this part of the scheme, we use all previously computed stages $\boldsymbol{s}_i$ to calculate new solution vector . It consists of following calculations:

   (a) **Time vector $\boldsymbol{t}_i^{[n]}$**

   For i from 1 to number of steps N_r compute

$$t_i^{[n+1]} = \Delta t \sum_{j=1}^{N_s} b_{ij}^{EX} + \sum_{j=1}^{N_r} v_{ij} t_j^{[n]} \tag{3.29}$$

   (b) **Pre-solution $\boldsymbol{w}$**

   In analogy to pre-stage value, we calculate pre-solution. It is called pre-solution, because it is not necessarily in the approximation space V_δ.

$$\boldsymbol{w} = \Delta t \sum_{j=1}^{N_s} b_{1j}^{IM} \boldsymbol{g}(\boldsymbol{s}_j) + \Delta t \sum_{j=1}^{N_s} b_{1j}^{EX} \boldsymbol{f}(\boldsymbol{s}_j) + \sum_{j=1}^{N_r} v_{1j} \boldsymbol{u}_j^{[n]} \tag{3.30}$$

   (c) **Solution $\boldsymbol{u}_1^{[n+1]}$**

   Project pre-solution $\boldsymbol{w}$ into approximation space V_δ using Galerkin projection.

$$u_1^{[n+1]}(\boldsymbol{x}) = w(\boldsymbol{x}) \qquad \text{in } \Omega \tag{3.31}$$

$$\frac{\partial u_1^{[n+1]}}{\partial n}(\boldsymbol{x}) = u_N(\boldsymbol{x}, t_{n+1}) \qquad \text{on } \partial\Omega_N \tag{3.32}$$

$$u_1^{[n+1]}(\boldsymbol{x}) = u_D(\boldsymbol{x}, t_{n+1}) \qquad \text{on } \partial\Omega_D \tag{3.33}$$

   This is the second step in algorithm, where boundary conditions of original function u and SEM comes into play. We need to use *same* set of nodal points as in computation of stage values $\boldsymbol{s}_i$.

   (d) **Rest of solution vector $\boldsymbol{u}_i^{[n+1]}$**

   Rest of solution vector does not need to be in approximation space V_δ.

$$\boldsymbol{u}_i^{[n+1]} = \Delta t \sum_{j=1}^{N_s} b_{ij}^{IM} \boldsymbol{g}(\boldsymbol{s}_j) + \Delta t \sum_{j=1}^{N_s} b_{ij}^{EX} \boldsymbol{f}(\boldsymbol{s}_j) + \sum_{j=1}^{N_r} v_{ij} \boldsymbol{u}_j^{[n]} \tag{3.34}$$

4. Implementation of the Wilcox k-ω model

4.1 Spectral/hp element framework Nektar++

Nektar++ is modern open-source cross-platform spectral/hp element framework, written in C++ [Cantwell et al., 2015], published under MIT license.

It supports arbitrary-order spectral/hp element discretization in one, two or three dimensions. Elements can have complex geometry, described by high order polynomials. Expansion basis is not restricted only to polynomials (described in chapter 2), but also to Fourier basis. Framework enables us to use continuous Galerkin, discontinuous Galerkin and hybridized discontinuous Galerkin projections.

Nektar++ is efficient in parallel communication using MPI and suitable for large-scale simulations on high-performance computing (HPC) clusters with thousands of CPU.

It also contains wide range of solvers for different areas of research, ranging from both compressible and incompressible Navier-Stokes solver to pulse wave or cardiac electrophysiology solver.

Nektar++ was developed by international team, mainly based in Imperial College London and University of Utah. Community around Nektar++ is very active. Development of the framework still continues and new features are added regularly. As of May 2020, the latest version is 5.0.0, released in December 2019. There is also annual workshop, taking place in Imperial College London, aiming to familiarize broader public with the framework.

All these features make Nektar++ one of best choices among the open-source software for performing spectral/hp computations.

4.1.1 Basic structure

Nektar++ is written in modern C++ using objective oriented programming and encapsulation of key objects.

Its structure is briefly outlined in [Cantwell et al., 2015] and [Nektar++ User Guide, 2017], with the former describing basic sections (in fact namespaces) and the latter serving as manual for using the framework. Neither of them is meant for developing new solver within the framework.

More details are given by *developer guide* ([Developer Guide for Nektar++, 2017]), especially about spatial discretization. Unfortunately, developer guide is still largely incomplete and it contain less information about temporal discretization or individual solvers.

There is also low-level documentation called *doxygen*, [Doxygen, 2017]. Doxygen is documentation generated directly from annotated (commented) source code. It keeps track of variables, functions, classes, objects and dependencies between them. One can generate it from source code, or find it on the Nektar++ website.

Doxygen is useful source when exploring the framework, however, not all

parts of the code are commented properly. In addition, Nektar++ contains huge amount of code (more than 1 million lines), so when dealing with "raw" source code, one can easily get confused and lose track of the higher level of the abstraction.

In summary, it is difficult to find complete and detailed information about the framework, especially about temporal discretization or individual solvers. Thus, it can be intimidating to try to implement new solver.

We will try to address these issues in this chapter and also in chapter 4.

In this chapter, we briefly introduce entire structure of Nektar++, but also provide more details about temporal discretization.

These details would be useful during implementation of our turbulent model (covered in chapter 4). Turbulent model will also serve as an example of implementation of an individual solver.

4.1.2 Namespaces

Basic hierarchy of Nektar++ framework coincide with C++ namespaces. At the top level, we can find main namespace simply called *Nektar* and several other namespaces (*Blas*, *LAPACK*, *Polylib*...) that serves as wrappers for external libraries.

Main namespace *Nektar* is then divided into several section (again namespaces), the most important are

LibUtilities Libraries for different purposes, for example linear algebra routines, quadrature points distributions, time integration, definition of basis...

StdRegions Interface to standard element Ω_{st}. Given basis $\phi_j(\xi)$ and quadrature points distribution ξ_i (from LibUtilities) it provides user with core operations defined in Ω_{st}, such as

- **BwdTrans** Transformation from coefficients $\tilde{u}_j$ to values at quadrature points $u(\xi_i)$. Number N_{lm} corresponds to the number of standard modes (= dimension of basis) in Ω_{st}.

$$u(\xi_i) = \sum_{j=1}^{N_{lm}} \phi_j(\xi_i)\hat{u}_j \tag{4.1}$$

  In matrix form, where $B_{ij} = \phi_j(\xi_i)$

$$\boldsymbol{u} = \boldsymbol{B}\tilde{\boldsymbol{u}} \tag{4.2}$$

- **IProductWRTBase** Compute L^2 inner product (with weight $w = 1$) of arbitrary function f with the respect to base mode $\phi_j(\xi)$ in Ω_{st}. It uses Gaussian quadrature on Q quadrature points.

$$I_j = (f, \phi(\xi)_j)_{L^2} = \int_{\Omega_{st}} f(\xi)\phi_j(\xi)d\xi \approx \sum_{i=1}^{N_{qp}} w_i f(\xi_i)\phi_j(\xi_i) \tag{4.3}$$

  In matrix form, where $W_{ij} = w_i\delta_{ij}$ with w_i being quadrature weights and $B_{ij} = \phi_j(\xi_i)$

$$\boldsymbol{I} = \boldsymbol{B}^T\boldsymbol{W}\boldsymbol{f} \tag{4.4}$$

- **PhysDeriv** For given direction ξ_i returns values of the derivative $\frac{\partial u}{\partial \xi_i}$ of the function u. This routine works in physical space, both input and output vector are values at ξ_i. In matrix form, where D is derivative matrix, computed only once

$$\frac{\partial \boldsymbol{u}}{\partial \xi_i} = \boldsymbol{D}\boldsymbol{u} \tag{4.5}$$

Basis $\phi_j(\xi)$ in standard element is called *standard expansion*. In 2D and 3D case, basis is constructed as a tensor product of 1D basis functions.

Nektar++ supports different types of Ω_{st}, their overview is given in table 2.1. In Nektar++ framework, each type is represented by special class (for example *StdQuadExp* for quadrilateral shape). All of these specific classes inherit from base classes *StdExpansion* and *StdExpansionXD*, where X correspond to dimensionality of Ω_{st}. These base classes define core operations mentioned above as virtual functions. This enables user to perform operations regardless of the type of Ω_{st}.

SpatialDomains Part of the framework that is dealing with mesh and physical geometry of individual local elements Ω_e. It also constructs elements from provided mesh through the class *MeshGraph*.

Local element can be imagined as a deformed standard element located in specific point x in space (= in domain Ω). Geometry classes in SpatialDomains describe both deformation - through parametric mapping $\chi_e(\xi)$ - and location, through coordinates x_i of vertices of the Ω_e.

For each type of standard element, there is a geometry class in SpatialDomains (for example *QuadGeom*). They all inherit from base classes *Geometry* and *GeometryXD*. It is similar structure as in namespace StdRegions.

Besides parametric mapping $\chi_e(\xi)$ and coordinates of vertices, these classes also contain Jacobian $\frac{\partial \chi_e}{\partial \xi}$ of the mapping, in the member class *GeomFactors*, necessary for integration in Ω_e.

LocalRegions In this namespace, geometry information from *SpatialDomains* and standard expansion from *StdRegions* are put together.

Local element Ω_e is represented by class *Expansion*. This class inherits from class *StdExpansion* and contains member class *Geometry* from SpatialDomains.

Operation originally defined in *StdExpansion* are overwritten in *Expansion*, and altered by geometry (deformation) of local element. For example, backward transformation in local element becomes

$$u(x) = \sum_{j=1}^{N_{lm}} \phi_j(\chi_e^{-1}(x))\tilde{u}_j \tag{4.6}$$

where $\tilde{u}_j$ are local coefficients for given element Ω_e

Integration in Ω_e needs Jacobian $\frac{\partial \chi_e}{\partial \xi}$ of parametric mapping. Values of Jacobian in quadrature points are incorporated into quadrature weights w_i.

MultiRegions Contains classes and routines for dealing with solution on a global level, in the whole domain Ω.

The most important class here is *ExpList*, which is, as the name suggests, list (collection) of *Expansion* objects. This way, the instance of *ExpList* represents domain Ω - also just collection of elements Ω_e (represented by *Expansion* objects).

Each Expansion and thus each Explist stores coefficients and values at nodal points of only *one* scalar variable (field). For example, when you are solving incompressible Navier-Stokes equations in three dimension, you will need 4 instances of *Explist* - three for velocity vector components u, v, w and one for pressure p. Fortunately, Nektar++ builds those automatically according to input file.

Explist supports global operations, defined in Ω. To perform them, it combines global assembly matrix $\boldsymbol{A}$ (see (2.19)) and matrices $\boldsymbol{B}^e$ or $\boldsymbol{W}^e$ for individual elements (stored in *Expansion*).

Most important operations are:

- **BwdTrans** Transformation from local coefficients[1] $\tilde{u}_j$ to physical values at nodal points (see definition (2.46)) $u(x_i)$.
- **FwdTrans** Transformation from physical values at nodal points $u(x_i)$ to local coefficients $\tilde{u}_j$, in other words projection of function u to trial space V_δ with basis $\phi_j(x)$.
- **PhysDeriv** For given direction x_i returns values of the derivative $\frac{\partial u}{\partial x_i}$ of the function u. This routine works in physical space, both input and output vector are values at nodal points x_i.
- **HelmSolve** Solve Helmholtz-like equation (Helmholtz equation with opposite sign)
$$\Delta u - \lambda u = f \tag{4.7}$$
This equation is supplemented by appropriate boundary conditions stored in member variable of *Explist*, called *m_bndCondExpansions*. As we will see, HelmSolve is crucial part of time discretization framework in Nektar++.

Explist is a root of an inheritance hierarchy, similar to hierarchies we have seen in other namespaces.

MultiRegions also contain class *GlobalLinSys*, describing linear system arising from space discretization, and preconditioners for the linear systems.

SolverUtils Library with routines, tools and templates for creating solvers.

The most useful classes are:

- **Driver** The most high level object, created right after the start of program. Creates and manages solver (or even multiple solvers).

[1] When we are talking about *local* coefficients in the MultiRegions setting, we mean concatenated local coefficients from all Ω_e, in the sense of definition (2.17).

- **EquationSystem** The base class for all solvers. Processes input file (stored as *m_session*), creates inner representation of mesh (*m_graph*), stores fields (variables) to be solved (*m_fields* of type *Explist*).
- **UnsteadySystem** Derived class of *EquationSystem* and base class for all solvers working with time-dependent problems. Governs time-stepping and manages time integration schemes (stored as member class *m_intScheme* of type *TimeIntegrationWrapper*) and time integration solution vector (*m_intSoln* of type *TimeIntegrationSolution*).

4.1.3 GLM for vector case and its implementation

In section 3, we seen example of scalar equation, but the actual Nektar++ implementation is capable of time-stepping equation with multiple variables. If one of the variables is a vector, then it is simple split into scalar components.

More precisely, Nektar++ implementation is best suited for system of N_v equations for the same number of variables. The system should have form

$$\frac{\partial u^k}{\partial t} = g^k(u^1, ..., u^{N_v}, t) + f^k(u^1, ..., u^{N_v}, t) \tag{4.8}$$

where $k \in \{1, ..., N_v\}$, u^k are scalar variables, f^k and g^k are differential operators. In following section, we will always use index k to denote variables, and also we reuse notation from section 3.

The algorithm is again performed in *physical space*, with values at nodal points x_i. Each variable has its own solution vector $(\boldsymbol{u}^{[n]})^k$, stages $(\boldsymbol{s}_i)^k$ and pre-stages $(\boldsymbol{\theta}_i)^k$. Coefficient matrices are the same for all of them.

During time-stepping, solution vectors are treated separately, but in parallel. For example, first step (calculating pre-stage vectors $(\boldsymbol{\theta}_i)^k$) is done for all k, then second step is done for all k and so on.

As solution vectors are treated separately, the only connection between variables is realized through f^k and g^k, which accept all variables as arguments. They are represented by vector of values at nodal points x_i.

$$(\boldsymbol{g})^k_j = \left[g^k((\boldsymbol{s}_j)^1, ..., (\boldsymbol{s}_j)^{N_v}, \tau_j)\Big|_{x_i} \right]^T \tag{4.9}$$

$$(\boldsymbol{f})^k_j = \left[f^k((\boldsymbol{s}_j)^1, ..., (\boldsymbol{s}_j)^{N_v}, \tau_j)\Big|_{x_i} \right]^T \tag{4.10}$$

where τ_j is stage time.

The implementation consists of following steps

1. **Calculating stages**

   For i from 1 to number of stages N_s compute

   (a) **Pre-stage values for all variables $(\boldsymbol{\theta}_i)^k$**

   For k from 1 to number of variables N_v compute

$$(\boldsymbol{\theta}_i)^k = \Delta t \sum_{j=1}^{i-1} a^{IM}_{ij} (\boldsymbol{g})^k_j + \Delta t \sum_{j=1}^{i-1} a^{EX}_{ij} (\boldsymbol{f})^k_j + \sum_{j=1}^{N_r} u_{ij} (\boldsymbol{u}^{[n]}_j)^k \tag{4.11}$$

For given i, all terms in the right hand side are already known, so this step is purely explicit.

(b) **Stage time τ_i**

$$\tau_i = \Delta t \sum_{j=1}^{i-1} a_{ij}^{EX} + \sum_{j=1}^{N_r} u_{ij} t_j^{[n]} \tag{4.12}$$

(c) **Stage value for all variables $(\boldsymbol{s}_i)^k$**

This step is conducted in routine *DoImplicitSolve* [2] , provided by solver.

$$\{(\boldsymbol{s}_i)^1, ..., (\boldsymbol{s}_i)^{N_v}\} = \text{DoImplicitSolve}\Big(\{(\boldsymbol{\theta}_i)^1, ..., (\boldsymbol{\theta}_i)^{N_v}\},\, \tau_i,\, \Delta t \cdot a_{ii}^{IM}\Big)$$

This routine is called only once, **all** variables are processed together. Here, terms $\{(\boldsymbol{\theta}_i)^1, ..., (\boldsymbol{\theta}_i)^{N_v}\}$ and $\{(\boldsymbol{s}_i)^1, ..., (\boldsymbol{s}_i)^{N_v}\}$ are arrays holding stage and pre-stage for all variables at once, τ_i and $\Delta t \cdot a_{ii}^{IM}$ are real numbers.

In the routine, *independent* PDEs for individual variables are usually solved. Example of such a routine is equation (3.25) in scalar case.

(d) **Implicit stage derivative for all variables $(\boldsymbol{g})_i^k$**

For k from 1 to number of variables N_v compute

$$(\boldsymbol{g})_i^k = \frac{(\boldsymbol{s}_i)^k - (\boldsymbol{\theta}_i)^k}{\Delta t \cdot a_{ii}^{IM}} \tag{4.13}$$

(e) **Explicit stage derivative for all variables $(\boldsymbol{f})_i^k$**

For k from 1 to number of variables N_v compute

$$(\boldsymbol{f})_i^k = \text{DoOdeRhs}\Big(\{(\boldsymbol{s}_i)^1, ..., (\boldsymbol{s}_i)^{N_v}\},\, \tau_i\Big)$$

This routine [3] computes explicit (often nonlinear) part of the original multi-variable PDE. It is simpler than *DoImplicitSolve*, because we do not need to solve anything, we just evaluate given term at the nodal points.

2. **Assembling new solution vector $\boldsymbol{u}^{[n+1]}$ and time vector $\boldsymbol{t}^{[n+1]}$**

(a) **Time vector $\boldsymbol{t}^{[n]}$**

For i from 1 to number of steps N_r compute

$$t_i^{[n+1]} = \Delta t \sum_{j=1}^{N_s} b_{ij}^{EX} + \sum_{j=1}^{N_r} v_{ij} t_j^{[n]} \tag{4.14}$$

[2] Actual implementation of routine DoImplicitSolve returns *void* and takes 4 arguments, instead of 3, the additional argument being $\{(\boldsymbol{s}_i)^1, ..., (\boldsymbol{s}_i)^{N_v}\}$. It is, however, passed only as an output array and its values are not used in routine. It serves only as a storage for results.

[3] Actual implementation of routine DoOdeRhs takes 3 arguments, instead of 2, the additional argument being $\{((\boldsymbol{f})_i^1, ..., (\boldsymbol{f})_i^{N_v}\}$, the reason is similar as above.

(b) **Pre-solution for all variables $\boldsymbol{w}^k$**

For k from 1 to number of variables N_v compute

$$\boldsymbol{w}^k = \Delta t \sum_{j=1}^{N_s} b_{1j}^{IM} (\boldsymbol{g})_j^k + \Delta t \sum_{j=1}^{N_s} b_{1j}^{EX} (\boldsymbol{f})_j^k + \sum_{j=1}^{N_r} v_{1j} (u_j^{[n]})^k \tag{4.15}$$

(c) **Solution for all variables $(\boldsymbol{u}_1^{[n+1]})^k$**

Solution $(u_1^{[n+1]})^k$ needs to be projected into proper space, as in the scalar case. It is done in routine *DoProjection*,

$$\{(\boldsymbol{u}_1^{[n+1]})^1, ..., (\boldsymbol{u}_1^{[n+1]})^{N_v}\} = \text{DoProjection}\Big(\{(\boldsymbol{w})^1, ..., (\boldsymbol{w})^{N_v}\},\, t_{n+1}\Big) \tag{4.16}$$

(d) **Rest of solution vector $\boldsymbol{u}_i^{[n]}$**

For k from 1 to number of variables N_v compute

For i from 1 to number of steps N_r compute

$$(u_i^{[n]})^k = \Delta t \sum_{j=1}^{N_s} b_{1j}^{IM} (\boldsymbol{g})_j^k + \Delta t \sum_{j=1}^{N_s} b_{1j}^{EX} (\boldsymbol{f})_j^k + \sum_{j=1}^{N_r} v_{1j} (u_j^{[n]})^k \tag{4.17}$$

GLM framework is part of namespace *LibUtilities*. The main class is called *TimeIntegrationScheme* and implements the above algorithm in the function *TimeIntegrate*.

4.2 Derivation of time discretization scheme of Wilcox model

Our goal is to develop efficient implementation of Wilcox model. To do this, we first need to rewrite some of the terms in the model equations.

First of them is a production term $t_{ij}\frac{\partial v_i}{\partial x_j}$ that appears in evolution equation for k (1.12). We will denote it P_k. Using (1.10)

$$\begin{aligned} P_k = t_{ij}\frac{\partial v_i}{\partial x_j} &= \left(2\nu_T S_{ij} - \frac{2}{3}k\delta_{ij}\right)\frac{\partial v_i}{\partial x_j} \\ &= \left(2\nu_T S_{ij}\frac{\partial v_i}{\partial x_j} - \frac{2}{3}k\frac{\partial v_k}{\partial x_k}\right) = \nu_T\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right)\frac{\partial v_i}{\partial x_j} \end{aligned}$$

In (1.13) the production term is very similar. We will denote it P_ω Using previous result and definition of eddy viscosity (1.14) we get

$$P_\omega = \alpha\frac{\omega}{k}t_{ij}\frac{\partial v_i}{\partial x_j} = \alpha\frac{\omega}{\tilde{\omega}}\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right)\frac{\partial v_i}{\partial x_j}$$

We denote common part of both production terms as P.

$$P = \left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right)\frac{\partial v_i}{\partial x_j} \tag{4.18}$$

Value of $\tilde{\omega}$ depends on condition (1.14) with term $S_{ij}S_{ij}$. This term can be rewritten using P. Recalling (1.16)

$$S_{ij}S_{ij} = S_{ij}\frac{\partial v_i}{\partial x_j} = \frac{1}{2}P$$

With this identity, we can simplify expression for P_ω even more. When

$$C_{lim}\sqrt{\frac{2S_{ij}S_{ij}}{\beta^\star}} > \omega \tag{4.19}$$

we have

$$P_\omega = \alpha\frac{\omega}{\tilde{\omega}}P = \alpha\frac{\omega}{C_{lim}\sqrt{\frac{2S_{ij}S_{ij}}{\beta^\star}}}P = \alpha\frac{\omega}{C_{lim}\sqrt{\frac{P}{\beta^\star}}}P = \frac{\alpha\sqrt{\beta^\star}}{C_{lim}}\omega\sqrt{P}$$

In the opposite case, when

$$C_{lim}\sqrt{\frac{2S_{ij}S_{ij}}{\beta^\star}} \leq \omega \tag{4.20}$$

we have $\tilde{\omega} = \omega$ and

$$P_\omega = \alpha\frac{\omega}{\tilde{\omega}}P = \alpha P$$

Therefore, we can write

$$P_\omega = \begin{cases} \frac{\alpha\sqrt{\beta^\star}}{C_{lim}}\omega\sqrt{P}, & C_{lim}\sqrt{\frac{P}{\beta^\star}} > \omega \\ \alpha P, & C_{lim}\sqrt{\frac{P}{\beta^\star}} \leq \omega \end{cases} \tag{4.21}$$

Similarly, we rewrite expression for P_k and ν_T using P

$$P_k = \nu_T P \tag{4.22}$$

$$\nu_T = \begin{cases} \frac{k\sqrt{\beta^\star}}{C_{lim}\sqrt{P}}, & C_{lim}\sqrt{\frac{P}{\beta^\star}} > \omega \\ \frac{k}{\omega}, & C_{lim}\sqrt{\frac{P}{\beta^\star}} \leq \omega \end{cases} \tag{4.23}$$

We note eddy viscosity (and also k and ω) should be always non-negative in the entire domain Ω.

Our model is divided into two sections. First of them takes care of incompressible Navier-Stokes equation with variable viscosity, i.e. equation (1.11). Second section computes viscosity and provides it as an input to the first section. In this case, second section is Wilcox k-ω turbulence model.

Time discretization scheme for velocity equation (1.11) is based on rotational form of velocity correction scheme (VCS). Nektar++ is already using VCS in the one of its solvers and it is built around algorithm presented in [Karniadakis et al., 1991], which rigorously analyzed by [Guermond and Shen, 2003].. Unfortunately, scheme is useful only for flows with constant viscosity ν and we need more than that.

In addition to "normal" constant viscosity ν, the model requires eddy viscosity $\nu_T(x,t)$, which varies in both space and time. Thus, we implement modified

version of VCS, suitable for variable viscosity flows, as presented in [Karamanos and Sherwin, 2000]. Variable viscosity flows together with energy equation were also studied in [Pech, 2016b] and [Pech, 2016a].

Variable viscosity $\nu+\nu_T$ appears in the laplacian term of the velocity equation.

$$\frac{\partial}{\partial x_j}\left[(\nu+\nu_T)\left(\frac{\partial v_i}{\partial x_j}+\frac{\partial v_j}{\partial x_i}\right)\right] \tag{4.24}$$

The core idea of [Karamanos and Sherwin, 2000] is to split viscosity into two parts.

One part is a kind of "average viscosity" and it is changing sufficiently slowly to be de facto constant in time. It is represented by function $c_v^{IM}(x)$. Second part describes rapid changes of viscosity, both in time and space and it is modeled by function $c_v^{EX}(x,t)$.

Function $c_v^{IM}(x)$, "average value of viscosity", should be good approximation of total variable viscosity and for the sake of numerical stability, we also require it to be non-negative.

Laplacian term is then split up, terms containing $c_v^{EX}(x,t)$ are treated explicitly, in the same step as nonlinear advection; whereas $c_v^{IM}(x)$ multiplied by $\frac{\partial v_i}{\partial x_j}$ plays the same role as viscous term in the "classical" velocity correction scheme (with constant viscosity).

To incorporate Wilcox model of turbulence (equations (1.12) and (1.13)), we use implicit-explicit (IMEX) scheme. Again, there are laplacian (viscosity-like) terms

$$\frac{\partial}{\partial x_j}\left[\left(\nu+\sigma^\star\frac{k}{\omega}\right)\left(\frac{\partial k}{\partial x_j}\right)\right] \qquad \frac{\partial}{\partial x_j}\left[\left(\nu+\sigma\frac{k}{\omega}\right)\left(\frac{\partial \omega}{\partial x_j}\right)\right]$$

and we extend previous idea of splitting also to them.

Further explanation of VCS (and IMEX scheme for k and ω) will be done later. Now we formally define all the new functions and use them to rewrite evolution equations.

Let c_v^{IM} , c_k^{IM} and c_ω^{IM} be non-negative functions of spatial variable x, but not of time t. We will call them *implicit Helmholtz coefficients* for variables v, k and ω.

Let c_v^{EX} , c_k^{EX} and c_ω^{EX} be functions of spatial variable x and time t. We will call them *explicit Helmholtz coefficients* for variables v, k and ω.

If we want to refer to all of implicit (or explicit) Helmholtz coefficients at once (or to anyone of them), we will omit the variable index and use symbol $c_\star^{IM}$ (or $c_\star^{EX}$).

For all x and t, these coefficients satisfy relations

$$c_v^{IM}(x)+c_v^{EX}(x,t)=\nu+\nu_T(x,t) \tag{4.25}$$

$$c_k^{IM}(x)+c_k^{EX}(x,t)=\nu+\sigma^\star\frac{k(x,t)}{\omega(x,t)} \tag{4.26}$$

$$c_\omega^{IM}(x)+c_\omega^{EX}(x,t)=\nu+\sigma\frac{k(x,t)}{\omega(x,t)} \tag{4.27}$$

$$c_\star^{IM}(x)>0 \tag{4.28}$$

With coefficients c_v^{IM}, c_v^{EX} and incompressibility condition, viscous term in (1.11) can be rewritten

$$\begin{aligned}\frac{\partial}{\partial x_j}\left[(\nu+\nu_T)\left(\frac{\partial v_i}{\partial x_j}+\frac{\partial v_j}{\partial x_i}\right)\right] &= \frac{\partial}{\partial x_j}\left[\left(c_v^{IM}+c_v^{EX}\right)\left(\frac{\partial v_i}{\partial x_j}+\frac{\partial v_j}{\partial x_i}\right)\right] \\ &= \frac{\partial}{\partial x_j}\left[c_v^{EX}\left(\frac{\partial v_i}{\partial x_j}+\frac{\partial v_j}{\partial x_i}\right)\right]+\frac{\partial c_v^{IM}}{\partial x_j}\frac{\partial v_j}{\partial x_i}+\frac{\partial}{\partial x_j}\left(c_v^{IM}\frac{\partial v_i}{\partial x_j}\right)\end{aligned}$$

Viscous terms in (1.12) and (1.13) can be also altered

$$\begin{aligned}\frac{\partial}{\partial x_j}\left[\left(\nu+\sigma^\star\frac{k}{\omega}\right)\left(\frac{\partial k}{\partial x_j}\right)\right] &= \frac{\partial}{\partial x_j}\left[c_k^{IM}\left(\frac{\partial k}{\partial x_j}\right)\right]+\frac{\partial}{\partial x_j}\left[c_k^{EX}\left(\frac{\partial k}{\partial x_j}\right)\right] \\ \frac{\partial}{\partial x_j}\left[\left(\nu+\sigma\frac{k}{\omega}\right)\left(\frac{\partial \omega}{\partial x_j}\right)\right] &= \frac{\partial}{\partial x_j}\left[c_\omega^{IM}\left(\frac{\partial \omega}{\partial x_j}\right)\right]+\frac{\partial}{\partial x_j}\left[c_\omega^{EX}\left(\frac{\partial \omega}{\partial x_j}\right)\right]\end{aligned}$$

Let the explicitly evaluated terms $(E_v)_i$, E_k, E_ω be defined by following relations

$$(E_v)_i = -v_j\frac{\partial v_i}{\partial x_j}+\frac{\partial}{\partial x_j}\left[c_v^{EX}\left(\frac{\partial v_i}{\partial x_j}+\frac{\partial v_j}{\partial x_i}\right)\right]+\frac{\partial c_v^{IM}}{\partial x_j}\frac{\partial v_j}{\partial x_i} \tag{4.29}$$

$$E_k = -v_j\frac{\partial k}{\partial x_j}+P_k-\beta^\star k\omega+\frac{\partial}{\partial x_j}\left[c_k^{EX}\left(\frac{\partial k}{\partial x_j}\right)\right] \tag{4.30}$$

$$E_\omega = -v_j\frac{\partial \omega}{\partial x_j}+P_\omega-\beta\omega^2+\frac{\sigma_d}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j}+\frac{\partial}{\partial x_j}\left[c_\omega^{EX}\left(\frac{\partial \omega}{\partial x_j}\right)\right] \tag{4.31}$$

All three evolution equations (1.11), (1.12), (1.13) now have more compact form

$$\frac{\partial v_i}{\partial t} = (E_v)_i-\frac{\partial p}{\partial x_i}+\frac{\partial}{\partial x_j}\left(c_v^{IM}\frac{\partial v_i}{\partial x_j}\right) \tag{4.32}$$

$$\frac{\partial k}{\partial t} = E_k+\frac{\partial}{\partial x_j}\left[c_k^{IM}\left(\frac{\partial k}{\partial x_j}\right)\right] \tag{4.33}$$

$$\frac{\partial \omega}{\partial t} = E_\omega+\frac{\partial}{\partial x_j}\left[c_\omega^{IM}\left(\frac{\partial \omega}{\partial x_j}\right)\right] \tag{4.34}$$

and are prepared for time discretization.

Let t_0 be a constant. Then we define sequence of time instants $t_n = n\Delta t + t_0$. Let f^n be value of function f at time t_n.

Let asterisk $(*)$ be time extrapolation operator of J-th order, so that

$$f^{*,n+1} = \sum_{q=0}^{J-1}\alpha_q f^{n-q} = \begin{cases} f^n, & J=1 \\ 2f^n-f^{n-1}, & J=2 \\ 3f^n-3f^{n-1}+f^{n-2}, & J=3 \end{cases} \tag{4.35}$$

where α_q are extrapolation coefficient of order J.

Let β_q be coefficients of backward differentiation formula (BDF) of order J. For example

$$\frac{\partial f^{n+1}}{\partial t} = \sum_{q=0}^{J}\beta_q f^{n+1-q} = \begin{cases} \frac{1}{\Delta t}\left(f^{n+1}-f^n\right), & J=1 \\ \frac{1}{\Delta t}\left(\frac{3}{2}f^{n+1}-2f^n+\frac{1}{2}f^{n-1}\right), & J=2 \\ \frac{1}{\Delta t}\left(\frac{11}{6}f^{n+1}-3f^n+\frac{3}{2}f^{n-1}-\frac{1}{3}f^{n-2}\right), & J=3 \end{cases} \tag{4.36}$$

Values of both α_q and β_q depend on the order J. Thus, strictly speaking, they should have another index J, but we will omit it as it will be clear from context. Generally speaking, for method to have order J, BDF and extrapolation operator should have order J.

The IMEX scheme for variables k and ω is quite simple, built around evolution equations (4.33) and (4.34). Time derivative is discretized using (4.36), backward differentiation. Difficult-to-evaluate, nonlinear terms are grouped in E_k (and E_ω) and are extrapolated from previous time levels explicitly, using (4.35). Remaining term is linear, but stiff, and should be treaded implicitly. This results in quick, but robust and stable scheme.

Following equations represent final discretization of advection-diffusion problem for k and ω.

$$\frac{1}{\Delta t}\left(\beta_0 k^{n+1} + \sum_{q=1}^{J} \beta_q k^{n+1-q}\right) = E_k^{*,n+1} + \frac{\partial}{\partial x_j}\left[c_k^{IM}\left(\frac{\partial k^{n+1}}{\partial x_j}\right)\right] \tag{4.37}$$

$$\frac{1}{\Delta t}\left(\beta_0 \omega^{n+1} + \sum_{q=1}^{J} \beta_q \omega^{n+1-q}\right) = E_\omega^{*,n+1} + \frac{\partial}{\partial x_j}\left[c_\omega^{IM}\left(\frac{\partial \omega^{n+1}}{\partial x_j}\right)\right] \tag{4.38}$$

Velocity correction scheme is more sophisticated. Instead of directly solving Navier-Stokes equations, with velocity and pressure coupled together, it tries to separate them and solve for each of them individually.

To this end, VCS introduces intermediate velocity field $\hat{v}$ and divides each time step into two sub-steps. This approach can be thought as a kind of predictor-corrector method.

At time t_{n+1}, in the first sub-step, we are calculating the value of field $\hat{v}^{n+1}$, that serves as a prediction for actual velocity field v^{n+1}. We suppose $\hat{v}^{n+1}$ is incompressible $(\nabla \cdot \hat{v}^{n+1}) = 0$ and satisfies following equation

$$\begin{aligned}\frac{1}{\Delta t}\left(\beta_0 \hat{v}_i^{n+1} + \sum_{q=1}^{J} \beta_q v_i^{n+1-q}\right) = ((E_v)_i)^{*,n+1} - \frac{\partial p^{n+1}}{\partial x_i} \\ + \frac{\partial c_v^{IM}}{\partial x_j}\left(\frac{\partial v_i}{\partial x_j}\right)^{*,n+1} - c_v^{IM}\left(\nabla \times \nabla \times \boldsymbol{v}\right)_i^{*,n+1}\end{aligned} \tag{4.39}$$

which is derived from (4.32). Last two terms are obtained using common Laplace operator identity

$$\Delta \boldsymbol{v} = \nabla\left(\nabla \cdot \boldsymbol{v}\right) - \left(\nabla \times \nabla \times \boldsymbol{v}\right) \tag{4.40}$$

which yields (as $\nabla \cdot \boldsymbol{v} = 0$)

$$\frac{\partial}{\partial x_j}\left(c_v^{IM}\frac{\partial v_i}{\partial x_j}\right) = \frac{\partial c_v^{IM}}{\partial x_j}\left(\frac{\partial v_i}{\partial x_j}\right) - c_v^{IM}\left(\frac{\partial v_i}{\partial x_j \partial x_j}\right) \tag{4.41}$$

$$= \frac{\partial c_v^{IM}}{\partial x_j}\left(\frac{\partial v_i}{\partial x_j}\right) - c_v^{IM}\left(\nabla \times \nabla \times \boldsymbol{v}\right)_i \tag{4.42}$$

There are two unknown terms in the first sub-step (4.39), pressure p^{n+1} and field $\hat{v}^{n+1}$ itself. To take advantage of incompressibility of $\hat{v}^{n+1}$ we can apply

divergence on (4.39) and rearrange it to get Poisson equation for pressure

$$\frac{\partial p^{n+1}}{\partial x_i \partial x_i} = \frac{\partial}{\partial x_i}\left[-\frac{1}{\Delta t}\sum_{q=1}^{J}\beta_q v_i^{n+1-q} + ((E_v)_i)^{*,n+1} + \frac{\partial c_v^{IM}}{\partial x_j}\left(\frac{\partial v_i}{\partial x_j}\right)^{*,n+1} - c_v^{IM}\left(\nabla \times \nabla \times \boldsymbol{v}\right)_i^{*,n+1}\right] \tag{4.43}$$

where all terms on the right hand side are known.

Pressure equation should be supplemented with appropriate boundary conditions on $\partial\Omega$. The most straightforward way do get them is to multiply (4.39) with normal $\boldsymbol{n}$ to boundary $\partial\Omega$. This yields Neumann boundary conditions for pressure p

$$n_i\frac{\partial p^{n+1}}{\partial x_i} = n_i\left[\frac{1}{\Delta t}\left(-\beta_0\hat{v}_i^{n+1} - \sum_{q=1}^{J}\beta_q v_i^{n+1-q}\right) + ((E_v)_i)^{*,n+1} + \frac{\partial c_v^{IM}}{\partial x_j}\left(\frac{\partial v_i}{\partial x_j}\right)^{*,n+1} - c_v^{IM}\left(\nabla \times \nabla \times \boldsymbol{v}\right)_i^{*,n+1}\right] \tag{4.44}$$

It it sometimes called *high order pressure boundary condition* or *HOPBC* in Nektar++ source code.

It may seem that we do not know value of term $\hat{v}_i^{n+1}$ at the time we are solving for pressure p^{n+1}, but we *have* $\hat{v}_i^{n+1}$ on *boundary* from Dirichlet conditions for $\boldsymbol{v}$.

After solving (4.43), we got value of p^{n+1} and can compute $\hat{v}^{n+1}$ without solving anything else (just by using explicit formula (4.39).

Second sub-step is viscous correction, where we correct value of $\hat{v}^{n+1}$ to get velocity v^{n+1}. We do *not* suppose that $\nabla \cdot v^{n+1} = 0$. For each component of v_i^{n+1} we have

$$\frac{1}{\Delta t}\left(\beta_0 v_i^{n+1} - \beta_0\hat{v}_i^{n+1}\right) = \frac{\partial}{\partial x_j}\left[c_v^{IM}\frac{\partial v_i^{n+1}}{\partial x_j}\right] - \frac{\partial c_v^{IM}}{\partial x_j}\left(\frac{\partial v_i}{\partial x_j}\right)^{*,n+1} + c_v^{IM}\left(\nabla \times \nabla \times v\right)_i^{*,n+1} \tag{4.45}$$

With a little bit of rearranging we get scalar PDE

$$v_i^{n+1} - \frac{\Delta t}{\beta_0}\frac{\partial}{\partial x_j}\left[c_v^{IM}\frac{\partial v_i^{n+1}}{\partial x_j}\right] = f_i \tag{4.46}$$

where f_i stands for all remaining (already known) terms.

This form of the equation looks similar to scalar Helmholtz equation. Only differences being the sign and the variable coefficients c_v^{IM} of the laplacian term.

One can also note IMEX equations for k and ω could be manipulated to have the same form as (4.46). Fortunately, Nektar++ framework enables us to solve exactly this type of equation easily, in the routine *HelmSolve*, mentioned in section 4.1.2 (routine 4.7).

In a nutshell, during each time step of velocity correction scheme, we need to solve $d+1$ scalar PDEs (d is number of velocity components) - one Poisson

equation for pressure and d Helmholtz like equation for velocity components. All other operations are explicit and we do not need to solve anything.

We derived time discretization schemes for both velocity (Navier-Stokes) and k-ω systems. But there are several questions remaining. How will we exactly calculate the value of $c_\star^{IM}$ and $c_\star^{EX}$ for variables $\boldsymbol{v}$, k and ω? How are the two systems connected to each other?

Let us concentrate on the first question. For the purposes of velocity correction scheme, value of c_v^{IM} is constant in time. Thus, we determine explicit coefficient c_v^{EX} at time t_n using (4.25)

$$(c_v^{EX})^n = \nu + \nu_T^n - c_v^{IM} \tag{4.47}$$

We also require c_v^{IM} to approximate total viscosity

$$c_v^{IM} \approx \nu + \nu_T^n \tag{4.48}$$

for all n, because of the numerical stability of the scheme.

Ratio between c_v^{IM} and c_v^{EX} relates to ratio between explicit and implicit part of viscous term. If the explicit part is too large, scheme becomes unstable.

To satisfy (4.48) at time $t = 0$, we set

$$c_v^{IM} = \nu + \nu_T^0 \tag{4.49}$$

where ν_T^0 is initial value of eddy viscosity. As c_v^{IM} is not a function of time, condition (4.48) holds only if ν_T^n does not differ too much from the initial value ν_T^0, for all $x \in \Omega$.

In general case, such "nice" behavior of ν_T^n cannot be expected, so when

$$R_v^{min} < \frac{\nu + \nu_T^n}{c_v^{IM}} < R_v^{max} \tag{4.50}$$

it is necessary to *reset* value c_v^{IM}

$$c_v^{IM} = \nu + \nu_T^n \tag{4.51}$$

Choice of number R_v^{min} and R_v^{max} influences stability of the scheme. The closer they are to 1, the more stable is the scheme.

The exact same line of though is used for c_k^{IM}, c_k^{EX} and c_ω^{IM}, c_ω^{EX}.

Turbulence (k-ω) model influences velocity system only through coefficients c_v^{IM} and c_v^{EX}, that are determined by eddy viscosity ν_T. On the other hand, velocity system influence turbulence equations through the value of velocity $\boldsymbol{v}$.

When we look at the scheme carefully, we see both these influences are worked out *explicitly*. At time t_{n+1}, in the velocity system we need coefficients c_v^{IM} from time levels n or lower and in turbulence system, we need velocity $\boldsymbol{v}$ from time level n or lower. We effectively decoupled those two systems.

That brings us one big advantage - modularity. We can easily implement other turbulence systems that use eddy viscosity.

4.3 Time discretization scheme summary

Scheme uses backward differentiation formula for time derivative and extrapolation for nonlinear terms, both of order J. This is the only free parameter of the model, with values ranging from 1 to 3. All other parameters are given in section 1.1.

At the beginning of each time step, we have values of velocities v_i^n, turbulent kinetic energy k^n and specific dissipation ω^n at time t_n. In addition, we know implicit Helmholtz coefficients c^{IM} for all variables.

To get values at time t_{n+1} we stick to the following

1. **Compute auxiliary values**

   Calculate term P^n from velocities v_i^n according to (4.18), eddy viscosity v_T^n according to (4.23) and production terms P_k^n and P_ω^n according to (4.22) and (4.21).

2. **Check the ratio between implicit and total coefficients**

   The scheme can be unstable, if for any of the variables $\boldsymbol{v}$, k, ω the ratio between $c_\star^{IM}$ and total viscosity differ from 1 significantly.

   To keep the difference under control, we require

   $$\begin{aligned} R_v^{min} &< \frac{\nu + \nu_T^n}{c_v^{IM}} < R_v^{max} \\ R_k^{min} &< \frac{\nu + \sigma^\star \frac{k^n}{\omega^n}}{c_k^{IM}} < R_k^{max} \\ R_\omega^{min} &< \frac{\nu + \sigma \frac{k^n}{\omega^n}}{c_\omega^{IM}} < R_\omega^{max} \end{aligned}$$

   for all x in Ω (we recall $c_\star^{IM}$ is function of x). Constants R depend on the nature of the problem, default value is $\frac{1}{2}$ for R^{min} and 2 for R^{max} for all variables.

   If the above inequality is broken for given variable, we reset the value of corresponding $c_\star^{IM}$ using one of the following formulas.

   $$\begin{aligned} c_v^{IM} &= \nu + \nu_T^n \\ c_k^{IM} &= \nu + \sigma^\star \frac{k^n}{\omega^n} \\ c_\omega^{IM} &= \nu + \sigma \frac{k^n}{\omega^n} \end{aligned}$$

   If the inequalities are satisfied, we keep the values of $c_\star^{IM}$ from previous time step.

3. **Compute explicit Helmholtz coefficients**

After we have determined implicit Helmholtz coefficients, we calculate explicit ones

$$
\begin{aligned}
(c_v^{EX})^n &= \nu + \nu_T^n - c_v^{IM} \\
(c_k^{EX})^n &= \nu + \sigma^\star \frac{k^n}{\omega^n} - c_k^{IM} \\
(c_\omega^{EX})^n &= \nu + \sigma \frac{k^n}{\omega^n} - c_\omega^{IM}
\end{aligned}
$$

4. **Solve Poisson equation for pressure p^{n+1}**

   In the previous section, this step was called first sub-step of velocity correction scheme. We derived there pressure equation (4.43) with appropriate boundary conditions.

   Because of the implementation, we rewrite the right hand side to isolate term $\frac{\beta_0}{\Delta t}$ and get

$$
\begin{aligned}
\frac{\partial p^{n+1}}{\partial x_i \partial x_i} = \frac{\beta_0}{\Delta t} \frac{\partial}{\partial x_i} \Bigg[& - \sum_{q=1}^{J} \frac{\beta_q}{\beta_0} v_i^{n+1-q} + \frac{\Delta t}{\beta_0} \left((E_v)_i \right)^{*,n+1} \\
& + \frac{\Delta t}{\beta_0} \left(\frac{\partial c_v^{IM}}{\partial x_j} \left(\frac{\partial v_i}{\partial x_j} \right)^{*,n+1} - c_v^{IM} \left(\nabla \times \nabla \times \boldsymbol{v} \right)_i^{*,n+1} \right) \Bigg]
\end{aligned} \tag{4.52}
$$

   We used definition (4.35) of extrapolation operator $(\cdot)^{\star,n+1}$, backward differentiation coefficients β_q (see (4.36)) and term $(E_v)_i$ according to (4.29).

   Poisson equation is supplemented by Neumann boundary condition, again with isolated $\frac{\beta_0}{\Delta t}$.

$$
\begin{aligned}
n_i \frac{\partial p^{n+1}}{\partial x_i} = \frac{\beta_0}{\Delta t} n_i \Bigg[& - \hat{v}_i^{n+1} - \sum_{q=1}^{J} \frac{\beta_q}{\beta_0} v_i^{n+1-q} + \frac{\Delta t}{\beta_0} \left((E_v)_i \right)^{*,n+1} \\
& + \frac{\Delta t}{\beta_0} \left(\frac{\partial c_v^{IM}}{\partial x_j} \left(\frac{\partial v_i}{\partial x_j} \right)^{*,n+1} - c_v^{IM} \left(\nabla \times \nabla \times \boldsymbol{v} \right)_i^{*,n+1} \right) \Bigg]
\end{aligned} \tag{4.53}
$$

5. **Solve Helmholtz-like equations for velocities $\boldsymbol{v}^{n+1}$, turbulent kinetic energy k^{n+1} and specific dissipation ω^{n+1}**

   We merge second sub-step of velocity correction scheme (4.45) with IMEX scheme (4.37) and (4.38) for turbulent variables k and ω.

   Also, in (4.45), we substitute for intermediate field $\hat{v}_i^{n+1}$ from (4.39) and get rid of the intermediate field completely.

   Again, for the sake of implementation, we multiply these equations with $\frac{\Delta t}{\beta_0}$

to obtain

$$v_i^{n+1} - \frac{\Delta t}{\beta_0}\frac{\partial}{\partial x_j}\left[c_v^{IM}\frac{\partial v_i^{n+1}}{\partial x_j}\right] = -\sum_{q=1}^{J}\frac{\beta_q}{\beta_0}v_i^{n+1-q} - \frac{\Delta t}{\beta_0}\frac{\partial p^{n+1}}{\partial x_i} + \frac{\Delta t}{\beta_0}((E_v)_i)^{*,n+1} \tag{4.54}$$

$$k^{n+1} - \frac{\Delta t}{\beta_0}\frac{\partial}{\partial x_j}\left[c_k^{IM}\left(\frac{\partial k^{n+1}}{\partial x_j}\right)\right] = \frac{\Delta t}{\beta_0}(E_k)^{*,n+1} - \sum_{q=1}^{J}\frac{\beta_q}{\beta_0}k^{n+1-q} \tag{4.55}$$

$$\omega^{n+1} - \frac{\Delta t}{\beta_0}\frac{\partial}{\partial x_j}\left[c_\omega^{IM}\left(\frac{\partial \omega^{n+1}}{\partial x_j}\right)\right] = \frac{\Delta t}{\beta_0}(E_\omega)^{*,n+1} - \sum_{q=1}^{J}\frac{\beta_q}{\beta_0}\omega^{n+1-q} \tag{4.56}$$

4.4 Implementation

Implementation in Nektar++ is based on class *UnsteadySystem* and makes use of GLM framework (in class *TimeIntegrationScheme*), summarized in section 4.1.3.

GLM is best suited for multi-variable equations that satisfy general form

$$\frac{\partial u^k}{\partial t} = g^k(u^1, ..., u^{N_v}, t) + f^k(u^1, ..., u^{N_v}, t) \tag{4.57}$$

where $u^1, ..., u^n$ are unknown scalar fields defined in domain Ω, f is function we evaluate explicitly at each time-step, g is function solved implicitly and $k \in \{1, ..., n\}$.

If the system of PDEs fits into the general form, implementing time discretization method is easy. All you need to do is to provide class *TimeIntegrationScheme* with 3 routines - *DoImplicitSolve*, *DoOdeRhs* and *DoProjection* and find right coefficient matrices $A_{IM}, A_{EX}, B_{IM}, B_{EX}, U$ and V for the given scheme. In addition, user can change the method by changing only the matrices and *not* the routines. It will still yield consistent results.

Velocity correction scheme with turbulence model, however, have more complicated structure than (4.57). The problem stems from Poisson equation for pressure p^{n+1} (4.52), that is somehow outside of scope of GLM algorithm.

We can try to incorporate pressure equation into g_k - routine *DoImplicitSolve*, or into f_k - routine *DoOdeRhs*, but we will soon realize it is hopeless. Terms contributing to pressure equation does not match to arguments of g_k or f_k (or arguments of corresponding routines).

Thus, for our model, routines cannot be used in the standard, generic form, suitable for different coefficient matrices. They will be "method specific", consistent just with the matrices belonging to our scheme. User will only be able to choose the order of precision J.

Still, the goal is the same - define those routines and find coefficient matrices.

4.4.1 Formulation of solution vector

In GLM framework, time discretization scheme for Wilcox model, defined in section 4.3, can be characterized in following way [4].

Our scheme has one stage ($N_s = 1$), where the stage is equal to new solution, and multiple steps ($N_r = 2J$).

[4] During description of the scheme, we will use notation from section 3

It consists of one vector and two scalar variables - velocity $\boldsymbol{v}$, turbulent kinetic energy k, and specific dissipation rate ω. Vector $\boldsymbol{v}$ is decomposed into elements v_i. Total number of scalar variables N_v is $d+2$, where d is number of components of $\boldsymbol{v}$.

Each scalar variable is spatially discretized using common space V_δ, on the same set of nodal points x_i. Each k-th variable has its own solution vector $(u^{[n]})^k$, which are stored in 3-dimensional structure $\boldsymbol{\Psi}$, in "vector of solution vectors". For example, when $d = 2$ we have

$$\boldsymbol{\Psi} = \begin{pmatrix} (\boldsymbol{u}^{[n]})^1 \\ (\boldsymbol{u}^{[n]})^2 \\ (\boldsymbol{u}^{[n]})^3 \\ (\boldsymbol{u}^{[n]})^4 \end{pmatrix} = \begin{pmatrix} \boldsymbol{v}_1^{[n]} \\ \boldsymbol{v}_2^{[n]} \\ \boldsymbol{k}^{[n]} \\ \boldsymbol{\omega}^{[n]} \end{pmatrix} = \begin{pmatrix} \left[\boldsymbol{v}_{1,1}^{[n]} \middle| \boldsymbol{v}_{1,2}^{[n]} \middle| \ ... \ \middle| \boldsymbol{v}_{1,2J}^{[n]} \right] \\ \left[\boldsymbol{v}_{2,1}^{[n]} \middle| \boldsymbol{v}_{2,2}^{[n]} \middle| \ ... \ \middle| \boldsymbol{v}_{2,2J}^{[n]} \right] \\ \left[\boldsymbol{k}_1^{[n]} \middle| \boldsymbol{k}_2^{[n]} \middle| \ ... \ \middle| \boldsymbol{k}_{2J}^{[n]} \right] \\ \left[\boldsymbol{\omega}_1^{[n]} \middle| \boldsymbol{\omega}_2^{[n]} \middle| \ ... \ \middle| \boldsymbol{\omega}_{2J}^{[n]} \right] \end{pmatrix} \tag{4.58}$$

where we used definition of solution vector (3.19). We recall index n relates to time level t_n and each component of solution vector (e.g. $\boldsymbol{\omega}_2^{[n]}$ or $\boldsymbol{v}_{1,2J}^{[n]}$), is itself a vector of values at x_i.

For all variables, solution vector $(u^{[n]})^k$ have the same structure

$$(\boldsymbol{u}^{[n]})^k = \begin{pmatrix} (\boldsymbol{u}^{[n]})_1^k \\ (\boldsymbol{u}^{[n]})_2^k \\ \vdots \\ (\boldsymbol{u}^{[n]})_{2J}^k \end{pmatrix} = \begin{pmatrix} (\boldsymbol{u}^{n})^k \\ \vdots \\ (\boldsymbol{u}^{n-J+1})^k \\ \Delta t(\boldsymbol{f}^{n})^k \\ \vdots \\ \Delta t(\boldsymbol{f}^{n-J+1})^k \end{pmatrix} \tag{4.59}$$

Time vector is defined as

$$\boldsymbol{t}^{[n]} = [\, t_n, ..., t_{n-J+1}, \Delta t, ..., \Delta t \,]^T \tag{4.60}$$

Term $(\boldsymbol{f}^n)^k$ is a vector of values at nodal points, in the sense of definition (4.10), and it is defined as

$$(\boldsymbol{f}^n)^k = (\boldsymbol{f}^n)^{k^\star} \begin{cases} ((\boldsymbol{E}_v)_{k^\star})^n, & 1 \leq k^\star \leq d \\ (\boldsymbol{E}_k)^n, & k^\star = d+1 \\ (\boldsymbol{E}_\omega)^n, & k^\star = d+2 \end{cases} \tag{4.61}$$

The k in the sense of "index of variable" is substituted by $k^\star$ to avoid confusion with k in the sense of "turbulent kinetic energy".

One may wonder, where is pressure p? It has a special position. It is not independent variable, but just an auxiliary function for computing velocity $\boldsymbol{v}$. More importantly, we do not need its *time derivative*, so we do not need any time stepping method for it.

Thus, for the purpose of *time* discretization, pressure neither count as one of the variables of the scheme nor it has its own solution vector.

Nevertheless, for the purpose of *spatial* discretization p is a standard field, with its own vector of physical values and coefficients. It is not necessary to use

the same approximation space V_δ as for v_i, k and ω, but it is mandatory to use the same set of nodal points. Pressure has its own object *Explist*, boundary condition etc.

With this object, we are able to solve Poisson equation for pressure (4.52) and use the results during routines.

4.4.2 Coefficient matrices

Coefficient matrices are derived from Helmholtz-like equations (4.54), (4.55) and (4.56), in the last step of the scheme. If we omit the pressure term from velocity equation, then all of them has the same general form, very similar to (3.25). We can use this analogy to identify coefficients.

First of all, in the all three equations, there is always one implicit term with the coefficient $\frac{\Delta t}{\beta_0}$. As we have only one stage, coefficient matrix $\boldsymbol{A}^{IM} = a_{11} = \frac{1}{\beta_0}$.

We define pre-stage $(\boldsymbol{\theta})^k$ as the right hand side of the equations (without pressure)

$$(\boldsymbol{\theta})^k = \begin{cases} -\sum_{q=1}^{J} \frac{\beta_q}{\beta_0} \boldsymbol{v}_i^{n+1-q} + \frac{\Delta t}{\beta_0} ((\boldsymbol{E}_v)_i)^{*,n+1} \\ -\sum_{q=1}^{J} \frac{\beta_q}{\beta_0} \boldsymbol{k}^{n+1-q} + \frac{\Delta t}{\beta_0} (\boldsymbol{E}_k)^{*,n+1} \\ -\sum_{q=1}^{J} \frac{\beta_q}{\beta_0} \boldsymbol{\omega}^{n+1-q} + \frac{\Delta t}{\beta_0} (\boldsymbol{E}_\omega)^{*,n+1} \end{cases} \tag{4.62}$$

Then using definition of extrapolation operator $(\star)$ (4.35), term $(\boldsymbol{f}^n)^k$ (4.61) and our solution vector $(\boldsymbol{u}^{[n]})^k$ (4.59) we get

$$\begin{aligned} (\boldsymbol{\theta})^k &= -\sum_{q=1}^{J} \frac{\beta_q}{\beta_0} (\boldsymbol{u}^{n+1-q})^k + \sum_{q=0}^{J-1} \Delta t \frac{\alpha_q}{\beta_0} (\boldsymbol{f}^{n+1-q})^k \\ &= \sum_{q=1}^{J} -\frac{\beta_q}{\beta_0} (\boldsymbol{u}^{[n]})_q^k + \sum_{q=0}^{J-1} \Delta t \frac{\alpha_q}{\beta_0} (\boldsymbol{u}^{[n]})_{J+q+1}^k \end{aligned} \tag{4.63}$$

Now we compare this last result (4.63) with general definition of pre-stage (4.11) to determine coefficient matrix $\boldsymbol{U} = u_{ij}$.

$$(\boldsymbol{\theta})^k = \sum_{j=0}^{2J} u_{1j} (\boldsymbol{u}^{[n]})_j^k = \sum_{q=1}^{J} -\frac{\beta_q}{\beta_0} (\boldsymbol{u}^{[n]})_q^k + \sum_{q=0}^{J-1} \Delta t \frac{\alpha_0}{\beta_0} (\boldsymbol{u}^{[n]})_{J+q+1}^k \tag{4.64}$$

it is also evident, that $\boldsymbol{A}^{EX} = 0$.

We successfully determined $\boldsymbol{A}^{IM}$, $\boldsymbol{A}^{EX}$ and $\boldsymbol{U}$ (all these matrix have only one row), so stage value $(\boldsymbol{s})^k$ can be computed.

According to the scheme, stage value $(\boldsymbol{s})^k$ is equal to solution at new time level $(\boldsymbol{u}^{[n+1]})_1^k$. Thus, first row of $\boldsymbol{B}^{IM}$, $\boldsymbol{B}^{EX}$ and $\boldsymbol{V}$ is equal to $\boldsymbol{A}^{IM}$, $\boldsymbol{A}^{EX}$ and $\boldsymbol{U}$.

The rest of the matrices $\boldsymbol{B}^{IM}$, $\boldsymbol{B}^{EX}$ and $\boldsymbol{V}$ is chosen to "roll-over" the solution vector into now time level $n+1$.

Coefficient matrices can be compactly written in the from

$$\left[\begin{array}{c:c|c} A^{IM} & A^{EX} & U \\ \hline B^{IM} & B^{EX} & V \end{array}\right] \tag{4.65}$$

For $J = 1$ we have

$$\left[\begin{array}{c:c|cc} 1 & 0 & 1 & 1 \\ \hline 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{array}\right] \tag{4.66}$$

For $J = 2$ we have

$$\left[\begin{array}{c:c|cccc} \frac{2}{3} & 0 & \frac{4}{3} & -\frac{1}{3} & \frac{4}{3} & -\frac{2}{3} \\ \hline \frac{2}{3} & 0 & \frac{4}{3} & -\frac{1}{3} & \frac{4}{3} & -\frac{2}{3} \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right] \tag{4.67}$$

For $J = 3$ we have

$$\left[\begin{array}{c:c|cccccc} \frac{6}{11} & 0 & \frac{18}{11} & -\frac{9}{11} & \frac{2}{11} & \frac{18}{11} & -\frac{18}{11} & \frac{6}{11} \\ \hline \frac{6}{11} & 0 & \frac{18}{11} & -\frac{9}{11} & \frac{2}{11} & \frac{18}{11} & -\frac{18}{11} & \frac{6}{11} \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right] \tag{4.68}$$

4.4.3 DoOdeRhs

In this routine, terms $(\boldsymbol{f}^n)^k$ are computed, according to definitions (4.29), (4.30), (4.31).

Also, part of right hand side of Poisson equation for pressure (4.52) and its boundary conditions are prepared, in the object *StandardExtrapolation*.

Main part, present both in boundary condition and Poisson equation, is

$$p_f = \frac{\partial c_v^{IM}}{\partial x_j}\left(\frac{\partial v_i}{\partial x_j}\right)^{*,n+1} - c_v^{IM}\left(\nabla \times \nabla \times \boldsymbol{v}\right)_i^{*,n+1} \tag{4.69}$$

and its values $\boldsymbol{p}_f$ at nodal points are stored in object *m_varVisTermsPlusCurlCurl*.

The extrapolation of this part is done in *StandardExtrapolation*, according to order of precision J.

4.4.4 DoImplicitSolve

Firstly, right hand side of pressure equation is determined, using $(\boldsymbol{\theta})^k$ and $\boldsymbol{p}_f$, according to (4.52).

Then the pressure equation is solved and pressure term is added to appropriate $(\boldsymbol{\theta})^k$.

Lastly, all Helmholtz-like equations (4.54), (4.55) and (4.56) for velocities, k and ω are solved. They all have the same form

$$(\boldsymbol{s})^k - (\Delta t \cdot a_{11})\frac{\partial}{\partial x_j}\left[\boldsymbol{c}_k^{IM}\frac{\partial(\boldsymbol{s})^k}{\partial x_j}\right] = (\boldsymbol{\theta})^k \tag{4.70}$$

where k indexes variables. Results are stage values $(\boldsymbol{s})^k$.

4.4.5 DoProjection

The solution $(\boldsymbol{u}^{[n]})_1^k$ is the same, as the stage values $(\boldsymbol{s})^k$. Projection step is not needed (stage values are already in the proper space V_δ).

4.5 Setting Nektar++ input file

Nektar++ uses input file in *XML* format to specify parameters of the solver. In this section, we show how to set up the input file to use implemented turbulence model. For complete information on *XML* input file, we refer reader to [Nektar++ User Guide, 2017].

In the section `<EXPANSIONS>`, user needs to set `FIELDS` (for 2D case) as

```
FIELDS="u,v,k,omega,p"
```

where `u` and `v` denotes velocity fields, `k` and `omega` fields of turbulent variable, `p` is pressure. It is also important to write them in this order.

Section `<SOLVERINFO>` should contain following items

```
<I PROPERTY="SolverType" VALUE="VarVisSimple" />
<I PROPERTY="EQTYPE" VALUE="UnsteadyNavierStokes" />
<I PROPERTY="AdvectionForm" VALUE="Convective" />
<I PROPERTY="Projection" VALUE="Galerkin" />
<I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder2" />
```

where the values of `EQTYPE`, `AdvectionForm`, `Projection` are tried and tested. Other values of these properties can cause problems.

Value `VarVisSimple` switches on the turbulent model. Time integration method can be `IMEXOrderX`, where `X` is 1,2 or 3.

Section `<VARIABLES>` must correspond with `FIELDS`

```
<VARIABLES>
    <V ID="0"> u </V>
    <V ID="1"> v </V>
    <V ID="2"> k </V>
    <V ID="3"> omega </V>
    <V ID="4"> p </V>
</VARIABLES>
```

In the first chapter, in section 1.2, we have shown Wilcox model can be reduced to Kolmogorov by changing values of model constants.

Values of constants are set in the section `<PARAMETERS>`.

```
<PARAMETERS>
    .
    <P> Alpha     = 13.0/25.0   </P>
    <P> Beta   = 0.0708   </P>
    <P> Beta_Star   = 9.0/100.0   </P>
    .
</PARAMETERS>
```

When the a constant is not listed, solver sets the default value automatically. The default values correspond to Wilcox model and reader can find them in model summary 1.1. Names of constants to be used in Nektar++ *xml* input file are in the table 4.1.

Table 4.1: Names of Wilcox model constants in Nektar++

Name of constant in the Nektar++	Name of constant in the thesis
`Kinvis`	ν
`Alpha`	α
`Beta`	β
`Beta_Star`	$\beta^\star$
`Beta_0`	β_0
`Sigma`	σ
`Sigma_Star`	$\sigma^\star$
`Sigma_V`	σ_v
`CD_const`	σ_{do}
`C_lim`	C_{lim}
`F_Beta`	f_β

5. Testing and numerical experiment

All tests are conducted for Wilcox model. The reason is that Wilcox model more complex and if it passes the test, it is quite certain Kolmogorov's model will pass it too; implementation in Nektar++ is the same, only the constants would be different.

Numerical experiment - channel flow in 2D is also conducted only for Wilcox model, because it can be integrated in the entire domain up to the walls. On the wall, we know boundary conditions for turbulent variables.

Kolmogorov's model would require either wall functions or different domain Ω (such that it would not touch solid walls), because it could not be integrated up to the walls.

5.1 Source terms

In order to test time discretization scheme and the entire implementation, we need to prepare some test cases.

Ideally, we would take analytical solution of Wilcox model in a domain Ω and compare it with the numerical solution, produced by our Nektar++ implementation. Furthermore, the analytical solution should not be trivial. In fact, it should be sophisticated, such that all terms in (1.11),(1.12) and (1.13) are non-zero. Only then we can be sure, that there are no flaws in the implementation of all terms.

Unfortunately, to the best of our knowledge, we could not find any non-trivial analytical solution for the model. The alternative is to use *source terms*.

Source terms $S_k(\boldsymbol{x},t)$ and $S_\omega(\boldsymbol{x},t)$ are scalar functions defined as

$$S_k(\boldsymbol{x},t) = \frac{\partial k}{\partial t} + v_j\frac{\partial k}{\partial x_j} - \tau_{ij}\frac{\partial v_i}{\partial x_j} + \beta^\star k\omega - \frac{\partial}{\partial x_j}\left[\left(\nu + \sigma^\star\frac{k}{\omega}\right)\left(\frac{\partial k}{\partial x_j}\right)\right] \tag{5.1}$$

$$S_\omega(\boldsymbol{x},t) = \frac{\partial \omega}{\partial t} + v_j\frac{\partial \omega}{\partial x_j} - \alpha\frac{\omega}{k}\tau_{ij}\frac{\partial v_i}{\partial x_j} + \beta\omega^2 - \frac{\sigma_d}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j} - \frac{\partial}{\partial x_j}\left[\left(\nu + \sigma\frac{k}{\omega}\right)\left(\frac{\partial \omega}{\partial x_j}\right)\right] \tag{5.2}$$

Their purpose is to balance out equations (1.12) and (1.13). If we put S_k and S_ω on the right hand side of these equations, all terms would cancel out.

With source terms, we do not need analytical solution of turbulence equations to test implementation. We just choose arbitrary functions $\boldsymbol{v}_0$, k_0 and ω_0, compute S_k and S_ω and put them among other explicit terms.

$$E_k^{src} = -v_j\frac{\partial k}{\partial x_j} + P_k - \beta^\star k\omega + \frac{\partial}{\partial x_j}\left[c_k^{EX}\left(\frac{\partial k}{\partial x_j}\right)\right] + S_k \tag{5.3}$$

$$E_\omega^{src} = -v_j\frac{\partial \omega}{\partial x_j} + P_\omega - \beta\omega^2 + \frac{\sigma_d}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j} + \frac{\partial}{\partial x_j}\left[c_\omega^{EX}\left(\frac{\partial \omega}{\partial x_j}\right)\right] + S_\omega \tag{5.4}$$

in accordance with definition (4.30) and (4.31). Now, instead of E_k and E_ω, we use E_k^{src}, E_ω^{src} everywhere in program. We will called altered model **Wilcox with source terms**.

It is clear the solution of turbulence equations with source terms is k_0 and ω_0. To test implementation, we compare numerical solution to these functions.

Advantage of source terms is that we can construct analytical expression before running the simulation. Although it is tedious and error prone to calculate them by hand, we can take advantage of computer algebra systems such as *Mathematica* or *MATLAB*. Nektar++ supports evaluating arbitrary analytical expressions during simulation, which makes implementation of source terms easier.

Variables k_0 and ω_0 can be constructed in any way we want, which gives us complete control of value of turbulent viscosity ν_T.

We do not need source terms for velocity equation. The reason is the additional degree of freedom represented by ν_T. Using ν_T, we can easily construct analytical solutions of Navier-Stokes equations [1].

5.2 Spectral (exponential) convergence test

Distinguishing feature of spectral elements method is the ability to reach exponential convergence with the respect to polynomial order n. Our model also should have this capability.

Analytical solution of Wilcox model with source terms S_k and S_ω (defined by (5.1) and (5.2)) is

$$\begin{aligned}
\boldsymbol{v} &= [e^y, e^x]^T \\
p &= -e^{x+y} - e^{y-x} - e^{x-y} \\
k &= 200 * (e^{-x} + e^{-y}) \\
\omega &= 200 \\
\nu &= 0 \\
\nu_T &= e^{-x} + e^{-y}
\end{aligned}$$

All functions are defined in the domain $\Omega = [-1, 1] \times [-1, 1]$.

Initial conditions are equal to the solution. Model is supplemented by Dirichlet boundary conditions for $\boldsymbol{v}$, k and ω, that equal solution evaluated on $\partial\Omega$.

For variable p we use Neumann high order pressure boundary conditions (4.53) on $-1 \times [-1, 1]$ and $1 \times [-1, 1]$ and Dirichlet condition on the rest.

The solution was carefully constructed, so that all terms in velocity and pressure equations (also in high order pressure boundary conditions) are non-zero. Therefore their implementation can be properly tested.

Domain Ω was divided into 6x6 structured grid; in other words into 36 local elements Ω_e - squares with side equal to $\frac{2}{6}$.

[1] We try to avoid using source term in velocity equation for several reasons. Firstly, each source term is unnecessary complication and another point of failure in the program. Secondly, implementation of velocity source term is more complicated then S_k or S_ω, because of additional pressure equation. Lastly, analytical solutions of Navier-Stokes with variable viscosity are valuable for various purposes, so it is worth to find some of them.

Figure 5.1 is showing exponential spatial convergence, for each unit increase of maximal polynomial order L^2 error decreases by two orders of magnitude.

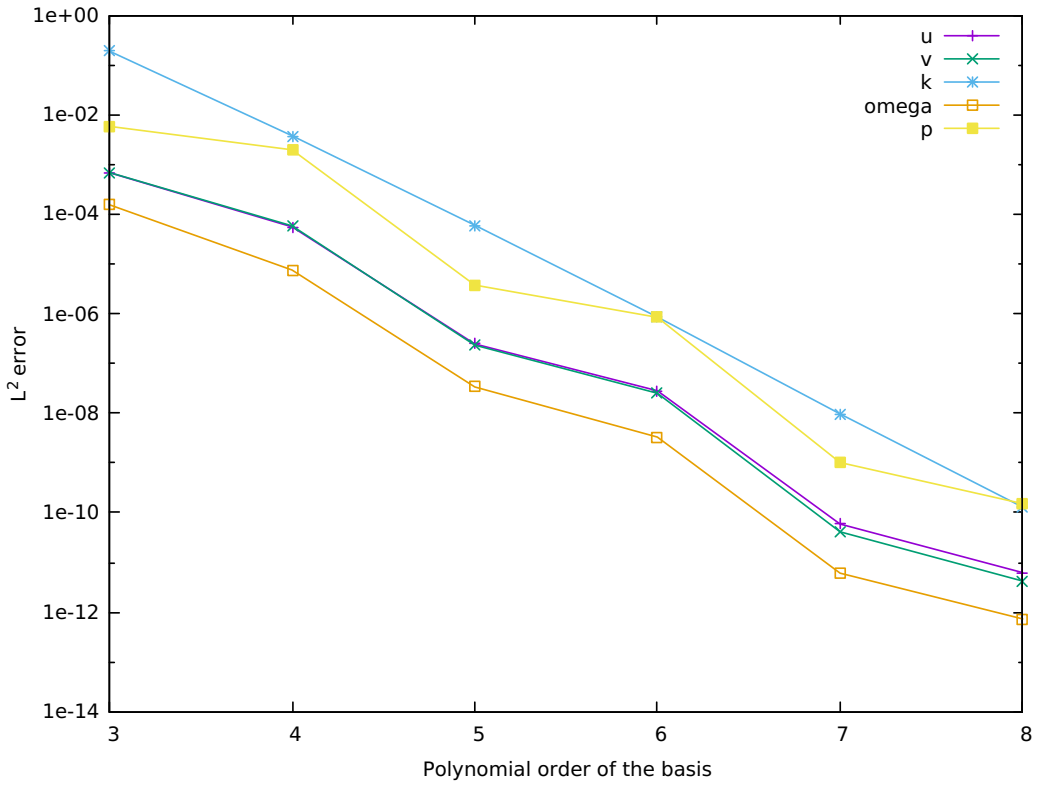

Figure 5.1: Spatial convergence test. Dependence of L^2 error of numerical solution on maximal polynomial order of the basis.

5.3 Temporal convergence test

According to analysis preformed by Guermond and Shen [2003], velocity correction scheme with order of extrapolation J has precision $O(\Delta t^J)$ for $\boldsymbol{v}$ and $O(\Delta t^{\frac{2J-1}{2}})$ for p. Our time discretization scheme (summarized in section 4.3) is based on velocity correction scheme, so we expect similar precision for velocity and pressure.

Also, equations for k and ω have consistent order of extrapolation J, their precision should be also $O(\Delta t^J)$.

Our test case is again based on analytical solution of Wilcox model with source

terms S_k and S_ω (defined by (5.1) and (5.2)).

$$\begin{aligned}
\boldsymbol{v} &= f(t)\left[1-y^2, 0\right]^T \\
p &= -\frac{1}{3}f'x^3 - f'y^2x - 2\nu f x - f'x \\
k &= 200\left(\frac{f'}{2f}x^2 + \frac{f'}{3f}y^2\right) \\
\omega &= 200 \\
\nu &= \nu_0 \\
\nu_T &= \frac{f'}{2f}x^2 + \frac{f'}{3f}y^2
\end{aligned}$$

where ν_0 is constant and $f(t)$ is differentiable function of time, chosen so that $\nu + \nu_T > 1/100$. This last inequality ensures total kinematic viscosity is not too small, which could lead to instability during simulation.

Domain Ω and spatial discretization is still the same as in the previous test. Also the boundary and initial condition have the same form (with respect to the new analytical solution).

Figures 5.2 and 5.3 are showing temporal convergence for time discretization scheme of order $J = 1$, respectively $J = 2$. All variables have precision $O(\Delta t^J)$, with the exception of pressure, which has precision $O(\Delta t^{\frac{2J-1}{2}})$.

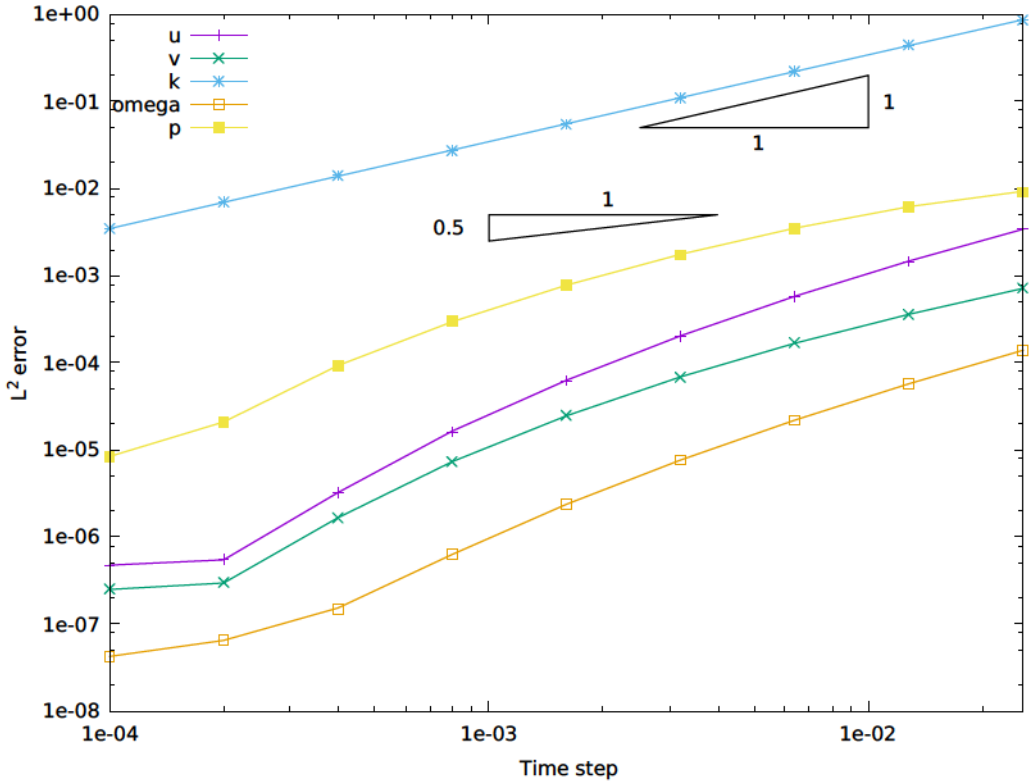

Figure 5.2: Temporal convergence test. Dependence of L^2 error of numerical solution on time step for scheme of order $J = 1$.

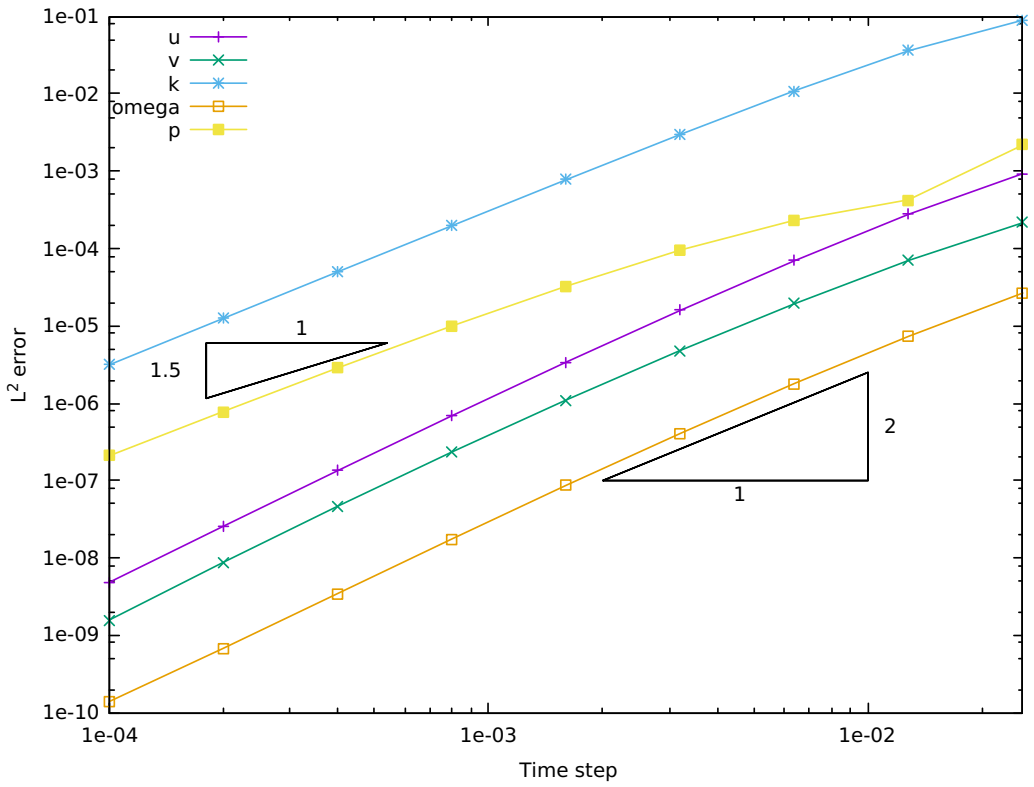

Figure 5.3: Temporal convergence test. Dependence of L^2 error of numerical solution on time step for scheme of order $J = 2$.

5.4 Channel flow in 2D - description

In order to compare our implementation with DNS or other turbulent models, we simulated channel flow with fully developed turbulence.

Channel flow is flow between two horizontal planes - walls, which are parallel to both x and z axes, creating channel. Planes are separated by a distance d. Thanks to the symmetry of the problem along z axis, we take computational domain Ω to be intersection of channel with the plane $z = 0$. Thus, domain Ω is only two dimensional.

Domain Ω is rectangle, with length $2d$ and height d. Top and bottom boundaries correspond to wall of channel, left boundary is inlet, right is outlet as shown in Fig. 5.4.

Ω is divided into 143 local elements Ω_e, which are all rectangles. In x direction, all Ω_e have same size. In y direction, elements are denser towards walls, to improve resolution of large wall gradients of velocity and turbulent variables as number of modes is uniform over whole mesh.

In interior local element Ω_e, basis functions are $\phi^e_{ij}(x, y)$ defined as follows

$$\phi^e_{ij}(x, y) = \phi^e_{ij}(\chi^1_e(\xi_1, \xi_2), \chi^2_e(\xi_1, \xi_2)) = \phi^A_i(\xi_1)\phi^A_j(\xi_2) \tag{5.5}$$

where $i \in \{1, ..., P_1 + 1\}$ and $j \in \{1, ..., P_1 + 1\}$. Definition (2.48) of $\phi^A_i(\xi)$ is presented in section 2.9. Numbers P_1 and P_2 are maximal polynomial orders of ξ_1 and ξ_2. In our case, $P_1 = P_2 = 4$.

In boundary elements, basis is the same, with the exception of boundary modes, which have to satisfy Dirichlet boundary condition.

All basis functions together form approximation space V_δ, where we are looking for solution of each of the model variables.

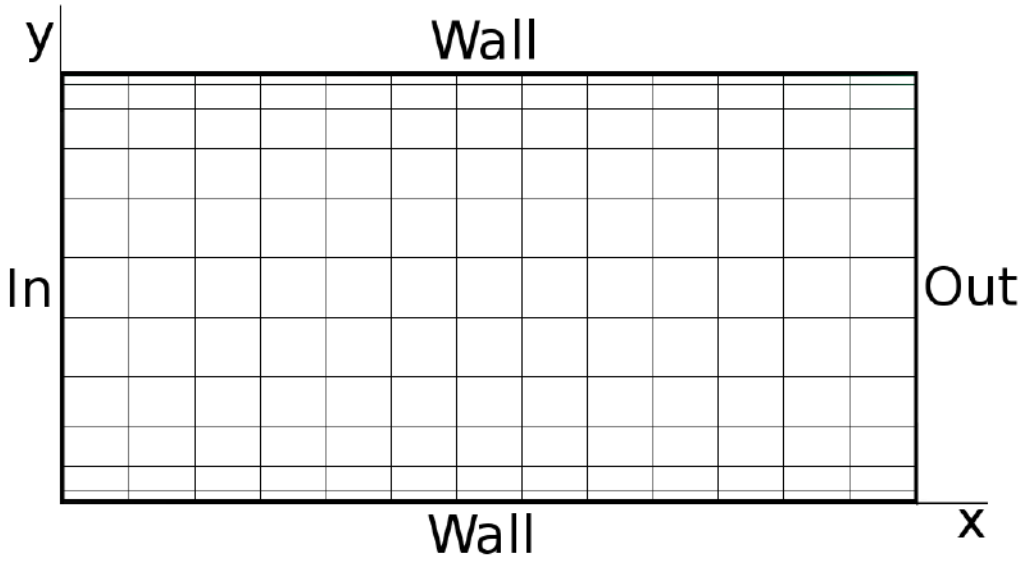

Figure 5.4: Domain Ω for channel flow, with mesh. *In* corresponds for inlet, *Out* for outlet. Distance of walls is d, length of channel is $2d$.

Direct numerical simulation in the same geometry was conducted by [Kim et al., 1987], we will refer to their results as *DNS*. The simulation was computed for $Re = 5600$ satisfying the requirement for fully developed turbulence. They used periodic boundary condition for velocities at inlet and outlet. Results for other turbulence models on the same geometry can be found in [Příhoda, 2007] (in Czech language).

From DNS results, we have profile of v_1 (x velocity [2]) in the channel, we denote it v_{DNS}. Velocity v_2 should be 0, from the symmetry of the problem.

Velocity is normalized by V_m, bulk velocity, defined as

$$V_m = \frac{1}{2}\int_{-\frac{d}{2}}^{\frac{d}{2}} v_1 dy \tag{5.6}$$

Reynolds number is based on V_m and channel width d, $Re = 5600$.

Pressure gradient, driving the flow forward, should be balanced by friction on the walls. So for pressure gradient we have

$$\frac{dp}{dx} = -C_f \tag{5.7}$$

where $-C_f$ is skin friction coefficient. We will use the value from DNS.

$$C_f = C_f^{DNS} = 0.073 Re^{\frac{-1}{4}} \tag{5.8}$$

Boundary conditions are specified in the table 5.1. We expect velocity profile to be similar to v_{DNS}. Thus, inlet velocity is result of interpolation of v_{DNS} into

[2] When we refer to the properties (velocity or pressure) of turbulent flow, we always consider averaged values, to eliminate influence of turbulent fluctuation.

V_δ. Also, pressure p should have gradient equal to $-C_f$. On the walls, no-slip condition is prescribed.

As both components of velocity on the walls are 0, turbulent kinetic energy k (energy of velocity fluctuations) equals to 0.

We denote value of specific dissipation rate ω on wall as ω_w. Specific dissipation rate ω diverges on wall. We could not set ω_w to infinity, but it is enough if ω_w satisfies

$$\omega_w \gg \max_{\Omega}\{\text{all other variables}\} \tag{5.9}$$

In our case, it suffices $\omega_w = 100$.

The model does not specify inlet nor outlet boundary condition for k and ω. We can only hope that in fully developed channel flow, values would not change much between inlet and outlet and use periodic or homogeneous Neumann condition.

Table 5.1: Boundary conditions for channel flow

Variable	Boundary	Type	Value
v_1	Inlet	Dirichlet	v_{DNS}
	Outlet	Neumann	0
	Walls	Dirichlet	0
v_2	Inlet	Dirichlet	0
	Outlet	Neumann	0
	Walls	Dirichlet	0
k	Inlet	Periodic	To Outlet
	Outlet	Periodic	To Inlet
	Walls	Dirichlet	0
ω	Inlet	Neumann	0
	Outlet	Neumann	0
	Walls	Dirichlet	$\omega_w = 100$
p	Inlet	Dirichlet	$2C_f d$
	Outlet	Dirichlet	0
	Walls	*HOBPC* [3]	not specified

Initial conditions are specified in the table 5.2. Velocity and pressure are set according to DNS.

Similarly, initial conditions for turbulent variables k and ω are unknown. Values in the table are just educated guess, based on trial simulations. It is important to note, however, that different initial condition for k and ω should not lead to different results. Flow is driven by inlet velocity and pressure gradient. After some time turbulent variables should converge to the same values even from different initial conditions.

It may appear, that initial and boundary conditions for some variables do not match. Before simulation, Nektar++ projects initial condition into V_δ. Space V_δ satisfies boundary condition, so any mismatch is resolved before actual computation.

Table 5.2: Initial conditions for channel flow

Variable	Value
v_1	v_{DNS}
v_2	0
k	0.01
ω	1
p	$2C_f dx$

5.5 Channel flow in 2D - results

The system behaves dynamically and does not reach stationary state. At first, model has to cope with imprecise initial conditions, especially for turbulent variables. Then, after it finds suitable values of k and ω, system stabilizes. Nevertheless, there are still time periods, when all fields oscillate, which are followed by periods, when the fields look almost stationary.

It is probably caused by the nature of turbulent model. Turbulent viscosity ν_T, which damps oscillations, is determined by k and ω. Production term for k contains velocity gradient, so it reacts on velocity oscillations by increasing the production of k. Larger k results in larger ν_T and damps oscillation.

When fields do not oscillate, production of k decrease and so does ν_T, which results in non-stationary velocity field.

Inlet velocity (see figure 5.5) v_{in} differs from v_{DNS} because of the imprecise interpolation to space V_δ. Outlet velocity v_{out} is similar, but not the same, which means that we did not "guess" the right inlet boundary condition and flow in channel is slowly changing its profile.

The biggest disagreement between DNS and our model is in pressure p. Gradient $\frac{\partial p}{\partial x}$ in our simulations is approximately twice as large as the gradient from DNS, see figure 5.8. Inlet pressure profile (see 5.9) should be constant. But, probably because of the inappropriate inlet velocity condition, it drops near the walls almost to 0. The problem can be easily seen also from 3D view (figure 5.10.

On the other hand, turbulent variables behave as we expected, see 5.6 and 5.7.

Discrepancies in velocity and pressure between our model and DNS can have several reasons.

We already mentioned wrong inlet boundary conditions for velocity. But also, since the resulting flow from simulation is neither stationary nor periodic, periodic boundary conditions for k may be wrong. Spectral method is very sensitive to inappropriate boundary condition.

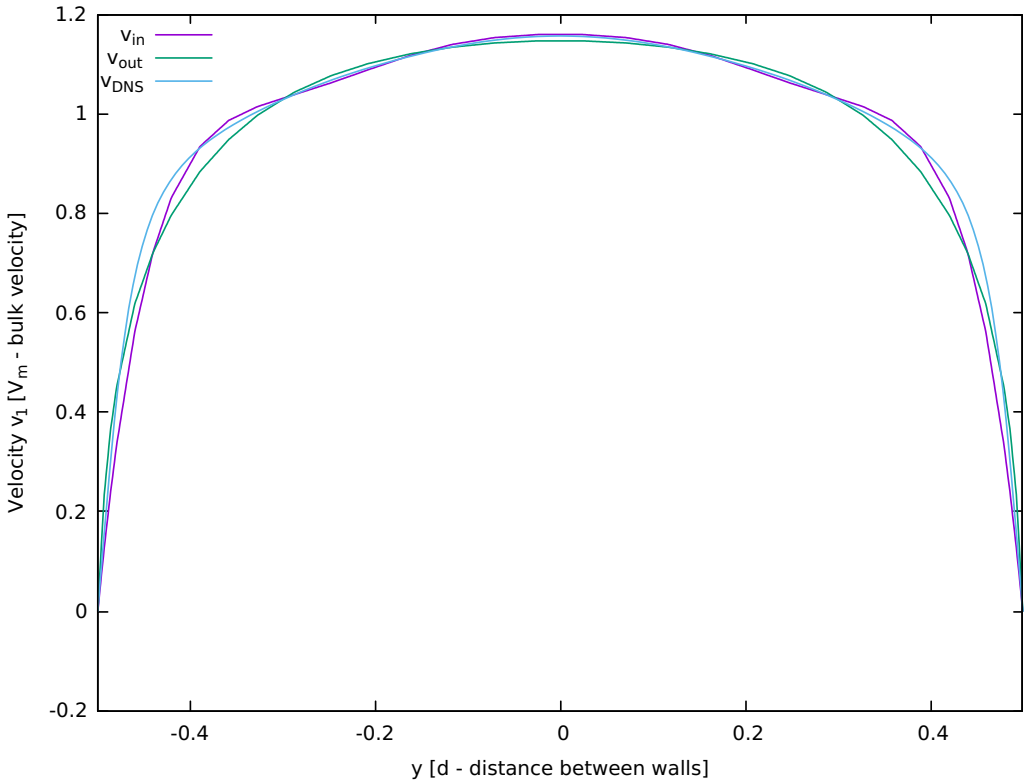

Figure 5.5: Profiles of velocity v_1 at inlet and outlet, compared with profile from DNS.

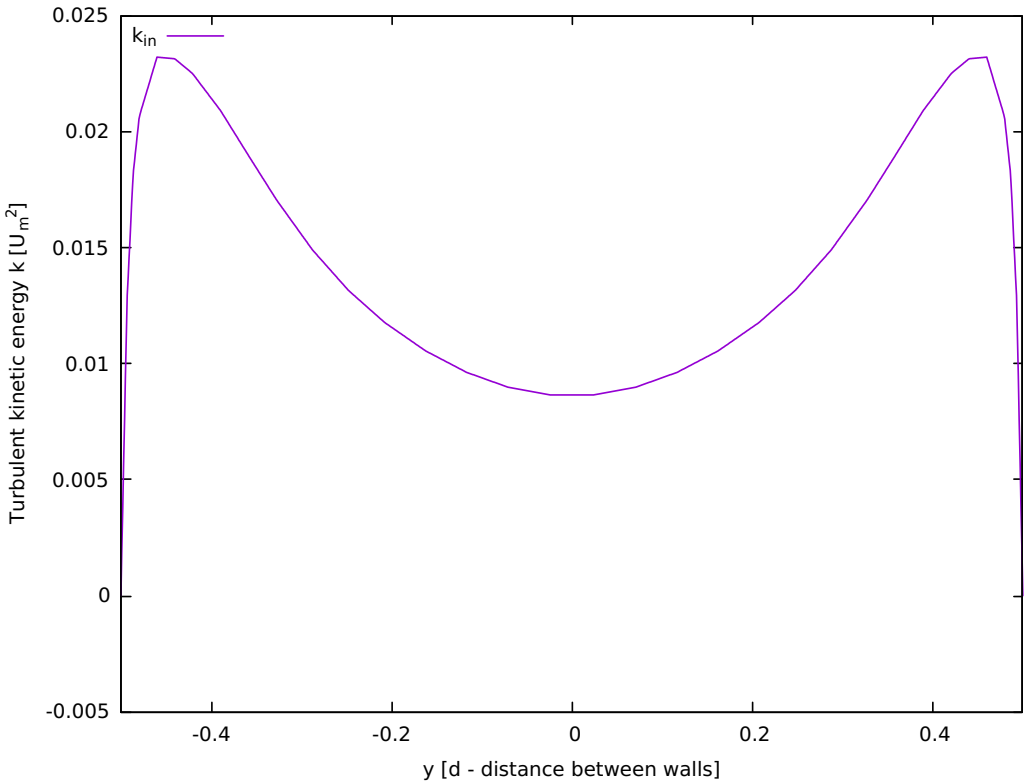

Figure 5.6: Profile of turbulent kinetic energy k at inlet. Because of periodic BC, profile at outlet is the same.

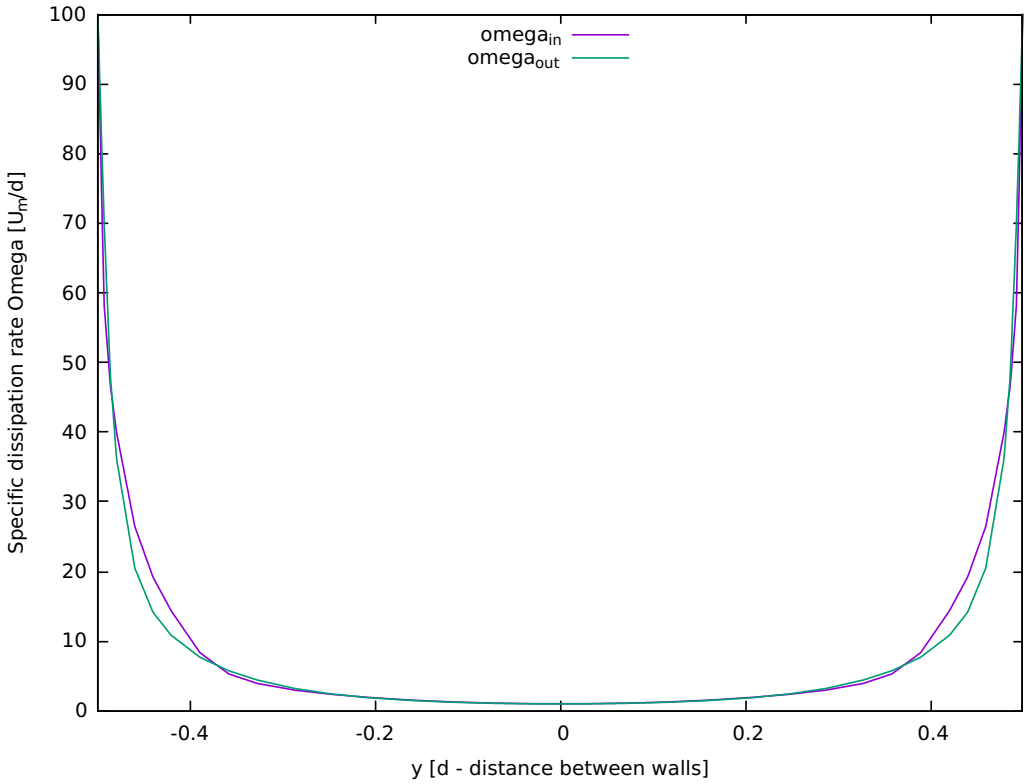

Figure 5.7: Profile of specific dissipation ω at inlet and outlet.

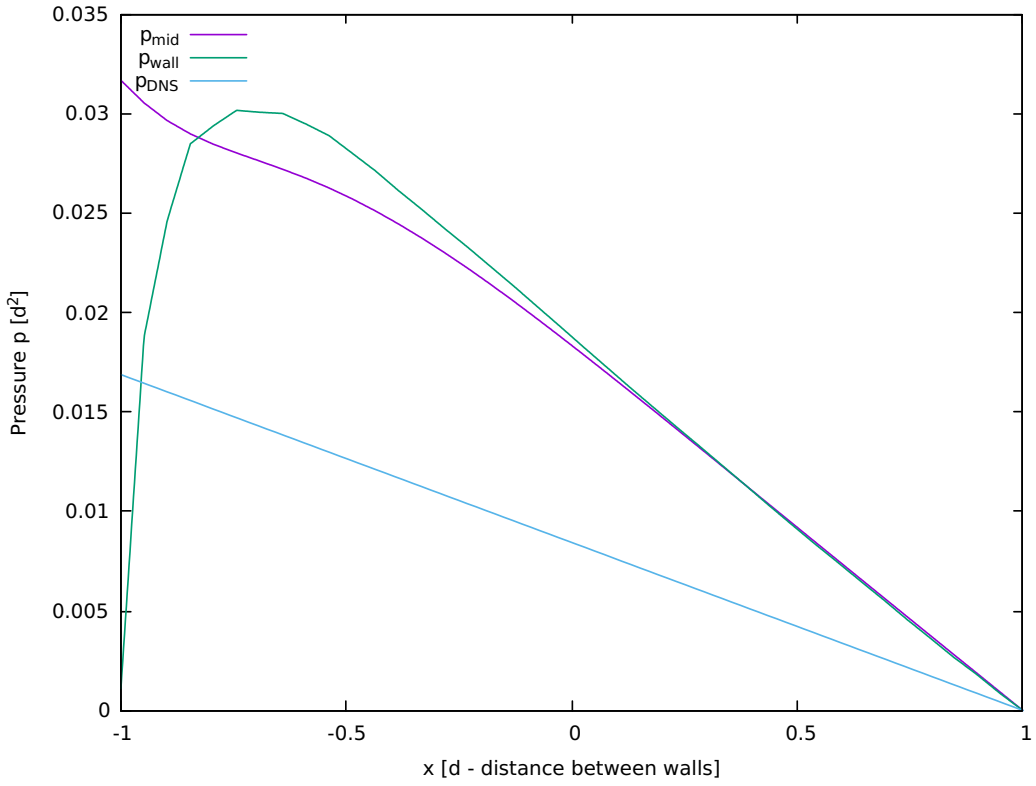

Figure 5.8: Pressure profile *along* channel, parallel to x axis, compared with profile from DNS. p_{wall} is in the middle of the channel, p_{wall} is pressure along the wall.

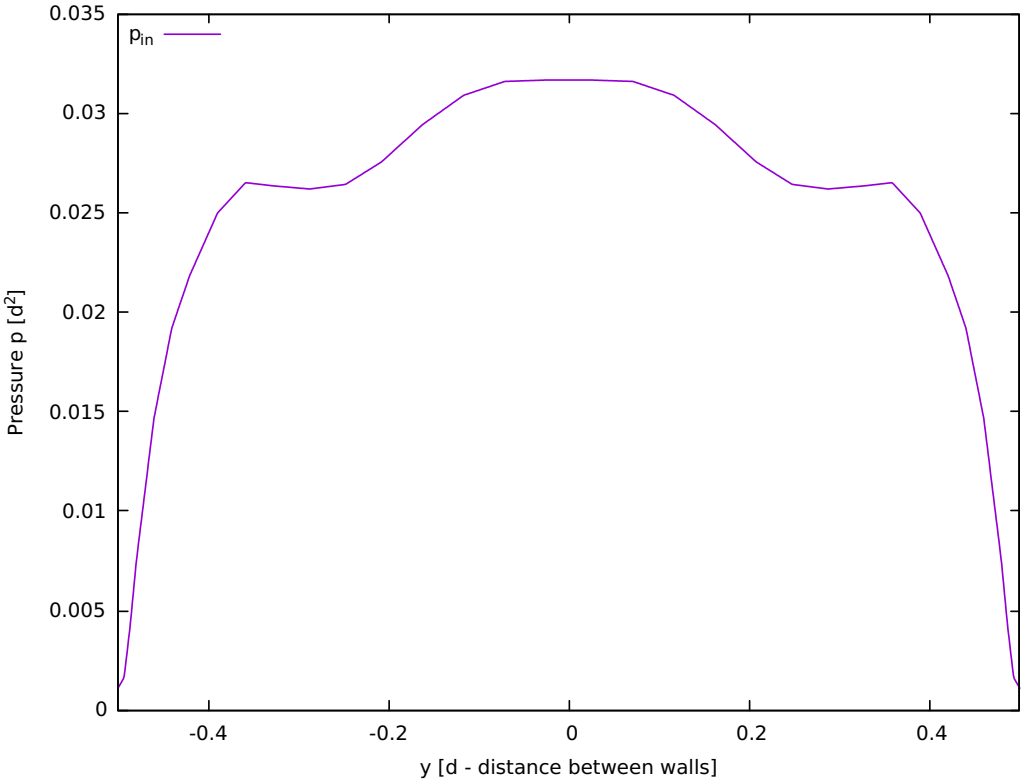

Figure 5.9: Profile of pressure at inlet.

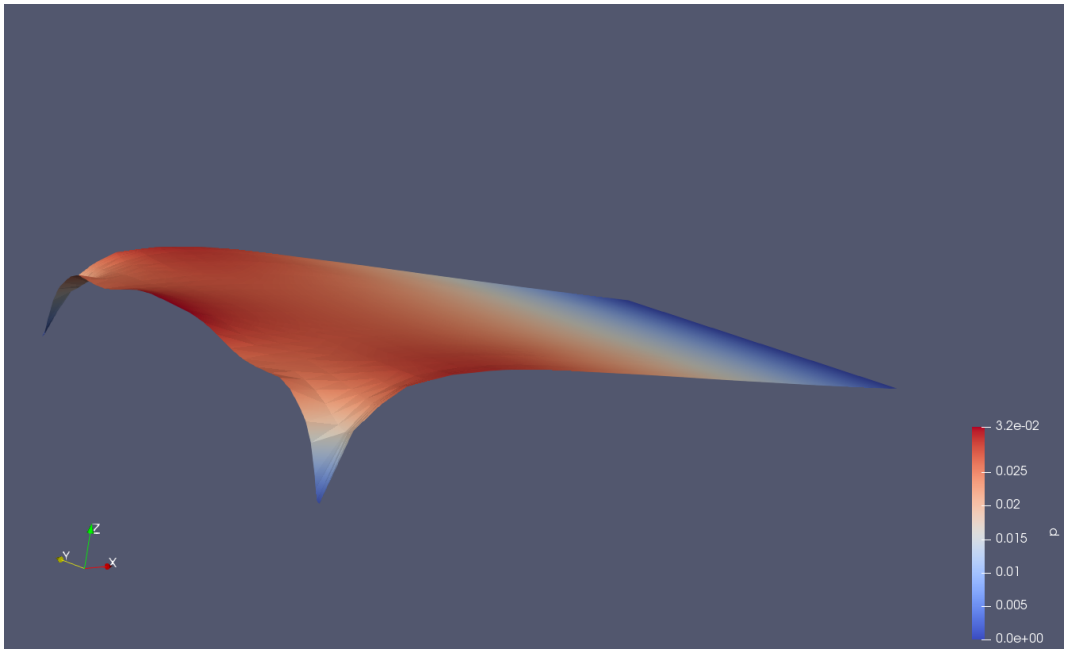

Figure 5.10: 3D view of pressure field.

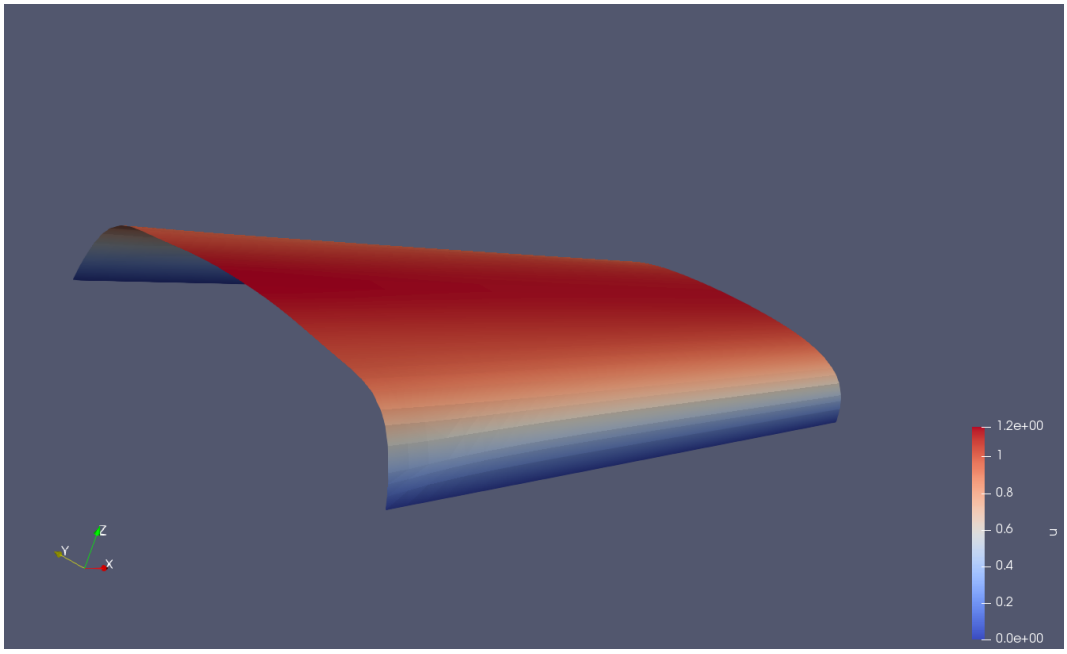

Figure 5.11: 3D view of velocity field.

Conclusion

We have implemented 2D formulation of two turbulent models, Kolmogorov's and Wilcox, into spectral element computational framework, Nektar++.

These models belong to class of k-ω two-equation models and they use non-constant *turbulent viscosity* to propagate the effect of turbulence into Navier-Stokes equations for incompressible fluid.

Spatial discretization is based on spectral element method, so we summarized its fundamental aspects in chapter 2.

Time discretization stems from the velocity correction scheme, which decouple velocity and pressure system. Velocity correction scheme was invented for system with constant viscosity, so we had to modify the scheme and split viscosity into implicit (constant in time) and explicit (time dependent) part. The same splitting was applied to viscous terms of evolution equations for k and ω.

We briefly explained abstract *general linear method* (GLM) framework , which is the main object used for time stepping in Nektar++. Schemes in GLM framework are characterized by coefficient matrices and external routines. We derived these matrices and routines for our time discretization scheme.

Kolmogorov's and Wilcox model belong to larger category of *turbulent viscosity* models. Within this category, differences are only in the calculation of the turbulent viscosity. The implementation in C++ is therefore divided into two parts - Navier-Stokes equations with variable viscosity and turbulence equations for k and ω, where the viscosity is computed. This design is modular and ready for easy implementation of any model from the same category.

Also, our models share one common codebase. It is possible to switch between them just by changing the input file.

We found exact solutions of Navier-Stokes equations with variable viscosity and tested the implementation against them. These solution could be valuable as a test cases for whole category of *turbulent viscosity* models.

We were not able to find exact non-trivial solutions of turbulence equations, so we tested implementation with *source terms*. Source term are auxiliary terms on the right hand side of turbulence equations. For given functions k_0 and ω_0, we constructed them, so they cancel out all other terms in turbulence equations. This effectively makes k_0 and ω_0 to be exact solution.

In the tests, we have shown spectral and temporal convergence for all variables.

We simulated channel flow in 2D with Wilcox model and compared results with direct numerical simulation. There were discrepancies in velocity profiles across the channel and also in pressure gradients. Observed differences might be explained by high sensitivity of spectral element method to boundary and initial condition, which are unknown, because turbulent model does not specify them.

Model and solver contain many parameters that can improve the computations. Also, there are different choices for boundary and initial conditions. The thesis was centered around the implementation, complex numerical experiments and more realistic problems might be addressed in future studies.

Bibliography

I. Babuska, B. A. Szabo, and I. N. Katz. The p-version of the finite element method. *SIAM Journal on Numerical Analysis*, 18(3):515–545, jun 1981. doi: 10.1137/0718033.

J.P. Boyd. *Chebyshev and Fourier Spectral Method*. Dover Publications Inc., 2001. ISBN 0486411834. URL `https://www.ebook.de/de/product/3344687/boyd_chebyshev_and_fourier_spectral_meth.html`.

M. Bulíček and J. Málek. Large data analysis for kolmogorov's two-equation model of turbulence. 2016.

J.C. Butcher. General linear methods. *Computers & Mathematics with Applications*, 31(4-5):105–112, feb 1996. doi: 10.1016/0898-1221(95)00222-7.

C.D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R.M. Kirby, and S.J. Sherwin. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205–219, jul 2015. doi: 10.1016/j.cpc.2015.02.008.

C. Canuto, M. Y. Hussaini, A. Quarteroni, and Th. A. Zang. *Spectral Methods: Evolution to Complex Geometries and Applications to Fluid Dynamics*. Springer-Verlag GmbH, 2007. ISBN 3540307273. URL `https://www.ebook.de/de/product/5226601/claudio_canuto_m_yousuff_hussaini_alfio_quarteroni_thomas_a_zang_spectral_methods.html`.

J. D. Cook. Orthogonal polynomials and gaussian quadrature. 2008. URL `https://www.johndcook.com/OrthogonalPolynomials.pdf`. Accessed: 2020-01-04.

Developer Guide for Nektar++. *Developer Guide for Nektar++: Spectral/hp Element Framework, version 4.4.1*, September 2017.

Doxygen. Doxygen for nektar++, version 4.4.1, 2017. URL `http://doc.nektar.info/doxygen/4.4.1/`. Accessed: 2020-05-31.

D. Gottlieb. *Numerical analysis of spectral methods : theory and applications*. Society for Industrial and Applied Mathematics, Philadelphia, 1977. ISBN 9780898710236.

J. L. Guermond and J. Shen. Velocity-correction projection methods for incompressible flows. *SIAM Journal on Numerical Analysis*, 41(1):112–134, jan 2003. doi: 10.1137/s0036142901395400.

G.-S. Karamanos and S.J. Sherwin. A high order splitting scheme for the navier–stokes equations with variable viscosity. *Applied Numerical Mathematics*, 33(1-4):455–462, may 2000. doi: 10.1016/s0168-9274(99)00112-9.

G. Karniadakis. *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press, New York, 2005. ISBN 9780198528692.

G. E. Karniadakis, S. A. Orszag, and M. Israeli. High-order splitting methods for the incompressible Navier-Stokes equations. *Journal of Computational Physics*, 97:414–443, December 1991. doi: 10.1016/0021-9991(91)90007-8.

J. Kim, P. Moin, and R. Moser. Turbulence statistics in fully developed channel flow at low reynolds number. *Journal of Fluid Mechanics*, 177:133–166, apr 1987. doi: 10.1017/s0022112087000892.

A. N. Kolmogorov. Equations of turbulent motion in an incompressible fluid. *Dokl. Akad. Nauk SSSR*, 30(4):299–303, 1941.

D. Lars. An introduction to turbulence models. *Gotemburgo: Chalmers University of Technology*, 2017.

Nektar++ User Guide. *User Guide for Nektar++: Spectral/hpElement Framework, version 4.4.1*, November 2017. URL `http://doc.nektar.info/userguide/4.4.1/`.

J. Oden. *An introduction to the mathematical theory of finite elements*. Dover Publications, Mineola, N.Y, 2011. ISBN 9780486462998.

A. T. Patera. A spectral element method for fluid dynamics: Laminar flow in a channel expansion. *Journal of Computational Physics*, 54(3):468–488, jun 1984. doi: 10.1016/0021-9991(84)90128-1.

J. Pech. On computations of temperature dependent incompressible flows by high order methods. *EPJ Web of Conferences*, 114:02089, 2016a. doi: 10.1051/epjconf/201611402089. URL `http://dx.doi.org/10.1051/epjconf/201611402089`.

J. Pech. *Numerical modeling of unstable fluid flow past heated bodies*. PhD thesis, Mathematical Institute of Charles University, 2016b. URL `http://hdl.handle.net/20.500.11956/81818`.

S. B. Pope. An explanation of the turbulent round-jet/plane-jet anomaly. *AIAA Journal*, 16(3):279–281, mar 1978. doi: 10.2514/3.7521.

J. Příhoda. *Matematicke modelování turbulentního proudění*. Nakladatelství CVUT, Praha, 2007. ISBN 9788001036235.

T. Rannacher. Finite Element Methods for the Incompressible Navier-Stokes Equations. August 1999. URL `https://ganymed.math.uni-heidelberg.de/Oberwolfach-Seminar/CFD-Course.pdf`. Accessed: 2020-01-04.

V. M. Tikhomirov. *Selected Works of A.N. Kolmogorov. Volume I: Mathematics and Mechanics*. Mathematics and its Applications. Springer, 1991. ISBN 9027727961,9789027727961.

P. E.J. Vos, C. Eskilsson, A. Bolis, S. Chun, R. M. Kirby, and S. J. Sherwin. A generic framework for time-stepping partial differential equations (PDEs): general linear methods, object-oriented implementation and application to fluid problems. *International Journal of Computational Fluid Dynamics*, 25(3):107–125, mar 2011. doi: 10.1080/10618562.2011.575368.

D. Wilcox. *Turbulence modeling for CFD*. DCW Industries, La Canada, Calif, 2006. ISBN 9781928729082.

List of Figures

List of Tables

A. Orthogonal polynomials

General class of orthogonal polynomials are Jacobi polynomials $J_i^{\alpha,\beta}(\xi)$, which have defining property

$$\int_{-1}^{1} J_i^{\alpha,\beta}(\xi) J_j^{\alpha,\beta}(\xi)(\xi+1)^{\alpha}(-\xi+1)^{\beta} d\xi = C_{ij}\delta_{ij} \tag{A.1}$$

and $J_0^{\alpha,\beta}(\xi) = (\xi+1)^{-\alpha}(-\xi+1)^{-\beta}$.

Equivalent definition is given by Rodrigues formula

$$J_i^{\alpha,\beta}(\xi) = \frac{(-1)^i}{2^i n!}(1+\xi)^{-\alpha}(1-\xi)^{-\beta}\frac{d^i}{d\xi^i}\left[(1+\xi)^{\alpha}(1-\xi)^{\beta}(1-\xi^2)^i\right] \tag{A.2}$$

Special case of Jacobi polynomials are Legendre polynomials $L_i(\xi)$, defined as

$$L_i(\xi) = J_i^{0,0}(\xi) \tag{A.3}$$

B. Structure of attachments

All attachments are *.zip* file.

B.1 Source code

In folder *nektar* there is complete source code of Nektar++. It includes implementation of Wilcox and Kolmogorov models.

To separate the implementation of the models from the rest of the code, we created file *complete_diff.txt*.

It is a list of all differences between original code and the version with turbulent models, created by git.

Original code refers to commit with hash *2a989b33* by Michael Turner. It can be found in git repository *https://gitlab.nektar.info/nektar/nektar/*.

In the same way, we created file *name_of_changed_files.txt*. There are the names of files that have been edited during development.

B.2 Test and channel flow

The folder *time_test_IMEX1*, *time_test_IMEX2* and *spectral_test* contain input XML files and results of the tests from Chapter 5. The folder *channel_flow* contains everything that is needed for the channel flow simulation, including velocity profile from DNS.

B.3 MATLAB script for source terms

Source terms are too complicated to be calculated by hand. MATLAB script has been created to calculate them. It can be found in the folder *source_terms*.

There is also *bash* script, that takes output of MATLAB script and substitute the results into generic XML input file, that is also included.