FLOW-SESE Course 2021

Nek5000

Theory, Implementation and Optimization

**Edited by Saleh Rezaeiravesh, Niclas Jansson,
Adam Peplinski, Jonathan Vincent, Philipp Schlatter**

Swedish e-Science Research Centre

Table of contents

Preface

This book contains the 10 group reports prepared during the Nek5000 course organised at KTH during the spring term 2021, with support by the FLOW Centre and Swedish e-Science Education (SESE) as part of the Swedish e-Science Research Centre (SeRC).

Overview:

In this course, we discuss the theory, numerics and implementation of the various methods in Nek5000. Nek5000 is a spectral-element based solver for CFD, and is available as an open source package. The code has become popular among researchers worldwide, mainly because of its relatively high accuracy per grid point, its parallel efficiency and the wide user community and available packages (e.g. statistics, linear stability, adjoints etc.). The development of the various methods goes back to the mid 80ies, and a number of different approaches are now implemented, both when it comes to the integration of Navier-Stokes, the communication kernels and efficient single-core performance. For SeRC, Nek5000 is one of the two main codes to focus on, and in Sweden about 30 people are actively working with Nek5000. Therefore, we believe that offering an in-depth course on the internal workings of Nek5000, the relevant theory of SEM, implementations and physical models is a relevant contribution to SESE.

Intended learning outcomes:

- Understand the basic theory of solutions to the Navier-Stokes equations and its implications to the numerical treatment
- Understand the formulation of the pressure treatment for both PnPn and PnPn-2 methods
- Spectral-element discretisation and necessary stabilization, time integrators
- Workflow of Nek5000, and the various iterative solvers (including preconditioners, projections)
- Imposition of boundary conditions and limitations
- Implementation of meshing and metrics
- Understanding of the basic communication kernels (GS lib)
- Pre-requisites and target group
- Basic knowledge in scientific computing, programming and CFD. Basic interest in fluid mechanics is recommended as most practical examples are related to fluid mechanics.

The course is aimed at researchers for academia (MSc, PhD students, researchers) and industry that use (or intend to use) Nek5000 in their work. The focus is on the underlying theory and implementation, and not on running Nek5000 for specific cases. Of course, support to install and get started with Nek5000 can be provided if necessary.

Course schedule

The course started with 6 meetings:

- Thursday, 11/3, 13-16: Introduction and outline, style files. Short tutorial how to run Nek5000
- Friday, 12/3, 10-12: Fortran 77 Tutorial
- Monday 15/3 13-15: Fourier and Chebyshev methods part I
- Tuesday 16/3 13-15: Fourier and Chebyshev methods part II
- Wednesday 17/3 13-15: Fourier and Chebyshev methods part III
- Friday 11/6 11-13: Computer Architectures/HPC

The second part of the course consists of group work and presentations, spread over 11 meetings during the sprint term 2021:

- SEM discretisation (09.04.2021, 9-11) Legendre polynomials + SEM discretisation. Properties and weak form, discretisation of advection/diffusion. Continuous level
- Discrete operators (09.04.2021, 11-13) Matrix-free formulation, tensor operations, implementation of operators, axhelm, mxm, gather-scatter. Discrete level.
- Pressure correction method and PN-PN-2 formulation (16.04.2021, 13-15)
- PN-PN formulation (23.04.2021, 13-15)
- Iterative solvers and projection method (30.04.2021, 13-15)
- Time integration, OIFS, characteristics (07.05.2021, 13-15)
- Pressure preconditioners/Schwarz Multigrid (14.05.2021, 13-15)
- Direct stiffness summation (21.05.2021, 13-15)
- Solver stabilisation (28.05.2021, 13-15)
- Work balancing(04.06.2021, 13-15)
- Meshing and boundary conditions (11.06.2021, 13-15)

Recommended book

Deville, Fischer, Mund, High-Order Methods for Incompressible Fluid Flow, Cambridge University Press, 2002

Link to Nek5000: https://nek5000.mcs.anl.gov

Stockholm, 2021-11-17

Saleh Rezaeiravesh, Niclas Jansson, Adam Peplinski, Jonathan Vincent, Philipp Schlatter

Spectral element method and discrete operators

A. Perez A. V. Mohanan M. Moniri Z. Yuan

April 13, 2021

Abstract

This report describes the topics covered by groups G01 and G02. The report was made collaboratively by the authors in an online HackMD document with accompanying presentation. The report is organized as follows. In sections 1 and 2, continuous Galerking method and basis functions in spectral element methods are covered. In section 3, spectral element method is applied to solve the advection-diffusion problem with different type of boundary conditions and the simple time integration method. In the end of this section, relationship between modal and nodal bases is compared. In section 4 we show how the formulation is extended to multiple sub-domains, while in section 5 the formulation for multiple dimensions is shown. Finally, in section 6 we take a look at how discrete spectral operators are efficiently implemented in practice with examples of code from Nek5000 version 19.

1 Continuous Galerkin formulation

Based on posing problems in their variational (weak) form which is an equivalent integral form to that of their classical representation. Take advection-diffusion equation as an example: expressed in strong form as

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}. \tag{1}$$

A **residual** can be defined as:

$$L(u) = \frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2}. \tag{2}$$

The weak form is obtained by multiplying the residual by a **test function** $v(x)$ and integrating over the domain:

$$(v, L) = \int_\Omega v(\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2})dx = 0. \tag{3}$$

The order of the second derivative in the diffusion term can be reduced by integrating by parts:

$$\int_\Omega (v\frac{\partial u}{\partial t} + cv\frac{\partial u}{\partial x} + \nu\frac{\partial v}{\partial x}\frac{\partial u}{\partial x})dx = 0, \tag{4}$$

by choosing a basis $v(x)$ that vanishes at the boundaries of the domain $\partial\Omega$. The Galerkin formulation is then the following.

Find u such that:

$$\int_\Omega (v\frac{\partial u}{\partial t} + cv\frac{\partial u}{\partial x} + \nu\frac{\partial v}{\partial x}\frac{\partial u}{\partial x})dx = 0. \tag{5}$$

A potential candidate for $u(x,t)$ can be represented by a combination of **trial functions** $\psi(x)$ such that:

$$u_N(x,t) = \sum_{n=0}^{N} u_n(t)\psi_n(x) = \underline{\psi}(x)^T \cdot \underline{u}(t), \tag{6}$$

where $\underline{\psi}(x)$ and $\underline{u}$ are the column vectors for the basis functions and the coefficients, respectively. In the Galerkin method, the test function $v(x)$ can also be represented by the same set of trial functions.

$$v(x) = \sum_{n=0}^{N} v_n\psi_n(x) = \underline{\psi}(x)^T \cdot \underline{v}. \tag{7}$$

Applying this to the weak formulation of the Advection/Diffusion equation and considering $c = \nu = 1$ for simplicity, the following is obtained:

$$\sum_{i=0}^{N}\sum_{j=0}^{N} v_i(\int_\Omega \quad _i \quad _j dx)\frac{du_j}{dt} + \sum_{i=0}^{N}\sum_{j=0}^{N} v_i(\int_\Omega \quad '_i \quad '_j dx)u_j + \sum_{i=0}^{N}\sum_{j=0}^{N} v_i(\int_\Omega \quad _i \quad '_j dx)u_j = 0, \tag{8}$$

where $'$ stands for a derivative with respect to x. The above equation in matrix form is:

$$v^T M\frac{d\underline{u}}{dt} = -v^T K\underline{u} - v^T C\underline{u}, \tag{9}$$

which implies,

$$M\frac{d\underline{u}}{dt} = -K\underline{u} - C\underline{u}, \tag{10}$$

and,

$$M[i,j] = \int_\Omega \quad _i(x) \quad _j(x)dx, \tag{11}$$

$$K[i,j] = \int_\Omega \frac{d\psi_i(x)}{dx}\frac{d\psi_j(x)}{dx}dx, \tag{12}$$

$$C[i,j] = \int_\Omega \quad _i(x)\frac{d\psi_j(x)}{dx}dx, \tag{13}$$

where M, K, and C are mass, stiffness, and convective matrices, respectively. There are still some questions to be answered: **what type of basis functions should be used?**

2 Basis Functions in Spectral Element Methods

Both modal and nodal bases are used in Nek5000 which is based on spectral-element method.

In the modal approach, the chosen bases are known and the coefficients in the expansion must be calculated. In the nodal approach, the coefficients are simply the nodal values of a given function, while the polynomial basis must be constructed. They each provide different advantages or properties that can be exploited.

2.1 Legendre Polynomials (Modal Basis)

Legendre polynomial of order k, which is denoted by $L_k(x)$, is the eigensolution of the Legendre differential equation which is shown below:

$$-\frac{d}{dx}\left(\left(1-x^2\right)\frac{d}{dx}L_k(x)\right) = k(k+1)L_k(x). \tag{14}$$

Legendre polynomials can be defined in many ways, and the various definitions highlight different aspects as well as suggest connections to different mathematical structures and physical and numerical applications. In physical settings, the Legendre differential equation arises naturally whenever one solves the Laplace equation by separation of variables in spherical coordinates.

2.1.1 Visual representation

The first six Legendre polynomials are:

n	$L_n(x)$
0	1
1	x
2	$\frac{1}{2}\left(3x^2-1\right)$
3	$\frac{1}{2}\left(5x^3-3x\right)$
4	$\frac{1}{8}\left(35x^4-30x^2+3\right)$
5	$\frac{1}{8}\left(63x^5-70x^3+15x\right)$

The polynomials are plotted in figure 1 for $x \in [-1, 1]$.

2.1.2 Properties

- Orthogonality: The set of Legendre polynomials form an orthogonal family, that means:

$$\int_{-1}^{1} L_m(x)L_n(x)dx = \frac{2}{2n+1}\delta_{mn}. \tag{15}$$

- Completeness: Legendre polynomials are complete. This means that the given piece-wise continuous function $f(x)$ with finitely many discontinuities

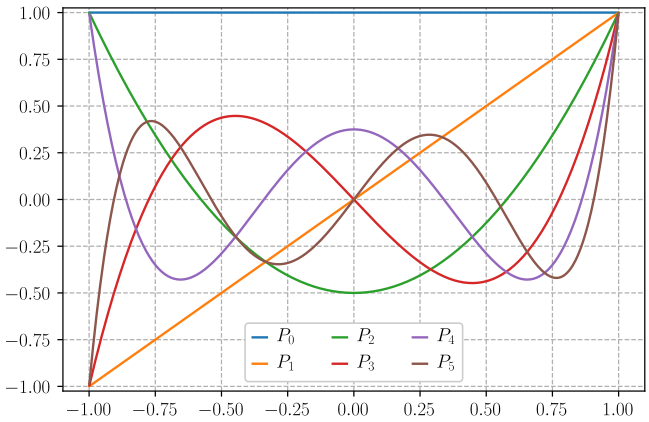


Figure 1: Legendre Polynomials up to order 5.

in the interval [-1,1], can be approximated by the following sum:

$$f_n(x) = \sum_{l=0}^{n} a_l L_l(x), \tag{16}$$

in which a_l is a coefficient and $L_l(x)$ is the Legendre polynomial of order l, and $f_n(x) \to f(x)$ as $n \to \infty$.

The completeness property can be written in the following form:

$$\sum_{l=0}^{\infty} \frac{2l+1}{2} L_l(x) L_l(y) = \delta(x-y). \tag{17}$$

Note x, $y \in [-1, 1]$ and $\delta(x-y) = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{ip(x-y)} dp$.

- Bonnet's recursion formula: The Legendre polynomials can also be defined as the coefficients in a formal expansion in powers of t of the generating function Abramowitz [1974]:

$$\frac{1}{\sqrt{1-2xt+t^2}} = \sum_{n=0}^{\infty} L_n(x) t^n. \tag{18}$$

By differentiating the generating function with respect to t, the following is obtained:

$$\frac{x-t}{\sqrt{1-2xt+t^2}} = (1-2xt+t^2) \sum_{n=1}^{\infty} n L_n(x) t^{n-1}. \tag{19}$$

By substituting the denominator of the left-hand-side of the above equation with the summation and rearranging the whole equation we get:

$$nL_n(x)t^{n-1} - (2n+1)xL_n(x)t^n + (1+n)L_n(x)t^{n+1} = 0. \tag{20}$$

Equating terms with identical powers of t we find:

$$(1+n)L_{1+n}(x) - (2n+1)xL_n(x) + nL_{n-1}(x) = 0, \quad n \geq 2. \tag{21}$$

- Other properties
  - $(2n+1)L_n(x) = L'_{n+1}(x) - L'_{n-1}(x)$,
  - $L_n(x)$ is even or odd if n is even or odd: $L_n(-x) = (-1)^n L_n(x)$,
  - $L_n(1) = 1$, $L_n(-1) = (-1)^n$.

2.2 Lagrange Interpolation Polynomials (Nodal Basis)

For an elemental grid on $\hat{\Omega}$ ($\hat{\Omega} := \{\xi | -1 \leq \xi \leq 1\}$) with $p+1$ nodes $\Xi_{p+1} := \xi_0, \cdots, \xi_p$ the Lagrange interpolation polynomial of a smooth function $f(\xi)$ on $[-1, 1]$ is defined as

$$I_p f(\xi) = \sum_{i=0}^{p} f(\xi_i)\pi_i(\xi) \quad \xi \in \hat{\Omega}, \tag{22}$$

and

$$\pi_i(\xi) = \prod_{\substack{j=0 \\ i \neq j}}^{p} \frac{\xi - \xi_i}{\xi_i - \xi_j} \quad 0 \leq i, j \leq p. \tag{23}$$

2.2.1 Properties

- An interesting property of the Lagrangian polynomials is the fact that: $\pi_i(x_j) = \delta_{ij}$.
- If the Lagrange polynomials are applied to the Gauss-Lobatto-Legendre (GLL) points, it is also possible to define them as

$$\pi_j(\xi) = \frac{-1}{N(N+1)} \frac{(1-\xi^2)L'_N(\xi)}{(\xi - \xi_j)L_N(\xi_j)} \quad 0 \leq j \leq N. \tag{24}$$

In other words, Lagrange interpolation polynomials can be expressed in terms of Legendre polynomials.

- Lagrange polynomials have **Local support** on $\hat{\Omega}$, which means that they are non-zero only in [-1,1] and vanish outside these boundaries.

3 Spectral Element Method in 1D

In Spectral Element Method, the integration domain Ω is partitioned into intervals (elements). In one dimension, a partition of $(a, b) \in \Omega$ with E elements, which is denoted by ΔE, can be written as

$$\Delta E : a = x_0 < x_1 < \cdots < x_E = b. \tag{25}$$

A reference or parent element is also defined as $\hat{\Omega} : [-1 < \xi < 1]$. Particularly in this report, the points ξ correspond to the **GLL** points. For the convenience of numerical integration (GLL quadrature rule), we need to map the element from physical domain to the reference domain $\hat{\Omega}$.

In 1D, a simple mapping that relates the upper boundary x_u^e and the lower boundary x_l^e of element e with the reference element is

$$x^e(\xi) = \frac{1-\xi}{2}x_l^e + \frac{1+\xi}{2}x_u^e = \frac{1+\xi}{2}(x_u^e - x_l^e) + x_l^e, \tag{26}$$

where $(x_u^e - x_l^e)$ is the element size h. Also, the inverse mapping can be defined as

$$\xi(x^e) = 2\frac{x^e - x_l^e}{h} - 1. \tag{27}$$

For illustration, the 1D spectral element mesh of order 5 with 1 element in a domain $\Omega = [0, 2]$ is shown in Figure 2:

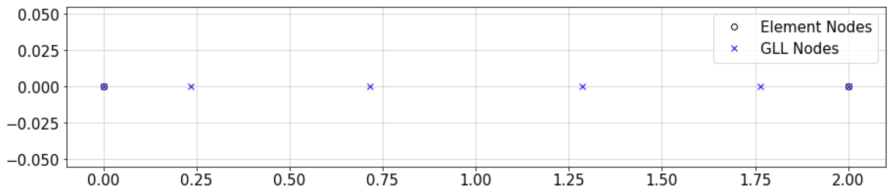


Figure 2: GLL nodes within one element in a domain Ω= [0,2].

3.1 SEM formulation for Advection-Diffusion Equation with a Single Element

In this section, the advection-diffusion problem is solved within a single element. Having chosen the GLL points for the reference grid, and by choosing to follow a nodal approach, i.e. the Lagrange polynomials as bases, it is possible to complete the discretization of the advection-diffusion equation:

$$\int_\Omega (v\frac{\partial u}{\partial t} + cv\frac{\partial u}{\partial x} + \nu\frac{\partial v}{\partial x}\frac{\partial u}{\partial x})dx = 0. \tag{28}$$

For this purpose, let us recall the process to derive the stiffness matrix by only analyzing the diffusion term and considering $\nu = 1$. Starting from the following expression for the element e with domain Ω:

$$\int_{\Omega^e} (\frac{\partial v}{\partial x}\frac{\partial u}{\partial x})dx, \tag{29}$$

there are two main aspects to keep in mind:

3.1.1 Chain Rule

In (6), we showed a function $u(x,t)$ can be represented by a combination of trial functions and corresponding coefficients. Here, the relationship can be rewritten as the following form:

$$u(x)|_{\Omega^e} = \sum_{i=0}^{N} u_i^e \pi_i(\xi), \quad \xi \in [-1,1]. \tag{30}$$

As such in the derivative of u with respect to x it is important to consider the chain rule:

$$\left.\frac{\partial u(x)}{\partial x}\right|_{\Omega^e} = \sum_{i=0}^{N} u_i^e \frac{\partial \pi_i}{\partial \xi}\frac{\partial \xi}{\partial x}, \quad \xi \in [-1,1]. \tag{31}$$

3.1.2 Change of Integration Domain

Due to the mapping, the integration domain to evaluate the weak form is changed. Consequently, the Jacobian determinant must be introduced as a scaling factor, which for this case is:

$$J(\xi) = \frac{\partial x}{\partial \xi} = \frac{h}{2}. \tag{32}$$

Introducing the previous notation into the diffusive term in (28), the following is obtained:

$$\int_{\Omega^e} (\frac{\partial v}{\partial x}\frac{\partial u}{\partial x})dx = \sum_{i=0}^{N}\sum_{j=0}^{N} v_i^e (\int_{\hat{\Omega}} \frac{\partial \pi_i}{\partial \xi}\frac{\partial \pi_j}{\partial \xi}(\frac{\partial \xi}{\partial x})^2 J(\xi)d\xi) u_j^e. \tag{33}$$

This yields a similar form as the one shown in the second term if formulation of the Galerkin method (8); however, additional terms that account for the mapping and geometry of the element in physical domain are included now. For the stiffness matrix in 1D, the geometry term becomes a constant since:

$$(\frac{\partial \xi}{\partial x})^2 J(\xi) = (\frac{2}{h})^2 \frac{h}{2} = \frac{2}{h}. \tag{34}$$

Substitute (34) into (33) and rewrite it into matrix form, yields:

$$\int_{\Omega^e} (\frac{\partial v}{\partial x}\frac{\partial u}{\partial x})dx = (\underline{v}^e)^T K^e \underline{u}^e, \tag{35}$$

where

$$K^e[i,j] = \frac{2}{h}\int_{\hat{\Omega}} \frac{\partial \pi_i}{\partial \xi}\frac{\partial \pi_j}{\partial \xi} d\xi, \quad \hat{\Omega} \in [-1,1]. \tag{36}$$

Up to the point, we have assumed that all integrals are evaluated analytically. As we have seen, within each elemental domain we want to evaluate integrals of the form

$$\int_{-1}^{1} f(\xi) d\xi. \tag{37}$$

The form of $f(\xi)$ is, however, problem specific and therefore we need an automated way to evaluate such integrals. This suggests the use of numerical integration , and particularly a quadrature rule. The fundamental concept is the approximation of the integral by a finite summation of the form

$$\int_{-1}^{1} f(\xi) d\xi = \sum_{i=0}^{Q-1} \rho_i f(\xi_i) + R(f), \tag{38}$$

where ρ_i are specified weights and ξ_i represent Q distinct points in the interval $-1 \leq \xi_i \leq 1$, and $R(f)$ is the residual. In particular, we introduce Gauss-Lobatto-Legendre quadrature here:

$$\xi_i = \begin{cases} -1 & i = 0, \\ \xi_{i-1}^{1,1} & i = 1, 2...Q-2, \\ 1 & i = Q-1, \end{cases} \tag{39}$$

$$\rho_i = \frac{2}{N(N+1)} \frac{1}{[L_N(\xi_i)]^2}, \tag{40}$$

$$R(f) = 0 \qquad \text{if} \quad f(\xi) \in \mathbf{P}_{2Q-3}([-1,1]). \tag{41}$$

In the above formulae $L_Q(\xi)$ is the Legendre polynomial. The zeros of the Jacobi polynomial $\xi^{\alpha,\beta}$ (here $\alpha = \beta = 1$, it represents the zeros of $L'_Q(\xi)$) do not have an analytic form and commonly the zeros and weights are tabulated. Tabulation of data can lead to copying errors and therefore a better way to evaluate the zeros is the use of a numerical algorithm such as a Newton-Raphson technique. Having determined the zeros, the weights can be evaluated from this technique. This is done by generating the Legendre polynomial from a recursion relationship (Find description on Wiki here). The GLL quadrature is accurate if the order of polynomial integrand is no more than $2Q-3$.

Then it is possible to numerically integrate and differentiate the terms such that the final form of the matrix is obtained:

$$K^e[i,j] = \frac{2}{h} \sum_{m=0}^{N} \rho_m D_{N,mi}^{(1)} D_{N,mj}^{(1)}, \tag{42}$$

where D_N is the differentiation matrix obtained as follows:

$$D_{N,ij}^{(1)} = \left. \frac{d\pi_j}{d\xi} \right|_{\xi=\xi_i} = \begin{cases} \frac{L_N(\xi_i)}{L_N(\xi_j)} \frac{1}{\xi_i - \xi_j} & i \neq j, \\ -\frac{(N+1)N}{4} & i = j = 0, \\ \frac{(N+1)N}{4} & i = j = N, \\ 0 & \text{otherwise .} \end{cases} \tag{43}$$

Following the same procedure as for the stiffness matrix, the mass and advection matrices can also be obtained, keeping in mind that $\pi_i(x_j) = \delta_{ij}$:

$$M_{ij}^e := \frac{h}{2}\rho_i \delta_{ij}, \tag{44}$$

$$C^e[i,j] = \sum_{m=0}^{N} \rho_m \pi_{mi} D_{N,mj}^{(1)} = \rho_i D_{N,ij}^{(1)}. \tag{45}$$

Extra bit: What about non-linearities? The non-linear term in the Navier-Stokes equation and Burger's equation are treated in a similar way as that of the constant convective term. For the case with non-linear term $u\frac{du}{dx}$, the same procedure is followed, the orthogonality of the basis functions used and taking advantage of the quadrature rules, the following expression is obtained:

$$C^e[i,j] = \rho_i u_i^e D_{N,ij}^{(1)}. \tag{46}$$

Note that the advective matrix depends on u which needs to be taken into consideration for time stepping. In a explicit method, the advective matrix would need to be evaluated at each iteration.

3.1.3 Applying Boundary Conditions: Homogeneous Dirichlet BC

- If $u \in X_0^N$, the basis coefficients on the boundary are zero:

$$u_0 = u_N = 0. \tag{47}$$

- In our definition of global assemble matrix, the index is ranged from 0 to n on the global vectors v and u.
- Therefore we need to construct a restriction matrix R and prolongation matrix R^T that eliminate u_0 and u_{n+1}. Note we can generate a matrix $R^T R$ which can map a function from X^N to X_0^N.

3.1.4 Neumann Boundary Condition

Let's consider the Poisson equation as the example for simplicity:

$$-\frac{d^2u}{dx^2} = f(x), \tag{48}$$

$$u(-1) = 0, \frac{du}{dx}(1) = g. \tag{49}$$

After applying Galerkin method, integrating by parts and applying Neumann BC on the right boundary and Dirichlet BC on the left boundary, we can end up with following equation:

$$\int_{-1}^{1} \frac{dv}{dx}\frac{du}{dx}dx = \int_{-1}^{1} vf(x)dx + v(1)g. \tag{50}$$

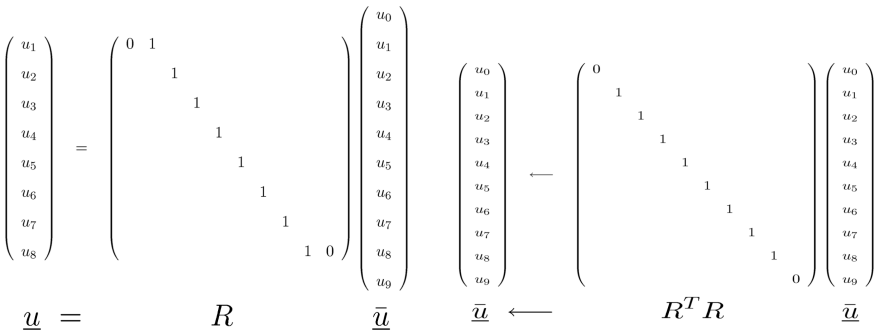


Figure 3: Left: applying the homogeneous Dirichlet BC to global basis coefficients. Right: the way to map a function from X^N to X_0^N.

We can continue with the derivation and reach to the linear system:

$$Au = F. \tag{51}$$

Rewrite it into the matrix format:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & & & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix}; \quad = \begin{pmatrix} \rho_1 f_1 \\ \rho_2 f_2 \\ \vdots \\ \rho_N f_N \ + \ g \end{pmatrix}.$$

We can see only the last term in the forcing vector is modified due to the Neumann BC.

3.1.5 Time Integration

We have linear system:

$$u_t = \lambda u, \tag{52}$$

where $\lambda \in \mathbb{C}$ is a system parameter which mimics the eigenvalues of linear systems of differential equations. The equation is stable if $\text{Real}(\lambda) \leq 0$. In this case the solution is exponentially decaying. ($lim_{t\to\infty} u(t) = 0$).

For time integration, the explicit Euler method is used here:

$$u^{n+1} = u^n + h\lambda u^n = (1 + h\lambda)u^n = (1 + h\lambda)^{n+1} u^0, \tag{53}$$

where $h = \Delta t$. So for explicit Euler method, for a solution to be stable as $t \to \infty$, condition $|1 + \lambda h| < 1$ should be satisfied. In the complex plane, we can find the optimal h by forcing every λh inside the neutral stability curve.

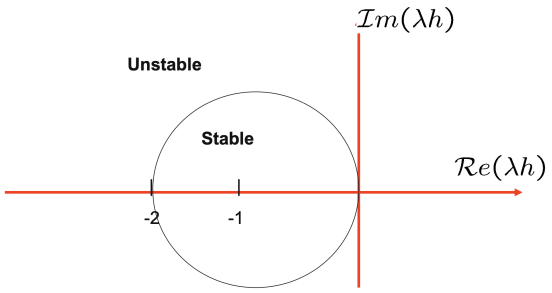


Figure 4: Neutral stability curve for explicit Eular method.

For the illustration purpose, the advection-diffusion problem is solved with Dirichlet BCs and Euler backwards time integration method. The final result with polynomial order $p = 32$, total time $t = 0.1$ and initial condition can be compared in figure 5.

Figure 5: Final result at $t = 0.1$ with $p = 32$, with Dirichlet BC applied on both boundaries.

3.2 Relationship between Modal and Nodal Bases

The key historical distinction between a spectral element and a p-type finite element is whether the expansion is nodal or modal. As previously mentioned, the Galerkin approximation is the minimising solution independent of the polynomial approach so if there are no integration errors then the methods are mathematically equivalent. However each approach does have different numerical properties in terms of efficiency of implementation, ability to vary the polynomial order and the conditioning of the global matrix systems. The details can be found in Karniadakis [2005].

3.2.1 Nodal Approach

Till now, we used the Lagrange interpolations at GLL points to build the trail function. The elemental mass matrix using this expansion is full, if we evaluate the inner product $M[p][q] = (\pi_p, \pi_q)$ exactly. If, however, we use the Gauss-Legendre-Lobatto quadrature rule corresponding to the same choice of nodal points on which the expansion was defined, the mass matrix is diagonal due to

the Kronecker delta property:

$$M[p][q] = (\pi_p, \pi_q) \approx \sum_{i=0}^{Q} \rho_i \pi_p(\xi_i)\pi_q(\xi_i) = \sum_{i=0}^{Q} \rho_i \delta_{pi}\delta_{qi} = \rho_p \delta_{pq}, \tag{54}$$

where ρ_i are the weights for the Gauss-Legendre-Lobatto rule using $Q+1$ points. The quadrature rule using $Q+1$ points is exact only for polynomials of order $2Q-1$. The diagonal components of the elemental mass matrix using the quadrature rule are equal to the row sum of the elemental mass matrix using exact integration. Summing the rows and using this as the entry of a diagonal matrix is common practice in finite elements and is known as lumping the mass matrix. In the standard finite element case, lumping the mass matrix is an approximation, but in the spectral element case this lumping has a direct equivalence.

$$\sum_{q=0}^{Q} M[p][q] = \sum_{q=0}^{Q} (\pi_p, \pi_q) = (\pi_p(\xi), \sum_{q=0}^{Q} \pi_q(\xi)). \tag{55}$$

We note that the sum of the Lagrange basis over all modes is simply '1': $\sum_{q=0}^{Q} \pi_q(\xi) = 1$, and so the sum of the p^{th} row becomes:

$$\sum_{q=0}^{Q} M[p][q] = (\pi_p(\xi), \sum_{q=0}^{Q} \pi_q(\xi)) = (\pi_p(\xi), 1). \tag{56}$$

The last term is defined as the weight corresponding to the p^{th} point in the Gauss-Lobatto-Legendre quadrature rule using $Q+1$ points.

As a final point, we note that the elemental Laplacian matrix using the spectral element expansion does not have any notable properties of this type and is full for all choices of quadrature order.

3.2.2 Modal Approach

Thanks to the orthogonality of Legendre polynomials:

$$\int_{-1}^{1} P_m(x)P_n(x)dx = \frac{2}{2n+1}\delta_{mn}, \tag{57}$$

we can construct the elemental mass matrix easily using this relationship. It is worth to note that more generally, the sequence of polynomial functions $\{p_k\}_{k=0}^{\infty}$, where the degree of p_k is equal to k, forms a system of orthogonal polynomials with respect to ω if:

$$(p_k, p_l)_{L_w^2} := \int_{-1}^{1} \omega(x)p_k(x)p_l(x)dx = \gamma_k \delta_{kl}, \tag{58}$$

where L_w^2 is the Hilbert space of Lebesgue-weighted-square-integrable functions and ω denotes any non-negative integrable weight function: $\gamma_k := \|p_k\|_{L_w^2}^2$. More detailed description can be found in Deville et al. [2004].

Now, we can plot the structure of mass matrices via different approaches shown in figure 6.

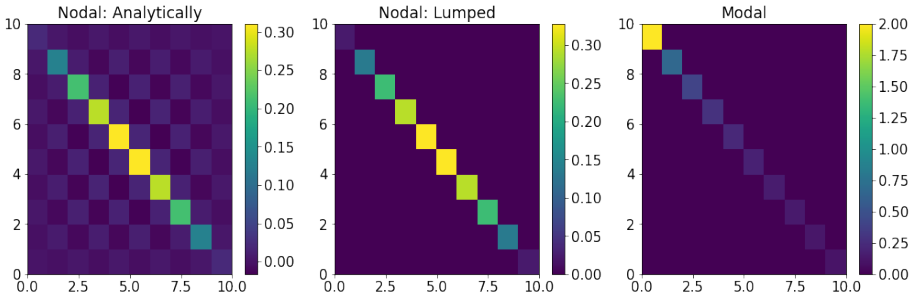


Figure 6: The structure of mass matrix via different approaches.

4 Spectral Element Method in 1D - Multiple Elements

The following section is based on chapter 2 and 5 in Deville et al. [2004]. The subdivision of the physical domain Ω into multiple elements has been stated in section 3 , but the practical application of it has not. To start, it is very important to analyze continuity between the elements.

4.1 Continuity (Scatter-Gather)

One of the advantages of using Lagrangian basis in SEM is the fact that function continuity is enforced by equating coincident nodal values, that is:

$$x_i^e = x_{\hat{i}}^{\hat{e}} \Rightarrow u_i^e = u_{\hat{i}}^{\hat{e}}. \tag{59}$$

Defining $\mathcal{N}$ as the number of distinct nodes in Ω, the previous equation represents $(N+1)^d E - \mathcal{N}$ constrains on the choice of the local nodal value u_i^e.

The constraint can be recasted in matrix form, keeping in mind that the global nodal values can be represented in vector form $\underline{u}$ and so can the local element wise nodal values $\underline{u}^e$. If a collection of element nodal values is defined as a collection of the local vectors $\underline{u}_L$ such that:

$$\underline{u}_L = \begin{pmatrix} \underline{u}^1 \\ \underline{u}^2 \\ \cdots \\ \underline{u}^e \\ \cdots \\ \underline{u}^E \end{pmatrix} = \begin{pmatrix} u_0^1 \\ u_1^1 \\ \cdots \\ u_0^e \\ u_1^e \\ \cdots \\ u_N^E \end{pmatrix}. \tag{60}$$

If u is continuous, then there exists a Boolean connectivity matrix Q that maps $\underline{u}$ to $\underline{u}_L$ ensuring that the constraints are satisfied. Such that:

$$\underline{u}_L = Q\underline{u}. \tag{61}$$

The matrix Q is known as the **Scatter** operator and its action is to **copy** entries from the global domain into the local one.

An additional matrix Q^T is also defined, which is known as the **Gather** operator:

$$\underline{v} = Q^T \underline{u}_L. \tag{62}$$

The function of this operation is to **sum entries** from corresponding nodes. It is then important to keep in mind that $\underline{v} \neq \underline{u}$

For instance, the shape of Q in 1D for 2 elements and a basis of order 1 is as shown in (63).

$$Q = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{63}$$

With this definition, $\underline{u}_L$ can be found by applying the operator to $\underline{u}$, as shown in (64)

$$\underline{u}_L = \begin{pmatrix} u_0^1 \\ u_1^1 \\ u_0^2 \\ u_1^2 \end{pmatrix} = \begin{pmatrix} u_0 \\ u_1 \\ u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} = Q\underline{u}. \tag{64}$$

It is clear that using the scatter operator, the values from the global matrix are copied to the local one. For example, the global value u_1, which is the value of the common grid between elements 1 and 2, is copied to local coefficients u_1^1 and u_0^2.

A gather-scatter operation can be performed in sequence such that $\Sigma' = QQ^T$. Denoted as **Direct Stiffness Summation**. Defined, for example by Deville et al. [2004], as a local to local transformation that amounts to summing shared interface variables and redistributing them to their original locations, leaving the interior nodes unchanged.

Note: In practice, the matrices Q and Q^T are never constructed. Rather, their actions are implemented using indirect addressing.

4.2 Construction of a Global Basis

For simplicity the process will be initially shown for the diffusion term. It has been stated that for a given element, the following expression holds:

$$\int_{\Omega^e} (\frac{\partial v}{\partial x}\frac{\partial u}{\partial x})dx = (\underline{v}^e)^T K^e \underline{u}^e. \tag{65}$$

The extension of (65) to the whole domain is given in:

$$\int_{\Omega} (\frac{\partial v}{\partial x}\frac{\partial u}{\partial x})dx = \sum_{e=1}^{E} (\underline{v}^e)^T K^e \underline{u}^e = (\underline{v}_L)^T K_L \underline{u}_L. \tag{66}$$

Where $\underline{u}_L$ and $\underline{v}_L$ are the local vectors as previously defined and K_L is known as the **Unassembled Stiffness Matrix**, which has the following form:

$$K_L = \begin{pmatrix} K^1 & 0 & 0 & 0 \\ 0 & K^2 & 0 & 0 \\ 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & K^E \end{pmatrix}. \tag{67}$$

Since the previous finding holds for continuous u, v, based on the discussion in the previous section, it is possible to say that there exists Boolean matrices and vectors $\underline{u}, \underline{v}$ such that $\underline{u}_L = Q\underline{u}$ and $\underline{v}_L = Q\underline{v}$. Applying this to (66) yields:

$$\int_\Omega (\frac{\partial v}{\partial x}\frac{\partial u}{\partial x})dx = (\underline{v})^T Q^T K_L Q \underline{u}, \tag{68}$$

where $K = Q^T K_L Q$ is known as the **Assembled Stiffness Matrix**. Following the same reasoning, it is also possible to find an expression for other operators:

$$M = Q^T M_L Q. \tag{69}$$

$$C = Q^T C_L Q. \tag{70}$$

The problem is then solved in the global domain as stated in (71). The solution for the 1D advection diffusion problem can be seen in figure 7.

$$M\frac{d\underline{u}}{dt} = -K\underline{u} - C\underline{u}. \tag{71}$$

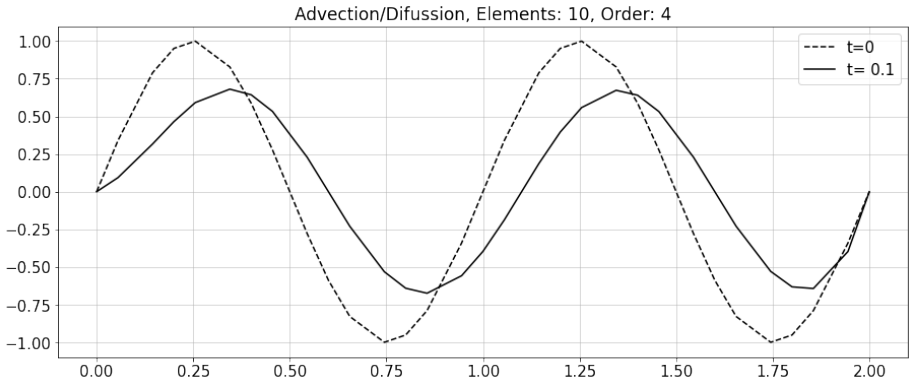


Figure 7: Numerical solution of the 1D advection-diffusion in multiple subdomains

4.2.1 Note of Caution - Discontinuous terms.

The use of Q to map from the local to global is restricted to continuous functions. If in the original equation, there exists a source term f such that:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} + f, \tag{72}$$

an additional term is required to be included in the discretization procedure which has the following form:

$$\int_\Omega (vf)\, d\mathbf{x}. \tag{73}$$

Performing the same procedure as for the other terms, i.e expressing f as $f(\mathbf{x}) = \sum_{i=0}^{N} f_i \pi_i(\xi)$, constructing element matrix and then the global one yields (74).

$$\int_\Omega (vf)\, d\mathbf{x} = (\underline{v})^T Q^T M_L \underline{f}_L. \tag{74}$$

For this case, the expression $\underline{f}_L = Q\underline{f}$ can not be applied *a priori* since the source terms might be discontinuous across the boundaries.

4.2.2 Working in Local terms

If all the scatter and gather operations are expressed explicitly, (71) can be expressed as:

$$Q^T M_L Q \frac{d\underline{u}}{dt} = -Q^T K_L Q \underline{u} - Q^T C_L Q \underline{u}. \tag{75}$$

Multiplying the global formulation by Q yields:

$$\Sigma' M_L \frac{d\underline{u}_L}{dt} = -\Sigma' K_L \underline{u}_L - \Sigma' C_L \underline{u}_L. \tag{76}$$

This new formulation brings some advantages: It is now possible to work on each element locally, evaluating operations without the need to map from global to local terms. Additionally, the gather and scatter operations are now performed in sequence, which for parallel processes reduces the overhead produced due to data communication.

5 Spectral Element Method - Multiple Dimensions

For the extension of the formulation to multiple dimensions, it is beneficial to explain some concepts first.

5.1 Definition: Tensor-Product

Let A and B be $k \times l$ and $m \times n$ matrices. A $km \times ln$ matrix C of the following form can be defined:

$$C = \begin{pmatrix} a_{11}B & a_{12}B & \cdots & a_{1l}B \\ a_{21}B & a_{22}B & \cdots & a_{2l}B \\ \cdots & \cdots & \cdots & \cdots \\ a_{k1}B & a_{k2}B & \cdots & a_{kl}B \end{pmatrix}. \tag{77}$$

For this case, C is said to be the tensor product of A and B and can be denoted as:

$$C = A \otimes B. \tag{78}$$

The definition of tensor-product is very important, as many operations in the multidimensional problems using SEM end up acquiring this form, and using tensor-products allows simplifications on the application of linear operators to the quantities.

5.2 Definition: Matrix-Free Formulation - Explained in 2D

Consider the next representation of u in the reference domain $(x, y) \in \hat{\Omega} := [-1, 1]^2$

$$u(x, y) = \sum_{i=0}^{M} \sum_{j=0}^{N} u_{ij} \pi_{M,i}(x) \pi_{N,j}(y). \tag{79}$$

It is possible to use a **vector representation** of the coefficients as follows:

$$\underline{u} := (u_1, u_2, \cdots, u_l, \cdots, u_{\mathcal{N}})^T = (u_{00}, u_{10}, \cdots, u_{IJ}, \cdots, u_{MN})^T, \tag{80}$$

where $\mathcal{N} = (M + 1)(N + 1)$ is the number of basis coefficients and the mapping $l = 1+i+(M+1)j$ translates the two-index coefficient representation to standard vector form with the leading coefficient advancing more rapidly.

The derivative with respect to x of $u(x, y)$ at the GLL points is:

$$w_{pq} = \left.\frac{\partial u}{\partial x}\right|_{\xi_{M,p}, \xi_{N,q}} = \sum_{i=1}^{M} \sum_{j=1}^{N} u_{ij} \pi'_{M,i}(\xi_{M,p}) \pi_{N,j}(\xi_{N,q}) = \sum_{i=1}^{M} u_{iq} \pi'_{M,i}(\xi_{M,p}), \tag{81}$$

Which can be represented in the following matrix-vector product form:

$$\underline{w} = D_x \underline{u} = \begin{pmatrix} \hat{D}_x & 0 & 0 & 0 \\ 0 & \hat{D}_x & 0 & 0 \\ 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \hat{D}_x \end{pmatrix} \begin{pmatrix} u_{00} \\ u_{10} \\ \cdots \\ u_{MN} \end{pmatrix}. \tag{82}$$

Here, $\hat{D}_x$ is the one-dimensional derivative matrix associated with the GLL points, which is applied to each row of the coefficients matrix. The y derivative would then be defined following a similar process but applying the matrix $\hat{D}_y$ to each column. The general derivative operators can be defined very easily by using the tensor-product definitions, which yields:

$$D_x = I \otimes \hat{D}_x \quad D_y = \hat{D}_y \otimes I, \tag{83}$$

where I is the identity matrix. In the evaluation of a multidimensional PDE, matrix-vector multiplications of the form $\underline{v} = (A \otimes B)\underline{u}$ are very common and normally defined as:

$$v_{ij} = \sum_{l=1}^{n}\sum_{k=1}^{m} a_{jl} b_{ik} u_{kl} \quad , i = 1, \cdots, m \quad j = 1, \cdots, n. \tag{84}$$

In vector form, the same expression can be recast thanks to the associative property of tensor product:

$$C\underline{u} = (A \otimes I)(I \otimes B)\underline{u}. \tag{85}$$

In practice, however, the matrix C does not need to be formed, as one can evaluate the effect of matrix C on the vector as follows:

- Compute $\underline{w} = (I \otimes B)\underline{u}$

$$w_{ij} = \sum_{k=1}^{m} b_{ik} u_{kj}, \quad i = 1, \cdots, m \quad j = 1, \cdots, n. \tag{86}$$

- Then compute $\underline{v} = (A \otimes I)\underline{w}$

$$v_{ij} = \sum_{l=1}^{m} a_{jl} w_{il} = \sum_{l=1}^{m} w_{il} a_{lj}^{T}, \quad i = 1, \cdots, m \quad j = 1, \cdots, n. \tag{87}$$

This way, the evaluations can be done with a fewer number of operations.

5.2.1 Matrix-Matrix Operators

It is important to note that if the vector $\underline{u}$ is viewed as a matrix U such that $\{U\}_{ij} = u_{ij}$, then the tensor-products can be recasted in the following form:

$$(A \otimes B)\underline{u} = BUA^{T}. \tag{88}$$

These properties are extremely beneficial to the performance of SEM solvers.

5.3 Obtaining the Element Operators

The are many similarities with the process to get the element operators in 1D to extended dimensions. Note that in the SEM, each element on the physical domain $\mathbf{x} \in \Omega$ must be mapped to the reference element in the computational domain $\mathbf{r} \in \hat{\Omega}$.

Usually iso-parametric mappings are employed, for instance for a 2D case:

$$x(r,s)|_{\Omega^e} = \sum_{i=0}^{N}\sum_{j=0}^{N} x_{ij}^{e} \pi_i(r)\pi_j(s), \quad (r,s) \in (-1,1)^2. \tag{89}$$

The representation of a variable in the physical domain is then expressed as a function of Lagrange interpolants which vary in the computational domain,

$$u(x,y) = \sum_{i=0}^{M}\sum_{j=0}^{N} u_{ij} \pi_{M,i}(r)\pi_{N,j}(s). \tag{90}$$

Equal care must be taken with certain aspects, just as in the one-dimensional SEM.

5.3.1 Chain Rule

Derivatives taken with respect to variables in the original domain must take into consideration the chain rule:

$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial r}\frac{\partial r}{\partial x} + \frac{\partial u}{\partial s}\frac{\partial s}{\partial x}, \tag{91}$$

or in $\mathcal{R}^d$:

$$\frac{\partial u}{\partial x_k} = \sum_{k=1}^{d} \frac{\partial u}{\partial r_i}\frac{\partial r_i}{\partial x_k}. \tag{92}$$

5.3.2 Change of Integration Domain

Changing the integration domain, from the physical to the reference, brings with itself the necessity to include the appropriate scaling factor. Thus, it is needed to recall the definition of the **Jacobian Determinant**:

$$J(\mathbf{r}) = \det \frac{\partial x_i}{\partial r_j} = det \begin{pmatrix} \frac{\partial x_1}{\partial r_1} & \cdots & \frac{\partial x_1}{\partial r_d} \\ \cdots & \cdots & \cdots \\ \frac{\partial x_d}{\partial r_1} & \cdots & \frac{\partial x_d}{\partial r_d} \end{pmatrix}. \tag{93}$$

The discretization of the diffusive term is presented in order to illustrate the process. The d-dimensional form of the bi-linear diffusive term for a given element is written as:

$$\mathcal{A}(v,u) = \sum_{k=1}^{d} \int_{\Omega^e} \frac{\partial v}{\partial x_k}\frac{\partial u}{\partial x_k} d\mathbf{x}. \tag{94}$$

The chain rule is applied to the derivatives and a change of variable is done in the integration domain from Ω^e to $\hat{\Omega}$:

$$\mathcal{A}(v,u) = \sum_{i=1}^{d}\sum_{j=1}^{d} \int_{\hat{\Omega}} \frac{\partial v}{\partial r_i}\Big(\sum_{k=1}^{d} \frac{\partial r_i}{\partial x_k}\frac{\partial r_j}{\partial x_k} J(\mathbf{r})\Big)\frac{\partial u}{\partial r_j} d\mathbf{r}. \tag{95}$$

Following this, all the metrics and geometry-associated variables can be grouped together into a variable:

$$\mathcal{G}_{ij}(\mathbf{r}) = \sum_{k=1}^{d} \frac{\partial r_i}{\partial x_k}\frac{\partial r_j}{\partial x_k} J(\mathbf{r}), \quad 1 \leq i,j \leq d. \tag{96}$$

The integrals in the diffusion term can be numerically evaluated using the quadrature rules, such that the following expression is obtained:

$$\mathcal{A}(v,u) = \sum_{i=1}^{d}\sum_{j=1}^{d}\sum_{klm}^{N}\left[\frac{\partial v}{\partial r_i}\mathcal{G}_{ij}(\mathbf{r})\frac{\partial u}{\partial r_j}\right]_{(\xi_k,\xi_l,\xi_m)}\rho_k\rho_l\rho_m. \tag{97}$$

All that is left is to expand u and v in terms of the Lagrange polynomials. In this case, the differentiation procedures follows similar to the one already shown. An example for a derivative in one of the directions is the following:

$$\frac{\partial u}{\partial r_1}|_{(\xi_k,\xi_l,\xi_m)} = \sum_{p=0}^{N}\hat{D}_{kp}u_{plm} = (I\otimes I\otimes\hat{D})\underline{u}, \quad k,l,m\in\{0,\cdots,N\}^3. \tag{98}$$

Grouping the geometric terms and quadrature weights into a set of d^2 diagonal matrices $G_{ij}, i,j\in 1,\cdots,d^2$ such that:

$$(G_{ij})_{\hat{k}\hat{k}} = [\mathcal{G}_{ij}]_{(\xi_k,\xi_l,\xi_m)}\,\rho_k\rho_l\rho_m, \tag{99}$$

with $\hat{k} = 1+k+(N+1)l+(N+1)^2m$ where $k,l,m\in\{0,\cdots,N\}^3$and defining:

$$D_1 = (I\otimes I\otimes\hat{D}), \quad D_2 = (I\otimes\hat{D}\otimes I), \quad D_3 = (\hat{D}\otimes I\otimes I). \tag{100}$$

In a final step, it is possible to achieve one compact form of the energy inner-product:

$$\mathcal{A}(v,u) = \underline{v}^T\begin{pmatrix}D_1\\D_2\\D_3\end{pmatrix}^T\begin{pmatrix}G_{11}&G_{12}&G_{13}\\G_{21}&G_{22}&G_{23}\\G_{31}&G_{32}&G_{33}\end{pmatrix}\begin{pmatrix}D_1\\D_2\\D_3\end{pmatrix}\underline{u} = \underline{v}^TD^TGD\underline{u}. \tag{101}$$

From this expression, the element stiffness matrix is defined as $K^e = D^TGD$. The mass matrix is obtained by an extension of the previously shown procedure and yields (in 3D) the following diagonal form:

$$M_{\hat{i}\hat{i}} = J(\xi_k,\xi_l,\xi_m)\rho_i\rho_j\rho_k \quad \hat{i} = 1+i+(N+1)j+(N+1)^2k. \tag{102}$$

6 Implementation: Matrix-Free Formulation

What follows is based on sections 8.1 - 8.3 in Deville et al. [2004]. In the previous sections we learned how tensor-products are formally represented on paper. Tensor-products are required for evaluation of the spectral operators, but it is not feasible to form large assembled global matrices, because global matrix operations are memory- and performance-wise expensive. Consider the example of computing the diffusion term using the element stiffness matrix (K^e). The overall operation can be represented by the following:

- **Local matrix-vector products**: which is a local operation within an element,
$$\mathcal{A}^{\urcorner} = \underline{v}^{eT} K^e \underline{u}^e ,$$
which in turn is a vectorizable algorithm and thus its performance is CPU bounded; and
- **Direct stiffness summation**: which involves consecutive gather (Q^T) and scatter (Q),
$$\mathcal{A}_{\mathcal{L}} = Q Q^T \mathcal{A}^{\urcorner} ,$$
which in turn is dominated by communications and thus its performance is memory bandwidth bounded.

By splitting the operation to a two-step procedure allows Nek5000 to fine-tune and optimize the overall performance.

6.1 Short detour: Vectorization and performance

Let us have a quick look at how to avoid common pitfalls to favour vectorization. Compilers are smart and capable of vectorizing loops, provided:

- Dependencies are avoided, e.g.: backward substitution of a tri-diagonal solve,

```
do i = n-1, 1, -1
   x(i) = (b(i) - u(i) * x(i+1)) / d(i)
enddo
```

would be *much slower* than something like,

```
do i = 1, n
   x(i) = a(i) + b(i)
enddo
```

- Branching avoided: subroutines, functions, **if** statements, I/O;
- Data locality: strides in memory are aligned with cache lines.

6.2 Example of 3D gradient computation in Nek5000

To demonstrate all these concepts we take a look at how a gradient of any array is computed in Nek5000. Let us recall that derivatives in each directions are:

- defined as tensor products $(I \otimes I \otimes D)\underline{u}^e$, etc.,
- implemented as matrix-matrix multiplications $\Sigma_p D_{ip} u^e_{pjk}$.

This ensures data locality and vectorization. To compute the gradient of an array, the function gradm1 (see listing 1) is often employed. This ultimately calls the matrix-matrix multiplication mxm subroutine.

For the 3-D case the gradient is *locally* evaluated with a matrix of shape (lx1 , lx1, lx1). Note that lx1 = m1 = n+1 is the polynomial order plus 1 to include element boundaries. Within the subroutine local_grad3 the derivative matrices D and D^T are multiplied against a single element of the u array using subroutine mxm.

```
      subroutine gradm1(ux,uy,uz,u)
c
c     Compute gradient of T -- mesh 1 to mesh 1 (vel. to vel.)
c     Output: ux, uy, uz  | Input: u
c
      include 'SIZE'
      include 'DXYZ'
      include 'GEOM'
      ...
c
      parameter(lxyz=lx1*ly1*lz1)
      real ux(lxyz,1),uy(lxyz,1),uz(lxyz,1),u(lxyz,1)

      ! Scratch arrays
      common /ctmp1/ ur(lxyz),us(lxyz),ut(lxyz)

      integer e

      nxyz = lx1*ly1*lz1
      ntot = nxyz*nelt

      n = lx1-1
      do e=1,nelt
         if (if3d) then
            call local_grad3(ur,us,ut,u,n,e,dxm1,dxtm1)
            do i=1,lxyz
               ux(i,e) = jacmi(i,e)*(ur(i)*rxm1(i,1,1,e)
     $                             + us(i)*sxm1(i,1,1,e)
     $                             + ut(i)*txm1(i,1,1,e) )
               uy(i,e) = jacmi(i,e)*(ur(i)*rym1(i,1,1,e)
     $                             + us(i)*sym1(i,1,1,e)
     $                             + ut(i)*tym1(i,1,1,e) )
               uz(i,e) = jacmi(i,e)*(ur(i)*rzm1(i,1,1,e)
     $                             + us(i)*szm1(i,1,1,e)
     $                             + ut(i)*tzm1(i,1,1,e) )
         else
            ...
      enddo
c
      return
      end
```

Listing 1: Subroutine to compute gradient of an array u. Code snippet from Nek5000/core/navier5.f

```
      subroutine local_grad3(ur,us,ut,u,N,e,D,Dt)
c     Output: ur,us,ut         Input:u,N,e,D,Dt
      real ur(0:n,0:n,0:n),us(0:n,0:n,0:n),ut(0:n,0:n,0:n)
      real u (0:n,0:n,0:n,1)
      real D (0:n,0:n),Dt(0:n,0:n)
      integer e
c
      m1 = n+1
      m2 = m1*m1
c
      call mxm(d  ,m1,u(0,0,0,e),m1,ur,m2)
      do k=0,n
         call mxm(u(0,0,k,e),m1,dt,m1,us(0,0,k),m1)
      enddo
      call mxm(u(0,0,0,e),m2,dt,m1,ut,m1)
c
      return
      end
```

Listing 2: Subroutine to compute the *local gradient* of a spectral element. Code snippet from Nek5000/core/navier5.f

Derivative operators The matrices D and D^T are initialized once and stored in the include file DXYZ within a common block. In listing 3 we find the code responsible for generating these operators.

Matrix multiplication The subroutine mxm is responsible for matrix-matrix multiplication and has several optimized implementations in listing 4. If n2 = 4 and no preprocessor macros are active then mxf4 would be called by mxmf2 (see listing 5).

6.2.1 Back to `local_grad3`

Now that we understand how the subroutine `mxm` works and how the derivative operators are initialized, we turn our attention back to listing 2. Here, all three derivatives on a 3-D matrix $u_{(m,m,m)}$ one direction at a time. To do so the matrix is reshaped such that the leading dimension is the direction along which the derivative operator would be multiplied.

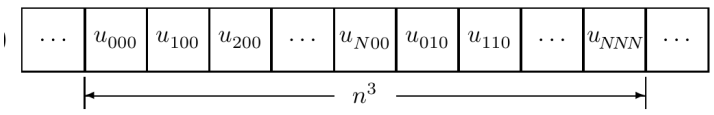


Figure 8: Interpretation of 3D data array as a contiguous vector in memory. Source: Deville et al. [2004].

1st `mxm` $u_r = (I \otimes I \otimes \hat{D}) = \hat{D}_{(m,m)} u_{(m,m^2)}$. See figure 9.

```
subroutine local_grad3(ur,us,ut,u,N,e,D,Dt)
Output: ur,us,ut         Input:u,N,e,D,Dt
...
m1 = n+1
m2 = m1*m1
```

```
      subroutine get_dgll_ptr (ip,mx,md)
c
c     Get pointer to GLL-GLL interpolation dgl() for pair (mx,md)
c
      include 'SIZE'

c     dgradl holds GLL-based derivative / interpolation operators

      parameter(ldg=lxd**3,lwkd=4*lxd*lxd)
      common /dgradl/ d(ldg),dt(ldg),dg(ldg),dgt(ldg),jgl(ldg),jgt
     (ldg)
     $              , wkd(lwkd)

      if (ip.eq.0) then
      ...
         call gen_dgll(d(ip),dt(ip),md,mx,wkd)
      ...
      return
      end
c-----------------------------------------------------------------------
      subroutine gen_dgll(dgl,dgt,mp,np,w)
c
c     Generate derivative from np GL points onto mp GL points
c
c        dgl  = derivative matrix, mapping from velocity nodes to
     pressure
c        dgt  = transpose of derivative matrix
c        w    = work array of size (3*np+mp)
c
c        np   = number of points on GLL grid
c        mp   = number of points on GL  grid
c
c
c
      real dgl(mp,np),dgt(np*mp),w(1)
c
c
      iz = 1
      id = iz + np
c
      call zwgll (w(iz),dgt,np)  ! GL points
      call zwgll (w(id),dgt,mp)  ! GL points
c
      ndgt = 2*np
      ldgt = mp*np
      call lim_chk(ndgt,ldgt,'ldgt ','dgt  ','gen_dgl  ')
c
      n  = np-1
      do i=1,mp
         call fd_weights_full(w(id+i-1),w(iz),n,1,dgt) ! 1=1st
     deriv.
         do j=1,np
            dgl(i,j) = dgt(np+j)                       !
     Derivative matrix
         enddo
      enddo
c
      call transpose(dgt,np,dgl,mp)
c
      return
      end
```

Listing 3: Subroutines to compute and initialize the derivative operators. Code snippet from Nek5000/core/navier1.f

```
      subroutine mxm(a,n1,b,n2,c,n3)
c
c     Compute matrix-matrix product C = A*B
c     for contiguously packed matrices A,B, and C.
c
      real a(n1,n2),b(n2,n3),c(n1,n3)
c
      ...

#ifdef BGQ
      ! Uses IBM Blue Gene specific subroutines
      ...
      goto 111
#endif

#ifdef XSMM
      ! Uses LIBXSMM: a library for dense and sparse matrix
   operations
      ...
      goto 111
#endif

#ifdef BLAS_MXM
      ! Uses the BLAS library bundled with Nek5000 / provided at
   compile-time
      call dgemm('N','N',n1,n3,n2,1.0,a,n1,b,n2,0.0,c,n1)
      goto 111
#endif

      ! Uses the loop-unrolled matrix multiplication subroutines
   in mxm_std.f
  101 call mxmf2(a,n1,b,n2,c,n3)

  111 continue
      ...
      return
      end
```

Listing 4: Various matrix multiplication implementations. Code snippet from Nek5000/core/mxm_wrapper.f

```
      subroutine mxf4(a,n1,b,n2,c,n3)
c
      real a(n1,4),b(4,n3),c(n1,n3)
c
      do j=1,n3
         do i=1,n1
            c(i,j) = a(i,1)*b(1,j)
     $           + a(i,2)*b(2,j)
     $           + a(i,3)*b(3,j)
     $           + a(i,4)*b(4,j)
         enddo
      enddo
      return
      end
```

Listing 5: Matrix multiplication implementation when n2 = 4. Code snippet from Nek5000/core/mxm_std.f

```
call mxm(d  ,m1,u(0,0,0,e),m1,ur,m2)
...
```

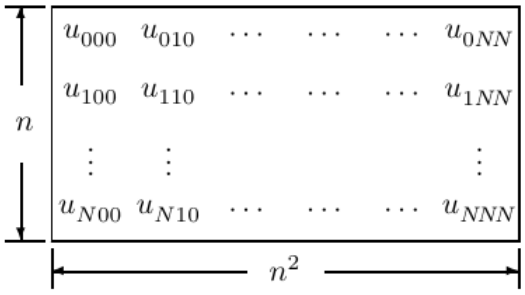


Figure 9: Interpretation of 3D data array as a $n \times n^2$ matrix. Source: Deville et al. [2004].

2nd `mxm` $u_s = I \otimes \hat{D} \otimes I = u_{(m,m)} \hat{D}^T_{(m,m)}$. See figure 10.

```
subroutine local_grad3(ur,us,ut,u,N,e,D,Dt)
Output: ur,us,ut          Input:u,N,e,D,Dt
...
m1 = n+1
m2 = m1*m1
...
do k=0,n
   call mxm(u(0,0,k,e),m1,dt,m1,us(0,0,k),m1)
enddo
...
```

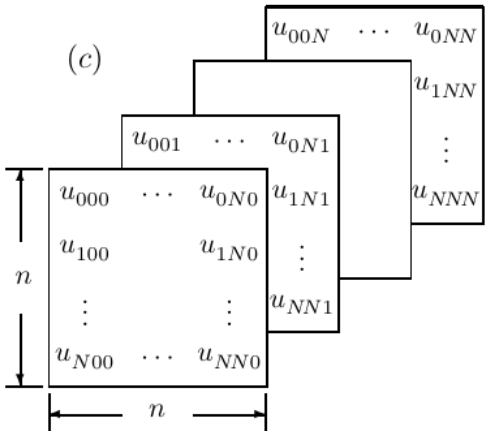


Figure 10: Interpretation of 3D data array as a sequence of n matrices of shape $n \times n$. Source: Deville et al. [2004].

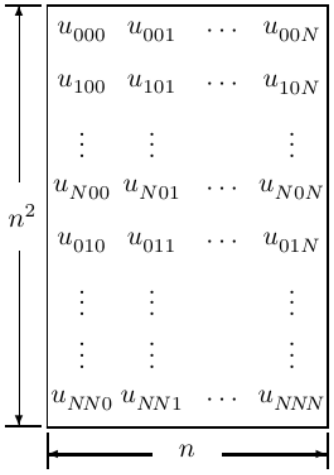


Figure 11: Interpretation of 3D data array as a $n^2 \times n$ matrix. Source: Deville et al. [2004].

3rd `mxm` $u_t = \hat{D} \otimes I \otimes I = u_{(m^2,m)} \hat{D}^T_{(m,m)}$. See figure 11.

```
subroutine local_grad3(ur,us,ut,u,N,e,D,Dt)
Output: ur,us,ut          Input:u,N,e,D,Dt
...
m1 = n+1
m2 = m1*m1
...
call mxm(u(0,0,0,e),m2,dt,m1,ut,m1)

return
end
```

Final step Once the derivatives in the transformed coordinates u_r, u_s and u_t are computed, these arrays are multiplied with metric terms and the Jacobian inside the subroutine `gradm1` (listing 1) to yield the components of the gradient in the original 3-D coordinate system.

References

Legendre wiki. https://en.wikipedia.org/wiki/Legendre_polynomials#Recurrence_relations. Accessed: 2021-05-20.

Milton Abramowitz. *Handbook of Mathematical Functions, With Formulas, Graphs, and Mathematical Tables,*. Dover Publications, Inc., USA, 1974. ISBN 0486612724.

M. Deville, Paul Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge University Press, 2004. ISBN 0-511-03760-0.

George Karniadakis. *Spectral/hp element methods for computational fluid dy-*

namics. Numerical mathematics and scientific computation. Oxford University Press, Oxford, 2nd ed.. edition, 2005. ISBN 1-4294-2205-X.

$\mathbb{P}_n - \mathbb{P}_{n-2}$ Methods for Incompressible Flows (I): Fractional-step versus LU-decomposition

Saumitra Vinay Joshi* Hamidreza Abedi†

June 7, 2021

1 Introduction

The non-dimensionalized unsteady incompressible Navier-Stokes equations are

$$\boldsymbol{u}_t + (\boldsymbol{u} \cdot \boldsymbol{\nabla})\, \boldsymbol{u} = -\boldsymbol{\nabla} p + \frac{1}{\mathrm{Re}} \boldsymbol{\nabla}^2 \boldsymbol{u}, \tag{1a}$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{u} = 0. \tag{1b}$$

These can be discretized using a number of spatial and temporal schemes. In this report, I summarize studies by J. Blair Perot [4] comparing two very similar approaches of discretizing and solving them. The first is the fractional step method, and the second is the block LU-decomposition method. The studies reveal that the block LU-decomposition method is a more appropriate choice because:

- It does not require boundary-conditions for the intermediate velocity variable, and
- It has no restriction of the order of accuracy of the splitting to first-order and can be increased to arbitrarily high orders.

2 Fractional-step Method

Equations (1a) and (1b) are discretized in time, leaving the space-discretization for later. Using the first-order backward Euler method for the diffusive terms and first-order forward Euler method for the convective terms results in

$$\frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^n}{\Delta t} + (\boldsymbol{u}^n \cdot \boldsymbol{\nabla})\, \boldsymbol{u}^n = -\boldsymbol{\nabla} p^{n+1} + \frac{1}{\mathrm{Re}} \boldsymbol{\nabla}^2 \boldsymbol{u}^{n+1}, \tag{2a}$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{u}^{n+1} = 0. \tag{2b}$$

*ETSIAE, School of Aeronautics, Universidad Politecnica de Madrid, Plaza Cardenal Cisneros 3, Madrid E-28040 Spain, saumitravinay.joshi@alumnos.upm.es

†Chalmers University of Technology, Sven Hiltins Plats 1, Göteborg 41258, hamidreza.abedi@chalmers.se

The fractional-step method approximates Equation (2a) by calculating an intermediate velocity $\boldsymbol{u}^*$ by omitting the pressure. It then uses the pressure to project $\boldsymbol{u}^*$ into the space of "discretely incompressible" functions, giving the final velocity. Mathematically, this looks like a time-splitting of (2a) as

$$\frac{\boldsymbol{u}^* - \boldsymbol{u}^n}{\Delta t} + (\boldsymbol{u}^n \cdot \boldsymbol{\nabla})\,\boldsymbol{u}^n = \frac{1}{\mathrm{Re}}\boldsymbol{\nabla}^2\boldsymbol{u}^*, \tag{3a}$$

$$\frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^*}{\Delta t} = -\boldsymbol{\nabla} p^{n+1}. \tag{3b}$$

To determine the pressure in Equation (3b), take the divergence of Equation (3b) and invoke the incompressibility condition of Equation (2b), giving the pressure Poisson equation,

$$(\boldsymbol{\nabla} \cdot \boldsymbol{\nabla})\, p^{n+1} = \frac{1}{\Delta t}\boldsymbol{\nabla} \cdot \boldsymbol{u}^*. \tag{4}$$

It can be solved as a repetition of three steps for every timestep, as

$$\frac{1}{\Delta t}\left[I - \frac{\Delta t}{\mathrm{Re}}\boldsymbol{\nabla}^2\right]\boldsymbol{u}^* = \frac{1}{\Delta t}\boldsymbol{\nabla}^2\boldsymbol{u}^n - (\boldsymbol{u}^n \cdot \boldsymbol{\nabla})\,\boldsymbol{u}^n, \tag{5a}$$

$$\Delta t\,(\boldsymbol{\nabla} \cdot \boldsymbol{\nabla})\, p^{n+1} = \boldsymbol{\nabla} \cdot \boldsymbol{u}^*, \tag{5b}$$

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^* - \Delta t \boldsymbol{\nabla} p^{n+1}. \tag{5c}$$

The spatial discretization in the fractional-step method is only applied at this stage on equations (5a) to (5c). While this makes the method independent of any particular discretization scheme, it leads to the following two issues:

1. *Need for additional boundary-conditions*: As each of the three equations (5a) to (5c) are discretized separately, they all require boundary-conditions. This creates the need for explicitly providing boundary-conditions for $\boldsymbol{u}^*$ and p. Boundary conditions for $\boldsymbol{u}^*$ lead to several uncomfortable questions, such as which field out of $\boldsymbol{u}^*$ and $\boldsymbol{u}$ is more physically correct [2]. There is considerable debate on the appropriate boundary-conditions for discretized p [5, 1, 6].

2. *Difficulty in reducing splitting error*: By adding equations (3b) and (3a) and comparing with Equation (2a), the error term is $\frac{\Delta t}{\mathrm{Re}}\boldsymbol{\nabla}^2\boldsymbol{\nabla} p^{n+1}$. Thus the fractional-step scheme is first-order accurate in time, and increasing its temporal accuracy is known to be very difficult [3].

3 Generalized Block LU-decomposition Method

Consider discretizing equations (1a) and (1b) using any discretization scheme in space and using first-order forward Euler method for the diffusive terms and

first-order backward Euler method for the convective terms in time. Their discrete form is written as

$$B\frac{\boldsymbol{v}^{n+1}-\boldsymbol{v}^{n}}{\Delta t}-C\left(\boldsymbol{v}^{n}\right)=D^{T}\boldsymbol{p}^{n+1}+\frac{1}{\mathrm{Re}}A\boldsymbol{v}^{n+1}, \tag{6a}$$

$$-D\boldsymbol{v}^{n+1}=0, \tag{6b}$$

where $C\left(\cdot\right)$ is some conservative discrete convective operator, D^T the discrete gradient operator, A the discrete laplacian operator and D the discrete divergence operator. Note that the boundary-values are included as constants in appropriate locations in the solution-vectors.

Let us rearrange Equation (6a) as

$$\left(\frac{1}{\Delta t}B+\frac{1}{\mathrm{Re}}A\right)\boldsymbol{v}^{n+1}-D^{T}\boldsymbol{p}^{n+1}=\boldsymbol{h}, \tag{7}$$

where $\boldsymbol{h}=C\left(\boldsymbol{v}^{n}\right)+\frac{1}{\Delta t}B\boldsymbol{v}^{n}$. Let $H=\left(\frac{1}{\Delta t}B+\frac{1}{\mathrm{Re}}A\right)$. Add $D^T\boldsymbol{p}^n$ to both sides of Equation (7) to reformulate the momentum equation in terms of the update in pressure between consecutive timesteps (as is done in NEK5000) as

$$H\boldsymbol{v}^{n+1}-D^{T}\left(\boldsymbol{p}^{n+1}-\boldsymbol{p}^{n}\right)=\boldsymbol{h}+D^{T}\boldsymbol{p}^{n}, \tag{8}$$

With an as-of-yet-unspecified operator Q, add to both sides of Equation (8) the term $-HQD^{T}\left(\boldsymbol{p}^{n+1}-\boldsymbol{p}^{n}\right)$. Rearranging, we get

$$\begin{aligned}H\boldsymbol{v}^{n+1}-HQD^{T}\left(\boldsymbol{p}^{n+1}-\boldsymbol{p}^{n}\right)&=\boldsymbol{h}+D^{T}\boldsymbol{p}^{n}\\&\quad-\left(HQ-I\right)D^{T}\left(\boldsymbol{p}^{n+1}-\boldsymbol{p}^{n}\right),\end{aligned} \tag{9}$$

where the term on the second line is the error. Denoting $\boldsymbol{p}^{n+1}-\boldsymbol{p}^{n}$ by $\boldsymbol{\phi}^{n+1}$ and $-\left(HQ-I\right)D^{T}\left(\boldsymbol{p}^{n+1}-\boldsymbol{p}^{n}\right)$ by $\boldsymbol{e}^{n+1}$, we get the following system:

$$\begin{pmatrix}H & -HQD^{T}\\ -D & 0\end{pmatrix}\begin{pmatrix}\boldsymbol{v}^{n+1}\\ \boldsymbol{\phi}^{n+1}\end{pmatrix}=\begin{pmatrix}\boldsymbol{h}+D^{T}\boldsymbol{p}^{n}\\ 0\end{pmatrix}+\begin{pmatrix}\boldsymbol{e}^{n+1}\\ 0\end{pmatrix}, \tag{10}$$

The LHS-matrix of Equation (10) can be block LU-decomposed as

$$\begin{pmatrix}H & -HQD^{T}\\ -D & 0\end{pmatrix}=\begin{pmatrix}H & 0\\ -D & -DQD^{T}\end{pmatrix}\begin{pmatrix}I & -QD^{T}\\ 0 & I\end{pmatrix} \tag{11}$$

such that the system is decomposed into two sub-systems as

$$\begin{pmatrix}H & 0\\ -D & -DQD^{T}\end{pmatrix}\begin{pmatrix}\boldsymbol{v}^{*}\\ \boldsymbol{\phi}^{n+1}\end{pmatrix}=\begin{pmatrix}\boldsymbol{h}+D^{T}\boldsymbol{p}^{n}\\ 0\end{pmatrix}+\begin{pmatrix}\boldsymbol{e}^{n+1}\\ 0\end{pmatrix} \tag{12}$$

and

$$\begin{pmatrix}I & -QD^{T}\\ 0 & I\end{pmatrix}\begin{pmatrix}\boldsymbol{v}^{n+1}\\ \boldsymbol{\phi}^{n+1}\end{pmatrix}=\begin{pmatrix}\boldsymbol{v}^{*}\\ \boldsymbol{\phi}^{n+1}\end{pmatrix}. \tag{13}$$

These correspond to the following three steps per timestep:

$$H\boldsymbol{v}^{*}=\boldsymbol{h}+D^{T}\boldsymbol{p}^{n}, \tag{14a}$$

$$-DQD^{T}\boldsymbol{\phi}^{n+1}=D\boldsymbol{v}^{*}, \tag{14b}$$

$$\boldsymbol{v}^{n+1}=\boldsymbol{v}^{*}+QD^{T}\boldsymbol{\phi}^{n+1} \tag{14c}$$

with the splitting error $\boldsymbol{e}^{n+1}$. Some points to note at this stage:

1. As all modifications to the fully discrete equations (6a) and (6b) are algebraic, there is no requirement for additional boundary-conditions for $\boldsymbol{v}^*$.

2. While papers claim that there is no boundary treatment on the pressure [4], it was shown in [2] that the LU-decomposition method (referred to therein as "Inexact Factorization") weakly imposes a zero-normal-gradient condition on $\boldsymbol{p}$ at the boundaries.

3. There is no error or approximation introduced in the discrete continuity equation. As a result, mass conservation is guaranteed.

3.1 Choice of Q

The choice of Q controls the splitting error and the simulation cost. Consider again the splitting error

$$\boldsymbol{e}^{n+1} = \left[\underbrace{\left(\frac{1}{\Delta t} B + \frac{1}{\mathrm{Re}} A \right)}_{H} Q - I \right] D^T \left(\boldsymbol{p}^{n+1} - \boldsymbol{p}^n \right). \tag{15}$$

Choosing $Q = H^{-1}$ eliminates the splitting error. This is known as the Uzawa method [7]. However, this introduces a nested approximate matrix inversion for *every* iteration of Equation (14b), which can be expensive.

For high-Re flows, the term $\frac{1}{\mathrm{Re}} A$ is small, and we can choose $Q = \Delta t B^{-1}$. For the spectral element method, the mass matrix B is easily invertible and hence the nested iterations of the Uzawa method are eliminated. In this case, the splitting error is first order

$$\boldsymbol{e}^{n+1} = \frac{\Delta t}{\mathrm{Re}} A B^{-1} D^T \left(\boldsymbol{p}^{n+1} - \boldsymbol{p}^n \right). \tag{16}$$

Other choices of Q can increase the order of accuracy of the splitting [4].

4 Implementation on NEK5000

The following steps show the implementation of the fractional-step method in NEK5000. For the fluid problems, `"subroutine fluid (igeom)"` existing in `"drive2.f"` is called in `"drive1.f"`. The subroutine `"fluid (igeom)"` is a driver for solving the incompressible Navier-Stokes equations.

In `"drive1.f"`:

- ...
  ...
  ```
  if (ifflow) call fluid (igeom)
  !(Solve for fluid, velocity and pressure.)
  ```
  ...
  ...

Subroutine "fluid (igeom)" is located at "drive2.f".

In "drive2.f":

- ...
  ...
  subroutine fluid (igeom)
  !(Driver for solving the incompressible Navier-Stokes equations.)
  ...
  ...
  call plan3 (igeom)
  !(iftran=1, ifrich=0)
  !(Solve time-dependent (transient) equations, velocity and pressure.)
  ...
  ...
  return
  end

Subroutine "plan3 (igeom)"is placed in "planx.f". It computes pressure and velocity coupling using fractional step method. In "planx.f", "subroutine incomprn (ux,uy,uz,up)" located at induct.f is called.

In "planx.f":

- SUBROUTINE PLAN3 (IGEOM)
  ...
  ...
  IF (IGEOM.EQ.1) THEN

  ! Old geometry

  CALL MAKEF
  !(Evaluate RHS (term h) in Eq. 14a.)
  !(Extrapolate pressure, solve for intermadiate velocity.)

  ELSE
  !(IGEOM.EQ.2)

  ! New geometry, new b.c.

  intype = -1
  !(Implicit)

  call sethlm (h1,h2,intype)
  !(Set the variable property arrays h_1 (diffusion) and h_2 (helmholtz operator) in the Helmholtz equation.)

  call cresvif (resv1,resv2,resv3,h1,h2)
  !(Compute start-residual/RHS in the velocity solver, second term in RHS of Eq. 14a.)

```
call ophinv (dv1,dv2,dv3,resv1,resv2,resv3,h1,h2,tolhv,nmxv)
!(Invert H in Eq. 14a as "out = (h1*A + h2*B)^-1 * inp".)

call opadd2 (vx,vy,vz,dv1,dv2,dv3)
!(Add the old velocities plus the differences.)

The intermediate velocities (v*) in Eq. 14a are made.

call incomprn(vx,vy,vz,pr)
!(Update the intermediate velocity (v*) by computing Eq. 14b and
Eq. 14c.)
!(Located at "induct.f".)

ENDIF
...
...
RETURN
END
```

In "induct.f":

- ```
  subroutine incomprn (ux,uy,uz,up)
  ...
  ...
  call opdiv (dp,ux,uy,uz)
  !(Calculate RHS of Eq. 14b, Dv*.)
  ...
  ...
  call esolver (dp,h1,h2,h2inv,intype)
  !(Solving φ^{n+1} in Eq. 14b, called dp here.)
  ...
  ...
  call add2(up,dp,ntot2)
  !(Add dp to the pressure, up.)

  call opgradt (w1 ,w2 ,w3 ,dp)
  !(Calculate the gradient of the solution, D^T φ^{n+1}.)

  call opbinv (dv1,dv2,dv3,w1 ,w2 ,w3 ,h2inv)
  !(Inverted mass matrix.)
  ...
  call opadd2cm (ux ,uy ,uz ,dv1,dv2,dv3, dtb )
  !(Make the second term (QD^T φ^{n+1}) in Eq. 14c to update v*.)
  ...
  ...
  RETURN
  END
  ```

Subroutine "ESOLVER (RES,H1,H2,H2INV,INTYPE)" is located at navier0.f.

In "navier0.f":
  ```

- SUBROUTINE ESOLVER (RES,H1,H2,H2INV,INTYPE)

```
...
...

if (.not.  ifsplit) then
if (param(42).eq.1) then
!(P042:  linear solver for the pressure equation (0:  GMRES, 1:
CG))
!(param(42) is hard-coded and set to be zero)
CALL UZAWA (RES,H1,H2,H2INV,INTYPE,ICG)
else
call uzawa_gmres(res,h1,h2,h2inv,intype,icg)
!("subroutine uzawa_gmres(res,h1,h2,h2inv,intype,iter)" is located
at "gmres.f")
!(Linear solver for the pressure equation (GMRES))
endif
else
...
...
RETURN
END
```

References

[1] Philip M. Gresho and Robert L. Sani. On pressure boundary conditions for the incompressible navier-stokes equations. *International Journal for Numerical Methods in Fluids*, 7(10):1111–1145, October 1987.

[2] J.L. Guermond, P. Minev, and Jie Shen. An overview of projection methods for incompressible flows. *Computer Methods in Applied Mechanics and Engineering*, 195(44):6011–6045, 2006.

[3] Yvon Maday, Anthony T Patera, and Einar M Rønquist. An operator-integration-factor splitting method for time-dependent problems: application to incompressible fluid flow. *Journal of Scientific Computing*, 5(4):263–292, 1990.

[4] J.Blair Perot. An analysis of the fractional step method. *Journal of Computational Physics*, 108(1):51–58, 1993.

[5] Dietmar Rempfer. On Boundary Conditions for Incompressible Navier-Stokes Problems. *Applied Mechanics Reviews*, 59(3):107–125, 05 2006.

[6] R. L. Sani, J. Shen, O. Pironneau, and P. M. Gresho. Pressure boundary condition for the time-dependent incompressible navier–stokes equations. *International Journal for Numerical Methods in Fluids*, 50(6):673–682, February 2006.

[7] Roger Temam. Navier-stokes equations: Theory and numerical analysis. 2:500 pp, 01 2001.

Part 2: Solution-algorithms for the generalized block-LU decomposition using SEM

Yashas Bharadhwaj*

June 5, 2021

*Dept. of Mechanics and Maritime sciences, Chalmers University of Technology, Göteborg, Sweden

1 Solution to the Stokes Problem

The stokes problem in matrix form is given by:

$$\begin{bmatrix} H & & -D_1^{\mathrm{T}} \\ & H & -D_2^{\mathrm{T}} \\ -D_1 & -D_2 & 0 \end{bmatrix} \begin{pmatrix} \underline{u}_1 \\ \underline{u}_2 \\ \underline{p} \end{pmatrix} = \begin{pmatrix} \underline{f}_1 \\ \underline{f}_2 \\ f_p \end{pmatrix} \tag{1}$$

with $H = \left(\frac{\beta_0}{\Delta t} B + \frac{1}{Re} A\right)$ is the *Helmholtz operator*[1].

The solution to the Stokes problem demands the use of iterative solvers. The system presented in equation(1) can be solved with two approaches; A preconditioned conjugate gradient method with the choice of Legendre-based quadrature (i.e. Chebyshev) or to decouple pressure and velocity by formally carrying out Block LU Decomposition (*Uzawa Method*) on the system.

Procedure undertaken by Nek5000 to solve for the N-S equation is as described below:

Step 1. Compute intermediate velocity components

$$\Delta u^* = H^{-1} \left(h^n + D^T p^{n-1} - H u^{n-1}\right) \tag{2}$$

Step 2. Add contribution to velocity from previous time step

$$u^* = u^{n-1} + \Delta u^* \tag{3}$$

Step 3. Solve the Poisson problem

$$E \Delta p^n = D u^* \tag{4}$$

Step 4. Add pressure contribution from previous time-step

$$p^n = p^{n-1} + \Delta p^n \tag{5}$$

Step 5. Obtain updated velocity

$$u^n = u^* + \Delta u^n \tag{6}$$

1.1 Solution scheme for updated velocity field

In $P_N - P_{N-2}$ method, solution to the updated velocity field is obtained by contribution from two consecutive time steps, i.e.

$$H u^n = h^{n-1} + D^T p^{n-1} - \frac{\Delta t}{\beta_0} H B^{-1} D^T \Delta p^n \tag{7}$$

$$H\left(u^{n-1}+\Delta u^{n}\right)=h^{n-1}+D^{T}p^{n-1}-\frac{\Delta t}{\beta_0}HB^{-1}D^{T}\Delta p^{n} \tag{8}$$

$$H\Delta u^{n}=h^{n-1}+D^{T}p^{n-1}-Hu^{n-1}-\frac{\Delta t}{\beta_0}HB^{-1}D^{T}\Delta p^{n} \tag{9}$$

Hence, upon solving for equations (2)-(6) along with Δu^n (9), the difference between two consecutive time steps is incorporated into the solver.

1.2 Preconditioning and pressure calculation

The ability of any given iterative method to arrive at the exact solution upon certain iterations is highly dependent on properties of the matrix. This serves as a justification for preconditioners.

For larger values of Δt the Helmholtz operator reduces to $H^{-1} \approx Re.A^{-1}$, which can very well serve as preconditioner (Stokes preconditioner). However, when the choice of Q such that $H^{-1} \approx (\Delta t/\beta_0)\, B^{-1} = Q$ is made, the previous approximation holds inconsistent. As a consequence, it leads to splitting error which then requires the use of Uzawa method to resolve this problem. Furthermore, Uzawa method with its nested iterations for the pressure problem in (5) is to be solved.

To solve for the pressure, its perturbation equivalent (equation (4)) is solved by invoking the `esolver` within `incomprp`. Furthermore, depending on the fla raised by `param(42)`, equation (5) is solved. If `param(42)` $= 0$, then GMRES method (`uzawa_gmres`) is implemented (see Algorithm 1), else CG solver (`uzawa`) is implemented if `param(42)` $= 1$ (see Algorithm 2)

Implementation of both the iterative methods consists of evaluation of an input vector by the `cdabdtp` routine (Application of E operator) (see Algorithm 3). Different choices for Q are implemented by supplying them through the coefficient arrays of the Helmholtz operator H1 and H2. Solution for pressure solver is implemented in the `incomprp` subroutine.

Algorithm 1 Uzawa Method: GMRES

1: **procedure** `uzawa|`(r_{cg})
2: **call** `CHKTCG2` → check that tolerances are not very small for the solver
3: **call** `CDABDTP` → Apply E operator
4: **if** conv. criterion is satisfied:
5: break the loop
6: **else:** perform preconditioning based on `param(43)`
7: **if** `param(43) == 1` → Schwarz preconditioning
8: **else:** HSMG preconditioning
9: **call** `CDABDTP` → Apply E operator
10: **call** `ortho` → orthogonalize w w.r.t null-space
11: **call** `ortho` → orthogonalize *res* w.r.t null-space
12: **return** *res*

Algorithm 2 Uzawa Method: preconditioned conjugate gradient (pcg)

procedure `uzawa|`(r_{cg}) input array $(r_{cg}) = D_i u_i^*$
call `CHKTCG2` → check that tolerances are not very small for the *pcg solver*
call `UZPREC` → precondition the input array
call `convprn` → check if convergence criterion satisfied
if conv. criterion is satisfied:
break the loop
else:
call `CDABDTP` → Apply E operator
call `ortho` → orthogonalize r_{cg} w.r.t null-space
call `UZPREC` → precondition the input array
call `ortho` → orthogonalize r_{cg} w.r.t null-space
return r_{cg}

Algorithm 3 Application of E operator

procedure `CDABDTP|`(ap, wp, h1, h2, h2inv, intype)
call `OPGRADT`(TA1,TA2,TA3,WP) → ta_i
if *inptype* == 0 or *inptype* == 1:
call `OPHINV`(TB1,TB2,TB3,TA1,TA2,TA3,H1,H2,TOLHIN,NMXH) → tb_i
else:
call `OPBINV`(TB1,TB2,TB3,TA1,TA2,TA3,H1,H2,TOLHIN,NMXH) → tb_i
end if
call `OPDIV`(AP,TB1,TB2,TB3) → **return** ap
end procedure=0

2 References

References

[1] M. O. Deville, P. F. Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge, 2002.

Pn-Pn Formulation and its Implementation in Nek5000

David Krantz*, Masumeh Gholamisheeri† and Martin Karp‡

October 6, 2021

1 Introduction

In this report, the Pn-Pn formulation for incompressible flows is derived and its implementation in Nek5000 is introduced. Initially, the higher-order time integration of the Navier-Stokes (NS) equations is demonstrated, which is based on a semi-implicit explicit approach and thus a proper treatment of the pressure term and stability and accuracy of the mixed scheme should be investigated. In this report, the *Karniadakis* scheme [2] is used for the splitting of the integrated NS equations. By considering some assumption, a Poisson equation for the pressure is derived and its corresponding pressure boundary condition is introduced. However, since the pressure boundary condition contains terms that should be evaluated at the new time step (i.e. $n+1$) and previous time steps (i.e. n), solving for the exact pressure boundary condition is difficult and computationally expensive. In section 4, possible solutions to overcome this difficulty are mentioned and the most stable scheme is used. However, using this scheme causes mass conservation errors known as splitting errors. In section 5 the order of these errors is demonstrated and it is expressed that the solution lies in the control of boundary values (i.e. imposing suitable pressure boundary conditions). The last part of this report deals with the low-Mach number formulation of the compressible NS equations and a brief explanation of the numerical approach for solving these equations. In the end, the implementation of the Pn-Pn approach in Nek5000 is provided in section 7.

2 Navier-Stokes Equations

Navier-Stokes Formulation: The governing equations for Newtonian incompressible flows with constant properties reads as

$$\frac{\partial \mathbf{v}}{\partial t} = -\nabla p + \nu \mathbf{L}(\mathbf{v}) + \mathbf{N}(\mathbf{v}) \quad \text{in } \Omega, \tag{1}$$

and the incompressibility constraint is defined as

$$Q = \nabla \cdot \mathbf{v} = 0 \quad \text{in } \Omega. \tag{2}$$

*KTH Numerical Analysis, davkra@kth.se
†KTH Engineering Mechanics, masumeh@kth.se
‡KTH Computational Science and Engineering, makarp@kth.se

Here, $\mathbf{v}$ is the velocity vector, $(= u\hat{x}+v\hat{y}+w\hat{z})$, p is the pressure, and kinematic viscosity is shown by ν. The linear and nonlinear operators in NS equations are represented by $\mathbf{L}$ and $\mathbf{N}$, and are defined as

$$\mathbf{L}(\mathbf{v}) = \nabla^2\mathbf{v} = \nabla(\nabla\cdot\mathbf{v}) - \nabla\times(\nabla\times\mathbf{v}) \tag{3}$$

$$\mathbf{N}(\mathbf{v}) = -\frac{1}{2}[\mathbf{v}\cdot\nabla\mathbf{v} + \nabla(\mathbf{v}\cdot\mathbf{v})]. \tag{4}$$

The nonlinear operator is written in the skew-symmetric form, which is a linear combination of convective and conservative forms [4]. Note that the skew-symmetric form is not used in Nek5000, however, it was convenient to have it in the convective formulation. The above-described NS equations require discretization in both time and space. Here, we integrate the equation (1) with the use of high-order time-stepping schemes. This type of schemes give rise to mixed explicit-implicit schemes. As a result, proper treatment of the pressure term and the stability and accuracy of such mixed schemes should be investigated [2].

Integrating equation (1) over one time step, Δt, results in

$$\mathbf{v}^{n+1} - \mathbf{v}^n = -\int_{t_n}^{t_{n+1}} \nabla p dt + \nu\int_{t_n}^{t_{n+1}} \mathbf{L}(\mathbf{v})dt + \int_{t_n}^{t_{n+1}} \mathbf{N}(\mathbf{v})dt, \tag{5}$$

where the superscript index n refers to time level $t_n = n\Delta t$. The pressure term is written as

$$\int_{t_n}^{t_{n+1}} \nabla p dt = \Delta t \nabla \overline{p}^{n+1} \tag{6}$$

where $\overline{p}^{n+1}$ is the scalar field that ensures that the final velocity field is incompressible at the end of time level $(n+1)$ [2], meaning that the velocity field is divergence free. The nonlinear and linear terms are approximated using explicit and implicit schemes, respectively. The former is approximated using the J_e order scheme from the Adams-Bashforth family, while the latter is approximated by a scheme of order J_i from the Adams-Moulton family. Hence,

$$\int_{t_n}^{t_{n+1}} \mathbf{N}(\mathbf{v})dt = \Delta t \sum_{q=0}^{J_e-1} \beta_q \mathbf{N}(\mathbf{v}^{n-q}) \tag{7}$$

and

$$\int_{t_n}^{t_{n+1}} \mathbf{L}(\mathbf{v})dt = \Delta t \sum_{q=0}^{J_i-1} \gamma_q \mathbf{L}(\mathbf{v}^{n+1-q}), \tag{8}$$

where β_q and γ_q are the appropriately chosen weights for explicit and implicit schemes, respectively [1]. It should be noted that the reason for the choice of implicit scheme for the linear term is the stability of the scheme. Also, it is noteworthy that the left hand side of equation (5) is an illustration of a first order method, and to obtain a so-called stiffly stable scheme we use a higher-order method to discretize $\partial\mathbf{v}/\partial t$, we can generalize this for different orders as

$$\frac{\partial\mathbf{v}}{\partial t} = \frac{\gamma_0\mathbf{v}^{n+1} - \sum_{q=0}^{J_i-1}\alpha_q\mathbf{v}^{n-q}}{\Delta t}, \tag{9}$$

where α_q are the coefficients of the stiffly stable scheme of order J_i. As for γ_0 it holds that $\gamma_0 = \sum_{q=0}^{J_i-1}\alpha_q$. The solution to the above semi-discrete system, (5), can beobtained by using the discretiziations shown in ,(6)-(9), and splitting the computation into three substeps as is provided in section 3.

3 Karniadakis Scheme

Commonly known as the Pn-Pn formulation, the Karniadakis scheme is not necessarily constrained to only Spectral Element Methods, even if it was its original focus and usage today. At its core, it simply decouples the pressure and velocity from each other and can be applied to other methods as well. In this section, we will introduce the splitting and provide an overview of how the scheme is used to decouple the pressure and velocity. By inserting (9) and discretizations of the linear and non-linear term similar to those in (7) and (8) into (1), we arrive at the following semi-discrete system

$$\frac{\gamma_0 \mathbf{v}^{n+1} - \sum_{q=0}^{J_i-1} \alpha_q \mathbf{v}^{n-q}}{\Delta t} = -\nabla \bar{p}^{n+1} + \nu \sum_{q=0}^{J_i-1} \gamma_q \mathbf{L}(\mathbf{v}^{n+1-q}) + \sum_{q=0}^{J_e-1} \beta_q \mathbf{N}(\mathbf{v}^{n-q}). \tag{10}$$

In order to solve this system, the Karniadakis scheme splits the system into three substeps. The idea being that we can use equation (10) to solve the following split system

$$\frac{\hat{\mathbf{v}} - \sum_{q=0}^{J_i-1} \alpha_q \mathbf{v}^{n-q}}{\Delta t} = \sum_{q=0}^{J_e-1} \beta_q \mathbf{N}(\mathbf{v}^{n-q}) \qquad \text{in} \quad \Omega \tag{11}$$

$$\frac{\hat{\hat{\mathbf{v}}} - \hat{\mathbf{v}}}{\Delta t} = -\nabla \bar{p}^{n+1} \qquad \text{in} \quad \Omega \tag{12}$$

$$\frac{\gamma_0 \mathbf{v}^{n+1} - \hat{\hat{\mathbf{v}}}}{\Delta t} = \nu \nabla^2 \mathbf{v}^{n+1} \qquad \text{in} \quad \Omega \tag{13}$$

with Dirichlet boundary conditions

$$\mathbf{v}^{n+1} = \vec{v}_0 \qquad \text{on} \quad \partial\Omega. \tag{14}$$

However, in this system the velocity and pressure at time step $n+1$ are still coupled and the velocity is not discretized. We need to address this and we do so by first making the observation that the incompressibility condition must be satisfied for $\mathbf{v}$,

$$\nabla \cdot \mathbf{v} = 0 \qquad \text{in} \quad \Omega. \tag{15}$$

By taking the divergence of (10) we arrive at the following Poisson equation for the pressure

$$\nabla^2 \bar{p}^{n+1} = \nabla \cdot \left(\frac{\hat{\mathbf{v}}}{\Delta t} + \nu \sum_{q=0}^{J_i-1} \gamma_q \mathbf{L}(\mathbf{v}^{n+1-q}) \right) \qquad \text{in} \quad \Omega \tag{16}$$

However, this system is still coupled since we need to evaluate $\nabla^2 \mathbf{v}^{n+1}$. Following the derivation of [3], we consider the decomposition of the velocity field $\mathbf{v}$ into its irrotational and solenoidal components, $\mathbf{v}_I$ and $\mathbf{v}_S$ respectively, namely

$$\mathbf{v} = \mathbf{v}_I + \mathbf{v}_S, \tag{17}$$

where $\nabla \times \mathbf{v}_I = 0$, and $\nabla \cdot \mathbf{v}_S = 0$. For incompressible flows the irrotational part $\mathbf{v}_I$ conveniently vanishes. By utilizing this decomposition and the identity

$\nabla^2\mathbf{v}^{n+1} = \nabla(\nabla\cdot\mathbf{v}^{n+1}) - \nabla\times(\nabla\times\mathbf{v}^{n+1})$ shown in (3) we can rewrite $\nabla^2\mathbf{v}^{n+1}$ as

$$\nabla^2\mathbf{v}^{n+1} = -\nabla\times(\nabla\times\mathbf{v}^{n+1}), \tag{18}$$

since $\mathbf{v}_I$ is 0 for incompressible flows. What we now may do is approximate this equation with an explicit scheme leading us to our final Poisson equation for the pressure

$$\nabla^2\bar{p}^{n+1} = \nabla\cdot\left(\frac{\hat{\mathbf{v}}}{\Delta t} + \sum_{q=0}^{J_e-1}\beta_q\left(-\nabla\times(\nabla\times\mathbf{v}^{n-q})\right)\right), \quad \text{in } \Omega. \tag{19}$$

We now have almost all the tools needed to solve the system. First we use (11) to obtain the nonlinear terms, then we use this to calculate the pressure at the next time step according to (19) and lastly we use the computed pressure field to solve the Helmholtz equation for the velocity in (13). This splitting of the pressure equation and velocity is at the very core of the Karniadakis scheme used in Nek5000. The small changes necessary for the Low Mach formulation is described in [5].

However, the steps described here are not sufficient in order to obtain a solution. While we have prescribed boundary conditions for the volcity, the boundary conditions for the pressure is an open question. Imposing proper boundary conditions for the pressure Poisson equation (19) is incredibly important for the accuracy of this method and we will address this in the next section.

4 Pressure Boundary Condition

Imposing a suitable boundary condition to the Poisson equation for the pressure in (19) is not trivial. We approach this problem by first deriving the pressure boundary condition from the semi-discrete formulation and then approximating the unknown implicit terms using a high-order extrapolation scheme. Solving the semi-discrete form in (10) for $\nabla\bar{p}^{n+1}$ gives

$$\begin{aligned}\nabla\bar{p}^{n+1} = &-\frac{\gamma_0\mathbf{v}^{n+1} - \sum_{q=0}^{J_i-1}\alpha_q\mathbf{v}^{n-q}}{\Delta t} + \nu\gamma_0\mathbf{L}(\mathbf{v}^{n+1}) \\ &+ \nu\sum_{q=1}^{J_i-1}\gamma_q\mathbf{L}(\mathbf{v}^{n+1-q}) + \sum_{q=0}^{J_e-1}\beta_q\mathbf{N}(\mathbf{v}^{n-q}),\end{aligned} \tag{20}$$

where we have taken the first term of the linear summation term outside of its summation. We note that by doing this both summations are explicitly known since they only depend on velocities up to time step n. Then, by taking the partial derivative of (20) with respect to the surface normal at the boundary gives

$$\begin{aligned}\frac{\partial\bar{p}^{n+1}}{\partial n} = &-\mathbf{n}\cdot\left(\frac{\gamma_0\mathbf{v}^{n+1} - \sum_{q=0}^{J_i-1}\alpha_q\mathbf{v}^{n-q}}{\Delta t}\right) + \nu\gamma_0\left(\mathbf{n}\cdot\mathbf{L}(\mathbf{v}^{n+1})\right) \\ &+ \mathbf{n}\cdot\left(\nu\sum_{q=1}^{J_i-1}\gamma_q\mathbf{L}(\mathbf{v}^{n+1-q}) + \sum_{q=0}^{J_e-1}\beta_q\mathbf{N}(\mathbf{v}^{n-q})\right), \quad \text{on } \partial\Omega.\end{aligned} \tag{21}$$

Since we assume that we have Dirichlet boundary conditions on the velocity we treat both the first and last term of (20) as known. The difficulty now lies in computing $\mathbf{L}(\mathbf{v}^{n+1}) = \nabla(\nabla\cdot\mathbf{v}^{n+1}) - \nabla\times(\nabla\times\mathbf{v}^{n+1}) = \nabla Q^{n+1} - \nabla\times(\nabla\times\mathbf{v}^{n+1})$ since $\mathbf{v}^{n+1}$ is unknown.

There are different ways of handling this problematic term and we will briefly mention four approaches, and then present the one that is used in Nek5000. The first approach is to treat the term as is, but the problem with this is that it results in a coupled system between the pressure and velocity at time step $n+1$, which is what we want to avoid by performing the aforementioned splitting. Next, due to the incompressibility constraint, we require $\nabla Q^{n+1} = 0$. Thus, it would not be unreasonable to set the problematic term to zero. However, this approach introduces large errors at boundaries. More specifically, this error, called the splitting error, becomes the dominating error factor, which in turn limits the resulting scheme to first order regardless of the order of the temporal scheme [3]. The third alternative is to approximate the velocity field using what we know, i.e. replace $\mathbf{L}(\mathbf{v}^{n+1})$ with $\mathbf{L}(\mathbf{v}^{n})$, or approximating $\mathbf{v}^{n+1}$ using a high-order explicit extrapolation scheme. This approach results in a high-order scheme. However, it may experience weak instability [2]. The last approach, which is the one used in Nek5000, is to replace $\mathbf{L}(\mathbf{v}^{n+1})$ by the rotational form of the Laplacian $\omega^{n+1} = \nabla\times(\nabla\times\mathbf{v}^{n+1})$ as in (17)-(18), which also yields a high-order scheme. The advantage of this scheme is that we gain a factor $\Delta t^{1/2}$ in accuracy compared to the previous one [3].

The latter approach was then extended in [2] by considering an approximation of the unknown ω^{n+1} using an explicit scheme

$$\tilde{\omega}^{n+1} = \sum_{q=0}^{J_e-1} \beta_q(\nabla\times(\nabla\times\mathbf{v}^{n-q})). \tag{22}$$

Inserting (18) and (22) into (21) gives the final expression of the pressure boundary condition

$$\begin{aligned}\frac{\partial\bar{p}^{n+1}}{\partial n} = -\mathbf{n}\cdot\left(\frac{\gamma_0\mathbf{v}^{n+1} - \sum_{q=0}^{J_i-1}\alpha_q\mathbf{v}^{n-q}}{\Delta t}\right) + \mathbf{n}\cdot\Bigg(\sum_{q=0}^{J_e-1}\beta_q\mathbf{N}(\mathbf{v}^{n-q}) \\ +\nu\sum_{q=0}^{J_i-1}\gamma_q\nabla Q^{n+1-q} + \nu\sum_{q=0}^{J_e-1}\beta_q(-\nabla\times(\nabla\times\mathbf{v}^{n-q}))\Bigg), \quad \text{on } \partial\Omega.\end{aligned} \tag{23}$$

Note that the first term of the third summation, $\gamma_0\nabla Q^{n+1}$, is set to zero since we require $Q^{n+1} = 0$ in order to fulfil the incompressibility constraint. Solving the Poisson equation for the pressure in (19) with the Neumann boundary conditions in (23) requires a compatibility condition. However, this condition is always satisfied if $\hat{\mathbf{v}}$ is defined as in (11), see [2] for more details.

5 Splitting Error

This section describes how the splitting error affects the pressure computations but also how it relates to the accuracy of the global solution.

Rewriting the pressure boundary condition in (21) using the rotational form of the Laplacian gives

$$\begin{aligned}\frac{\partial \bar{p}^{n+1}}{\partial n} = &-\mathbf{n}\cdot\left(\frac{\gamma_0\mathbf{v}^{n+1}-\sum_{q=0}^{J_i-1}\alpha_q\mathbf{v}^{n-q}}{\Delta t}\right)\\ &+\mathbf{n}\cdot\left(\sum_{q=0}^{J_e-1}\beta_q\mathbf{N}(\mathbf{v}^{n-q})+\nu\sum_{q=0}^{J_i-1}\left(\gamma_q\nabla Q^{n+1-q}+\omega^{n+1}\right)\right).\end{aligned} \tag{24}$$

By comparing (24) with (23) we see that an error is introduced due to the approximation $\tilde{\omega}^{n+1}$ of ω^{n+1}. This difference originates from the splitting errors. More specifically, the difference is

$$\|\omega^{n+1}-\tilde{\omega}^{n+1}\| = \mathcal{O}\left(\Delta t^{J_e}\right), \tag{25}$$

where J_e is the order of the explicit extrapolation scheme.

If we take the divergence of the semi-discrete formulation in (10) we have

$$\begin{aligned}&\frac{\gamma_0 Q^{n+1}-\sum_{q=0}^{J_i-1}\alpha_q Q^{n-q}}{\Delta t}-\gamma_0\nabla^2 Q^{n+1} = -\nabla^2\bar{p}^{n+1}\\ &+\nabla\cdot\left(\nu\sum_{q=1}^{J_i-1}\gamma_q\mathbf{L}(\mathbf{v}^{n+1-q})+\sum_{q=0}^{J_e-1}\beta_q\mathbf{N}(\mathbf{v}^{n-q})\right).\end{aligned} \tag{26}$$

Assuming that the divergence of the velocity field is zero for all time steps up to $n+1$, then (26) reads

$$\nabla^2\bar{p}^{n+1} = \nabla\cdot\left(\frac{\hat{\mathbf{v}}}{\Delta t}+\nu\sum_{q=0}^{J_i-1}\gamma_q\mathbf{L}(\mathbf{v}^{n+1-q})\right), \tag{27}$$

which is the Poisson equation for the pressure previously seen in (16). Here, $\hat{\mathbf{v}}$ is defined in (11). Suppose that $\nabla^2\bar{p}^{n+1}$ is computed using (27), then

$$\frac{\gamma_0 Q^{n+1}-\sum_{q=0}^{J_i-1}\alpha_q Q^{n-q}}{\Delta t}-\gamma_0\nabla^2 Q^{n+1} = 0 \tag{28}$$

should hold. Assume instead that the divergence of the velocity field is zero for all time steps prior to $n+1$, then Q^{n+1} is found as

$$Q^{n+1}-\nu\Delta t\nabla^2 Q^{n+1} = 0, \tag{29}$$

which is a Helmholtz equation for Q^{n+1}. Since (29) is elliptic we have a maximum value principle interpretation of it that says that Q^{n+1} takes its maximum value on $\partial\Omega$. From (29) we also see that if $Q^{n+1}\neq 0$ on $\partial\Omega$, then Q^{n+1} decays exponentially to zero with a rate of s/l, where s is a general coordinate normal to the boundary and $l=\sqrt{\nu\Delta t}$ is known as the (numerical) boundary layer thickness. We note that $l\to 0$ as $Re\to\infty$ since $\nu\propto Re^{-1}$, i.e. that the approximation becomes better for large Reynolds numbers.

The above shows that the divergence at step $n+1$ is dominated by its boundary values $\partial Q/\partial n$ or by the divergence in the previous step. The problem of controlling Q^{n+1} thus lies in the control of its boundary values, which in turn

corresponds to imposing suitable pressure boundary conditions [3]. Furthermore, it was shown in [2] that the time-differencing error of the velocity field is one order smaller in Δt than the corresponding error of boundary divergence $\partial Q/\partial n$. But, the accuracy of $\partial Q/\partial n$ depends on the treatment of the pressure boundary condition. We thus expect that approximating the pressure boundary condition with e.g. a first-order method would result in second-order results in the velocity field.

To conclude, the temporal accuracy of the global solution is directly influenced by the boundary values of the divergence, which in turn depends on the treatment of the pressure boundary condition [2].

6 Low-Mach Number Formulation

In the sections above, the Navier-Stokes equations for incompressible flows were approximated with an explicit-implicit schemes. It is interesting to look into the approximate equations for compressible flows in the absence of high frequency acoustic waves. The resulting equations when the Mach number approaches zero are as follow [5];

$$\rho c_p(\frac{\partial T}{\partial t} + \mathbf{v}\cdot\nabla T) = \nabla\cdot\lambda\nabla T + \sum_{i=1}^{N} h_i^\circ \dot{\Psi}_i - \nabla\cdot\rho\mathbf{T}\sum_{i=1}^{N} c_{p,i}Y_i\mathbf{V}_i + \frac{\partial P_0}{\partial t}\,, \quad (30)$$

$$\rho(\frac{\partial Y_i}{\partial t} + \mathbf{v}\cdot\nabla Y_i) = -\nabla\cdot\rho Y_i\mathbf{V}_i + \dot{\Psi}_i. \quad i = 1,\ldots,N\,, \quad (31)$$

$$P_0 = \rho RT\,, \quad (32)$$

$$\rho(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}) = -\nabla p + \nabla\cdot\mu\left(\nabla\mathbf{v} + (\nabla\mathbf{v})^T - \frac{2}{3}(\nabla\cdot\mathbf{v})\mathbf{I}\right)\,, \quad (33)$$

$$\frac{\partial\rho}{\partial t} + \mathbf{v}\cdot\nabla\rho = \rho\nabla\cdot\mathbf{v}\,. \quad (34)$$

In the above equations, temperature field is shown by T, ρ is the density, and Y_i and $\mathbf{V}_i$ are the mass fraction and diffusion velocity of the i-th species [6]. The heat of formation and rate of production of the i-th species are denoted by h_i° and $\dot{\Psi}_i$, respectively. The heat conductivity and dynamic viscosity are shown by λ and μ, and $c_{p,i}$ is the specific heat capacity of the i-th species. The equations (31)-(34) are derived from the balance of ϵ°, where $\epsilon = \gamma M^2$, where γ is the ratio of specific heats, and M is the Mach number. Here, p_0 is the thermodynamic pressure and p is the dynamic pressure, where p_0 can only be a function of time. In an open system where the pressure has to reach a constant value at infinity, (i.e. atmospheric pressure), the last term of equation (30) vanishes.

The phenomena that are involved in a system of chemically-reactive flow includes hydrodynamic, diffusion, transport and thermo-chemistry [5]. To present a numerical approach for the integration of the governing equations, several simplified assumptions should be made. 1) The transport processes are ignored, leading to a single-step reaction mechanism. This means that all species have the same molecular weight, specific heat capacity and binary diffusion coefficients, (i.e. D_{ij}, that appears in the definition of $\mathbf{V}_i$). 2) All dynamic transport

coefficients μ, λ, ρD, and specific heat capacity c_p are assumed to be independent of temperature. As a result, the kinematic transport coefficients, ν, α and D become directly proportional to temperature. 3) Thermodynamic pressure is assumed to be constant in both time and space (open system assumption). In addition, the reaction rate is written in the Arrhenius form. Arrhenius equation provides the dependence of a reaction rate constant of a chemical reaction to the absolute temperature.

$$k = Ae^{\frac{-E_a}{RT}} \tag{35}$$

where k is the rate constant (i.e. frequency of collisions resulting in a reaction), A is the pre-exponential factor that is a constant for each chemical reaction, E_a is the activation energy for the reaction and T is the absolute temperature (in Kelvin).

Once the equations (30)-(34) are nondimensionalized with appropriate quantities, and considering the above-mentioned simplifying assumptions, the system of equations reads as

$$\frac{\partial \tilde{T}}{\partial \tilde{t}} + \tilde{\mathbf{v}} \cdot \tilde{\nabla} \tilde{T} = \frac{\tilde{\alpha}}{RePr} \tilde{\nabla}^2 \tilde{T} + Da \sum_{i=1}^{N} \tilde{h_i^\circ}\ \dot{\tilde{\Psi}}_i', \tag{36}$$

$$\frac{\partial \tilde{Y_i}}{\partial \tilde{t}} + \tilde{\mathbf{v}} \cdot \tilde{\nabla} \tilde{Y_i} = \frac{\tilde{D}}{ReSc} \tilde{\nabla}^2 Y_i + Da\ \dot{\tilde{\Psi}}_i', \quad i = 1, \ldots, N\,, \tag{37}$$

$$1 = \tilde{\rho} \tilde{T}\,, \tag{38}$$

$$(\frac{\partial \tilde{\mathbf{v}}}{\partial \tilde{t}} + \tilde{\mathbf{v}} \cdot \tilde{\nabla} \tilde{\mathbf{v}}) = -\frac{1}{\tilde{\rho}} \tilde{\nabla} \tilde{p} + \frac{\tilde{\nu}}{Re} (\tilde{\nabla}^2 \tilde{\mathbf{v}} + \frac{1}{3} \tilde{\nabla} (\tilde{\nabla} \cdot \tilde{\tilde{v}}))\,, \tag{39}$$

$$\tilde{\nabla} \cdot \tilde{\mathbf{v}} = \frac{1}{RePr} \tilde{\nabla}^2 \tilde{T} + Da \sum_{i=1}^{N} \frac{\tilde{h_i^\circ}}{\tilde{T}}\ \dot{\tilde{\Psi}}_i', \tag{40}$$

In the above equations, all variables with tildes are nondimensional, and Re, Pr and Sc are Reynolds, Prandtl (ratio of momentum diffusivity to the thermal diffusivity) and Schmidt (ratio of viscous diffusion to mass diffusion) numbers respectively, Da is the pre-exponential Damköhler number (ratio of the flow time scale to the chemical time scale) for one step reaction. The continuity equation (34), is simplified using energy (30) and state (32) equations, and it shows that the only nonzero divergence of the velocity field is the heat release by chemical reactions and diffusive heat transfer.

6.1 Numerical approach for low-Mach number

The terms of the energy and species equations (36)-(37) are advanced in time implicitly, (due to the presence of chemical reaction source term that involves different time scales and introduces stiffness), except for the convective terms. For the convective term, a high-order explicit extrapolation for the velocity is used. For time discretization of the momentum equation, a semi-implicit splitting method is used. Since the velocity field responds to changes in temperature and density on a slower inertial time scale, the updated temperature and species

field is used to determine density from equation (38), divergence of the velocity field (40) and kinematic viscosity. The discretized system reads

$$\frac{1}{\Delta t}\sum_{q=0}^{J}\alpha_q T^{n+1-q} = -(\sum_{q=0}^{J-1}\beta_q \mathbf{v}^{n-q})\cdot\nabla T^{n+1} + \frac{\alpha}{RePr}\nabla^2 T^{n+1} + Da\sum_{i=1}^{N} h_i^{\circ}\dot{\Psi}_i{}', \tag{41}$$

$$\frac{1}{\Delta t}\sum_{q=0}^{J}\alpha_q Y_i^{n+1-q} = -(\sum_{q=0}^{J-1}\beta_q \mathbf{v}^{n-q})\cdot\nabla Y_i^{n+1} + \frac{D}{ReSc}\nabla^2 Y_i^{n+1} + Da\dot{\Psi}_i{}', \tag{42}$$

$$Q_T^{n+1} = \frac{1}{ReSc}\nabla^2 T^{n+1} + Da\sum_{i=1}^{N}\frac{h_i^{\circ}\dot{\Psi}_i{}'}{T^{n+1}}, \tag{43}$$

$$\frac{1}{\Delta t}\sum_{q=0}^{J}\alpha_q \mathbf{v}^{n+1-q} = -\sum_{q=0}^{J-1}\beta_q(\mathbf{v}\cdot\nabla\mathbf{v})^{n-q} - \frac{1}{\rho}\nabla p + \frac{\nu}{Re}(\nabla^2\mathbf{v} + \frac{1}{3}\nabla(\nabla\cdot\mathbf{v}))^{n+1}, \tag{44}$$

where Q_T^{n+1} is the thermal divergence of the velocity field and α_q and β_q are the coefficients of implicit and explicit parts of the J-th order time integration scheme. All quantities which are functions of temperature and mass fractions in the general case (α, D, ν, ρ and $\dot{\Psi}_i'$) are evaluated using T^{n+1} and Y_i^{n+1}. All convective terms in all equations are integrated explicitly or semi-explicitly. The time integration method is based on the backward differentiation, similar to the integration of species and energy equations. Similar to the incompressible flows, a pressure Poisson equation is derived that is accounting for the non-zero thermal divergence of the velocity field.

The integration of (44) is explicit for the non-linear convective terms and implicit for the viscous and pressure terms. It starts with the integration of the convective terms

$$\frac{\hat{\mathbf{v}}}{\Delta t} - \frac{1}{\Delta t}\sum_{q=0}^{J-1}\alpha_q\mathbf{v}^{n-q} = -\sum_{q=0}^{J-1}\beta_q(\mathbf{v}\cdot\nabla\mathbf{v})^{n-q}. \tag{45}$$

Then a variable coefficient Poisson equation is derived for the pressure p, by taking the divergence of (44).

$$\nabla\cdot(\frac{\nabla p}{\rho^{n+1}}) = \frac{\nabla\cdot\hat{\mathbf{v}} - \gamma_0(\nabla\cdot\mathbf{v})^{n+1}}{\Delta t} + \frac{1}{Re}\nabla\cdot\nu^{n+1}(\nabla^2\mathbf{v} + \frac{1}{3}\nabla(\nabla\cdot\mathbf{v}))^{n+1}. \tag{46}$$

To decouple the pressure and velocity calculation, the terms involving $\mathbf{v}^{n+1}$ in the pressure equation (46) have to be expressed in terms of the known quantities. Hence,

$$\nabla\cdot(\frac{\nabla p}{\rho^{n+1}}) = \frac{\nabla\cdot\hat{\mathbf{v}} - \gamma_0(\nabla\cdot\mathbf{v})^{n+1}}{\Delta t} + \frac{1}{Re}\nabla\cdot\nu^{n+1}(\frac{4}{3}\nabla(\nabla\cdot\mathbf{v}) - \nabla\times(\nabla\times\mathbf{v}))^{n+1}. \tag{47}$$

Now the velocity field is decomposed into two parts, irrotational and solenoidal, $\mathbf{v} = \mathbf{v_s} + \mathbf{v_I}$, where the $\mathbf{v}_I$ is treated implicitly using thermal divergence of the velocity field, (i.e. $(\nabla\cdot\mathbf{v})^{n+1} = (\nabla\cdot\mathbf{v}_I)^{n+1} \approx Q_T^{n+1}$) and $\mathbf{v}_s$ is treated explicitly

(i.e. $(\nabla \times \mathbf{v})^{n+1} = (\nabla \times \mathbf{v}_s)^{n+1} \approx \sum_{q=0}^{J-1} \beta_q \omega^{n-q}$), where ω is the vorticity. Consequently, the pressure equation is written as

$$\nabla \cdot (\frac{\nabla p}{\rho^{n+1}}) = \frac{\nabla \cdot \hat{\mathbf{v}} - \gamma_0 Q_T^{n+1}}{\Delta t} + \frac{1}{Re} \nabla \cdot \nu^{n+1} (\frac{4}{3} \nabla Q_T^{n+1} - \sum_{q=0}^{J-1} \beta_q \nabla \times \omega^{n-q}) \,. \quad (48)$$

The boundary condition for the pressure equation is derived by taking dot product of (44) in the direction normal to the boundaries $\hat{n}$. This results in a Neumann pressure boundary condition and Dirichlet boundary conditions for velocity

$$\frac{1}{\rho^{n+1}} \frac{\partial p}{\partial n} = -\frac{\partial (\hat{n} \cdot \mathbf{v})}{\partial t} - \hat{n} \cdot \sum_{q=0}^{J-1} \beta_q (\mathbf{v} \cdot \nabla \mathbf{v})^{n-q} + \frac{\nu^{n+1}}{Re} \hat{n} \cdot (\frac{4}{3} \nabla Q_T^{n+1} - \sum_{q=0}^{J-1} \beta_q \nabla \times \omega^{n-q}) \,. \quad (49)$$

The first term in the right hand side of (49) is known for Dirichlet boundary condition. The rest of the splitting scheme consists of the incorporation of the pressure correction to the velocity field and integration of the viscous part of the momentum equation in the following two steps:

$$\frac{\hat{\hat{\mathbf{v}}} - \hat{\mathbf{v}}}{\Delta t} = -\frac{\nabla p}{\rho^{n+1}} \,, \quad (50)$$

$$-\frac{\nu^{n+1}}{Re} \nabla^2 \mathbf{v}^{n+1} + \frac{\gamma_0 \mathbf{v}^{n+1}}{\Delta t} = \frac{\hat{\hat{\mathbf{v}}}}{\Delta t} + \frac{1}{3} \frac{\mathbf{v}^{n+1}}{Re} \nabla Q_T^{n+1} \,. \quad (51)$$

The splitting scheme described above gives overall high-order of accuracy in time and minimal errors in mass conservation i.e. splitting errors. It was shown that the splitting errors are always smaller than the formal truncation error $O(\Delta t^J)$ of the J-th order integration scheme [5]. To complete the discretization of the system mentioned above, a global spectral or spectral element method can be used for the spatial discretization.

7 Implementation

In this section we will go through the implementation of the Pn-Pn method with support for low-Mach reactive flows [5] in Nek5000. Overall, everything related to this implementation is located in the file `plan4.f`. In this file we can find the main subroutine `plan4` that computes one time step according to the Karniadakis scheme. This subroutine `plan4` is shown in Listing 1. For clarity, some common blocks have been omitted. Overall, the subroutine can be split into three parts, roughly equivalent to the equations for the explicit factors (11) (lines 8 - 19), the pressure (19) (lines 33 - 43) and the momentum (13) (lines 48 - 54). We will cover each of these parts from a higher level, and we will also provide a stepping point for where related subroutines are located if one wants a more detailed understanding. One thing to note, is that in Nek5000 we do not solve for $\bar{p}^{n+1}$, but rather always for the difference $\Delta p = \bar{p}^{n+1} - \bar{p}^n$, the impact of this is purely algebraic though.

7.1 Explicit terms

The first steps of computing one time step is to compute the forcing contribution from the nonlinear terms as well as the external forces, corresponding to $\hat{\mathbf{v}} = \mathbf{v}^*$. This is implemented in Nek5000 in the subroutine `makef` (located in `navier1.f`) which calculates the explicit contributions from the convection and external forces such as gravity or a set pressure gradient. In addition to this, the BDF contributions from previous velocity fields are computed on lines 10-12. With the values from makef `bfx, bfy, bfz` and the explicit velocity contributions `vx_e, vy_e, vz_e`, the right hand side of the pressure Poisson equation can then be computed.

7.2 Pressure solve

Using the values from `makef` and `sumab`, the right hand side for the pressure equation is computed in the call to `crespsp` on line 33. This subroutine is also located in `plan4.f`. In this subroutine, `crespsp`, the operations necessary to obtain the correct right hand side shown in (19), including the Neumann boundary conditions for the pressure are computed. This Neumann boundary condition is enforced by setting the appropriate values on the boundary for the right hand side. In addition, if we solve for low-Mach and reactive flows, these contributions, as shown in (48), are also taken into account. With the residual computed in `crespsp` the system is then solved in the call to `hsolve` with a preconditioned GMRES solver.

7.3 Velocity solve

Using the results from the pressure solve, the right hand side for the velocity as shown in equation (13) is then computed in `cresvspr` on line 52. This is less involved than for the pressure since we are only dependent on the explicit terms and the gradient of the new pressure field. This linear Helmholtz equation is then solved for in the call on line 54 to `ophinv` which solves for the velocity in the x, y and z directions respectively with a preconditioned CG solver.

References

[1] C William Gear. Numerical initial value problems in ordinary differential equations. *Prentice-Hall series in automatic computation*, 1971.

[2] George Em Karniadakis, Moshe Israeli, and Steven A Orszag. High-order splitting methods for the incompressible navier-stokes equations. *Journal of Computational Physics*, 97(2):414–443, 1991.

[3] Steven A Orszag, Moshe Israeli, and Michel O Deville. Boundary conditions for incompressible flows, 1986.

[4] Einar Malvin Rønquist. *Optimal spectral element methods for the unsteady three-dimensional incompressible Navier-Stokes equations*. PhD thesis, Massachusetts Institute of Technology, 1988.

[5] AG Tomboulides, JCY Lee, and SA Orszag. Numerical simulation of low mach number reactive flows. *Journal of Scientific Computing*, 12(2):139–167, 1997.

[6] F. A. Williams. *Combustion Theory*. CRC Press, 1985.

```
      subroutine plan4 (igeom)

      NTOT1  = lx1*ly1*lz1*NELV

      if (igeom.eq.1) then

         ! compute explicit contributions bfx,bfy,bfz
         call makef

         call sumab(vx_e,vx,vxlag,ntot1,ab,nab)
         call sumab(vy_e,vy,vylag,ntot1,ab,nab)
         if (if3d) call sumab(vz_e,vz,vzlag,ntot1,ab,nab)

      else

         if(iflomach) call opcolv(bfx,bfy,bfz,vtrans)

         ! add user defined divergence to qtl
         call add2 (qtl,usrdiv,ntot1)

         if (igeom.eq.2) call lagvel

         ! mask Dirichlet boundaries
         call bcdirvc  (vx,vy,vz,v1mask,v2mask,v3mask)

         ! compute pressure
         call copy(prlag,pr,ntot1)
         if (icalld.eq.0) tpres=0.0
         icalld=icalld+1
         npres=icalld
         etime1=dnekclock()

         call crespsp  (respr)
         call invers2  (h1,vtrans,ntot1)
         call rzero    (h2,ntot1)
         call ctolspl  (tolspl,respr)
         napproxp(1) = laxtp
         call hsolve   ('PRES',dpr,respr,h1,h2
     $                        ,pmask,vmult
     $                        ,imesh,tolspl,nmxp,1
     $                        ,approxp,napproxp,binvm1)
         call add2     (pr,dpr,ntot1)
         call ortho    (pr)

         tpres=tpres+(dnekclock()-etime1)

         ! compute velocity
         if(ifstrs .and. .not.ifaxis) then
            call bcneutr
            call cresvsp_weak(res1,res2,res3,h1,h2)
         else
            call cresvsp      (res1,res2,res3,h1,h2)
         endif
         call ophinv(dv1,dv2,dv3,res1,res2,res3,h1,h2,tolhv,nmxv)
         call opadd2(vx,vy,vz,dv1,dv2,dv3)

      endif
      return
      END
```

Listing 1: One step of the Karniadakis scheme as implemented in Nek5000.

Iterative solvers and projection method

Simon Kern*, Valerio Lupi† and Vitor Kleine‡

June 9, 2021

1 Iterative solvers and projection method

Consider the $n \times n$ linear system

$$Ax = b. \tag{1}$$

The computational complexity of direct algorithms to solve the linear system is $O(n^3)$. The linear systems solved in Nek5000 can easily have an order of 10^6 or greater. For such large sparse matrices, the computational cost of direct methods is prohibitively expensive. Instead, Nek5000 employs iterative methods, such as the Generalized Minimal Residual (GMRES) method and the Conjugate Gradient (CG) iteration, that rely on projection methods to find an approximate solution.

Another advantage of iterative methods is that the matrix A does not need to be formed or stored explicitly. Nek5000 is matrix-free, meaning that only the product Ax, which is of size n, is available through a function, as opposed to the matrix A which, although typically sparse, is $n \times n$ and is never explicitly formed.

This report gives an overview of projection methods, iterative solvers, preconditioners, and their implementation in Nek5000. For detailed discussion about the methods, the reader is referred to section 2.7.4 of [2], chapter IV of [8] and chapters 5 and 6 of [7].

1.1 Overview of the methods

In this section we present the theoretical foundation of projection methods, the central component of the numerical schemes used in Nek5000 to solve large, sparse linear systems of equations. Projection methods are a broad class of methods used for approximating the solution of linear systems on a subspace. In order to find the solution to the linear system of the form $Ax = b$, Nek5000 employs two projection approaches:

- Iterative solvers that are in the broad category of projection methods (such as GMRES and CG); and

*KTH Engineering Mechanics, skern@mech.kth.se
†KTH Engineering Mechanics, lupi@mech.kth.se
‡KTH Engineering Mechanics, vitok@mech.kth.se

- Explicit projection onto prior solutions of the linear system, before calling the iterative solvers.

Both GMRES and CG, described in the following sections, search for approximate solutions $\tilde{x}$ belonging to a lower-dimensional Krylov subspace. The m-order Krylov subspace, $\mathcal{K}_m(A, b)$, generated by the matrix A and vector b, is the subspace spanned by the vectors b, Ab, A^2b, ..., $A^{m-1}b$:

$$\mathcal{K}_m(A, b) = \text{span}\{b, Ab, A^2b, \ldots, A^{m-1}b\}. \tag{2}$$

The exact solution x is guaranteed to lie in a Krylov subspace if A is non-singular [5]. However, an acceptable approximation $\tilde{x}$ can be found, within a desired tolerance, far before x is found exactly, especially if efficient preconditioning is applied. Besides that, the Krylov subspace has properties that are exploited by the iterative solvers in order to find the solution efficiently.

1.1.1 GMRES

The generalized minimal residual method (GMRES) finds the approximate solution $\tilde{x}$ in the subspace $\mathcal{K}_m(A, b)$ that minimizes $\|b - Ax\|_2$ using the Arnoldi iteration.

The Arnoldi iteration (described in steps 1-7 of Algorithm 1) is a method that finds a $(m+1) \times m$ upper Hessenberg matrix $\bar{H}_m$ and a $n \times m$ orthogonal matrix $V_m = [v_1\ v_2\ \ldots\ v_m]$ whose columns represent an orthonormal basis of $\mathcal{K}_m(A, b)$, with $v_1 = b/\|b\|$ such that:

$$AV_m = V_{m+1}\bar{H}_m. \tag{3}$$

Every vector $\tilde{x}$ in $\mathcal{K}_m(A, b)$ can be written as $\tilde{x} = V_m y$ ($y \in \mathbb{R}^m$). The minimization problem becomes:

$$\begin{aligned} \min_{x \in \mathcal{K}_m} \|b - Ax\|_2 &= \min_{y \in \mathbb{R}^m} \|b - AV_m y\|_2 = \min_{y \in \mathbb{R}^m} \| v_1\|b\| - V_{m+1}\bar{H}_m y\|_2 \\ &= \min_{y \in \mathbb{R}^m} \|V_{m+1}(e_1\|b\| - \bar{H}_m y)\|_2 = \min_{y \in \mathbb{R}^m} \| e_1\|b\| - \bar{H}_m y\|_2 \end{aligned} \tag{4}$$

where $e_1 = (1, 0, 0, \ldots, 0)^T \in \mathbb{R}^{m+1}$ is the first vector of the standard basis. The approximate solution $\tilde{x}$ at step m can be calculated from:

$$\tilde{x} = V_m \tilde{y} \tag{5}$$

with

$$\tilde{y} = \arg\min_{y \in \mathbb{R}^m} \| e_1\|b\| - \bar{H}_m y\|_2. \tag{6}$$

The minimization problem (6) is a linear least-square problem. It is an overdetermined linear system of dimension $(m+1) \times m$ that is solved in Nek5000 using Givens rotation matrices G_j (described in steps 8-11 of Algorithm 1), following the original procedure of GMRES [6]. In this context, Givens rotation matrices are unitary matrices designed to transform the Hessenberg matrix $\bar{H}_m$ into the $(m+1) \times m$ matrix $\bar{R}_m = G_m \ldots G_2 G_1 \bar{H}_m$ that is an upper triangular matrix with an added row of zeros. Since G_j are unitary, the minimization problem becomes:

$$\tilde{y} = \arg\min_{y \in \mathbb{R}^m} \| e_1\|b\| - \bar{H}_m y\|_2 = \arg\min_{y \in \mathbb{R}^m} \|\gamma_m - \bar{R}_m y\|_2 \tag{7}$$

where $\gamma_m = G_m \dots G_2 G_1 e_1 \|b\|$. Also, it can be shown that the residual can be read directly from the last element of γ_m. Hence, there is no need to calculate $\tilde{y}$ at every iteration. At the end of the iterative process, the linear system $R_m \tilde{y} = \gamma_m$ is solved by backward substitution (where the square upper triangular matrix R_m is formed by the first m rows of $\bar{R}_m$). Further details on how Givens rotations are applied to GMRES can be found in [6] or section 6.5.3 of [7].

Algorithm 1 Standard GMRES iteration

1:	$\gamma = \|b\|_2 e_1$	! Initialize right hand side
2:	$v_1 = b/\|b\|_2$	! Initialize V with v_1
3:	**for** $j = 1, \dots, m$ **do**	! m-step Arnoldi factorization
4:	$w = Av_j$	! Apply operator
5:	$h_j = V^T w$	! Compute new column h_j of H (Project w onto V)
6:	$w = w - Vh_j$	! Orthogonalize against V (Project w onto $V_\perp$)
7:	$v_{j+1} = w/\|w\|_2$	! Compute new column v_{j+1} of V
8:	$r_j = G_j \cdots G_2 G_1 h_j = Qh_j$	! Compute new column r_j of R using j Givens rotators $j-1$ previous, 1 new
9:	$\gamma = G_j \gamma$	! Update right hand side
10:	**end for**	! m steps or $\|\gamma_{m+1}\| < \text{tol}$
11:	$\tilde{y} = R^{-1}\gamma$	! Compute $\tilde{y}$
12:	$\tilde{x} = V\tilde{y}$	! Compute approximate solution

1.1.2 CG

The conjugate gradient (CG) method is also a Krylov subspace iteration like GMRES with the major difference that it is derived under the assumption that A is **symmetric positive definite** (SPD). We therefore require

- symmetry: $A = A^T$,
- positive definiteness: $x^T Ax > 0, \quad \forall x \neq 0$.

The first relation has a direct implication for the Arnoldi factorisation. The upper Hessenberg matrix H_m, i.e. the projection of A on $\mathcal{K}_m(A, b)$, is given by

$$H_m = V_m^T A V_m = V_m^T A^T V_m = (V_m^T A V_m)^T = H_m^T, \tag{8}$$

since $A = A^T$. H_m is both upper Hessenberg and symmetric and therefore tridiagonal. This structure can be exploited to compute it more efficiently than using the Arnoldi iteration since we can find a three-term recurrence relation for subsequent columns in a procedure similar to the Lanczos iteration (see, e.g. Lecture 36, chapter IV in [8]).

The second relation can be used to define a matrix norm related to A such that

$$\|x\|_A = \sqrt{x^T Ax} = \sqrt{\langle x, x \rangle_A}\,, \tag{9}$$

where $\langle \cdot, \cdot \rangle_A$ denotes the A-inner product. The definition is analogous for A^{-1} which is also SPD.

The CG iteration finds the best approximation $\tilde{x}$ to x in the Krylov subspace $\mathcal{K}_m(A,b)$ w.r.t the A^{-1}-norm, i.e. it minimises the residual

$$\|r_m\|_{A^{-1}} = \|A\tilde{x} - b\|_{A^{-1}} = \min_{x\in\mathcal{K}_m} \|Ax - b\|_{A^{-1}}, \quad \forall m \geq 1 \tag{10}$$

In particular, this definition of the residual allows us to reformulate the minimisation problem as follows. Squaring both sides, we obtain

$$\|r_m\|^2_{A^{-1}} = \min_{x\in\mathcal{K}_m} \|Ax - b\|^2_{A^{-1}} = \min_{x\in\mathcal{K}_m} (Ax-b)^T A^{-1}(Ax-b) \tag{11}$$

$$= \min_{y\in\mathbb{R}^m} (AV_m y - b)^T A^{-1}(AV_m y - b) \tag{12}$$

$$= \min_{y\in\mathbb{R}^m} y^T V_m^T A V_m y - 2b^T V_m y + b^T A^{-1} b \tag{13}$$

$$= \min_{y\in\mathbb{R}^m} (AV_m y - 2b)^T V_m y + b^T A^{-1} b, \tag{14}$$

where in (12) we have used that $x \in \mathcal{K}_m(A,b) \Leftrightarrow \exists y \in \mathbb{R}^m, x = V_m y$. The result is an unconstrained quadratic optimisation problem where the gradient and the hessian with respect to y are given by $p = 2(AV_m y - b)^T V_m$ and $h = 2V_m^T A V_m$, respectively. With $V_m^T A V_m$ being SPD since A is SPD, the hessian is also SPD so that the local optimality condition (i.e. $p = 0$) becomes the optimality condition for the global minimum and we have

$$2(AV_m y - b)^T V_m = 0 \qquad \Longrightarrow \qquad r_m^T V_m = 0, \tag{15}$$

which means that the residual is always orthogonal to the search space $\mathcal{K}_m(A,b)$.

Combining the orthogonality of the residual with the recurrence relations for the current solution $\tilde{x}$, the residual r_j and the gradient p_j to their values at the previous timestep, we can define the CG iteration. The standard (unoptimised) CG iteration given in Algorithm 2 shows the recurrence relations for $\tilde{x}, r_j$ and p_j in steps 4,5 and 7, respectively, involving the coefficients α_j (step 3) and β_j (step 6) recomputed at each step. For more details on the derivation of this relation we refer to Lecture 38 in chapter IV in [8] or section 6.7 in [7]. In addition to relation (15), it can be shown that $\forall i \neq j$ we have

$$r_i \perp r_j, \tag{16}$$

$$p_i \perp_A p_j, \tag{17}$$

where the second condition relates the gradients that are A-orthogonal or A-conjugate, thus giving the method its name.

The important point here is that, unlike GMRES, we only need to store 3 vectors in memory (namely $\tilde{x}$, r_j and p_j) because the orthogonality conditions are enforced implicitly without requiring the costly orthogonalisation w.r.t a growing basis. Furthermore, the recurrence relation implies that the cost per timestep is constant thus avoiding the need to restart. The corresponding advantages in terms of memory requirements and computational cost make CG the go-to method for the iterative solution of linear systems. Only when the assumptions on A are not met one must resort to other methods, in the case of Nek5000, restarted GMRES.

Algorithm 2 Standard CG iteration [4]

1: $x_0 = 0, r_0 = b, p_0 = r_0$! Initialize vectors
2: **for** $j = 1, \ldots, m$ **do**
3: $\quad \alpha_j = r_{j-1}^T r_{j-1} / p_{j-1}^T A p_{j-1}$! Compute step length
4: $\quad \tilde{x} = \tilde{x} + \alpha_j p_{j-1}$! Compute current solution
5: $\quad r_j = r_{j-1} - \alpha_j A p_{j-1}$! Update residual
6: $\quad \beta_j = r_j^T r_j / r_{j-1}^T r_{j-1}$! Compute residual improvement
7: $\quad p_j = r_j + \beta_j p_{j-1}$! Update search direction
8: **end for** ! `maxit` steps or $\|r_m\| <$ tol.

1.1.3 Preconditioning

Iterative solvers may suffer from low convergence rate for matrices that arise from many applications, such as fluid dynamics simulations [7]. Preconditioning can improve both robustness and efficiency of the iterative methods. Left preconditioning consists of multiplying both sides of the system by $n \times n$ matrix P^{-1} and solving for the equivalent preconditioned system

$$P^{-1}Ax = P^{-1}b. \tag{18}$$

Right preconditioning is equivalent to solving

$$AP^{-1}u = b \qquad \text{and} \qquad x = P^{-1}u \tag{19}$$

The choice of P should make the matrix $P^{-1}A$ (or AP^{-1}) better conditioned for the iterative methods, in the sense that the number of iterations is reduced. For the preconditioning to be effective in terms of reducing the time to solution, applying P^{-1} should be cheap in comparison to applying A^{-1}.

For diagonally dominant matrices typical of fluid dynamics applications, one option is the Jacobi (or diagonal) preconditioner $P := \texttt{diag}(A)$, where the matrix P is a diagonal matrix formed by the elements on the main diagonal of A. Other types of preconditioners are shown in Lecture 40 of [8] and chapters 9 and 10 of [7].

In the following we will not focus on preconditioning but rather treat the generic preconditioning matrix P as a black box. For more details on preconditioning in Nek5000 we refer to the reports by group G7.

1.1.4 Projection onto prior solutions

Integrating the Navier-Stokes equations in time leads at every timestep to the solution of linear systems of the form $Ax = b$ where both the operator A and the inhomogeneity b, and hence the solution, changes little from one timestep to the next. One way to exploit this structure in the solution process is to project the current solution onto prior solutions and to solve only for the solution update. In most cases this leads to a considerable reduction of iteration count of the iterative solvers.

Assuming a set of k solutions $X_p = [x_1, \ldots, x_k]$ and right hand sides $B_p = [b_1, \ldots, b_k]$ are known (such that $Ax_j = b_j$), any linear combination $\bar{x} = \alpha X_p = \alpha_1 x_1 + \cdots + \alpha_k x_k$ of these vectors is also a solution of the linear system

$$A\bar{x} = A(\alpha_1 x_1 + \cdots + \alpha_k x_k) = \alpha_1 b_1 + \cdots + \alpha_k b_k = \alpha B_p = \bar{b}. \tag{20}$$

Hence, finding the solution of $Ax = b$ is equivalent to finding the solution of

$$A\delta x = r, \tag{21}$$

where $\delta x = x - \bar{x}$ is the solution update and $r = b - \bar{b}$ is the corresponding right hand side.

In practice, different options are available for the projection step. In Nek5000, x is projected onto the column space of X_p using the A-inner product (for SPD A) reflecting its relevance for the problem that we already encountered in the Krylov subspace methods, in particular for CG iteration. This particular choice of norm will also prove useful in the practical aspects of the projection technique discussed below. Since the subsequent solutions x_k are typically close to linearly dependent, instead of the ill-conditioned matrix X_p, an A-orthogonal basis X is constructed using an oblique Gram-Schmidt (GS) process to ensure numerical stability. The basis for the column space of B_p is then chosen as $B = AX$ to ensure that $\bar{b} = A\bar{x} \in \text{span}\{B\}$.

The projection of the unknown x onto X using the A-inner product is equivalent to projecting the known b onto X using the Euclidian inner product:

$$\alpha = \langle x, X\rangle_A = X^T Ax = X^T b = \langle b, X\rangle_2, \tag{22}$$

which allows the A-orthogonal projection to be carried out without explicit knowledge of x and without an additional call to "Ax". This can also be interpreted as the projection of b onto B using the A^{-1}-inner product:

$$\langle b, B\rangle_{A^{-1}} = B^T A^{-1} b = (A^{-1}B)^T b = X^T b. \tag{23}$$

The equivalent linear system (21) can then be solved efficiently using Krylov subspace iteration as described in the previous sections. The complete solution is then reconstructed from the solution update δx and the linear combination of the previous solutions: $x = \delta x + \bar{x}$. Algorithm 5 shows the steps of the projection method as it is implemented in Nek5000. Once the new solution is found, the basis of previous solutions is updated to include it. Further details can be found in [3] and [1].

1.2 GMRES, CG and projections in Nek5000

In this section we describe how the theoretical tools for the iterative solution of large linear systems are combined in practice in Nek5000. The general workflow of the solvers in Nek5000 is shown in Fig. (1). The projection and reconstruction steps can be seen as pre- and postprocessing of the linear system, respectively, that generates the input to the inner preconditioned Krylov subspace iterations performed with GMRES or CG.

1.2.1 Right-preconditioned restarted GMRES iteration

The pseudo-code reflecting important details of the implementation of the right-preconditioned GMRES(m) in Nek5000 is given in Algorithm 3.

Implementation notes for GMRES:

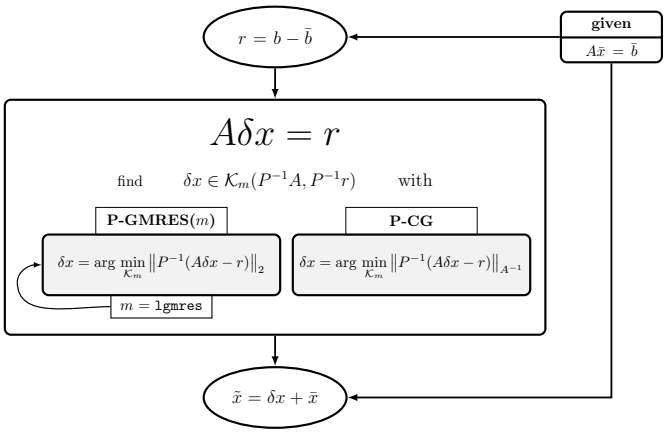


Figure 1: Diagram of the components of the iterative solution process for linear systems $Ax = b$ in Nek5000 including projections followed by preconditioned Krylov subspace iteration. $\tilde{x}$ is the approximate solution of the linear system, $\bar{x}$ is the (oblique) projection of x onto the space of previous solutions with $A\bar{x} = \bar{b}$, r is the residual after projection, δx is the corresponding solution update, $\mathcal{K}_m(A, b)$ is the mth Krylov subspace of A applied to b and P is a generic preconditioning matrix.

- L, U (lines 3,8) is a secondary diagonal preconditioner ("Uzawa splitting") with $I = LU$ based on the mass matrix B.

  `uzawa_gmres` $L = \sqrt{B^{-1}}$ and $U = \sqrt{B}$

  `hmh_gmres` $L = U = I$ (void)

- The Givens rotations (lines 12-13) are not computed as explicit matrix products but using atomic Givens rotations acting on 2 vector elements at a time.

- The GMRES algorithm is restarted after $m =$ `lgmres` steps using the last residual as a starting vector. `lgmres` is set in `SIZE` and should not exceed about 40 for performance reasons.

- The approximate solutions (x^*) of the subsequent Arnoldi factorizations are additive to form the final approximation $\tilde{x}$ since the residual is set to $r - A\tilde{x}$ at every restart (i.e. removing contributions from previous restarts).

- Maximum (inner + outer) iteration count `maxit` = 100 for `uzawa_gmres` (hardcoded in `gmres.f`) and `maxit` = 200/1000 for `hmh_gmres` (hardcoded in `drive2.f` for transient/steady case and passed to the solvers as arguments).

- Orthogonalization (lines 10-11) is carried out via Classical Gram-Schmidt (CGS). Double GS (DGS) and modified GS (MGS) is also implemented but commented out.

- The vector norm $\|\cdot\|$ is the Euclidean norm computed using the standard global inner product. On the pressure mesh the inner product is

Algorithm 3 Right-preconditioned restarted GMRES iteration for $A\tilde{x} = b$

```
1:  x̃ = 0, r_0 = b                          ! Initialize vectors
2:  while not done do                        ! Until convergence or maxit reached
3:      r = L r_0                            ! Apply L (= U^{-1})
4:      r = r − A x̃                          ! Subtract known solution (for restart)
5:      γ = ‖r‖ e_1                          ! Initialize right hand side
6:      v_1 = r/‖r‖                          ! Initialize V with v_1
7:      for j = 1, ..., m do                 ! m-step Arnoldi factorization
8:          z = P^{-1} U v_j                 ! Apply U and P (preconditioners)
9:          w = A z                          ! Apply operator
10:         h_j = V^T w                      ! Compute new column h_j of H
11:         w_⊥ = w − V h_j                  ! Project w onto V_⊥
12:         r_j = G_j ··· G_1 h_j = Q^T h_j  ! Compute new column r_j of R
                                                 using j Givens rotators
13:         γ = G_j γ                        ! Update right hand side
14:         v_{j+1} = w_⊥/‖w_⊥‖_2            ! Compute new column v_{j+1} of V
15:     end for                              ! m = lgmres steps
16:     y = R^{-1} γ                         ! Compute y
17:     x̃ = x̃ + V y                          ! Update solution vector
18:     if ‖w‖ < tol then
19:         Done.                            ! GMRES converged
20:     else if maxit reached then
21:         Done.                            ! GMRES diverged
22:     else
23:         Restart iteration with r_0 := x̃
24:     end if
25: end while
```

unweighted (`wt` = 1) while on the velocity mesh the multiplicity of the element boundaries needs to be taken into account (`wt` = `vmult`)[1].

- For the pressure equation, the average is subtracted using the subroutine `ortho` (orthogonalization with respect to null space of E).

1.2.2 Preconditioned CG iteration

The pseudo-code reflecting important details of the implementation of P-CG in Nek5000 is given in Algorithm 4. A major difference to the standard CG algorithm is the possibility of using a lagged residual (flex-CG).

Implementation notes for CG:

- The inner products (lines 8, 10) are computed using the Euclidean inner product. On the pressure mesh the inner product is unweighted (`wt` = 1) while on the velocity mesh the multiplicity of the element boundaries need to be considered (`wt` = `vmult`)[1].

[1] A commonly used quantity is the energy norm that is computed using the weighted Euclidean inner product using the mass matrices `bm1`/`bm2` as weight. These matrices already account for mesh multiplicities where necessary.

Algorithm 4 Right-preconditioned (flexible) CG iteration for $A\tilde{x} = b$

1: $\tilde{x}_0 = 0, r_0 = b, p_0 = 0, \rho_1 = 1$! Initialize vectors, ρ_1
2: $z_0 = P^{-1} r_0$! Initialize lagged residual (flex)
3: **for** $j = 1, \ldots,$ `maxit` **do**
4: $z_j = P^{-1} r_j$! Apply preconditioner P
5: **if** *flex* **then**
6: $z_j = z_j - z_{j-1}$! Use lagged residual
7: **end if**
8: $\rho_2 = r_j^T z_j$! Update residual norm
9: $\beta = \rho_2 / \rho_1$! Compute residual improvement
10: $\rho_1 = r_{j-1}^T z_{j-1}$! Lagged residual norm
11: $p_j = p_{j-1} + \beta z_j$! Update search direction
12: $w_j = A p_j$! Compute step length
13: $\alpha_j = \rho_1 / p_j^T w_j$! using $\langle p_j \rangle_A$
14: $\tilde{x}_j = \tilde{x}_{j-1} + \alpha_j p_j$! Update solution
15: $r_{j+1} = r_j - \alpha_j w_j$! Update residual
16: **if** $\|r_{j+1}\|_m <$ tol **then**
17: break ! CG converged
18: **end if**
19: **end for**

- The A-inner product is computed sequentially (lines 12-13) using the Euclidean inner product:

$$\langle p_j, w_j \rangle_2 = p_j^T w_j = p_j^T A p_j = \langle p_j, p_j \rangle_A. \tag{24}$$

1.2.3 Implementation of the projection scheme

Algorithm 5 Projection technique for the solution of $Ax_n = b_n$

1: Given $X = [\hat{x}_1, ..., \hat{x}_k]$! A-orthonormal basis of prior solutions
2: Given $B = [\hat{b}_1, ..., \hat{b}_k] = AX$! RHS corresponding to X
3: $\alpha = X^T b_n$! Oblique projection of x_n onto X
$\alpha = \langle x_n, X \rangle_A = \langle b_n, X \rangle_2$
4: $\bar{x} = \alpha X, \bar{b} = \alpha B = A\bar{x}$! Compute linear combinations $\bar{x}, \bar{b}$
5: $r = b_n - \bar{b}$! Define residual s.t. $r \perp X$
6: $A\delta x = r$! Solve for solution update δx
7: $x_n = \delta x + \bar{x}$! Reconstruct solution x_n
8: $\hat{b} = A x_n$! Recompute exact $\hat{b}$ corresponding
to approximate solution x_n
9: Update X, B ! A-orthonormalize x_n against X.
Update B with corresponding
transformed version of $\hat{b}$

Implementation notes for the projections:

- Projections are disabled for the first 5 timesteps.

- Three different implementations of projections exist in Nek5000 (see subroutine glossary):

  1. Standard velocity projections
  2. Velocity projections for the stress formulation (coupled velocity fields)
  3. Pressure projections

  where the velocity projections are mutually exclusive (they have overlapping common blocks!) but are independent of the pressure projections. In fact, projections are typically used for both velocity and pressure simultaneously.

- All projection schemes in Nek5000 assume A to be SPD (since the A-inner product is used).

- Velocity projections ($A = H$):

  - Size of projection space is set to `mmx = (mxprev-4)/2` (line 1174 in `navier4.f`).
  - $\bar{b}$ is computed as $\bar{b} = \alpha B$ on step 4. No additional call to "Ax" is required.
  - `h1` and `h2` (from the last timestep to determine a change in H) as well as X and B are stored contiguously in memory (in this order) and accessed via the same variable `rvar`.
  - Update and reorthonormalization of the basis (steps 8-9) are performed according to [1]. Some of the implementation details are:
    * If H changes, the column spaces of X and B are fully reorthonormalized w.r.t the new H before projection (steps 1 and 2).
    * The newest solution is prepended to the space X and the oblique QR factorization is completed performing an orthogonal transformation via a matrix H built using Givens rotations.
    * If the new vectors are linearly dependent with respect to the previous ones, they are redundant and discarded.
    * Since the solution update δx is only computed up to a set solver tolerance, instead of the b_n, the exact right hand side $\hat{b} = Ax_n$ corresponding to the approximate solution $x_n = \delta x + \bar{x}$ is used to update B (step 8).
    * The weights for the projection during the basis update are computed as $\alpha = \frac{1}{2}(X^T b + B^T x)$ since the two terms are mathematically identical and neither basis is inherently better suited:

$$X^T b = X^T A x = X^T A^T x = (AX)^T x = B^T x. \tag{25}$$

- Pressure projections ($A = E$):

  - Size of pressure projection space is `mxprev` set in `SIZE`.
  - Only the basis of prior solutions X is stored. $\bar{b}$ is computed as $\bar{b} = A\bar{x}$ on step 4. Steps 2 and 8 do not apply.
  - When the projection space is full, X is discarded entirely and restarted with current solution.

  - The constant pressure mode (null space of E operator) removed using subroutine `ortho`.

- The orthonormalization procedure is performed differently for each set of projection routines (see subroutine glossary).

1.2.4 Subroutine glossary

This section gathers the subroutines for "Ax" operations and the iterative solvers as well as projection and oblique orthogonalization routines used in Nek5000 including their location in the core. For the "Ax" routines and the iterative solvers we differentiate between the subroutines that contain the actual code and "wrappers" which represent an intermediate call layer.

1. **Matrix-vector multiplies (Ax)**

   a) Helmholtz equation

   $$H = h_1 A + h_2 B \tag{26}$$

`axhelm`	"Standard" Helmholtz operator	`hmholtz.f`
`axhmsf`	Stress formulation, coupled Helmholtz	`sub1.f`
`hxdg`		`hmholtz.f`
`hxdg_surfa`		`hmholtz.f`

   b) Pressure equation

   $$E = \sum_{i}^{ldim} D_i H^{-1} D_i^T \tag{27}$$

`cdabdtp`[2]		`navier1.f`

   c) Wrappers: `ax`, `axstrs`, `axstrs_nds`, `ophx`, ...

2. **Iterative solvers**

   a) P-GMRES(m)

`uzawa_gmres`	Pressure equation	`gmres.f`
`hmh_gmres`	Helmholtz equation	`gmres.f`

   b) P-CG

`uzawa`	Pressure equation	`navier1.f`
`cggo`[3]	Helmholtz equation	`hmholtz.f`
`hmh_flex_cg`	Helmholtz equation	`hmholtz.f`
`cggosf`	Coupled Helmholtz equations	`subs1.f`
`cggo_dg`	Helmholtz equation	`hmholtz.f`

   c) Wrappers: `hmholtz`, `hmholtz_dg`, `incomprn`, `ophinv`, `esolver`, `hmhzpf`, `hsolve`, `laplacep`, `hmhzsf`, `cggo`[3]

3. **Projection, reconstruction and oblique GS routines**

[2] There are 3 choices (input argument `intype`) to control the structure of $H = h_1 A + h_2 B$ within the operator E by varying h_1 and h_2.

[3] contains both solver code itself and branches with calls to other solvers, in particular to GMRES (despite the name).

a) Velocity projections (standard)

`project1_a`	Steps $3-5$, `proj_ortho`	`navier4.f`
`project2_a`	Steps $7-9$, `proj_ortho`	`navier4.f`
`proj_ortho`	H-orthogonal double GS (MGS option available)	`navier4.f`

b) Velocity projections (stress formulation)

`strs_project_a`	Steps $3-5$, `strs_orthok`	`subs1.f`
`strs_project_b`	Steps $7-9$, no reorthonormalization	`subs1.f`
`strs_orthok`	H-orthogonal modified GS	`subs1.f`

c) Pressure projections

`setrhsp`	Steps $3-5$, Classical GS (hardcoded)	`induct.f`
`gensolnp`	Steps 7 and 9, `econjp`	`induct.f`
`econjp`	E-orthogonal classical GS (see [3])	`induct.f`

4. **Preconditioners**

   The preconditioners are either hardcoded for the specific solver routine or accessible via parameters in the `.par`-file. For details, see the reports by group G7.

References

[1] Nicholas Christensen. Efficient projection space updates for the approximation of iterative solutions to linear systems with successive right hand sides. Master's thesis, University of Illinois at Urbana-Champaign, 2017.

[2] M. O. Deville, P. F. Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge, 2002.

[3] Paul F Fischer. Projection techniques for iterative solution of $Ax = b$ with successive right-hand sides. *Computer methods in applied mechanics and engineering*, 163(1-4):193–204, 1998.

[4] M.R. Hestenes and E Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.

[5] Ilse CF Ipsen and Carl D Meyer. The idea behind krylov methods. *The American mathematical monthly*, 105(10):889–899, 1998.

[6] Youcef Saad and Martin H Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[7] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[8] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

Time integration in Nek5000

Ananda Subramani Kannan* and Timofey Mukha†

June 28, 2021

Contents

*Chalmers University of Technology, Department of Mechanics and Maritime Sciences ananda@chalmers.se

†KTH Royal Institute of Technology, Department of Engineering Mechanics, tmu@kth.se

1 Introduction

This report provides an overview of the time integration strategies employed in the spectral-element code `Nek5000`. The material is split into three parts. In Section 2, basic theory of linear multistep methods is presented, focusing on Adams-Bashforth and backward differencing schemes. A deliberate choice has been made to derive all schemes via the introduction of associated interpolating polynomials (which is somewhat atypical for the case of backward differencing). Fundamental properties of time integration schemes—consistency, stability, and convergence—are also discussed, and linear stability analysis of the introduced methods is performed. A large amount of the material in the Section is directly based on the classical textbook on higher order methods by Deville et al. [3], which is recommended for a deeper dive in the presented topics.

Section 3 deals with more specialized time-integration techniques present in `Nek5000`, such as high-order extrapolation of the convective term and the Operator-Integration Factor Scheme. Furthermore, the previously introduced schemes are generalized to support a variable time step. Finally, the last two subsections are dedicated to expose how time-stepping interacts with the spectral element-based spatial discretization used in `Nek5000` to obtain a fully discrete problem.

The concluding section gives an overview of the implementation of the discussed methods and provides pointers to specific functions in the `Nek5000` codebase.

2 Fundamental properties of linear multistep methods

2.1 General form of a linear multistep method

The exposition of time-integration schemes will be based on the following model problem:

$$\frac{du}{dt} = f(u,t), \tag{1}$$

where, $u(t)$ is an unknown scalar function of time, and $f(u,t)$ is some—generally non-linear—function. Below, the time-derivative will also be denoted using a prime, when convenient, i.e. u'.

Here, we will focus on methods for solving (1) falling under the category of so-called linear multi-step methods. As the name implies, such numerical schemes are based on linear combinations of the values of u and f taken at multiple time steps. Mathematically, this can be expressed as

$$\frac{\tilde{u}^{n+1} - \sum_{j=1}^{k} a_j \tilde{u}^{n+1-j}}{h} = \sum_{j=0}^{k} b_j f^{n+1-j},$$

where k is the number of known values of u and f the method is based on, $\tilde{u}$ indicates a numerically approximated value of u, with the convention $\tilde{u}^{n+1-j} := \tilde{u}\left(t^{n+1-j}\right), f^{n+1-j} := f\left(\tilde{u}^{n+1-j}, t^{n+1-j}\right)$, and h is the time step, which will for now be considered constant. Commonly, the above equation is rewritten in the following form.

$$\tilde{u}^{n+1} - h b_0 f^{n+1} = \sum_{j=1}^{k} a_j \tilde{u}^{n+1-j} + h \sum_{j=1}^{k} b_j f^{n+1-j}, \tag{2}$$

The coefficients a_j and b_j depend on the choice of the numerical scheme. Note that $f^{n+1} = f\left(\tilde{u}^{n+1}, t^{n+1}\right)$ is unknown. This makes the coefficient b_0 special. If $b_0 = 0$ then $\tilde{u}^{n+1}$ is readily computed from known quantities. Such schemes are referred to as *explicit*. On the other had, if $b_0 \neq 0$, eq. (2) constitutes a system of equations that has to be solved. Quite generally, this extra price is compensated by increased stability, a concept that we will be explained in Section 2.4.

Based on the above, we can distinguish the following subclasses of the linear multistep method:

- If $k = 1$ and $b_0 = 0 \implies$ Single-step explicit methods
- If $k = 1$ and $b_0 \neq 0 \implies$ Single-step implicit methods
- If $k > 1$ and $b_0 = 0 \implies$ Multi-step explicit methods
- If $k > 1$ and $b_0 \neq 0 \implies$ Multi-step implicit methods

Some commonly used representatives of these classes are shown in Fig. 1. In the following sections we will discuss Adams-Bashforth and Backward differencing schemes in detail since these are actually used in `Nek5000`.

An important limitation of multi-step methods is that these methods need starting procedures, since several numerical values are needed to advance the time-marching process, while only the initial condition is given. Usually, this is accomplished using a single-step method, or using a different class of methods, such as Runge–Kutta.

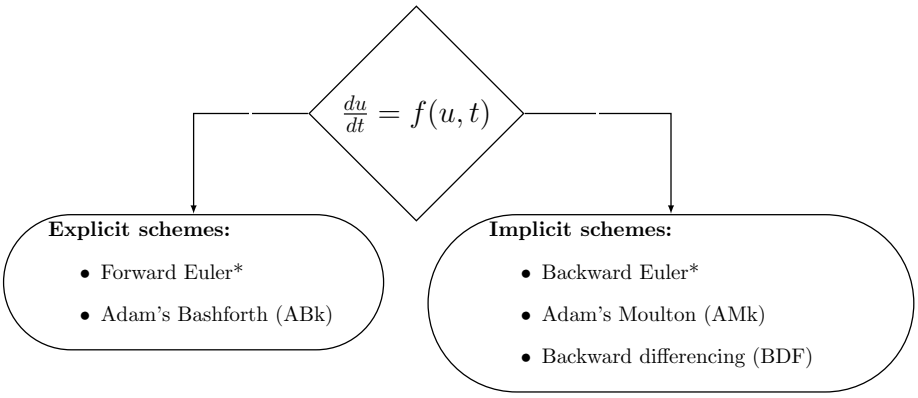


Figure 1: Overview of the common strategies and the corresponding numerical schemes used for temporal discretization. The '*' superscript represents the single-step methods. Further, the single-step forward and backward Euler methods are a first-order specializations of Adams-Bashforth and Adams-Moulton schemes, respectively.

2.2 Adams-Bashforth schemes (*ABk*)

Adams-Bashforth is a set of explicit schemes based on the observation that eq. (1) formally admits the following solution

$$u(t) - u(t_0) = \int_{t_0}^{t} f(u,s)ds, \tag{3}$$

A k^{th}-order numerical scheme is constructed by applying this at each time step and approximating f by a Newton polynomial N_{k-1} built on the values of f at k previous time steps.

$$\tilde{u}^{n+1} - \tilde{u}^{n} = \int_{t^n}^{t^{n+1}} N_{k-1}dt. \tag{4}$$

At this point, it is recommended that the reader checks Appendix Section 5, which provides a review of Newton polynomials. Considering $k = 1, 2$, we adapt the formulae in eq. (58) found in the appendix to get:

$$\begin{aligned} N_0(t) &= f^n, \\ N_1(t) &= f^n + \frac{f^n - f^{n-1}}{h}\left(t - t^n\right). \end{aligned} \tag{5}$$

These polynomials are substituted into eq. (4) to correspondingly yield first and second order approximations for $u(t)$. The first order accurate AB, a.k.a. forward Euler, approximation is written as:

$$\tilde{u}^{n+1} - \tilde{u}^n = \int_{t^n}^{t^{n+1}} N_0 dt = h f^n, \tag{6}$$

while the second order AB approximation (AB2) is written as:

$$\tilde{u}^{n+1} - \tilde{u}^n = \int_{t^n}^{t^{n+1}} P_1 dt = f^n h + \frac{h}{2}\left(f^n - f^{n-1}\right) = \frac{3h}{2} f^n - \frac{h}{2} f^{n-1}. \tag{7}$$

It is instructive to reflect on how these schemes fit into the general linear multi-step template given by eq. (2). As expected, $b_0 = 0$ since the scheme is explicit, and the only non-zero a_j is $a_1 = 1$. The coefficients b_j, $j \neq 0$ depend on the order of the scheme. For example, for the second-order scheme $b_1 = 1.5$ and $b_2 = -0.5$.

Finally, note how the potential non-linearity of f poses no threat. This is an important advantage of explicit methods!

2.3 Backward differencing schemes (*BDFk*)

A family of implicit schemes is now considered, referred to as the backward differencing schemes of order k (*BDFk*). Again, the Newton polynomial N_k is put to use, but instead of building an interpolant for f, one is built for u. The interpolant is constructed starting with the unknown value $\tilde{u}^{n+1}$ at t^{n+1} and then incorporating known values at previous time steps to increase the order of accuracy. The approximation of u' is obtained by taking the derivative of the polynomial and evaluating its value at $t = t^{n+1}$. The right-hand-side f is evaluated at t^{n+1} owing to the implicit nature of the method. According to the above, the scheme can be expressed as:

$$N_k'(t^{n+1}) = f^{n+1}. \tag{8}$$

Note that a k^{th}-order scheme is achieved with a k^{th}-order polynomial, unlike *ABk*, where N_{k-1} was sufficient.

When writing out the polynomials, we should take into account that points are added going backwards in time to get the correct signs:

$$\begin{aligned} N_0(t) &= \tilde{u}_{n+1} \\ N_1(t) &= \tilde{u}^{n+1} + \frac{\tilde{u}^{n+1} - \tilde{u}^n}{h}\left(t - t^{n+1}\right) \\ N_2(t) &= \tilde{u}^{n+1} + \frac{\tilde{u}^{n+1} - \tilde{u}^n}{h}\left(t - t^{n+1}\right) + \frac{\tilde{u}^{n+1} - 2\tilde{u}^n + \tilde{u}^{n-1}}{2h^2}\left(t - t^n\right)\left(t - t^{n+1}\right) \end{aligned} \tag{9}$$

Differentiating these polynomials yields:

$$
\begin{aligned}
N_0'(t) &= 0, \\
N_1'(t) &= \frac{\tilde{u}^{n+1} - \tilde{u}^n}{h}, \\
N_2'(t) &= \frac{\tilde{u}^{n+1} - \tilde{u}^n}{h} + [(t - t_{n+1}) + (t - t_n)] \frac{\tilde{u}^{n+1} - 2\tilde{u}^n + \tilde{u}^{n-1}}{2h^2}.
\end{aligned} \tag{10}
$$

Out of these, $N_1'(t^{n+1})$ and $N_2'(t^{n+1})$ can respectively be used as an approximation for $\tilde{u}'^{,n+1}$. For a first-order scheme *BDF1*, we use N_1:

$$
N_1'(t^{n+1}) = \frac{\tilde{u}_{n+1} - \tilde{u}_n}{h} = f^{n+1} \Longleftrightarrow \tilde{u}_{n+1} = \tilde{u}_n + hf^{n+1}, \tag{11}
$$

which is also commonly referred to as backward Euler.

For $BDF2$ we make use of N_2:

$$
\begin{gathered}
N_2'(t^{n+1}) = \frac{3\tilde{u}^{n+1} - 4\tilde{u}^n + \tilde{u}^{n-1}}{2h} = f^{n+1} \Longleftrightarrow \\
\frac{3}{2}\tilde{u}^{n+1} = 2\tilde{u}^n - \frac{1}{2}\tilde{u}^{n-1} + hf^{n+1}.
\end{gathered} \tag{12}
$$

The *BDF3* scheme is constructed similarly using N_3.

However, this is not the end of the story, because if f is non-linear, the resulting equations are non-linear as well. What is typically done is some sort of linearization of f. One possibility is using a Taylor expansion:

$$
f\left(\underline{u}^{n+1}\right) \approx f\left(\tilde{u}^n\right) + J\left(\tilde{u}^n\right)\left(\tilde{u}^{n+1} - \tilde{u}^n\right), \tag{13}
$$

where $J = df/du$, but more generally for the case of a system of equations it is the Jacobian matrix. For details, see Section 3.2 in [3]. There are other options, but all of them involve either loss of accuracy or the necessity for an iterative procedure, in which the linear system is solved multiple times to resolve the non-linearity correctly. Non-linearities are thus difficult to handle with implicit schemes.

In conclusion, it is interesting to see how the constructed schemes fit into the general linear multistep template eq. (2). We see that b_0 is obtained by dividing 1 by the coefficient in front of $\tilde{u}^{n+1}$, whereas b_j, $j \neq 0$ are all equal to zero. On the other and, the coefficients a_j depend on the order of accuracy k.

2.4 Necessary properties of a time integration method

A time-integration scheme can only be expected to approach the true solution of the model problem (eq. (1)) if it satisfies three fundamental properties.

I *Consistency:* A k-step method is consistent of order r if the local truncation error is $O(h^{r+1})$. Consistency compels the solution of to satisfy the difference scheme (2) aside from the small truncation errors that vanish with step size.

II *Convergence:* A k-step method is convergent if the estimated solution $\tilde{u}(t)$ approaches the true solution as follows:

$$\lim_{h \to 0} \tilde{u}^n = u(t), \tag{14}$$

and, similarly, the set of starting values $\tilde{u}_i^0$ approach the initial condition in the same limit:

$$\lim_{h \to 0} \tilde{u}_i^0 = u^0 \tag{15}$$

III *Stability:* A k-step method is stable when small perturbations in the initial conditions elicit only small changes in the solution (prevents truncation errors or round-off errors from amplifying.)

The relationship between stability and convergence, hinted at by Courant-Friedrichs and Lewy (CFL) [2], was brought into an organized form by Lax and Richtmyer [7] through the well-known Lax equivalence theorem:

"*For a well-posed initial-value problem and a consistent discretization scheme, stability is the necessary and sufficient condition for convergence.*"

Consequently, defining the stability region for a numerical discretization is of paramount importance while evaluating their applicability to deal with the typical problems in fluid mechanics.

2.5 Linear stability analysis based on the model equation

There is no straight-forward way for analyzing the stability of a non-linear equation, such as (1). Therefore, the stability of schemes is commonly evaluated using a linearized version of f, which can be obtained via eq. (13). The considered model equation thus has the following form:

$$\frac{du}{dt} = \lambda u(t), \quad u\left(t^0\right) = 1, \quad t \in \left(t^0, T\right], \tag{16}$$

where λ is a complex constant. Unfortunately, linear stability of a method, does not necessarily imply its stability when applied to the original non-linear equation.

In the context of eq. (16), the region of stability can be defined as:

"*The set of values of h and of λ for which any perturbation to the solution introduced at any instant will not be amplified at later times.*"

It is instructive to demonstrate that (16) remains relevant when the method is applied to a system of equations, meaning that $u(t) = \underline{u}(t)$ is a vector of unknowns. In this case, the linearized problem is generally expressed as:

$$\frac{d\underline{u}}{dt} = L\underline{u}, \tag{17}$$

where L is a linear operator.

Assuming that L can be diagonalized, eq. (17) can be transformed to the following form:

$$\frac{d\underline{\hat{u}}}{dt} = \Lambda\underline{\hat{u}}. \tag{18}$$

Here, $Z = (z_1, ..., z_N)$ is the matrix of eigenvectors of L, and the diagonal matrix $\Lambda = diag(\lambda_k)$ holds the eigenvalues of L. Eq. (18) is obtained through the substitutions: $\underline{\hat{u}} = Z^{-1}\underline{u}$ and $\Lambda = Z^{-1}LZ$.

As a result of the applied transformation, eq. (18) has been decomposed into a set of problems described by eq. (16): $u' = \lambda_k u$. Clearly, the employed scheme should remain stable for all λ_k.

In fluid mechanics, the nature of the eigenvalues is well understood. Diffusion contributes a negative real component to λ, while convection contributes a purely imaginary component. Furthermore, the second-order elliptic operator that governs diffusion scales with $O(N^4)$ for spectral discretizations of degree N, while the eigenvalues for the convection operator scale as $O(N^2)$ [3].

2.6 Region of stability for *ABk* and *BDFk* schemes

By applying the linear multistep method eq. (2) to the model problem eq. (16), the following k^{th}-order homogeneous difference equation with constant coefficients for u is obtained:

$$(1 - b_0\lambda h)\,\tilde{u}^{n+1} = \sum_{j=1}^{k} (a_j + b_j\lambda h)\,\tilde{u}^{n+1-j}. \tag{19}$$

To determine the region of stability, we replace $\tilde{u}$ with a new unknown ζ, which is a complex number. The superscript is then treated as a power rather than time level. Furthermore, $z := \lambda h$ is introduced. The obtained polynomial equation admits k solutions in the complex plane, $(\zeta_\ell)^n\,(\ell = 1, \ldots, k)$:

$$(1 - b_0 z)\,\zeta_\ell^k - \sum_{j=1}^{k} (a_j + b_j z)\,\zeta_\ell^{k-j} = 0, \quad \zeta_\ell = |\zeta_\ell|\,e^{i\phi_\ell}, \tag{20}$$

or,

$$\rho\left(\zeta_\ell\right) - z\sigma\left(\zeta_\ell\right) = 0, \quad 1 \leq \ell \leq k. \tag{21}$$

The polynomials $\rho(\zeta) := \zeta^k - \sum_{j=1}^{k} a_j \zeta^{k-j}$ and $\sigma(\zeta) := \sum_{j=0}^{k} b_j \zeta^{k-j}$ are referred to as the characteristic (or generating) polynomials of the multi-step method.

It is evident that for eigenvalues λ in the left half of the complex plane $[\Re(\lambda) < 0]$, the numerical solution of the test problem (16) using the linear multi-step method with time step h must eventually tend to zero. This requires that the k solutions $\zeta_l, 1 \leq l \leq k$ to the characteristic equation (21) satisfy the condition $|\zeta_l| < 1$. The roots obviously depend on z, so the job is to find the values of z that lead to the condition on the roots to be satisfied. Consequently, the stability region is re-defined as:

"*The (open) set of (complex) values of z such that the elements on the domain boundary correspond to $|\zeta_l| < 1$.*"

In practice, to find the boundary, we let $\zeta = e^{i\phi}$ and solve for z.

The stability regions of the *ABk* and *BDFk* schemes are derived using the corresponding form of the characteristic equation. More details on this are available in [3]. These regions, plotted in the real ($\Re$) and imaginary ($\Im$) axes, for the factor $z = \lambda h$ are depicted in Fig. 2.

The stability region for the explicit *ABk* schemes (see Fig. 2a) are located inside the closed curves. The consistency order of the *ABk* schemes is improved on increasing k, however this requires a refinement in the time step h to accommodate the eigenvalues of J in $\Re(\lambda) < 0$.

Recall that the eigenvalues of the convective operator are located on the imaginary axis. Thus, among the schemes depicted in Fig. 2a, AB3, which has the widest region of stability in the purely imaginary $\Im$-axis, (the intersection of the stability curve and the $\Im$-axis is located at $y = \pm 0.72362$), can be considered as most suitable for tackling fluid mechanical problems [3].

The stability region for the fully implicit *BDFk* schemes (i.e. Fig. 2b) are located outside the closed curves. The *BDF1* and *BDF2* schemes have a stability region that includes the entire negative half-plane $\Re(\lambda h) < 0$. Thus, in principle, when using these two implicit methods there are no non-physical growths in the solutions for any chosen time step h. Such methods are called *A-stable* and this is a convenient property in practice. However, it is important to remember that stability alone does not guarantee accuracy, especially for stiff problems. It is noticeable with the higher order *BDFk* schemes ($k > 3$) that the stability curves steadily penetrate into $\Im(\lambda h) < 0$ with increasing values of k. Therefore, they are no longer unconditionally stable for hyperbolic fluid mechanical problems.

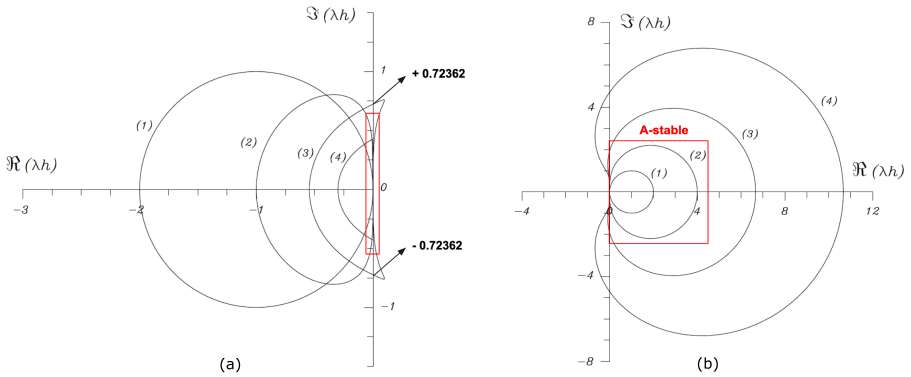


Figure 2: Stability regions for k$^{\text{th}}$ order schemes (with $1 \leq k \leq 4$) – a) explicit Adams-Bashforth (ABk) schemes (intersection with the imaginary axis depicted by the red box) and b) implicit backward differencing ($BDFk$) schemes. Note that the stability region for the ABk schemes are located within the closed curves while those for the $BDFk$ are located outside the region occupied by the closed curves. This figure is adapted from Deville et al. [3].

2.7 Stability and choice of time-step

Stability diagrams provide the span of admissible values of λh. These should be related to the choice of the time step. For fluid dynamical problems, it is commonly the imaginary eigenvalues of the convective operator that are difficult to accommodate and, therefore, drive the selection of the time-step.

Considering convection by a known velocity field $c(x,t)$, the convective operator is $c(x,t) \cdot du(x,t)/dx$. Using spectral elements (or any other spatial discretization approach), a discrete form of this operator can be derived, and, ultimately, it is the egienvalues of the discrete operator that dictate the time step, together with the choice of the time-integration scheme.

It can be shown that largest eigenvalue scales as $c/\Delta x$ [3]:

$$\max_k |\lambda_k h| = S \cdot \max_{c,\Delta x} \left| \frac{ch}{\Delta x} \right|, \tag{22}$$

where S is an order-unity coefficient. From the r.h.s., the Courant–Friedrichs–Lewy (CFL) number is defined:

$$\text{CFL} := \max_{c,\Delta x} \left| \frac{ch}{\Delta x} \right|. \tag{23}$$

The CFL number is easily computable and thus provides a very convenient estimate of z. The value of the coefficient S is larger than 1 and depends on the spatial discretization. In the case of the spectral element method, it also depends on the order of the interpolating polynomials. The value of S as a

function of the order N is plotted in Fig. 3, and the associated bounds are $1.16 \leq S \leq 1.5$.

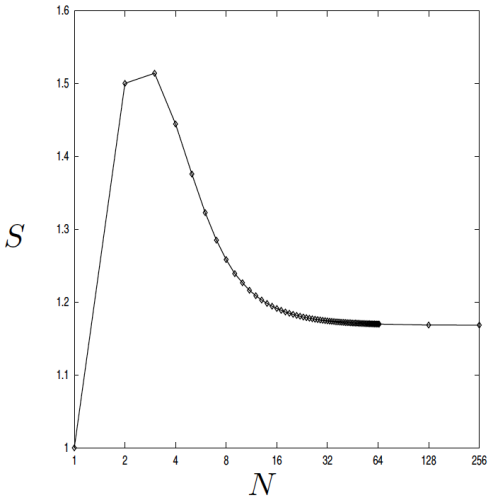


Figure 3: Scale factors S for spectral-element discretizations as a function of polynomial order N. This figure is adapted from Deville et al. [3].

The CFL number, S, and the stability diagram of the time-integration scheme readily provide the maximum time step that can be selected. For a time-varying $c(x,t)$, the time step may be changed accordingly, and this option is available to users of `Nek5000`.

As a concrete example, consider Fig. 2a, which shows that the stability diagram for AB3 cuts the imaginary axis at ± 0.72362, implying:

$$S \cdot \mathrm{CFL} \leq 0.72362. \tag{24}$$

The importance of satisfying this criterion is illustrated using Fig. 4 (adapted from [3]), which represents the convection of a Gaussian pulse ($u^0 := e^{-\zeta^2}, \zeta := 15(x - 0.5)$) at unit-speed. This problem is simulated on a periodic domain $[0, 1]$ using 256 linear finite elements. The solution is advanced using the *AB3* scheme at $CFL = 0.72$ and $CFL = 0.770$. The initial pulse (bottom) moves to the right, leaves the domain at $x = 1$, and reenters at $x = 0$ [3]. In the figure it is evident that the solution with $CFL = 0.720$ is stable.The pulse convects indefinitely (with error increasing linearly with t). However, for $CFL = 0.77$, instabilities appear at time $t \approx 1$, shortly after which the solution exhibits explosive growth (not shown). The unstable mode consists of $2\Delta x$ waves — the most oscillatory waves that can be represented on the grid, and those associated with the maximum (in modulus) eigenvalue [3]. Thus, the CFL criterion corresponding to the chosen discretization should be satisfied while choosing the respective time-steps.

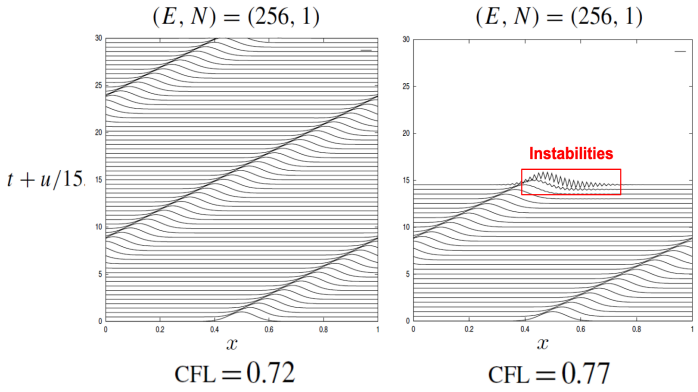


Figure 4: *AB3* solutions to $u_t + u_x = 0, u(0,t) = u(1,t)$ on a periodic domain $[0,1]$ using 256 linear finite elements. Note the build-up of instabilities in the solution when the CFL criteria given by eq. (24) is violated. This figure is adapted from Deville et al. [3].

3 Temporal discretization of the unsteady Navier-Stokes equations

The Navier-Stokes equations represent the conservation of mass and momentum of a fluid. These generic transport equations include both convective and diffusive modes of flow transport. Consequently, as stated in Section 2, the corresponding eigenvalues of the solution contain a negative real component (diffusive) and a purely imaginary component (convective). Furthermore, the diffusive term is linear and symmetric, while the convective term is nonlinear and non-symmetric.

In this section we build on the fundamentals presented above, and consider time integration practices implemented in `Nek5000` targeting the Navier-Stokes. This includes special treatment of the convective term (Sections 3.1, 3.2, and 3.3), variable time-stepping (Section 3.4), and how time integration fits into the overall discretization and solution procedure (Sections 3.5 and 3.6).

3.1 Extrapolation of the convective term (*EXTk*)

Recall that using implicit methods such as *BDFk* for non-linear equations leads to a non-linear system, unless some form of linearization is used. For this reason, explicit treatment of the convective terms is preferable. Another consideration leading to the same conclusion is the asymmetry of the convective term meaning that efficient linear solvers specialized for symmetric matrices cannot be used.

Karniadakis et al. [1] proposed using higher order extrapolation (a.k.a. *EXTk*) of the convective term. To that end, the Newton polynomial N_k is again put to use. In particular, the polynomial is built on k known values of the convective term

at previous time-steps and then evaluated at t^{n+1} to provide an extrapolated value. A k^{th}-order polynomial leads to scheme of order $k+1$.

Let ψ denote the approximated value of the convective term, for conciseness. Then using for formula for N_1 and N_2 (see eq. (58)), schemes *EXT2* and *EXT3* are derived:

$$
\begin{aligned}
N_1(t^{n+1}) &= \psi^{n-1} + \frac{\psi^n - \psi^{n-1}}{h}(t^{n+1} - t^{n-1}) = 2\psi^n - \psi^{n-1} \\
N_2(t^{n+1}) &= \psi^{n-2} + \frac{\psi^{n-1} - \psi^{n-2}}{h}(t^{n+1} - t^{n-2}) + \\
&+ \frac{\psi^n - 2\psi^{n-1} + \psi^{n-2}}{2h^2}(t^{n+1} - t^{n-1})(t^{n+1} - t^{n-2}) = \\
&= 3\psi^{n-1} - 2\psi^{n-2} + 3 \cdot (\psi^n - 2\psi^{n-1} + \psi^{n-2}) = 3\psi^n - 3\psi^{n-1} + \psi^{n-2}.
\end{aligned}
$$

For convenience and reference—skipping the intermediate steps in the above—*EXT2* and *EXT3* are respectively defined as:

$$\psi^{n+1} = 2\psi^n - \psi^{n-1} \tag{25}$$

$$\psi^{n+1} = 3\psi^n - 3\psi^{n-1} + \psi^{n-2}. \tag{26}$$

In addition, `Nek5000` employs a scheme referred to as *EXT2a*, which is derived by introducing a factor α in front of the quadratic term in N_2. This can be used to manipulate the scheme to have better stability, which is discussed below. For reference, the coefficients of the *EXT2a* scheme with $\alpha = 4/3$ are provided:

$$\psi_{n+1} = \frac{8}{3}\psi_n - \frac{7}{3}\psi_{n-1} + \frac{2}{3}\psi_{n-2}. \tag{27}$$

Note that in `Nek5000` *EXTk* schemes are not only used for the convection term, but also for the source terms.

3.2 Stability of the *BDFk/EXTk* method

In `Nek5000`, the momentum equation is discretized using a multistep method based on combination of *BDFk* treatment of the viscous term and temporal derivative, and the *EXTk* for the convective term and source terms. The stability of such methods must be studied per se, although, as usual, the convective term is the main contributor to the limitation of the time step [3].

Considering BDF2/EXT2, and applying it to $u' = \lambda u$ the following is obtained:

$$3u^{n+1} - 4u^n + u^{n-1} = 2\lambda h \left(2u^n - u^{n-1}\right).$$

We seek solutions of the form $u^m = (\zeta)^m, \zeta \in C$ and let $z = \lambda h$:

$$
\begin{gathered}
3\zeta^{n+1} - 4\zeta^n + \zeta^{n-1} = 2z\left(2\zeta^n - \zeta^{n-1}\right) \implies \\
3\zeta^2 - 4\zeta + 1 = 2z(2\zeta - 1).
\end{gathered}
$$

To find the region of stability, we set $\zeta = e^{i\theta}, \theta \in [0, 2\pi]$ (i.e. the unit circle), and solve for z :

$$z = \frac{3e^{i2\theta} - 4e^{i\theta} + 1}{2\left(2e^{i\theta} - 1\right)}.$$

For $BDF3/EXT3$ and $BDF2/EXT2a$ the procedure is, of course, similar.

The obtained stability regions are shown in Fig. 2. The regions for ABk schemes are also shown, for comparison. It is clear that the $BDF2/EXT2a$ (with $\alpha = 4/3$) combination offers stability matching that of *AB3*, whereas *BDF3/EXT3* are somewhat less stable.

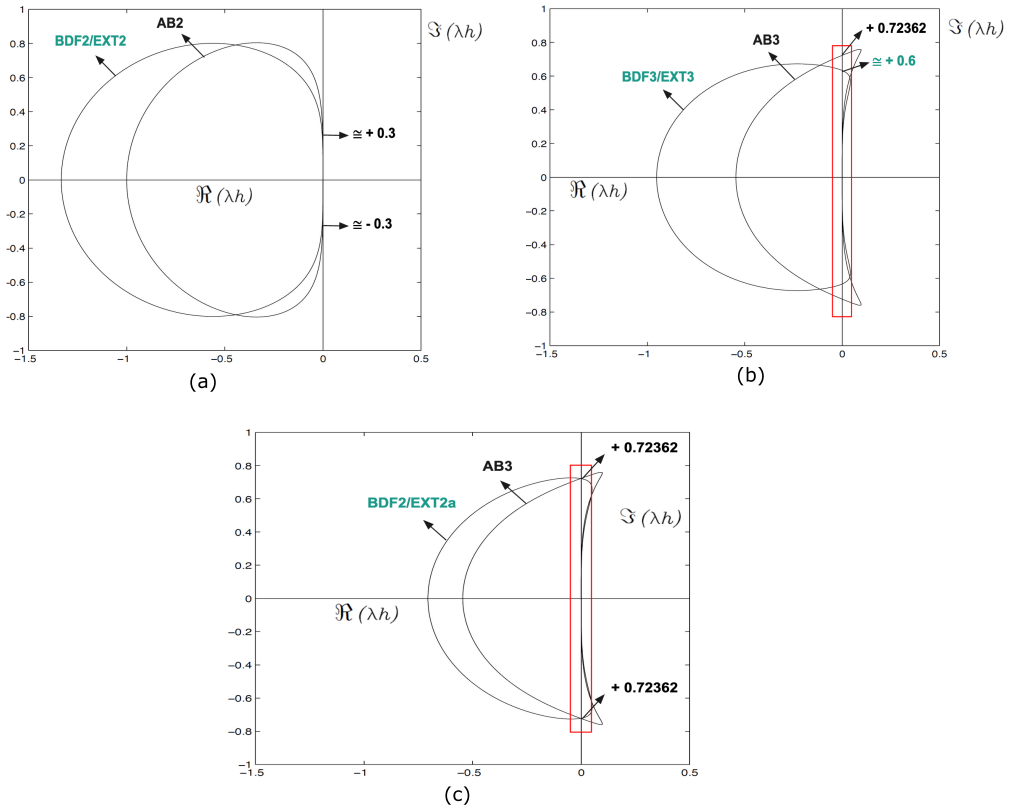


Figure 5: Stability regions for – (a) *AB2* and *BDF2/EXT2*, (b) *AB3* and *BDF3/EXT3*, and (c) *AB3* and *BDF2/EXT2a*. Note the increased region of the imaginary $\Im$-axis encompassed by the stability diagram for the *BDF2/EXT2a* scheme. This figure is adapted from [4].

3.3 The Operator-Integration Factor Scheme (OIFS)

The operator-integration factor scheme (OIFS) offers an alternative way of treating the time derivative and convective terms. Details may be found in this report by Fischer [4], and also in [3]. This strategy is briefly described below.

Considering the convection-diffusion equation:

$$\frac{\partial u}{\partial t} + \mathbf{c} \cdot \nabla u = \frac{1}{Pe} \nabla^2 u + f, \tag{28}$$

where c is a known convecting velocity field, f an external force, and Pe is the Peclet number (defined as the ratio of the rate of advection to the rate of diffusion), we combine the two terms on the left-hand-side into a material (Lagrangian) derivative:

$$\frac{Du}{Dt} = \frac{1}{Pe} \nabla^2 u + f. \tag{29}$$

The idea is then to apply *BDFk* directly to (29). For simplicity, we consider the case of *BDF2* with a constant time step:

$$\frac{3u^{n+1} - 4\hat{u}^n + \hat{u}^{n-1}}{2h} = \frac{1}{Pe} \nabla^2 u^{n+1} + f^{n+1}. \tag{30}$$

Since we are discretizing a Lagrangian derivative, $\hat{u}^{n+1-q}$ are lagged both in space and in time with respect to u^{n+1}. In particular, $\hat{u}^{n+1-q}$ pertains to the value of u at some $\mathbf{X}^{n+1-q}$ located on the characteristic passing through the node where u^{n+1} is sought:

$$\hat{u}^{n+1-q}(\mathbf{x}) := u\left(\mathbf{X}^{n+1-q}(\mathbf{x}), t^{n+1-q}\right).$$

In principle, it is possible to find $\mathbf{X}^{n+1-q}$ by marching backwards along the characteristic for a time qh. However, this straightforward approach would require a large amount of interpolation operations, rendering it too computationally expensive. OIFS offers an alternative to that, but instead introducing a PDE sub-problem of the following form:

$$\frac{\partial \hat{u}}{\partial s} + \mathbf{c} \cdot \nabla \hat{u} = 0, \qquad s \in \left[t^{n+1-q}, t^{n+1}\right] \tag{31}$$

$$\hat{u}\left(\mathbf{x}, t^{n+1-q}\right) = u\left(\mathbf{x}, t^{n+1-q}\right) \qquad \hat{u}(\mathbf{x}, t) = u(\mathbf{x}, t) \, \forall \mathbf{x} \in \partial\Omega_{\mathbf{c}},$$

Here, $\delta\Omega_c$ is defined as the part of the boundary having incoming velocity characteristics.

This models the convection of $\hat{u}$ in a sub-interval $[t^{n+1-q}, t^{n+1}]$. The trick is that as a result of the convection, the value of $\hat{u}^{n+1-q}$ will end up at the location of a computational node. This, because we know that the characteristic crosses it at time t^{n+1}. The necessity for interpolation and explicit search of $\mathbf{X}^{n+1-q}$ is now avoided.

For a *BDFk* discretization of (29), equation (3.3) will have to be solved k times, with the integration interval increasing with q. However, due to the linearity of

the equation, it is possible to rearrange (3.3) into a system, in which each of the k equations is only solved on a single $\left[t^{n+1-q}, t^{n+1-q+1}\right]$ subinterval. We refer the reader to [4] for details.

It remains to be discussed how this is solved numerically. The idea is to use a multi*stage* method, in particular, a fourth-order Runge-Kutta integrator (commonly referred to as *RK4*). These have not been discussed in this document, but we assume that the reader has some familiarity with *RK4* since it is the subject of basic numerical ODE integration theory. In practice, OIFS allows to violate the CFL condition and thus a larger time-step. However, for the convection problem (3.3) the CFL condition has to be respected. Whether OIFS overall offers improved performance compared to *BDFk/EXTk* is thus unclear, and can be said to be case-dependent.

3.4 Variable time stepping

So far, all the time marching schemes were introduced assuming a constant time step h. However, the consistent derivation of all schemes through associated interpolating polynomials N_k makes it straight-forward to derive corresponding schemes with a variable time step. This amounts to considering the general form of the Newton polynomials (57) instead of the equidistant version (58) used previously.

Let $\Delta t^n = t^{n+1} - t^n$. The variable time step *AB2*, *BDF2* and *EXT2* schemes are now derived. The third-order schemes are obtained in a similar fashion. For AB2, we consider the polynomial

$$N_1(t) = f^n + \frac{f^n - f^{n-1}}{\Delta t^n}\,(t - t^n)$$

and insert it into

$$\tilde{u}^{n+1} - \tilde{u}^n = \int_{t^n}^{t^{n+1}} N_1 dt.$$

Performing the integration leads to the following scheme:

$$\tilde{u}^{n+1} - \tilde{u}^n = \Delta t^n \left(1 + \frac{1}{2}\frac{\Delta t^n}{\Delta t^{n-1}}\right) f^n - \frac{1}{2}\frac{\Delta t^n}{\Delta t^{n-1}} \Delta t^n f^{n-1}. \tag{32}$$

For *EXT2* we make use of N_1 to construct an interpolant through ψ^n and ψ^{n-1} and evaluate it at t^{n+1}:

$$\psi^{n+1} = N_1(t^{n+1}) = \psi^{n-1} + \frac{\psi^n - \psi^{n-1}}{\Delta t^{n-1}}(t^{n+1} - t^{n-1}) = \tag{33}$$

$$= \left(1 + \frac{\Delta t^n}{\Delta t^{n-1}}\right)\psi^n - \frac{\Delta t^n}{\Delta t^{n-1}}\psi^{n-1}. \tag{34}$$

For *BDF2*, we should evaluate the derivative of N_3 at t^{n+1}. However, we will take an entirely different approach, which is more faithful to how the coefficients are computed in `Nek5000`. We closely follow the presentation by Nishikiwa in [9], where one also treats the *BDF3* case [9]. We can, in general, consider *BDF2* to be an approximator of the time-derivative based on a linear combination of the values of the unknown, u, at three points in time:

$$\left(\frac{du}{dt}\right)^{n+1}_{BDF2} \equiv \alpha_1 u^{n+1} + \alpha_0 u^n + \alpha_{-1} u^{n-1}.$$

Here, the coefficients α_i are considered unknown.

Next, we Taylor-expand the solution around the time-level $(n+1)$ to get:

$$\begin{aligned}
\left(\frac{du}{dt}\right)^{n+1}_{BDF2} &\equiv \alpha_1 u^{n+1} + \alpha_0 \left(u^{n+1} - \left.\frac{du}{dt}\right|^{n+1} \Delta t^n + \frac{1}{2}\left.\frac{d^2u}{dt^2}\right|^{n+1} (\Delta t^n)^2 \cdots \right) \\
&+ \alpha_{-1} \left(u^{n+1} - \left.\frac{du}{dt}\right|^{n+1} \left(\Delta t^n + \Delta t^{n-1}\right) + \frac{1}{2}\left.\frac{d^2u}{dt^2}\right|^{n+1} \left(\Delta t^n + \Delta t^{n-1}\right)^2 \cdots \right) \\
&= \left(\alpha_1 + \alpha_0 + \alpha_{-1}\right) u^{n+1} - \left[\alpha_0 \Delta t^n + \alpha_{-1}\left(\Delta t^n + \Delta t^{n-1}\right)\right] \left.\frac{du}{dt}\right|^{n+1} \\
&+ \frac{1}{2}\left[\alpha_0 \left(\Delta t^n\right)^2 + \alpha_{-1}\left(\Delta t^n + \Delta t^{n-1}\right)^2\right] \left.\frac{d^2u}{dt^2}\right|^{n+1} + O\left(\Delta t^3\right)
\end{aligned}$$

Our goal is to have a locally third-order accurate scheme, meaning

$$\left(\frac{du}{dt}\right)^{n+1}_{BDF2} = \left.\frac{du}{dt}\right|^{n+1} + O\left(\Delta t^3\right).$$

It is easy to see that this implies the following constraints on the unknown coefficients α_i:

$$\begin{aligned}
\alpha_1 + \alpha_0 + \alpha_{-1} &= 0 \\
\alpha_0 \Delta t^n + \alpha_{-1}\left(\Delta t^n + \Delta t^{n-1}\right) &= -1 \\
\alpha_0 \left(\Delta t^n\right)^2 + \alpha_{-1}\left(\Delta t^n + \Delta t^{n-1}\right)^2 &= 0.
\end{aligned} \tag{35}$$

For the case of *BDF2* it is easy to solve (35) analytically, see [9]. It is also straightforward to see that given $\Delta t^n = \Delta t^{n-1}$ one arrives to the standard *BDF2* coefficients for the constant time-step case.

For a general *BDFk*, the similarly assembled linear system can be solved using a direct solver. The latter is done in `Nek5000` (see subroutine `bdsys`), although

the structure of the system is slightly different. In particular, the matrix coefficients and the right-hand-side are circularly shifted to the right by one position. So, the last coefficient column becomes the right-hand-side, the right-hand-side becomes the first column in the coefficient matrix, etc. Due to this, some additional manipulations on the solution are necessary to compute α_i. Another implementational aspect is that the coefficients pertaining to previous time steps are stored with the opposite sign. This is explained by the fact that the corresponding terms are moved to the right-hand-side of the momentum equation, thus changing the sign.

3.5 Spectral element discretization of the unsteady convection-diffusion problem

In order to demonstrate the application of these aforementioned explicit and implicit temporal discretization strategies, the coefficients (or pre-factors) that have been derived in Sections 2.2, 2.3, 3.4 ,3.1 and 3.3 are directly incorporated in the spatially discretized equations. These equations are presented here in a compact form. A more detailed insight into the mathematical details of this discretization is available in the lecture notes by Fischer [6]. For simplicity, the discretized equations are derived at a second order.

Considering the re-written form of the convection-diffusion equation (eq. (28)) in 1D and its corresponding initial value form, we have:

$$\begin{aligned} &\frac{\partial \tilde{u}}{\partial t} + c \cdot \nabla \tilde{u} = \nu \nabla^2 \tilde{u} + f \\ &u(0,t) = u(1,t) = 0, \quad u(x,0) = u_0(x). \end{aligned} \tag{36}$$

Here, f is any external forcing function and ν is the kinematic viscosity of the fluid. Rearranging and evaluating each term in eq. (36) at t^{n+1}:

$$\left.\frac{\partial \tilde{u}}{\partial t}\right|_{t^{n+1}} - \nu \nabla^2 \tilde{u}|_{t^{n+1}} = (f - c \cdot \nabla \tilde{u})_{t^{n+1}} \,. \tag{37}$$

Using the implicit *BDF2* scheme for the viscous contributions ($\nu \cdot \nabla^2 \tilde{u}$) and the explicit *EXT2* scheme for the rest of the terms we have:

$$\begin{aligned} &\left.\frac{\partial \tilde{u}}{\partial t}\right|_{t^{n+1}} \approx \frac{3u^{n+1} - 4u^n + u^{n-1}}{2h} \\ &\text{and,} \\ &g^{n+1} \approx 2g^n - g^{n-1}, \quad g^k := (f - c \cdot \nabla \tilde{u})^k \end{aligned} \tag{38}$$

Now re-arranging, the following is obtained:

$$\frac{3u^{n+1} - 4u^n + u^{n-1}}{2h} - \nu\nabla^2\tilde{u}|_{t^{n+1}} = 2g^n - g^{n-1} \implies$$
$$\frac{3}{2}u^{n+1} - h\nu\nabla^2\tilde{u}|_{t^{n+1}} = 2hg^n - hg^{n-1} + 2u^n - \frac{1}{2}u^{n-1}$$

Discretizing eq. (38) with the weighted residuals (WR) method at N Gauss-Lobatto-Legendre (GLL) quadrature points [6]:

$$\frac{3}{2}\left(v, u^{n+1}\right)_N + (h\nu)\cdot a_N\left(v, u^{n+1}\right) = \left(v, q^{n+1}\right)_N, \tag{39}$$

where,

$$q^{n+1} = 2u^n - \frac{1}{2}u^{n-1} + 2hg^n - hg^{n-1}. \tag{40}$$

Here, v is the ansatz coefficient used in the WR method. Inserting the relevant basis functions and coefficient vectors in eq. (39):

$$\frac{3}{2}\underline{v}^T B\underline{u}^{n+1} + h\nu\underline{v}^T A\underline{u}^{n+1} = \underline{v}^T Q^T B_L \underline{q}_L^{n+1}, \quad \forall\underline{v}\in\mathbb{R}^n, \tag{41}$$

or, in a more compact manner:

$$H\underline{u}^{n+1} = Q^T B_L \underline{q}_L^{n+1}. \tag{42}$$

Here, $H = B + \nu\Delta t A$ is the discrete Helmholtz operator. Note that Q is a Boolean matrix that connects the global ($\underline{\bar{u}}$) and local ($\underline{u}_L$) node representations i.e. $\underline{u}_L = Q\underline{\bar{u}}$. Further, B is the mass matrix and A is the global stiffness matrix [6].

3.6 Extension towards the Navier-Stokes equations

The general procedure applied in the spectral element discretization of the unsteady convection-diffusion problem above can be extended towards the Navier-Stokes equations. Consider the incompressible Navier-Stokes equations in the space Ω:

$$\begin{aligned}\frac{\partial\mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} &= -\nabla p + \frac{1}{Re}\nabla^2\mathbf{u},\\ \nabla\cdot\mathbf{u} &= 0.\end{aligned} \tag{43}$$

Here, $\mathbf{u}$ is the fluid velocity vector, Re is the Reynolds number of the flow and p is the fluid pressure. As in Section 3.5, the *BDF2/EXT2* operator splitting is

used, where the non-linear terms are treated explicitly and the remaining linear Stokes problem is treated implicitly, giving:

$$\frac{3\mathbf{u}^{n+1} - 4\mathbf{u}^{n} + \mathbf{u}^{n-1}}{2h} = S\left(\mathbf{u}^{n+1}\right) + NL^{n+1}, \tag{44}$$

where, $S\left(\mathbf{u}^{n+1}\right)$ is a linear symmetric Stokes operator and the non-linear NL^{n+1} term is given by the extrapolant:

$$\sum_{j} \alpha_j \mathbf{u}^{n+1-j} \cdot \nabla \mathbf{u}^{n+1-j}. \tag{45}$$

For the *EXT2* scheme, $\alpha_1 = 2$ and $\alpha_2 = -1$ (eq. (25)), while for *EXT2a*, $\alpha_1 = 8/3$, $\alpha_2 = -7/3$ and $\alpha_3 = 2/3$ (eq. (27)). The Stokes system is solved at each time step as:

$$\mathscr{H}\mathbf{u}^{n+1} - \nabla p^{n+1} = \mathbf{f}^{n+1} \tag{46}$$

$$\nabla \cdot \mathbf{u}^{n+1} = 0, \tag{47}$$

where, the Helmholtz operator (analogous to H in eq. (42)) for $k = 2$ is given as:

$$\mathscr{H} := \left(\frac{3}{2h} - \frac{1}{Re}\nabla^2\right). \tag{48}$$

Insertion of the spectral element basis [6] into the Stokes system (i.e. eq. (46)), yields:

$$H\underline{\mathbf{u}}^{n+1} - D^T \underline{p}^{n+1} = B\underline{\mathbf{f}}^{n+1}, \quad D\underline{\mathbf{u}}^{n+1} = 0. \tag{49}$$

Here, the stiffness matrix, A, is defined in terms of the spectral differentiation matrix, D [6, 5]. Moreover, H, given as:

$$H = B\frac{3}{2h} - A\frac{1}{Re}, \tag{50}$$

is the discrete analogue of eq. (48). Solving Stokes equation system (49) using the P_N-P_{N-2} strategy [8] the pressure and velocity are decoupled as:

$$E\delta\underline{p} = -D\hat{\mathbf{u}}, \quad \underline{\mathbf{u}}^{n+1} = \hat{\underline{\mathbf{u}}} + hB^{-1}D^T\delta\underline{p}, \quad \underline{p}^{n+1} = \underline{p}^{n} + \delta\underline{p}, \tag{51}$$

which in the matrix form (after a single round of block Gaussian elimination) can be represented as [6, 5]:

$$\begin{bmatrix} \mathbf{H} & -\frac{2}{3}h\mathbf{H}B^{-1}\mathbf{D}^{\mathrm{T}} \\ 0 & E \end{bmatrix} \begin{pmatrix} \mathbf{u}^{n+1} \\ \underline{p}^{n+1} - \underline{p}^{n} \end{pmatrix} = \begin{pmatrix} \mathbf{Bf} + \mathbf{D}^{\mathrm{T}}\underline{p}^{n} \\ \underline{g} \end{pmatrix}. \tag{52}$$

Here, E, given as:

$$E = -\frac{2}{3}h\mathbf{D}\mathbf{B}^{-1}\mathbf{D}^{\mathrm{T}}, \tag{53}$$

is the Stokes-Schur complement governing the pressure in the absence of the viscous term. This is because, in the P_N-P_{N-2} method, the pressure is only defined on Gauss-Lobatto (GL) points in contrast to the velocity fields which are defined on Gauss-Lobatto-Legendre (GLL) points so as to remove spurious pressure modes. The advantage of this splitting procedure is that matrix vector products involving E can be computed without system solves, since B is diagonal. Note that the Galerkin approach implies that the governing system is symmetric and that the matrices H, A, and B are all symmetric positive definite. This equation system can be solved with a preconditioned conjugate gradient (PCG) or generalized minimal residual (GMRES) method.

Thus, in summary, the time advancement of the Navier-Stokes equations involves advancing the convective terms through the solution of eq. (46), solving for the viscous contribution in the construction of g in eq. (52), solving for the pressure using eqs. (52) and (53) and finally computing the divergence free solution $\mathbf{u}^n$ using eq. (52). H and E are solved in an iterative manner. Usually, when h/Re is small, H is diagonally dominant and readily amenable to solution via Jacobi preconditioned conjugate gradients [5].

4 Implementation pointers

Here we provide some pointers to the code implementing time integration in `Nek5000`. A call graph of relevant functions can be found in Figure 6, and a small description of the purpose of these functions is found in Table 1.

One can generally distinguish two sets of relevant functions. One is responsible for assembling the right-hand side, and is therefore called by `makef`. The others are called by `settime` and are responsible for auxiliary tasks such as determining a variable time-step, using input parameters from the user, computing the BDFk coefficients, etc. The majority of the code is relatively straightforward, except the code for OIFS, which is difficult to understand.

Useful prior knowledge is that the `bd` and `ab` are used to store the coefficients for the *BDFk* and *ABk/EXTk* schemes, respectively. The values of the coefficients for the case of a constant time step are given in Table 4. Note that the *ABk* schemes are only used in conjunction with *BDF1*, which would typically only be used to start up *BDF2/3*. However, it is possible to choose *BDF1* explicitly in the `.par` configuration file. Fore relevant keywords, please refer to the online documentation currently available at `https://nek5000.github.io/NekDoc/`.

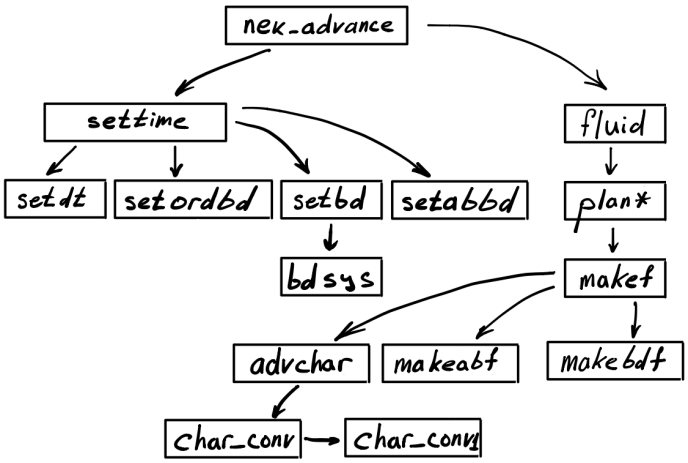


Figure 6: Call graph for functions handling time-stepping.

Function	Location	Purpose
`settime`	`drive2.f`	Calls several routines below to get the time-step and scheme coefficients.
`setdt`	`subs1.f`	Computes the next time-step.
`setordbd`	`navier1.f`	Sets the order of the BDF scheme for the current time-step.
`setbd`	`navier1.f`	Computes the coefficients for the BDFk scheme.
`bdsys`	`navier1.f`	Sets up the linear system to compute the BDFk coefficients.
`setabbd`	`navier1.f`	Computes the coefficients for the EXTk scheme.
`makeabf`	`navier1.f`	Adds convective term contribution to the right-hand-side.
`makebdf`	`navier1.f`	Adds $\partial/\partial t$ term contribution to the right-hand-side.
`advchar`	`navier1.f`	Contribution from characteristics-based time-stepping. Wraps `char_conv`.
`char_conv`	`convect.f`	Basically just wraps `char_conv1`.
`char_conv1`	`convect.f`	Computes contribution from the characteristics-based time-stepping.

Table 1: Key functions governing the time-stepping functionality.

5 Appendix. Newton's polynomial interpolation

The Newton polynomial is an interpolating polynomial passing through a given set of $n+1$ points (x_i, y_i), and build on the following basis functions:

$$n_0(x) = 1, \quad n_i(x) = \prod_{j=0}^{i-1} (x - x_j), \quad (i = 1, \cdots, n), \tag{54}$$

The form of the polynomial is thus

`Torder`	*ABk*	*BDFk*	*EXTk*
1	[1 0 0]	[1 1 0 0]	[1 0 0]
2	[3/2 -1/2 0]	[3/2 2 -1/2 0]	[2 -1 0]
2a	-	-	[8/3 -7/3 2/3]
3	[23/12 -4/3 5/12]	[11/6 3 -3/2 1/3]	[3 -3 1]

Table 2: Coefficients for schemes schemes used in `Nek5000` under the assumption of a constant time step.

$$
\begin{aligned}
N_n(x) = \sum_{i=0}^{n} c_i n_i(x) &= c_0 + \sum_{i=1}^{n} c_i \left(\prod_{j=0}^{i-1} (x - x_j) \right) \\
&= c_0 + c_1 (x - x_0) + c_2 (x - x_0)(x - x_1) + \cdots + c_n \prod_{j=0}^{n-1} (x - x_j) .
\end{aligned}
\tag{55}
$$

For this n$^{\text{th}}$ degree polynomial to pass all $n+1$ points, it needs to satisfy $n+1$ equations, which can be shown to lead to the following expression for the coefficients.

$$
c_k = y[x_0, \cdots, x_k] = \sum_{j=0}^{k} \frac{y(x_j)}{\prod_{i=0, i\neq j}^{k} (x_j - x_i)}, \quad (k = 0, \cdots, n). \tag{56}
$$

Here, $y[x_0, \cdots, x_k]$ are commonly referred to as divided differences.

It will be useful to have the forms of N_n written out explicitly for $n = 0$, 1, 2.

$$
\begin{aligned}
N_0 &= y_0 \\
N_1 &= y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) \\
N_2 &= y_0 + \frac{y_1 - y_0}{x_1 - x_0} + \frac{\frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0}(x - x_0)(x - x_1)
\end{aligned}
\tag{57}
$$

The particular case when x_i are equidistant with step h is also useful

$$
\begin{aligned}
N_0 &= y_0 \\
N_1 &= y_0 + \frac{y_1 - y_0}{h}(x - x_0) \\
N_2 &= y_0 + \frac{y_1 - y_0}{h}(x - x_0) + \frac{y_2 - 2y_1 + y_0}{2h^2}(x - x_0)(x - x_1)
\end{aligned}
\tag{58}
$$

Note how the c_1 and c_2 are clearly finite-difference approximations of the first and second derivatives, respectively.

References

[1] High-order splitting methods for the incompressible navier-stokes equations. *Journal of Computational Physics*, 97(2):414–443, 1991.

[2] R. Courant, K. Friedrichs, and H. Lewy. Über die partiellen Differenzengleichungen der mathematischen Physik. *Math. Ann.*, 100:32–74, 1928.

[3] M. O. Deville, P. F. Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge, 2002.

[4] Paul F. Fischer. Implementation considerations for the OIFS/characteristics approach to convection problems.

[5] Paul F Fischer. An overlapping schwarz method for spectral element solution of the incompressible navier–stokes equations. *Journal of Computational Physics*, 133(1):84–101, 1997.

[6] Paul F. Fischer. Lecture notes on spectral element method for flow simulation, February 2016.

[7] P. D. Lax and R. D. Richtmyer. Survey of the stability of linear finite difference equations. *Communications on Pure and Applied Mathematics*, 9(2):267–293, 1956.

[8] Yvon Maday and Anthony T. Patera. *Spectral element methods for the incompressible Navier-Stokes equations*. State of the Art Surveys in Computational Mechanics. A.K. Noor (Ed.), ASME, New York, 1989.

[9] H. Nishikawa. Derivation of BDF2/BDF3 for variable step size. *Technical note*, 2021.

Pressure preconditioning

Daniele Massaro[*], Donnatella Xavier[†] and Hamidreza Eivazi[‡]

July 7, 2021

1 Introduction

Nek5000 is an open source high-order solver for Computational Fluid Dynamics (CFD). It models incompressible and low Mach-number Navier-Stokes (NS) equations, but as regards this report the first formulation is considered.

The incompressible assumption for NS equations leads to a divergence free flow, where the pressure perturbations travel at infinite sound speed. Due to the elliptic nature of pressure equation, any modification in a given point of our domain gives an effect in each point of the entire domain instantaneously. The linear pressure sub-problem can become very ill-conditioned, making its solution the most expensive phase of the simulation using iterative solvers (around 80% of overall cost). It follows that a robust parallel preconditioning strategy is pivotal to guarantee good performances. *Nek5000* implements a spectral element method (SEM) discretization (Deville [1], Patera [10]), and in this context two possible approaches are presented. The first approach is based on additive overlapping Schwarz method [5, 3], while the second one uses a hybrid Schwarz-multigrid method [9, 8].

1.1 Problem formulation

As starting point let consider the incompressible NS equations for a Newtonian fluid in a non dimensional form. In order to be complete the set of equations 1 has to be combined with proper boundary and initial conditions.

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} &= -\nabla p + \frac{1}{Re}\nabla^2\mathbf{u} + \mathbf{f}, \\ \nabla\cdot\mathbf{u} &= 0 \end{aligned} \tag{1}$$

According to Galerkin approach the equations are multiplied by a test function and integrated over the domain. In this way the weak form can be derived. As local support basis function the Legendre polynomials are considered. In order to satisfy LBB stability condition $P_N - P_{N-2}$ is used: velocity and pressure spaces

[*]KTH Engineering Mechanics, dmassaro@kth.se
[†]KTH Engineering Mechanics, dgxavier@kth.se
[‡]University of Tehran, KTH Engineering Mechanics, hamidre@kth.se

are spanned by Lagrangian interpolants on Gauss-Lobatto-Legendre (GLL) and Gauss-Legendre (GL) respectively. The time integration is performed via semi-implicit scheme: non linear term explicitly and remaining unsteady Stokes problem implicitly. Eventually $\mathbf{f}^n$ incorporates all terms explicitly known at time t^n. The more in depth derivation can be found in [3].

$$\begin{aligned} \frac{1}{Re}(\nabla \mathbf{v}, \nabla \mathbf{u}^n)_{GLL} + \frac{1}{\Delta t}(\mathbf{v}, \mathbf{u}^n)_{GLL} - (\nabla \cdot \mathbf{v}, p^n)_{GL} &= (\mathbf{v}, \mathbf{f}^n)_{GLL}, \\ (q, \nabla \cdot \mathbf{u}^n)_{GL} &= 0 \end{aligned} \tag{2}$$

This system 2 can be rewritten in the matrix form:

$$\begin{pmatrix} \mathbf{H} & -\mathbf{D}^T \\ -\mathbf{D} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{u}^n \\ p^n \end{pmatrix} = \begin{pmatrix} \mathbf{f}^n \\ \mathbf{0} \end{pmatrix} \tag{3}$$

and according to generalized LU decomposition, which implements Uzawa decoupling approximating Q, it reads:

$$\begin{pmatrix} \mathbf{H} & -\frac{\delta t}{\beta_0}\mathbf{H}\mathbf{B}^{-1}\mathbf{D}^T \\ \mathbf{0} & E \end{pmatrix} \begin{pmatrix} \mathbf{u}^n \\ \delta p \end{pmatrix} = \begin{pmatrix} \mathbf{B}\mathbf{f}^n + \mathbf{D}^T p^{n-1} \\ g \end{pmatrix}$$

where $\mathbf{H}$ is the Discrete Helmholtz operator, $\mathbf{H} = \frac{1}{Re}\mathbf{A} + \frac{1}{\Delta t}\mathbf{B}$ where $\mathbf{A}$ is the discrete Laplacian and $\mathbf{B}$, the discrete mass matrix respectively, and $\mathbf{D}$ is the discrete divergence operator. The overall algorithm is made by 4 principal steps:

1. Calculate residual $\mathbf{f}^n$
2. Solve Helmholtz operator for intermediate velocity

$$\mathbf{H}\mathbf{u}^* = \mathbf{f}^n$$

3. Project on a divergence-free space solving

$$E\delta p = -\mathbf{D}\mathbf{u}^*$$

4. Update pressure and velocity

$$p^n = p^{n-1} + \delta p$$

$$\mathbf{u}^n = \mathbf{u}^* + \mathbf{D}^T \delta p$$

The report is focused on the solution of pressure equation (step 3), especially how to precondition properly the matrix E, which is usually very ill conditioned.

$$E = \frac{\delta t}{\beta_0}\mathbf{D}\mathbf{B}^{-1}\mathbf{D}^T \tag{4}$$

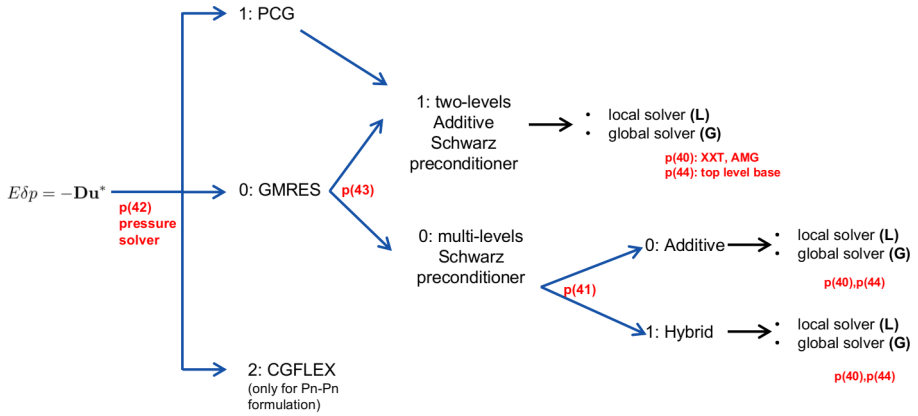


Figure 1: The flow diagram of the *Nek5000* implementation for the pressure equation solution is here presented.

1.2 *Nek5000* flow diagram

Figure 1 shows the overall implemented scheme and the available options to solve the pressure equation. First of all a pressure solver has to be defined. In this sense there are three possible choices.

The Preconditioned Conjugate Gradient (*PCG*) is a Krylov subspace iterative method which works only for symmetric positive definite operator (SPD). It uses only two levels Additive Schwarz preconditioner and is selected by setting (*param(42)=1*) in the .usr file (**userdat2**). Actually the runtime parameters, as the ones introduced below, are set by the user directly in the .par file. The flexible Conjugate Gradient (*CGFLEX*) is suitable for complex and very deformed mesh, but works only for $P_n - P_n$ formulation. Differently to PCG and GMRES, where a geometric multigrid approach is implemented, this is a fully algebraic multigrid method. In a nutshell: in the geometric approach the sequence of grids is defined apriori in a geometrically natural way, e.g. the coarse operator is defined only on the vertices of the initial mesh. The algebraic technique builds the coarse operator directly from the fine operator, hence the coarse grid matrix can be any linear combinations of upper level operator. CGFLEX is defined in the .par file ([**PRESSURE)**], **solver**, *param(42)=2*). In the interest of this discussion the Generalized minimal residual method (GMRES) is the only analysed. It's more flexible and the default choice for time dependent NS problem in $P_n - P_{n-2}$ formulation. It is defined in the .par file ([**PRESSURE)**], **solver**, *param(42)=0*).

GMRES is an iterative solver for linear system which looks for a solution in a Krylov subspace, minimizing the 2-norm of the residual. A two or multi levels preconditioners are available for this pressure solver. Before describing these possibilities, let step back and motivate the meaning of preconditioning. The preconditioner is an accelerating convergence technique extremely useful for linear system with badly conditioned operator. It consists in the system,

multiplication by a matrix M^{-1}, i.e. the preconditioner:

$$E\delta p = -\mathbf{Du}^* \rightarrow M^{-1}E\delta p = -M^{-1}\mathbf{Du}^* \tag{5}$$

From here on GMRES pressure solver is only considered and the Schwarz preconditioning is applied. The overlapping Schwarz method is described in details in §2, but here it is briefly introduced. Setting the *param(43)* two main options are available: two levels (fine and coarse) or multilevel (a V cycle with up to three additional levels). The multilevel approach (*param(43)*=0) is the default choice, but two-levels is also available for testing or in Adaptive Mesh Refinement (AMR). Both these algorithms have been discussed in detail in §2.1.4.

As equation 6 suggests The preconditioner is built by adding local to global contribute.

$$M^{-1} = R_0^T \hat{A}_0^{-1} R_0 + \sum_k R_k^T \hat{A}_k^{-1} R_k \tag{6}$$

In the multilevel case not only the additive technique is available. Indeed defining the *param(41)*, the hybrid option can be used as well. In the Hybrid Schwarz preconditioner a multiplicative approach is followed: local and global problems are coupled by multiplying those terms in the M^{-1} definition:

$$M^{-1} = R_0^T \hat{A}_0^{-1} R_0 \left[\sum_k R_k^T \hat{A}_k^{-1} R_k \right] \tag{7}$$

Since the hybrid approach is hard coded and used only by more expert users, the next section has elaborated only the additive Schwarz.

Eventually the local and global problems are solved. As regards the local problem, the *param(44)* defines the basis used to build up the local operator. Spectral Element Method (SEM) is the default one, but also a Finite Element (FEM) basis is available, even if only for axisymmetric flows. This last possibility is coupled with CGFLEX solver, good for very deformed mesh. In the following discussion only the SEM basis is taken into account.

Regarding the global problem solution, two coarse solvers are available, e.g. XXT and AMG. Both are described in the last section (see *param(40)*). The default choice is XXT, however the initialising pressure solver routine **prinit** (in **drive2.f**) sets a limit of the max element number to 350000.

2 Overlapping Additive Schwarz method

This section discusses the overlapping additive Schwarz preconditioner in some detail, with an overview of theory, followed by its implementation in Nek5000 solver.
The overlapping Schwarz method is an efficient domain decomposition method, developed on the principles of Schwarz's alternating numerical scheme of 1870. Domain decomposition is, as the name implies, the method of perceiving the

global domain as a union of simpler subdomains; formulating independent problems in these separate subdomains and then combining the solution through iterative or some other method. Domain decomposition methods are therefore, inherently parallel and have the advantages of both direct and iterative solvers. Preconditioners based on domain decomposition techniques are a natural choice for spectral element method discretization because the data in Nek5000, is structured within an element but is otherwise unstructured [11]. We shall first introduce the essential components that make up the Additive Schwarz method and then discuss the two-level and multi-level versions of it. Consider the discretized Poisson problem $A\mathbf{u} = \mathbf{f}$ where $A \in \mathbb{R}^{n\times n}$ and f and the solution $u \in \mathbb{R}^n$. Consider the domain partitioned into 2 overlapping subdomains, hence the set of indices $\{1, 2, ...n\}$ is partitioned into two sets as: $\mathcal{N}_1 \coloneqq \{u_1, u_2, \cdots u_s\}$ and $\mathcal{N}_2 \coloneqq \{u_{s-o}, \cdots u_n\}$ where o denotes the number of overlapping elements [1]. Accordingly two solutions exist $\mathbf{u}_1 \coloneqq (u_k)_{k\in\mathcal{N}_1} \coloneqq \mathbf{u}_{\mathcal{N}_1}$ and $\mathbf{u}_2 \coloneqq (u_k)_{k\in\mathcal{N}_1} \coloneqq \mathbf{u}_{\mathcal{N}_2}$. Two quantities that are necessary for this method, which form the link between the local subdomains and global subdomain. These are:

- A restriction operator. A matrix, denoted as R_i that restricts a vector $\mathbf{u} \in \mathbb{R}^{\mathcal{N}}$ to a subdomain Ω_i for $1 < i < N$, where R_i is rectangular $\mathcal{N}_i \times \mathcal{N}$ Boolean matrix. This restriction of the vector can be expressed by the product $R_i\mathbf{u}$
- The extension operator which is the transpose of R_i^T which performs the extension by 0 from $\mathbb{R}^{\mathcal{N}_i}$ to the global domain $\mathbb{R}^{\mathcal{N}}$

The solution scheme involves solving iteratively in the i subdomains, the system $\mathbf{u}^m \to \mathbf{u}^{m+1}$ from the residuals of the BVP:

$$\mathbf{u}_{\mathcal{N}_i}^{m+1} = \mathbf{u}_i^m + A_i^{-1}R_i(\mathbf{f} - A\mathbf{u}^m) \tag{8}$$

where $\mathbf{u}_i^m = R_i\mathbf{u}^m$ and $A_i = R_iAR_i^T$. The iterative Additive Schwarz preconditioner algorithm is the preconditioned fixed point iteration defined by [2]:

$$\mathbf{u}^{m+1} = u^m + M_{ASM}^{-1}\mathbf{r}^m \qquad \text{where} \quad \mathbf{r}^n \coloneqq \mathbf{f} - A\mathbf{u}^m \tag{9}$$

and the preconditioner matrix is a symmetric positive definite matrix given as :

$$M_{ASM}^{-1} = \sum_{i=1}^{n} R_i^T(R_iAR_i^T)R_i \tag{10}$$

The Additive Schwarz preconditioner discussed so far is still in "one" level. This one level additive Schwarz method *does not allow scalability.* The number of iterations needed for convergence increases linearly with the number of subdomains in one direction [2]. However the addition of a **coarse space correction** is the remedy for this. A "two"-level method allows to enrich the preconditioner by solving a *coarse* problem whose size is of the order of the number of subdomains. Elaborating further on the need for this "two-level" from a linear algebra point of view, using the one-level preconditioner M_{ASM}^{-1} would remove the influence of the very large eigenvalues of the coefficient matrix, which corresponds to the high frequency modes, but the low frequency modes in the spectrum of the preconditioned problem remain and lead to a stagnation in the convergence of

[1] eg. $n = 5, o = 2$ and with 2 subdomains, $\mathcal{N}_1 \coloneqq \{u_1, u_2, u_3, u_4\}$ and $\mathcal{N}_2 \coloneqq \{u_2, u_3, u_4, u_5\}$

the error. These low frequency eigenmode values represent certain global information and needs to be dealt with in an efficient manner. The introduction of a coarse grid or coarse space correction can be used to couple all the subdomains at each iteration of the iterative method.

Suppose we have some apriori knowledge of the small eigenvalues of the preconditioned system $M_{ASM}^{-1}A$ used to solve the linear system $A\mathbf{u} = \mathbf{f}$. For a Poisson problem, these slow modes correspond to constant functions that are in the null space (kernel) of the Laplace operators [2, 12]. Let Z be the rectangular matrix whose columns correspond to these slow mode eigenvectors. The best way to incorporate this information algebraically so as to accelerate convergence is by solving the minimization problem

$$min_\beta \|A(\mathbf{y} + Z\beta) - \mathbf{f}\|_{A^{-1}} \tag{11}$$

It corresponds to finding the best correction to an approximate solution y by a vector $Z\beta$ in the vector space spanned by the n_c columns of Z [2]. The solution to this $\beta = (Z^T AZ)^{-1}Z^T(\mathbf{f} - A\mathbf{y})$. Multiplying by Z gives the correction term as $Z\beta = Z(Z^T AZ)^{-1}Z^T(\mathbf{f} - A\mathbf{y})$. If the additive Schwarz method were to be used, the best correction that belongs to vector space spanned by the columns of Z would be $Z\beta = R_0^T(R_0AR_0^T)^{-1}R_0\mathbf{r}$ where $R_0 := Z^T$ and residual $\mathbf{r} = \mathbf{f} - A\mathbf{y}$. The **two-level** additive Schwarz preconditioner is thus defined as

$$M^{-1} = R_0^T(R_0AR_0^T)^{-1}R_0 + \sum_{i=1}^{n} R_i^T(R_iAR_i^T)^{-1}R_i \tag{12}$$

where the R_i's are the restriction operators to the overlapping subdomains and $R_0 = Z^T$. The local stiffness matrix $\hat{A}_k = R_iAR_i^T$ is derived from a tensor product of 1D element basis, and the minimal overlapping occurring in nek is shown in figure 2. The original sub-domain Ω^k is extended by two GL points on both sides. For an internal element homogeneous Dirichlet BCs are set on the external one, thereby only one degree of freedom is added per side, ζ_0 and ζ_N respectively.

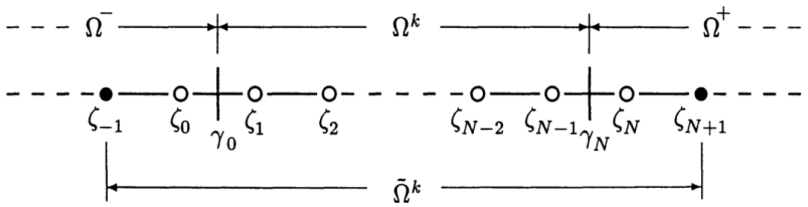


Figure 2: The minimal overlapping is shown for a 1D internal element. In the black points ζ_{-1} and ζ_{N+1} homogeneous Dirichlet boundary conditions are applied, see [5].

Summarising, the one level Schwarz method required solving only local subproblems in parallel, where as in the two level Schwarz method alongwith the local subproblems, an additional linear system with matrix $R_0AR_0^T$ needs to

be solved, which is *global* in nature. This global problem is called the coarse problem and it is global because it couples all the subdomains at each iteration. If the number of subdomains in not too large, the cost of the coarse solve is insignificant compared to the gain, because its matrix is a small square matrix $\mathcal{O}(n \times n)$. For coupling the local and global contributes, the **Hybrid Schwarz method multigrid preconditioner** [9] was developed. This is a multiplicative combination of the additive Schwarz smoother at the fine scale and a coarse grid correction. For 2 levels, the same Hybrid Schwarz multigrid looks as follows:

$$M^{-1} = R_0^T \hat{A}_0^{-1} R_0 \left[\sum_{k=1}^{n} R_k^T \hat{A}_k^{-1} R_k \right] \tag{13}$$

where the following two-level multigrid scheme would arise:

1. $\mathbf{u}^1 = \sum_k R_k^T \hat{A}_k^{-1} R_k \mathbf{f}$, where $\mathbf{f}$ is the righthand vector for $A\mathbf{u} = \mathbf{f}$
2. The residual is given as $\mathbf{r} = \mathbf{f} - A\mathbf{u}^1$.
3. $\mathbf{e} = R_0^T \hat{A}_0^{-1} R_0 \mathbf{r}$ is the coarse grid error.
4. $\mathbf{u} = \mathbf{u}^1 + \mathbf{e}$ is the solution

This method can be extended to general multilevel solver which performs the full V cycle, with smoothing and coarsening. In fact replacing the $\mathbf{r} = \mathbf{f}$, we obtain the additive Schwarz preconditioner, so multilevel scheme can be applied at 2 levels as well, with just one level of smoothing at local solve.

2.1 Implementation of Additive Schwarz Preconditioner in Nek5000

The parameter *param(43)* controls the selection of the preconditioner, whose default value is 0 representing *two-level additive Schwarz*. The additive Schwarz preconditioner in equation 12 can be rewritten as

$$M^{-1} = R_0^T \hat{A}_0^{-1} R_0 + \sum_{k=1}^{n} R_k^T \hat{A}_k^{-1} R_k \tag{14}$$

where $\hat{A}_k = R_k A R_k^T$, where A would be the spectral element discretization of the the PDE $A\mathbf{u} = \mathbf{f}$ on the finest mesh. $k = 0$ stands for the discretization of the PDE on the coarse mesh. Thus based on the discussion in previous section, *4 components* needed for the additive schwarz preconditioner are:

1. **Local Restriction R_k and prolongation R_k^T matrices** with Boolean entries returning only degrees of freedom inside each subdomain. An important fact to note here is that elements in Nek5000 SEM have no overlap, so R_k is actually an operator constructing elements with overlap. This is done using the gather scatter operator QQ^T in Nek5000, because Q is the binary operator duplicating basis coefficients in adjoining subdomains. Recall that $f_L = Qf$ is the global to local mapping of the function f and $Q^T f_L$ sums any multiple contributions to global degree of freedom from their local values. R_k and R_k^T

are built on top of the QQ^T routines

2. **Local stiffness matrices** $\hat{A}_k$. First recall $\hat{A}_k = R_k A R_k^T$, where A is the global stiffness matrix and the solution vector in each of the subdomains is related to the solution of global domain as $\mathbf{u} = R_k^T \mathbf{u}_k$ for k subdomains. $\hat{A}_k$ matrix is constructed in such a way that it can be inversed with *Fast Diagonalization Method (FDM)*, see [5], because it is needed to have the inverse of this matrix (see equation 14) Here in Nek5000, the local stiffness matrices are constructed by evaluating the tensor products on GLL quadrature points $A_{ij}^k = \int \frac{dh_i}{dx}\frac{dh_j}{dx}$. (figure 3). Note that this matrix is never really formed in Nek5000 but is written in terms of QQ^T. It is in fact the action of local 1D operators (small matrix-matrix multiplication) combined with QQ^T.

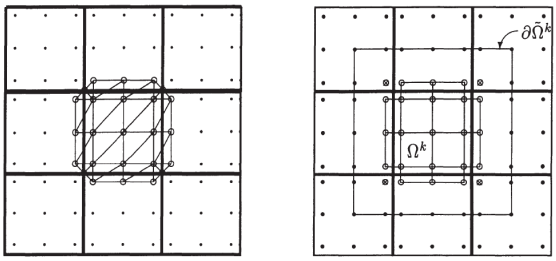


Figure 3: Degrees-of-freedom (open circles) for FEM based (left) and tensor-product based (right) discretizations of local problems. Values at nodes marked with $\otimes$ are marked zero by the restriction matrix R_k. Dirichlet boundary conditions are applied on $\partial\tilde{\Omega}^k$ [5].

In Nek5000, the deformation of elements is neglected and regularized surrogates of the elements are built by averaging the element faces and replacing them with parallelepipeds. For a regular parallelepiped the local stiffness matrix can be written in a separable form. For example for a two dimensional rectangle $Lx_k \times Ly_k$, the local stiffness matrix would be [9]:

$$A_k = \frac{Lx_k}{Ly_k}\mathbf{B}\otimes\mathbf{A} + \frac{Ly_k}{Lx_k}\mathbf{B}\otimes\mathbf{A} \qquad \text{and} \qquad \mathbf{A} := \mathbf{D}^T\mathbf{B}\mathbf{D} \tag{15}$$

where $\mathbf{B}$ is the one dimensional mass matrix composed of GL quadrature weights and $\mathbf{D} = \frac{dh_j}{dx}$ the derivative matrix. This form has a readily computable inverse given by fast diagonalization method :

$$A_k^{-1} = Z\otimes Z\left[\frac{Lx_k}{Ly_k}I\otimes\Lambda + \frac{Ly_k}{Lx_k}\Lambda\otimes I\right]^{-1}(Z^T\otimes Z^T) \tag{16}$$

where Z is the matrix of eigenvectors and Λ the matrix of eigen values satisfying $AZ = BZ\Lambda$ and $Z^TBZ = I$. The term in brackets in equation 16 is diagonal and its pseudo inverse in computed by inverting nonzero elements and retaining

zeros elsewhere. Thus in three dimensions, the inverse of A_k is given as

$$A_k^{-1} = (Z_3 \otimes Z_2 \otimes Z_1) D^{-1} (Z_3^T \otimes Z_2^T \otimes Z_1^T), \tag{17}$$

$$\text{with } D = I \otimes I \otimes \Lambda_1 + I \otimes \Lambda_2 \otimes I + \Lambda_3 \otimes I \otimes I \tag{18}$$

the subscript on the Z matrix indicates the associated coordinate direction in the reference element. In *nek* preconditioning any deformation is neglected and a regularised surrogates of the element is considered. Averaging dimensions are taken into account to recover a parallelepiped shape. This might give a quite bad approximation for very deformed elements. Since acting at preconditioning level and allowing great computational efficiency, thanks to fast diagonalization, this approximation is acceptable.

3. **Coarse grid stiffness matrix** $\hat{A}_0$: At the coarse level, the matrix is full and hence each coarse grid solve requires an all-to-all communication, as every entry of the distributed input has a nontrivial impact on every output value. In Nek5000, $\hat{A}_0$ is constructed by defining local SEM based Neumann operators that perform projection of local stiffness matrices A^k evaluated on the GLL quadrature points on the set of coarse base functions b_i representing linear finite element base on GLL grid. The coarse base functions are defined in $\hat{\Omega}$ as a tensor product of 1D functions $\hat{b}_{01}(r) = 0.5 * (r+1)$ and $\hat{b}_{10}(r) = 0.5 * (1-r)$for $r \in [-1, 1]$. , each of them corresponds to the single elements vertex for which the function's value is 1 [11]. The local contribution to $\hat{A}_0$ is given by $b_i^T A_k b_j$ and full $\hat{A}_0$ is finally assembled by local-to-global mapping summing contributions to global degree of freedom from their local counterparts. $\hat{A}_0$ is actually formed in Nek5000.

4. **Restriction matrix** R_0 and its transpose, the **prolongation operator** R_0^T for the coarse grid. R_0^T is the operator that interpolates coarse grid solution onto the tensor product array of GL points in the reference element. The prolongation operator is thus simply a bilinear or trilinear interpolant from the coarse grid vertices to the GL points in the reference element and hence can be cast as a sequence of efficient matrix-matrix products. $R_0 = Q_0^T Q_p^T I_p^T$ where I_p represents local interpolation from subdomain vertices to the Gauss points, Q_p^T represents the direct stiffness summation of vertex values within each processor and Q_0^T represents the interprocessor direct stiffness summation step [5].

2.1.1 Pressure preconditioner algorithm:

The pressure preconditioner routine is in uzawa_gmres. Either of the two preconditioning methods can be called here depending on *param*(43): two-level (uzprec ; navier1.f) additive or the multilevel (hsmg_solve; hsmg.f) Schwarz algorithm. The second method could be implemented as additive or hybrid. First, we discuss the two-level preconditioner algorithm as it is in Nek5000. uzprec (see Algorithm 1)is a general routine providing preconditioners for different type of operators implicit, steady and explicit. It starts with interpolating operator parameters $h1$ and $h2$ from velocity to pressure mesh. Next it selects proper operator action. The *unsteady explicit case* is relevant to us and here, the code

Algorithm 1 Pressure Preconditioner $M^{-1}E\delta p = -M^{-1}\mathbf{Du}^*$

```
1: procedure UZPREC(rpcg, rcg, h1m1, h2m1, intype, wp)
2:     CALL MAP12(...)                          ▷ interpolate h1, h2 on press. mesh
3:     if intype = 0 then
4:         steady case; not relevant
5:     else if intype = −1 then
6:         implicit case; not relevant
7:     else if intype = 1 then
8:         if ifnals then
9:             version not in use
10:        else
11:            CALL EPREC2(rpcg, rcg)
12:        end if
13:    else
14:        CALL COPY(rpcg, rcg)                 ▷ not in use
15:    end if
16:    CALL ORTHO(rcpg)                         ▷ remove mean; comm.
17: end procedure
```

executes uzprec2 (navier3.f), as $ifanls$ is by default set to $.false.$, and finishes with removing mean from the solution.

On the other hand, uzprec2 calls directly dd_solver (navier3.f), which performs local (local_solves_fdm ; fasts.f) and global (crs_solve_l2 ; navier8.f) solves in equation 14 and at the end sums their scaled results (see Algorithm 2).

Algorithm 2 Additive Schwarz two-level $M^{-1} = R_0^T \hat{A}_0^{-1} R_0 + \sum_{k=1}^{n} R_k^T \hat{A}_k^{-1} R_k$

```
1: procedure DD_SOLVER(u, v)
2:     CALL LOCAL_SOLVES_FDM(u, v)              ▷ local Schwarz operator
3:     CALL CRS_SOLVE_L2(uc, v)                 ▷ global operator
4:     alpha = 10.                              ▷ scaling factor
5:     CALL ADD2S2(u, uc, alpha, ntot)          ▷ local + global solution
6: end procedure
```

2.1.2 Local solver

Recall that the local stiffness matrix in equation 14 is assembled from tensor products and can be easily invertible. The fast diagonalization method used for the local solver, in Nek5000 is implemented in local_solves_fdm. It applies fast diagonalistaion method to the element interior appended by the face values of the face-neighbouring elements neglecting all edge or vertex neighbors. This operation assumes as well rectangular element neglecting element deformations. To append a given element with its neighbours face values the routine starts by copying pressure element $(N-1)^{dim}$ data into the middle of velocity sized element $(N+1)^{dim}$ leaving face, edge and vertex values blank. Next it extends interior values to the bigger element faces (dface_ext; fasts.f), sums

Algorithm 3 Local Solves $\hat{A}_k = R_k A R_k^T$ in $\sum_{k=1}^{n} R_k^T \hat{A}_k^{-1} R_k$

procedure LOCAL_SOLVES_FDM(u, v)
  CALL RZERO($v1, ...$) ▷ zero work array
  $v1(ix+1, iy+1, iz+iz1, e) = v(ix, iy, iz, e)$ ▷ fill element interior
  CALL DFACE_EXT($v1$) ▷ face ← interior sum
  CALL DSSUM($v1, ...$) ▷ face value exchange; **comm.**
  CALL DFACE_ADD1SI($v1, -1$) ▷ face ← interior subtract
  CALL FASTDM1($v1, ...$) ▷ fast diagonalisation
  CALL S_FACE_TO_INT($v1, -1$) ▷ interior ← face subtract
  CALL DSSUM($v1, ...$) ▷ face value exchange; **comm.**
  CALL S_FACE_TO_INT($v1, -1$) ▷ interior ← face sum
  CALL DO_WEIGHT_OP($v1$) ▷ non-symmetric weighting
  $u(ix, iy, iz, e) = v1(ix+1, iy+1, iz+iz1, e)$ ▷ extract data
end procedure

face values with neighbours (dssum; dssum.f) and finally subtracts own contribution (dface_add1si; fasts.f). In such prepared elements ($(N+1)^{dim}$ size) a solution is calculated with fast diagonalisation method (fastdm1; fasts.f) performing local matrix-matrix and vector-vector multiplications. This is followed by a redistribution of the solution, which is similar to appending step, however this time interior not face values are modified. For GMRES iterative solver ($param(42) = 0$) additional non-symmetric weights are applied by do_weight_op; (fasts.f; notice in this routine array index start from 0). The routine ends with extracting $(N-1)^{dim}$ date from N^{dim} sized elements.

Algorithm 4 Fast Diagonalization for the Local Solves $A_k^{-1} = (Z_3 \otimes Z_2 \otimes Z_1) D^{-1} (Z_3^T \otimes Z_2^T \otimes Z_1^T)$

procedure FASTDM1($r, df, sr, ss, st, w1, w2$)
  CALL TENSR3($...$) ▷ matrix-matrix
  CALL COL2($..$) ▷ vector-vector
  CALL TENSR3($...$) ▷ matrix-matrix
end procedure

2.1.3 Global solver

Finally the global (coarse grid) problem in equation 14 is solved in crs_solve_l2 (navier8.f) and consists of three steps: mapping element interior to the vertices (R_0), solving coarse grid problem on reduced number of degrees of freedom ($\hat{A}_0^{-1}$) and mapping solution to element interior (R_0^T)

Restriction and prolongation operations are performed by spectral interpolation between pressure mesh (GL points) and vertices of velocity mesh (GLL points) using maph1_to_l2t, maph1_to_l2 and specmpn from **navier8.f**. The coarse grid problem $\hat{A}_0^{-1}$ is not directly related to SEM formulation and will be discussed in a separate section.

Algorithm 5 Coarse Grid Solver $R_0^T \hat{A}_0^{-1} R_0$

procedure CRS_SOLVE_L2(uf, vf)
  CALL MAP_F_TO_C_L2_BILIN(uf, vf, w) ▷ R_0
  FGSLIB_CRS_SOLVE$(xxth(ifield), uc, uf)$ ▷ $\hat{A}_0^{-1}$
  CALL MAP_C_TO_F_L2_BILIN(uf, uc, w) ▷ R_o^T
end procedure

2.1.4 Multi-level preconditioner

Unlike two-level method this preconditioner is devoted to solving unsteady explicit problems only, but on the other hand it could be used in a form of hybrid Schwarz-multigird method as well [11]. Hybrid option is turned off by default, so we will not discuss it here. However, the implementation in Nek5000, involves an additive implementation and not a product as in equation 13 The main difference between these two additive methods are the restriction and prolongation operators for global problem in equation 14.

hsmg_solve starts with calculation of average density $rhoavg$, which is required

Algorithm 6 Hybrid Schwarz MultiGrid solver :
$M^{-1} = R_0^T \hat{A}_0^{-1} R_0 + \left[\sum_{k=1}^{n} R_k^T \hat{A}_k^{-1} R_k \right]$ at each refinement level

procedure HSMG_SOLVE(e, r)
  $rhoavg = glsc2(...)/volvm1$ ▷ relevant for hybrid only; **comm.**
  $l = mg\_lmax$ ▷ level number
  CALL LOCAL_SOLVES_FDM(e, r) ▷ local solution
  hybrid-multigrid section ▷ omitted
  $mg\_work2(i) = r(i)$ ▷ initialise residual
  for $l = mg\_lmax - 1, 2, -1$ **do** ▷ level loop
    CALL HSMG_RSTR(...) ▷ restriction; **comm.**
    CALL HSMG_SCHWARZ(...) ▷ Schwarz solve; **comm.**
    CALL HSMG_SCHWARZ_WT(...) ▷ weights multiplication
  end for
  CALL HSMG_RSTR_NO_DSDUM(...) ▷ restriction
  HSMG_DO_WT(...) ▷ weights multiplication
  HSMG_COARSE_SOLVE(...) ▷ coarse grid solve
  HSMG_DO_WT(...) ▷ weights multiplication
  for $l = 2, mg\_lmax - 1$ **do** ▷ level loop
  end for

  hybrid-multigrid section ▷ omitted
  $e(i) = e(i) + copt2 * mg\_work2(i)$ ▷ local + global solution
  CALL ORTHO(e) ▷ remove mean; **comm.**
end procedure

by hybrid solver only and could be omitted to avoid unnecessary communication. Next a local problem in equation 14 is solved in the similar way to two-level method by call to local_solves_fdm (fasts.f). The global step starts with copy of the residual r to a **multigrid work array** *mg_work* . At each level

three operations are performed: restriction (hsmg_rstr; hsmg.f), solution of local problem at given level (hsmg_schwarz;hsmg.f) and final weights multiplication (hsmg_schwarz_wt; hsmg.f).

The restriction operator performs following actions: multiplication of the element faces by weights (hsmg_do_wt; hsmg.f), interpolation to lower resolution grid (hsmg_tnsr; hsmg.f) using matrix-matrix multiplication and finally sum face values using communication (hsmg_dssum; hsmg.f). Last routine is similar to original dssum, but operates at given level. Solution of a local problem at given

Algorithm 7 Hybrid Schwarz MultiGrid Restriction operator

```
1: procedure HSMG_RSTR(uc, uf, l)
2:     CALL HSMG_DO_WT(...)        ▷ weights multiplication
3:     CALL HSMG_TNSR(...)         ▷ interpolation
4:     CALL HSMG_DSSUM(...)        ▷ face sum; comm.
5: end procedure
```

level done by hsmg_schwarzis analogical to action of local_solves_fdm, but contains additional steps. It starts with masking Dirichlet b.c. using weight multiplication routine hsmg_do_wt and next embeds the data in a bigger array with blank faces (hsmg_do_wt; hsmg.f). Next the data is appended with neighbour's interior values using hsmg_extrude and hsmg_schwarz_dssum. The difference with local_solves_fdm is the row/collumn number to be copied. Note that there are 2 communication routines hsmg_dssum and hsmg_schwarz_dssum. They are required as elements size differs by 2. Next local solve in element with fast diagonalistion method is performed with hsmg_fdm. This step is almost identical with it's two-level counterpart, however the result is stored in different array part. From this point we have to distinguish input and solution arrays. Following operations (8-11 in the routine scheme) give the same result as steps (8-10) in local_solves_fdm routine (although with more subroutine calls and additional data copy), but act on different columns/rows. The routine hsmg_schwarz ends with data extraction from a bigger element (hsmg_schwarz_toreg3d), face summation (hsmg_dssum), and masking Dirichlet b.c. (hsmg_do_wt).

Next operation in hsmg_solve is restriction to the coarsest level done by hsmg_rstr_no_dssum . It is identical with hsmg_rstr except the final exchange of face values (no call to hsmg_dssum). This restriction is followed by weights multiplication (hsmg_do_wt), coarse grid solve hsmg_coarse_solve and second weights multiplication (hsmg_do_wt).

3 Coarse problem

As it is mentioned in §1, the preconditioner combines local problems $R_k^T \hat{A}_k^{-1} R_k$ and a coarse grid problem $R_0^T \hat{A}_0^{-1} R_0$, see equation 6. In this section, we focus on the solution of the coarse grid operator, $\hat{A}_0$, which corresponds to a Laplace operator defined on the element vertices only. Because of its low number of degrees of freedom and global extent, the scalability of the coarse grid problem is mostly limited by communication and latency. We note that the term "coarse grid" here refers to the fact that $\hat{A}_0$ is defined on the vertices of the spectral elements only. Two choices are available in *Nek5000* to solve this problem and

Algorithm 8 Hybrid Schwarz MultiGrid solve procedure

```
1: procedure HSMG_SCHWARZ(e, r, l)
2:     HSMG_DO_WT(...)                                   ▷ masking
3:     CALL HSMG_SCHWARZ_TOEXT3D(..)                     ▷ extend array
4:     CALL HSMG_EXTRUDE(...)          ▷ face input ← interior input sum
5:     CALL HSMG_SCHWARZ_DSSUM(...)      ▷ face value exchange; comm.
6:     CALL HSMG_EXTRUDE(...)       ▷ face input ← interior input subtract
7:     CALL HSMG_FDM                                     ▷ local solve
8:     CALL HSMG_EXTRUDE(...)            ▷ face input ← face result sum
9:     CALL HSMG_SCHWARZ_DSSUM(...)      ▷ face value exchange; comm.
10:    CALL HSMG_EXTRUDE(...)         ▷ face result ← face input subtract
11:    CALL HSMG_EXTRUDE(...)         ▷ interior result ← face result sum
12:    CALL HSMG_SCHWARZ_TOREG3D(..)                     ▷ extract result array
13:    CALL HSMG_DSSUM(...)              ▷ face value exchange; comm.
14:    HSMG_DO_WT(...)                                   ▷ masking
15: end procedure
```

can be chosen using *param(40)*. The first one is a direct, sparse basis projection method, called XXT [13]. The second option uses an algebraic multigrid (AMG) method, which is more efficient for massively parallel (more than 10,000 cores) large simulations (more than 100,000 elements) [4, 6]. Table 1 presents a brief comparison between the XXT and the AMG coarse grid solvers. In the following, we discuss these two methods in more detail.

Table 1: Comparison of available coarse grid solvers in *NEK5000*

XXT	AMG
Direct	Iterative
Slow for large cases (max element number 350000)	Favored for large cases
Rapid setup	Complex setup

3.1 Coarse grid solvers

3.1.1 XXT

The XXT method is a fast direct solver for parallel solution of "coarse grid" problems, $A\underline{x} = \underline{b}$, such as arise when domain decomposition or multigrid methods are applied to elliptic partial differential equations in d space dimensions. The approach is based upon a Cholesky factorization of the inverse of A into XX^T with a convenient refactoring of the underlying matrix to maximize the sparsity pattern of X^T. If A is $n \times n$ and the number of processors is P, the algorithm requires $O(n^\gamma \log P)$ time for communication and $O(n^{1+\gamma}/P)$ time for computation, where $\gamma \equiv \frac{d-1}{d}$. The method is particularly suited to leading-edge multicomputer systems having thousands of processors. It achieves minimal message startup costs and substantially reduced message volume and arithmetic complexity compared to competing methods. The XXT method is based upon creating a sparse A-conjugate basis for $\mathbb{R}^n$, to be denoted by the columns of the

matrix

$$X_n = (\underline{x}_1, \underline{x}_2, \cdots, \underline{x}_n) \tag{19}$$

and it leads to a sparse (not necessarily triangular) factorization of the full matrix $XX^T \equiv A^{-1}$.

3.1.2 Sparse basis projection method in XXT

The projection approach in the XXT method incorporates a matrix of n basis vectors, $X \equiv X_n$, which is as sparse as possible and which yields significantly reduced computational and communication complexities. Here, we describe the implementation of the method and discuss communication considerations. We consider a $\sqrt{n} \times \sqrt{n}$ grid problem. Let the unit vectors $\hat{\underline{e}}_i$ and $\hat{\underline{e}}_j$ denote the ith and jth column of the $n \times n$ identity matrix. Let $\mathcal{N}_j$, the neighborhood of j, be the set of row indices corresponding to non-zeros in column j of A. Then

$$\hat{\underline{e}}_i^T A \hat{\underline{e}}_j = 0 \quad \forall i \notin \mathcal{N}_j \tag{20}$$

This situation is illustrated in figure 4a for the case where A arises from a 9-point discretization. From this figure it is clear that at least $n/\max|\mathcal{N}_j|$ of the unit vectors are A-conjugate to one another, where $|\mathcal{N}_j|$ denotes the cardinality of $\mathcal{N}_j$. The generation of a sparse basis for X starts with finding a maximal (or near-maximal) set of k_1 A-conjugate unit vectors. The first k_1 columns of X will have only one nonzero entry. Additional entries in X are generated via Gram-Schmidt orthogonalization. Let $X_k = (X_{k-1}, \underline{x}_k)$ denote the $n \times k$ matrix with columns $(\underline{x}_1, \underline{x}_2 \ldots, \underline{x}_k)$, and let $V = (\underline{v}_1, \underline{v}_2, \ldots, \underline{v}_n)$ be an appropriate column permutation of the identity matrix. Then the procedure

1: **for** $k = 1, \ldots, n$ **do**
2: $\quad \underline{w} := \underline{v}_k - X_{k-1} X_{k-1}^T A \underline{v}_k$
3: $\quad \underline{x}_k := \underline{w} / ||\underline{w}||_A$
4: $\quad X_k := (X_{k-1} \underline{x}_k)$
5: **end for**

ensures that $X = X_n$ is the desired factor of A^{-1}. An efficient procedure for selecting the permutation matrix, V, can be developed by defining separators which recursively divide the domain (or graph) associated with A into nearly equal subdomains. Figure 4b shows the first such separator for a $\sqrt{n} \times \sqrt{n}$ grid. Since the stencil for $A\hat{\underline{e}}_j$ does not cross the separator, it is clear that every unit vector $\hat{\underline{e}}_i$ associated with the left half of the domain in figure 4b is A-conjugate to every unit vector $\hat{\underline{e}}_j$ associated with the right half. If V is arranged such that vectors associated with the left half of the domain are ordered first, vectors associated with the right half second, and vectors associated with separator last, then application of Gram-Schmidt orthogonalization will generate a matrix X with worst-case fill depicted by figure 4c (X is shown here with the rows ordered according to the same permutation used for the columns of V). This procedure can be repeated to order the vectors within each subdomain, giving rise to the structure shown in figure 4d. To complete the construction we recur until no more separators can be found. The computational complexity of each solve is proportional to the amount of nonzero fill in the factor X. For the $\sqrt{n} \times \sqrt{n}$ grid it can be observed in figure 4d that the number of non-zeros in each row is

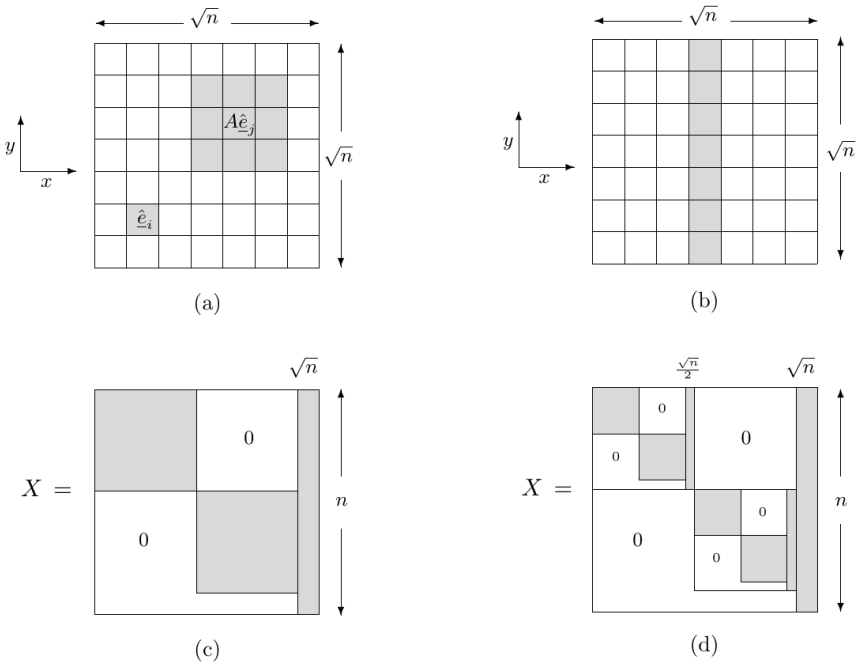


Figure 4: (a) geometric support (shaded) of orthogonal vectors $\underline{\hat{e}}_i$ and $A\underline{\hat{e}}_j$. (b) support of separator set. (c) zero/fill structure for X resulting from ordering the separator set last. (d) zero/fill structure after second round of recursion.

bounded by the sequence

$$\sqrt{n} + \frac{\sqrt{n}}{2} + \frac{\sqrt{n}}{2} + \frac{\sqrt{n}}{4} + \frac{\sqrt{n}}{4} + \cdots < 3\sqrt{n}$$

implying a total bound on the amount of fill in X of $3n\sqrt{n}$. Since we can evenly distribute the work among processors this leads to a computational complexity of $O(n^{\frac{3}{2}}/P)$. Similar arguments in three-dimensions lead to a computational complexity of $O(n^{\frac{5}{3}}/P)$. Both the two and three-dimensional cases provide a clear gain over the $O(n^2/P)$ cost incurred by the full inverse approach.

3.1.3 Algebraic multigrid (AMG)

Here we briefly outline the basic principles behind the Algebraic Multigrid (AMG) method. Let consider a system of linear algebraic equations in the form $Au = f$, where A is an $n \times n$ matrix. Multigrid methods are based on the recursive use of a two-grid scheme, which combines: (1) relaxation, or smoothing iteration, which is a simple iterative method such as Jacobi or Gauss-Seidel, and (2) coarse grid correction for solving the residual equation on a coarser grid. Transfer between grids is described with transfer operators P (prolongation or interpolation) and R (restriction). A setup phase of a generic algebraic multigrid (AMG) algorithm may be described as follows:

- Start with a system matrix $A_1 = A$.

- While the matrix A_i is too big to be solved directly:
  1. Introduce prolongation operator P_i, and restriction operator R_i.
  2. Construct coarse system using Galerkin operator: $A_{i+1} = R_i A_i P_i$.
- Construct a direct solver for the coarsest system A_L.

Note that in order to construct the next level in the AMG hierarchy, we only need to define transfer operators P and R. Also, the restriction operator is often chosen to be a transpose of the prolongation operator: $R = P^T$. Having constructed the AMG hierarchy, we can use it to solve the system as follows:

- Start at the finest level with initial approximation $u_1 = u_0$.
- Iterate until convergence (V-cycle):

  At each level of the grid hierarchy, finest-to-coarsest:

  1. Apply smoothing iterations (pre-relaxation) to the current solution $u_i = S_i(A_i, f_i, u_i)$.
  2. Find residual $e_i = f_i - A_i u_i$ and restrict it to the RHS on the coarser level: $f_{i+1} = R_i e_i$.
  3. Solve the coarsest system directly: $u_L = A_L^{-1} f_L$.

  At each level of the grid hierarchy, coarsest-to-finest:

  1. Update the current solution with the interpolated solution from the coarser level: $u_i = u_i + P_i u_{i+1}$.
  2. Apply smoothing iterations (post-relaxation) to the updated solution: $u_i = S_i(A_i, f_i, u_i)$.

So, in order to fully define an AMG method, we need to choose transfer operators P and R, and smoother S.

In the particular case of *Nek5000*, the AMG solver performs a single V-cycle and a fixed number of Chebyshev iterations, computed during the setup, is applied during the smoothing part. This method has the big advantage to avoid the inner product, thus reducing communication, at the expense of requiring some knowledge about the eigenvalues of the operator. More information about the theoretical background for the setup can be found in [7].

References

[1] M. O. Deville, P. F. Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge, 2002.

[2] V. Dolean, Pierre Jolivet, and F. Nataf. An introduction to domain decomposition methods - algorithms, theory, and parallel implementation. 2015.

[3] P. Fischer. An overlapping schwarz method for spectral element solution of the incompressible navier stokes equations. *Journal of Computational Physics, 133:84–101*, 1997.

[4] P Fischer, J Lottes, D Pointer, and A Siegel. Petascale algorithms for reactor hydrodynamics. *Journal of Physics: Conference Series*, 125:012076, jul 2008.

[5] P. Fischer, N. Miller, , and H. Tufo. An overlapping schwarz method for spectral element simulation of three-dimensional incompressible flow. *In P. Bjorstad and M. Luskin, editors, Parallel Solution of Partial Differential Equations, volume 120 of The IMA Volumes in Mathematics and its Applications, pages 159–180*, 2000.

[6] J Lottes. Independent quality measures for symmetric algebraic multigrid components. *Argonne National Laboratory, Mathematics & Computer Science Division*, 2005.

[7] James Lottes. *Towards robust algebraic multigrid methods for nonsymmetric problems*. Springer Theses, 2017.

[8] P. Fischer Lottes J. W. Hybrid multigrid/schwarz algorithms for the spectral element method. *Journal of Scientific Computing, 24(1):45–78*, 2005.

[9] Lottes J. W. P. Fischer. Hybrid schwarz-multigrid methods for the spectral element method: Extensions to navier-stokes. *Springer Berlin Heidelberg, Berlin, Heidelberg*, 2005.

[10] A. T. Patera. A spectral element method for fluid dynamics: Laminar flow in a channel expansion. *J. Comput. Phys., 54(3):468 – 488*, 1984.

[11] A. Peplinski, N. Offermans, P. F. Fischer, and P. Schlatter. Non-conforming elements in nek5000: Pressure preconditioning and parallel performance. In *Spectral and High Order Methods for Partial Differential Equations ICOSAHOM 2018*, pages 599–609. Springer International Publishing, 2020.

[12] JM Tang, R Nabben, C Vuik, and YA Erlangga. Comparison of two-level preconditioners derived from deflation, domain decomposition and multigrid methods. *Journal of Scientific Computing*, 39(3):340–370, 2009.

[13] H.M Tufo and P.F Fischer. Fast parallel direct solvers for coarse grid problems. *Journal of Parallel and Distributed Computing*, 61(2):151–177, 2001.

Direct Stiffness Summation

Alexandros Katsinos*, Dimitrios Papageorgiou† and Georgios Tsekouras‡

July 6, 2021

1 Contents

2 Direct Stiffness Summation

Direct stiffness summation (DSS) is a noninvertible, local to local transformation that amounts to summing shared interface variables to enforce continuity by redistributing them to their original locations, leaving the interior nodes unchanged.

In Nek5000 each process owns nel elements N^D nodes which are locally numbered and stored in the local vectors u_L

DSS (Fig. 1) involves the conversion of the local to global numbering (gather), the performance of a summation operation to enforce continuity between the aligned elements and the backward global-to-local transformation (scatter). Hence, it is totally a local-to-local process (subroutine dssum).

2.1 Scatter

Scatter is a global to local operation performed after the summation of the neighboring elements has been performed ensuring the continuity:

$$\underline{u}_L = Q\underline{u}$$

are Q is the connectivity matrix,$\underline{u}$ the global vector and $\underline{u}_L$ the local vector.

*AUTh Engineering Laboratory of Applied Thermodynamics, katsinos@meng.auth.gr
†AUTh Engineering Laboratory of Applied Thermodynamics, dpapageor@meng.auth.gr
‡AUTh Engineering Laboratory of Applied Thermodynamics, gctsekour@meng.auth.gr

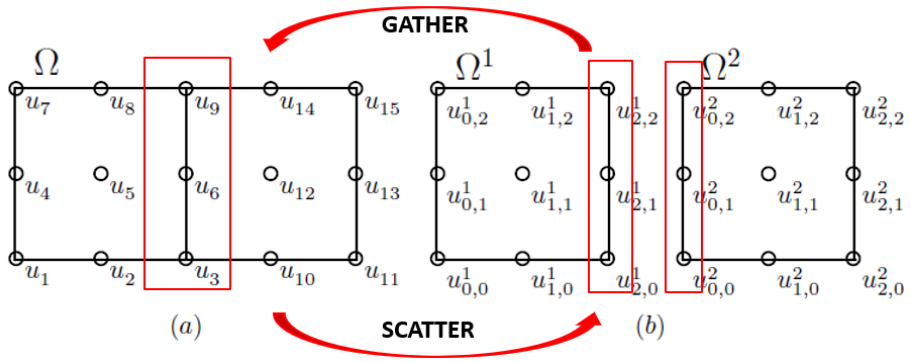


Figure 1: dss

In Fig. 1 two neighbouring elements with a shared interface are represented. The shared interface consists of three nodes, named as u_3,u_6,u_9 in the global system. These nodes belong to both elements, hence their multiplicity number is two. As we move towards each element's the local system, via the scatter operation (left to right operation in Fig.1), the nodes are renamed as $u^1_{2,0}$, $u^1_{2,1}$, $u^1_{2,2}$ in the left element respectively and $u^2_{0,0}$, $u^2_{0,1}$, $u^2_{0,2}$ in the right element (look in section for the naming convention), and are assigned with the global values.

In Fig. 2 the connectivity matrix of the aforementioned mesh is formed. As

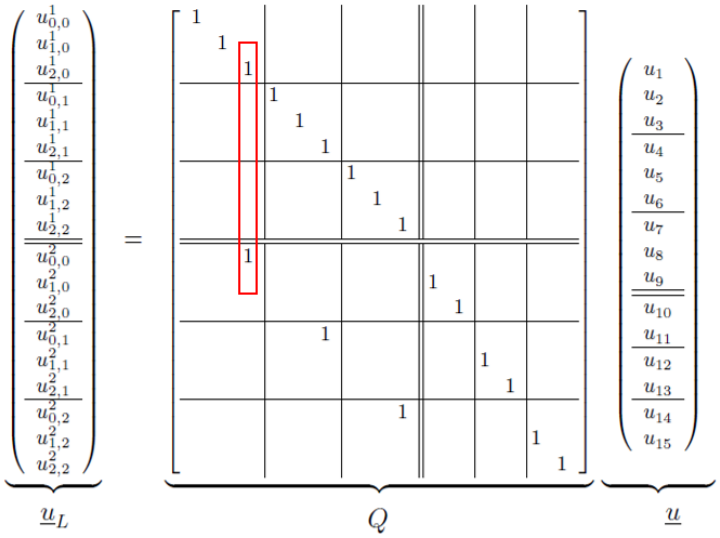


Figure 2: Connectivity matrix

it is expected u_3 equals both $u^1_{2,0}$ and $u^2_{0,0}$ as it is placed at the interface. Its multiplicity number can be estimated from the number of the non-zero elements in the respective column.

In practise the matrix Q is never constructed (matrix-free formulation). The actions of Q and Q^T are implemented via indirect addressing, as it is illustrated

at the below pseudocodes.

Algorithm 1 The scatter kernel $u_L = Qu$

```
1: procedure SCATTER(u_L)
2:     for e = 1, 2 ..., E do
3:         for i, j, k = 1, 2 ..., n do
4:             î = global_index(i, j, k, e)
5:             u^e_ijk = u_î
6:         end for
7:     end for
8: end procedure
```

2.2 Gather

Gather is the local to global operation

$$\underline{v} = Q^T \underline{u}_L$$

that sums the entries of the local nodes that refer to the same global node. Since each element is solved independently from its neighboring the results at the interfaces differ. With the gather operation a single value (the sum) is stored to the global element ensuring the continuity.

A correction of the shared nodal values is performed afterwards, taking into account the multiplicity number (this operation is not implemented in the dssum subroutine).

Algorithm 2 The gather kernel $u = Q^T u_L$

```
1: procedure SCATTER(u_L)
2:     for e = 1, 2 ..., E do
3:         for i, j, k = 1, 2 ..., n do
4:             î ← loctglb(i, j, k, e)
5:             u_î ← u_î + u^e_ijk
6:         end for
7:     end for
8: end procedure
```

3 Parallel Computing

Parallel computers provide a means for combining P individual processors to work in concert on a common task. There are two possibilities here:

1. Each processor executes the same instruction on different data, corresponding to the single instruction, multiple-data (SIMD) model

2. Each processor is allowed to operate asynchronously, following its own instruction set, corresponding to the multiple-instruction, multiple-data (MIMD) model

Another issue is whether the memory should be shared or distributed among the processors.

1. In the shared-memory model, each processor has access to all the data.
2. In the distributed-memory model, each processor has access to its own data. In this case data from other processors must be obtained by alternative means.

Nek5000 uses the single-program, multiple-data (SPMD) model in which each processor independently executes a copy of the same program and a loose synchronization among the processors is enforced emulating this way the SIMD programming model while preserving the flexibility of the MIMD approach. The SPMD model is predicated on each processor having a private (or local) address space, implying that a given variable or array entry can have different values on different processors. Data transfers between address spaces (processors) are made through subroutine calls which we will see in the next slide. One advantage of message passing is that synchronization is implicit in the message transfer; data are sent only when ready (we will see an example later on). This communication is performed using the Message Passing Interface (MPI).

The basic functions of the MPI implementation are:

1. **call mpi comm rank**(mpi comm world,MYID ,ierr): tells the calling processor its node identification (MYID := p).
2. **call mpi comm size**(mpi comm world,NPROCS,ierr): tells the number of processors participating in the current simulation (NPROCS := P).
3. **call mpi send**(X,n,mpi byte,id,mtag,mpi comm world,ierr): sends the contents of the vector X to processor id$\epsilon\{0, ..., P-1\}$, N is the amount of data in bytes, and the data are identified with an integer tag (mtag) that allows the receiving processor to distinguish this message from others that it might receive.
4. **call mpi recv**(Y,n,mpi byte,id,mtag,mpi comm world,stat,ierr): direct incoming data to a specific variable or array location: receives incoming data from node id having tag mtag and places it in Y. The mpi comm world argument is a handle that allows MPI to distinguish between calls made by the user's program and those made by library routines that might potentially send messages with the same integer tags. The ierr argument is an error condition that normally returns zero.

A common application of parallel processing in Nek5000 is the MPI implementation of a vector reduction process. In Eq. 1 the scalar reduction operation in a single processor is presented:

$$s = \sum_{i=1}^{n} \underline{x}_i \tag{1}$$

where $n = lx \cdot ly \cdot lz \cdot E$, where E is the total number of elements.

In parallel computation the vector reduction is implemented as:

$$s = \sum_{i=1}^{p} s^{(P)}, s^{(P)} = \sum_{i=1}^{p} s^{(P)}, \underline{x} = \bigcup_{p=0}^{p-1} \underline{x}^{(}P) \tag{2}$$

where $E^{(}P)$ is the number of elements in the P process.

As it is can be observed the total vector is split into (P) vectors, which are as many as the number of processes. In the SPMD model each vector comprises different contents than the other vectors. Afterwards, the reduction of each local vector takes place and finally the total output is formed from the (P) different values. The Algorithm 3 illustrates the MPI implementation of vector reduction, where the vector x is distributed across P processors. The mpi allreduce call performs the sum across all processors and returns the final result. At this point a loose implicit synchronization is performed as each processor is providing its result (s) to the allreduce function and waits for this process to be executed to every single processor, calculating the total output. After this point the calculated value is distributed synchronously in each processor so that the code execution can continue. Thus, all processors have the same global sum value at the end of the call. In the mpiallreduce function the third argument determines the length of the vector to be summed across processors. In this case, there is only one input (s) and output (glsum). The fourth argument tells MPI the size and type of data being transmitted, and the fifth determines which reduction operation to perform.

Algorithm 3 Vector reduction

```
1:  procedure GLSUM(s)
2:      real*8 function glsum(x,n)
3:      real*8 x(n),s
4:      include 'mpif.h'
5:      s=0
6:      for i = 1, n do
7:          s = s + x(i)
8:      end for
9:      call mpi allreduce(s,glsum,1, mpi double precision,
10:     & mpi sum,mpi comm world,ierr)
11: end procedure
```

4 Element alignment

When it comes to the challenge of element alignment, the information between the elements should be transferred correctly and this is ensured through a global numbering process of the nodes.

4.1 Methodology of global numbering

In this section a methodology, on how the global numbering is done in NEK, is established. The methodology is based on a 2d mesh that consists of 4 elements.

The steps that are followed, are described below:

1. Assign hypercube ordering of vertices. Numbering can start either from the bottom left corner or upper left corner, and just continuity on the numbering is needed. All vertices along the first row are scanned and the same procedure is continued until all rows are scanned. Here, I have to say that the global vertex number is an input to the whole procedure and it is generated during a pre-processing phase by genmap or genconn programs. V19 of Nek5000 does not provide global vertex during the initialisation (It obtains it from a file), but there are plans to add it in next release. In Figure 3, the result is presented.

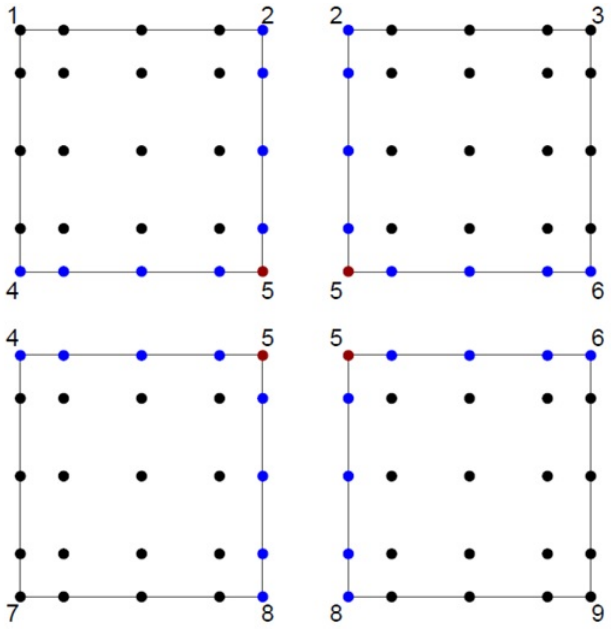


Figure 3: Assign hypercube ordering of vertices

2. Sort edges, by bounding vertices. Each edge is defined by two vertices. When sorting the edges, we go from lower vertex numbers to higher. That means that edge 1 will consist of vertex numbers 1 and 2. After that edge 2 will consist of vertices of 1 and 4, edge from vertices 2 and 3 and so on, until every single one is sorted.

3. Numbering every SEM node on each edge. After completing sorting, numbering every Spectral Element Method node on each edge is done. The nodes are numbered again in the direction from the lower to the higher vertex number. In Figure 4, the result is presented.

4. Numbering of the interior points (if needed). The final step will be the numbering of the interior points, which will only happen, if the flag ifcenter in the arguments of the setvert2D subroutine is turned to true. However, in most cases the interior points are not numbered, as information is only shared for the aligned nodes.

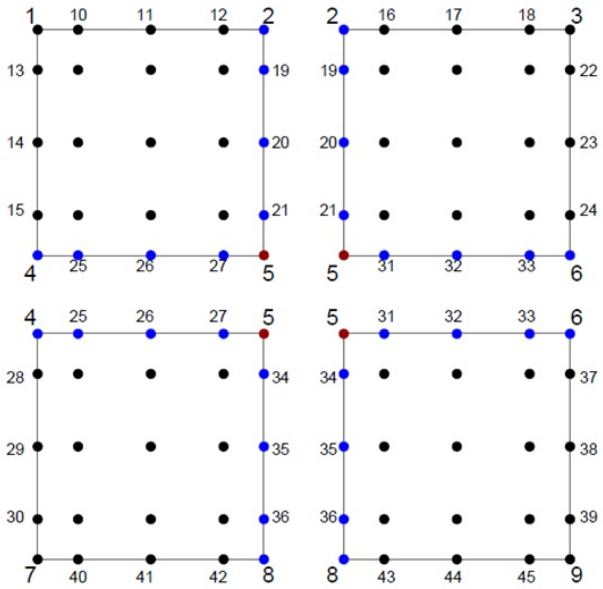


Figure 4: Numbering SEM nodes on each edge

5 Gather-Scatter Library

After completing element alignment, the gather scatter library is ready to be used. Its basic uses are:

1. Gather-Scatter operation
2. Either gather or scatter operation only (depending on mapping)
3. Computation of min/max value of nodes
4. Performing any kind of specific communication between nodes (proper mapping)
5. Count elements sharing each node
6. Matrix-vector products for row- and column-partitioned matrices. This point is related to a coarse grid solver for pressure preconditioning.
7. Generate consistent masks for external domain boundaries, that are used for external boundary conditions
8. Correction of internal faces in fix_geom, in order to reduce the complexity of the geometry

5.1 Initialization and Setup

In order to use gs library, initialization and setup of a handle, based on local-to-global mapping, are needed. This handle is a structure that contains the information about how do the communication. After its setup, this handle can be used for every gs operation on corresponding data. So the main idea needed

to be implemented is the creation of local-to-global mapping. The aim of the mapping is to allow gs_lib to create topology for sending data between processes and apply gs_op.

5.2 Basic Functions of gs_lib

In this section, some basic functions of the gs_lib are presented. Generally, these operations can be done both in C, but also in Fortran, where the calls are similar, but the library there has more limited options. Starting with the initialization of gs_lib, In C, the prototype to call the setup is the following:

struct *gs_data* ∗ *gs_setup* (const slong *id, uint n, const struct *comm* ∗ *comm*, int unique, gs_method method, int verbose)

In Fortran the format is much simpler and it follows the formula given below:

subroutine gs_setup (gs_handle, id, n, comm, mp)

After the setup is completed, the gather scatter operation is performed. Again here, there are two types of formats. In C, the corresponding function is given by the formula below:

void gs (void *u, gs_dom dom, gs_op op, unsigned transpose, struct gs_data *gsh, buffer *buf)

In Fortran, the formula is shown here:

subroutine gs_op (gs_handle, u, dom, op, transpose)

For more information, about the inputs and the outputs of the functions, as well as, an implementation example, see the study of Nicolas Offermans (Reference [3]).

6 Networks

An important part of communication are the networks, as data here are distributed based on hypercube topology. The basic numerical operation that takes place is the matrix-vector multiplication is defined by Equation 3:

$$yi = \sum_j T_{ij} x_j \tag{3}$$

,where T_{ij} is the connectivity matrix. As it is already mentioned, a very important aspect of direct stiffness summation is the reduction of communication cost. For this reason, the information along the shortest path is needed to be transferred in a consistent way. So based on the values of the elements of the connectivity matrix, the network can be either regular or irregular.

6.1 Regular Networks

When all the elements are non-zero, the network is called regular and that means it is fully interconnected. One of the most common algorithms that are used for a regular network is the index algorithm. The index algorithm orders the communication action according to the channel number c, having chosen the shortest paths for all packets. In this figure, we can see the time evolution of the cube memory during the index algorithm. We first advance all packets one step through channels c =0, then through channels c = 1 etc. In order to perform the summation, the components in each node have to be ordered to a horizontal vector. We see that finally the initial package is ordered vertically downward and as long as it goes in this direction, the packet stays in the current intermediate node.

6.2 Irregular Networks

However, in the most cases, the network is either characterized by sparse or medium range connectivity. It is convenient to consider sparse and intermediate connectivity as the special case of the full connectivity with some T_{ij} elements equal zero to address the problem of optimal summation in this case. In this case, instead of the index algorithm, the crystal router algorithm is formulated, which follows the index logics, but only the non-zero elements of the connectivity matrix are moving. Specifically, the "travel with ticket" analogy scheme is followed, which indicates that each packet is routed through the hypercube together with the destination node number. In Figure 5, a problem, that is related to general purpose all-to-all communication, is presented. It can be easily observed that the communication is bad, as the processors send messages to multiple ranks at the same time.

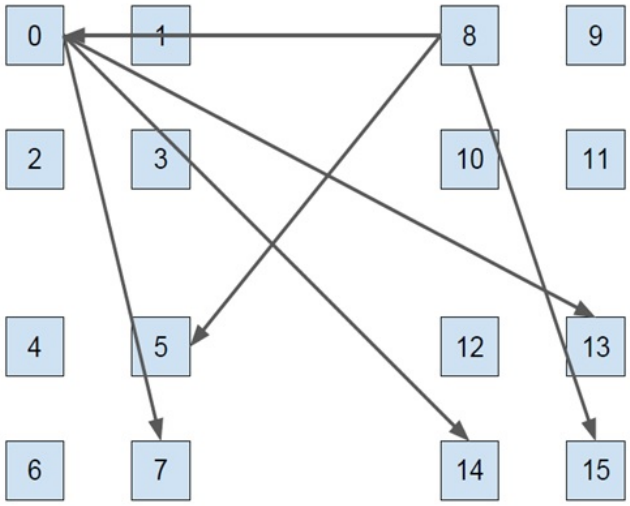


Figure 5: Communication Complexity

So this network is characterized by great complexity and communication cost, and we need to order the message in order to minimize the communication

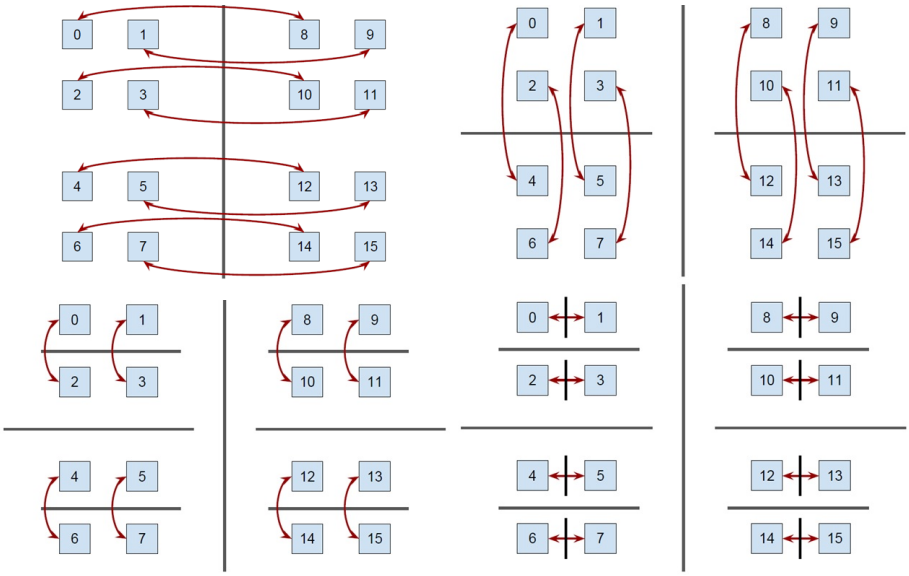


Figure 6: Crystal Router Flow Chart

cost. For this reason, we are introducing the methodology that the crystal router algorithm follows. In Figure 5, the flow chart chart of crystal router is described. Initially, a barrier in the middle is set and messages across this barrier are sent. After sending the messages, the partial summations are made. After completing the summations, a barrier at a different dimension is added and the same procedure is repeated.

7 Flow Chart

Direct stiffness summation takes place with the help of several subroutines inside the code. The general flow chart is presented in the Figure 7 below.

The routine responsible for the direct stiffness summation is `setup_topo` which is located in the `connect1.f` file. Its responsibilities include determining the connectivity of the element structure, verifying the right-handedness of the elements, defining the multiplicity of the elements and setting up the direct stiffness summation arrays.

For initiating the direct stiffness summation, firstly it is necessary to set up the element-processor mapping which is the domain partitioning and shortly after to establish the global numbering. As far as direct stiffness summation is concerned, the focus is on the establishment of the global numbering of the nodes, and particularly on the `setupds` routine which lies on the `dssum` file, which is where direct stiffness summation begins to take form.

In the `setupds` routine, the global numbering is achieved through the call of the `set_vert` routine. The `set_vert` routine is supplied by the vertices of the elements which are already globally numbered by the preprocessor and according to the dimension of the problem, begins to globally number the edges or the faces of the elements.

After the global numbering is complete, the program returns to the `setupds`

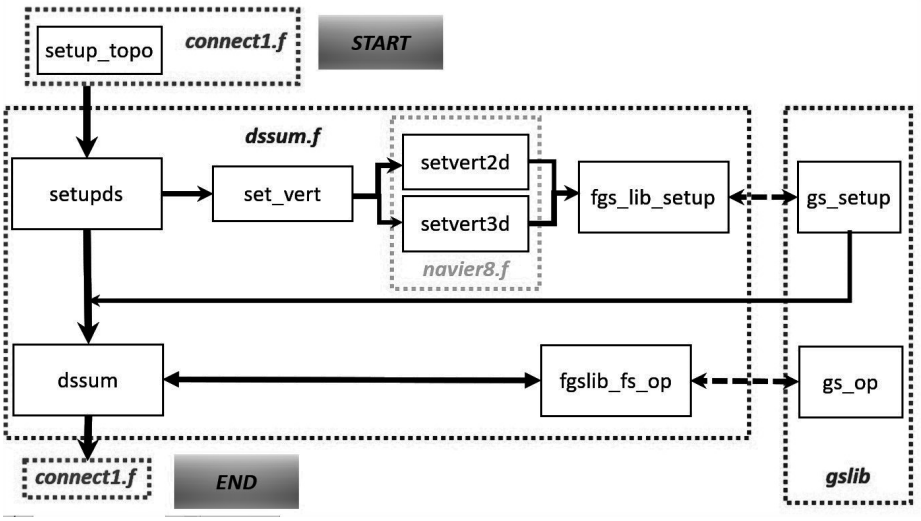


Figure 7: Flow chart

and continues to the `fgs_lib_setup` routine. This routine is essentially a mapped C to Fortran, which is invoked from the gs library. Its responsibility is to create a handle for the solution field, which is the information required on how to achieve the communication between the processors and the elements. Here is where the the type of communication is determined automatically, *e.g.* crystal router or pairwise communication, as the library makes some tests and chooses the methods which performs the best for the particular setup of our problem. The handle then is passed to the `dssum` routine, whereas now that the communication way is well established, the gather scatter operations can be achieved. These operations are again done through mapped C routines via the `gslib`. After the selected matrix manipulations are performed, the final result can be given back to the program to continue its course.

7.1 set_vert

The `set_vert` routine is the main routine handling the global numbering of the nodes. It is located in the `navier8.f` file.

```
      subroutine set_vert(glo_num,ngv,nx,nel,vertex,ifcenter)
c
c     Given global array, vertex, pointing to hex vertices, set up
c     a new array of global pointers for an nx^ldim set of elements
      .
c
      include 'SIZE'
      include 'INPUT'
c
      integer*8 glo_num(1),ngv
      integer vertex(1),nx
      logical ifcenter

      if (if3d) then
         call setvert3d(glo_num,ngv,nx,nel,vertex,ifcenter)
      else
```

```
         call setvert2d(glo_num,ngv,nx,nel,vertex,ifcenter)
      endif

c     Check for single-element periodicity 'p' bc
      nz = 1
      if (if3d) nz = nx
      call check_p_bc(glo_num,nx,nx,nz,nel)

      if(nio.eq.0) write(6,*) 'call usrsetvert'
      call usrsetvert(glo_num,nel,nx,nx,nz)
      if(nio.eq.0) write(6,'(A,/)') ' done :: usrsetvert'

      return
      end
```

Listing 1: set_vert routine

Depending on the problem dimensions, an IF structure selects between the 2D or 3D version of the routine, `setvert2d` and `setvert3d` accordingly. The output of the `set_vert` routine is the global numbering of all GLL nodes along the edges of each element of the domain.

7.2 setvert2d

The `setvert2d` is the 2D version of the `set_vert` routine and will be examined here for the sake of simplicity.

As inputs arguments there are `nx` which is the polynomial order of the interpolation scheme plus one, `nel` which is the number of local elements and `vertex` array, which for the 2D case consists of 4 integers which define the location of each vertex. These numbers represent the coordinates of the nodes of the edges that share the particular vertex. Lastly, there is `ifcenter` which is a logical switch. By default it is set to false. This means that the `set_vert` routine won't number the points inside the elements, (it will number only the points on its exterior on the edges). The numbering of the inner points is not necessary for most of implementations of gather-scatter.

The output arguments are `glo_num` array which will be consisting of a list of the global domain numbers of the points that the current processor is handling and `ngv` which will inform of how many of the points that were numbered are distinct.

In the beginning, the number of unique vertices is calculated by finding the higher number contained in the vertex array for the given processor. The vertices are numbered from the preprocessing. Some vertices will share common indices. The higher index contained in the vertex array is the number of unique vertices that has already been calculated. Next, those vertices are put in the `glo_num`. Even if the vertices do already possess a global number, they must be arranged in the `glo_num` array as well as the remaining nodes.

```
      ny  = nx
      nz  = 1
      nxyz = nx*ny*nz
c
      key(1)=1
      key(2)=2
      key(3)=3
c
```

```
c     Count number of unique vertices
      nlv  = 2**ldim
      ngvv = iglmax(vertex,nlv*nel)
      ngv  = ngvv
c
c     Assign hypercube ordering of vertices.
      do e=1,nel
         do j=0,1
         do i=0,1
c           Local to global node number (vertex)
            il  = 1 + (nx-1)*i + nx*(nx-1)*j
            ile = il + nx*ny*(e-1)
            glo_num(ile)   = vertex(i,j,e)
         enddo
         enddo
      enddo
      if (nx.eq.2) return
```

Listing 2: setvert2d routine, Part 1

Next thing is to find the global number of the interior points of the edges.

```
c     Assign edge labels by bounding vertices.
      do e=1,nel
         do j=0,1
         do i=0,1
            edge(i,j,1,e) = vertex(i,j,e)  ! r-edge
            edge(j,i,2,e) = vertex(i,j,e)  ! s-edge
         enddo
         enddo
      enddo

c     Sort edges by bounding vertices.
      do i=0,4*nel-1
         if (edge(0,i,1,1).gt.edge(1,i,1,1)) then
            kswap = edge(0,i,1,1)
            edge(0,i,1,1) = edge(1,i,1,1)
            edge(1,i,1,1) = kswap
         endif
         etuple(3,i+1) = edge(0,i,1,1)
         etuple(4,i+1) = edge(1,i,1,1)
      enddo

c     Assign a number (rank) to each unique edge
      m    = 4
      n    = 4*nel
      nmax = 4*lelt*nsafe  ! nsafe for crystal router factor of
     safety

      call gbtuple_rank(etuple,m,n,nmax,cr_h,nid,np,ind)
      do i=1,4*nel
         enum(i,1) = etuple(3,i)
      enddo
      n_unique_edges = iglmax(enum,4*nel)
```

Listing 3: setvert2d routine, Part 2

This is done by firstly defining the edges with the global number of their corresponding vertices. After listing all the edges, those that are unique must be distinguished. In order to do so, they are sorted out by their indices. As it was mentioned before, the edges are characterized by two numbers. Firstly,

the indices of each edge are swapped if necessary so that the smallest index in the first position of the array and the biggest index in the second. Then they are sorted out according to the first index, and the edges with the same first index, are sorted by their second index. After we sorting is finished,those edges that share the same numbers means that they are the same edge so they are not unique.

Lastly, after the numbering of all the points on the edges is done, there is the option of numbering the points in the interior of the elements. By default, this option is deactivated as their numbering is not necessary for the upcoming calculations in most cases. However, in case there is such a need, this can be done here, as long as the `ifcenter` logical switch is turned to True.

7.3 fgslib_gs_setup

The global numbering of all the points handled by the processor was achieved by the `set_vert` routine. This information is contained in the `glo_num` array. After returning to the main routine `setupds`, the next routine that is invoked is the `fglib_gs_setup` routine. The particular characteristing of this routines as well as some others which share a similar name, is that they come from the `gslib`, a compilation of code which exists on the `3rd_Party/gslib`, destination of Nek5000. These routines are written in C language and are responsible for most of the communication between the processors along with the actual implementation of Gather – Scatter.

7.4 Fortran - C gslib Interface

In order to be able to call them in the main program of Nek5000, a Fortran Interface must be firstly defined. This interface will allow the necessary C routines whose name begin mostly with gs to be mapped in Fortran. To distinguish the mapped Fortran routine from their C language counterpart, an f is added in their prefix.

```
#define fgs_setup_pick FORTRAN_NAME(gs_setup_pick ,GS_SETUP_PICK)
#define fgs_setup      FORTRAN_NAME(gs_setup      ,GS_SETUP      )
#define fgs            FORTRAN_NAME(gs_op         ,GS_OP         )
#define fgs_vec        FORTRAN_NAME(gs_op_vec     ,GS_OP_VEC     )
#define fgs_many       FORTRAN_NAME(gs_op_many    ,GS_OP_MANY    )
#define figs           FORTRAN_NAME(igs_op        ,IGS_OP        )
#define figs_vec       FORTRAN_NAME(igs_op_vec    ,IGS_OP_VEC    )
#define figs_many      FORTRAN_NAME(igs_op_many   ,IGS_OP_MANY   )
#define fgs_wait       FORTRAN_NAME(gs_op_wait    ,GS_OP_WAIT    )
#define fgs_fields     FORTRAN_NAME(gs_op_fields  ,GS_OP_FIELDS  )
#define fgs_free       FORTRAN_NAME(gs_free       ,GS_FREE       )
#define fgs_unique     FORTRAN_NAME(gs_unique     ,GS_UNIQUE     )
```

Listing 4: Fortran - C gslib interface

Thus after the mapping is defined, in `setupds` routine,`fgslib_gs_setup` is called, which is the routine responsible for creating the handle needed for the communication of the Gather Scatter operations.

7.5 dssum

After the creation of `gs_handle` and the establisment of the mpi communication, `dssum` can be initiated. The direct stiffness summation is done through the call of the `fgslib_gs_op`.

7.6 fgslib_gs_op

Another basic C to Fortran routine is `fgslib_gs_op`. This routine needs the particular handle that we have recently created to begin gather scatter operations on the matrices. Here is where the summation takes place and updates the values of local elements through this global operation and returns the updated values to their local place back again.

The `gs_op` subroutine takes the corresponding handle for each solution field and performs the selected operation for the communication and direct stiffness summation.

```
      subroutine dsop(u,op,nx,ny,nz)
      include 'SIZE'
      include 'PARALLEL'
      include 'INPUT'
      include 'TSTEP'
      include  'CTIMER'

      real u(1)
      character*3 op
      character*10 s1,s2
c
c     o gs recognized operations:
c
c             o "+" ==> addition.
c             o "*" ==> multiplication.
c             o "M" ==> maximum.
c             o "m" ==> minimum.
c             o "A" ==> (fabs(x)>fabs(y)) ? (x) : (y), ident=0.0.
c             o "a" ==> (fabs(x)<fabs(y)) ? (x) : (y), ident=
     MAX_DBL
c             o "e" ==> ((x)==0.0) ? (y) : (x),        ident=0.0.
c
c             o note: a binary function pointer flavor exists.
c
c     o gs level:
c
c             o level=0 ==> pure tree
c             o level>=num_nodes-1 ==> pure pairwise
c             o level = 1,...num_nodes-2 ==> mix tree/pairwise.
c
      ifldt = ifield
c     if (ifldt.eq.0)       ifldt = 1
      if (ifldt.eq.ifldmhd) ifldt = 1

c     if (nio.eq.0)
c    $   write(6,*) istep,' dsop: ',op,ifield,ifldt,gsh_fld(ifldt)

      if(ifsync) call nekgsync()

      if (op.eq.'+  ') call fgslib_gs_op(gsh_fld(ifldt),u,1,1,0)
```

Listing 5: dsop routine

The performed Gather Scatter operation can be either addition or any operation that the user thinks is necessary for the solution of his problem.

References

[1] G. C. Fox, W. Furmanski, "Hypercube Algorithms for Neural Network Simulation The Crystal Accumulator and the Crystal Router", California Institute of Technology Pasadena, CA 91125, February 1988

[2] M. O. Deville, P. F. Fischer, E. H. Mund, High-Order Methods for Incompressible Fluid Flow, Cambridge University Press, 2002

[3] Nicolas Offermans, "Gather-scatter library in Nek5000", Documentation of the gs library developed by James Lottes, November 2012

Solver stabilization

Marco Atzori*, Shahab Mirzareza † and
Arivazhagan Geetha Balasubramanian ‡

June 29, 2021

1 Introduction

In this report, we briefly describe stabilization techniques that are relevant for the spectral-element code *Nek5000*. If we consider the example of a simple partial differential equation (PDE), discretized as follows

$$\begin{cases} u_N^{n+1} = \mathcal{A}(\Delta x, \Delta t) u_N^n \\ \text{IC} : u_N^0 \end{cases}, \tag{1}$$

stabilization is needed in general because the numerical discretization can result in an amplification factor $||\mathcal{A}(\Delta x, \Delta t)||$ larger than 1. In this case, numerical errors will progressively grow, causing the solution to be nonphysical. The material in the report is summarized in Figure 1. In Section 2, we describe staggered grid and bubble functions, which are methods to avoid spurious modes in the solution of the Navier-Stokes equation that are due to coupling between velocity and pressure. In Section 3 and 4, we describe explicit filtering and relaxation-base filtering, respectively, which are two different approaches to remove energy from the highest modes in spectral and spectral-element methods. In Section 5, we discuss why over-integration or dealiasing is beneficial in the discretization of non-linear terms.

*KTH Engineering Mechanics, atzori@kth.se
†KTH Engineering Mechanics, shahabmi@kth.se
‡KTH Engineering Mechanics, argb@kth.se

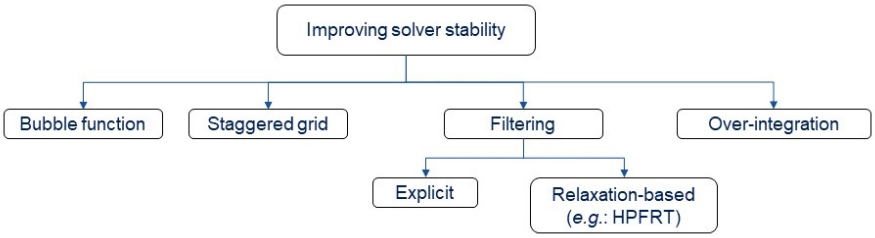


Figure 1: Overview of the stabilization techniques considered in the report.

2 Staggered grid and bubble functions

Using the same grid for pressure and velocity in the solution of the incompressible Navier-Stokes equation can result in spurious coupling modes. Since this phenomenon is well-known in computational fluid dynamics (CFD) and also observable in finite-different and finite-volume methods, we only discuss very briefly how to counteract it in *Nek5000*. A first possible approach is to use two different grids to represent pressure and velocity. In a spectral-element code, a natural way to implement this approach is to use different polynomial orders, which leads to the so-called $P_N - P_{N-2}$ formulation. In this formulation, velocity and pressure are expressed in terms of polynomials of order P_N and P_{N-2}, respectively, which can use either the same quadrature (collocated SEM) or a different (shifted) one (staggered SEM). Note that a "$P_N - P_{N-1}$ formulation" is not feasible when multiple elements are present because it does not satisfy the *inf-sup* condition [2].

A second possible approach to prevent the growth of the spurious modes is using the same grid for pressure and velocity but instead modifying the basis functions to restrict the solution space. This approach is denoted by $P_N - P_N$, because polynomials of the same order are used for velocity and pressure. The space of the solution is modified adding so-called "bubble functions", $X_N^b = X_n \oplus B_N$, so that the velocity can be expanded as:

$$u(x) = \sum_0^N u_i \phi_i(x) + \sum_0^N u_{b,i} b_i(x)\,. \tag{2}$$

Note that, in this approach for the given basis functions $b(x)$, the coefficients $u_{b,i}$ can be evaluated directly. However, as a general approach, the linear system of equations is rearranged so that only u_i are computed. The presence of the bubble functions leads to an additional term in the continuity equation to dampen the spurious modes in the solution of the full set of equations.

In *Nek5000*, both the $P_N P_{N-2}$ and $P_N P_N$ methods are implemented. In the first case, Gauss-Lobatto-Legendre (GLL) points and Gauss-Lobatto (GL) points are used for velocity and pressure, respectively. In the second case, GLL points are used for velocity and pressure.

We refer to Sections 3.6, 4.5, and 5.6 of the textbook by Deville *et al.* [2] for more details on these topics.

3 Explicit filtering

The key idea in explicit filtering is to suppress the spurious modes at the end of each time step. In the method introduced by Fischer and Mullen [3], instead of adding a dissipation term directly to the Navier-stokes equation, a low-pass filter function built in the modal space is used. A solution u in physical space can be expanded using the modal bases as,

$$u(x) = \sum_{k=0}^{N} \hat{u}_k \phi_k(\xi)\,, \tag{3}$$

where, $\hat{u}_k$ are the spectral coefficients and ϕ_k are the polynomial basis functions (*e.g.* Boyd-transformed basis, Legendre polynomials, and etc). In the

interpolation-based technique introduced by Fischer and Mullen [3], the spectral representation of the solution is replaced with a filtered solution such that the spectral coefficients are scaled by a filter transfer function σ. As a result, the filtered solution becomes,

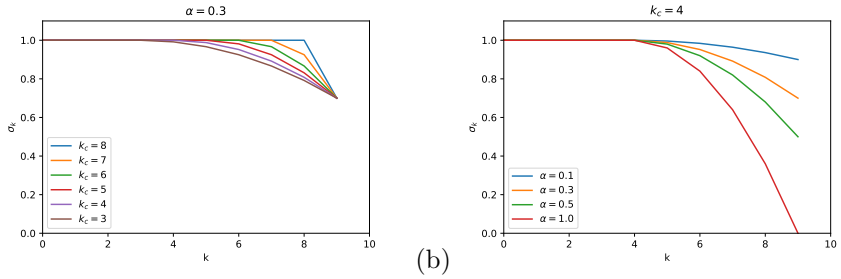
$$\bar{u} = \hat{F}(u) = \sum_{k=0}^{N} \sigma_k \hat{u}_k \phi_k(\xi)\,, \tag{4}$$

where, the filter-transfer function σ as outlined in [2] is defined as,

$$\sigma_k = \begin{cases} 1 - \alpha \left(\frac{k-k_c}{N-k_c}\right)^2, k > k_c \\ 1, k \leq k_c \end{cases}\,. \tag{5}$$

The filter-transfer function is characterized by:

- amplitude α,
- cut-off mode k_c.



(a) (b)

Figure 2: Variation of the filter-transfer function with respect to (a) cut-off mode (b) filter strength

The variation of the filter-transfer function is shown in Figure 2. Typically, a filter strength α of 0.05 to 0.3 is chosen to yield smoother results [2].

3.1 Example: Stokes problem

The filtering procedure applied to an unsteady Stokes problem is briefly explained here. The intermediate unfiltered solution is obtained as,

$$\begin{aligned} \mathbf{H}\tilde{\mathbf{u}} - \mathbf{D}^T p^{n+1} &= \mathbf{B}\mathbf{f}^n\,, \\ \mathbf{D}\tilde{u} &= 0\,, \end{aligned} \tag{6}$$

where, $\mathbf{H}$ is the discrete equivalent of the Helmholtz operator given by,

$$\mathbf{H} = \frac{1}{Re}\mathbf{A} + \frac{1}{\Delta t}\mathbf{B}\,,$$

and, $\tilde{\mathbf{u}}$ is the intermediate unfiltered solution, $-\mathbf{A}$ is the discrete Laplacian, $\mathbf{B}$ is the mass matrix, $\mathbf{D}$ is the discrete divergence operator, $\mathbf{f}^n$ is the explicit treatment of non-linear terms. Now, the solution at t^{n+1} becomes,

$$\mathbf{u}^{n+1} = F_\alpha \tilde{\mathbf{u}}\,. \tag{7}$$

Once the divergence-free solution is calculated, the filtering is performed on the divergence-free solution to obtain the solution at t^{n+1}. Here, the filtering operator with a particular filter strength α is given by, F_α. One of the requirements of the filter function would be to avoid the loss of C^0 continuity and that the values at the element boundaries are preserved after the application of the filter, which would make the filtering operation to be local to each spectral-element.

3.2 Advantages

Apart from the simplicity of the method, when the low-pass filter is applied, the inter-element (C^0) continuity is lost and this can be recovered by using appropriate basis which would avoid the need for element to element information transfer and hence, this method is efficient. The interpolation-based technique as introduced in the next section will ensure that the inter-element continuity is preserved when filtering operation is performed on the basis provided by the Boyd transform (discussed in § 3.3.3). The interpolation error $\|\tilde{\mathbf{u}} - \Pi_{N-1}\tilde{\mathbf{u}}\|$ tends to 0 as $N \to \infty$ at an exponential rate and hence, spectral convergence is not majorly affected.

3.3 Interpolation-based filtering technique: framework

To understand the effect of interpolation-based filtering procedure, a polynomial $u_N \in \mathbb{P}_N$ is considered. The polynomial can be expanded as,

$$u_N(x) = \sum_{k=0}^{N} u_k \pi_{N,k}(x)\,, \tag{8}$$

where u_k are the values of the solution in the nodes $\{\xi_{N,k}\}$ associated to Gauss-Lobatto-Legendre (GLL) quadrature rule, and $\pi_{N,k}$ are the Lagrangian interpolation polynomials. The nodes are provided by,

$$\xi(k) = \begin{cases} -1, k = 0 \\ L_N^{'}(x) = 0, 1 \leq k \leq N-1 \\ 1, k = N \end{cases},$$

where, $L_N(x)$ corresponds to the Legendre polynomials of degree N. Defining the matrix operators for a polynomial degree, N, and order of the scheme, M),

$$\left(I_N^M\right)_{i,j} = \pi_{N,j}\left(\xi_{M,i}\right) \in \mathbb{R}^{(M+1)\times(N+1)}, 0 \leq i \leq M, 0 \leq j \leq N \tag{9}$$

Here, the matrix operator I_M^N corresponds to Lagrangian polynomial of degree N in $M+1$ GLL nodes. The pseudo-projector operator is defined as the product of the matrix operators as,

$$\Pi_{N-1} = I_{N-1}^N I_N^{N-1} \in \mathbb{R}^{(N+1)\times(N+1)} \tag{10}$$

If the pseudo-projector is applied to u_N, it provides Lagrangian interpolated polynomials of degree $N-1$ on the GLL grid ξ_N. This Lagrangian interpolation of polynomial u_N with the degree $N-1$ automatically eliminates the high-frequency content, thus adding some numerical dissipation and improving the stability [2]. The reason for pseudo-projection eliminating high frequency content is due to the interlacing property of Legendre polynomials.

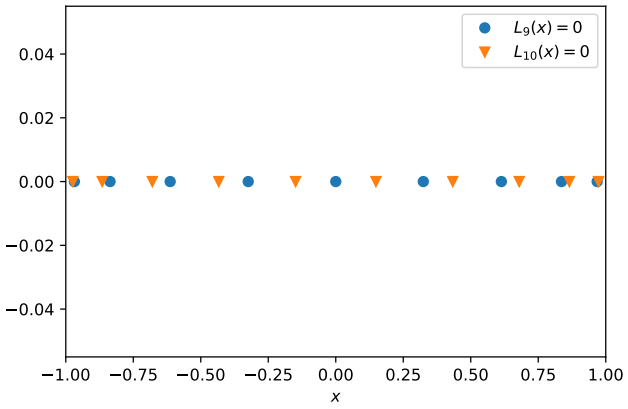


Figure 3: Interlacing property of the Legendre polynomials

3.3.1 Interlacing

The zeros of the Legendre polynomials are real and distinct and lie within the interval of $(-1, 1)$. The theorem for interlacing of zeros of orthogonal polynomials (as stated in the notes by Krestin [4]) is that, "*If* $\{p_n(x)\}_{n=0}^{\infty}$ *is a sequence of orthogonal polynomial on an interval* (a, b) *with respect to the weight function* $w(x)$*, then the zeros of* $p_n(x)$ *and* $p_{n+1}(x)$ *separate each other.*"

So, if $\xi_{N,k}$ and $\xi_{N+1,k}$ denote the zeros of the Legendre polynomials L_N and L_{N+1} respectively, then in each sub-interval formed by L_N we have exactly one zero of L_{N+1} as shown in Figure 3. i.e., $-1 < \xi_{N+1,1} < \xi_{N,1} < \xi_{N+1,2} < \xi_{N,2} < .. < \xi_{N+1,N} < \xi_{N,N} < \xi_{N+1,N+1} < 1$. Hence, when the pseudo-projection is performed from degree $N+1$ to degree N, there is an effect in the high frequency component.

3.3.2 Effect of pseudo-projection

To provide an understanding on the effect of the pseudo-projection, the difference between the actual and pseudo-projected polynomial is considered.

$$v(\xi) = u(\xi) - \Pi_{N-1} u(\xi)\,. \tag{11}$$

For a polynomial $u \in \mathbb{P}_N$, we have $v \in \mathbb{P}_N$. Using the relationship between the Legendre polynomials and their derivatives we obtain,

$$\begin{aligned} v(\xi) &= \tilde{u}_N \left(1 - \xi^2\right) L_{N-1}^{'}(\xi) \\ &= \hat{u} \left[L_N(\xi) - L_{N-2}(\xi)\right] \\ &= \hat{u} \phi_N(\xi)\,, \end{aligned} \tag{12}$$

where, $\hat{u}$ is a constant and ϕ is a basis-function. So, basically the effect of pseudo-projection is in the last mode provided by the basis functions ϕ.

3.3.3 Boyd's transform

The basis functions as discussed in the previous section are provided by the difference between two Legendre polynomials. The new basis suggested by Boyd [1]

is,

$$\phi_k = \begin{cases} \frac{1-\xi}{2}, k = 0 \\ \frac{1+\xi}{2}, k = 1 \\ L_k - L_{k-2}, k > 1 \end{cases} . \tag{13}$$

Though the filtering procedure is simple, it does not satisfy the boundary condition of the solution, if the explicit filtering operation is performed in the original basis. The solution to this problem is to apply the filtering in the transformed basis provided by Boyd such that, $u_N(\pm 1) = \bar{u}_N(\pm 1)$.

3.3.4 Filter construction

For a polynomial $u \in \mathbb{P}_N$, the expansion in modal basis given by the Boyd transform is,

$$u_i = u(\xi_{N,i}) = \sum_{k=0}^{N} \hat{u}_k \phi_k(\xi_{N,i}), \quad i = 1, 2, .., N \tag{14}$$

For the interpolation-based filtering, the one-dimensional filter is given by,

$$\hat{F}_\alpha = \alpha \Pi_{N-1} + (1 - \alpha) I_N^N \tag{15}$$

where, Π_{N-1} is the pseudo-projection operator to $\mathbb{P}_{N-1}$ and I_N^N is the identity matrix. From equation (15) using a filter strength α, only a fraction of the mode can be filtered. If $\alpha = 1$, then it would correspond to full projection onto $\mathbb{P}_{N-1}$.

For a general explicit filter construction using a filter transfer function α, a transformation matrix, $(Z)_{i,k} = \phi_k(\xi_{N,i})$, is defined. The modal to physical transformation can be performed as: $u = Z\hat{u}$. An arbitrary 1D filter can be constructed that scales $\hat{u}_k$ for $k > 1$ as,

$$\hat{F}_\Sigma = Z \Sigma Z^{-1}, \tag{16}$$

where, diag(Σ) contains the filter transfer function σ_k for $k > 1$ and $\sigma_0 = \sigma_1 = 1$. In higher space dimensions, the filter function is given as the tensor-product of the filter function in 1D.

$$F = \hat{F} \otimes \hat{F} \otimes \hat{F}$$

3.4 Limitations

The limitations of the explicit filtering operation as highlighted by Deville *et al.* [2] are that,

- It is time-dependent,
- It is non-dissipative, and
- It violates the divergence-free condition.

The last two points are discussed in detail.

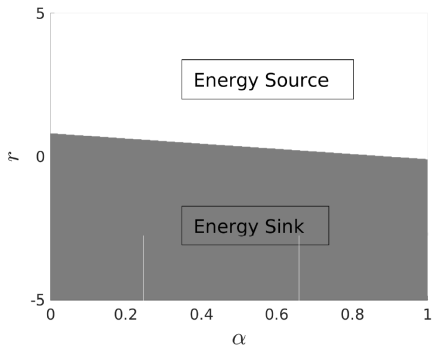


Figure 4: Dissipative and non-dissipative characteristics of filtering for a polynomial of order 10. Here, $r = \frac{\hat{u}_{N-2}}{\hat{u}_N}$. Source: Negi *et al.*, (2017) [7]

3.4.1 Non-dissipative nature of explicit filtering

Because of the Boyd transform, the filtering operation is strictly non-dissipative and maybe a source of energy in certain situations. To understand the dissipative characteristics of the filter, if only a fraction of α is filtered in the last mode ϕ_N, the mode ϕ_{N-2} needs to be modified to preserve the boundary conditions. The difference in the energy between the filtered and the unfiltered fields in this case is given by,

$$\begin{aligned}\Delta E &= \|u_N\|^2 - \|\bar{u}\|^2 \\ &= \hat{u}^2 \left[\left(\frac{\hat{u}_{N-2}^2}{\hat{u}_N^2} - \left(\frac{\hat{u}_{N-2}}{\hat{u}_N} + \alpha \right)^2 \right) \phi_{N-2}^2 + \left(1 - (1-\alpha)^2 \right) \phi_N^2 \right]. \end{aligned} \tag{17}$$

The value of $\Delta E/\hat{u}_N^2$ is plotted as a function of the filter strength α and $\hat{u}_{N-2}/\hat{u}_N$ as shown in Figure 4. From Figure 4, it is observed that in the large portion of the represented area, the filter acts as an energy source and introduces energy into the solution rather than acting as a sink by dissipating the energy.

3.4.2 Violation of the divergence-free condition

The violation of the divergence-free condition due to explicit filtering operation is basically because of the fact that filter and the spatial derivative operators do not commute. Since, the explicit filtering is performed on the divergence-free field, the filtered field is not divergence free.

A plot in time for the violation of divergence-free condition is shown in Figure 5. In this case, a doubly periodic domain negates the errors due to the boundary terms. Also, since the non-linear term is calculated using over-integration to preserve skew-symmetry of the advection term, the sources of instability can be either from characteristic time-stepping scheme or the violation of divergence free condition. From the Figure 5, it is observed that in the beginning there is less difference between the filtered and un-filtered case. This is because, there is less energy at the highest mode during the start of the simulation and as it

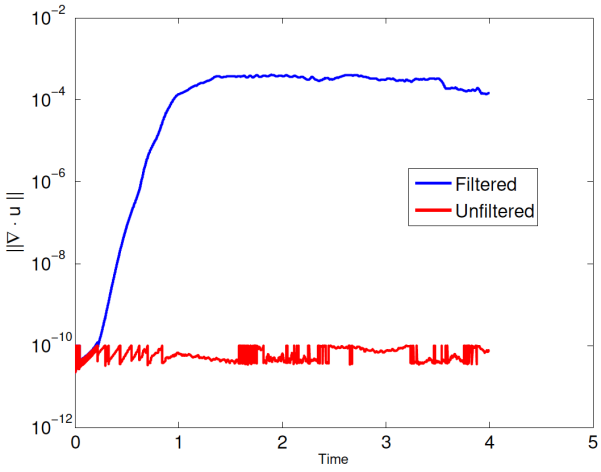


Figure 5: Violation of divergence-free condition in a double shear layer case in 2D domain. Here, the simulation was performed with 256 spectral grids with 16 Legendre polynomials. The tolerance for divergence-free condition is 10^{-10}. Source: Negi *et al.*, (2017) [7]

proceeds, there is a significant energy in the smaller scales and the difference increases. This divergence error can be severe for marginally-resolved cases.

4 Relaxation-based filtering

As shown in the results of explicit filtering, there are some drawbacks associated with it. In such a scenario, alternative methods like relaxation-term based filtering can be considered in a way that they remedy potential drawbacks of the explicit filtering, while preserving its advantages, namely efficiency and simplicity [7].

Considering explicit filtering applied to a dynamical system equation, the following equations are obtained:

$$\frac{\partial u}{\partial t} + F(u) = 0\,, \tag{18}$$

$$\mathbf{u}^{n+1} = \mathbf{u}^n - F(\mathbf{u}^n)\Delta t + O(\boldsymbol{\Delta}\mathbf{t}^2)\,, \tag{19}$$

$$\mathbf{u}^{n+1} = G(\mathbf{u}^*)\,, \tag{20}$$

where $F(u)$ and G are evolution and low-pass filter operators, respectively. By performing an implicit time-relaxation filtering on the evolution equation, we get:

$$\frac{\partial u}{\partial t} + F(u) = -XH(u)\,, \tag{21}$$

which can be integrated in time as,

$$\mathbf{u}^* = \mathbf{u}^n - F(\mathbf{u}^n)\Delta t + O(\boldsymbol{\Delta}\mathbf{t}^2)\,, \tag{22}$$

$$\mathbf{u}^{n+1} = \mathbf{u}^* - XH(\mathbf{u}^*)\Delta t + O(\boldsymbol{\Delta}\mathbf{t}^2)\,. \tag{23}$$

By considering $X = \frac{1}{\delta t}$, the latter is written as:

$$\mathbf{u}^{n+1} = G(\mathbf{u}^*)\,. \tag{24}$$

The relaxation-term in the above equations can be interpreted as a low-pass filter operation G performed every $\frac{1}{X\delta t}$ time-steps. Thus the "explicit filter-based stabilization" is equivalent to "relaxation-term-based stabilization", which is referred to as **"RT stabilization"** [9].

RT filter can be applied to the Navier-Stokes equations by a simple addition of a relaxation-term to the right-hand-side as:

$$\frac{\partial u}{\partial t} + u.\nabla u = -\frac{\nabla p}{\rho} + \nu\nabla^2 u - XH(u)\,, \tag{25}$$

, and

$$\nabla.u = 0\,. \tag{26}$$

Here $H(u)$ is a high-pass-filtered velocity field. The parameter X in RT filters can be used as a weighting parameter similar to the filter weight α defined in equation (15).

4.1 Relaxation-based filter parameters

In the context of spectral-elements, high-pass filter H for the relaxation-based stabilization can be expanded as:

$$H(u_N) = \sum_{k=0}^{N} \gamma_k a_k L_k\,, \tag{27}$$

where L_k and a_k are Legendre polynomials used as basis functions and spectral coefficients for the finite series expansion of the solution respectively. γ_k is the filter transfer function defined as:

$$\gamma(k) = \begin{cases} 0, k \leq kc \\ (\frac{k-k_c}{N-kc})^2 k > k_c \end{cases}\,, \tag{28}$$

where k_c is the cutoff mode and N is the polynomial order.

4.2 Effect of relaxation-based filtering on the eigenvalues of the system

Applying an RT filtering causes the eigenvalues to shift towards the negative real plane. Therefore, dissipation is added to the system. As an illustration as shown in [7] an unstable system following the spectral-element framework was built in MATLAB and the linear advection operator was stabilized by a relaxation term. While the RT filter clearly has a stabilizing effect on the system, it can also be destabilizing for a certain range of parameters. For a particular temporal discretization such as the BDF-EXT-3 [2], the region of stability may cover a finite region of the negative plane. For a system with eigenvalues such that $\lambda\delta t$ falls outside the stability region of the time-stepping scheme, the stabilization procedure itself becomes numerically unstable and the time-step size needs to be reduced in order to render the simulation numerically stable again [7].

4.3 Cutoff ratio versus cutoff mode

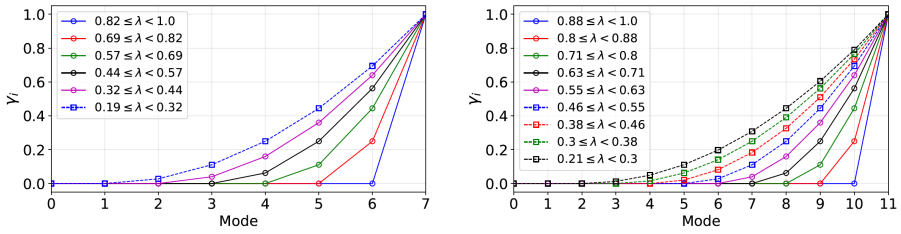
In both of the explicit and RT filtering approaches, the cutoff mode k_c appears. However, as an input to the **.par* files in *Nek5000* simulations, one can set a value for the cutoff ratio, λ and filter weight. The filter cutoff ratio is then converted to the filter cutoff mode using the following relations:

$$\lambda^* = max(nint((N+1))(1-\lambda)) - 1.0)\,, \tag{29}$$

and,

$$k_c = int(N - int(\lambda^*) - 1)\,. \tag{30}$$

where, N is the polynomial order, `int(.)` converts an input number to integer, and `nint(.)` converts an input to the nearest integer number. In Figure 4.3, γ_i is plotted versus mode number [8].



(a) (b)

Figure 6: Normalized amplitude γ_i of the hpfrt filter versus mode number for $N = 7$ (a) and $N = 11$ (b), taken from [8].

As the plots show, For a given N, different values of λ within a certain range yield the same integer value for k_c

4.4 Implementation

Types of the filtering used in *Nek5000* are *none*, *explicit*, *hpfrt* which can be chosen in **.par* file. Filtering weight and filter cutoff ratio which are known as **p103** and **p101** can also be specified in the **.par* file as shown in Listing 1

```
filtering=hpfrt          ! none, explicit, hpfrt
filterWeight=10          ! filter weight
filterCutoffRatio=0.9    ! filter cut off ratio
```

Listing 1: Filter settings specified in the **.par* file

Filter type, filter weight and filter cutoff ratio can then be read by *reader-par.f* as shown in Listings 2, 3, 4.

```
if (index(c_out,'NONE').eq.1 ) then
 fltertype=0
 goto 101
else if (index(c_out,'EXPLICIT').eq.1 ) then
 fltertype=1
 call ltrue(iffilter,size(iffilter))
else if (index(c_out,'    HPFRT').eq.1 ) then
 fltertype=2
```

```
 call ltrue(iffilter,size(iffilter))
else
 write(6,*) 'value: ', c_out
 write(6,*) 'is invalid for general:filtering!'
 goto 999
endif
```

Listing 2: Filter type in *reader-par.f*

```
call finiparser_getDbl(d_out,'general:filterWeight',ifnd)
if (ifnd .eq.1 ) then
 param(103)=d_out
else
 write(6,*) 'general:filterWeight'
 write(6,*) 'is required for general:filtering'
 goto 999
endif
```

Listing 3: Filter weight in *reader-par.f*

```
call finiparser_getDbl(d_out,'general:filterModes',ifnd)
if (ifnd .eq.1 ) then
 param(101)=int(d_out)-1
 if (int(param(101)) .eq. 0) filterType=0
else
 call finiparser_getDbl
$       (d_out, 'general:filterCutoffRatio',ifnd)
 if (ifnd .eq. 1) then
  dtmp=anint(lx1*(1.0-d_out))
  param(101)=max(dtmp-1,0.0)
  if (abs(1.0-d_out) .lt. 0.01) filterType=0
 else
  write(6,*) 'general:filterCutoffRatio or filterModes'
  write(6,*) 'is required for general:filtering!'
 goto 999
endif
```

Listing 4: Filter cut off ratio in *reader-par.f*

If explicit filtering is chosen to be implemented, the Boyd transform and filter transfer function are performed in the subroutine *Navier.f* are shown in Listings 5 and 6.

```
do j=1, nx
   z=zpts (j)
   call legendre_poly(Lj,z,n)
   kj=kj+1
   pht(kj)=Lj(2)
   do k=3,nx
      kj=kj+1
      pht(kj)= Lj(k) - Lj(k-2)
   enddo
enddo
```

Listing 5: Implementation of the Boyd transform in *Navier5.f*

```
do k=k0+1, nx
   kk=k+nx*(k-1)
   amp=wght*(k-k0)*(k-k0)/(kut*kut)
   diag(kk)=1.-amp
enddo
```

Listing 6: Implementation of explicit filter transfer function in `Navier5.f`

If hpfrt is chosen, there will be no need for the Boyd transform as Legendre polynomials are directly used. Filter transfer function for hpfrt is performed in the subroutine *hpf-trns-fcn* and shown in Listing 7

```
k0=nx-kut      !kut=additional modes
do k=k0+1,nx
   kk= k+nx*(k-1)
   amp=((k-k0)*(k-k0)+0.)/(kut*kut+0.)
   diag(kk)=1.-amp
enddo
```

Listing 7: Implementation of hpfrt transfer function in *Navier5.f*

5 Over-integration

In spectral and spectral-element methods, the spectral and physical representations of the solution are both used to carry out different operations. In particular, derivatives are computed in spectral space and multiplications, which would be convolutions in spectral space, are computed in physical space. We intend with "over-integration" to select a larger set of quadrature points in physical space to represent the variables that will be multiplied, compared with the number of points needed to represent each variable individually.

In a numerical code such as *Nek5000*, where the velocity is expressed in terms of a nodal base of Legendre polynomials, no transformation is in principle needed between the two representations but performing a point-wise multiplication in physical space still requires to expand the number of quadrature points.

In the present section, we will summarise the results of Kirby and Karniadakis [5] and Malm *et al.* [6], who explained how over-integration is beneficial in different scenarios, and we describe its implementation in *Nek5000*, version 19.

5.1 Over-integration as dealiasing

Kirby and Karniadakis [5] discussed how aliasing error can cause numerical errors if over-integration is not employed to evaluate in polynomial multiplications.

In the case of Gauss-Lobatto-Legendre quadrature, Q quadrature points (and associated weights) allow to integrate exactly polynomials $u(\xi) \in P_{2Q-3}$. Inverting this relation allows to estimate the number of quadrature points needed to integrate polynomials of a given order, in particular:

$$\begin{aligned} u(\xi) \in P_N \quad &\text{requires} \quad Q = (N+3)/2\,, \\ u(\xi) \in P_{2N} \quad &\text{requires} \quad Q = (2N+3)/2\,, \\ u(\xi) \in P_{3N} \quad &\text{requires} \quad Q = (3N+3)/2\,. \end{aligned} \tag{31}$$

In Galerkin methods, we have inner products of polynomial of the same degree and, given that $u(\xi) \in P_N$ has at most $M = N+1$ modal coefficients, it follows that:

$$\begin{aligned} [u(\xi)]^2 \in P_{2N} \quad &\text{requires} \quad Q = M + 1/2 = M + 1\,, \\ [u(\xi)]^3 \in P_{3N} \quad &\text{requires} \quad Q = 3M/2\,. \end{aligned} \tag{32}$$

Note that $[u(\xi)]^2 \in P_{2N}$ and $[u(\xi)]^3 \in P_{3N}$ correspond to linear terms (products of two polynomials) and the advection term (product of three polynomials), respectively.

Kirby and Karniadakis [5] firstly considered a test case for the advection term, using the following procedure:

1. One element in $[-1, 1]$ with $M = 16$ modes is created.
2. All modal coefficients are set equal to 1 (this condition corresponds to a case with very low resolution).
3. Evaluate the modal form on Q quadrature points.
4. Pointwise square the values at quadrature points.
5. Apply derivative matrix (rank $Q \times Q$).
6. Compute modal coefficients with Gaussian integration.

In this test case, the only parameter is the number of quadrature points Q, and the procedure is repeated increasing Q from $M + 1 = 17$ (the required values for the linear terms) to 25 ($Q = 3M/2 = 24$ is the required values for the advection term). Comparing the results of the different tests with that of the highest number of quadrature points, $Q = 25$, shows the occurrence of the aliasing error up to $Q = 24$, confirming that the cause of this kind of error is the increase of the polynomial order in the multiplication.

A second example considered in Ref. [5] is the Burgers equation,

$$\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial u^2}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}\,, \tag{33}$$

integrated with $M = 16$ in a one-dimensional domain with 5 spectral elements. The initial condition in this test case exhibits a steep gradient in the central element, which therefore has high values for all most modal coefficients. In this example, the aliasing error caused numerical instability for low viscosity values, when the natural dissipation is not sufficient to avoid disturbance growth. For high values of the viscosity, the solution is stable, but the aliasing error still causes an higher discrepancy with the analytical solution when over-integration is not used.

Kirby and Karniadakis [5] also considered two different sets of simulations of incompressible flow: laminar-to-turbulent transition in a triangular duct and under-resolved simulations of a turbulent channel flow. The duct simulations show good agreement between results obtain with over-integration and with an higher polynomial order but without over-integration, showing that the aliasing error is less severe when a very high resolution is used and the highest modes contain a small amount of energy. The under-resolved simulations of the turbulent channel flow show that both over-integration and spectral vanishing viscosity (SVV) have similar effects, stabilizing the solution, as the aliasing error eventually results in injecting energy in the system.

5.2 Advection problem and geometrical transformation

Although the results by Kirby and Karniadakis [5] illustrate how aliasing errors can lead to instabilities, an increase of the polynomial order can also occur in operations for which aliasing is not possible. Starting from previous observations that stability issues also arise in linear scalar transport equation and that stabilization is sometimes not needed when the skew-symmetric form of the convective term is used, Malm *et al.* [6] investigated different scenarios where a numerical errors can be avoided using over-integration.

In the model problem of the advection-diffusion equation,

$$\frac{\partial u_i}{\partial t} + c_j \frac{\partial u_i}{\partial x_j} = \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j}\,, \tag{34}$$

if the convective field $\boldsymbol{c}$ is divergence-free (*i.e.* $\partial c_i/\partial x_i = 0$), the advection term can be expressed in three different forms:

$$c_j \frac{\partial u_i}{\partial x_j} = \frac{\partial c_j u_i}{\partial x_j} = \frac{1}{2}\left(c_j \frac{\partial u_i}{\partial x_j} + \frac{\partial c_j u_i}{\partial x_j} \right) \tag{35}$$

denoted "convective", "conservative", and "skew-symmetric" forms, respectively. Although these forms are equivalent from an analytical perspective, this is not necessarily the case for the corresponding discretized operator. In particular, the skew-symmetric form is less prone to numerical instabilities than the convective or the conservative forms. This fact strongly suggests that instabilities are due to a lost of skew-symmetry when discretization is applied.

In a Galerkin method, the problem is solved in the weak form, writing the advection-diffusion equation as:

$$\frac{\mathrm{d}}{\mathrm{d}t}(v,u) + (v, \boldsymbol{c}\cdot\nabla u) = -\nu(\nabla v, \nabla u)\,, \quad \forall v \in H_0^1\,, \tag{36}$$

and finding a function $u \in X^N$ for which the equation is satisfied $\forall v \in H_0^1$, where H_0^1 is the space of the test functions and $X^N \subset H_0^1$. In this expression, $(\dots,\dots)$ denotes a bilinear form defined in H_0^1. Solution and test functions are expressed in terms of a base $\Phi_1(x),\dots,\Phi_2(x)$, so that:

$$u(x) = \sum_i u_i \Phi_i(x) \qquad v(x) = \sum_i v_i \Phi_i(x)\,, \tag{37}$$

and the semi-discretized equation is an ordinary differential equation of the form:

$$B\frac{\mathrm{d}\underline{u}}{\mathrm{d}t} + C\underline{u} = -\nu A\underline{u} \tag{38}$$

Where the three matrices A, B, and C, are defined as:

$$A_{ij} = (\nabla\Phi_i, \nabla\Phi_j)\,, \qquad B_{ij} = (\Phi_i, \Phi_j) \quad \textit{and} \quad C_{ij} = (\Phi_i, \boldsymbol{c}\cdot\nabla\Phi_j) \tag{39}$$

are the stiffness, mass, and convective matrices, respectively. The stifness and mass matrices are symmetric and positive-definite. Applying integration by parts, it can be shown that:

$$C_{ij} = -\int_\Omega \Phi_j\, \boldsymbol{c}\cdot\nabla\Phi_i \,\mathrm{d}V - \int_\Omega \Phi_i\Phi_j\, \nabla\cdot\boldsymbol{c}\,\mathrm{d}A + \int_{\partial\Omega} \Phi_i\Phi_j\, \boldsymbol{c}\cdot\hat{\boldsymbol{n}}\,\mathrm{d}A\,. \tag{40}$$

```
parameter(lx1=9)      ! GLL points per element along each direction
parameter(lxd=14)     ! GL  points for over-integration (dealiasing)
parameter(lx2=lx1-2)  ! GLL points for pressure (lx1 or lx1-2)
```

Listing 8: Array dimensions for $P_N P_{N-2}$ with dealiasing.

```
1.000000          p99     dealiasing: if <0 disable
```

Listing 9: Parameter **p99** set in a **.rea* file.

The second last and the last integrals in this example, vanish because $\boldsymbol{c}$ is divergence-free and due to the boundary conditions, showing that the convective matrix is skew-symmetric *i.e.* $C_{ij} = -C_{ji}$. The stability of the solution is particularly interesting in the limit $\nu \to 0^+$, which describes a situation when features of the solution becoming progressively smaller and eventually finer than the grid spacing. In this limit, the semi-descretized advection-diffusion equation becomes:

$$\frac{\mathrm{d}\underline{u}}{\mathrm{d}t} = -B^{-1} C\, \underline{u}\,. \tag{41}$$

If the convective matrix is skew-symmetric, all eigenvalues of the system are purely imaginary, and the solution will be absolutely stable for a small enough time steps.

The fact that the stability of the solution is linked to the skew-symmetry of the convective matrix explains that instability can be a consequence of numerical errors resulting in the lost of skew-symmetry and, on the other hand, that removing the source of these errors and recovering the skew-symmetry is an effective way of stabilizing the solution.

Malm *et al.* [6] showed different instances when numerical errors due to insufficient quadrant can occur in the advection-convective equation. In the test case of the convected-cone problem, over-integration is not needed for a vortical convective field (*i.e.* $\boldsymbol{c} = (-y, x)$) but it is beneficial for a stagnation field (*i.e.* $\boldsymbol{c} = (-x, y)$). Another source of quadrature errors can be the Jacobian that needs to be included when deformed elements are considered. In general, over-integration is not required when C_{ij} is skew-symmetric independently of the quadrature, which can be proved for certain transformations or convective field.

When the Navier-Stokes equation is considered, the same considerations apply, but it is important to consider that the divergence-free condition is imposed only up to a certain tolerance, making over-integration not enough to assure the skew-symmetry of the convective matrix.

5.3 Implementation

To enable over-integration in *Nek5000* requires that a) arrays of the proper size are allocated, and b) that parameter **p99** has a value larger than 0. Different values of **p99** can be used in different cases. The array sizes are determined before compilation in the *SIZE* file of each case, as reported in Listing 8 from */Nek5000/examples/cone/cone016/SIZE*. In this example, $lx1$ and $lx2$ are the array sizes of velocity and pressure, respectively, and lxd is the size of the array used for over-integration. Parameter **p99** can be set either directly, as shown in */Nek5000/examples/cone/cone016/base.rea* (Listing 9), or using a the appropri-

```
dealiasing = yes
```

Listing 10: Parameter **p99** set in a *.par* file.

```
 if (param(99).gt.-1 .and. (lxd.lt.lx1 .or. lyd.lt.ly1 .or.
&    lzd.lt.lz1)) then
   if(nid.eq.0) write(6,*)
&    'ABORT: Dealiasing space too small; Check lxd,lyd,lzd in SIZE '
   call exitt
 endif
```

Listing 11: Lines 1153-1158 in *core/reader_par.f*

ate keyword in a **.par* file, as shown in */Nek5000/examples/cone/phill/phill.rea* (Listing 10). Note, however, that the default value of **p99** is already larger than 0.

Before the first time step, if over-integration is enabled, the array dimensions are checked in *core/reader_par.f* (Listing 11). During the simulation, the subroutines where over-integration is performed are called at each evaluation of the convective term. In case of **p99** equal to 4, which is the default value for a simulation of incompressible flow this corresponds to Listing 12, in *core/navier1.f*, and the subroutine is *convect_new*.

```
elseif (param(99).eq.4) then
 if (ifpert) then
  call convect_new (conv,fi,.false.,vx,vy,vz,.false.)
 else
  call convect_new (conv,fi,.false.,vxd,vyd,vzd,.true.)
endif
```

Listing 12: Lines 3198-3203 in *core/navier1.f*

The subroutine *convect_new* is implemented in *core/convect.f*. Listing 13 shows two selected parts of *convect_new*, corresponding to the interpolation from the coarse grid to a fine grid before the multiplication, and the interpolation from the fine grid to the coarse grid after the multiplication.

```
 subroutine convect_new(bdu,u,ifuf,cx,cy,cz,ifcf)
[...]
    call intp_rstd(fx,cx(ic),lx1,lxd,if3d,0) ! 0 --> forward
    call intp_rstd(fy,cy(ic),lx1,lxd,if3d,0) ! 0 --> forward
    if (if3d) call intp_rstd(fz,cz(ic),lx1,lxd,if3d,0) ! 0 -->
    forward
[...]
    call intp_rstd(bdu(ib),uf,lx1,lxd,if3d,1) ! Project back to
    coarse
```

Listing 13: Lines 638-720 in *core/convect.f*

Note as the interpolations in both direction are computed using the subroutine *intp_rstd* with two values of the last argument.

References

[1] John P Boyd. Two comments on filtering (artificial viscosity) for chebyshev and legendre spectral and spectral element methods: preserving boundary conditions and interpretation of the filter as a diffusion. *Journal of Computational Physics*, 143(1):283–288, 1998.

[2] M. O. Deville, P. F. Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge, 2002.

[3] Paul Fischer and Julia Mullen. Filter-based stabilization of spectral element methods. *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics*, 332(3):265–270, 2001.

[4] Krestin Jordaan. Lecture notes in properties of orthogonal polynomials, 2017.

[5] Johan Malm, Philipp Schlatter, Paul F Fischer, and Dan S Henningson. De-aliasing on non-uniform grids: algorithms and applications. *Journal of Computational Physics*, 191:249–264, 2003.

[6] Johan Malm, Philipp Schlatter, Paul F Fischer, and Dan S Henningson. Stabilization of the spectral element method in convection dominated flows by recovery of skew-symmetry. *Journal of Scientific Computing*, 57(2):254–277, 2013.

[7] Prabal Negi, Philipp Schlatter, and Dan Henningson. A re-examination of filter-based stabilization for spectral-element methods, 2017.

[8] Saleh Rezaeiravesh, Ricardo Vinuesa, and Philipp Schlatter. On numerical uncertainties in scale-resolving simulations of canonical wall turbulence. *Computers Fluids*, 227:105024, 2021.

[9] Stolz S. Kleiser Schlatter, P. Les of transitionalflows using the approximate deconvolution model. *International Journal of Heat and Fluid Flow*, 25(3):549–558, 2003.

Group 10: Work Balancing in Nek5000

Ronith Stanly [*] and Fermin Mallor [†]

June 24, 2021

1 Parallel computing

1.1 Motivation

Consider one has a serial code that can be run on one processor in a normal laptop. Now if one has to either make the code run faster, or if one has to solve a much larger problem in the same amount of time (or in fact, have a combination of both), the ideal solution is to have a much larger, single processor (and hence which has shared memory), so that the same serial code can be run faster for the bigger problem size. However, building such a machine that scales to such extremes is technologically and economically unfeasible.

Hence, the solution is to connect several of the individual processors (that we initially have) using a network and interconnects, to make a big machine. However, a new problem arises here: since the new, big machine is made as a combination of smaller processors with individual memory (and hence the big machine is a *distributed memory system*), here one has to ensure a way to communicate between processors by sending and receiving messages across the network. This is how parallel computing is done.

1.2 Issues

As we make bigger and bigger machines, we need to do more communications between processors to exchange information and to perform the parallel simulation. Hence one has to make sure that the system scales well for a fixed problem size (strong scaling), as well as for a scaled problem (weak scaling); and to ensure that the communication overhead does not become the bottleneck of the simulation.

One factor to consider in this context is [1] *Amdhal's law*, which states that the speed-up of a code is limited by the percentage of serial section of the code. For instance, if a serial code takes 20 hours to run on one processor and it has one part of the code which takes 1 hour to run which cannot be parallelized (*serial*

[*] KTH Engineering Mechanics, ronith@kth.se
[†] KTH Engineering Mechanics, mallor@kth.se
[1] https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

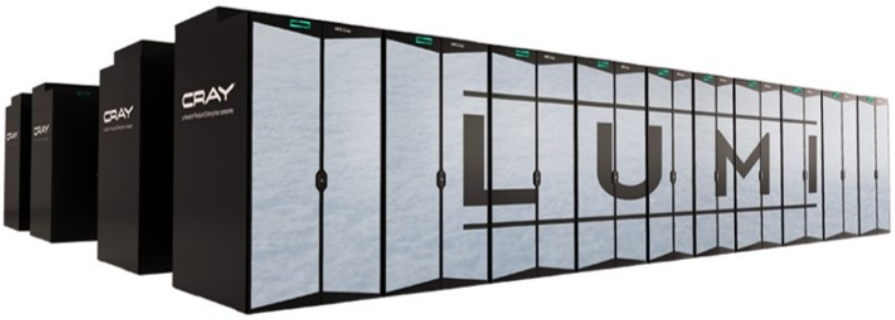


Figure 1: Lumi Supercomputer

section) and even if the code that takes the remaining 19 hours of execution time can be parallelized, then regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical one hour. Hence, the theoretical speedup is limited to at most 20 times (when N = ∞, speedup = 1/s = 20). As such, the parallelization efficiency decreases as the amount of resources increases. For this reason, parallel computing with many processors is useful only for highly parallelized programs.

Hence to utilize large parallel systems efficiently, the best solution is to ensure the code has good strong scaling. Another way is to increase the problem size with increasing number of processors (weak scaling).

1.3 Benefits

There are several benefits of having an efficiently parallelized code to run on a parallel computer. The obvious one is that it allows one to have a much reduced time to solution. Some other advantages are that it enables one to have highly resolved numerical simulations by permitting to have a large number of grid points. This allows us to understand more physical insights from a simulation. It also enables us to solve more detailed mathematical models, rather than having simplified ones.

2 Graph Partitioning

In order to run a simulation on multiple processors, we have to efficiently map the unstructured grid onto different processors (as shown in Fig, 3) in such a way that each processor gets an almost equal amount of work while minimizing the amount of communication between them. Such a map is found by solving a graph partitioning problem.

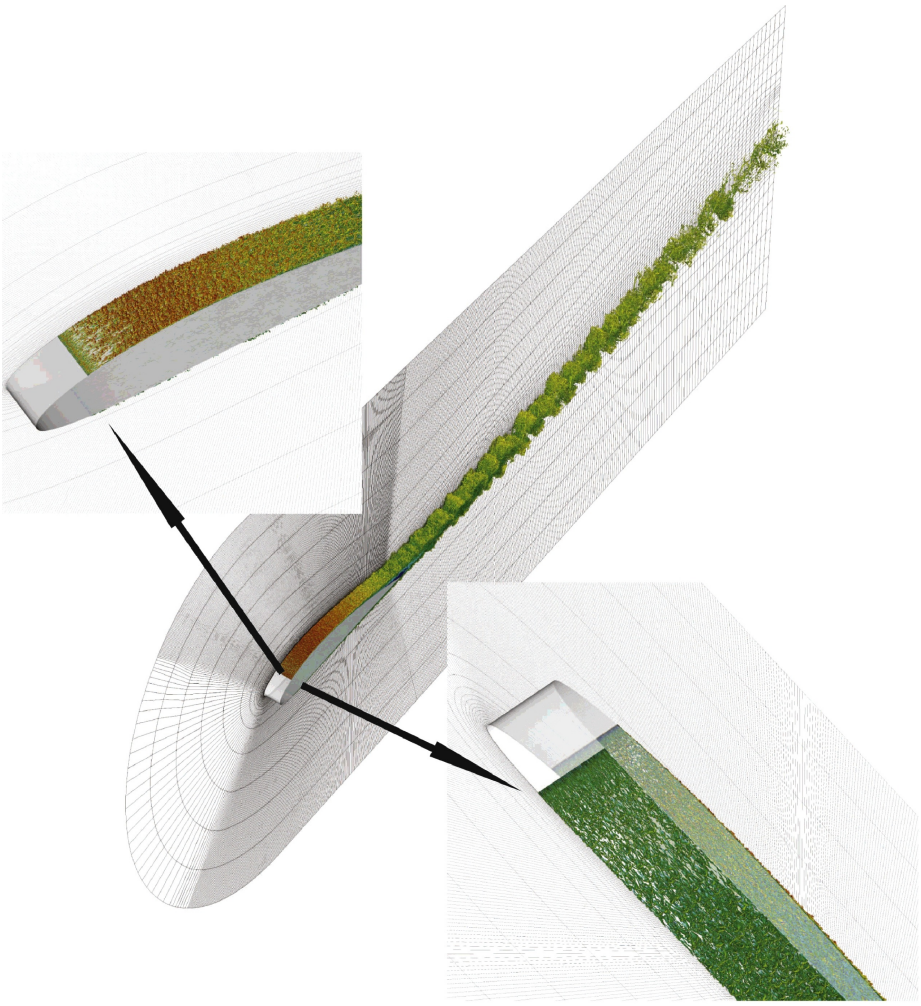


Figure 2: Highly resolved simulation of an airfoil from [2]

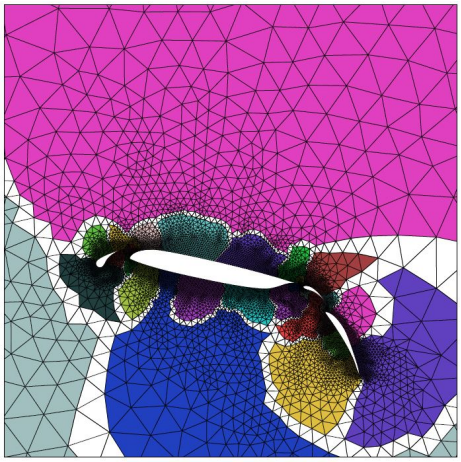


Figure 3: An unstructured grid around an airfoil mapped onto different processors as shown by the different colours [4]

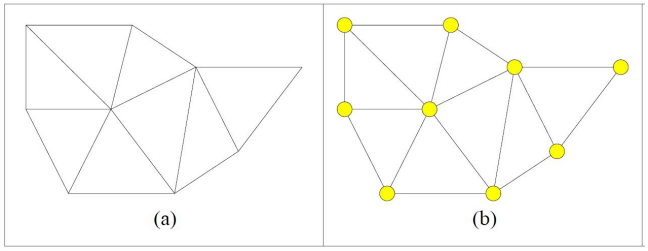


Figure 4: (a) A 2D unstructured grid (b) The graph of the grid showing vertices in yellow and its associated edges (representing communication between vertices) [4]

Before we go deeper into the graph partitioning problem, it is important to understand what a graph is and to know some terms related to it. In order to compute a mapping of a mesh onto a set of processors via graph partitioning, it is first necessary to construct the *graph* that models the structure of the computation, as shown in Fig. 4. In general, computation of a scientific simulation can be performed on the mesh nodes, the mesh elements, or both of these. If the computation is mainly performed on the mesh nodes then this graph is straightforward to construct. A *vertex* (V) exists for each mesh node, and an *edge* (E) exists on the graph for each edge between the nodes. If the partitioning is computed such that each subdomain has the same number of vertices, then each processor will have an equal amount of work during parallel processing. The total volume of communications incurred during this parallel processing can be estimated by counting the number of edges that connect vertices in different subdomains. Therefore, a partitioning should be computed that minimizes this metric (which is referred to as the *edge-cut*).

For complicated problems involving multi-physics, adaptive mesh refinement

and so on, it is possible to define weights on edges and vertices depending on the local complexity. For instance, in a combustion simulation, if a particular species is formed at a certain part of the grid and more reactions start happening there, the vertices and edges can be given higher weights there in order to partition the graph more efficiently.

Given a weighted, undirected graph $G = (V; E)$ for which each vertex and edge has an associated weight, the *k-way graph partitioning* problem is to split the vertices of V into k (for k processors) disjoint subsets (or subdomains) such that each subdomain has roughly an equal amount of vertex weight (balance constraint), while minimizing the sum of the weights of the edges whose incident vertices belong to different subdomains [4].

2.1 Types of Graph Partitioning

The graph partitioning problem is an NP-complete problem, which means that it is difficult to compute an optimal partition in a reasonable amount of time. However, different approximation methods exit. Some of them are:

1. Geometric techniques
2. Combinatorial techniques
   (a) KL/FM Algorithm
3. Multilevel schemes
   (a) Multi-level k-way Partitioning (parMETIS)
4. Spectral methods
   (a) Recursive Spectral Bisection (parRSB)
5. Combined schemes

Of the methods mentioned above, the ones coloured in blue are available in Nek5000 and will be mentioned in detail in subsequent sections.

3 Partitioning schemes in Nek5000

The multi-level k-way partitioning method is available in Nek5000 through a third-party library called parMETIS, whereas the Recursive Spectral Bisection method is present within a module called parRSB.

3.1 ParMETIS

The multi-level k-way partitioning method present in Nek5000 is a combination of a multi-level scheme and a modified KL/FM algorithm. Hence the multi-level k-way partitioning method will be explained by first going through what a multi-level scheme is, followed by describing the KL/FM algorithm and then specifying what modifications are made to the KL/FM algorithm to obtain the multi-level k-way partitioning method present in Nek5000 (through parMETIS).

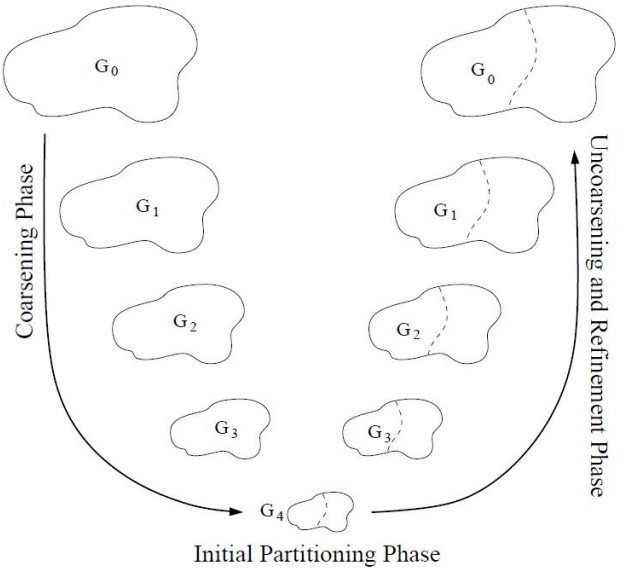


Figure 5: The three phases of the multilevel graph partitioning paradigm. During the coarsening phase, the size of the graph is successively decreased by bundling together nodes and edges to have smaller number of them. During the initial partitioning phase, a bisection is computed, During the uncoarsening and refinement phase, the bisection is successively refined as it is projected to the larger graphs. G_0 is the input graph, which is the finest graph. G_{i+1} is the next level coarser graph of G_i. G_4 is the coarsest graph. [4]

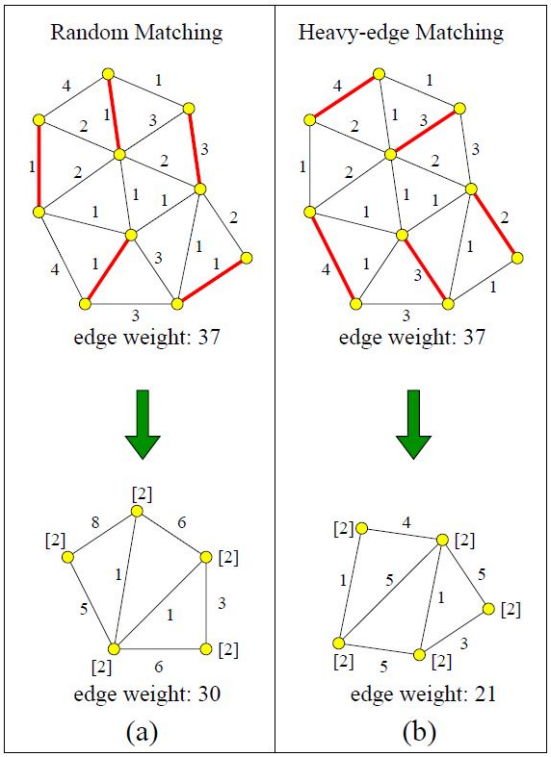


Figure 6: A random matching of a graph along with the coarsened graph in (a). The same graph is matched (and coarsened) with the heavy-edge heuristic in (b). The heavy-edge matching minimizes the exposed edge weight. [4]

3.1.1 Multi-level Scheme

The multi-level scheme of graph partitioning consists of three levels, as shown in Fig.5:

1. **Graph Coarsening**
   A series of graphs are constructed by collapsing together selected vertices of the input graph in order to form a related coarser graph. There are several methods to do this, random matching and heavy-edge matching, as shown in Fig.6, are two of them. Random matching collapses together a random set of edges, no two of which (edges) are incident on the same vertex. The coarsening in Fig. 6 can be understood if you eliminate the red edges and stack the other edges and nodes. Heavy-edge matching selects edges with higher weights.

2. **Initial Partitioning**
   Computation of the initial bisection is performed on the coarsest (and hence smallest or the one with least number of edges and vertices) of these graphs, and so is very fast.

3. **Multi-level refinement**
   Partition refinement is performed on each level of the graph, from the

coarsest to the finest (i.e., original graph) using a KL/FM-type algorithm, which is described in the next section.

3.1.2 KL/FM Algorithm

Closely related to the graph partitioning problem is that of partition refinement. Given a graph with a sub-optimal partitioning, the problem is to improve the partition quality while maintaining the balance constraint. Essentially, this differs from the graph partitioning problem only in that it requires an initial partitioning of the graph. The Kernighan-Lin / Fiduccia-Mattheyses (KL/FM) algorithm refines the initial bisection. The KL/FM algorithm consists of a small number of passes through the vertices. During each pass, the algorithm moves one vertex at a time between two sub-domains. To track the order in which vertices are moved, two priority queues (one for each domain) are created. Prior to each pass, the gain of every vertex is computed (i.e., the amount by which the edge-cut will decrease if the vertex changes subdomains). Then it is placed in the queue and ordered based on gain. If the top vertex (i.e., the one with the highest gain) in only one of the priority queues is able to switch subdomains while still maintaining the balance constraint, then that vertex is moved to the other subdomain. If the top vertices of both of the priority queues can be moved while maintaining the balance, then the vertex that has the highest gain among these is moved Ties are broken by selecting the vertex that will most improve the balance. When a vertex is moved, it is removed from the priority queue and the gains of its adjacent vertices are updated. The pass ends when neither priority queue has a vertex that can be moved. At this point, the highest quality bisection that was found during the pass is restored.

The algorithm explained above can be illustrated using an example as shown in figures 7 and 8: KL/FM-type algorithms are able to escape from some types of local minima because they explore moves that temporarily increase the edge-cut. Figure 7 illustrates this process. Figure 7(a) shows a bisection of a graph with an edge-cut of six. Here, the weights of the vertices and edges are one. There are twenty vertices in the graph. Therefore, a perfectly balanced bisection will have subdomain weights of ten. However, in this case we allow the subdomains to be up to 10% imbalanced. Therefore, subdomains of weight eleven are acceptable. Figure 7(b) shows the gain of each vertex. Since all of the gains are negative, moving any vertex will result in the edge-cut increasing. Therefore, the bisection is in a local minima. However, the algorithm will still select one of the vertices with the highest gain and move it. The white vertex is selected. Figure 7(c) shows the new bisection as well as the updated vertex gains. There are now two positive gain vertices. However, neither of these can be moved at this time. The black vertex has just moved, and so it is ineligible to move again until the end of the pass. The other vertex with +1 gain is unable to move as this will violate the balance constraint. Instead, one of the highest negative-gain vertices (shown in white) from the left subdomain is selected. Figure 7(d) shows the results of this move. Now

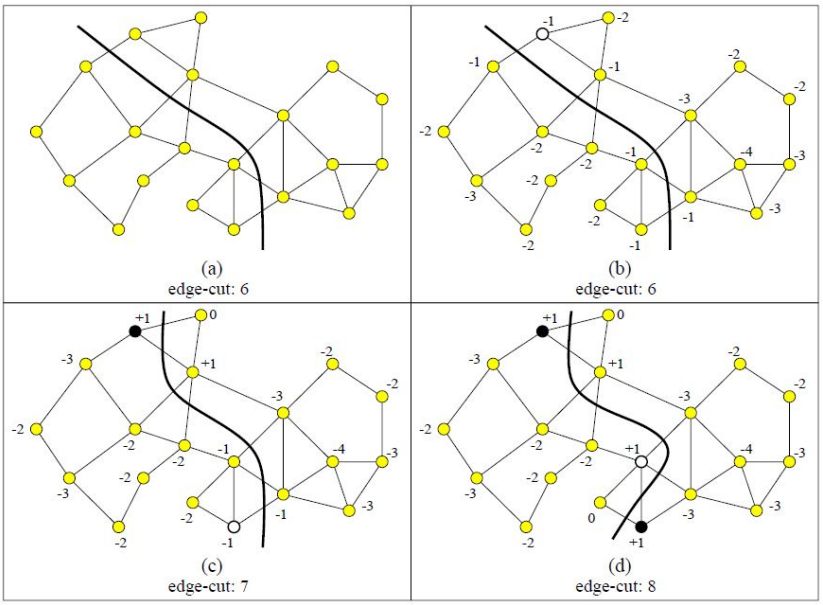


Figure 7: A bisection of a graph refined by a KL/FM algorithm. The white vertices indicate those selected to be moved. In (a) the partitioning is in a local minima. In (b) the algorithm explores moves that increase the edge-cut. In (c) and (d) the edge-cut is increased, but now there are edge-cut reducing moves to be made [4]

there are two positive gain vertices that are able to move and two that are ineligible to move. The white vertex is selected. Figure 8 shows the results of continued refinement. By Figure 8(d), the bisection has reached another minima with an edge-cut of two. The refinement algorithm has succeeded in climbing out of the original local minima and reducing the edge-cut from six to two [4].

3.1.3 Multi-level k-way partitioning (parMETIS)

As mentioned before, the multi-level k-way partitioning method available in Nek5000 through parMETIS is a combination of a multilevel scheme and a modification of the KL/FM algorithm. The modification of the KL/FM algorithm that is done is that the graph is first coarsened down to a small number of vertices, a k-way partitioning of this much smaller graph is computed, as shown in Fig. 9, (instead of the bisection which was done in the basic KL/FM method), and then this partitioning is projected back toward the original graph (finer graph) by successively refining the partitioning at each intermediate level. For refinement, there are k(k-1) queues to consider instead of 2. Hence a variation of KL/FM algorithm is used for refinement, where only the boundary vertices are visited and moved based on reducing edge-cuts and maintaining balance constraint [3].

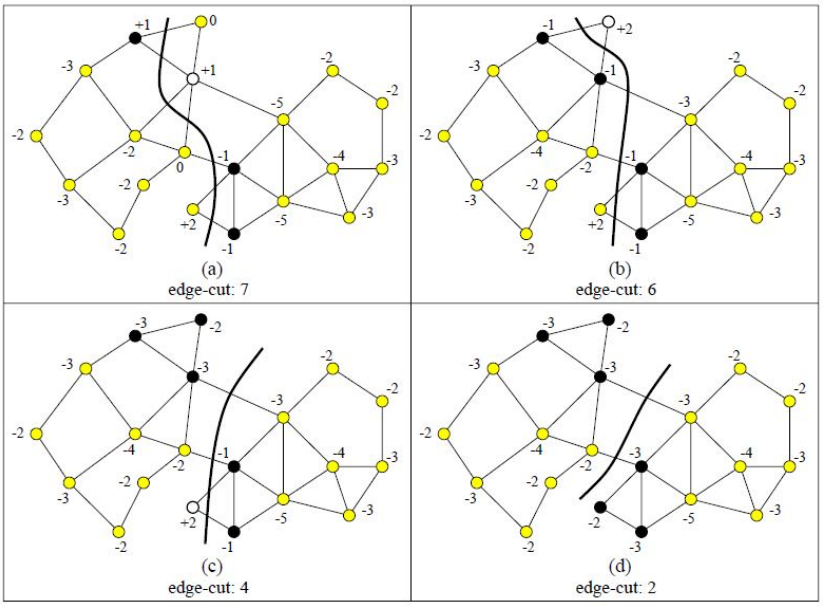


Figure 8: The KL/FM algorithm from Figure 7 is continued here. Edge-cut reducing moves are shown from (a) through (d). By (d), the refinement algorithm has reached a local minima [4]

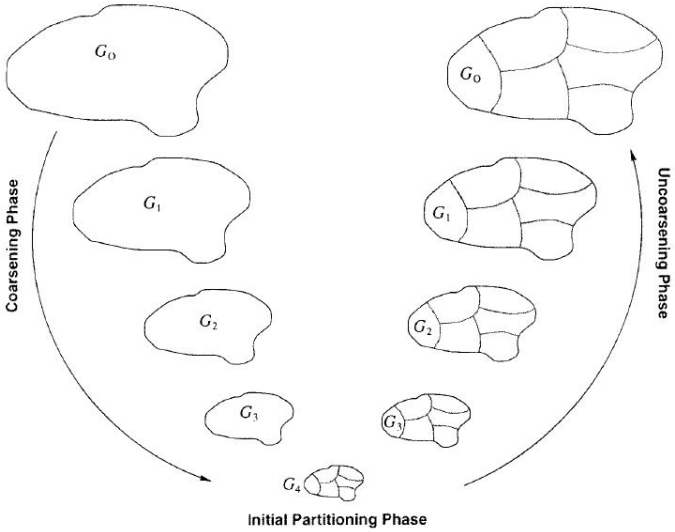


Figure 9: The various phases of the multilevel k-way partitioning algorithm. During the coarsening phase, the size of the graph is successively decreased; during the initial partitioning phase, a k-way partitioning of the smaller graph is computed (a 6-way partitioning in this example); and during the uncoarsening phase, the partitioning is successively refined as it is projected to the larger graphs [3]

3.2 ParRSB

Unlike *ParMETIS*, *ParRSB* uses no geometric information for the mesh partitioning. *ParRSB* is based on an algorithm known as recursive spectral bisection, in which a purely mathematical representation of the graph (mesh) is used for the partitioning process, which is formulated as a (relaxed) optimization of a quadratic function. In the following lines, the theory behind spectral partitioning will be explained (following a similar explanation to that given by Elsner in chapter 3.3 of reference [1]).

Consider the partition of a graph $G = (V; E)$ into two subdomains V_1 and V_2. One can distinguish the vertices belonging to each of them by using a discrete approach:

$$x_i = \begin{cases} -1, & \text{if } v_i \in V_2 \\ 1, & \text{if } v_i \in V_1 \end{cases} \tag{1}$$

The edge cut number (which serves as a proxy for the communication load) can be computed as:

$$f(x) = \frac{1}{4} \sum_{(i,j)\in E} (x_i - x_j)^2 = \frac{1}{4} \Bigg(\sum_{(i,j)\in E} (x_i + x_j)^2 - \sum_{(i,j)\in E} 2x_i x_j \Bigg) \tag{2}$$

After the decomposition, both sums can be expressed as a diagonal and an off-diagonal matrix, respectively:

$$\sum_{(i,j)\in E} (x_i + x_j)^2 = x^T D x; \quad - \sum_{(i,j)\in E} 2x_i x_j = x^T A x \tag{3}$$

Here, the diagonal matrix D is known as the degree matrix, and it models the degree of connectivity of each vertex, and the off-diagonal matrix A is the adjacency matrix, modelling the connections between vertices (edges). Essentially, the mesh is represented as a lumped mass system, in which the vertices correspond to the masses and the edges to the couplings (similar to a mass-spring system). Both matrices can be combined back into what is known as the Laplacian of the graph, $L = D - A$. The graph partitioning problem can be formulated as:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{4} x^T L x \\ \text{subject to} \quad & x \in \pm 1, \\ & x^T e = 0. \end{aligned} \tag{4}$$

where e is the unit vector.
However, this is a NP-complete problem. In order to circumvent this issue, the problem is relaxed by replacing the discrete variable x by a real vector z with the same Euclidean length as x. By doing so, one can find a lower bound for the original minimization problem.

$$\begin{aligned} \text{minimize} \quad & \frac{1}{4} z^T L z \\ \text{subject to} \quad & z^T z = n, \\ & z^T e = 0. \end{aligned} \tag{5}$$

where n is the number of elements.

The Laplacian matrix is a positive, semidefinite matrix, which means that it has real-valued, non-negative eigenvalues (λ_i) with real-valued, orthogonal eigenvectors (q_i). Following the Perron-Frobenius theorem, the second eigenvalue will be non-zero if the graph is connected. This eigenvalue (λ_2) is also known as the algebraic connectivity and q_2, the *Fiedler* vector. Bisecting the median of q_2 minimizes the cut edges while balancing the load.

Algorithm 1 The recursive spectral bisection

```
1: procedure BISECT(G(V, E))
2:     for e = 1, 2 ..., log2(n_el) do
3:         Compute Laplacian of graph
4:         Compute λ2 and q2 using Lanczos algorithm
5:         compute median (m_p) of q2
6:         Choose V1 := {v_i ∈ V | q_i < m_q}, V2 := {v_i ∈ V | q_i > m_q}
7:     end for
8: end procedure
```

4 Implementation in Nek5000

In Nek5000, both partitioning methods (*ParMETIS* and *ParRSB*) are included as external libraries, and as such they can be found under the */3rd_party* directory of the main code. In order to select the partitioner before runtime, a flag (either *METIS* or *RSB*) is used during compiling. Moreover, the partitioner chosen is set in the *.par* file under the [MESH] section. The partitioner will be called during initialization or during run-time if adaptive mesh refinement (AMR) is used.

Inside the */core* folder, the *partitioner.c* routine takes care of the calls to the partitioner libraries. This routine is called from the *map2.f* at initialization (or from *amr_part.f* when AMR is performed). The main functions inside *partitioner.c* are the following:

- $fpartMesh$: Calls either *parMETIS* or *parRSB* in order to perform the load-balancing.
- $redistributeData$: redistributes the elements to the correct processors after partitioning is finished.
- $printPartStat$: prints load-balancing results and stats to the log file.

After the partitioner redistributes the element data to each processor, each processor can access locally its own elements through the *lglel* array. Globally, the *gllel* array assigns a global element to its local correspondent. and *gllnid* assigns a global element to its processor.

5 Conclusions

As computational power increases and machines become larger, load-balancing becomes a critical issue for parallel computing. Correct load-balancing allows to

reduce serial bottlenecks in scientific codes, improving greatly their efficiency. In order to perform load-balancing in scientific computation, the domain (mesh) is subdivided into N processors using different partitioning algorithms (geometric, spectral, multi-level,...).

In Nek5000, two different partitioners (included as 3rd party libraries) are available for load balancing: *parMETIS* and *parRSB*. The former is based on the multi-level k-way partitioning algorithm, and the later on the recursive spectral bisection algorithm. Overall, the main differences come from the fact that the k-way method uses geometric information (and allows for weighting of the edges and nodes), whereas the spectral method is based on a purely mathematical representation of the graph. Overall, both methods give good-quality partitions, but for homogeneous, well connected problems, as those in Nek5000, spectral bisection (*RSB*) creates higher-quality and better-connected partitions.

References

[1] Ulrich Elsner. Graph partitioning - a survey. *Technische Universität Chemnitz*, 1997.

[2] S.M. Hosseini, R. Vinuesa, P. Schlatter, A. Hanifi, and D.S. Henningson. Direct numerical simulation of the flow around a wing section at moderate reynolds number. *International Journal of Heat and Fluid Flow*, 61:117–128, 2016. SI TSFP9 special issue.

[3] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[4] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. 2000.

Nek5000 Group 11

Harrison Nobis*, Jose Faúndez† and Thales Fava‡

June 24, 2021

1 Introduction

In some of the derivations used in this chapter we will make use of the elliptic problem that reads

$$-\nabla \cdot [p(\mathbf{x})\nabla u(\mathbf{x})] + q(\mathbf{x})u(\mathbf{x}) = f(\mathbf{x}) \quad \text{in } \Omega. \tag{1}$$

The weak formulation of (1) is given by: Find $u \in V_0$ such that

$$\mathcal{A}(u,v) = \mathcal{F}(v) \quad \forall v \in V_0, \tag{2}$$

where

$$V_0 := \{v \in H^1(\Omega); v = 0 \text{ on } \partial\Omega_D\}, \tag{3}$$

$$\mathcal{A}(u,v) := \int_\Omega [p(\mathbf{x})\nabla u \cdot \nabla v + q(\mathbf{x})uv]\,\mathrm{d}\mathbf{x}, \tag{4}$$

$$\mathcal{F}(u,v) := \int_\Omega f(\mathbf{x})v(\mathbf{x})\mathrm{d}\mathbf{x}. \tag{5}$$

2 Boundary Conditions

2.1 Homogeneous essential boundary conditions

To begin our discussion of boundary conditions we refer to the elliptic problem (1) with $q = 0$, subject to homogeneous Dirichlet and Neumann boundary conditions

$$u = 0 \text{ on } \partial\Omega_D, \qquad \nabla u \cdot \mathbf{n} = 0 \text{ on } \partial\Omega_N.$$

Multiplying equation (1) by the test function $v \in H^1(\Omega)$, integrating over Ω and integrating by parts yields

$$\int_\Omega p\nabla v \cdot \nabla u\, dV - \int_{\partial\Omega} vp\nabla u \cdot \mathbf{n}\, dS = \int_\Omega vf\, dV. \tag{6}$$

*KTH Engineering Mechanics, nobis@mech.kth.se
†KTH Engineering Mechanics, josfa@mech.kth.se
‡KTH Engineering Mechanics, fava@mech.kth.se

Here we notice the additional boundary term arising from integration by parts and make two new definitions for classes of boundary conditions, which cancel these boundary terms in different ways:

1. Essential BCs: Referring to boundary conditions which are imposed explicitly, and are an essential requirement to obtain a unique solution. Here they are satisfied by restricting the elements of the admissible functions. Since u is absent in the boundary term of equation (6), it is the Dirichlet conditions which are treated in this way, resulting in u and v being restricted to $V_0 = \{v \in H^1(\Omega); v = 0 \text{ on } \partial\Omega_D\}$.

2. Natural BCs: Referring to the boundaries which are incorporated implicitly and are assumed to cancel the boundary term, which is why they are commonly referred to as *do nothing* boundary conditions. These boundary conditions are automatically satisfied in the limit when the approximate solution converges to the real solution. It can be seen that the Neumann condition falls into this category, however it should be made clear, that it is the integral over the boundary which is assumed to be zero, $\int_{\partial\Omega_N} vp\nabla u \cdot \mathbf{n}\, dS = 0$, and that this does not necessarily imply that $\nabla u \cdot \mathbf{n} = 0$ is satisfied point wise on the boundary Ω_N, which is why they are also commonly referred to as *weak* boundary conditions.

Finally, the weak formulation of equation (1) after integrating by parts reads: Find $u \in V_0$

$$\int_\Omega p\nabla v \cdot \nabla u\, dV = \int_\Omega v f\, dV, \quad \forall v \in V_0, \tag{7}$$

or

$$\mathcal{A}(u, v) = \mathcal{F}(v) \quad \forall v \in V_0. \tag{8}$$

Note here that the boundary term has cancelled on $\partial\Omega_D$ through the restriction of V_0, and has been cancelled to implicitly enforce the Neumann condition on $\partial\Omega_N$.

2.2 Discretized homogeneous essential boundary conditions

Moving attention now to a discretized setting, we have that $\underline{u} \in V_N$ and $\underline{v} \in V_N$ now refer to vectors of basis coefficients and $\bar{H}$ and M are the discrete Helmholtz operator and discrete mass matrix respectively in the homogeneous Neumann problem. If u and v are to satisfy the homogeneous essential boundary conditions, $u(\mathbf{x}) = v(\mathbf{x}) = 0$ on $\partial\Omega_D$, this condition must be incorporated before the weighted residual problem (8). To do so implies restricting the index range $\hat{\imath}$ and i of u_i and v_i, such that we exclude the index corresponding to essential boundary conditions.

Here we denote the restricted coefficients as $\tilde{\underline{u}} = R\underline{u}$, which in turn implies $\underline{u} = R^T\tilde{\underline{u}}$, where R is $\mathcal{N} \times (N+1)^d$ restriction which can be thought of as an identity with columns of zeros scattered throughout in locations corresponding to nodal points on $\partial\Omega_D$ (note: $\mathcal{N} \leq (N+1)^d$ refers to the number of points in $\Omega \setminus \partial\Omega_D$) One could express R in higher dimensions with tensor products, $R = R_1 \otimes \cdots \otimes R_d$ however in practice, R is never actually formed, as it is only

the action of R required.

Applying the above restrictions to (8), yields the linear system:

$$\begin{aligned} H\underline{\tilde{u}} &= RM\underline{f}, \\ \underline{u} &= R^T\underline{\tilde{u}}, \end{aligned} \tag{9}$$

with

$$H := R\bar{H}R^T.$$

A final remark can be made that H is symmetric positive definite and can hence be solved with PCG.

2.3 Inhomogeneous boundary conditions

Inhomogeneous essential boundary conditions can be handled by splitting the solution $u := u_h + u_b$, into two components, where u_h satisfies homogeneous boundary conditions and and u_b is any function in V_N. Applying this split to equation (9) yields

$$\begin{aligned} H\underline{\tilde{u}_h} &= RM\underline{f} - R\bar{H}\underline{\tilde{u}_b}, \\ \underline{u} &= R^T\underline{\tilde{u}_h} + \underline{\tilde{u}_b}. \end{aligned} \tag{10}$$

Before addressing Inhomogeneous Neumann conditions it is useful to examine the mixed Dirichlet/Neumann condition known as the Robin condition

$$\nabla u \cdot \mathbf{n} + \alpha u = \beta \text{ on } \partial\Omega_R. \tag{11}$$

Returning to the weak form described by equation (6) the boundary term can now be treated as

$$\begin{aligned} &\int_\Omega p\nabla v \cdot \nabla u \, dV - \int_{\partial\Omega} vp\nabla u \cdot \mathbf{n} \, dS = \int_\Omega vf \, dV, \\ \Rightarrow &\int_\Omega p\nabla v \cdot \nabla u \, dV - \int_{\partial\Omega_R} vp(\beta - \alpha u) \, dS. = \int_\Omega vf \, dV, \\ \Rightarrow &\int_\Omega p\nabla v \cdot \nabla u \, dV + \int_{\partial\Omega_R} vp\alpha u \, dS = \int_\Omega vf \, dV + \int_{\partial\Omega_R} vp\beta \, dS, \end{aligned} \tag{12}$$

Where the boundaries on $\partial\Omega_D$ and $\partial\Omega_N$ have cancelled for the same reasons as section 2.1 and equation (11) has been substituted on $\partial\Omega_R$. The treatment of such a boundary condition can be performed in a discrete setting by following a similar procedure to (10), now with a slight modification to $\bar{H}$. Finally it can be seen that inhomogeneous Neumann conditions follow from (12) by taking $\alpha = 0$.

2.4 Spectral-Element Operators

When referring to the spectral element discretization involving multiple elements it is more illustrative to replace the restriction matrix R with either the mask $\mathcal{M}$ or $\mathcal{M}_L$; where the former is applied globally and the latter is applied locally, effectively restricting $u \in V_N$ to $V_{N,0} := V_N \cap H_0^1$. For example, the following two

equations demonstrate how the stiffness matrix can be masked either globally or locally. However this procedure applies equally to any other matrix or operator.

$$\begin{aligned} K &= \mathcal{M}Q^T K_L Q \mathcal{M}, \\ &\text{or} \\ K &= Q^T \mathcal{M}_L K_L \mathcal{M}_L Q. \end{aligned} \tag{13}$$

It should be noted that in section 2.2, the restriction operation R reduces the size of the operators, and hence the size of the resulting linear system (9). By employing masks, this is not the case, and in fact strictly speaking the masked form of K is no longer invertable. However, if solved with iterative techniques this is not a problem so long as the underlying basis vectors on $\partial\Omega_D$ are always zero.

Returning to the elliptic problem formulated in equation (10), with inhomogeneous boundary conditions, if we use the global mask $\mathcal{M}$, we arrive at

$$\begin{aligned} \mathcal{M}Q^T H_L Q \mathcal{M}\underline{u}_0 &= \mathcal{M}Q^T(M_L \underline{f}_L - H_L \underline{u}_{b,l}), \\ H_L &:= K_L + qM_L. \end{aligned} \tag{14}$$

Here we note that u_b must be continuous, however any projection such as $\underline{u}_b := M^{-1}\Sigma' M_L \underline{u}_{b,L}$ can be used to ensure this. Observing the form of equation (14) it can be seen that $H = \mathcal{M}Q^T H_L Q \mathcal{M}$ is still symmetric. In practice, it is better to a formulation utilizing the local mask $\mathcal{M}_L$, in which case the resulting system is of the form

$$\mathcal{M}_L \Sigma' H_L \underline{u}_{0,L} = \mathcal{M}_L \Sigma'(M_L \underline{f}_L - H_L \underline{u}_{b,L}). \tag{15}$$

This form is advantageous because tensor product operations can be performed without additional scatter operations. Furthermore, using Σ' reduces communication overhead.

2.5 Natural outflow boundary conditions

To conclude the discussion of boundary conditions we move away from the simplified elliptic problem discussed in sections 2.1 - 2.4 and address the incompressible Navier stokes equations

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} &= -\nabla p + \frac{1}{Re}\nabla^2\mathbf{u} + \mathbf{f}, \\ \nabla\cdot\mathbf{u} &= 0. \end{aligned} \tag{16}$$

Following a similar procedure to section 2.1, multiplying by a test function $v \in H^1(\Omega)^d$, integrating in Ω, and integrating the viscous and pressure terms by parts one can arrive at

$$\begin{aligned} &\int_\Omega \mathbf{v}\cdot\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} - \mathbf{f}\right) dV = \\ &\int_\Omega (\nabla\cdot\mathbf{v}) p \, dV - \frac{1}{Re}\int_\Omega \nabla\mathbf{v}\cdot\nabla\mathbf{u}\, dV + \oint_{\partial\Omega} \mathbf{v}\cdot\left(-p + \frac{1}{Re}\nabla\mathbf{u}\right)\cdot\mathbf{n}\, dS. \end{aligned} \tag{17}$$

It can be seen from the additional boundary term that the natural outflow boundary condition is given by

$$-p\mathbf{n} + \frac{1}{Re}(\mathbf{n}\cdot\nabla\mathbf{u}) = 0 \text{ on } \Gamma_O. \tag{18}$$

This outflow condition is subtly problematic for two reasons. First, while this is formally one condition, we find that in practice it effectively sets $p = 0$ and $\frac{\partial \mathbf{u}}{\partial n} = 0$, implying the mean pressure at each outlet is zero. While this is not problematic for just one outflow, since the pressure can always be shifted by a scalar, it can yeild non-physical solutions for multiple outflows, particularly in non-symmetric domains such as that in Fig. 1.

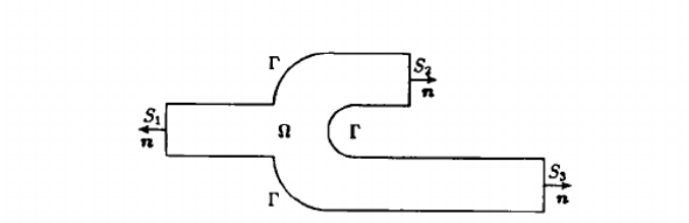


Figure 1: Scematic or a double pipe[7].

A second consideration is the growth of kinetic energy into the domain from the outflow boundary condition (18). By multiplying equation (16) by u, and integrating over Ω, an energy estimate can be obtained (see Bostrom et al [2] for a full derivation)

$$\frac{1}{2}\frac{d}{dt}||u||^2_{L^2(\Omega)} = -\frac{1}{Re}||\nabla u||^2_{L^2(\Omega)} + \oint_{\partial\Omega} u\cdot\left(-p + \frac{1}{Re}\nabla u - \frac{1}{2}|u|^2\right)\cdot\mathbf{n}. \tag{19}$$

For high Reynolds numbers it can be seen that the energy decay is dependent entirely on the boundary terms. In the case of Dirichlet boundaries the total energy change is controlled, however for outflow conditions the following stability constraint applies

$$\begin{aligned}\mathbf{u}\cdot\left(-p + \frac{1}{Re}\nabla\mathbf{u} - \frac{1}{2}|\mathbf{u}|^2\right)\cdot\mathbf{n} \le 0 \text{ on } \partial\Omega_O,\\ -\frac{1}{2}|\mathbf{u}|^2(\mathbf{u}\cdot\mathbf{n}) \le 0 \text{ on } \partial\Omega_O,\end{aligned} \tag{20}$$

where the first two terms have been cancelled from the natural boundary condition. Crucially, this implies

$$Re\to\infty \implies \begin{cases}\frac{1}{2}\frac{d}{dt}||u||^2_{L^2(\Omega)} \le 0 & \mathbf{u}\cdot\mathbf{n}|_{\partial\Omega_O} \ge 0,\\ \frac{1}{2}\frac{d}{dt}||u||^2_{L^2(\Omega)} < 0 & \mathbf{u}\cdot\mathbf{n}|_{\partial\Omega_O} < 0.\end{cases} \tag{21}$$

There exists several methods to overcome the stability issues of the outflow condition. Perhaps the simplest is the inclusion of a sponge region. Dong et al. [4], address this by adding the condition $u\cdot\left(-p + \frac{1}{Re}\nabla u - \frac{1}{2}|u|^2\right)\cdot\mathbf{n} = 0$ if there is an energy influx on $\partial\Omega_O$ (see `examples/turbJet`).

Furthermore, convective outflow conditions exist which replace the outflow boundary condition with a Dirichlet condition which attempts to predict the value of the velocity in the outflow region. This requires making an assumption about the convection speed, the choice of which is not trivial, especially for flows containing vorticies. (See Bostrom et al [2] for further details)

3 Mesh

3.1 Deformed Geometries

In 3D cases, around 90% of the operations come from a matrix-matrix product, so the implementation of this operation have to be highly optimized. The efficiency of `Nek5000`, and of high-order methods in general, comes from the choice of computationally convenient basis functions, basis coefficients that are also grid points allowing a diagonal mass matrix and low-cost operator, and especially the matrix-free fast tensor-product which reduces the work from $\mathcal{O}(nN^3)$ to $\mathcal{O}(nN)$. However, care must be taken when dealing with deformed geometries, where some extra computations have to be made and some further approximations are employed with respect to its parallelepipeds counterpart.

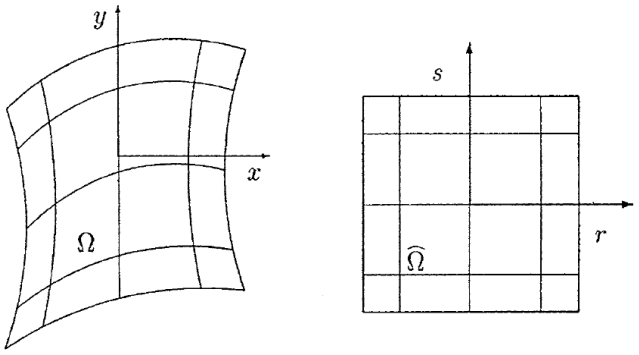


Figure 2: Example of deformed domain Ω and its transformation to reference domain $\hat{\Omega}$. Figure from [3].

In this section we consider the general case where the domain Ω is deformed, as the one shown in Fig. 2, treating rectangular parallelepiped geometries, with arbitrary orientation, as a particular case. A common practice in SEM is to represent the geometry in terms of the same polynomial basis used for the solution, that is for each element Ω we have the isoparametric map

$$\mathbf{x}(r,s,t) = \sum_{i=0}^{N}\sum_{j=0}^{N}\sum_{k=0}^{N} \mathbf{x}_{ijk}\pi_i(r)\pi_j(s)\pi_k(t), \tag{22}$$

and therefore it exists an invertible map $\mathbf{x}(\mathbf{r})$ from the physical deformed domain to the reference domain $\hat{\Omega}(\mathbf{r})$ with the Jacobian

$$J(\mathbf{r}) = \det \begin{pmatrix} \frac{\partial x_1}{\partial r_1} & \cdots & \frac{\partial x_1}{\partial r_d} \\ \vdots & & \vdots \\ \frac{\partial x_d}{\partial r_1} & \cdots & \frac{\partial x_d}{\partial r_d} \end{pmatrix}. \tag{23}$$

Assuming the case of constant coefficient $p(\mathbf{x})$ and $q = 0$ for the elliptic problem, the inner product (5) takes the form

$$\mathcal{A}(u,v) = \int_\Omega [p\nabla u \cdot \nabla v]\,\mathrm{d}\mathbf{x} = \sum_{k=1}^{d} \int_\Omega p \frac{\partial u}{\partial x_k}\frac{\partial v}{\partial x_k}\mathrm{d}\mathbf{x}. \tag{24}$$

To compute the integral in the reference space $\hat{\Omega}$, we evaluate the partial derivatives according to the the chain rule

$$\mathcal{A}(u,v) = \sum_{k=1}^{d} \int_{\hat{\Omega}} p \left(\sum_{i=1}^{d} \frac{\partial v}{\partial r_i}\frac{\partial r_i}{\partial x_k}\right)\left(\sum_{j=1}^{d} \frac{\partial u}{\partial r_j}\frac{\partial r_j}{\partial x_k}\right) J(\mathbf{r})\mathrm{d}\mathbf{r}. \tag{25}$$

Clearly, $J(\mathbf{r})$ and $\partial r_i/\partial x_k$ are constant for our particular rectangular parallelepiped case. Therefore, they would come out of the integral. This is not the situation for deformed elements, in which case we assemble them within a set of functions

$$\mathcal{G}_{ij}(\mathbf{r}) := \sum_{k=1}^{d} \frac{\partial r_i}{\partial x_k}\frac{\partial r_j}{\partial x_k} J(\mathbf{r}), \tag{26}$$

where for rectangular parallelepipeds,$\mathcal{G}_{ij}(\mathbf{r}) = 0$ for $i \neq j$ and constant otherwise. We can then evaluate the integral in (25) by using the numerical quadrature

$$\mathcal{A}(u,v) = \sum_{i=1}^{d}\sum_{j=1}^{d}\sum_{klm} \left[\frac{\partial v}{\partial r_i} p\mathcal{G}_{ij} \frac{\partial u}{\partial r_j}\right]_{(\xi_k,\xi_l,\xi_m)} \rho_k\rho_l\rho_k. \tag{27}$$

By combining the coefficient p, geometric term $\mathcal{G}_{ij}$, and quadrature weights into a set of d^2 diagonal matrices G_{ij}

$$G = \begin{pmatrix} G_{11} & G_{12} & G_{13} \\ G_{21} & G_{22} & G_{23} \\ G_{31} & G_{32} & G_{33} \end{pmatrix}, \tag{28}$$

$$(G_{ij})_{\hat{k}\hat{k}} := [p\mathcal{G}_{ij}]_{(\xi_k,\xi_l,\xi_m)}\,\rho_k\rho_l\rho_m, \tag{29}$$

with $\hat{k} := 1 + k + (N+1)l + (N+1)^2 m$. The derivatives in (27) can be written as

$$\left.\frac{\partial u}{\partial r_1}\right|_{klm} = \sum_{p=0}^{N} \hat{D}_{kp} u_{plm} = \underbrace{(I \otimes I \otimes \hat{D})}_{D_1}\,\underline{u}, \tag{30}$$

$$\left.\frac{\partial u}{\partial r_2}\right|_{klm} = \sum_{p=0}^{N} \hat{D}_{lp} u_{kpm} = \underbrace{(I \otimes \hat{D} \otimes I)}_{D2}\,\underline{u}, \tag{31}$$

$$\left.\frac{\partial u}{\partial r_3}\right|_{klm} = \sum_{p=0}^{N} \hat{D}_{mp} u_{klp} = \underbrace{(\hat{D} \otimes I \otimes I)}_{D_3}\,\underline{u}, \tag{32}$$

with $\hat{D}$ the one-dimensional derivative matrix. Combining the derivatives operators D_i and geometric operators G_{ij} we end up with the compact form for the energy inner product (25)

$$\begin{aligned} \mathcal{A}(u,v) &= \underline{v}^T \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix}^T \begin{pmatrix} G_{11} & G_{12} & G_{13} \\ G_{21} & G_{22} & G_{23} \\ G_{31} & G_{32} & G_{33} \end{pmatrix} \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix} \underline{u}, \qquad (33) \\ &= \underline{v} \underbrace{D^T G D}_{\bar{K}} \underline{u}. \qquad (34) \end{aligned}$$

The size of the full stiffness matrix is $(N+1)^3 \times (N+1)^3$, however we have that $G_{ij} = G_{ji}$ which reduces the leading-order storage to $6(N+1)^3$. The operator form in (34) also reduces the work required for a matrix-vector product. If we apply the $\bar{K}$ to a vector $\underline{u}$, we start by the tensor-product-based derivative evaluations

$$\underline{\hat{u}}_j = D_j \underline{u} \rightarrow 3 \times 2(N+1)^4, \qquad (35)$$

followed by the multiplication with the geometric factors

$$\tilde{u}_i = \sum_j G i j \underline{\hat{u}}_j \rightarrow 9 \times (N+1)^3, \qquad (36)$$

to finally sum across the tranposed derivative operator

$$\bar{K}\underline{u} = \sum_i D_i^T \tilde{u}_i \rightarrow 3 \times 2(N+1)^4, \qquad (37)$$

yielding to a total of $12(N+1)^4 + 9(N+1)^3$ operations, which is significantly less than the $2(N+1)^6$ operations required if the stiffness matrix is computed and stored explicitly.

For deformed elements, the mass matrix can be extended to

$$M_{\hat{i}\hat{i}} = J(\xi_i, \xi_j, \xi_k)\rho_i \rho_j \rho_k, \quad \hat{i} = 1 + i + (N+1)j + (N+1)^2 k. \qquad (38)$$

And for cases with $q(\mathbf{x})$ nonzero, the Helmoltz operator is created by augmenting the stiffness matrix $\bar{K}$ with the diagonal matrix with the nodal values $q(\mathbf{x}(\xi_i, \xi_j, \xi_k))$

$$\bar{H} := \bar{K} + QM. \qquad (39)$$

Surface integrals should also be defined for curved elements. They are important in the definition of the boundary conditions, for example, as well as for obtaining integrated quantities of interest. Consider that in the physical domain, a given surface is given by $\mathbf{x}^R(r,s) = \mathbf{x}(r,s,1)$. We omit the superscript ^{R} in the following equations. Infinitesimal displacement vectors over the surface in the physical domain ($\epsilon_\mathbf{r}$ and $\epsilon_\mathbf{s}$) can be obtained from

$$\epsilon_r = \frac{\partial \mathbf{x}}{\partial r} dr = \left(\frac{\partial x}{\partial r}, \frac{\partial y}{\partial r}, \frac{\partial z}{\partial r} \right)^T dr, \qquad (40)$$

$$\epsilon_s = \frac{\partial \mathbf{x}}{\partial s} ds = \left(\frac{\partial x}{\partial s}, \frac{\partial y}{\partial s}, \frac{\partial z}{\partial s} \right)^T ds. \qquad (41)$$

The area dS of the quadrilateral formed by $\epsilon_{\mathbf{r}}$ and $\epsilon_{\mathbf{s}}$ is just the norm of the cross product between these vectors, i.e.,

$$dS = ||\epsilon_{\mathbf{r}} \times \epsilon_{\mathbf{s}}|| = \left\| \frac{\partial \mathbf{x}}{\partial r} \times \frac{\partial \mathbf{x}}{\partial s} \right\| dr\ ds. \tag{42}$$

Note that the term with the norm is the surface Jacobian $\tilde{J}$, i.e.,

$$\tilde{J}_{ij} = \left\| \frac{\partial \mathbf{x}}{\partial r} \times \frac{\partial \mathbf{x}}{\partial s} \right\|_{ij}, \tag{43}$$

which is defined for every index i and j of the mesh over the surface. The unitary outward normal vector $\hat{\mathbf{n}}_{ij}$ is given by

$$\hat{\mathbf{n}}_{ij} = \frac{1}{\tilde{J}_{ij}} \left(\frac{\partial \mathbf{x}}{\partial r} \times \frac{\partial \mathbf{x}}{\partial s} \right)_{ij}. \tag{44}$$

If one wants to perform the two surface integrals of the Robin boundary condition (Eq. (12)), one have to add to the diagonal entries of $\bar{K}$ the values $\tilde{J}^R_{ij}\rho_i\rho_j\alpha p^R_{ij}$, and to the r.h.s., $\tilde{J}^R_{ij}\rho_i\rho_j\beta p^R_{ij}$.

The use of deformed elements induces some errors which are often referred as "variational crimes", due to the fact that some assumptions in the variational problem are violated. Considering the discrete approximation of the weak formulation for the elliptic problem that consists in finding $u \in V_N$ such that

$$\mathcal{A}(u, v) = \mathcal{F}(v) \quad \forall v \in V_N, \tag{45}$$

where $V_N \subset V$, with V_N being the finite dimension approximation of the infinite dimensional space V. V is a subspace of the Sobolov space H^1_0 that satisfies the Dirichlet boundary conditions. We outline two situations in which some assumptions are violated:

1. As explained before, the use of deformed elements requires a variable Jacobian and geometric factors that remains inside the integral. This increases the order of the polinomiyal to integrate, which in turns degrades the accuracy of the quadrature, replacing $\mathcal{A}(u, v)$ and $\mathcal{F}(v)$ by some approximation. An example of the lost in accuracy given by the integration over deformed geometries can be found in [15]. To overcome this issue, one could also increase the order of the integration rule, with the added cost of increasing the number of operations. This trade-off between accuracy of the quadrature and reduced number of operations was investigated by [9], who concluded that it was better to recover accuracy by simply increasing N. This conclusion is based on the fact that high-order methods tend to diminish the severity of this "variational crime".

2. For the case of elements with deformed boundaries, the mesh generated by the isoparametric map only approximates the actual domain and therefore the homogeneous boundary conditions cannot be satisfied exactly by the members of V_N, yielding to the situation where $V_N \not\subset V$. For simple geometries one could use a different mapping that represents the boundaries exactly. However, this would not be general nor useful from a computational point of view.

3.2 Generation of deformed geometries

When discrtizating the domain, the set of values $\mathbf{x}_{ijk}$ in the isoparametric map in equation (22) should satisfy two conditions

1. Have a smooth distribution $\mathbf{x}(\mathbf{r})$ in the interior of $\hat{\Omega}$ so the integrands are accurately evaluated.
2. Satisfy the prescribed boundary conditions $\mathbf{x}_{ijk} = \tilde{\mathbf{x}}(\xi_i, \xi_j, \xi_k)|_{\partial\hat{\Omega}}$

In `Nek5000` this is implemented by a blending technique derived by [6], which is based on the Boolean sum of interpolation operators. The type of operators we focus on are of the form

$$I_r f(\mathbf{r}) = \sum_{\hat{i}=0}^{m} \pi_{\hat{i}}^m(r) f(r_{\hat{i}}, s, t), \tag{46}$$

$$I_s f(\mathbf{r}) = \sum_{\hat{j}=0}^{n} \pi_{\hat{j}}^n(s) f(r, s_{\hat{j}}, t), \tag{47}$$

$$I_t f(\mathbf{r}) = \sum_{\hat{k}=0}^{l} \pi_{\hat{k}}^l(t) f(r, s, t_{\hat{k}}), \tag{48}$$

where f is a continuous function and the operators I_r, I_s, and I_t are denoted Lagrange interpolation operators. In order for the projection to be linear along the r_*-direction, we set $m = n = l = 1$. Then, the projection $I_{r_*} f(\mathbf{r})$ interpolates f along the r_* direction and coincides with f at $r_* = \pm 1$. The bivariate interpolation is then given by the tensor product

$$I_r I_s f = \sum_{\hat{i}=0}^{1} \sum_{\hat{j}=0}^{1} \pi_{\hat{i}}^1(r) \pi_{\hat{j}}^1(s) f(r_{\hat{i}N}, s_{\hat{i}N}). \tag{49}$$

By introducing the Boolean sum $I_{r_i} \oplus I_{r_j}$ $(i \neq j)$

$$I_{r_i} \oplus I_{r_j} := I_{r_i} + I_{r_j} - I_{r_i} I_{r_j}, \tag{50}$$

where finally the following relationships can be shown

$$\mathbf{x}(\mathbf{r}) = (I_r \oplus I_s)\tilde{\mathbf{x}}(\mathbf{r}) \quad (\text{in } \mathbb{R}^2), \tag{51}$$

$$\mathbf{x}(\mathbf{r}) = (I_r \oplus I_s \oplus I_t)\tilde{\mathbf{x}}(\mathbf{r}) \quad (\text{in } \mathbb{R}^3). \tag{52}$$

These expressions allow us to construct $\mathbf{x}$ from the boundary values $\tilde{\mathbf{x}}$, which are reproduced exactly. By expanding the Boolean sum, we can obtain a sequence of approximations, where ,in each step, information of successively higher dimension is included. For a 2D case this reads

$$(I_r \oplus I_s)\tilde{\mathbf{x}}(\mathbf{r}) = I_r I_s \tilde{\mathbf{x}} + I_r(\tilde{\mathbf{x}} - I_r I_s \tilde{\mathbf{x}}) + I_s(\tilde{\mathbf{x}} - I_r I_s \tilde{\mathbf{x}}). \tag{53}$$

The first term in the right hand side corresponds to the bilinear interpolant of the vertex only, whereas the second and third terms correspond to the linear interpolation of the s-edges along r and the linear interpolation of the r-edges along s, respectively.

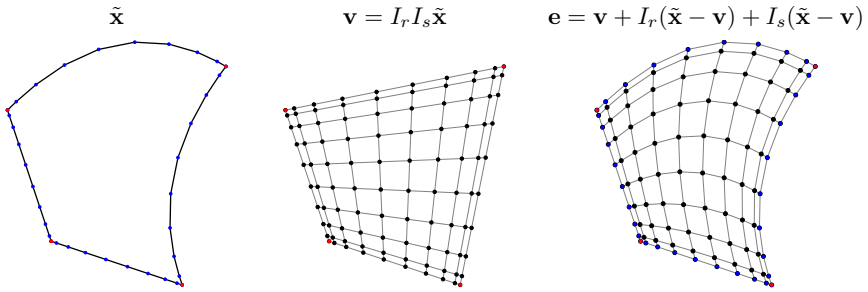
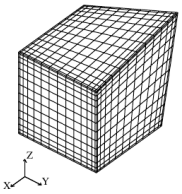


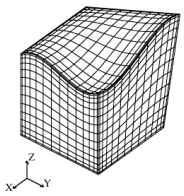
Figure 3: Gordon-Hall 2D.

$$\mathbf{v}_{ijk} = \sum_{\hat{\imath}\hat{\jmath}\hat{k}} \pi^1_{\hat{\imath}}(\xi^N_i)\pi^1_{\hat{\jmath}}(\xi^N_j)\pi^1_{\hat{k}}(\xi^N_k)\tilde{\mathbf{x}}_{\hat{\imath}N,\hat{\jmath}N,\hat{k}N}$$



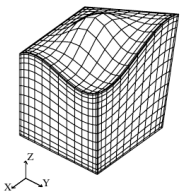
$\mathbf{v}$

$$\begin{aligned}\mathbf{e}_{ijk} = \mathbf{v}_{ijk} &+ \sum_{\hat{\jmath}\hat{k}} \pi^1_{\hat{\jmath}}(\xi^N_j)\pi^1_{\hat{k}}(\xi^N_k)(\tilde{\mathbf{x}}_{i,\hat{\jmath}N,\hat{k}N} - \mathbf{v}_{i,\hat{\jmath}N,\hat{k}N}) \\ &+ \sum_{\hat{\imath}\hat{k}} \pi^1_{\hat{\imath}}(\xi^N_i)\pi^1_{\hat{k}}(\xi^N_k)(\tilde{\mathbf{x}}_{\hat{\imath}N,j,\hat{k}N} - \mathbf{v}_{\hat{\imath}N,j,\hat{k}N}) \\ &+ \sum_{\hat{\imath}\hat{\jmath}} \pi^1_{\hat{\imath}}(\xi^N_i)\pi^1_{\hat{\jmath}}(\xi^N_j)(\tilde{\mathbf{x}}_{\hat{\imath}N,\hat{\jmath}N,k} - \mathbf{v}_{\hat{\imath}N,\hat{\jmath}N,k})\end{aligned}$$



$\mathbf{e}$

$$\begin{aligned}\mathbf{f}_{ijk} = \mathbf{e}_{ijk} &+ \sum_{\hat{\imath}} \pi^1_{\hat{\imath}}(\xi^N_i)(\tilde{\mathbf{x}}_{\hat{\imath}N,j,k} - \mathbf{e}_{\hat{\imath}N,j,k}) \\ &+ \sum_{\hat{\jmath}} \pi^1_{\hat{\jmath}}(\xi^N_j)(\tilde{\mathbf{x}}_{i,\hat{\jmath}N,k} - \mathbf{e}_{i,\hat{\jmath}N,k}) \\ &+ \sum_{\hat{k}} \pi^1_{\hat{k}}(\xi^N_k)(\tilde{\mathbf{x}}_{i,j,\hat{k}N} - \mathbf{e}_{i,j,\hat{k}N})\end{aligned}$$



$\mathbf{f}$

Figure 4: Gordon-Hall 3D [3].

Examples of the application of the sequence of approximation for 2D and 3D elements are shown in Fig. 3 and Fig. 4, respectively.

The implementation in `Nek5000` can be found in the file `navier5.f` under the subroutines `gh_face_extend_3d` and `gh_face_extend_2d`, which are called during the generation of the tri- or bilinear mesh in the `xyzquad` subroutine and `fix_geom`.

3.3 Description of the mesh files

In the following sections, we discuss some of the functions and files described in Fig. 5, which portrays the mesh generation process for `Nek5000`.

The ASCII `.rea` file contains the parameters defining the simulation, information about the mesh (vertices of the elements before the collocation of GLL points), and location of the boundary conditions. This file contains a list of run

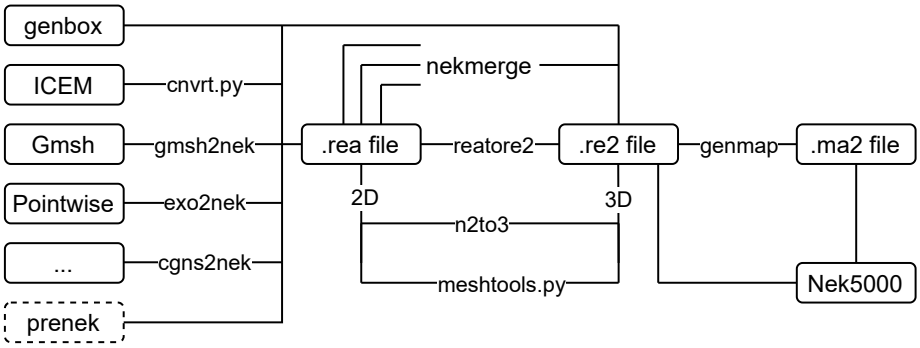


Figure 5: Diagram showing the mesh generation process for `Nek5000`.

parameters (`p01`, `p02`,...). However, these parameters are currently set elsewhere (`.par`, `.usr`, and `SIZE` files), leaving this section of the file non-functional. In the sequence, the mesh and location of the boundary conditions are defined in the `.rea` file. Notice that as the size of the meshes used in `Nek5000` has grown over time, the functional part of the `.rea` file was transferred to the binary `.re2` file, which is what `Nek5000` actually reads. However, since the latter is often generated from the former, through the use a converter such as the `reatore2` routine, it is still useful to describe the `.rea` file.

The beginning of the mesh definition section contains the parameters `NEL`, `NDIM`, and `NELV`, which are respectively the total number of elements (considering the solid and fluid elements), the number of dimensions of the problem, and the number of fluid elements. Solid elements may occur in problems of conjugate heat transfer. In the line starting by the string `ELEMENT`, the first value is the element number. In the following, the `.rea` contains the x and y (and z in 3D) coordinates of the nodes of the elements. The nodes are numbered from 1 to 4 in 2D and from 1 to 8 in 3D as portrayed in Fig. 6.

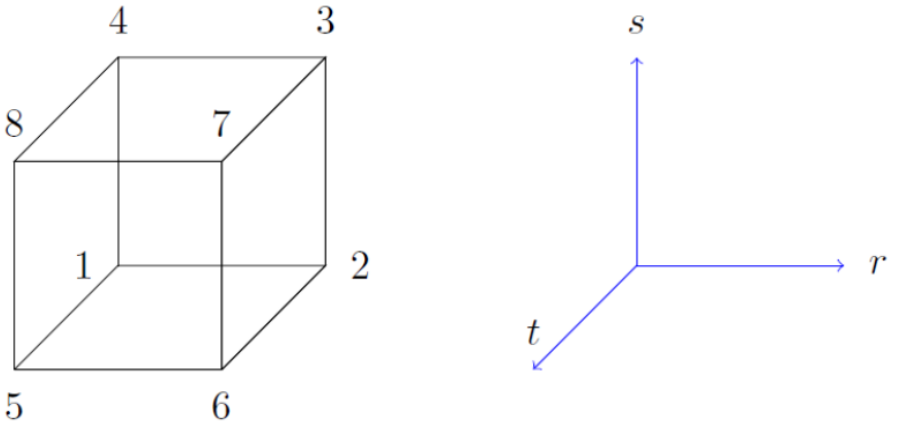


Figure 6: Numbering of the nodes of the elements [1].

A comparison between the 2D and 3D definition of the nodes of the elements in the `.rea` file is shown in Figs. 7 and 8. One can observe that the 2D file

presents only x and y points, whereas the 3D one also has the z component. The numbers in the figures indicate the node to which that coordinate pertains. In the case of the 3D file, the nodes from 5 to 8 lie on the face opposed to the one defined by the nodes from 1 to 4.

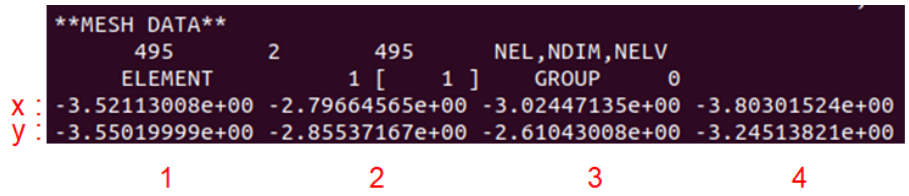


Figure 7: Definition of the nodes of the elements in the `.rea` file in 2D.

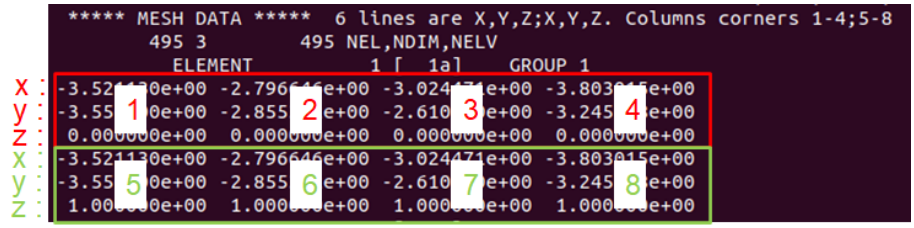


Figure 8: Definition of the nodes of the elements in the `.rea` file in 3D.

Curved edges of elements are defined in a special section of the `.rea` file. Figure 9 presents an excerpt of such section. The first value (`IEDGE`) indicates the edge number (1 to 4 in 2D and 1 to 12 in 3D). The second value (`IEL`) is the element number. In the current case, `IEDGE` is 4 and `IEL` is 18. The edge numbering convection is presented in Fig. 10. The second value is the element number. The meaning of the values in the 3rd to 7th positions of this line will depend on the type of edge, which is defined by the character in the 8th position of the line. The character `'m'` indicates that the curved edge is reconstructed quadratically. In this case, the x, y, and z coordinates of the midpoint of the edge need to be given at the 3rd, 4th, and 5th positions of the line. The character `'C'` indicates that the curved edge is a circular arc. It requires the radius of the circle to be given at the 3rd position of the line. Finally, the character `'s'` means that the edge lies on a sphere, which requires providing its radius at the 3rd position and its center at the 4th, 5th, and 6th positions of this line.

```
***** CURVED SIDE DATA *****
  34 Curved sides follow IEDGE,IEL,CURVE(I),I=1,5, CCURVE
4 18  7.573031e-02 -7.541230e-02  0.000000e+00  0.000000e+00  0.000000e+00 m
```

Figure 9: Definition of the curved elements in the `.rea` file.

Lastly, the `.rea` file also contains the type of boundary conditions that each face of the element has. Figure 11 presents an extract from this section. The 1st character (`'E'`) is the type of boundary condition. It can take values of `'E'` for internal faces, `'P'` for periodic faces, `'SYM'` for symmetric faces, `'O'` (`'o'`) for outflow faces with zero (nonzero) pressure, `'W'` for wall boundary condition, `'V'`

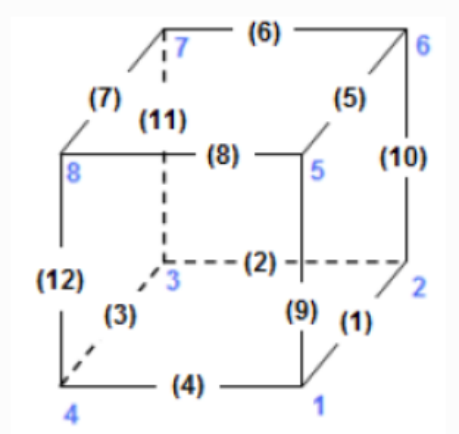


Figure 10: Definition of the edge numbering in the `.rea` file [1].

('v') for constant (user-specified) Dirichlet velocity boundary condition, among other possibilities. The 2nd value in the line (1) is the element number. The 3rd value (1) is the face number. The face number can be 1 in 2D and from 1 to 6 in 3D. More details about the face numbering can be seen in Fig. 12. The 4th and 5th values are the element connectivity and face connectivity.

```
  ***** BOUNDARY CONDITIONS *****
  ***** FLUID   BOUNDARY CONDITIONS *****
E     1  1 1.7200000e+02 2.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
```

Figure 11: Definition of boundary conditions in the `.rea` file.

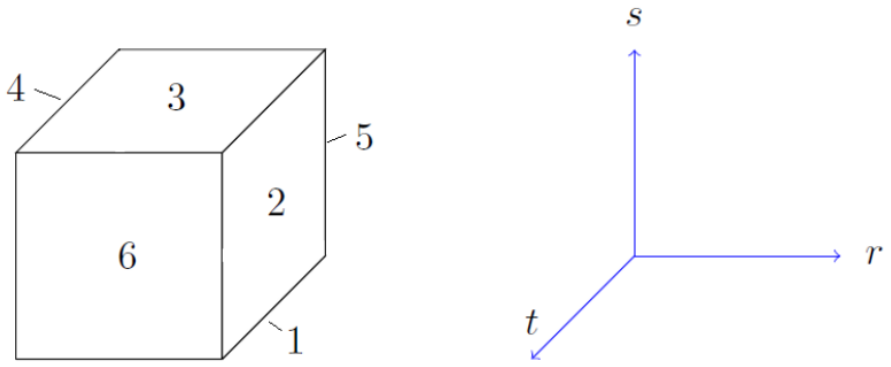


Figure 12: Definition of the face numbering in the `.rea` file [1].

In the following, the `genmap` routine is used to convert the `.rea` (`.re2`) file to the ASCII `.map` (binary `.ma2`) file, which contains information about the connectivity of the elements. In summary, `Nek5000` requires, among other files, the `.rea` and `.map` files in older versions and the `.re2` and `.ma2` files in newer versions.

3.4 Mesh Related Functions

There are several functions that allow the user to manipulate the mesh. This can be done either before or during the `Nek5000` run. Firstly, we consider the functions that are used in the set-up of the simulation. The functions `reatore2` and `genmap` pertain to this class but were already discussed in the previous section.

`genbox` is a routine that allows the user to generate rectangular meshes in 2D or 3D, and it requires a `.box` file. An extract from this file is presented in Fig. 13. The 1st and 2nd lines are the dimension of the generated mesh (2 or 3) and the number of fields, respectively. The latter can be more than 1 in problems involving velocity and temperature fields, for example. The first line after the string `box` specifies the number of elements in the x (22), y (16), and z (19) directions. The negative signs before these numbers indicate that `genbox` should generate the distribution of points based on the initial and final positions for each direction together with the ratio of growth of the mesh spacing in that direction. The following three lines indicate the initial and final positions in the x, y, and z directions of the domain and the aforementioned ratio. From the figure, we notice that the mesh in x should start at 0 and finish at 2, being uniformly space (ratio of 1), while in the y direction, it should start at 0 and finish at 1 with a ratio of growth of the elements of 1.15. Finally, the last line of the `.box` file provides the boundary conditions for the 1st and 2nd borders in the x direction ('`v`' and '`O`'), the 1st and 2nd borders in the y direction ('`W`' and '`v`'), and the 1st and 2nd borders in the z direction ('`P`' and '`P`').

```
-3                  spatial dimension
1                   number of fields (U)

Box
-22 -16  -19                        nelx,nely,nelz for Box
0.0 2.0 1.0                         x0,xn,ratio
0.0 1.0 1.15                        y0,yn,ratio
0.0 3.0 1.0                         z0,zn,ratio
v  ,O  ,W  ,v  ,P  ,P               BC's: (cbx0,cbx1,cby0,cby1,cbz0,cbz1)
```

Figure 13: Extract from the `.box` file.

However, in order to generate more complex meshes, such as one over curved airfoils, one may need to generate it outside the scope of `Nek5000` and convert it to `.rea` or `.re2`. Suitable programs to generate the mesh include `GAMBIT`, `Pointwise`, `Gmsh`, and `ICEM`. In the `Nek5000/tools` section, there are several converters to `.rea` or `.re2`: `exo2nek` for `EXODUS II` (`.exo`) files, `cgns2nek` for `.cgns` files, `gmsh2nek` for `.gmsh` files. In the case of the conversion from `.msh` format generated by `ICEM` to `.rea`, one needs to resort to an external converter such as the one available in [12]. For example, consider the case of a C mesh over an airfoil (2D) saved in the `airfoil.msh` file. The end of this file is shown in Fig. 14. The airfoil surface is defined by the curves `AFLE` (leading edge), `AFU` (upper part), `AFL` (lower part), and `AFTE` (trailing edge). The C boundary is described by the curves `INLET`, `FFU`, and `FFL`. The outlet boundary is given by `OUTLET`. The zone numbers of each one of these curves are given in the second column of the `.msh` file.

Figure 15 shows the application of the mesh converter in a `python3` environment. The 1st input parameter to the converter is the name of the `.msh` file. In the `curves` section, it is necessary to provide the zone number of each one

```
(0 "Zone Sections")
(39 (11 solid SOLID)())
(39 (12 interior int_SOLID)())
(39 (13 velocity-inlet INLET)())
(39 (14 velocity-inlet FFU)())
(39 (15 velocity-inlet FFL)())
(39 (16 pressure-outlet OUTLET)())
(39 (17 wall AFLE)())
(39 (18 wall AFU)())
(39 (19 wall AFL)())
(39 (20 wall AFTE)())
```

Figure 14: Extract from the `.msh` file.

of the curved edges of the mesh. In this case, the zone numbers of the curves describing the airfoil are provided together with the character `'m'`, for a second-order reconstruction of these edges. The edges non specified in this section are treated as linear ones. In the `bcs` section, the zone numbers should be provided along with a character describing the type of the boundary condition of each of these zones. For example, the curve `INLET`, whose zone number is 13, has a velocity Dirichlet boundary condition specified by `'v'`.

```
from mshc import *
convert('airfoil.msh',curves={17:{'type':'m'},18:{'type':'m'},19:{'type':'m'},20:{'type':'m'}}
                                ,bcs={17:'W',18:'W',19:'W',20:'W',13:'v',14:'v',15:'v',16:'O'})
```

Figure 15: Running the converter from `.msh` to `.rea`.

Another useful function is the `n2to3`. This routine allows for obtaining 3D meshes from 2D ones by extrusion. A call of `n2to3` leads to the request of parameters shown in Fig. 16. The 1st parameter is the name of the 2D `.rea` file without the suffix. The 2nd one is the name of the 3D file obtained from the extrusion. The 3rd parameter is 0 for the generation of a `.rea` file and 1 for a `.re2`. The 4th parameter is the number of elements in the extruded direction; `Nek5000` requires a minimum level of 3 elements. The 5th and 6th parameters are the minimum and maximum coordinates in the extruded direction. The 7th parameter describes how the elements are distributed in the extruded direction, and it allows either uniform or non-uniform spacing of the elements. The 8th parameter should be set to `yes` only in the case of simulations in Computational Electromagnetics (CEM). The last parameter is the boundary condition that should be applied at the two endings of the extruded mesh.

```
 Input .rea name to extrude:
airfoil
 Input .rea/.re2 output name
airfoil3D
 Input 0:ASCII or 1:BINARY
1
 input number of levels: (1, 2, 3,...; < 0 for circular sweep.):
3
 input z min:
0.0
 input z max:
0.1
 input gain (0=custom,1=uniform,other=geometric spacing):
1
 This is for CEM: yes or no:
no
 Enter Z(5) FLUID boundary condition (P,v,O,ect):
P
```

Figure 16: Running the mesh extruder `n2to3`.

However, the `n2to3` function only allows orthogonal extrusions, and one cannot refine locally at some regions of the mesh. In order to overcome these limitations, one may use the `meshtools.py` suite [8]. Figure 17 shows the types of extrusions that can be obtained. The left-most mesh is the 2D grid over an airfoil. The 2nd mesh shows the non-orthogonal extrusion of the 1st mesh in order to obtain a tapered wing. The 3rd picture shows the cylindrical mesh over a wind turbine. The last frame shows a cut of the previous mesh, portraying the local grid refinement.

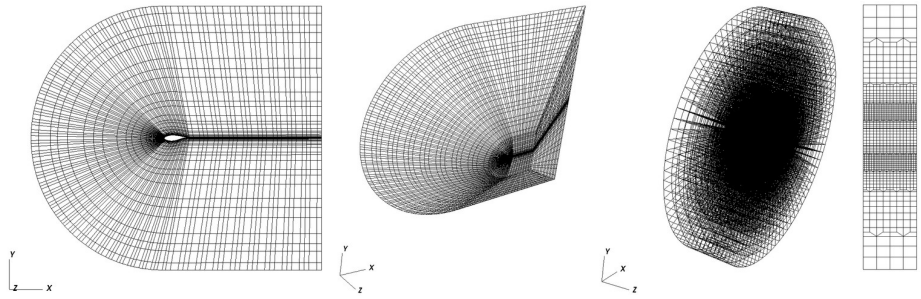


Figure 17: From left to right: 2D airfoil mesh; tapered-wing mesh (non-orthogonal extrusion); wind-turbine mesh; local grid refinement.

Now we consider the functions for the manipulation of the mesh during the `Nek5000` run. These functions are defined inside the `.usr` file. The first one is the `usrdat` function that is called before the GLL points are laid in the mesh. It can be used, for example, for altering the position of the vertices of the elements, such as scaling a box mesh with unitary length ($[0,1]^2$) to a box mesh with length 2π ($[0,2\pi]^2$). An extract from this function is shown in Fig. 18.

```
      subroutine usrdat
      include 'SIZE'
      include 'TOTAL'
      integer e

      one   = 1.
      twopi = 8.*atan(one)

      do e=1,nelv   !  Rescale mesh to [0,2pi]^2
      do i=1,4      !  Assumes original domain in .rea file on [0,1]
         xc(i,e) = twopi*xc(i,e)
         yc(i,e) = twopi*yc(i,e)
      enddo
      enddo

      return
      end
```

Figure 18: Extract from the `usrdat` function.

The `usrdat2` function is called after the GLL points are laid in the mesh. It can be used to modify the spectral grid and to generate more complex mesh geometries than those that can be obtained with `genbox`. For example, starting with a rectangular mesh $[0,9] \times [0,3]$ obtained with `genbox`, the x and y coordinates of the GLL points can be modified in the `usrdat2` function in order to obtain a periodic hill, as shown by the code in Fig. 19. Another use of `usrdat2` is to smooth the spectral mesh, as will be discussed later in this document. One should notice that all geometric entities and SEM operators need to be

regenerated after the modification of the mesh in `usrdat2`.

```
subroutine usrdat2()
implicit none
include 'SIZE'
include 'TOTAL'

integer ntot,i
real sa,sb,sc,xx,argx,A1

ntot = lx1*ly1*lz1*nelt

sa   = 4.5
sb   = 3.5
sc   = 1./6

do i=1,ntot
   xx   = xm1(i,1,1,1)
   argx = sb*(abs(xx-sa)-sb)
   A1   = sc + sc*tanh(argx)
   ym1(i,1,1,1) = ym1(i,1,1,1) + (3-ym1(i,1,1,1))*A1
enddo

return
end
```

Figure 19: Extract from the `usrdat2` function.

3.5 Mesh Smoothing

The main goal of mesh smoothing is to increase the computational efficiency, which translates into savings in CPU hours. In `Nek5000` this is implemented using the algorithm developed by [11], where the principal two savings are related to the increase in the maximum allowable time-step size and the reduction in the number of pressure iterations.

The most computationally expensive step in the solution of the incompressible Navier-Stokes equations is solving the pressure Poisson equation, which in `Nek5000` is performed with the generalized minimum residual (GMRES) method. The convergence of the solution is controlled by the iterative condition number κ_{iter}, which is the ratio between the maximum and minimum eigenvalues of the preconditioned pressure Poisson operator.

Mesh smoothing in `Nek5000` is performed through the initial application of an intra-element Laplace smoothing, which consists of updating the position of the nodes with the average of the position of the neighbouring nodes. This is followed by an optimization stage, which consists of the minimization of an objective function whose main part is the sum of the square of $||J||_F||J^{-1}||_F$ (condition number of J) over all nodes of all elements, where J is the Jacobian of the mapping, and $_F$ denotes the Frobenius norm. A weight function is defined to control how much each node should be displaced during the Laplace and optimization smoothing. In general, one would want to have a value close to zero near the boundaries in order to preserve the refinement in important regions such as the boundary layer. The two weight functions available are the exponential and tanh, whose plots as a function of the distance from the boundary are shown in Fig. 20 in the left and right, respectively.

The main difference between the exponential and tanh weight functions is that the latter allows for a region of nodes unaffected by smoothing near the boundaries (the tanh weight function can attain values very close to 0 for a

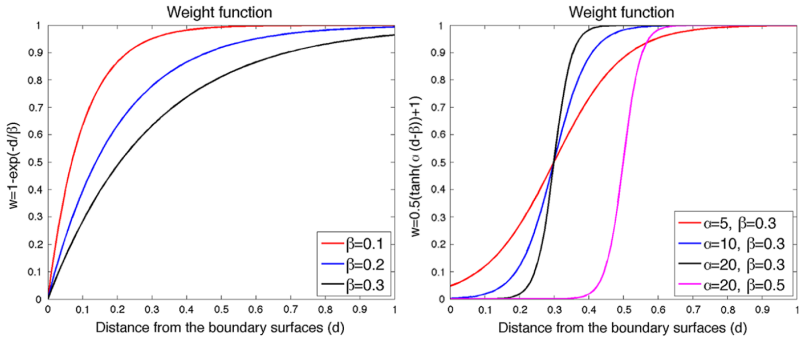


Figure 20: Exponential and tanh weight functions of the smoother [11].

distance from the boundary $\neq 0$). Moreover, the exponential function grows faster, which may not be desirable in order to keep the near-boundary resolution. The rise in β leads to slower growth of the weight function for both functions. The tanh function has an extra tuning parameter α, whose rise leads to slower near-boundary growth and faster free-stream increase.

Figure 21 shows a call of the mesh smoother for a C mesh over an airfoil. This is done inside the `usrdat2` function in the `.usr` file. `nbc` is set to 3, since we want to keep the resolution around the three boundaries of the mesh (airfoil surface, inlet, and outlet). These boundaries have three boundary conditions (`'W'`, `'v'`, and `'O'`), which are specified in `dcbc`. `idftyp` is the specification of the weight function (0 for exponential and 1 for tanh). The parameters `alpha` and `beta` of these functions are also specified. `nouter`, `nlap`, and `nopt` are the number of iterations of the outer loop (Laplacian + optimizer), the number of iterations of the Laplacian smoother, and the number of iterations of the optimization smoother. For stability, the minimum values for those parameters are 20, 20, and 40, respectively [11]. `idftyp = 1` (only option available) specifies that the objective function is based on the Jacobian.

```
      subroutine usrdat2
      include 'SIZE'
      include 'TOTAL'

      parameter(nbc=3)         !number of boundary conditions
      character*3 dcbc(nbc)
      save        dcbc
      data        dcbc /'W  ','v ','O '/  !BCs listed here

      idftyp = 1      !distance function - 0 -> exponential, 1-> tanh
      alpha = 20.     !Input for wall distance function
      beta  = 0.3     !

      nouter = 20      !total loops around laplacian and optimizer smoothing
      nlap = 20        !number of laplacian iterations in each loop
      nopt = 40        !number of optimization iterations in each loop

      mtyp = 1         !metric type

      call smoothmesh(mtyp,nouter,nlap,nopt,nbc,dcbc,idftyp,alpha,beta)

      return
      end
```

Figure 21: Running the mesh smoother.

Figure 22 shows the meshes of a turbine blade (top) and internal combustion

engine (bottom). The left figures present the meshes before smoothing, and the right ones after smoothing. It is clear that the smoothed meshes present a more gradual transition between elements and less sharp edge junctions. Moreover, the higher resolution near the walls is preserved by the smoother.

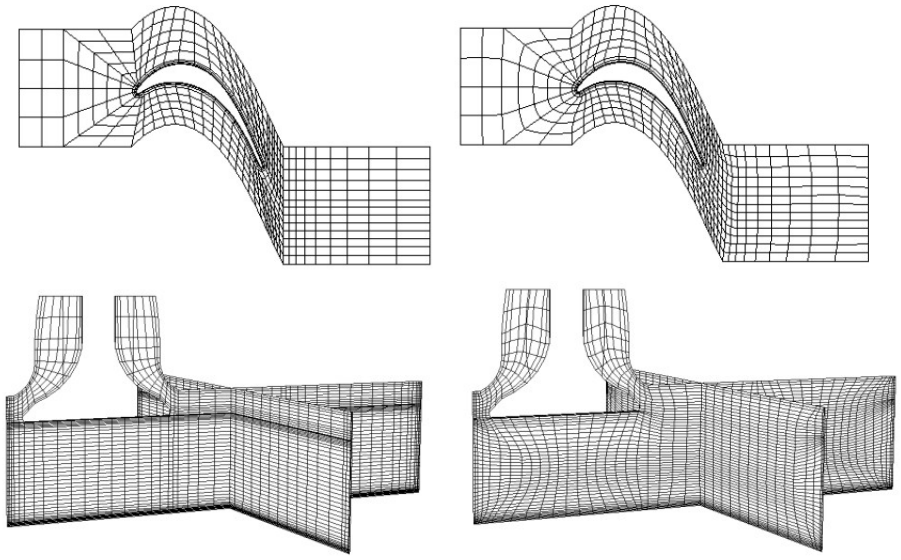


Figure 22: Results from the application of the mesh smoother. Top and bottom are the meshes of a turbine blade and ICE. Left and right are meshes before and after smoothing [10].

The smoother may allow a reduction in the number of iterations of the pressure solver of 4-47%, iterative condition number of 7-76%, and overall run time of 1.5-33% [11]. The worst results refer to the oscillatory flow around a 3D cylinder, and the best, to the flow around a 2D half-cylinder. To sum up, the mesh smoother allows savings in computational time while keeping the quality at key places such as boundary layers.

3.6 Mesh Quality

Meshing is one of the most time-consuming parts of the simulation setup. One needs to pay close attention to some parameters when generating a mesh for `Nek5000`. In the next, we will consider the case of a well-resolved LES simulation of an airfoil [14, 13] for a more concrete example of the parameters that one should consider when setting up a mesh and the order of magnitude of such parameters. In no case, the following text intends to be a one-size-fits-all recipe for mesh design. Further analyses are required for designing a mesh for cases different from the presented here.

The first thing that should be considered is whether the mesh has the appropriate resolution for capturing the flow physics. We first consider the near-wall resolution. For that, the following parameters are defined:

$$\Delta y^+_{wall} = \frac{\Delta y_{wall}}{\nu\sqrt{\tau_w/\rho}}, \quad \Delta x^+_{wall} = \frac{\Delta x_{wall}}{\nu\sqrt{\tau_w/\rho}}, \quad \Delta z^+_{wall} = \frac{\Delta z_{wall}}{\nu\sqrt{\tau_w/\rho}}, \tag{54}$$

where Δy_{wall} is the distance in the normal direction from the wall to the first GLL point, Δx_{wall} and Δz_{wall} are the distances between two consecutive GLL points in the streamwise and spanwise directions, ν is the kinematic viscosity, τ_w is the wall stress, and ρ is the density.

One would want to resolve the boundary layer over the airfoil. For that, it is often required $\Delta y^+_{wall} < 1$. Moreover, one would have to specify the growth of Δy moving away from the wall. This requires a mesh law, e.g., geometric or exponential, and the ratio of growth of the spacing between two consecutive elements, e.g., 1.15. It is also interesting to limit the maximum Δy^+, e.g., $\Delta y^+_{max} = 20$, otherwise the aspect ratio $\Delta y/\Delta x$ may become too high. Notice that, since τ_w tends to be higher in the turbulent region, the physical spacing Δy_{wall} would have to be reduced so that $\Delta y^+_{wall} < 1$.

A suitable value for Δx^+_{wall} should also be selected, which could be a value around 20. Notice that the pressure side may allow a larger value of Δx^+_{wall} since it often presents laminar flow. In the spanwise direction, the correct value of discretization is also critical because the lack of resolution in this direction may lead to the formation of 2D structures in the flow, such as vortices, that do not properly break down to turbulence. One could select, for instance, $\Delta z^+_{wall} = \Delta x^+_{wall}/2$.

In the wake zone, one should pay attention to the parameter $\Delta x_{wake}/\eta$, where Δx_{wake} is the distance between two consecutive GLL points, and η is the Kolmogorov length scale. In the near-wake region, i.e., up to one chord downstream of the airfoil, one may use $\Delta x_{wake}/\eta < 18$, whereas for more downstream positions, this value could be gradually increased in order to reduce the computational cost. Notice that both τ_w and η should be obtained either through an experimental run of `Nek5000`, with a trial mesh, or through a RANS computation. Moreover, the lack of resolution in some areas of the problem, such as the trailing edge, as well as a too close outflow boundary of type `'O'` may lead to the instability and not convergence of `Nek5000`.

Another consideration regarding mesh is the aspect ratio, which is the quotient between the maximum and minimum lengths of an element. Higher aspect ratio elements increase the stiffness of the pressure Poisson operator, leading the iterative method used to solve it to require more iterations to converge. This issue can be alleviated with the use of a Schwarz pre-conditioner for the pressure Poisson operator [5]; however, it is a good practice to keep the aspect ratio below a certain level, e.g., 40. Furthermore, very different length scales across the mesh also increase the stiffness of the aforementioned operator and should be avoided, if possible, with a smoothing step such as the one described in Section 3.5. Finally, it is also desirable that the edges of the elements be orthogonal to the boundaries of the domain and particularly to the airfoil wall. In this case, one could use a smoother that enforces orthogonality, such as the one available in the commercial software `ICEM`.

References

[1] Convention of element, edge, and face numbering in Nek5000. http://nek5000.github.io/NekDoc/problem_setup/case_files.html. Accessed: 08-06-2021.

[2] Erik Boström. Investigation of outflow boundary conditions for convection-dominated incompressible fluid flows in a spectral element framework. Master's thesis, KTH, Numerical Analysis, NA, 2015.

[3] M. O. Deville, P. F. Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge, 2002.

[4] S. Dong. A convective-like energy-stable open boundary condition for simulations of incompressible flows. *Journal of Computational Physics*, 302:300–328, Dec 2015.

[5] P. F. Fischer. An Overlapping Schwarz Method for Spectral Element Solution of the Incompressible Navier–Stokes Equations. *Journal of Computational Physics*, 133:84–101, 1997.

[6] William J Gordon and Charles A Hall. Transfinite element methods: blending-function interpolation over arbitrary curved element domains. *Numerische Mathematik*, 21(2):109–129, 1973.

[7] J. G. Heywood, R. Rannacher, and S. Turek. Artificial boundaries and flux and pressure conditions for the incompressible navier–stokes equations. *International Journal for Numerical Methods in Fluids*, 22(5):325–352, 1996.

[8] V. Kleine, T. Fava, and G. Chauvat. Tool for non-orthogonal extrusion of the mesh with refinement for Nek5000. https://github.com/vitorkleine/meshtools_pymech. Accessed: 10-06-2021.

[9] Yvon Maday and Einar M Rønquist. Optimal error analysis of spectral methods with emphasis on non-constant coefficients and deformed geometries. *Computer Methods in Applied Mechanics and Engineering*, 80(1-3):91–115, 1990.

[10] K. Mittal. Mesh smoothing in Nek5000. Nek5000/examples/smoother/README.pdf. Accessed: 08-06-2021.

[11] K. Mittal and P. Fischer. Mesh smoothing for the spectral element method. *Journal of Scientific Computing*, 78:1152–1173, 1996.

[12] M. Mortensen. Converter from .msh to .rea. https://github.com/mikaem/tools/blob/master/mshconvert/mshconvert.py. Accessed: 08-06-2021.

[13] P. Negi, R. Vinuesa, A. Hanifi, P. Schlatter, and D. S. Henningson. Unsteady aerodynamic effects in small-amplitude pitch oscillations of an airfoil. *International Journal of Heat and Fluid Flow*, 71:378–391, 2018.

[14] P. S. Negi, A. Hanifi, and D. S. Henningson. LES of the unsteady response of a natural laminar flow airfoil. In *Proceedings of the 2018 Applied Aerodynamics Conference, Atlanta, Georgia, 25-29 June 2018*, 2018.

[15] C. R. Schneidesch and M. O. Deville. Chebyshev collocation method and multi-domain decomposition for Navier-Stokes equations in complex curved geometries. *Journal of Computational Physics*, 106(2):234–257, 1993.