

The BoSSS
Discontinuous Galerkin solver
for incompressible fluid dynamics
and an
extension to singular equations.

Vom Fachbereich Maschinenbau an der Technischen
Universität Darmstadt zur Erlangung des akademischen
Grades eines Doktor-Ingenieurs (Dr.-Ing) genehmigte

DISSERTATION

vorgelegt von

Dipl.-Ing. Florian Kummer

geboren in Innsbruck, Österreich

Berichterstatter:	Prof. Dr.-Ing. habil. Martin Oberlack
Mitberichterstatter:	Prof. Dr.-Ing. habil. Johannes Janicka
Tag der Einreichung:	22. August 2011
Tag der mündl. Prüfung:	15. November 2011

Darmstadt, März 2012

D17

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit, abgesehen von den in ihr ausdrücklich genannten Hilfen, selbstständig verfasst habe.

Florian Kummer

Bitte zitieren Sie dieses Dokument als:

URN: urn:nbn:de:tuda-tuprints-28898

URL: <http://tuprints.ulb.tu-darmstadt.de/2889>

Dieses Dokument wird bereitgestellt von tuprints, E-Publishing-Service der TU Darmstadt.

<http://tuprints.ulb.tu-darmstadt.de>

tuprints@ulb.tu-darmstadt.de

Die Veröffentlichung steht unter folgender Creative Commons Lizenz:

Namensnennung-Keine kommerzielle Nutzung-Keine Bearbeitung 2.0 Deutschland

<http://creativecommons.org/licenses/by-nc-nd/2.0/de/>



Contents

Nomenclature	9
1 Introduction	14
2 Software design	20
2.1 Introduction	20
2.2 General issues	20
2.2.1 Modularization	20
2.2.2 .NET for scientific computing	22
2.2.3 Flexibility versus implementation efficiency	31
2.3 Closer view on selected parts of the BoSSS-framework	31
2.3.1 Implementation of spatial operators and fluxes	31
2.3.2 Class-structure of DG fields	34
2.3.3 BoSSS database	36
3 Formal definition and properties of the BoSSS – framework	44
3.1 General definitions and notation	44
3.2 DG approximation of fields on numerical grids	46
3.2.1 The numerical grid	46
3.2.2 Definition of DG fields	50
3.3 MPI – Parallelization of the BoSSS code	54
3.3.1 Parallel partitioning of the grid	54
3.4 The discretization of spatial differential operators	57
3.5 Some special spatial operators	60
4 GPU – accelerated, MPI – parallel sparse solvers	65
4.1 Introduction	65
4.2 Matrix storage formats	67
4.3 MPI – parallelization	73
4.4 Implementation issues	77
4.4.1 Library Design	77
4.4.2 Comments on performance	79
4.5 Performance measurements	81

5	An Incompressible Single-Phase Navier-Stokes – solver	86
5.1	Notation	86
5.2	High Level view on the projection scheme	87
5.3	Spatial discretization	93
5.4	Temporal discretization of the predictor	101
5.5	Numerical Results	103
5.6	Basic preconditioning techniques	114
6	The extended DG method and the Level-Set - framework	119
6.1	The Level-Set – framework	120
6.2	The XDG framework	126
6.3	Poisson equation	131
6.4	Brief overview about theory of distributions	138
6.5	Heat equation	145
6.6	Summary and Outlook	150
7	Conclusion, final remarks and outlook	151
8	Bibliography	154
9	Curriculum Vitae	160

Acknowledgements

This work was funded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) in the Collaborative Research Center 568 (Sonderforschungsbereich 568, SFB 568).

The research that is presented in this thesis, especially the development of the BoSSS software framework, has been carried out during my employment as a PhD student at the Chair of Fluid Dynamics at the TU Darmstadt, in the period of late 2006 to 2011.

I acknowledge my debt to those who helped and supported me along the way.

In particular, I have to thank my supervisor Prof. Dr.-Ing. habil. Martin Oberlack for not only giving me the opportunity of doing a PhD thesis, but also supported me in following my own ideas and providing a stimulating working environment.

I also want to thank my co-supervisor and speaker of the SFB 568, Prof. Dr.-Ing. Johannes Janicka.

Furthermore let me thank my colleagues who worked with me on the BoSSS code, in alphabetical order, Nehzat Emamy, Christina Kallendorf, Benedikt Klein, Said Moradi, Roozbeh Mousavi B.T. and Björn Müller. I also have to thank the students who contributed to the BoSSS code, in particular Christoph Busold, who worked with me on the GPU acceleration of sparse solvers, Jochen Kling, who programmed the Database explorer and Roozbeh Ashoori, who did valuable testing and performance evaluation in his Master thesis. Last but not least I have to thank Prof. Dr.-Ing.habil. Yonqi Wang, with whom I had very fruitful discussions on multiphase problems.

Special thanks are devoted to the people who did the proofreading of this manuscript, in particular Alina and Gabriela Schandenhofer, Christina Kallendorf and Nehzat Emamy.

Last but not least, I am deeply grateful to my parents and my brothers for their constant support.

Abstract

This PhD thesis contains three major aspects: (1) the BoSSS software framework (or simply BoSSS code) itself, (2) an incompressible Navier-Stokes solver that is based on the BoSSS framework and finally (3) the development of the Extended Discontinuous Galerkin (XDG) method.

One major result is the BoSSS software framework (or simply BoSSS code) itself. Its core aspects are discussed from both, software engineering and mathematical point of view. The software design itself features some novel aspects. Up to our knowledge, it is the first time someone implemented a large-scale, MPI-parallel CFD-application in the C# – language.

The implemented BoSSS software library is a general tool for the discretization of a systems of balance equations, e.g. $\frac{\partial}{\partial t} u_\gamma + \text{div}(\mathbf{f}_\gamma) + q_\gamma = 0$, for $1 \leq \gamma \leq \Gamma$ by means of the Discontinuous Galerkin (DG) method.

On the foundation of this software library, a solver for incompressible single-phase problems, based on the projection method, was developed.

Since the solution of the Poisson equation proved to be the dominating operation in the incompressible Navier-Stokes solver, the Conjugate Gradient solver was ported to GPU (Graphics processing Unit), yielding an acceleration factor in the range of 5 to 20 in comparison to CPU.

By the XDG method, it becomes possible to treat equations with singularities, i.e. jumps and kinks in the the solution, without regularizing these singularities (i.e. without “smearing them” out). The final aim of the XDG method is the treatment of immiscible multiphase flows. Since in single-phase settings it is commonly accepted that fractional-step - approaches like the Projection method offer better performance than ‘overall’-schemes, which assemble a large nonlinear, differential-algebraic system from the Navier-Stokes equations, it seems beneficial to extend these ideas to multiphase flows. Therefore, solvers for singular scalar equations were developed: for the Poisson equation with jump, as a proptotype for elliptic steady-state problems and for the instationary Heat equation as an example for a time-dependent equation with moving interface.

Abstract (German)

Die vorliegende Arbeit hat drei wesentliche Schwerpunkte: zum ersten das BoSSS software framework (auch als "BoSSS-code" bezeichnet) selbst, zum zweiten einen Löser für die inkompressible Navier-Stokes Gleichung, welcher auf BoSSS aufbaut und drittens die Entwicklung der Erweiterten Diskontinuierlichen Galerkin Methode (Extended Discontinuous Galerkin, XDG).

Ein zentrales Resultat der Arbeit ist der BoSSS-code selbst; ausgewählte Aspekte werden vom Standpunkt des Software Engineering als auch vom mathematischen Standpunkt aus betrachtet. Das Softwaredesign selbst ist in vielerlei Hinsicht neuartig: soweit uns bekannt, wurde zum ersten Male ein MPI-paralleler CFD-Löser mittels der C#-Sprache implementiert.

Das implementierte BoSSS software framework ist ein allgemeines Werkzeug zur Diskretisierung allgemeiner Systeme von Bilanzgleichungen, d.h. Systemen der Form $\frac{\partial}{\partial t} u_\gamma + \text{div}(\mathbf{f}_\gamma) + q_\gamma = 0$ mit $1 \leq \gamma \leq \Gamma$, mithilfe der Diskontinuierlichen Galerkin (DG) Methode.

Auf Basis dieser Softwarebibliothek wurde ein Löser für einphasige inkompressible Probleme formuliert, welcher auf der Projektionsmethode basiert.

Da sich bei einphasigen inkompressiblen Problemen die Lösung der Poissongleichung als dominierende Operation hinsichtlich der Laufzeit darstellt hat, wurde eine Portierung des Konjugierten-Gradienten Löser auf GPU's (Graphics Processing Unit) durchgeführt. Im Vergleich zur CPU-Implementierung ergibt sich dabei eine Beschleunigung um den Faktor 5 bis 20.

Sinn und Zweck der XDG Methode ist die Numerische Abbildung singulärer partieller Differentialgleichungen, d.h. von Gleichungen mit Sprüngen und Knicken in der Lösung, ohne diese zu Regularisieren d.h. zu "verschmieren". Letztendlich soll die XDG Methode zur numerischen Abbildung mehrphasiger Strömungsprobleme benutzt werden. Im Kontext einphasiger Löser werden Fractional Step - Verfahren wie die Projektionsmethode als effizienter im Gegensatz zu integralen Schemata, welche die

gesamte Navier-Stokes Gleichung in ein großes nichtlineares differentiell-algebraisches System diskretisieren, angesehen. Es scheint daher vielversprechen, diese Ideen auf Mehrphasensysteme auszudehnen. Zu diesem Zwecke wurden Löser für skalare singuläre Probleme implementiert: zum einen, als Prototyp für ein stationäres elliptisches Problem die Poissongleichung mit Sprung; zum zweiten, als Beispiel für ein transientes Problem die Wärmeleitungsgleichung mit bewegter Trennfläche.

Nomenclature

- $Cp(\mathbf{x}, X)$ closest point in X to \mathbf{x} , see §119
- φ if not noted otherwise, the Level-Set function, see §116
- $\delta_{n,m}$ Kronecker-Delta
- $\mathfrak{A}(t)$ time-dependent domain of phase \mathfrak{A} , see §114
- \mathfrak{A}^\times time-space notation of $\mathfrak{A}(t)$, see §115
- $\mathfrak{B}(t)$ time-dependent domain of phase \mathfrak{B} , see §114
- \mathfrak{B}^\times time-space notation of $\mathfrak{B}(t)$, see §115
- $\mathfrak{I}(t)$ time-dependent interface between phase \mathfrak{A} and \mathfrak{B} , see §114
- \mathfrak{I}^\times time-space notation of $\mathfrak{I}(t)$, see §115
- Γ_{Diri} Dirichlet boundary region of a Heat- or Poisson equation: $\Gamma_{\text{Diri}} \subset \partial\Omega$
- Γ_{Neu} Neumann boundary region of a Heat- or Poisson equation: $\Gamma_{\text{Neu}} \subset \partial\Omega$
- Γ_{Inl} Inlet region of the incompressible Navier-Stokes equation, see §79
- Γ_{Out} Outflow region of the incompressible Navier-Stokes equation, i.e. region with inhomogeneous Neumann b.c. for pressure, see §79
- Γ_{POut} Pressure Outlet region of the incompressible Navier-Stokes equation, i.e. region with Dirichlet b.c. for pressure, see §79
- Γ_{Wall} Solid or moving wall region of the incompressible Navier-Stokes equation, see §79
- $\mathfrak{E}_{\text{Bnd}}(p, h)$ inter-process boundary between MPI processes p and h , see §36

$\mathcal{E}\{-,-\}$ Semigroup-like notation, see §77
 $\langle f, g \rangle_K$ inner product of f and g in K , see §6
 $\mathcal{D}'(\Omega)$ space of distributions, see §146
 $\mathcal{D}(\Omega)$ space of test functions in distribution theory, see §144
 \mathfrak{E} set of all edges of the numerical grid
 \mathfrak{E}_{bnd} set of all boundary edges of the numerical grid
 \mathfrak{E}_{int} set of all internal edges of the numerical grid
 \mathfrak{K} set of all cells, i.e. $\mathfrak{K} = \{K_0, \dots, K_{J-1}\}$
 $\mathfrak{K}_{ext}(p)$ external/Ghost cells on MPI process p see §37
 $\mathfrak{K}_{loc}(p)$ locally updated cells on MPI process p , i.e. not Ghost cells on MPI process p , see §37
 $\text{Vol}_D(\cdot)$ D - dimensional measure: $\text{Vol}_D(X) := \int_X 1 dx$
 MPI_{comm} MPI communicator, see §34
 sz number of MPI processes, “MPI size”, see §34
 Ω computational domain, $\Omega \subset \mathbb{R}^D$ open and simply connected
 Ω^\times time-space domain, $\Omega^\times = \mathbb{R}_{>0} \times \Omega$, see §115
 part Grid partition, see §35
 $\phi_{j,n}^X$ elements of the XDG basis, see §129
 ϕ_n orthonormal polynomial basis of the reference element, see §10
 $\phi_{j,n}$ Orthonormal basis of the DG-space, see §27
 $\text{dim}(\dots)$ vector-space dimension
 \tilde{f} tilde: denotes the DG coordinates of $\underline{f} \in DG_p(\mathfrak{K})$, see §29
 \underline{f} underlining: denotes a member of $DG_p(\mathfrak{K})$, see §30

- δ Delta - distribution, see §148
- $\mathbf{1}_X$ characteristic function of $X \subset \Omega$, see §5
- \mathbf{e}_d standard basis vector, i.e. $\mathbf{e}_d := [\delta_{d,j}]_{j=1,\dots,D}$, see §91
- \mathbf{n} normal vector
- $\mathbf{n}_{\mathcal{J}}$ oriented normal field on \mathcal{J} , see §114
- \mathbf{n}_X normal vector on $X \subset \Omega$.
- \mathbf{T}_j matrix of mapping T_j , see §18
- $CH(\dots)$ convex hull, see §7
- $Cond_a(\dots)$ matrix condition number, see §110
- $Cont_{\mathfrak{A}}(\dots)$ Continuation of an extended DG field, see §133
- $Cut(\varphi)$ set of cells cut by the Level-Set \mathcal{J} , see §124
- D if not noted otherwise, the spatial dimension, see §1
- d if not noted otherwise, the spatial direction, $1 \leq d \leq D$
- $DG_p(\mathfrak{K})$ space of DG functions of polynomial degree p , see §25
- f^{in} inner value of f on a cell boundary, see §48
- f^{out} outer value of f on a cell boundary, see §48
- f^{-1} inverse mapping or inverse relation of mapping $f : A \rightarrow B$, i.e. $f^{-1}(y) = \{x \in A; f(x) = y\}$, see §66
- $Far_+(\varphi)$ set of cells which are far away from the interface \mathcal{J} , see §124
- $Far_-(\varphi)$ set of cells which are far away from the interface \mathcal{J} , see §124
- $Gl_D(\mathbb{R})$ group of non-singular $D \times D$ - matrices
- $H^l(\Omega)$ Sobolev space, see §3

- J if not noted otherwise, number of cells in the computational grid, see §13
- j if not noted otherwise, the cell-index with $0 \leq j < J$
- $j_0(p)$ if not noted otherwise, index of first cell on MPI process p , see §35
- $J_u(p)$ Number of updated cells (exclusive Ghost cells) on MPI process p , see §35
- $J_{ext}(p)$ number of external/Ghost cells on MPI process p see §37
- $J_{tot}(p)$ total number of (locally updated plus Ghost) cells on MPI process p see §37
- K_{ref} reference element of the grid, see §9
- K_C patching domain, see §138
- K_j cell with index j , see §13
- $L^2(\Omega)$ Hilbert space of square-integrable functions, see §3
- $map(\dots)$ coordinate mapping for a list of DG fields, see §32
- n if not noted otherwise, DG polynomial index in one cell, $0 \leq n < N_p$, see §27
- $N_{DOF}(j)$ for the XDG basis, the number of degree-of-freedoms in cell K_j , see §129
- N_p Number of DG polynomials in one cell, i.e. $N_p = \dim(DG_p(\{K_{ref}\}))$, see §27
- $Near(\varphi, -)$ set of cells in the neighbourhood of $Cut(\varphi)$, see §124
- p_{POut} Dirichlet value for pressure outlet, see §79
- $Proj_p$ Projector onto $DG_p(\mathfrak{K})$, see §29
- $T_j(\dots)$ transformation from reference element to cell K_j , see §13
- Tr trace operator, see §46

$\partial^\alpha|_X$ restriction of ∂^α onto $X \subset \Omega$, see §134

$f|_{\partial\Omega}$ trace of $f \in H^n(\Omega)$, see §46

1 Introduction

For the present thesis, as a starting point one should focus on the question: “*Why should one write a new numerical software for computational fluid dynamics or multi-phase flows?*”

Although it is an interesting task on its own it may not be sufficient to justify the investment. From an economical point of view, it seems to be more attractive to take an existing code and extend it to suite a specific problem. This evolutionary approach generally proves to be superior until a certain point, at which one calls a design to be at its edge. The BoSSS software framework (*Bounded Support Spectral Solver*) not only contains novel numerical developments like the Extended Discontinuous Galerkin method, but also offers a combination of numerical- and software-engineering - features that makes it very different to any other public available code that is currently in existence.

At first, there was the demand for a rather modern numerical method, like Discontinuous Galerkin (DG) with an accuracy, or convergence order (see §44) that is superior than the second order which can be achieved by classical Finite-Volume (FV) or linear Finite-Element (FE) method.

Apart from that, there is also a high requirement for modularity, especially in the field of physical modelling. For instance, it may be useful for a broad range of applications, if it is possible to add a scalar transport equation to a working Navier-Stokes solver.

The code should be suitable for High Performance Computing (HPC). Since nowadays all supercomputers are essentially clusters, there is no way around MPI parallelization¹. Within a decade (approximately 1995 to 2005), the evolution of supercomputers was quite predictable: since processor cores became faster from generation to generation, one just had to wait approximately two years to get a compute cluster with double performance *at the same number of processors*. Around 2005 processor clock frequencies reached a practical upper limit around 4.0 GHz. Higher frequencies seemed to be impractical from viewpoints such as power consump-

¹http://en.wikipedia.org/wiki/Message_Passing_Interface, 12th of Dec. 2011

tion and thermal dissipation. A very prominent example of this trend is the Intel NetBurst² architecture, commonly known as Pentium 4. Actually one can identify two trends in the HPC landscape: at first, a shift towards massively parallel systems. The most prominent examples for this are the IBM BlueGene systems. Secondly, an architecture shift towards heterogeneous multi- and many-core chips and even more parallel architectures like Graphics Processing Units (GPU's) is observed.

A rather challenging consequence of the architecture shift is that codes which were used for years are not portable to the new computers, since the development was not predictable when these codes were created. One workhorse of the new BoSSS code – the sparse solver – is already supporting GPU architectures (see chapter 4), while the second workhorse – the quadrature kernel – is designed in a way that will allow the addition of GPU support.

Finally, the aim towards superior numerics was to implement the Extended DG (XDG, see chapter 6) method. The extension of the piecewise polynomial DG space, in one or the other way, seems to be a quite obvious idea. Its essence could be probably written down in less than one page, see §126. Though, an *efficient* implementation is complex, especially into an existing code that was not intentionally designed for a very special feature like this.

There is already a myriad of software packages for the numerical treatment of partial differential equations available. Very coarsely, they could be separated into general libraries for PDE treatment and more focused ones. As examples of the former group – which have a philosophy that is, in certain points similar to BoSSS – we want to mention:

- DUNE³, the 'Distributed and Unified Numerics Environment' which supports FV, FE and Finite-Difference;
- the pretty famous and mature OpenFOAM⁴ ('Open Field Operation and Manipulation') code; it is based on the Finite Volume method, and is especially well established in fluid dynamics, although it is claimed that it is even suitable e.g. for PDE's in the field of financial mathematics.

²[http://en.wikipedia.org/wiki/NetBurst_\(microarchitecture\)](http://en.wikipedia.org/wiki/NetBurst_(microarchitecture)), 12th of Dec. 2011

³DUNE: <http://www.dune-project.org/dune.html>, 12th of Dec.

⁴OpenFOAM: <http://www.openfoam.com>, 12th of Dec. 2011

- GetFEM++⁵, a C++ - based library for Finite Element, that also supports Extended FEM (XFEM, see below).

More focused codes are for example:

- DROPS⁶, which is especially focused on incompressible multiphase flows, based on the XFEM (see below) method;
- Νεκταρ⁷, a high-performance compressible and incompressible single-phase Navier-Stokes solver.

All of these codes have been written by very prominent and well-recognized researchers. This list is also far from being complete; here we cited only some codes that are freely available. We do not claim to have detailed inside knowledge about these codes to rank them against BoSSS. Each of these codes is in some sense unique, but also shares some properties with other ones; this is the same with BoSSS.

However, we claim that the BoSSS package – especially, its combination of features, i.e. DG and XDG, implemented in C#, generic in grid and spatial dimension, modularity to make it suitable for a broad range of PDE's, partial GPU support with more to come – gives it a unique position among the other scientific codes.

Evolution of DG methods. According to Cockburn (Cockburn, Karniadakis & Shu 2000), the Discontinuous Galerkin method was invented by Reed and Hill in 1973, (Reed & Hill 1973), aiming towards neutron transport problems. For time discretization, basically two options are available: The first is to use DG for spatial discretization and classical ODE integrators like Runge-Kutta, Adams-Bashforth or Backward-Differencing (BDF) for time integration. The other alternative is to use space-time elements.

Cockburn and Shu developed Runge-Kutta DG schemes (RKDG) in 1989 (Cockburn & Shu 1989) for scalar hyperbolic problems. In order to work, these methods usually require total-variation-diminishing (TVD) Runge-Kutta schemes and generalized slope limiters. Later, these results have been extended to hyperbolic systems in multiple dimensions (Cockburn & Shu 1991, Cockburn & Shu 1998).

⁵GetFEM++: <http://download.gna.org/getfem/html/homepage>, 12th of Dec. 2011

⁶DROPS: <http://www.igpm.rwth-aachen.de/DROPS>, 12th of Dec. 2011

⁷Νεκταρ, or NekTar: <http://www.cfm.brown.edu/crunch/nektar.html>, 12th of Dec. 2011

For an incompressible Navier-Stokes solver, usually a Poisson equation has to be solved to compute the pressure, resp. to enforce conservation of mass. Furthermore, also the treatment of the viscous terms requires some discretization of the Laplace operator. In 1982, Douglas Arnold developed the so-called Interior Penalty discretization (Arnold 1982), see §52 which is essential for the solver that is presented in chapter 5. Different discretization methods for the Laplacian, e.g. symmetric and unsymmetric Interior Penalty, Local DG, the Bassi-Rebay - method, etc. were put into a unified framework in (Arnold, Brezzi, Cockburn & Marini 2002), for which general numerical analysis was done.

The application of DG onto incompressible Navier-Stokes problems is a quite recent development; one should mention the works (Girault, Rivière & Wheeler 2004, Cockburn, Kanschat & Schötzau 2005), which aim on steady-state solutions. Unsteady solvers have been proposed by (Girault, Rivière & Wheeler 2005) and (Shahbazi, Fischer & Ethier 2007). Girault et. al. use a variant of the Projection scheme: At first, an intermediate velocity is computed from the evolution of the momentum equation without pressure. In a second step a pressure is computed and the velocity is corrected in order to fulfil continuity. They provide detailed analysis for stability and error convergence rates. Shahabazi et. al. instead, use the pressure that is known from the previous timestep within the predictor and compute a pressure difference in the corrector. The new pressure is then given as the sum of the old one and the computed pressure difference.

Multiphase problems and numerical methods. A multiphase flow may be characterized as a mixture of at least two immiscible fluids. These flows can be classified into those with material and non-material interface. For material interfaces there is no mass flux across the interface which separates both fluids. Examples for this setting may be air-water or oil-water mixtures. Mass flux across the interface, also called phase-change, is induced either by physical processes such as evaporation or chemical reactions like combustion.

These flows can be modeled by time- and space dependent, but piecewise constant physical properties, i.e. density and viscosity. If these quantities are discontinuous at the interface between the two fluids, the pressure and velocity field will contain at least kinks. For material interfaces, the introduction of additional models for surface tension will induce a jump in the pressure field. Solutions for non-material interfaces contain jumps in velocity and pressure field, even without any surface tension models.

For the numerical treatment of multiphase problems, basically two options are available. Either, the jumps and kinks are regularised (‘smeared out’) or a special numerical discretisation which is capable of representing jumps and kinks is used.

A well-established method that realizes the latter option is the Extended FEM (XFEM) method (Moës, Dolbow & Belytschko 1999). Its basic idea is to introduce additional nodes, located on the interface, into a Finite Element mesh. At these nodes, discontinuous basis functions are used to represent discontinuities. Originally, this method was proposed to model cracks in solid objects. Several authors have extended it to two-phase flows. Just to give a few example, we mention the works (Marchandise & Remacle 2006, Marchandise, Geuzaine, Chevaugéon & Remacle 2007) and (Esser, Grande & Reusken 2010, Groß & Reusken 2007) and (Chen, Minev & Nandakumar 2004).

Graphics Processing Units (GPU) for Scientific computing The term GPU was at first used for “GeForce 256” chip of NVIDIA cooperation, released in 1999⁸, one of the first 3D graphics accelerators which was not only capable of triangle rasterization, but also of transforming vertices and lighting calculations (T&L) in 3D projective geometry. So, these processors were capable of performing almost all OpenGL 1.0 functions⁹ in hardware. This was far from what is required in scientific computing, with the exception of realtime-3D graphics, of course. In approximately 2001, the fixed-function T&L was superseded by more flexible, programmable Vertex- and Pixel-Shader units¹⁰. Given programmability, it was possible to “misuse” these Graphic units to implement basic numerical algorithms, in single-precision (Bolz, Farmer, Grinspun & Schröder 2003, Krüger & Westermann 2003). Because of the rapidly fast hardware evolution, these works are outdated now; however, they paved the way for general purpose GPU’s, by demonstrating how the enormous compute power of GPU’s could be applied to problems outside of the 3D graphics domain. With the introduction of the technologies CUDA¹¹ and OpenCL¹² in 2008, and the introduction of double-precision GPU’s, GPU-computing became

⁸http://en.wikipedia.org/wiki/GeForce_256, 16th of Jan. 2012

⁹<http://en.wikipedia.org/wiki/OpenGL>, 16th of Jan. 2012

¹⁰<http://de.wikipedia.org/wiki/Vertex-Shader>, 16th of Jan. 2012

¹¹http://de.wikipedia.org/wiki/Compute_Unified_Device_Architecture, 16th of Jan. 2012

¹²<http://de.wikipedia.org/wiki/OpenCL>, 16th of Jan. 2012

ready for the kind of scientific computing that this thesis is focused on. It should not be forgotten: all of these developments are owed to the desire for better looking computer games!

The application of GPU's to discontinuous Galerkin methods seems to be very promising, since in DG the number of operations is high in comparison to the number of operands, and because the data structures are simple. The number of talks and publications on this topic is at least quite high; as an example we just mention (Klöckner, Warburton, Bridge & Hesthaven 2009). Within the incompressible fluid solver presented in this work, in terms of runtime, the Conjugate Gradient (CG) solver which is used to solve the Poisson equation in the projection method shows to be the dominating operation. So we focused on implementing an efficient CG on GPU clusters, i.e. multiple GPU's that are connected via an MPI network.

Structure of this work. Within chapter 2, a raw overview of the BoSSS framework and some of its aspects will be given. The software layout and some design considerations will be discussed.

A formal definition of the used DG discretization is given in chapter 3. This includes the definition of the numerical grid and a specification of fluxes and sources that are available in BoSSS. Additionally, mathematical properties of the MPI-parallelization are given.

A special chapter (chapter 4) is devoted to the details of GPU acceleration of sparse solvers.

An important application of the BoSSS framework is the incompressible Navier-Stokes solver presented in chapter 5. The solver is restricted to constant physical properties, i.e. constant density and viscosity in the whole domain, for all times and therefore called "single-phase" – Navier-Stokes – solver.

Within chapter 6, an extension of the classical DG method for discontinuous PDE's is given (Extended DG, XDG). Although this method is still in an early stage, on a long term basis it may be used to develop new, much more precise methods for computing two-phase flows, e.g. primary jet breakup or premixed combustion in an engine or turbine.

2 Software design

2.1 Introduction

The main purpose of the BoSSS framework is the treatment of systems of conservation laws, e.g.

$$\frac{\partial}{\partial t} u_\gamma + \operatorname{div}(\mathbf{f}_\gamma) + q_\gamma = 0, \quad 1 \leq \gamma \leq \Gamma$$

by means of the Discontinuous Galerkin (DG) method. The DG method was, according to Cockburn, first introduced by (Reed & Hill 1973); an overview about its development is given in (Cockburn et al. 2000), a comprehensive actual textbook is e.g. (Hesthaven & Warburton 2008). The abbreviation BoSSS (*Bounded Support Spectral Solver*) is a non-standard alias for Discontinuous Galerkin (DG) method: by the term “bounded support” we indicate that the mathematical support of the DG basis functions is bounded, or finite, while the term “spectral” indicates that it shares some properties of spectral methods.

In order to relate this manuscript to the code, we will give footnotes in the shape `Namespace.Class-Name[.Member]` that point to the implementation of the discussed aspect.

2.2 General issues

2.2.1 Modularization

From the very beginning of the BoSSS development, a key design goal was to create a multi-purpose code that may be useful even beyond the computational fluid dynamics domain.

Instead of creating a monolithic software package, BoSSS was designed as a collection of rather small (in terms of lines-of-code) software libraries. In order to organize these libraries they are arranged in 6 layers, described

No.	Abbrev. & short name	Issue, Description
4	L4 – Application	various BoSSS applications, e.g. incompressible Navier-Stokes solver NSE2b
3	L3 – Solution	Templates for L4, time discretization, control file parsing, plotting, various utilities;
2	L2 – Foundation	Spatial discretization: grid management, DG discretization, quadrature, Input/Output, Cut-Cell DG framework (see chapter 6)
1	L1 – Platform	Utilities, geometrical algorithms;
0	ilPSP	intermediate language Parallel Scientific Plattform: .NET wrappers for MPI, METIS & ParMETIS; Sparse solvers: own developments (“monkey” package, see chapter 4) and wrappers for 3 rd party packages (Hypre, Pardiso).
-1	native	3 rd party code, non-.NET code: MPI, BLAS, LAPACK, Hypre, Pardiso, METIS & ParMETIS, CUDA;

Table 2.1: Overview about the software layers of BoSSS; The boundary between C# – code and other code (C, FORTRAN, binary) is indicated between layers -1 and 0.

in tables 2.1 and 2.2. A specific software library may depend on everything that is found in lower layers, but not the other way around. The topmost layer, named “Application” layer contains specific solvers like the one for the incompressible Navier-Stokes equation that is discussed in chapter 5. Within each other layer, one “master” library as well as several “satellite” packages can be identified. While the master depends only on lower layers, the satellites may additionally depend on the master and on each other. Of course, no loop dependencies are allowed; this is already forbidden by the C# - compiler.

A user of the BoSSS code may design a specific solver for the problem he is interested in on top of the BoSSS libraries, within the application layer. At this point we should clarify that we distinguish between users, and end-users. From our point of view, a user is a programmer who works within

Layer	No. of Files	Lines of Code	Lines of Comment
L4 – NSE2b only	17	2'538	1'905
L3 – Solution	58	10'018	6'895
L2 – Foundation	75	44'285*	16'007
L1 – Platform	13	2'358	1'869
iPSP	122	16'391	11'445

Table 2.2: Actual (July 2011) code statistics; Note for * that this figure contains approximately 26'000 lines of auto-generated tables for DG basis polynomials and quadrature.

the Application layer to create his own solver. In contrast, an end-user uses one of the predefined solvers in the Application layer and does not do any programming himself.

2.2.2 .NET for scientific computing

C#, .NET and Mono. The foundation of the C# - language is the so-called .NET framework (Albahari & Albahari 2010) (pronounced “dot-net”), both released by the Microsoft Corporation in 2002. The .NET framework, or runtime-environment consists of a comprehensive software library (the class-library) and a virtual machine. Both, C# and .NET are EMCA – standards. A C# program is compiled to platform-independent intermediate code, the so-called *Common Intermediate Language* (CLI). The virtual machine translates this intermediate code into machine code when the user executes the program.

The Mono project¹ has created a C# compiler and a runtime framework that is compatible with EMCA standards. This means, any C# program that is compiled with the Mono C# - compiler will run under the Microsoft .NET framework, and vice-versa.

Garbage collection In programming languages like C/C++ or FORTRAN 90 with dynamic memory management, the programmer is in duty of releasing every chunk of memory that he allocates. This is not necessary in languages or environments that feature a so-called Garbage collector. Here, a background thread will stop the application from time to time,

¹Xamarin Inc., <http://www.mono-project.com>, 29th of Nov., 2011

and scans *all* allocated objects to determine whether they are still in use or not. The unused objects are released and the remaining ones may be compacted, see figure 2.1, to avoid memory fragmentation. The biggest advantage for the programmer is that he does not need to care about releasing unused objects, which in big software projects is an issue on its own; usually it is difficult to define and enforce a clear policy of determining responsibility for releasing objects.

Using legacy code, Native libraries In “.NET-speak”, any code that is written in traditional programming languages like C or FORTRAN, i.e. everything that is not .NET - code, is called *native* or *legacy code*, while C#- or other .NET - code is also referred to as *managed code*. Since .NET is not designed to be an island, it is possible to combine managed and native code; from the experience during the development of BoSSS, there are three reasons for using native code in .NET applications for supercomputers: First, the need for near-system libraries like MPI, which are not available as .NET libraries. Second, the use of complex mathematical libraries like sparse solvers (“legacy code”), which would produce a large overhead to re-implement them in C#. And third, the need for performance. As shown in table 2.3, a general double-precision matrix-matrix product (DGEMM) from an hardware-vendor optimized library like the ACML (AMD Core Math Library²) is superior, in terms of speed, by nearly a magnitude of 10 to non-optimized (“naive”) implementations in *any* programming language.

P/Invoke. The standard way of using native code within C# is called *P/Invoke*. Native code is compiled and linked into a shared library (*.dll, i.e. “dynamic linked library” - files in Windows, *.so, i.e. “shared object” - file on Unix). In the C#-code, a function prototype is defined together with an attribute that points to the shared library. The arguments of a function call have to be converted into a form that is usable for the native code. This is called *marshalling*. For several classes of objects, like arrays or value-types, P/Invoke is able to do marshalling automatically. The most important thing is to prevent the garbage collector (which may run at any time in its own thread) from moving the base address (as illustrated in figure 2.1, source is³) of some object while an unmanaged function accesses it. Such locking of objects for the garbage collector is referred to as *pinning*.

²<http://developer.amd.com/libraries/acml/downloads/pages/default.aspx>,
1st of August 2011

³http://mono-project.com/Generational_GC, 29th of Nov., 2011

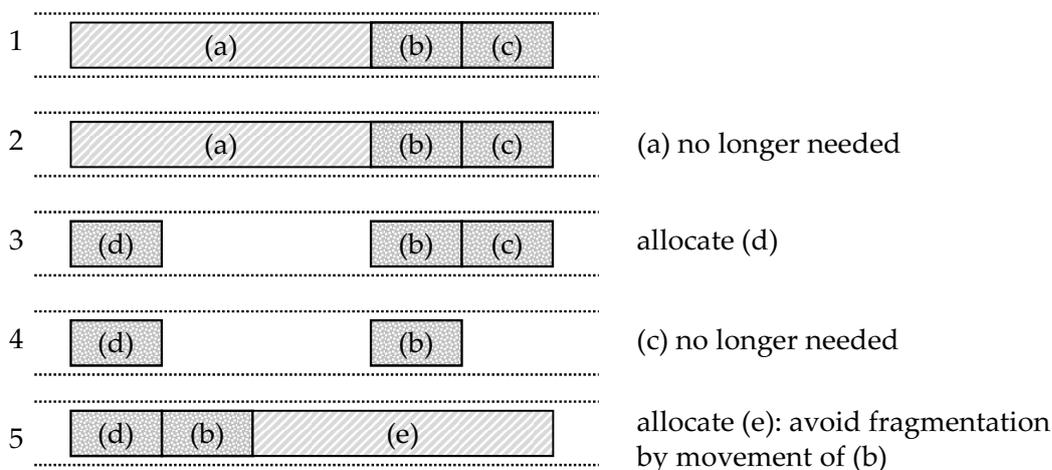


Figure 2.1: Memory reallocation by garbage collector

For standard function calls, which take some input data and process some output, pinning is performed automatically by P/Invoke.

Manual Pinning. Besides that, there are certain function calls which require that pinning continues for some time after they return - e.g. non-blocking MPI calls like `MPI_Irecv`. In .NET, it is relatively easy to control pinning manually (which is not possible in Java), by means of the `System.Runtime.InteropServices` - namespace. It is illustrated in figure 2.2, that the `GCHandle.Alloc(...)` - method can be used to acquire a lock on an object, while methods of the `Marshal` - class can be used to get the base address of the pinned array in memory. Furthermore the `Marshal` - class can be used to deal with string conversion (ANSI to Unicode and vice-versa), to allocate unmanaged memory, just to mention a few features.

Compile once – run everywhere. The use of .NET makes it possible that BoSSS is platform independent not only on a source-code level but even on a *binary* level. This means that once the .NET runtime, or on Unix systems the compatible Mono runtime⁴, and some native libraries, e.g. BLAS, MPI, LAPACK are installed on a cluster or supercomputer, the binaries from the development system, usually a local workstation, can be directly copied and executed on the supercomputer, without the need of recompiling the code.

⁴Xamarin Inc., <http://www.mono-project.com>, 29th of Nov., 2011

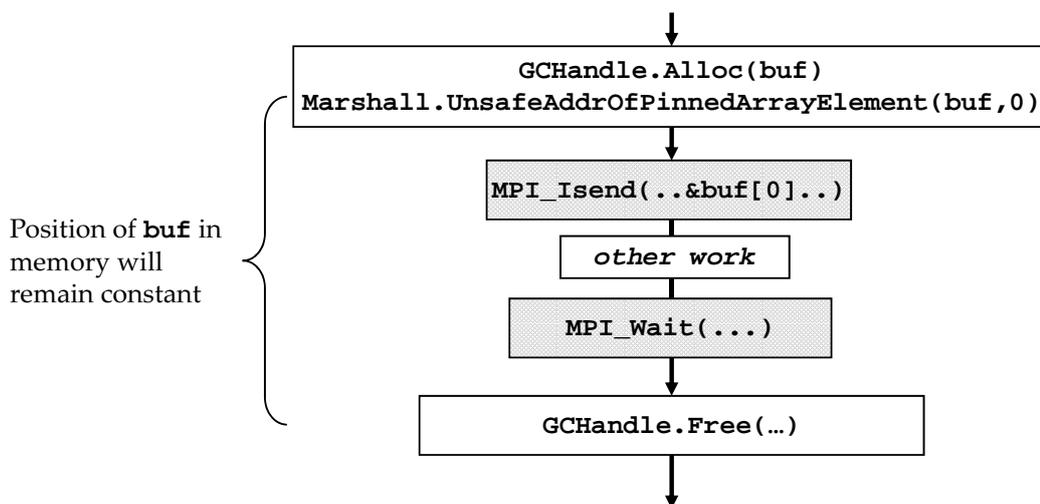


Figure 2.2: Manual pinning by `GCHandle.Alloc(...)` and `GCHandle.Free(...)`.

Portability. At this point, portability of .NET Programs to an arbitrary computer is a somewhat controversial issue on non- MS-Windows – platforms. There the portability depends on the availability of the Mono framework. In this context, we consider the Portable.NET - project ⁵, in terms of speed, stability, completeness of implementation and compatibility to other .NET - implementations being not ready for production yet. In the present situation, Mono is generally available for Linux on EM64T, AMD64 and IA-64 systems, which currently covers more than 91.4% of the Top 500 systems⁶. Here, we assume that most of these systems run Linux, which is true for 91% of the Top 500 systems.

Beside a low number (below 1%) of other architectures a rather big share of 9% of the Top500 – systems are based on the IBM Power architecture. We have no detailed information which operating system is used for that systems, but it seems a rather safe assumption that the majority of these systems are either IBM BlueGene systems, which run a very special edition of Linux, or IBM POWER{5,6,7} - systems which usually run AIX UNIX. For both of them, the principal answer to the question whether Mono compiles should be yes; Since Mono does not officially supports these platforms, we would expect a considerable high workload to do the port.

⁵www.gnu.org/software/dotgnu/pnet.html, 21st of Aug., 2011

⁶June 2011 edition of Top 500 systems, <http://top500.org/lists/2011/06>, 29th of Nov. 2011

Performance. It is a popular fallacy that C# - or Java - is not suitable for High Performance Computing, because it is an interpreted language. This is not the case and, in fact, not the experience with BoSSS. The output of the C#-compiler is so-called *Common Intermediate Language* - code (CLI) which looks rather similar to an assembler code, enhanced by additional metadata to represent object oriented structures. At load time, this CLI code is transformed to native-machine code. This last step is referred to as Just-In-Time - compilation (JIT).

Since the first JIT-versions of Java have been available, various authors have made performance comparisons between C and Java, e.g. (Java Grande Forum 1998) and (Bull, Smith, Pottage & Freeman 2001), just to mention two of them. These investigations concluded that Java, when properly used, is suitable for high performance computing. Since the concepts behind .NET and Java are very similar, one would expect to have similar results. A .NET - specific benchmark has been done in (Vogels 2003).

It seems that within recent years, authors have lost interest in doing excessive benchmarks. Indeed, in our opinion a general answer to the question: "how fast is C# in comparison to C or FORTRAN?" will be very difficult if not impossible to establish. Considering the release cycles of .NET, Mono, the classical compilers which compete against C#, and the release cycles of hardware, it is clear that any benchmark will be outdated soon. Therefore excessive benchmarking seems to be an ungrateful job. It is also unclear what a "fair" comparison really is. A line-by-line translation of a benchmark from one language into another would likely favour the original one. On the other hand, if both versions are optimized individually, it is at some point questionable whether it is still the same algorithm that is compared.

In the early development stages of BoSSS we performed some basic tests. The double-precision matrix-matrix product (DGEMM) and a prime number test, namely the Sieve of Eratosthenes, were implemented in C and C#. The DGEMM was implemented in a naive way, although it is a well-known fact that there are lots of optimization strategies. Our implementations of the operation ' $C = A \cdot B$ ' consists mainly of three loops; two of them to run over all rows i and all columns j of C , and the innermost one to compute the product $A_{i,-} \cdot B_{j,-}$. For the C# - versions, two variants have been tested, a rather unoptimized one and a second one which uses so-called unsafe code (see below) to speed up execution. Results are shown in table 2.3.

	DGEMM $N = 1000$	Sieve of Eratosthenes primes $< 2 \cdot 10^7$
C#, Debug, .NET 3.5	29.8 sec	39.6 sec
C#, Release, .NET 3.5	16.7 sec	37.6 sec
C#, Release, Mono 2.2	34.9 sec	38.9 sec
C#, unsafe code, .NET 3.5	9.8 sec	37.7 sec
C#, unsafe code, Mono 2.2	12.4 sec	39.5 sec
C, MinGW-gcc 3.4.5	16.8 sec	30.3 sec
C, MinGW-gcc 3.4.5 -O3	9.8 sec	20.9 sec
C, MS-cl 15.00.21022.08	15.5 sec	26.8 sec
C, MS-cl 15.00.21022.08 /O2	9.7 sec	37.6 sec
ACML-BLAS 4.1.0	1.28 sec	n.a.

Table 2.3: Runtime of (naively coded) DGEMM and the “Sieve of Eratosthenes” in managed and classical languages (System: Pentium 4 (Prescott), 3GHz, Windows XP service pack 3). Test performed in March 2009.

Unsafe code. In C#, the programmer has the possibility to declare specific sections of the code as *unsafe*. Within such unsafe sections, it is possible to use pointer arithmetics like in the C language, and therefore omitting the runtime security checks, such as array bounds - checking, that .NET performs for “safe” code.

The effect of unsafe optimizations is remarkable; as a benchmark problem, a C# - port⁷ of the Java - version of the LINPACK benchmark⁸ was used. Detailed results are presented in table 2.4. The original version of the C# - benchmark shows to be approximately 50% slower than the C - version⁹ of the LINPACK benchmark. By optimizing *just one subroutine* of LINPACK, namely the DAXPY operation ($y \leftarrow y + \alpha \cdot x$, for vectors x and y and a scalar α), the C# LINPACK is able to reach 87% of the performance of the C - version. Further improvements may be possible if other subroutines would be optimized as well. The code modifications are shown in figure 2.3. A rather impressive result is achieved by a variant¹⁰ of Mono that uses the “Low Level Virtual Machine” (LLVM, see (Lattner & Adve 2004)) for the JIT - process. While standard Mono, even under optimal conditions

⁷www.shudo.net/jit/perf/Linpack.cs, 17th of July 2011.

⁸www.netlib.org/benchmark/linpackjava, 17th of July 2011.

⁹www.netlib.org/benchmark/linpackc, 17th of July 2011.

¹⁰www.mono-project.com/Mono_LLVM, 17th of July 2011.

	LINPACK [kFlops]
C#, .NET 4.0	729
C#, mono 2.10.1	532
C#, .NET 4.0, unsafe	1'232
C#, mono 2.10.1 unsafe	855
C#, mono-llvm 2.10.1 unsafe	1'223
C, GCC 4.4.5 -O4	1'416

Table 2.4: LINPACK performance comparison, for a matrix of 500×500 entries of unsafe C# against managed code. (System: AMD Phenom II X4 940 3GHz, all tests on 64 Bit Windows 7 service pack 1, except the latest two on 64 Bit Debian GNU/Linux 6.0.2.1). Test performed in July 2011. Higher values indicate better performance.

is significantly slower than Microsoft .NET, the LLVM-version of Mono seems to match the performance of the Microsoft implementation.

Vectorization. In 2011, the era of vector computers seems to be over; additionally, there will very likely never be .NET - support for actual vector computers like the NEC SX9, nor from Microsoft .NET, neither from Mono. Yet, many BoSSS methods (aka. functions or subroutines) use the concept of vectorization for optimization issues. E.g. the evaluation of a DG field¹¹, i.e. the computation of the sum $\sum_{n=0}^{N-1} \phi_{j,n}(\xi_i) \cdot \tilde{u}_{j,n}$ (see §27) can be done for a series of L cells and K nodes (ξ_0, \dots, ξ_K) with one function call. Considering e.g. numerical integration (aka. quadrature, see §8) in multiple cells – a very important operation for a modal DG implementation – offers performance advantages and more opportunities for performance optimization than just a scalar approach:

- The overhead for a lot of function calls is avoided.
- The DG basis polynomials ϕ_n , which are very costly to evaluate, could be reused in every cell if the evaluation nodes ξ_i are equal in all cells.
- Several optimization techniques, like unsafe code or even porting certain parts to GPU are only effective if there are rather big loops which contain no function calls except language build-ins like sine or cosine.

¹¹BoSSS.Foundation.Field.Evaluate

```

void Daxpy(int n, double da, double[] dx, int dx_off, int incx, double[] dy, int dy_off, int incy) {
    int i, ix, iy;

    if ((n > 0) && (da != 0)) {
        if (incx != 1 || incy != 1) {
            // code for unequal increments or equal increments not equal to 1
            ix = 0; iy = 0;
            if (incx < 0) ix = (-n + 1) * incx;
            if (incy < 0) iy = (-n + 1) * incy;
            for (i = 0; i < n; i++) {
                dy[iy + dy_off] += da * dx[ix + dx_off];
                ix += incx;
                iy += incy;
            }
        } else {
            // code for both increments equal to 1
            for (i = 0; i < n; i++)
                dy[i + dy_off] += da * dx[i + dx_off];
        }
    }
}

-----

void Daxpy(int n, double da, double[] dx, int dx_off, int incx, double[] dy, int dy_off, int incy) {
    int i, ix, iy;

    unsafe { fixed (double* _pdx = &dx[0], _pdy = &dy[0]) {
        double* pdx = _pdx + dx_off;
        double* pdy = _pdy + dy_off;
        if ((n > 0) && (da != 0)) {
            if (incx != 1 || incy != 1) {
                // code for unequal increments or equal increments not equal to 1
                ix = 0; iy = 0;
                if (incx < 0) ix = (-n + 1) * incx;
                if (incy < 0) iy = (-n + 1) * incy;
                for (i = 0; i < n; i++) {
                    *pdy += da * *pdx;
                    pdy += incx;
                    pdx += incy;
                }
            } else {
                // code for both increments equal to 1
                for (i = 0; i < n; i++) {
                    pdy[i] += da * pdx[i];
                }
            }
        }
    }
}

```

Figure 2.3: Unsafe optimizations for the DAXPY operation in the LINPACK benchmark, which produce a significant performance increase for both, the Microsoft .NET and the Mono runtime. Above: original version, below: optimized version; modifications are marked *italic*.

- In some situations – like the example above – it is even possible to re-formulate two or three loops as a matrix-vector or matrix-matrix product. Level 2 or level 3 BLAS operations are usually much more efficient than hand-coded loops.

Conclusions on performance. From our own tests and benchmarks done by others we reason that...

- Optimized binary libraries from hardware vendors are usually superior to naive implementations built by any compiler.
- LINPACK and other scientific benchmarks show that classical languages are still faster, but not by much.
- The performance loss of using a managed language in the worst case is about 50%. Whether this is really observed in “real-world applications” remains uncertain.
- It is difficult to find proper benchmarks, because more complex programs cannot be translated one-by-one from C# to C or vice versa.
- Unsafe code performs very often head-to-head with state-of-the-art C compilers.
- Especially Mono seems to profit a lot from using unsafe code.
- It is not possible to give a short but overall valid answer on execution performance difference between managed and native code, because small benchmarks show that results may have a high dependence on small details.

Due to our experiences in the BoSSS development we recommend...

- By using .NET we may expect a performance loss in the range of 20 to 40%, in comparison to classical programming techniques; we consider this a “fair price” for the advantages we gain in software development.
- Data which performance-critical algorithms work on, should be organized in arrays of value types. Complex graphs of heap objects should be avoided.
- Performance-critical sections should be “vectorized”, i.e. implemented as multiple instruction-multiple data patterns. Loops should be preferred to multiple function calls, i.e. the stack should be kept shallow. This opens the possibility to optimize the code with unsafe sections, to use optimized BLAS in a large scale, or even to port these parts to GPU’s.

2.2.3 Flexibility versus implementation efficiency

A user of BoSSS – or any other grid-based CFD code – may find it helpful to have maximal flexibility when generating the grid. Thinkable are arbitrary hanging nodes, mixing different types of cells or also modifying the polynomial degree of the DG interpolation (see §27) from cell to cell.

Using a fully object-oriented representation of the grid and the DG fields, it might not have been too difficult to implement such features. However an approach like this has never seriously been considered, as we do not expect it to show decent performance.

Hence, for performance-critical data structures, object oriented design patterns were intentionally avoided. Instead, simple data structures like arrays are used; this yields a memory layout that is very similar to what e.g. a FORTRAN programmer may design.

Because of this desired simplicity, that is essential for a performant implementation, several tradeoffs on the “user wish list” were made. It is clear that if the parameters of the DG field evaluation, like quadrature nodes, change on a cell by cell - basis, it is not much what remains from vectorization and additionally coding will get more and more difficult. Therefore the grids in BoSSS are restricted to a single primitive type, and right now, the DG polynomial degree is constant in all cells.

For a different reason, non-Cartesian quadrilateral and hexahedral grids are omitted. It is clear that the transformation T from a trapezoidal cell to a quadratic reference element cannot be affine-linear (see §13). Therefore, the orthogonality between the basis polynomials ϕ_n (see §27) would be lost. For such cells K_j , an additional $N \times N$ matrix, where N is the number of DG polynomials, must be stored to transform the basis polynomials ϕ_n from the reference element to an orthonormal basis $\phi_{j,n}$ in the cell K_j .

2.3 Closer view on selected parts of the BoSSS-framework

2.3.1 Implementation of spatial operators and fluxes

Regarding spatial discretization, BoSSS is designed to give a user (i.e. a developer who works on Layer 4) full control over fluxes and Riemannians

(see §49), while freeing him from dealing with the technical issues of the discretization, such as cell indices, quadrature or MPI-parallelization.

The user defines operators in the sense of §42 resp. §43 by creating an instance of the SpatialOperator - class. The spatial operator represents a mapping from a list of domain variables¹² to a list of codomain variables¹³. Each codomain variable is defined as a sum of mathematical expressions (in BoSSS, they are called equation components¹⁴) which the user has to define. Therefore, he has to implement Riemannians, fluxes and sources in patterns that are defined by the equation-components-interfaces, i.e. the interfaces are derived from the IEquationComponet - interface. All available interfaces are presented in figure 2.4.

They may be classified in two ways; At first, one may distinct between nonlinear (¹⁵, ¹⁶, ¹⁷, ¹⁸;) and linear equation components (¹⁹, ²⁰, ²¹, ²², ²³;). The nonlinear part of a spatial operator can be evaluated directly. This means that the value of the codomain variables could be computed through ²⁴ or ²⁵ if the domain variables are given in form of DG fields. Linear components can be used to construct a matrix²⁶.

Another classification may be:

- *Fluxes*, i.e. discretizations of divergence - expressions: INonlinearFlux, INonlinearFluxEx and ILinearFlux.
- *Source terms*: INonlinearSource, ILinearDerivativeSource.
- *Penalitzation fluxes* can be used to ensure that a discretization is coercive, but have no representation in the continuous domain, see §93: ILinearDualValueFlux and IDualValueFlux.

¹²BoSSS.Foundation.SpatialOperator.DomainVar

¹³BoSSS.Foundation.SpatialOperator.CodomainVar

¹⁴BoSSS.Foundation.SpatialOperator.EquationComponents

¹⁵BoSSS.Foundation.INonlinearFlux

¹⁶BoSSS.Foundation.INonlinearFluxEx

¹⁷BoSSS.Foundation.INonlinearSource

¹⁸BoSSS.Foundation.IDualValueFlux

¹⁹BoSSS.Foundation.ILinearFlux

²⁰BoSSS.Foundation.ILinearSource

²¹BoSSS.Foundation.ILinearDerivativeSource

²²BoSSS.Foundation.ILinearDualValueFlux

²³BoSSS.Foundation.ILinear2ndDerivativeFlux

²⁴BoSSS.Foundation.SpatialOperator.Evaluate(...)

²⁵BoSSS.Foundation.SpatialOperator.Evaluator.Evaluate(...)

²⁶BoSSS.Foundation.SpatialOperator.ComputeMatrix(...)

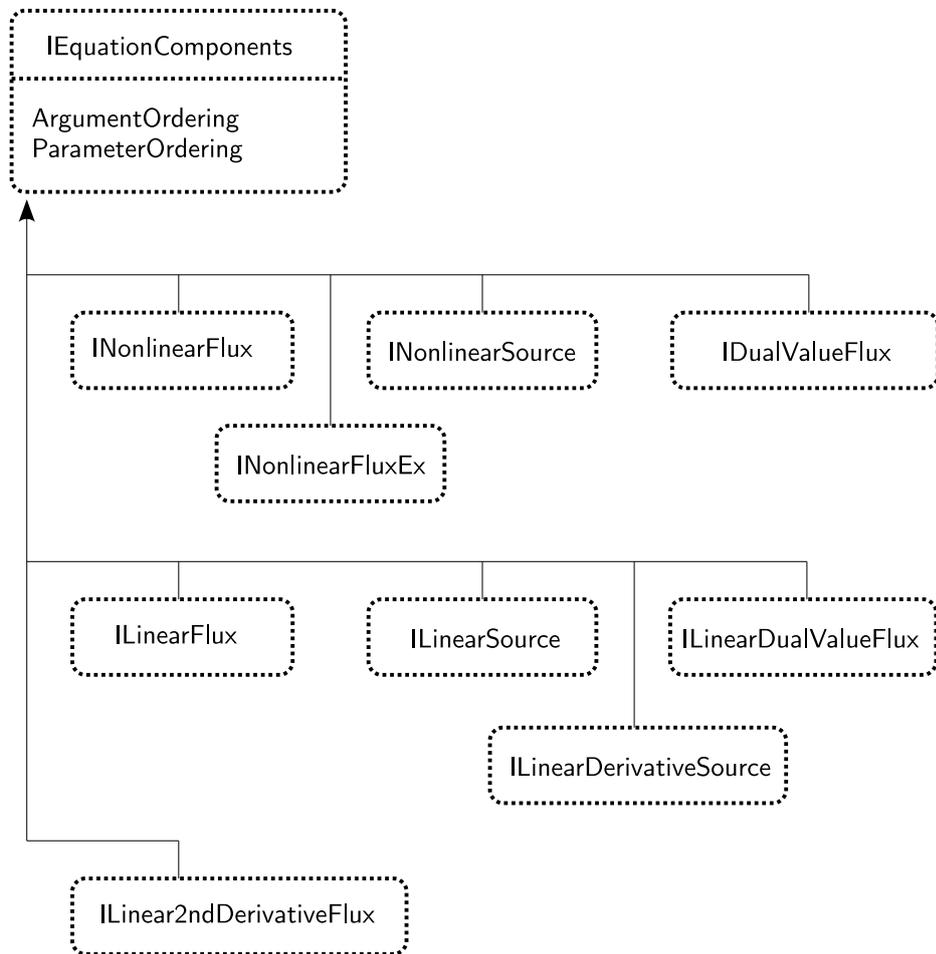


Figure 2.4: Available interfaces for creating equation components. Interface names are relative to the BoSSS.Foundation - namespace. The basic definition of a equation component is the IEquationComponents - interface; Derived from this are linear and nonlinear versions of fluxes, sources and dual-value fluxes. INonlinearFluxEx and ILinearDerivativeSource yet have no linear resp. nonlinear counterpart. A special component, mostly used for interior penalty discretisation of the Δ - operator, is ILinear2ndDerivativeFlux.

- *Second derivatives*, to discretize e.g. the Δ - operator: `ILinear2ndDerivativeFlux`
- *Local derivatives* just depend on the derivatives of the DG polynomial representation within each cell, but are not effected by the jumps at the cell boundaries: `ILinearDerivativeSource`. (Remark: it is not necessary to have a nonlinear counterpart for this, since that could be realized by computing a local derivative²⁷ and using a normal source.)

Additional to the list of domain- and codomain variables, spatial operators also allow to specify *parameter variables*²⁸. Their role is similar to those of the domain variables, and for nonlinear fluxes, there is no difference. However, for *linear components*, they can be used to define some space-dependency of the operator in the form of DG-fields. This may be useful e.g. if the linear operator is a linearisation of some nonlinear operator or for providing boundary conditions that are calculated in the code.

2.3.2 Class-structure of DG fields

All objects to store DG discretizations are instances of classes derived from the `Field`²⁹ - class. This includes both, the standard (or Single-Phase, in contrast to the Multi-Phase, extended DG) DG discretization³⁰ as well as the extended (XDG) discretization³¹. These classes and their relations to basis-objects³² are shown in figure 2.5.

For the extended DG framework, the relation to the Level Set tracker³³, the extended basis³⁴ and the Level Set class³⁵ are shown: to instantiate an extended DG field, one has to instantiate an extended basis, therefore he has to instantiate a Level Set tracker and therefore he has to instantiate at least one Level Set object. The purpose of the Level Set tracker is to identify those cells that are cut by the Level Set, those near to the Level Set and those far away (see §124).

²⁷`BoSSS.Foundation.Field.Derivative(...)`

²⁸`BoSSS.Foundation.SpatialOperator.ParameterVar`

²⁹`BoSSS.Foundation.Field`

³⁰`BoSSS.Foundation.SinglePhaseField`

³¹`BoSSS.Foundation.XDG.XDGField`

³²`BoSSS.Foundation.Basis`

³³`BoSSS.Foundation.XDG.LevelSetTracker`

³⁴`BoSSS.Foundation.XDG.XDGBasis`

³⁵`BoSSS.Foundation.XDG.LevelSet`

To represent vector-valued properties, the vector field container³⁶ may be used.

The DG field class provides two out-of-the-box methods for computing derivatives. The first one³⁷ computes the exact derivative of the DG polynomial cell by cell, while the second one³⁸ is based on a central - difference Riemannian. In a numerical algorithm, both of these methods should be used with care, because they feature no penalization of large jumps in between the cells. This means that their discretization is not coercive, see §53 but they may be in particular useful for post processing. The central difference - version features hard-coded homogeneous Neumann boundary conditions. For simplicity of use, there are no means to alter the behaviour of these derivative operations. Furthermore, such options would introduce redundancy to the spatial operator framework.

Based on the predefined derivatives, “composite operators” like divergence^{39, 40}, gradient^{41, 42}, the Laplacian^{43, 44}, 2D curl^{45, 46} and 3D curl^{47, 48} are defined.

2.3.3 BoSSS database

“How to document CFD results?”, may be a question that several researchers or CFD engineers ask themselves. Scientists have to summarize their results in scientific articles, or thesis works, but besides that, almost everyone who does large-scale numerics has to do some internal book-keeping of his solver runs. A typical beginner mistake is that someone just archives the plot-output (e.g. data files for a software like Tecplot) of a simulation. Within half a year or so, he probably will not remember how the results were achieved and he will not be able to reproduce the

³⁶BoSSS.Foundation.VectorField

³⁷BoSSS.Foundation.Field.Derivative

³⁸BoSSS.Foundation.Field.DerivativeByFlux

³⁹BoSSS.Foundation.Field.Divergence

⁴⁰BoSSS.Foundation.Field.DivergenceByFlux

⁴¹BoSSS.Foundation.VectorField.Gradient

⁴²BoSSS.Foundation.VectorField.GradientByFlux

⁴³BoSSS.Foundation.Field.Laplacian

⁴⁴BoSSS.Foundation.Field.LaplacianByFlux

⁴⁵BoSSS.Foundation.Field.Curl2D

⁴⁶BoSSS.Foundation.Field.Curl2DByFlux

⁴⁷BoSSS.Foundation.VectorField.Curl3D

⁴⁸BoSSS.Foundation.VectorField.Curl3DByFlux

results. Striving to archive as much information as possible about a specific solver, one has to put a lot of effort into the organisation of this data. I.e. one has to develop a directory structure, copy log-files, input files, and maybe additional info together, edit additional information, etc.. The BoSSS database is designed to overcome that burden.

Therefore, the BoSSS database is designed to log everything the application knows about itself, each time the application is started. Every instantiation of a BoSSS application is called a *session*. The information logged for each session include, but is not limited to:

- a copy of input files, called the control file.
- copies of the standard-output and -error streams, for each process.
- tracing information: which method (a.k.a. function aka. subroutine) was entered at which time, how long the execution of this method took; of course, this logging is limited to the 'major' - subroutines and the amount of logging could be adjusted by the user. Additionally, the trace files also contain information about the memory usage at certain points in the lifetime of an application instance.
- Execution information: date of run, name of the computer system on which the application has been executed, etc.

Parallel IO. The second major design criterion is IO parallelism. A standard solution may be to use parallel MPI IO, which allows multiple processes to access a single file. If a standard file system like NTFS (Microsoft Windows) or ext4 (Linux) is used, the degree of parallelism is then only limited by implementation of the file system and the computer hardware itself. It is obvious that, as long as a hard-disk is just connected by one cable, has got one read/write head, and as long as a file server has just one or two network interfaces, there must be some serialisation of input/output operations. These issues usually do not affect the programmer as long as he does not care that parallel I/O – operations may be serialized by the operating system.

Together with a parallel file system like Lustre⁴⁹ which cleverly combines many file servers and presents them as one file system, the use of MPI IO gives full parallelism, where multiple processes write to one file simultaneously. Finally, this file may be spread among multiple physical servers, but is presented as one logical file in the file system.

⁴⁹wiki.lustre.org, 15th of July 2011.

In BoSSS however, we – again – chose another solution. Unlike with parallel MPI IO, where multiple processes write into *one* file, in BoSSS each process writes *its own* file. Instead of running a parallel file system, this approach allows to balance input/output operations to multiple independent standard file servers.

The BoSSS database in comparison to “real” Database systems. As there are a lot of general-purpose, well-working and established database systems in existence, one has to justify the development of a customized system.

At first, one has to consider that BoSSS is designed to run on supercomputers and clusters. It will be almost impossible to convince the administrators of such systems to install and run a parallel database server on their system. Of course, they have good reasons for not doing this. This cancels out all database systems with client/server architecture.

The remaining systems, like SQLite⁵⁰ are not designed to run in parallel and to process several hundred Gigabytes of data. So, they could be used for only a fraction of the data that should be stored in a BoSSS database. Because of this the advantages of using a general-purpose database seem to be very little and that they are not worth paying the price (e.g. dependency on another library, maybe not even written in C#) that the use of such a system would introduce.

The BoSSS database in comparison to HDF5. Another technology someone may consider is HDF5 file format⁵¹. It allows to store objects like matrices and vectors and to organize these objects in a tree-structure that is similar to the directory structure of a file system. For classical C or FORTRAN codes, this is a big improvement. These languages – to be precise, their runtime libraries – offer no standardized way of storing the meta-information, like matrix dimension, datatype (single- or double precision float, integer, ...) or endianness⁵² together with the object itself. HDF5 does this job.

But regarding C#, the .NET - runtime library has a build-in feature called *serialisation* (see (Albahari & Albahari 2010), page 609) that is, in some

⁵⁰www.sqlite.org, 15th of July 2011.

⁵¹www.hdfgroup.org/HDF5, 15th of July 2011.

⁵²en.wikipedia.org/wiki/Endianness, 4th of August

sense, more advanced than HDF5. With .NET – serialisation, an almost arbitrary graph – not just a tree – of objects could be written to (“serialized”) or read from (“de-serialized”) from a data stream. Both operations could be coded with a single instruction and are therefore easier to use than the mid-level HDF5 - API (Application Program Interface). HDF5, on the other hand is specially suitable for large amounts of data; while with .NET - serialization the whole object graph must be de-serialized, i.e. loaded into memory to access a specific object, HDF5 allows random access to a specific node of the tree. For these reasons, HDF5 allows to handle much more data *in a single file* than .NET - serialization. By using multiple files for each timestep of the simulation, we overcome this issue.

Another mission of HDF5 is to improve data exchange between different codes and systems. This does not really apply to BoSSS, for two reasons: At first, HDF5 does not define a standard for unstructured grids. This is done by specifications like CGNS (see ⁵³) that build on top of HDF5 and are even more complicated. Second, because of the modal DG representation that is used in BoSSS (see §27) the DG coordinates of BoSSS are of no practical use for other applications, since they do not know the specific choice of the polynomial basis. On the other hand, a *loss-less* transformation of the DG polynomials into some *nodal* data would have been very cumbersome at the time of database implementation.

Furthermore, the HDF5 - library is written in C, and therefore one would have to install or compile it for every system which he or she wants to execute BoSSS on. Clearly this would compromise the “compile locally – run everywhere” philosophy of BoSSS. Additionally, at the time when the BoSSS database was designed and implemented no C# - binding for HDF5 was available.

Implementation. A BoSSS database is a quite simple directory structure:

```

$DB_ROOT
|
|-- data
|   |
|   |-- headers
|-- fields
|-- grids
|-- sessions

```

(2.1)

⁵³cgns.sourceforge.net

The naming and position of the database root directory `$DB_ROOT` within the file system is arbitrary. All objects in the database, i.e. files and directories, are identified by so-called *Global Unique Identifiers*, abbreviated as GUID or Guid, see (Albahari & Albahari 2010), page 245. These Guids are 128 Bit numbers that are generated by a special random number algorithm that ensures a negligible probability that the same number occurs more than once among all computers on earth. The file names of the database objects are hexadecimal notations of the 128 Bit number in the format `XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX`, e.g. `CAE7AD90-F4E9-4E3E-B303-E07E494BA9CF`.

Two different concepts of objects can be found in the database: data and metadata. Data objects/files are located in the `$DB_ROOT/data` - directory and contain – potentially bulky – vectors to store e.g. the grid or a DG field. Data objects may be stripped among multiple files, i.e. stored parallel. An object that is stripped into S parts, is stored in S files with names “*Guid.NofS.data*”, with $1 \leq N \leq S$.

Metadata objects/files are much smaller than data files and their size is independent from the size of the numerical problem. They contain descriptive information about how and which data objects form a numerical solution which was computed in a BoSSS session. Metadata objects are not stored in parallel – there is no need (because of their size) and there would be no benefit (because every MPI process needs the whole metadata information) from doing so.

For metadata objects, three different types may be distinguished: At the top level, there are the session objects. As mentioned above, each session corresponds to one run of the solver. The only mandatory information stored in a session is which DG-fields have been written in the corresponding solver run (`FieldStateLog2.txt`). Further optional or informational data stored in the session are copies of standard-output and -error streams, various logfiles, etc.; see above. In contrast to other metadata objects – that are stored as single files – the session object is realized as a directory, named after the sessions Guid and located in the `$DB_ROOT/session` directory. It contains various files, e.g. the already mentioned one for logging which DG fields have been written or `stdout.0.txt` that describe the properties of the session.

At the intermediate level, there are DG fields and grids, located in the `$DB_ROOT/fields` and `$DB_ROOT/grids` - directories. These objects may reference each other, e.g. every field refers to the grid which it has been declared for.

The lowest level of metadata objects are the headers for the data objects, located in the `$DB_ROOT/data/headers` directory. They only contain basic information about the data objects, most important how it is partitioned i.e. which index range of the data vector is in which file/part. Without knowing this, each MPI process would have to load *all* parts of a data vector to extract the range that it needs.

As already indicated, metadata objects refer to each other, but only from the top to the intermediate to the lowest level, and within the intermediate level. In particular, the session refers to fields (top to intermediate) and the fields refer to grids (within intermediate). Fields and grids themselves refer to the data headers that contain their vector data, e.g. DG coordinates or grid cells (intermediate to bottom).

Parallelism. There are two possibilities to realize parallel IO by the BoSSS database:

- If a parallel filesystem is available, the directory structure 2.1 may be created within the parallel filesystem and no further action is required to perform IO operations in parallel.
- Multiple instances of the directory structure 2.1 may be created on multiple file servers, usually one instance per server, and BoSSS balances the IO among them. In such cases, the full database is defined as the union over all database root directories.

Regarding the latter case, a few things should be mentioned. The implementation of the BoSSS database guarantees that no filename is used more than once over all database roots. Therefore, the union over all database roots is well defined.

In action, write operations of each MPI process of a BoSSS application are directed to the nearest (in terms of latency and bandwidth) file server.

For read operations, the situation is more difficult, since it cannot be assumed that a file required by a specific MPI process is located on the nearest file server. Therefore, in the worst case, all database roots have to be searched for a specific file. For future implementations this may be optimized by some journal file, that contains information about which file is stored at which database root. However, this process can only work if all database roots are accessible for all MPI processes; on Unix/Linux systems this may be achieved by mounting all file servers in a common directory tree; on Windows systems a good practice would be to make all file servers accessible under the same UNC namespace.

Database exploration and data visualization. While the database is constructed to file every information available, it may not be considered very end-user friendly. Especially the filenames like “15F9A3D2-EA67-4DCE-8755-AA78B8959275” that are generated from the Guid numbers, may be inconvenient for the end-user. To overcome this problem a database exploration software (DBE) with graphical user interface has been developed by Jochen Kling, a student who worked at the chair of fluid dynamics from June 2009 to September 2011. A screenshot is shown in figure 2.6. The DBE can be used to convert the data produced by BoSSS into common file formats like CGNS or Tecplot.

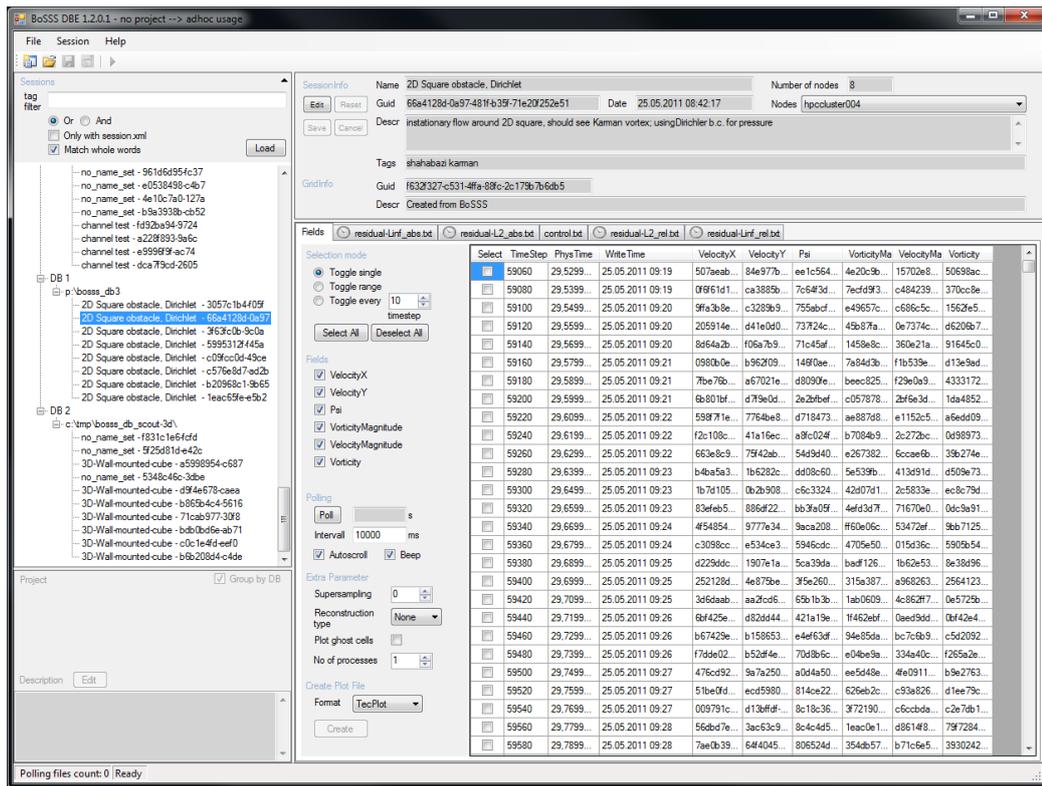


Figure 2.6: Screenshot of the BoSSS database explorer, programmed by Jochen Kling. In the left part of the window, a list of all sessions, in all databases is given. The right pane shows information about a specific session. This includes general information like name and description, computer name (of the system on which the session was computed), date and – most important – a list of all saved timesteps. These can be exported, partly or as a whole, to formats like Tecplot or CGNS.

3 Formal definition and properties of the BoSSS – framework

The topic of this chapter is to give a definition of the Discontinuous Galerkin method that is close to the BoSSS implementation. Therefore, also some rather technical definitions that have a direct representation in the code but which are typically not found in classical textbooks, will be given. One example for this is maybe the coordinate mapping (§32). Definitions like this are not necessarily important for understanding the principles of the DG method. Though, they are important concepts in the BoSSS code.

3.1 General definitions and notation

§1: Notation - Spatial dimension¹: Will be denoted as $D \in \mathbb{N}_{>0}$ in below;

§2: Notation - Vectors, Matrices and Indices: Let \mathbf{v} be an arbitrary vector of length L ; the i -th entry of \mathbf{v} may be denoted a $[\mathbf{v}]_i$. Consequently, the (i, j) -th entry of a matrix M may be denoted as $[M]_{i,j}$.

Furthermore, $[a_i]_{i=1,\dots,L}$ denotes a column-vector of length L , and $[B_{i,j}]_{\substack{i=1,\dots,I \\ j=1,\dots,J}}$ denotes an $I \times J$ - matrix.

For a vector $\mathbf{a} = (a_0, a_1, \dots, a_{L-1})$ of length L and a subset $I = \{i_1, \dots, i_N\} \subset \{0, \dots, L-1\}$ of the vector index set, with $i_1 < \dots < i_N$ $[\mathbf{a}]_I = (a_{i_1}, \dots, a_{i_N})$ denotes a sub-vector of \mathbf{a} .

§3: Notation - standard symbols: Within this text, the following symbols are used:

- the unit sphere $\mathcal{S}^D = \{\mathbf{x} \in \mathbb{R}^D; |\mathbf{x}|_2 = 1\}$

¹BoSSS.Foundation.Grid.Simplex.SpatialDimension

- the symmetric group

$$S^n = \{\tau : \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}; \tau \text{ is bijective}\},$$

i.e. the group of all permutations of length n ;

- the Hilbert space $L^2(\Omega) := \{f : \Omega \rightarrow \mathbb{R}; \int_{\Omega} f^2 dx < \infty\}$
- the Sobolev space $H^l(\Omega) := \left\{f \in L^2(\Omega); \sum_{|\alpha| \leq l} \|\partial^\alpha f\|_2^2 < \infty\right\}$,
where α denotes a multindex.

§4: Notation - D - dimensional measure: of a set $X \subset \Omega$ will be denoted as $\text{Vol}_D(X) := \int_X 1 dx$.

§5: Notation - characteristic function: of a set $X \subset \Omega$ will be denoted as

$$\mathbf{1}_X(\mathbf{x}) := \begin{cases} 1 & \mathbf{x} \in X \\ 0 & \text{otherwise} \end{cases}.$$

§6: Notation - inner product: The inner product in a domain $K \subset \mathbb{R}^D$, for $f, g \in L^2(K)$ will be denoted and is defined (in BoSSS) by

$$\langle f, g \rangle_K := \int_K f(\mathbf{x}) \cdot g(\mathbf{x}) dx.$$

§7: Definition - Convex Hull: For a given list of vertices, $(\mathbf{v}_1, \dots, \mathbf{v}_n) \in \mathbb{R}^{D \times n}$ the (D -dimensional) convex hull is defined as the set

$$\left\{ \sum_{i=1}^n \alpha_i \cdot \mathbf{v}_i \in \mathbb{R}^D, \text{ where } \sum_{i=1}^n \alpha_i \leq 1 \right\} =: CH(\mathbf{v}_1, \dots, \mathbf{v}_n) \subset \mathbb{R}^D$$

§8: Definition - Quadrature rule: A quadrature rule of order p_{ord} with k Nodes within a domain $K \subset \mathbb{R}^D$ is a Tuple

$$\left((\mathbf{y}_i)_{i=0, \dots, k-1}, (\beta_i)_{i=0, \dots, k-1} \right) \in \mathbb{R}^{D \times k} \times \mathbb{R}^k$$

for which the equation

$$\int_{\mathbf{x} \in K} p(\mathbf{x}) dx = \sum_{i=0}^{k-1} p(\mathbf{y}_i) \cdot \beta_i$$

holds for all polynomials p with $\text{deg}(p) \leq p_{\text{ord}}$. \mathbf{y}_i are called the nodes and β_i are called the weights of the quadrature rule.

3.2 DG approximation of fields on numerical grids

3.2.1 The numerical grid

It was an early design decision in the BoSSS development to introduce some restrictions on grids. It was decided that a grid should only consist of one type of cells, e.g. only of triangles. At first this keeps data structures simple. Second, for efficient vectorization, it is essential to be able to do the same operation – whatever this may be – in all cells.

§9: Definition - reference element: a reference element² (of dimension D) is characterized as an open set $K_{\text{ref}} \subset \mathbb{R}^D$, that is the interior of a convex hull of the reference element vertices³ $(\mathbf{w}_0, \dots, \mathbf{w}_{L-1}) \in \mathbb{R}^{D \times L}$, i.e. $K_{\text{ref}} := CH(\mathbf{w}_0, \dots, \mathbf{w}_{L-1}) \setminus \partial CH(\mathbf{w}_0, \dots, \mathbf{w}_{L-1})$.

An *edges-space* $F \subset \mathbb{R}^{D-1}$ of K_{ref} is characterized as an $(D - 1)$ – dimensional affine-linear manifold on ∂K_{ref} . An edge of the reference element, $\epsilon := F \cap \partial K_{\text{ref}}$, itself is a $(D - 1)$ dimensional convex hull.

The reference elements in BoSSS are coded in the Simplex-class. In addition to the mathematically essential properties in §9, a BoSSS-simplex provides some additional properties and has to follow a few conventions, which are laid out in subsequence.

§10: Remark - additional properties of the reference element implementation (aka. “simplex”⁴): All implemented reference elements fulfill the following properties:

- The center of K_{ref} is 0.
- For $D > 0$, $(\mathbf{w}_1 - \mathbf{w}_0, \dots, \mathbf{w}_D - \mathbf{w}_0)$ is a Basis of \mathbb{R}^D .

Additionally, in BoSSS further objects are associated with the reference element:

- A list of polynomials⁵, $(\phi_0(\mathbf{x}), \phi_1(\mathbf{x}), \dots)$, which are pair-wise orthonormal (i.e. $\langle \phi_n, \phi_m \rangle_{K_{\text{ref}}} = \delta_{n,m}$). For some $p \in \mathbb{N}$ there is an N so that $(\phi_0, \dots, \phi_{N-1})$ form a complete (orthonormal) basis of the

²BoSSS.Foundation.Grid.Simplex

³BoSSS.Foundation.Grid.Simplex.Vertices

⁴BoSSS.Foundation.Grid.Simplex

⁵BoSSS.Foundation.Basis.Polynomials

space of all polynomials of degree smaller or equal to p . This list is sorted by the polynomial degree, i.e. $\deg(\phi_0) = 0$ and for $m > n$ the relation $\deg(\phi_m) \geq \deg(\phi_n)$ holds.

- A set of quadrature rules⁶
- A $(D - 1)$ dimensional edge reference element⁷ $K_{\text{ref,edg}}$; to make this recursive definition complete, the 0-th dimensional the reference element is defined as $\{0\}$.
- an ordered, minimal list of edge transformations⁸, $Te_e : \mathbb{R}^{D-1} \rightarrow \mathbb{R}^D$ for $e \in \{0, \dots, E - 1\}$ which are affine-linear, for which

$$\bigcup_e \overline{Te_e(K_{\text{ref,edg}})} = \partial K_{\text{ref}}$$

holds. This list induces an ordering of all edges, i.e. $\epsilon_e := Te_e(K_{\text{ref,edg}})$.

§11: Notation - Reference coordinates: Coordinates within the domain of a simplex are called reference coordinates and will be denoted by $\xi := (\xi, \eta) \in \mathbb{R}^2$ or $\xi := (\xi, \eta, \nu) \in \mathbb{R}^3$ in subsequence.

§12: Remark : BoSSS offers the following simplices: Point⁹, Line¹⁰, Square¹¹, Triangle¹², Cube¹³, and Tetrahedron¹⁴; The individual properties of these simplices can be found in the source code or read out from the binaries.

§13: Definition - Grid: A grid with J cells for DG methods for the open, simply connected domain $\Omega \subset \mathbb{R}^D$, and a simplex $K_{\text{ref}} \subset \mathbb{R}^D$ is defined by an ordered list of affine-linear mappings

$$T_j : \mathbb{R}^D \rightarrow \mathbb{R}^D, \quad \text{for } 0 \leq j < J$$

that induce the sets $K_j := T_j(K_{\text{ref}})$, which are called the *cells* of the grid, so that the following properties are fulfilled:

⁶BoSSS.Foundation.Grid.Simplex.GetQuadratureRule(...)

⁷BoSSS.Foundation.Grid.Simplex.EdgeSimplex

⁸BoSSS.Foundation.Grid.Simplex.TransformEdgeCoordinates(...)

⁹BoSSS.Foundation.Grid.Point

¹⁰BoSSS.Foundation.Grid.Line

¹¹BoSSS.Foundation.Grid.Square

¹²BoSSS.Foundation.Grid.Triangle

¹³BoSSS.Foundation.Grid.Cube

¹⁴BoSSS.Foundation.Grid.Tetra

- $\bar{\Omega} = \bigcup_j \bar{K}_j$,
- for $j \neq k$, the D – dimensional measure of $\bar{K}_j \cap \bar{K}_k$ is zero
- for $j \neq k$, the $D - 1$ – dimensional measure of $\bar{K}_j \cap \bar{K}_k$ is zero, or $\bar{K}_j \cap \bar{K}_k$ is a common edge of K_j and K_k , i.e. there exists an edge ϵ_e in the reference element so that $T_j^{-1}(\bar{K}_j \cap \bar{K}_k) = \bar{\epsilon}_e$.

§14: Notation : The set of all cells of the grid is substituted by

$$\mathfrak{K} := \{K_0, \dots, K_{J-1}\}.$$

BoSSS, like most other CFD codes, usually does not use the computational grid in the order that is provided by the grid generation tool. Consequently, on every start of the application, a permutation of the grid cells may be chosen, e.g. due to a different number of MPI processes.

This has consequences for a restart of a solver, i.e. a BoSSS-application: A solver saves its data to hard disk, and the process is killed afterwards or just terminates. Then it is started again, maybe with a different grid ordering but should continue with data of the previous run.

In such a case it is necessary to know how the grid and some data vector (v_0, \dots, v_{J-1}) that is assigned to the grid, was permuted in between the runs.

To resolve these issues, BoSSS stores, for each run (i.e. each session), the permutation against the original cell numbering that was specified by the grid generation tool.

§15: Definition - Global Identification: The global identification (“GlobalId”) is a bijective mapping

$$\mathfrak{K} \ni K \mapsto id(K) \in \{0, \dots, J - 1\},$$

which assigns the number id to cell K .

The GlobalId – permutation¹⁵ $\tau \in S^J$ stores the GlobalId - for the current grid permutation, i.e.

$$\tau(j) = id(K_j).$$

¹⁵BoSSS.Foundation.GridCommons.GlobalID

§16: Definition and Remark - Invariance of GlobalId: Some cell K_j with index j in one solver instance with GlobalId - permutation τ is equal to cell K_l , with index l in an other solver instance with GlobalId - permutation σ , if, and only if $\tau(j) = \sigma(l)$, i.e. if their GlobalId's are equal.

§17: Notation - General linear group: The group of all real non-singular $D \times D$ - matrices is denoted as $Gl_D(\mathbb{R})$.

§18: Remark : The affine-linear transformation from (13) can be written as

$$T_j(\xi) = \mathbf{T}_j \cdot \xi + \mathbf{a}_j,$$

with the matrix $\mathbf{T}_j \in Gl_D(\mathbb{R})$ and the vector $\mathbf{a}_j \in \mathbb{R}^D$. This very trivial fact is mentioned here to note that, in BoSSS, \mathbf{T}_j is found at¹⁶, \mathbf{a}_j is found at¹⁷, \mathbf{T}_j^{-1} is found at¹⁸, and the transformation T_j can be computed with¹⁹. The total number of cells in a grid, J , is found at²⁰.

§19: Notation : To distinct between points/vectors in K_{ref} and Ω , elements of K_{ref} are denoted as $\xi = (\xi, \eta) = (\xi_0, \xi_1)$ or $\xi = (\xi, \eta, \nu) = (\xi_0, \xi_1, \xi_2)$, while elements of Ω are written as $\mathbf{x} = (x, y) = (x_0, x_1)$ or $\mathbf{x} = (x, y, z) = (x_0, x_1, x_2)$. For $\mathbf{x} \in K_j$, they transform like

$$\begin{array}{ccc} & T_j & \\ & \rightleftharpoons & \\ K_{\text{ref}} & & K_j \\ & T_j^{-1} & \\ T_j^{-1}(\mathbf{x}) = \xi & \leftrightarrow & \mathbf{x} = T_j(\xi). \end{array}$$

§20: Remark : In BoSSS, for reasons of simplicity and software optimizations, a grid consists only of one type of cells, e.g. it is not possible to mix triangle elements with square elements.

§21: Definition - Vertices of a cell: The elements of the set $\{\mathbf{v}_1, \dots, \mathbf{v}_L\} \subset \mathbb{R}^D$, for which $CH(\mathbf{v}_1, \dots, \mathbf{v}_L) = \overline{K_j}$ holds, are called the vertices of the cell K_j if and only if the set is minimal.

¹⁶BoSSS.Foundation.Grid.GridData.Transformation

¹⁷BoSSS.Foundation.Grid.GridData.AffineOffset

¹⁸BoSSS.Foundation.Grid.GridData.InverseTransformation

¹⁹BoSSS.Foundation.Grid.GridData.TransformLocal2Global(...)

²⁰BoSSS.Foundation.Grid.GridData.GlobalNoOfCells

§22: Remark : For (modal) DG methods, the author recommends to specify the transformation between the simplex and the cells explicitly. By knowing the set of simplex vertices and cell vertices, this transformation is uniquely defined up to the symmetries of the simplex. In a finite volume method, where values are constant within a cell, this choice may not matter. Hence in a DG method, properties depend on the choice of T_j . Therefore, in the specification of a BoSSS grid, the cell vertices are given as an ordered list ²¹.

§23: Definition - Edges of the grid: The set of all edges is defined by

$$\mathfrak{E} := \{\varepsilon \subset \mathbb{R}^D; \exists T_j, \epsilon_e : \varepsilon = T_j(\epsilon_e)\},$$

where ϵ_e denotes the edges of the simplex from §10. \mathfrak{E} can be decomposed into two disjoint subsets, \mathfrak{E}_{int} and \mathfrak{E}_{bnd} of *internal* and *boundary* edges. Internal edges lay on the border of exactly two cells, while boundary edges lay on the border of only one cell and are subsets of ∂K .

§24: Remark - Graph of the grid: Each edge $\varepsilon \in \mathfrak{E}_{int}$ can be identified by its neighboring cells, i.e. by a set $\{K, L\} \subset \mathfrak{K}$. The tuple $(\mathfrak{K}, \mathfrak{E}_{int})$ forms a nondirectional graph in the common sense, where \mathfrak{K} are the nodes and \mathfrak{E}_{int} are the edges of the graph.

3.2.2 Definition of DG fields

§25: Definition - The discontinuous Galerkin space $DG_p(\mathfrak{K})$: The *discontinuous Galerkin space*, or *DG-space* of degree p is defined as

$$DG_p(\mathfrak{K}) := \left\{ f \in L^2(\Omega); \forall 0 \leq j < J : f|_{K_j} \text{ is polynomial} \right. \\ \left. \text{and } \deg(f|_{K_j}) \leq p \right\}.$$

§26: Notation - dimension of a vector space: V will be denoted as $\dim(V)$ in subsequence.

²¹BoSSS.Foundation.Grid.GridCommons.Vertices

§27: Definition and Remark - An orthonormal basis for DG_p : The polynomials

$$\phi_{j,n}(\mathbf{x}) := \begin{cases} \frac{1}{\sqrt{|\det \mathbf{T}_j|}} \cdot \phi_n(T_j^{-1}(\mathbf{x})) & \text{if } \mathbf{x} \in \bar{K}_j \\ 0 & \text{elsewhere} \end{cases}$$

are an orthonormal basis of $DG_p(\mathfrak{K})$, for all n with $\deg(\phi_n) \leq p$. Here, ϕ_n denotes the orthonormal basis associated with the reference element, see §10. Note that the members of the orthonormal basis in the reference domain K_{ref} is notated as ϕ_n , while those of the orthonormal basis in cell K_j are notated as $\phi_{j,n}$.

Furthermore, let

$$N_p := \max\{n; \deg(\phi_n) \leq p\}$$

be the number of basis polynomials in one cell. All single-phase DG-fields²² are defined with respect to that basis, i.e. some field $f \in DG_p$ can be written as

$$f(\mathbf{x}) = \sum_{n=0}^{N-1} \phi_{j,n}(\mathbf{x}) \tilde{f}_{j,n} \quad \text{for } \mathbf{x} \in K_j.$$

The coefficients $\tilde{f}_{j,n} \in \mathbb{R}$ are called the DG-coordinates of f and are found at²³. The gradient $\nabla_{\mathbf{x}} \phi_{j,n}$ is given/defined as

$$\nabla_{\mathbf{x}} \phi_{j,n} := \begin{cases} \frac{1}{\sqrt{|\det \mathbf{T}_j|}} \cdot (\mathbf{T}_j^{-1})^T \cdot \nabla_{\boldsymbol{\xi}} \phi_n(\boldsymbol{\xi}) & \text{if } \mathbf{x} \in K_j \\ 0 & \text{elsewhere} \end{cases}.$$

Here $\nabla_{\mathbf{x}} := \left(\frac{\partial}{\partial x_0}, \dots, \frac{\partial}{\partial x_{D-1}} \right)$, $\boldsymbol{\xi} := T_j^{-1}(\mathbf{x})$ and $\nabla_{\boldsymbol{\xi}} := \left(\frac{\partial}{\partial \xi_0}, \dots, \frac{\partial}{\partial \xi_{D-1}} \right)$.

Rationale: The factor $\frac{1}{\sqrt{|\det \mathbf{T}_j|}}$ in $\phi_{j,n}$ is there to ensure $\delta_{n,m} \stackrel{!}{=} \langle \phi_{j,n}, \phi_{j,m} \rangle_{K_j}$: $\langle \phi_{j,n}, \phi_{j,m} \rangle_{K_j} = |\det(\mathbf{T}_j)| \int_{K_{\text{ref}}} \phi_{j,n}(T_j(\boldsymbol{\xi})) \phi_{j,m}(T_j(\boldsymbol{\xi})) d\boldsymbol{\xi} = \int_{K_{\text{ref}}} \phi_n(\boldsymbol{\xi}) \phi_m(\boldsymbol{\xi}) d\boldsymbol{\xi} = \delta_{n,m}$. End of Rationale.

§28: Remark - Dimension of DG_p : For a constant polynomial degree per cell

$$\dim(DG_p(\mathfrak{K})) = N_p \cdot J,$$

where J denotes the number of cells.

²²BoSSS.Foundation.Field

²³BoSSS.Foundation.Field.Coordinates

§29: Definition and Remark - Projection onto $DG_p(\mathfrak{K})$: The projector

$$Proj_p : L^2(\Omega) \rightarrow DG_p(\mathfrak{K}), \quad f \mapsto \underline{f} := \sum_{j,n} \phi_{j,n} \tilde{f}_{j,n}$$

is given by the property

$$(f - \underline{f}) \perp DG_p(\mathfrak{K}),$$

i.e. $\langle f - \underline{f}, \underline{g} \rangle = 0$ for all $\underline{g} \in DG_p(\mathfrak{K})$. For the *orthonormal* basis functions $(\phi_{j,n})$, the DG-coordinates of f are given by

$$\tilde{f}_{j,n} = \langle \phi_{j,n}, f \rangle_{\Omega}.$$

§30: Remark : Usually, underlining will denote that some property is a member of $DG_p(\mathfrak{K})$, i.e. $\underline{f} \in DG_p(\mathfrak{K})$. \underline{f} is usually an approximation of $f \in L^2(\Omega)$, e.g. $\underline{f} = Proj_p(f)$; the DG-coordinates of \underline{f} are decorated by a tilde, i.e. $\underline{f} = \sum \phi_{j,n} \cdot \tilde{f}_{j,n}$. In Finite Volume methods sometimes a subscript $-h$ is used to indicate that the accuracy depends on the cell width h . In DG it is also the polynomial degree p that matters. Since the notation in this manuscript already uses a lot of indices, we found adding a (h, p) -superscript pair to members of $DG_p(\mathfrak{K})$ not very instructive.

§31: Remark : In BoSSS Vector fields can be composed from scalar fields, or the vector-field – container²⁴ could be used.

§32: Definition and Remark - Coordinate mapping²⁵: Let

$$\underline{U} := (\underline{u}_0, \dots, \underline{u}_{\Lambda-1}) \in \underbrace{DG_{p_0}(\mathfrak{K}) \times \dots \times DG_{p_{\Lambda-1}}(\mathfrak{K})}_{=: DG_{(p_0, \dots, p_{\Lambda-1})}(\mathfrak{K})}$$

be some list of DG-fields and $\tilde{u}_{\delta,j,n} \in \mathbb{R}$ the DG-coordinate of the DG-field \underline{u}_{δ} in cell j for the n -th basis polynomial, i.e.

$$\underline{u}_{\delta}(\mathbf{x}) = \sum_{j,n} \phi_{j,n}(\mathbf{x}) \tilde{u}_{\delta,j,n}.$$

²⁴BoSSS.Foundation.VectorField

²⁵BoSSS.Foundation.CoordinateMapping

All DG-coordinates can be arranged in a vector $\tilde{U} \in \mathbb{R}^E$, with

$$E = \dim \left(DG_{(p_0, \dots, p_{\Lambda-1})}(\mathfrak{K}) \right) = J \cdot \left(\sum_{\delta} N_{p_{\delta}} \right).$$

In BoSSS, this choice is

$$[\tilde{U}]_{\text{map}(\delta, j, n)} = \tilde{u}_{\delta, j, n}$$

with

$$\text{map}(\delta, j, n) := j \cdot \underbrace{\left(\sum_{\gamma=0}^{\Lambda-1} N_{p_{\gamma}} \right)}_{\text{“degrees-of-freedom per cell”}} + \sum_{\gamma=0}^{\delta-1} N_{p_{\gamma}} + n,$$

In other words, the cell index j is rotating slowest, while the index for the basis polynomial, n , is rotating fastest. By this choice, a vector-space isomorphism between \mathbb{R}^E and $DG_{p_0}(\mathfrak{K}) \times \dots \times DG_{p_{\Lambda-1}}(\mathfrak{K})$, in BoSSS referred to as *coordinate mapping*, is defined. Subsequently, the coordinate mapping for a list \underline{U} will be denoted by \tilde{U} :

$$\tilde{U} = \left[\begin{array}{ccc} \left. \begin{array}{c} \tilde{u}_{0,0,0} \\ \vdots \\ \tilde{u}_{0,0,N_0-1} \end{array} \right\} & \begin{array}{c} \text{0-th DG-field} \\ \\ \end{array} & \left. \vphantom{\begin{array}{c} \tilde{u}_{0,0,0} \\ \vdots \\ \tilde{u}_{0,0,N_0-1} \end{array}} \right\} \text{0-th cell} \\ \vdots & & \\ \left. \begin{array}{c} \tilde{u}_{\Lambda-1,0,0} \\ \vdots \\ \tilde{u}_{\Lambda-1,0,N_{\Lambda-1}-1} \end{array} \right\} & \begin{array}{c} (\Lambda-1)\text{-th DG-field} \\ \\ \end{array} & \left. \vphantom{\begin{array}{c} \tilde{u}_{\Lambda-1,0,0} \\ \vdots \\ \tilde{u}_{\Lambda-1,0,N_{\Lambda-1}-1} \end{array}} \right\} \\ \vdots & & \\ \vdots & & \\ \left. \begin{array}{c} \tilde{u}_{0,J-1,0} \\ \vdots \\ \tilde{u}_{0,J-1,N_0-1} \end{array} \right\} & \begin{array}{c} \text{0-th DG-field} \\ \\ \end{array} & \left. \vphantom{\begin{array}{c} \tilde{u}_{0,J-1,0} \\ \vdots \\ \tilde{u}_{0,J-1,N_0-1} \end{array}} \right\} \text{(J-1)-th cell} \\ \vdots & & \\ \left. \begin{array}{c} \tilde{u}_{\Lambda-1,J-1,0} \\ \vdots \\ \tilde{u}_{\Lambda-1,J-1,N_{\Lambda-1}-1} \end{array} \right\} & \begin{array}{c} (\Lambda-1)\text{-th DG-field} \\ \\ \end{array} & \left. \vphantom{\begin{array}{c} \tilde{u}_{\Lambda-1,J-1,0} \\ \vdots \\ \tilde{u}_{\Lambda-1,J-1,N_{\Lambda-1}-1} \end{array}} \right\} \end{array}$$

§33: Notation : By the coordinate mapping, we can notate the DG basis elements $\phi_{j,n}$ as a vector $\underline{\Phi}$ with

$$[\underline{\Phi}]_{map(0,j,n)} = \phi_{j,n} \quad \forall j, n.$$

By that, $\underline{u} \in DG_p(\mathcal{K})$ with DG coordinate vector $\tilde{u} \in \mathbb{R}^{J \cdot N_p}$ can be represented as $\underline{u} = \underline{\Phi} \cdot \tilde{u}$ and $\tilde{u} = \langle \underline{\Phi}, \underline{u} \rangle_{\Omega}$, and for $u \in L^2(\Omega)$, $Proj_p(u) = \underline{\Phi} \cdot \langle \underline{\Phi}, u \rangle_{\Omega}$.

3.3 MPI – Parallelization of the BoSSS code

Users which are not intending to implement extensions to BoSSS layer 2 may skip this section. For others, it is mainly important to understand the concepts of grid partitioning, locally updated and external cells (see §37 and figure 3.1) and further how local cell indices translate into global ones (see 40) and what a GlobalID - number is (see §15).

3.3.1 Parallel partitioning of the grid

§34: Notation - MPI - communicator: The set of sz MPI-processes, i.e. the MPI-communicator of size sz , is identified with the set $\{0, \dots, sz - 1\} =: MPI_{comm}$. In BoSSS, sz is found at²⁶.

§35: Definition - Grid partition: A partitioning of the grid with J cells is a monotonally increasing, surjective mapping

$$part : \{0, \dots, J - 1\} \rightarrow MPI_{comm}.$$

Further, the number

$$J_u(p) := \#part^{-1}(p) = \#\{j; part(j) = p\}$$

is called the *size of the partition on process p*. Further,

$$j_0(p) := \sum_{q=0}^{p-1} J_u(q)$$

is the index of the “first” cell on MPI process p .

²⁶BoSSS.Foundation.CommMaster.Size

The grid partition defines which cells are assigned to which MPI-process, i.e. cell K_j is assigned to process $\text{part}(j)$. In BoSSS, $J_u(p)$ is found at ²⁷, the partitioning is found at ²⁸.

§36: Definition - Process boundary: The process boundary between processes p and $h \in \text{MPI}_{\text{comm}}$ is defined as the set

$$\{\{K_j, K_l\} \in \mathfrak{E}_{\text{int}}; \text{part}(j) = p \text{ and } \text{part}(l) = h\} =: \mathfrak{E}_{\text{Bnd}}(p, h).$$

§37: Definition - Locally updated cells, external cells: For process p , the set of *locally updated cells* is defined as

$$\{K_j \in \mathfrak{K}; \text{part}(K_j) = \text{proc}_p\} =: \mathfrak{K}_{\text{loc}}(p).$$

These cells are called “locally updated”, because values which are associated with those cells are usually computed in process p . Note that $\#\mathfrak{K}_{\text{loc}}(p) = J_u(p)$, called the “number of locally updated cells”. The *external cells* are defined as the set

$$\{K \in \mathfrak{K}; \exists \{M, L\} \in \mathfrak{E} : \text{Part}(M) = \text{proc}_p \text{ and } \text{Part}(L) \neq \text{proc}_p\} =: \mathfrak{K}_{\text{ext}}(p).$$

Furthermore, $J_{\text{ext}}(p) := \#\mathfrak{K}_{\text{ext}}(p)$ denotes the number of external cells on process p and $J_{\text{tot}}(p) := J_u(p) + J_{\text{ext}}(p)$ denotes the total number of cells associated with process p .

§38: Remark - Motivation for external cells: Usually, in DG methods it is required to compute integrals over edges, and these integrals involve values from both neighbouring cells. However, if a partition with $\text{sz} > 1$ is given, the neighbouring cells K_j and K_l for some internal edge $\varepsilon = \{K_j, K_l\} \in \mathfrak{E}_{\text{int}}$ may be assigned to different processors, i.e. $\text{part}(j) \neq \text{part}(l)$. One possible solution to this issue is to store a layer of external cells on each processor. This will be described more precisely in below.

§39: Definition - Local cell index: On processor p , a mapping

$$\{0, \dots, J_{\text{tot}}(p) - 1\} \ni i \rightarrow j_{\text{gl}}(i) \in \{0, \dots, J - 1\}$$

relates the local cell index $i \in \{0, \dots, J_{\text{tot}}(p) - 1\}$ with the global cell index $j = j_{\text{gl}}(i)$.

²⁷BoSSS.Foundation.Grid.GridData.NoOfLocalUpdatedCells

²⁸BoSSS.Foundation.Grid.GridData.GridPartition

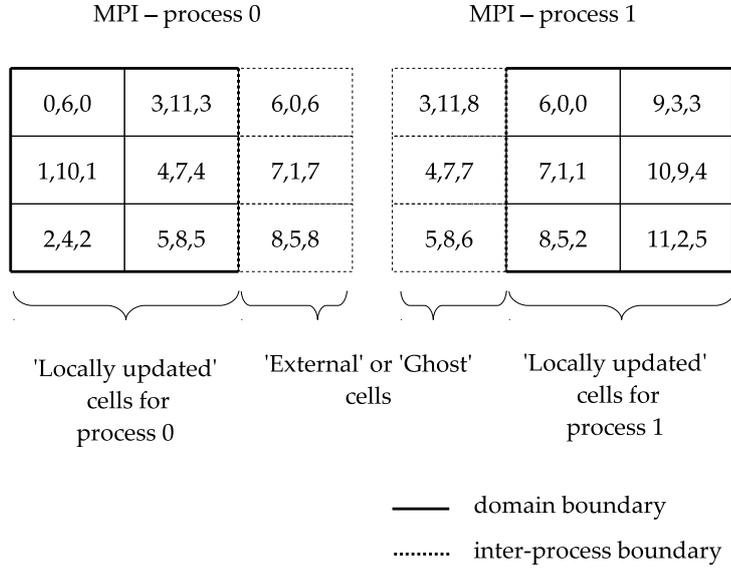


Figure 3.1: Illustration of grid - partitioning: the triplets denote GlobalId,GlobalIndex,LocalIndex.

i is called to be in the “locally updated range” if $i < J_u(p)$. In this case $j_{gl}(i) = j_0(p) + i$. For $i \geq J_u(p)$ no natural choice can be given. BoSSS computes j_{gl} , on every MPI process, on every start-up, in a way that

$$j_{gl}(\{J_u(p), \dots, J_{tot}(p) - 1\}) = \{j; K_j \in \mathfrak{K}_{ext}(p)\}.$$

The global cell indices for external cells, i.e.. the list $(j_{gl}(J_u(p)), \dots, j_{gl}(J_{tot}(p) - 1))$ is stored at²⁹.

§40: Remark : The relation between local and global indices, between GlobalId and cell can be illustrated by the following table:

	local updated range	external range
local index i :	$0, \dots, J_u(p) - 1$	$J_u(p), \dots, J_{tot}(p) - 1$
global index j :	$j = j_{gl}(i) = i + i_0(p)$	$j = j_{gl}(i)$
GlobalId id :	$\tau(j)$	$\tau(j_{gl}(i))$
cell:	K_j	$K_{j_{gl}(i)}$

So, at runtime, BoSSS takes two choices which determine the ordering of cells in memory: at first, the GlobalId - permutation, which is chosen globally over all processors, determines the ordering of locally updated cells. Second, on each process, a local index for external cells is chosen.

²⁹BoSSS.Foundation.Grid.GirdData.GlobalIndicesExternalCells

§41: Remark : Given a DG-coordinate vector $\tilde{U} \in \mathbb{R}^{J \cdot N_p}$, its associated mapping $map(0, -, -)$ and a partition part of the grid, the local part of \tilde{U} is given by

$$[\tilde{U}]_{map^{-1}(\{0\} \times part^{-1}(p) \times \{0, \dots, N_p - 1\})} = \begin{bmatrix} \tilde{U}_{map(0, j_0(p), 0)} \\ \vdots \\ \tilde{U}_{map(0, j_0(p), N_p - 1)} \\ \vdots \\ \tilde{U}_{map(0, j_0(p) + J_u(p) - 1, 0)} \\ \vdots \\ \tilde{U}_{map(0, j_0(p) + J_u(p) - 1, N_p - 1)} \end{bmatrix},$$

i.e. the part of \tilde{U} that is assigned to MPI process p .

3.4 The discretization of spatial differential operators

Within this section, the discretization of differential operators by the DG method will be discussed. Basically, a spatial operator in BoSSS is a mapping from some domain variable vector U to a codomain vector W . The mapping can be a differential operator, an algebraic or analytic function, or a combination of these. In the domain of DG, differential operators are usually expressed by the means of Riemannian flux functions.

§42: Notation - spatial operator: For arbitrary linear function spaces $Dom(\Omega)$ and $Cod(\Omega)$, referred to as *domain* and *codomain*,

- a member $U := (u_0, \dots, u_{\Lambda-1}) : \Omega \rightarrow \mathbb{R}^\Lambda$ of $Dom(\Omega)$ is called a list of *domain variables*, and
- a member $W := (w_0, \dots, w_{\Gamma-1}) : \Omega \rightarrow \mathbb{R}^\Gamma$ of $Cod(\Omega)$ is called a list of *codomain variables*.

A mapping

$$Dom(\Omega) \ni U \mapsto W = \mathcal{F}(U) \in Cod(\Omega),$$

is called a *spatial operator* from $Dom(\Omega)$ to $Cod(\Omega)$.

§43: Remark - Discretization of spatial operators by the DG method: To introduce the DG discretization of some operator \mathcal{D} , as introduced in §42, let be

- $Dom_{DG}(\mathfrak{K}) := \prod_{\delta} DG_{p_{\delta}}(\mathfrak{K})$ the DG-space to approximate members of $Dom(\Omega)$ and
- $Cod_{DG}(\mathfrak{K}) = \prod_{\gamma} DG_{l_{\gamma}}(\mathfrak{K})$ the DG-space to approximate members of $Cod(\Omega)$.

The following diagram explains the relation between a spatial operator \mathcal{F} , its DG-discretization \mathbf{F} and the coordinate representation $\tilde{\mathbf{F}}$:

$$\begin{array}{ccc}
Dom(\Omega) & \xrightarrow{\mathcal{F}} & Cod(\Omega) \\
\text{Proj} \downarrow & & \downarrow \text{Proj} \\
Dom_{DG}(\mathfrak{K}) & \xrightarrow{\mathbf{F}} & Cod_{DG}(\mathfrak{K}) \\
\text{map} \uparrow \|\cdot\| & & \|\cdot\| \downarrow \text{map} \\
\mathbb{R}^{l_1} & \xrightarrow{\tilde{\mathbf{F}}} & \mathbb{R}^{l_2}
\end{array}$$

with $l_1 := \dim(Dom_{DG}(\mathfrak{K}))$ and $l_2 := \dim(Cod_{DG}(\mathfrak{K}))$ and $Proj$ being an abbreviation for $(Proj_{p_0}, \dots, Proj_{p_{\Lambda-1}})$ resp. $(Proj_{l_0}, \dots, Proj_{l_{\Gamma-1}})$. The upper part of the diagram is *not* commutative, because in general $Proj(\mathcal{F}(U)) \neq \mathbf{F}(Proj(U))$ for an arbitrary $U \in Dom(\Omega)$. However, it is at least the minimal goal of any decent discretization that this relation holds at least approximately:

$$\begin{array}{ccc}
U & \xrightarrow{\mathcal{F}} & W = \mathcal{F}(U) \\
\text{Proj} \downarrow & & \downarrow \text{Proj} \\
\underline{U} & \xrightarrow{\mathbf{F}} & \underline{\mathbf{F}(U)} \quad \approx \quad \underline{W}
\end{array}$$

This ‘‘approximately’’ commutative diagram motivates §44.

§44: Definition - Consistency of operators: Using the notation and symbols of §43, an operator \mathbf{F} is called consistent with an operator \mathcal{F} with convergence order k in the norm $\|\cdot\|$ if

$$\|\mathbf{F}(Proj_p(U)) - Proj_p(\mathcal{F}(U))\| \leq h_{\max}^k \cdot c(U)$$

for any $U \in Dom(\Omega)$ and a constant $c(U) \in \mathbb{R}_{\geq 0}$ which depends on U .

Usually, a differential equation $\mathcal{F}(u) = f$ also requires some boundary conditions, i.e. the problem reads as $\mathcal{F}(u) = g$ in Ω and e.g. $u|_{\partial\Omega} = g$ on the boundary $\partial\Omega$. Which boundary conditions yield well-posed problems usually depends on the type of the differential operator \mathcal{F} , for further reading e.g. consult (Renardy 2004).

In DG, those boundary conditions coalesce with the discrete operator \mathbf{F} . So, instead of solving “PDE in Ω and BC on $\partial\Omega$ ” one just solves $\mathbf{F}(\underline{u}) = \underline{f}$ for $\underline{u} \in DG_p(\mathfrak{K})$, with $\underline{f} = Proj_p(f)$.

In functional analysis, the trace operator (see e.g. (Triebel 1980)) is used to define boundary conditions for members of Sobolev spaces $H^n(\Omega)$, $n \in \mathbb{N}_{>0}$.

§45: Notation - normal vector: In subsequence, \mathbf{n} always denotes an outer or oriented normal field on some orientable set $X \subset \Omega$. If X is not clear from the context, it is denoted as an index, i.e. \mathbf{n}_X .

§46: Reminder - Trace operator: The linear operator

$$\mathcal{C}^n(\overline{\Omega}) \ni u \mapsto Tr(u) = u|_{\partial\Omega} \in L^2(\partial\Omega)$$

with $n \in \mathbb{N}_{>0}$ is bounded, i.e.

$$\exists C \in \mathbb{R}_{>0} : \forall u \in \mathcal{C}^n(\overline{\Omega}) : \|Tr(u)\|_{L^2(\partial\Omega)} \leq C \cdot \|u\|_{H^1(\Omega)}.$$

Because $\mathcal{C}^n \subset H^n(\Omega)$ dense, there exists a unique continuous extension of Tr resp. $-|_{\partial\Omega}$ onto $H^n(\Omega)$.

§47: Definition - Consistency of DG operators with boundary conditions: Let $\bigcup_{i=1}^I \Gamma_i = \partial\Omega$ be a finite partition of $\partial\Omega$, with $(D-1)$ -dimensional measure of each Γ_i unequal to zero. Given is the operator

$$H^n(\Omega)^\Lambda \ni U \mapsto \mathcal{F}(U) = w \in L^2(\Omega)$$

and boundary conditions

$$\forall i \in \{1, \dots, I\} : Tr(\mathcal{E}_i U) = g_i \text{ on } \Gamma_i,$$

where $\mathcal{E}_1, \dots, \mathcal{E}_I$ may be differential operators: e.g. for a Dirichlet boundary condition $\mathcal{E}_i(u) = u$ and for a Neumann boundary $\mathcal{E}_i(u) = \nabla u \cdot \mathbf{n}$. A discretization of \mathcal{D} ,

$$DG_p(\mathfrak{K})^\Lambda \ni \underline{U} \mapsto \mathbf{F}(\underline{U}) \in DG_l(\mathfrak{K})$$

is called to be consistent with the given boundary conditions if it is consistent with \mathcal{F} in the sense of §44 and

$$\|\mathbf{F}(Proj(U_k)) - Proj_l(\mathcal{F}(u_k))\|_2 \rightarrow \infty$$

for any series $(U_1, U_2, \dots) \in (H^n(\Omega)^\Lambda)^\mathbb{N}$ that does not fulfill the boundary conditions, i.e.

$$\sum_{i=1}^I \|Tr(\mathcal{E}_i U_k) - g_i\|_{L^2(\Gamma_i)}^2 \rightarrow \infty.$$

3.5 Some special spatial operators

The following is just a helper definition:

§48: Notation - inner and outer values: For a property $f : \Omega \rightarrow \mathbb{R}$, that is continuous within all cells in \mathfrak{K} , but not necessarily on the cell boundaries, i.e. $f \in \mathcal{C}^0(\Omega \setminus \bigcup_{j=0}^{J-1} K_j)$ the *inner-* and *outer value* of f for some cell $K \in \mathfrak{K}$ at some point $\mathbf{x} \in \partial K_j$ are defined as

$$f^{\text{in}}(\mathbf{x}) := \lim_{\substack{\mathbf{y} \rightarrow \mathbf{x} \\ \mathbf{y} \in K_j}} (f(\mathbf{y})) \text{ and } f^{\text{out}}(\mathbf{x}) := \lim_{\substack{\mathbf{y} \rightarrow \mathbf{x} \\ \mathbf{y} \notin \overline{K_j}}} (f(\mathbf{y})),$$

respectively.

Subsequently, a classification of all types of spatial operators supported by BoSSS is given.

§49: Corollary and Definition - conservative operators: A spatial operator

$$H^1(\Omega)^\Lambda \ni U \mapsto \mathcal{F}(U) = w \in L^2(\Omega)$$

is called a *conservative operator* if it could be written in the form

$$w = \text{div}(\mathbf{f}(\mathbf{x}, U(\mathbf{x}))).$$

It is, given a proper *Riemannian* $\hat{f}(\mathbf{x}, U^{\text{in}}, U^{\text{out}}, \mathbf{n}_x)$, discretized by

$$\prod_{\delta=0}^{\Lambda-1} DG_{p_\delta}(\mathfrak{K}) \ni \underline{U} \mapsto \mathbf{F}(\underline{U}) = \underline{w} \in DG_l(\mathfrak{K})$$

where the DG coordinates of \underline{w} are given by

$$\tilde{w}_{j,n} = \int_{\partial K_j} \hat{f} \phi_{j,n} \, dS - \int_{K_j} \mathbf{f} \cdot \nabla \phi_{j,n} \, dx \quad \forall j, n.$$

Riemannians must fulfill the following necessary properties to yield a consistent discretization \mathbf{D} of \mathcal{D} , see e.g. (LeVeque 2002):

- *Symmetry*: $\widehat{f}(\mathbf{x}, U, V, \mathbf{n}) = -\widehat{f}(\mathbf{x}, V, U, -\mathbf{n})$ i.e. “what leaves one cell must enter the neighbour cell”.
- *Consistency with \mathbf{f}* : $\left| \widehat{f}(\mathbf{x}, U, V, \mathbf{n}) - \mathbf{n} \cdot \mathbf{f}(\mathbf{x}, U) \right| \leq L|U - V|$ for some Lipschitz constant $L \in \mathbb{R}_{\geq 0}$, i.e. “ \widehat{f} is an approximation to $\mathbf{n} \cdot \mathbf{f}$ ”. This implies that $\widehat{f}(\mathbf{x}, U, U, \mathbf{n}) = \mathbf{n} \cdot \mathbf{f}(\mathbf{x}, U)$

§50: Definition - source operators: A spatial operator

$$L^2(\Omega)^\Lambda \ni U \mapsto \mathcal{Q}(U) = w \in L^2(\Omega)$$

is called a *source* if it could be written in the form

$$w = q(\mathbf{x}, U).$$

The DG discretization,

$$\prod_{\delta=0}^{\Lambda-1} DG_{p_\delta}(\mathfrak{K}) \ni \underline{U} \mapsto \mathbf{Q}(\underline{U}) = \underline{w} \in L^2(\Omega)$$

is straightforward, where the DG coordinates of \underline{w} are given by

$$\tilde{w}_{j,n} = \int_{K_j} q_\gamma \phi_{j,n} \, d\mathbf{x} \quad \forall j, n.$$

Using the operators defined so far, one is able to discretize an equation with higher derivatives by decomposition into a system of first-derivative equations. Usually it is more efficient to discretize higher-derivative equations directly, if possible. This can be done e.g. for the Laplace operator. The framework introduced in §51 could be used for various discretizations of the Laplace operator, e.g. like presented in (Arnold et al. 2002). The prototype example for this is the interior penalty discretization of the Laplace operator.

§51: Corollary and Definition - linear second-derivative operators: A spatial operator

$$H^2(\Omega)^\Lambda \ni U \mapsto \mathcal{L}(U) = w \in L^2(\Omega)$$

is called a *linear second-derivative operator* if it could be written in the form

$$w = \operatorname{div}(v(\mathbf{x}) \cdot \nabla G(U)).$$

with a linear function $G(U) = M \cdot U$, where the matrix $M \in \mathbb{R}^{1 \times \Lambda}$ may *not* depend on \mathbf{x} and $v \in \mathcal{C}^1(\Omega)$.

Given a linear Riemannian for the *primal variable* $\widehat{G}(U^{\text{in}}, U^{\text{out}})$ and a linear Riemannian for the *derivative variable* $\widehat{\sigma}(U^{\text{in}}, U^{\text{out}}, \nabla U^{\text{in}} \cdot \mathbf{n}, \nabla U^{\text{out}} \cdot \mathbf{n}, \mathbf{n})$, the discretization

$$\prod_{\delta=0}^{\Lambda-1} DG_{p_\delta}(\mathfrak{K}) \ni \underline{U} \mapsto \mathbf{L}(\underline{U}) = \underline{w} \in DG_l(\mathfrak{K})$$

where the DG coordinates of \underline{w} are given by

$$\tilde{w}_{j,n} = \int_{\partial K_j} \left(\widehat{\sigma} \cdot \Phi_{j,n} - v \cdot (\widehat{G} - G) \cdot \nabla \Phi_{j,n} \right) \cdot \mathbf{n} \, dS - \int_{K_j} (v \cdot \nabla \Phi_{j,n} \cdot \nabla G) \, dx$$

for all j, n .

§52: Example - prototype for linear second-derivative operators: Given is the operator $u \mapsto \text{div}(v(\mathbf{x}) \cdot \nabla u)$ with $v \in \mathcal{C}^1(\Omega)$, (i.e. in the notation of §51 $G \equiv 1$) and the partition $\partial\Omega = \Gamma_{\text{Neu}} \cup \Gamma_{\text{Diri}}$ and boundary conditions

$$\begin{aligned} u|_{\Gamma_{\text{Diri}}} &= g_{\text{Diri}} \quad \text{and} \\ (\mathbf{n} \cdot \nabla u)|_{\Gamma_{\text{Neu}}} &= g_{\text{Neu}}. \end{aligned}$$

The symmetric interior penalty discretization with penalty parameter μ (Arnold et al. 2002, Shahbazi 2005) is given by the Riemannians

$$\begin{aligned} \widehat{G}(u^{\text{in}}, u^{\text{out}}) &= \frac{1}{2} (u^{\text{in}} + u^{\text{out}}) \quad \text{and} \\ \widehat{\sigma}(u^{\text{in}}, u^{\text{out}}, \nabla u^{\text{in}} \cdot \mathbf{n}, \nabla u^{\text{out}} \cdot \mathbf{n}, \mathbf{n}) &= \frac{v}{2} (\nabla u^{\text{in}} + \nabla u^{\text{out}}) \\ &\quad - \mu \cdot v \cdot \mathbf{n} \cdot (u^{\text{in}} - u^{\text{out}}) \end{aligned}$$

on the interior edges $\mathfrak{E}_{\text{int}}$. On Γ_{Diri} the Riemannians are

$$\begin{aligned} \widehat{G}(u^{\text{in}}) &= g_{\text{Diri}} \quad \text{and} \\ \widehat{\sigma}(u^{\text{in}}, \nabla u^{\text{in}} \cdot \mathbf{n}, \mathbf{n}) &= v \cdot \nabla u^{\text{in}} - \mu \cdot v \cdot \mathbf{n} \cdot (u^{\text{in}} - g_{\text{Diri}}). \end{aligned}$$

and on Γ_{Neu} they are defined as

$$\begin{aligned} \widehat{G}(u^{\text{in}}) &= u^{\text{in}} \quad \text{and} \\ \widehat{\sigma}(u^{\text{in}}, \nabla u^{\text{in}} \cdot \mathbf{n}, \mathbf{n}) &= \mathbf{n} \cdot v \cdot g_{\text{Neu}}. \end{aligned}$$

These choices yield a consistent discretization of $\text{div}(v \cdot \nabla -)$, consistent with the given boundary conditions. Further the operator matrix (see §55) will be symmetric.

§53: Remark - about the penalty parameter: The so-called penalty parameter $\mu \in \mathbb{R}_{>0}$ in §52 must be chosen large enough to ensure coercivity of the operator matrix; However, a larger penalty parameter increases the condition number of the matrix and therefore increases the cost of running an iterative sparse solver.

For computing μ , a lower-bound estimation from (Shahbazi 2005) is used; for each $K_j \in \mathfrak{K}$

$$c_j := \alpha \cdot \frac{(p+1)(p+D)}{D} \cdot \frac{\text{Vol}_{D-1}(\partial K_j \setminus \partial \Omega) / 2 + \text{Vol}_{D-1}(\partial K_j \cap \partial \Omega)}{\text{Vol}_D(K_j)},$$

where p is the polynomial degree of the used DG approximation. In BoSSS, c_j can be found at³⁰. The factor α , which is not present in the originally published formula, can be used as a tuning factor, we suggest to choose α so that $0.95 \leq \alpha \leq 1.3$. On grids with a very high ratio of biggest against smallest cell, adjustment of α may help improving the numerical convergence of iterative sparse solvers. Numerical experiments show that the lower-bound estimation seems to be pretty good, so values of α smaller than 1 should be used with care.

On an interior edge $\{K_j, K_l\} \in \mathfrak{E}_{int}$, i.e. $\mathbf{x} \in \overline{K_j} \cap \overline{K_l}$,

$$\mu(\mathbf{x}) = \max\{c_j, c_l\}.$$

On boundary edges, with $\mathbf{x} \in \overline{K_j}$ and $\mathbf{x} \in \partial \Omega$,

$$\mu(\mathbf{x}) = c_j.$$

§54: Remark - Additive component decomposition of conservative operators in BoSSS: A *spatial operator*³¹ object in BoSSS specifies at first, domain³² and codomain³³ variables are specified as lists of symbolic names.

For each codomain variable, an additive composition of atomic operators, which are called *equation components*³⁴ is defined. Each of the equation components can be seen as spatial operator with exactly one codomain variable, i.e. it maps $\prod_{\delta} DG_{p_{\gamma}} \rightarrow DG_l$. The domain variables of the equation components are specified as some sub-list³⁵ of the domain variables

³⁰BoSSS.Foundation.Grid.GridData.cj

³¹BoSSS.Foundation.SpatialOperator

³²BoSSS.Foundation.SpatialOperator.DomainVar

³³BoSSS.Foundation.SpatialOperator.CodomainVar

³⁴BoSSS.Foundation.IEquationComponent

³⁵BoSSS.Foundation.IEquationComponent.ArgumentOrdering

list of the spatial difference operator. The equation components may be classified into *fluxes* and *sources*, and whether they are either *linear* or *non-linear*:

- nonlinear fluxes^{36, 37}, as defined in §49.
- nonlinear sources³⁸, like defined in §50.
- linear fluxes³⁹, like defined in §49.
- linear sources⁴⁰, like defined in §50.
- linear second-derivative fluxes⁴¹, like defined in §51.
- linear penaltization (dual-value flux)⁴², like used in §93.
- nonlinear penaltization (dual-value flux)⁴³, like used in §93.
- linear sources that depend on derivatives⁴⁴, like used in §93.

If more than one equation component is specified for one codomain variable, they are added⁴⁵.

§55: Remark - Evaluation of conservative operators in BoSSS: A conservative operator, or spatial differential operator \mathbf{F} , which is the discretization of \mathcal{F} can be evaluated in BoSSS by⁴⁶ or⁴⁷. If \mathbf{F} is affine-linear, i.e.

$$\tilde{\mathbf{F}}(\tilde{U}) = \mathcal{M}_{\mathcal{F}} \cdot \tilde{U} + b,$$

the (sparse) matrix $\mathcal{M}_{\mathcal{F}}$ and the affine vector b can be computed by⁴⁸.

³⁶BoSSS.Foundation.INonlinearFlux

³⁷BoSSS.Foundation.INonlinearFluxEx

³⁸BoSSS.Foundation.INonlinearSource

³⁹BoSSS.Foundation.ILinearFlux

⁴⁰BoSSS.Foundation.ILinearSource

⁴¹BoSSS.Foundation.ILinear2ndDerivativeFlux

⁴²BoSSS.Foundation.ILinearDualValueFlux

⁴³BoSSS.Foundation.IDualValueFlux

⁴⁴BoSSS.Foundation.ILinearDerivativeSource

⁴⁵BoSSS.Foundation.SpatialOperator.EquationComponents

⁴⁶BoSSS.Foundation.SpatialOperator.Evaluate(...)

⁴⁷BoSSS.Foundation.SpatialOperator.Evaluator.Evaluate(...)

⁴⁸BoSSS.Foundation.SpatialOperator.ComputeMatrix(...)

4 GPU – accelerated, MPI – parallel sparse solvers

4.1 Introduction

In BoSSS, all available sparse solvers can be found in the `ilPSP.LinSolvers` – namespace. `ilPSP` (*intermediate language Parallel Scientific Plattform*) is a software package independent from BoSSS and published as a stand-alone library at Sourceforge.org¹. Apart from wrappers to the HYPRE²- (Falgout, Jones & Yang 2006) and the PARDISO³ – libraries (Schenk, Gärtner & Fichtner 2000, Schenk 2002, Schenk 2004, Schenk & Gärtner 2006), it contains our own C# – based solver library `monkey`⁴, which offers GPU - acceleration but offers, in absence of GPU's, decent performance on CPU's, too.

GPU (Graphics Processing Unit) - acceleration was the primary motivation for creating the “monkey” – sparse-solver library, in addition to the already existing ones. Beyond that there are further reasons for a special .NET - based sparse solver for BoSSS:

- BoSSS is designed to be “compiled once – run everywhere”; therefore it is convenient to be independent of native sparse solvers, like HYPRE, because they usually have to be compiled on every super-computer individually.
- It turned out that in some cases the usual black-box approach in which sparse solver libraries are designed is very limited. Usually libraries like HYPRE provide functionality to pass the matrix to the solver, but they are lacking methods for retrieving the matrix back from the solver, or altering specific entries⁵

¹sourceforge.net/projects/ilpsp, 1st of August 2011

²`ilPSP.LinSolvers.HYPRE` – namespace

³`ilPSP.LinSolvers.PARDISO` – namespace

⁴`ilPSP.LinSolvers.monkey` – namespace

⁵ To be precise, HYPRE provides means to read and write matrix entries if their position, i.e. row- and column - index is known. However, to read out the full matrix again, these indices must be stored separately by the HYPRE - user. This implies that the

There are many cases where the values of matrix entries are changed in between solver runs (e.g. implicit Euler schemes with varying timestep size, where the matrix of the linear system has the form $\frac{1}{\Delta t}I - \mathcal{M}$, or the linearised convectional part of a SIMPLE-type Navier-Stokes – solver), but the position of zero/non-zero entries remains constant. In such cases it is beneficial to have more advanced access methods to matrix entries, like defined by the *mutable-matrix*⁶ - interface.

To program GPU's, one has the choice between the Cuda- (NVIDIA Corporation 2011) or the OpenCL-API (Khronos Group 2011) (API: Application Program Interface). Our experience is that Cuda offers higher performance and is better documented, but the disadvantage of Cuda is that it is only supported by the NVIDIA Corporation, which is unchallenged market-leader for GPUs for scientific purposes and also market leader for GPUs in general. The monkey library supports Cuda as well as OpenCL, although the latter is still work-in-progress.

We restricted the monkey-library to the top-range of current GPUs. First this is necessary because only those GPUs support double-precision floating point operations⁷ and secondly it is helpful, because we do not need to optimize for entry- and mid-range GPUs that have reduced hardware capabilities and would require different optimization techniques.

It should be mentioned that Christoph Busold did the implementation of the GPU-core routines in CUDA-C, as well as the OpenCL port and the performance measurements, while the author of this manuscript developed the architecture of the monkey-package, the CPU-implementation and the MPI-paralleization.

nonzero - indices are stored at least twice, once within Hypr and once in the HYPRE - user application. Taking into account that the matrices of linear systems are very memory-consuming objects, this situation is not desirable: Especially in DG, consider a problem with J cells and N DOF per cells (e.g. $N = 10$ for polynomial degree 2 in 3D) and K neighbour cells for each cell (e.g. $K = 6$ for a 3D - Cartesian grid). Then the matrix can have up to $J \cdot N^2 \cdot K$ entries!

⁶ilPSP.LinSolvers.IMutableMatrixEx

⁷ Double - precision floating point support is only given for GPUs with Cuda Compute-Capability 1.3 or higher (see (NVIDIA Corporation 2010), page 14), which is currently only available for high-end consumer GPUs like the NVIDIA GTX 480 or the professional NVIDIA Tesla, Quadro or Fermi cards.

4.2 Matrix storage formats

The core operation of an iterative sparse solver is usually the sparse matrix-vector product (SpMV). For an $N \times N$ - sparse matrix, this operation has $\mathcal{O}(N)$ operands and $\mathcal{O}(N)$ floating-point operations. Especially, each nonzero matrix entry is only involved in one multiplication. For these reasons, execution speed of the SpMV is bound by the memory bandwidth and not so much by the speed of the floating-point unit. This fact is even true for CPU execution. While CPU's feature very efficient caching- and pre-fetching - technologies, that optimize memory access without the programmer's notice or control, GPUs require much more programmer attention to achieve optimal performance of memory access⁸; it is essential that the GPU threads access consecutive memory addresses. This is known as *coalesced memory access* (see (NVIDIA Corporation 2010), page 84).

There is already a lot of information available about GPU memory hierarchies and how to exploit them. A very good work which focuses on SpMV is (Wafei 2009), which was also the starting point for this work. An – up to our knowledge – novel matrix format which offers better performance than the commonly known ones is defined in §63.

For reasons of completeness, formal definitions of all implemented matrix storage formats are given. The most instructive approach to these definitions is probably the study of figures 4.1, 4.2 and 4.3.

Four matrix formats will be presented subsequently; at first the Compressed Row Storage (CSR) format, see §57, second the Block-Compressed Row Storage format, §59, third the ELLPACK – format, see §61 and finally a variant of ELLPACK with additional cache optimization (ManualCache-ELLPACK), see §4.1.

§56: Notation : Within the remains of this chapter, let ...

- M be an $N \times L$ matrix,
- N_Z be the number of non-zero entries in M
- be $e_{i,j} = [\delta_{i,k} \cdot \delta_{j,l}]_{\substack{i=0,\dots,N-1 \\ j=0,\dots,L-1}}$, i.e. an $N \times L$ – matrix with an 1 at entry (i, j) and 0 everywhere else

⁸Although the latest NVIDIA Fermi GPUs offer some caching, it is still necessary to optimize their memory access pattern.

§57: Definition - CSR – matrix format⁹: The matrix M is given as

$$M = \sum_{j=0}^{N-1} \sum_{i=r(j)}^{r(j+1)-1} e_{j,c(i)} \cdot m(i),$$

where

- $r \in \mathbb{N}^{N+1}$ denotes the row-start - vector¹⁰,
- $c \in \mathbb{N}^{Nz}$ denotes column.index - vector¹¹ and
- $m \in \mathbb{R}^{Nz}$ contains the matrix entries¹².

§58: Remark - On the CSR format:

- On GPUs, the CSR format shows inferior performance and is only implemented for reference purpose. One reason for this is that the matrix values are stored row-by-row. If each GPU thread processes one row of the matrix, it is clear that memory access cannot be coalesced.
- On CPU's, the CSR format works with decent performance and is currently the only implemented format.
- The CSR format is specially suitable if the number of nonzeros varies a lot in between the cells.

Within the whole chapter, the term “block” refers to Cuda-thread-blocks and not to sub-matrices, as usual, to avoid confusion. Instead sub-matrices are referred to as “cells”, because they can usually be assigned to cells of the numerical grid, if M is a Finite-Volume or DG discretization of a partial differential equation. However, for the naming of the Block-CSR – format an exception is made, because well-established names should not be changed.

For a more instructive approach to §59, refer to figure 4.1.

⁹ilPSP.LinSolvers.monkey.MatrixBase.CSR

¹⁰ilPSP.LinSolvers.monkey.MatrixBase.CSR.RowStart

¹¹ilPSP.LinSolvers.monkey.MatrixBase.CSR.Collnd

¹²ilPSP.LinSolvers.monkey.MatrixBase.FormatBase.Val

§59: Definition - Block-CSR – matrix format¹³: Let the cell-size C be a divider of N and L . Then M is split into $N/C \times L/C$ sub-matrices, referred to as *cells*¹⁴. Then M is given as

$$M = \sum_{j=0}^{N/C-1} \sum_{i=r(j)}^{r(j+1)-1} \sum_{\substack{i'=0 \\ j'=0}}^{C-1} e^{(j \cdot C + i'), (c(i) \cdot C + j')} A_{i,i',j'},$$

where

- N_B is the total number of non-zero cells¹⁵,
- $r \in \mathbb{N}^{N/C+1}$ is the cell-row-start – vector¹⁶ and
- $c \in \mathbb{N}^{N_B}$ denotes the cell-column – vector¹⁷.
- The non-zero cells of the matrix: $A_0, \dots, A_{N_B-1} \in \mathbb{R}^{C \times C}$

The cells themselves are encoded as a vector m (see¹⁸), with a given *column-stride* $S_{Co} \geq C$ and a given *cell-stride* $S_{Cl} \geq C^2$ so that

$$A_{i,i',j'} = m(i \cdot S_{Cl} + j' \cdot S_{Co} + i').$$

- §60: Remark - On the Block-CSR format:**
- The format is especially efficient if the cell-size C is a not-too-small power of two (e.g. 64, 128, 256, ...). For DG methods of low polynomial degree (e.g. $p = 1, 2, 3$), this is usually not the case.
 - The GPU processes the matrix cell-by-cell, and within each cell row-by-row; each row is processed by an individual GPU thread.

Next, the formal definition of the ELLPACK-format is given. For a more instructive approach to §61, refer to figure 4.2.

§61: Definition - ELLPACK – matrix format: Let N_{Zmax} be the maximum number of nonzeros per row; then a compressed-entries - matrix $A \in \mathbb{R}^{N \times N_{Zmax}}$ and a compressed-index - matrix $I \in \mathbb{N}^{N \times N_{Zmax}}$ are defined so that

$$M = \sum_{n=0}^{N-1} \sum_{j=0}^{N_{Zmax}-1} e_{n,I_{n,j}} \cdot A_{n,j}.$$

¹³ilPSP.LinSolvers.monkey.MatrixBase.BCSR

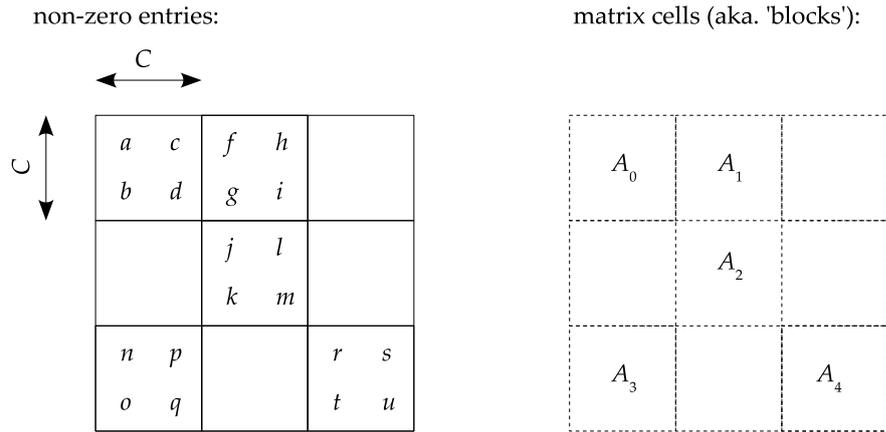
¹⁴Rem.: we chose the name “cells” name to distinct (sub-matrix) blocks form CUDA (thread) blocks

¹⁵ilPSP.LinSolvers.monkey.MatrixBase.CelledFormats.TotalNumberOfCells

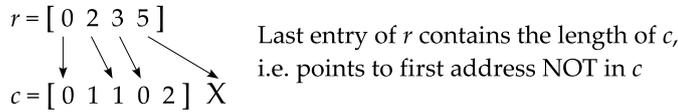
¹⁶ilPSP.LinSolvers.monkey.MatrixBase.BCSR.CellRowStart

¹⁷ilPSP.LinSolvers.monkey.MatrixBase.CelledFormats.CellColumn

¹⁸ilPSP.LinSolvers.monkey.MatrixBase.FormatBase.Val



cell-row-start- and cell-column - vector



cell storage: cell stride S_{C_0} and column stride S_{C_1}

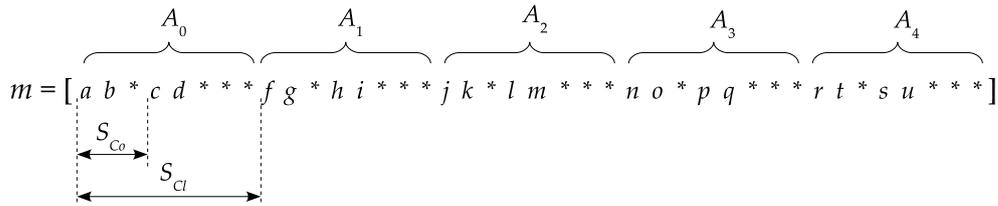


Figure 4.1: Illustration of the Block-CSR – matrix format; the matrix is basically stored cell-by-cell (rem.: “cell” \equiv submatrix). The entries of all occupied, i.e. non-zero cells (in this example A_0, \dots, A_4) are stored in vector m , cell-by-cell and column-per-column within each cell. S_{C_0} and S_{C_1} give the column and cell stride, i.e. the distance, in number-of-entries, from the first entry of one column/cell to the next one. These numbers may be greater than the actual number of entries per column/cell, in order to align the first entries to certain memory addresses, which is beneficial for GPU performance. Vector c denotes the cell-column of the non-zero cells, therefore the length of c is equal to the number of non-zero cells. E.g. for A_2 , which is in cell-row 1 and cell-column 1 (indices start at 0), $c_2 = 1$. The length of vector r is equal to the number of cell-rows plus 1. r_i denotes the index of the first cell in cell-row i . E.g. A_3 is the first occupied cell in cell-row 2, so $r_2 = 3$. Therefore, also the number of occupied cells in cell-row i is $r(i + 1) - r(i)$.

Given a thread-block size S_{Blk} (see ¹⁹) and a column-stride $S_{Co} \geq S_{Blk}$ (see ²⁰), A is encoded as a vector v (see ²¹ and ²²) so that

$$A_{i \cdot S_{Blk} + i', j} = v(i \cdot S_{Co} \cdot N_{Zmax} + j \cdot S_{Co} + i').$$

The compressed-index - matrix I is encoded in a completely analog fashion (see ²³).

§62: Remark - On the ELLPACK – matrix format: The general assumption behind ELLPACK is that almost every row has N_{Zmax} non-zero entries and only a few rows have less entries. This is usually true if M represents the discretization of a partial differential equation on some regular grid, by a finite-volume (FV) or DG method. It may not hold for a finite- or spectral-element method. In FV or DG however, each row or M is typically associated with the discretization of one cell in the grid, and because the number of neighbours of all inner cell is constant, each row has the same number of non-zero entries. Only boundary cells have less neighbours, therefore the associated rows will have less non-zeros. In a typical grid, the number of internal cells is much higher than the number of boundary cells. Therefore, the waste of memory by assigning N_{Zmax} for all rows of M is acceptable.

- At first, ELLPACK just looks like CSR. One important difference to the CSR format is that the matrix is stored column by column. This is possible by assuming that each row has N_{Zmax} entries. Therefore, GPU threads that process consecutive matrix rows (again, one GPU thread processes one matrix row) are able to access consecutive memory addresses, which is a key factor in exploiting the memory bus bandwidth²⁴
- The number S_{Blk} is used as the Cuda block size, see (NVIDIA Corporation 2010), page 8.

Next, a formal definition of the ManualCache-ELLPACK – format will be given. For a more instructive approach to §63, refer to figure 4.3.

¹⁹iIPSP.LinSolvers.monkey.MatrixBase.ELLPACKlike.PacK.RowsPerBlock

²⁰iIPSP.LinSolvers.monkey.MatrixBase.ELLPACKlike.PacK.ColStride

²¹iIPSP.LinSolvers.monkey.MatrixBase.ELLPACKlike.MtxEntries

²²iIPSP.LinSolvers.monkey.MatrixBase.FormatBase.Val

²³iIPSP.LinSolvers.monkey.MatrixBase.ELLPACKmod.Collnd

²⁴ Note that, a Compressed Sparse Column format (CSC), i.e. storing M^T in CSR-form is not a solution to the problem.

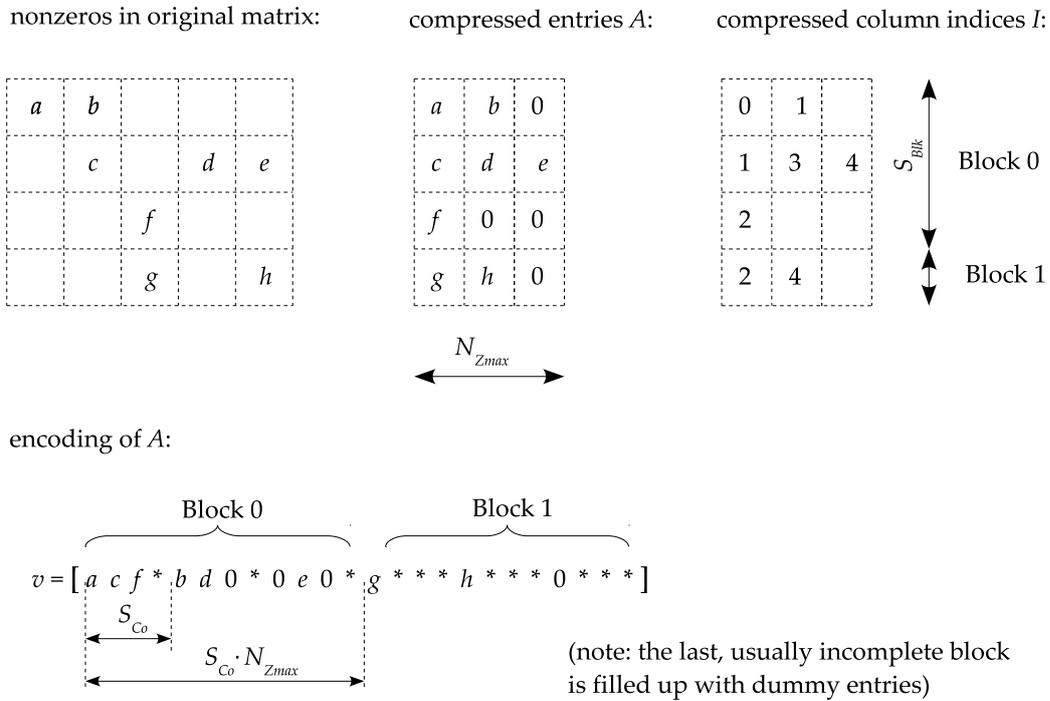


Figure 4.2: Illustration of the (modified) ELLPACK – matrix format for GPU's: At first, the original matrix is compressed into matrix A . Its width, N_{Zmax} , is determined by the maximum number of nonzeros per row in the original matrix (3 in the example). In addition to matrix A , the matrix I stores the column indices (of the corresponding entries in A) of the original matrix. Matrix A is then packed into vector v . Completely the same is performed for I , which is not show in the diagram. At first, A is split into blocks – which correspond to the CUDA thread-blocks – of S_{Blk} rows. S_{Blk} must bot necessarily be a divider of the number of rows, so the last block may have less rows. The blocks are packed into v , column per column. The column stride S_{Co} denotes the distance from the first element of one column to the next one.

§63: Definition - ManualCache-ELLPACK – matrix format: Let N_{Zmax} be the maximum number of nonzeros per row and $S_{Blk} > 0$ the thread-block size²⁵; then a compressed entries matrix $A \in \mathbb{R}^{N \times N_{Zmax}}$, a compressed block-local index matrix $I \in \mathbb{N}^{N \times N_{Zmax}}$ and $T \in \mathbb{N}^{\lceil N/S_{Blk} \rceil \times N_{Zmax}}$, which transforms block-local column indices into global column indices is given so that

$$M = \sum_{n=0}^{N-1} \sum_{j=0}^{N_{Zmax}-1} e_{n, T_{\text{floor}(j/S_{Blk}), I_{n,j}}} \cdot A_{n,j}.$$

The matrices A and I are stored in the same way as described in §61.

§64: Remark - On the ManualCache-ELLPACK – matrix format: This is a modification of the classical, broadly used ELLPACK format and to our knowledge, a novel approach.

For the SpMV operation $b \rightarrow \alpha \cdot M \cdot x + \beta \cdot b$ with scalars $\alpha, \beta \in \mathbb{R}$, it is clear that caching strategies can only be applied to the entries of vector x . Each entry of M is only involved in one multiplication, so caching makes no sense, and also for the entries b one needs just a local variable per GPU thread. By knowing the local-to-global column index transformation T , a block of CUDA threads is able to load those entries of x , which are required for its corresponding block of matrix rows, into the block-shared memory (see (NVIDIA Corporation 2010), page 11) of the GPU, where they can be accessed much faster than from the global memory.

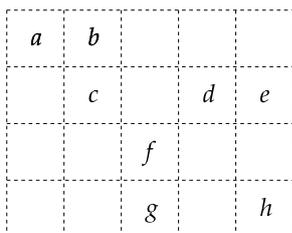
Furthermore, because the column indices I are only local within one block, and are therefore relatively small numbers, it is sufficient to encode them as 16 bit values, saving overall bandwidth.

4.3 MPI – parallelization

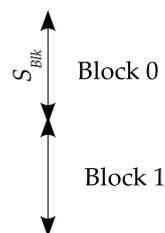
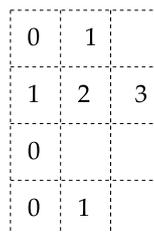
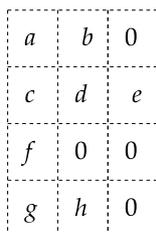
When performing the sparse matrix-vector product (SpMV) on a distributed memory system (in the following only referred to as “MPI - parallel”), one has to specify which rows of the matrix are assigned to which process in the MPI communicator $MPI_{\text{comm}} = \{0, \dots, sz - 1\}$ (see also §34).

²⁵Rem.: corresponds to CUDA thread blocks.

nonzeros in original matrix:



compressed entries *A*:



Transformation between block-local compressed column indices and column indices:

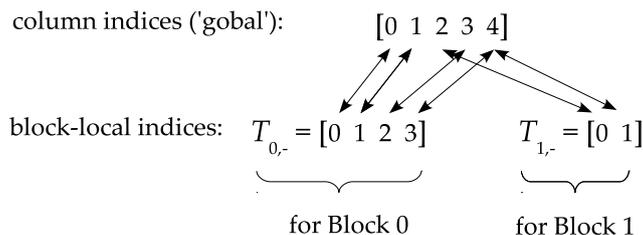


Figure 4.3: Illustration of the ManualCache-ELLPACK – matrix format for GPU’s. The main difference to the classical ELLPACK (for GPU’s, see figure 4.2), is that ‘caching’ is in some sense software-controlled. For block i , all column indices of occupies entries are collected in the $T_{i,-}$ - list. In the product $M \cdot x$, this list contains all indices of x that are required for the computation of the block. During computation, this sub-vector $(x_{T_{i,0}}, \dots, x_{T_{i,N}}) =: x_{sub}$ is loaded into the block-shared memory of the GPU. Consequently, the column-index-matrix contains only indices into x_{sub} . Because of the rather small length of x_{sub} , it is sufficient to encode the indices with 16-Bit values, saving overall bandwidth.

§65: Definition - matrix and vector Partitions²⁶: A partition for the set $\{0, \dots, N - 1\}$ is a monotonically increasing, surjective mapping

$$\text{part} : \{0, \dots, N - 1\} \rightarrow \text{MPI}_{\text{comm}}.$$

A partition of the index set $\{0, \dots, L - 1\}$ of a vector x of length L is called *the partition* of x .

Similar, for the row- and column index sets $\{0, \dots, N - 1\}$ and $\{0, \dots, L - 1\}$ of an $N \times L$ - matrix M , one defines the row- and column partition of M .

§66: Notation : Subsequently, part^{-1} will denote the inverse relation of part , i.e. $\text{part}^{-1}(p) = \{i; \text{part}(p) = i\}$ and $\#\text{part}^{-1}(p)$ denotes the cardinality of $\text{part}^{-1}(p)$, i.e. the number of indices that are assigned to MPI process p .

§67: Definition : For a vector x with L entries and a partition part of its index set, we define

$$[x]_{\text{part}^{-1}(p)} := \left(x_{i_0}, \dots, x_{i_0 + (\#\text{part}^{-1}(p) - 1)} \right)$$

for the part of x that is assigned to MPI process p . Here $i_0 = \sum_{l=0}^{p-1} \#\text{part}^{-1}(l)$.

§68: Remark - representation of a partition: A partition $\text{part} : I \rightarrow \text{MPI}_{\text{comm}}$ is, because of its monotonicity, fully specified by giving, for each MPI rank $p \in \text{MPI}_{\text{comm}}$, the number of indices in I that are assigned with p , i.e. the number $\#\text{part}^{-1}(p)$ (Note that f^{-1} denotes the inverse relation of function f , not an exponent.)

§69: Definition - Internal and external parts of a matrix: Given a matrix $M \in \mathbb{R}^{N \times L}$ and its row- and column partition, part_R and part_C , the following decomposition of M into sub-matrices can be defined:

$$M = \begin{bmatrix} M_0^{\text{Int}} & M_{0,1}^{\text{Ext}} & \dots & M_{0,sz-1}^{\text{Ext}} \\ & \ddots & & \\ & & \ddots & \\ M_{sz-1,0}^{\text{Ext}} & \dots & M_{sz-1,sz-2}^{\text{Ext}} & M_{sz-1}^{\text{Int}} \end{bmatrix}$$

²⁶iIPSP.LinSolvers.Partition

with the internal parts $M_p^{\text{Int}} \in \mathbb{R}^{\#\text{part}_R^{-1}(p) \times \#\text{part}_C^{-1}(p)}$ and the external parts $M_{p,l}^{\text{Ext}} \in \mathbb{R}^{\#\text{part}_R^{-1}(p) \times \#\text{part}_C^{-1}(l)}$.

§70: Remark : With the notation introduced above, for the matrix-vector product $b := M \cdot x$, the part of b that is stored on MPI process p , i.e. $[b]_{\text{part}_R^{-1}(p)}$ is given as

$$[b]_{\text{part}_R^{-1}(p)} = \underbrace{M_p^{\text{Int}} \cdot [x]_{\text{part}_C^{-1}(p)}}_{\text{"internal part"}} + \underbrace{\sum_{\substack{l=0 \\ l \neq p}}^{sz-1} M_{p,l}^{\text{Ext}} \cdot [x]_{\text{part}_C^{-1}(l)}}_{\text{"external part"}}.$$

The internal part can be computed with information that is present on processor p . This part is accelerated by the GPU.

The external part is always computed on the CPU. For the efficiency of the MPI-parallel SpMV implementation it is required that most of the matrices $M_{p,l}^{\text{Ext}}$ are equal to zero. For those $M_{p,l}^{\text{Ext}} \neq 0$, it is still required that the sparsity is much higher than the sparsity of M_p^{Int} , i.e. they contain a lot of zero - rows.

These assumptions usually hold if M is the discretization of a spatial differential operator on a numerical grid with reasonable geometric domain decomposition, consider especially a DG- or finite volume (FV) - method; There, one defines "ghost-" or "external" cells, see figure 3.1. The external matrices $M_{p,l}^{\text{Ext}}$ directly correspond to those ghost-cells. Because of these reasons, it is beneficial for numerical performance to keep the process boundary (see §36), i.e. the total number of ghost cells, minimal.

To reduce the communication to the absolute necessary minimum each MPI process needs to know which entries of $[x]_{\text{part}_C^{-1}(p)}$ must be sent to which other MPI process. These communication lists are stored at²⁷.

§71: Remark - on the role of row- and column partition of a matrix: The row partition of a matrix M defines which rows are stored on which processor. This further implies that the "output" of b of the multiplication with a vector x , i.e. $b := M \cdot x$, will also be distributed with the same partition as the row partition of M .

²⁷ilPSP.LinSolvers.monkey.MatrixBase.SpMVCommPattern

The role of the column partition is different: it defines, on a given MPI process, which columns of the matrix are considered as internal and which that are considered as external. This implies that the partition input of the SpMV - operation, x , must be equal to the column partition of M .

4.4 Implementation issues

4.4.1 Library Design

The most important design goals for the monkey-library are:

- When no GPU is present on a given system, the solvers should be able to operate on the CPU with decent performance.
- The library should not be hard coded for single GPU API (or language, like Cuda (NVIDIA Corporation 2011) or OpenCL (Khronos Group 2011)). At the present time, it is still uncertain what will be the long-term standard for programming GPUs, so it should be possible to switch the GPU programming technology with reasonably few changes to the code.
- For the different implementations (CPU, Cuda, OpenCL), there ideally should be no redundancy. Especially the MPI communication and the computation of the external parts are always performed on CPU and are equal for each implementation.

The building blocks of iterative Krylov-Solvers like Conjugate Gradient are usually the sparse matrix-vector product (SpMV) and some vector operations, e.g. sum of two vectors, scaling a vector by a scalar, etc. compare (Meister 2004).

Taking into account the design goals mentioned above, we came up with the approach presented in Figure 4.4. The primitive operations are defined in the abstract classes for vectors²⁸ and matrices²⁹. A solver algorithm (e.g. the conjugate gradient³⁰) only works with objects of these types and therefore it does not depend on whether the objects and operations “live” on GPU or on CPU. The common base class for matrices and vectors is the *lockable* object³¹. The purpose of its methods Lock() and Unlock() is

²⁸iPSP.LinSolvers.monkey.VectorBase

²⁹iPSP.LinSolvers.monkey.MatrixBase

³⁰iPSP.LinSolvers.monkey.CG

³¹iPSP.LinSolvers.monkey.LockableObject

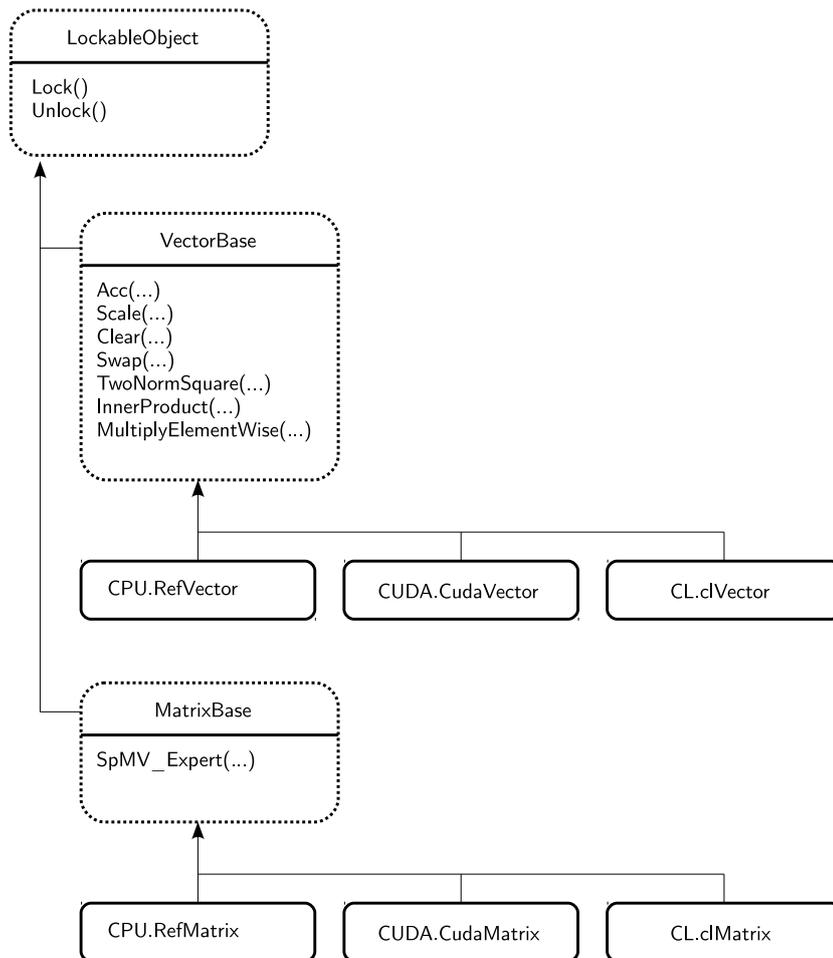


Figure 4.4: Class hierarchy of vector and matrix objects in monkey.

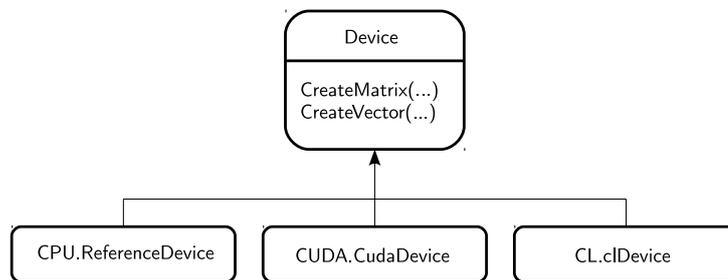


Figure 4.5: Class hierarchy for the factory objects that produce matrix and vector objects.

to load or unload the objects to or from the GPU. For CPU-based objects these methods just perform Garbage Collector – pinning (see sub section 2.2.2 and figure 2.1). Any call to the primitive, accelerated operations of VectorBase and MatrixBase is only valid if the object has been locked before.

4.4.2 Comments on performance

Performance issues for the CPU implementation: Within the CPU reference implementation, a C# feature called “unsafe” code (see (Albahari & Albahari 2010), page 170 and see also online e.g at ³²) is used. Within an unsafe section, the usual C# runtime checks are omitted and pointer arithmetic like in C could be used.

Basic tests have shown that such code executes at the same speed as classical C code with compiler optimizations turned on, at least for SpMV implementations based on the CSR matrix format.

§72: Remark - Performance comparison of CG solvers in HYPRE (Falgout et al. 2006) and monkey: The matrix of the problem results from an interior penalty discretization of a Poisson equation (see (Shahbazi 2005)) on a domain $\omega = (0, 10) \times (-1, 1)$ with 256×128 equidistant cells. 2921 CG iterations were required to reach given residual threshold. The DG polynomial order is 2, so the matrix dimension is $196'608 \times 196'608$. The HYPRE library was compiled with “Microsoft C/C++ Optimizing Compiler Version 15.00.30729.01 for x86”, full optimizations turned on, monkey was compiled with “Microsoft (R) Visual C# 2008 Compiler ver-

³²[msdn.microsoft.com/en-us/library/chfa2zb8\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/chfa2zb8(v=vs.80).aspx), 10th of June 2011

sion 3.5.30729.5420”, again compiler optimization flags turned on. The test was performed in 32-Bit Mode (aka. x86, aka. Intel-32):

	HYPRE	monkey
AMD Phenom II X940 3 Ghz	42.34 sec	42.51 sec
Intel Xeon E5420 2.5 Ghz	51.68 sec	48.17 sec

GPU performance issues First, as already mentioned, one key factor to achieve decent performance on the GPU is ensuring a proper memory layout. Therefore much effort has been invested into data formatting.

Secondly, data transfer between GPU and CPU memory is a critical issue as well. For the special case of iterative sparse solvers the situation is the following: At first, the matrix of the linear system as well as an initial guess for the solution must be loaded onto the GPU (Lock() – operation, see section 4.4.1). Then, usually a few hundred to several thousand SpMV – operations, as well as various vector operations are carried out. When the solver algorithm is finished, i.e. the residual threshold (aka. “convergence criterion”) has been reached, the solution vector needs to be transferred back to the CPU.

At least in the serial case, when there is only one MPI process ($sz = 1$) that uses exactly one GPU, no data, except some scalars, e.g. the result of some inner product, needs to be transferred between CPU and GPU. In this case, data transfer is almost negligible.

However, in the parallel case ($sz > 1$) data must be exchanged via MPI between GPU’s that are assigned to different MPI processes. As the OpenCL implementation is presently (June 2011) still work in progress, subsequently we only focus on the CUDA implementation. Our experiments have shown that the fastest way of exchanging this data in CUDA is to use so - called page-locked memory (see (NVIDIA Corporation 2010), page 33) that can be ‘read-from’ an ‘written-to’ from both, the CPU and the GPU.

The whole CUDA SpMV (Rem.: $b := M \cdot x$) implementation works roughly in the following way:

1. Copy the entries of vector x which must be send to other MPI processes to special page-locked buffers³³. Following this, immediately start computing the internal part on the GPU.

³³ilPSP.LinSolvers.monkey.CUDA.CudaVector.CudaCommVector.h_SendBuffer

2. As soon as these buffers are filled, use non-blocking MPI-send routines to transfer the content of those buffers to other MPI processes.
3. Wait for incoming messages from other MPI processes³⁴. For each received data packet from MPI process l , a method³⁵ computes the product $M_{p,l}^{\text{Ext}} \cdot [x]_{\text{part}_C^{-1}(l)} =: c_l$. Here, p is the MPI rank of the actual process. The results c_l are again stored in page-locked buffers³⁶.
4. Combine internal and external parts c_l on the GPU, as soon as the computation of the internal part and all individual external parts has been finished³⁷.

4.5 Performance measurements

§73: Example and Discussion - Benchmarks of Conjugate Gradient solver for Interior Penalty DG Poisson equation: Benchmarks were done on two different systems:

- *CSC FUCHS*: Cluster FUCHS, operated by Center for Scientific Computing (CSC) of the Goethe University Frankfurt (<http://csc.uni-frankfurt.de/>): 18 Compute Nodes with $2 \times$ Intel(R) Xeon(R) X5460 3.16GHz, 16 GB main memory and $4 \times$ or $6 \times$ Tesla C1060 GPU; nodes are connected via 1 GBit Ethernet. Used .NET runtime: mono 2.8.1;
- *NEC Nehalem*: Cluster NEC Nehalem, operated by High Performance Computing Center Stuttgart (HLRS, <http://www.hlrs.de/>): 32 nodes (equipped with GPU) with Intel(R) Xeon(R) X5560 2.8 GHz, 12 GB main memory and $2 \times$ Tesla S1070 GPU; nodes are connected via DDR Infiniband. Used .NET runtime: mono 2.8.1;

The investigated problem is an Interior Penalty discretization (see (Shahbazi 2005)) of the Poisson problem

$$\begin{cases} \Delta T = 1 & \text{in } \Omega := (0, 10) \times (-1, 1) \times (-1, 1) \\ T = 0 & \text{on } \Gamma_{\text{Diri}} := \{\mathbf{x} \in \partial\Omega; x = 0\} \\ \nabla T \cdot \mathbf{n} = 0 & \text{on } \partial\Omega \setminus \Gamma_{\text{Diri}} \end{cases}$$

with the exact solution

$$T(x, y, z) = 50 - \frac{x^2}{2}.$$

³⁴iIPSP.LinSolvers.monkey.VectorBase.CommVector.WaitCommFinish

³⁵iIPSP.LinSolvers.monkey.MatrixBase.SpMV_External_RecvCallBack

³⁶iIPSP.LinSolvers.monkey.CUDA.CudaMatrix.h_ElementsToAcc

³⁷iIPSP.LinSolvers.monkey.MatrixBase.SpMV_External_Finalize

Equations of that type are an important part of an incompressible Navier-Stokes solver with the pressure projection, see §52. The numerical grid was chosen equidistant with $K_x \times K_y \times K_z$ cells. All problems were solved using the Conjugate Gradient (CG) algorithm. A Local-Block-Elimination (see §111) has been used, which is an explicit preconditioner i.e. it modifies the matrix of the system in advance and causes almost no overhead during the execution of the CG algorithm.

*Summarized Results and Discussion*³⁸:

- *Strong scaling*: For using clusters of GPUs, i.e. multiple GPUs distributed over at least two compute nodes, it is pretty clear that a high-speed and low-latency interconnect is required. As long as all GPUs are situated within one compute node, a quite good scaling is achieved on the CSC FUCHS system, but distribution of the MPI processes over more than one compute node drastically slows down the computation, very likely because of the rather slow Gigabit ethernet interconnect of the system. However, on the NEC Nehalem system (DDR Infiniband interconnect) decent scaling for a problem with 655'360 degrees-of-freedom was achieved when going up from 4 to 8 compute nodes, using 2 GPU's on each node. For detailed results, refer to table 4.5.
- "Fair" CPU to GPU comparison: On System 1 the CPU-to-GPU ratio is 2:1 and on System 2 it is 4:1, and with modern manycore-CPU's this ratio will increase. So, the question whether it is "fair" to compare on CPU core to one GPU remains. On the other hand, it is a fact that in manycore-CPU's all cores (e.g. 12 for the AMD Opteron 6172) share one memory interface. Because SpMV execution speed is bound by memory bandwidth it is also questionable whether manycore-CPU's will achieve a nearly linear scaling.
- GPU versus CPU: Here, always one CPU was matched against one GPU. For the ManualCache-ELLPACK format we achieved a speedup factor in the range of 16.4 to 28.3. ELLPACK came out least efficient, with a speedup in the range of 8.7 to 14.5, and for Block-CSR we got factors between 13.3 and 20.1. The polynomial degree of the DG approximation was 2, yielding 10 degrees-of-freedom per computational cell. With higher DG polynomial degree, and there-

³⁸ It should be mentioned that all thinkable, reasonable benchmarks (various DG degrees vs. various grid size vs. various CPU-to-GPU ratios vs. all implemented matrix formats vs. different GPU generations vs. ...) would probably fill more than 100 pages that would be outdated in at least one year due to rapid hardware evolution. So we restricted ourselves to a few combinations.

fore larger cell size, one would expect the Block-CSR – format to gain more. Detailed results can be found in table 4.5.

Problem	Nodes \times GPU's	runtime NEC-Nehalem [sec]
A	1 \times 2	6.8375
A	2 \times 2	3.6601
A	4 \times 2	2.5860
A	8 \times 2	2.8166
B	1 \times 2	out of mem.
B	2 \times 2	out of mem.
B	4 \times 2	12.901
B	8 \times 2	9.1131

Problem	Nodes \times GPU's	runtime CSC FUCHS [sec]
A	1 \times 2	6.8144
A	1 \times 4	3.8720
A	2 \times 4	5.6199
A	4 \times 4	7.5514
B	1 \times 2	45.967
B	1 \times 4	24.136
B	2 \times 4	18.398
B	4 \times 4	31.550

$$\begin{array}{l}
\text{NEC Nehalem, Pr. A: } 1 \times 2 \xrightarrow{\cdot 1.87} 2 \times 2 \xrightarrow{\cdot 1.41} 4 \times 2 \xrightarrow{\cdot 0.92} 8 \times 2 \\
\text{Pr. B: } \phantom{\xrightarrow{\cdot 1.87}} \phantom{\xrightarrow{\cdot 1.41}} 4 \times 2 \xrightarrow{\cdot 1.46} 8 \times 2 \\
\text{CSC FUCHS, Pr. A: } 1 \times 2 \xrightarrow{\cdot 1.76} 1 \times 4 \xrightarrow{\cdot 0.69} 2 \times 4 \xrightarrow{\cdot 0.74} 4 \times 4 \\
\text{Pr. B: } 1 \times 2 \xrightarrow{\cdot 1.90} 1 \times 4 \xrightarrow{\cdot 1.31} 2 \times 4 \xrightarrow{\cdot 0.58} 4 \times 4
\end{array}$$

Table 4.1: Strong scaling: Comparison of CG solver runtime for different numbers of GPU's for Problem A ($64 \times 64 \times 16$ at DG-degree 2, i.e. 655'360 DOF) and Problem B ($128 \times 128 \times 16$ at DG-degree 2, i.e. 2'621'440 DOF), both using the ManualCache-ELLPACK – format, on CSC FUCHS and NEC Nehalem Systems. The two lower diagrams show speedup factors for different Nodes \times GPU combinations, for Problem A and B, respectively.

Grid $K_x \times K_y \times K_z$	DOF	No. of (G/C)PU's	GPU rtm. [sec]	CPU rtm. [sec]	spdup.
Block-CSR – format:					
$32 \times 32 \times 16$	163'840	1	3.1119	41.444	13.3
		2	1.8065	21.896	12.1
		4	1.0021	17.210	17.2
$64 \times 64 \times 16$	655'360	1	18.740	268.87	14.3
		2	9.9742	136.30	13.7
		4	5.4616	109.67	20.1
ELLPACK – format:					
$32 \times 32 \times 16$	163'840	1	4.5017	41.444	9.2
		2	2.4870	21.896	8.7
		4	1.3162	17.210	13.3
$64 \times 64 \times 16$	655'360	1	27.635	268.87	9.7
		2	14.256	136.30	9.8
		4	7.5482	109.67	14.5
ManualCache-ELLPACK – format:					
$32 \times 32 \times 16$	163'840	1	2.1543	41.444	19.3
		2	1.3396	21.896	16.4
		4	0.8936	17.210	19.4
$64 \times 64 \times 16$	655'360	1	12.570	268.87	21.4
		2	6.8331	136.30	20.4
		4	3.8767	109.67	28.3

Table 4.2: Comparison of GPU vs. CPU runtime for CG solver. On the CPU, only the CSR format is implemented. Basic tests have shown that on CPU's, likely because of single-threaded execution, efficient caching and pre-fetching the influence of the storage format is negligible.

5 An Incompressible Single-Phase Navier-Stokes – solver

The solver (NSE2b) presented in this chapter is restricted to constant physical properties, i.e. constant density and viscosity in space and time and therefore called a “Single-Phase Navier-Stokes – solver”.

5.1 Notation

§74: Notation : Continuing the notation introduced in chapter 3 we recall/define the following symbols ...

- the spatial dimension $D \in \{2, 3\}$
- the (computational) domain $\Omega \subset \mathbb{R}^D$
- the Sobolev space $H^l(\Omega) = \left\{ f \in L^2(\Omega); \sum_{|\alpha| \leq l} \|\partial^\alpha f\|_2^2 < \infty \right\}$, where α denotes a multi-index.
- the trace operator $-|_\Gamma : H^l(\Omega) \rightarrow L^2(\Gamma)$ (see (Triebel 1980)), for some sufficiently smooth $\Gamma \subset \partial\Omega$, see §46.

We briefly recall the definition and basic properties of semigroup theory, because a similar notation will be used in subsequence. For further reading about semigroups, we refer to (Renardy 2004).

§75: Definition - Strongly continuous semigroup: (From (Renardy 2004))

Let X be a Banach space. A family $\mathbb{E} = \{\mathcal{E}(t); t \in \mathbb{R}_{\geq 0}\}$ of bounded linear operators in X (i.e. $\mathcal{E}(t) : X \rightarrow X$) is called a strongly continuous semigroup or C_0 - semigroup, if it satisfies the following properties:

1. $\mathcal{E}(t_1 + t_2) = \mathcal{E}(t_1)\mathcal{E}(t_2)$ for $t_1, t_2 \geq 0$.
2. $\mathcal{E}(0) = id$.
3. For every $u_0 \in X$, the mapping $\mathbb{R}_{\geq 0} \ni t \mapsto \mathcal{E}(t)u_0 \in X$ is continuous.

§76: Corollary - Infinitesimal generator: For some bounded operator A on Banach space X (e.g. a A can be an $n \times n$ - matrix on the space $X = \mathbb{R}^n$) a semigroup \mathbb{E} is given by the exponential series, i.e. $\mathbb{E} =$

$\{\exp(t \cdot A); t \in \mathbb{R}_{\geq 0}\}$. For $v_0 \in X$, the element $\exp(t \cdot (-A))v_0$ is a solution to the initial-value problem

$$\frac{\partial}{\partial t}v + Av = 0 \text{ with } v(t=0) = v_0.$$

(Remark: the concept of infinitesimal generators can be extended to unbounded operators A by the Hille-Yosida theorem.)

In similar fashion to the classical semigroup theory, we introduce a notation for nonlinear (partial) differential equations. These symbols will be useful in the notation of splitting schemes.

§77: Notation - Initial-value problems in semigroup-like fashion: On a Banach space X suppose the well-posed initial-value problem

$$\frac{\partial}{\partial t}u + Op(u) = 0 \text{ with } u(t=0) = u_0 \quad (5.1)$$

with an operator $Op : Y \rightarrow X$. Let be

$$\mathbb{R}_{\geq 0} \ni t \mapsto u(t) \in X$$

the unique solution to this problem. (Remark: (1) $u \in C^1(\mathbb{R}_{>0}, F(\Omega))$; (2) For linear operators, we know that Y must be a dense subset of X , to yield a well-defined theory. Since our goal is only the introduction of some notation, we do not go further.) We call

$$X \ni v \mapsto \mathcal{E}_{BC}^{\{\Delta t, Op(-)\}}(u_0) := u(\Delta t) \in X$$

a time-integrator of operator Op , under boundary condition BC, for an initial value v and a time Δt .

5.2 High Level view on the projection scheme

§78: Notation : For the velocity vectors, the symbols \mathbf{v} or \mathbf{u} will be used; usually, \mathbf{v} denotes the velocity of the “real” Navier-Stokes problem, while \mathbf{u} denotes the velocity in the numerical method. We identify, in 2D $\mathbf{v} = (v_1, v_2) = (u, v)$, as well as $\mathbf{u} = (u_1, u_2) = (u, v)$ and in 3D $\mathbf{v} = (v_1, v_2, v_3) = (u, v, w)$, as well as $\mathbf{u} = (u_1, u_2, u_3) = (u, v, w)$. Further, p will denote the pressure in the “real” NSE while Ψ denotes the pressure in the numerical method. By doing so, confusion with the DG polynomial degree p of the velocity components should be avoidable. The DG polynomial degree of the pressure is denoted as l .

§79: Axiom - Incompressible Navier-Stokes equation: An incompressible flow with constant fluid properties (density and viscosity) is described by a time- and space dependent vector field $\mathbf{v} \in \mathcal{C}^1(\mathbb{R}_{\geq 0}, H^2(\Omega)^D)$ for velocity and a scalar field $p \in \mathcal{C}^1(\mathbb{R}_{\geq 0}, H^1(\Omega))$ for pressure. The evolution of these properties in time is an initial-boundary value problem in the sense of definition 77 and described by the Navier-Stokes equation (NSE):

$$\text{in } \Omega \begin{cases} \partial_t \mathbf{v} + \text{div}(\mathbf{v} \otimes \mathbf{v}) + \nabla p = \frac{1}{\text{Re}} \Delta \mathbf{v} \\ \text{div}(\mathbf{v}) = 0 \end{cases}$$

Here, $[\mathbf{v} \otimes \mathbf{v}]_{d,l} = v_d \cdot v_l$. It is physically intuitive to define a disjoint partition of the boundary to distinct the different types of boundaries,

$$\partial\Omega = \Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}} \cup \Gamma_{\text{Out}} \cup \Gamma_{\text{POut}}.$$

where distinct types of boundary conditions hold, more specifically:

- *Inlet*¹: Γ_{Inl} : a velocity is given, i.e. $\mathbf{v}|_{\Gamma_{\text{Inl}}} = \mathbf{u}_{\text{Inl}}$;
- *Wall*²: Γ_{Wall} : also referred to as no-slip - boundary condition; a velocity is given, i.e. $\mathbf{v}|_{\Gamma_{\text{Wall}}} = \mathbf{u}_{\text{Wall}}$, with the restriction that \mathbf{u}_{Wall} must be tangential to the boundary, i.e. there is no flow through the wall;
- *Outflow*³: Γ_{Out} : physically, nothing is known (see discussion below);
- *Pressure Outlet*⁴: Γ_{POut} : at the pressure outlet, a Dirichlet - value for the pressure is known, i.e. $p|_{\Gamma_{\text{POut}}} = p_{\text{POut}}$.

It should be noted that all of the above mentioned boundary conditions, except the one for walls, have no representation in nature. They are necessary mathematical hints to limit the size of the computational domain.

(Remark: For the case $D = 2$ some existence results are known, but for $D = 3$, existence of solutions for the NSE is still an unsolved problem, so we do go deeper into the specification of boundary conditions and their implications on existence and uniqueness. When implementing a numerical solver, it is obvious that more boundary information than described above will be necessary, and therefore further assumptions will be made.)

§80: Notation : Subsequently, $\mathbf{u}_{\text{Inl,Wall}} = \begin{cases} \mathbf{u}_{\text{Inl}} & \text{on } \Gamma_{\text{Inl}} \\ \mathbf{u}_{\text{Wall}} & \text{on } \Gamma_{\text{Wall}} \end{cases}$

¹NSE2b.BcType.Velocity_Inlet

²NSE2b.BcType.Wall

³NSE2b.BcType.Outflow

⁴NSE2b.BcType.Pressure_Outlet

§81: Notation : We define the transport operator of the NSE:

$$\begin{aligned} \mathcal{N} : H^1(\Omega)^D &\rightarrow L_2(\Omega)^D \\ \mathbf{v} &\mapsto \mathcal{N}(\mathbf{v}) := \operatorname{div}(\mathbf{v} \otimes \mathbf{v}). \end{aligned}$$

§82: Notation : For $y := f(x)$, we may also write $x \xrightarrow{f} y$, for $y := x + k$ we may write $x \xrightarrow{+k} y$.

§83: Corollary - Projection Scheme for the NSE: For a timestep $\Delta t > 0$, and an initial value $\mathbf{u}^0 \in (H^2(\Omega))^D$ with $\operatorname{div}(\mathbf{u}^0) = 0$, the so-called Projection method is defined as

$$\mathbf{u}^0 \xrightarrow{\mathcal{E}\{\Delta t, \mathcal{N} - \frac{1}{\operatorname{Re}} \Delta\}} \mathbf{u}^* \xrightarrow{-\Delta t \cdot \nabla \Psi} \mathbf{u}^1,$$

where the pressure $\Psi \in L^2(\Omega)$ is a solution of the Poisson equation

$$\Delta \Psi = \frac{1}{\Delta t} \operatorname{div}(\mathbf{u}^*). \quad (5.2)$$

(Remark: (1) Boundary conditions will be discussed in subsequence. (2) Eq. 5.2 results from inserting $\mathbf{u}^1 = \mathbf{u}^* - \Delta t \cdot \nabla \Psi$ into the continuity equation $\operatorname{div}(\mathbf{u}^1) = 0$.) The first part of the method is called *Predictor*, while the second one is referred to as *projection* or *corrector*. The pair (\mathbf{u}^1, Ψ) is an approximation to the solution $(\mathbf{v}(\Delta t, -), p(\Delta t, -))$ of the NSE for initial value \mathbf{u}^0 , i.e. for some reasonable norm $\| - \|_x$ the inequality

$$\|\mathbf{u}^1 - \mathbf{v}(\Delta t, -)\|_x \leq \Delta t^k \cdot C_1$$

holds with $k \geq 1$. (For more details on convergence, refer to (E & Liu 1995) where the Projection method is extensively discussed).

§84: Remark - Boundary condition for Poisson equation: In (E & Liu 1995) it is suggested to use a Neumann boundary condition

$$\nabla \Psi \cdot \mathbf{n} = g_{\text{Neu}} \text{ on } \partial\Omega \setminus \Gamma_{\text{POut}} \quad (5.3)$$

for the Poisson equation (Eq. 5.2). Its value g_{Neu} can be derived from the momentum equation as follows: Since $\Psi \approx p$, E and Liu suggest to use the momentum equation

$$\nabla p = \frac{1}{\operatorname{Re}} \Delta \mathbf{u} - \mathcal{N}(\mathbf{u}) - \frac{\partial}{\partial t} \mathbf{u} \quad (5.4)$$

to compute g_{Neu} . Using the identity $\Delta \mathbf{u} = \text{curl}(\text{curl}(\mathbf{u}))$ (only for $\text{div}(\mathbf{u}) = 0$), and the assumption that $\frac{\partial}{\partial t} \mathbf{u} = 0$ on $\partial\Omega$ one gains from Eq. 5.4 that

$$g_{\text{Neu}} := \left(\frac{1}{\text{Re}} \text{curl}(\text{curl}(\mathbf{u}^0)) - \mathcal{N}(\mathbf{u}^0) \right) \cdot \mathbf{n} \text{ on } \partial\Omega \setminus (\Gamma_{\text{POut}} \cup \Gamma_{\text{Out}}). \quad (5.5)$$

Additionally, one has to guarantee the *existence* and *uniqueness* of a solution to the Poisson problem (Eq. 5.2), if Neumann boundary conditions are used everywhere ($\Gamma_{\text{POut}} = \{\}$). It is clear that a solution is only unique up to a constant, so an additional reference point must be added, e.g. $\Psi(\mathbf{x}_0) = 0$ for some $\mathbf{x}_0 \in \Omega$. From applying the Gaussian integral theorem on Eq. 5.2 and substituting the boundary condition Eq. 5.3 one gains a necessary condition for the existence of a solution to the Poisson equation:

$$\int_{\partial\Omega} \mathbf{u}^* \cdot \mathbf{n} dS = \int_{\partial\Omega} g_{\text{Neu}} dS. \quad (5.6)$$

Therefore, on Γ_{Out} one chooses $g_{\text{Neu}} = c$, with a constant $c \in \mathbb{R}$ for which

$$c \cdot \int_{\Gamma_{\text{Out}}} 1 dS = \int_{\Gamma_{\text{Out}}} \mathbf{u} \cdot \mathbf{n} dS + \int_{\partial\Omega \setminus \Gamma_{\text{Out}}} \mathbf{u} \cdot \mathbf{n} - g_{\text{Neu}} dS \quad (5.7)$$

holds, to fulfil the compatibility condition 5.6.

However, if $\Gamma_{\text{Out}} = \{\}$ and $\Gamma_{\text{POut}} = \{\}$ ⁵, the only choice for g_{Neu} is to set

$$g_{\text{Neu}} = \mathbf{u}^* \cdot \mathbf{n} \text{ on } \partial\Omega.$$

Within the following paragraph, we discuss the error that is introduced by the temporal splitting for a specific example. This error is much related to the boundary conditions for the intermediate velocity field. A modified boundary condition (found in (Kim & Moin 1985)) will be presented. The complete algorithm of the projection scheme that is used in the BoSSS NSE2b solver, including boundary conditions for all equations, but still leaving open the question of spatial discretization, is summarized in paragraph 86.

§85: Example and Discussion : Consider a channel flow with domain

$$\Omega = (0, 10) \times (-1, 1)$$

⁵In such examples, to be reasonable, also $\Gamma_{\text{Inl}} = \{\}$, i.e. a “closed” system is simulated.

and the boundaries

$$\begin{aligned}\Gamma_{\text{Inl}} &= \{(x, y) \in \overline{\partial\Omega}; x = 0\}, \\ \Gamma_{\text{Wall}} &= \{(x, y) \in \overline{\partial\Omega}; y = 1 \text{ or } y = -1\} \text{ and} \\ \Gamma_{\text{POut}} &= \{(x, y) \in \overline{\partial\Omega}; x = 10\}.\end{aligned}$$

As boundary conditions we specify

$$\mathbf{u}_{\text{Inl}} = (1 - y^2, 0)^T, \quad \mathbf{u}_{\text{Wall}} = 0 \text{ and } p_{\text{POut}} = 0.$$

Then, for an initial value

$$\mathbf{u}^0 = (1 - y^2, 0)^T$$

a solution to the NSE is easily obtained, namely

$$\mathbf{v}(t, \mathbf{x}) = (1 - y^2, 0)^T \text{ and } p(t, \mathbf{x}) = \frac{-2}{\text{Re}} \cdot x + \frac{20}{\text{Re}}.$$

Suppose momentarily that for the predictor on Γ_{Wall} the boundary condition $\mathbf{u}_{\text{Wall}} = 0$ would be used. This would imply that $\mathbf{u}^*|_{\Gamma_{\text{Wall}}} = 0$. Now, assume $\Psi = p$, i.e. the method finds the exact solution for the pressure, it immediately follows that $\mathbf{u}^1|_{\Gamma_{\text{Wall}}} \neq 0$, because $\mathbf{u}^1 = \mathbf{u}^* - \Delta t \cdot \nabla \Psi$.

On the other hand, if one wants to construct a method that enforces $\mathbf{u}^1|_{\Gamma_{\text{Wall}}} = 0$, this would require that $(\nabla \Psi)|_{\Gamma_{\text{Wall}}} = 0$ which is clearly a bad approximation for the pressure.

The first solution to this dilemma was, up to our knowledge, proposed in (Kim & Moin 1985). They propose a boundary condition

$$\mathbf{u}^* = \mathbf{u}_{\text{Inl,Wall}} + \Delta t \cdot \mathbf{p}_{\text{Neu}} \text{ on } \Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}},$$

for the intermediate step⁶, where

$$\mathbf{p}_{\text{Neu}} := \frac{1}{\text{Re}} \text{curl}(\text{curl}(\mathbf{u}^0)) - \mathcal{N}(\mathbf{u}^0). \quad (5.8)$$

Note that $g_{\text{Neu}} = \mathbf{n} \cdot \mathbf{p}_{\text{Neu}}$, compare Eq. 5.5. Using the modified boundary condition for the intermediate velocity, one gets $\mathbf{u}^* = \left(1 - y^2 - \frac{2\Delta t}{\text{Re}}, 0\right)^T$, which yields the solution $\mathbf{u}^1 = \mathbf{u}^0$, assuming that the computed pressure $\Psi = p$. So, in this particular example, no splitting error is induced by the projection scheme. However, in the general case, some splitting error may remain.

⁶ The general idea of modified intermediate boundary conditions for temporal splitting schemes, in order to increase their accuracy, can be traced back to (LeVeque & Olinger 1983).

§86: Summary - High-level view on the full projection method: Up to this point, the full algorithm of the projection method that is implemented in BoSSS, without going into details about spatial discretization, reads as follows:

1. Compute \mathbf{p}_{Neu} on $\partial\Omega$ according to Eq. 5.8.
2. Perform the prediction step:

$$\mathbf{u}_0 \xrightarrow{\mathcal{E}^{\{\Delta t, \mathcal{N} + \frac{1}{\text{Re}} \Delta\}}} \mathbf{u}^* \quad (5.9)$$

under boundary condition $\mathbf{u}|_{\Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}}} = \mathbf{u}_{\text{modDir}}(t, -)$ with modified Dirichlet boundary value

$$\mathbf{u}_{\text{modDir}}(t, -) := \mathbf{u}_{\text{Inl,Wall}} + t \cdot \mathbf{p}_{\text{Neu}} \text{ on } \Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}}. \quad (5.10)$$

3. Compute the value $g_{\text{Neu}} = \mathbf{p}_{\text{Neu}} \cdot \mathbf{n}$ of the Neumann pressure boundary condition (see Eq. 5.8 resp. 5.5). In order to fulfill the compatibility condition (Eq. 5.6) some modification of g_{Neu} on the Outflow domain Γ_{Out} may be necessary:
 - If there is a Dirichlet region for the pressure, i.e. a pressure outlet resp. $\Gamma_{\text{POut}} \neq \{\}$, g_{Neu} is defined by Eq. 5.5 on $\partial\Omega \setminus \Gamma_{\text{Out}}$. In this case, the compatibility condition does not apply.
 - If there is *no* Dirichlet region for the pressure, but at least an outflow resp. Neumann region for the pressure, i.e. $\Gamma_{\text{POut}} = \{\}$ and $\Gamma_{\text{Out}} \neq \{\}$, the compatibility condition does apply: compute $c \in \mathbb{R}$ according to Eq. 5.7 and re-define

$$g_{\text{Neu}} := \begin{cases} \text{as in Eq. 5.5} & \text{on } \partial\Omega \setminus (\Gamma_{\text{POut}} \cup \Gamma_{\text{Out}}) \\ c & \text{on } \Gamma_{\text{Out}} \end{cases}.$$

- If there is neither a pressure outlet nor an outflow region, i.e. $\Gamma_{\text{POut}} = \Gamma_{\text{Out}} = \{\}$, which is the case e.g. in periodic flows, there is no freedom to compute a boundary condition like in the previous case. Therefore, set $g_{\text{Neu}} = \mathbf{u}^* \cdot \mathbf{n}$ on $\partial\Omega$ to fulfill the compatibility condition.
4. Solve the Poisson equation:

$$\begin{cases} \Delta\Psi = \frac{1}{\Delta t} \text{div}(\mathbf{u}^*) & \text{in } \Omega \\ \nabla\Psi \cdot \mathbf{n} = g_{\text{Neu}} & \text{on } \partial\Omega \setminus \Gamma_{\text{POut}} \\ \Psi = p_{\text{POut}} & \text{on } \Gamma_{\text{POut}} \end{cases} \quad (5.11)$$

5. Compute $\mathbf{u}^1 = \mathbf{u}^* - \Delta t \cdot \nabla\Psi$.

5.3 Spatial discretization

Up to now, the projection scheme is still defined by differential operators. The discretization of those operators by means of the Discontinuous Galerkin method will be the issue of this section. The upcoming two paragraphs define/recall the notation of DG method. Subsequently, the DG discretization of nonlinear operator \mathcal{N} , viscous operator $\frac{-1}{\text{Re}}\Delta$ and Laplace operator, divergence and gradient will be given.

§87: Reminder : Continuing the notation introduced in chapter 3, recall the following symbols ...

- the computational grid: $\mathfrak{K} = (K_0, \dots, K_{J-1})$ with $\bigcup_{j=0}^{J-1} \overline{K_j} = \overline{\Omega}$.
- the space of Discontinuous Galerkin (DG) functions of degree p , $DG_p(\mathfrak{K})$
- an orthonormal basis of $DG_p(\mathfrak{K})$,

$$\{\phi_{j,n}; j = 0, \dots, J-1 \text{ and } n = 0, \dots, N-1\}$$

that fulfils $\text{supp}(\phi_{j,n}) = \overline{K_j}$

- $\underline{f} \in DG_p(\mathfrak{K})$ can be represented as $f(\mathbf{x}) = \sum_{n=0}^{N-1} \Phi_{j,n} \cdot \tilde{f}_{j,n}$ for $\mathbf{x} \in K_j$. The numbers $\tilde{f}_{j,n}$ are called the DG coordinates of \underline{f} .
- for DG fields $\underline{U} := (\underline{u}_0, \dots, \underline{u}_{\Lambda-1}) \in (DG_p(\mathfrak{K}))^\Lambda$, with DG coordinates $\tilde{u}_{\delta,j,n}$, $0 \leq \delta < \Lambda$ we define the DG coordinate vector $\tilde{U} \in \mathbb{R}^{\Lambda \cdot J \cdot N \times 1}$ so that $\tilde{U}_{\text{map}(\delta,j,n)} = \tilde{u}_{\delta,j,n}$ for the bijective mapping $\text{map} : \{0, \dots, \Lambda-1\} \times \{0, \dots, J-1\} \times \{0, \dots, N-1\} \rightarrow \{0, \dots, \Lambda \cdot J \cdot N - 1\}$, i.e.

§88: Notation : For a matrix $\mathbf{Q} \in \mathbb{R}^{K \times K}$, its spectrum (set of all Eigenvalues) is denoted as $\text{spec}(\mathbf{Q})$.

§89: Corollary - DG discretization of transport terms: For the discretization of the transport operator \mathcal{N} , which will be in subsequence referred to as

$$DG_p(\mathfrak{K})^D \ni \underline{\mathbf{u}} \mapsto \mathbf{N}_{\mathbf{w}}(\underline{\mathbf{u}}) =: \underline{\mathbf{h}} \in DG_p(\mathfrak{K})^D,$$

respectively the induced coordinate mapping

$$\mathbb{R}^{\dim(\text{Dom})} \ni \tilde{\mathbf{u}} \mapsto \tilde{\mathbf{N}}_{\mathbf{w}}(\tilde{\mathbf{u}}) =: \tilde{\mathbf{h}} \in \mathbb{R}^{\dim(\text{Cod})},$$

we use Riemannian fluxes as described in (Shahbazi et al. 2007). The parameter \mathbf{w} denotes the boundary value on Γ_{Inl} and will be substituted with \mathbf{u}_{Inl} .

As a remainder, note that the flux for the d -th component of \mathcal{N} is given by $\mathbf{f}_d(\mathbf{x}, \mathbf{u}) = u_d \cdot \mathbf{u} = [\mathcal{N}(\mathbf{u})]_d$.

The discretized transport operator \mathbf{N}_w is given by \mathbf{f}_d and the following Riemannian (refer to notation of §49):

- In interior edges $\mathfrak{E}_{\text{int}}$, the Riemannian

$$\widehat{f}_d(\mathbf{x}, \mathbf{u}^{\text{in}}, \mathbf{u}^{\text{out}}, \mathbf{n}_x) = \mathbf{n}_x \cdot \left(\frac{1}{2} \cdot (\mathbf{u}^{\text{in}} + \mathbf{u}^{\text{out}}) \right) + \lambda \cdot (u_d^{\text{in}} - u_d^{\text{out}})$$

is used, with

$$\lambda := \max \left\{ |\lambda|; \lambda \in \text{spec} \left(\mathbf{Q} \left(\overline{\mathbf{u}^{\text{in}}} \right) \right) \cup \text{spec} \left(\mathbf{Q} \left(\overline{\mathbf{u}^{\text{out}}} \right) \right) \right\},$$

where the matrix

$$\mathbf{Q}(\mathbf{u}) = \left[\frac{\partial}{\partial u_j} \left(\sum_{l=1}^D u_i u_l n_l \right) \right]_{\substack{i=1, \dots, D \\ j=1, \dots, D}}$$

and $\overline{\mathbf{u}^{\text{in}}}$, $\overline{\mathbf{u}^{\text{out}}}$ denote the mean values of \mathbf{u}^{in} , \mathbf{u}^{out} within $K^{\text{in}}, K^{\text{out}}$, respectively.

- On Γ_{Wall} , $\widehat{f}_d(\mathbf{x}, \mathbf{u}^{\text{in}}, \mathbf{n}_x) = 0$ (Remark: the physical interpretation of this is that there is no flux through walls).
- On $\Gamma_{\text{Out}} \cup \Gamma_{\text{POut}}$, $\widehat{f}_d(\mathbf{x}, \mathbf{u}^{\text{in}}, \mathbf{n}_x) = u_d^{\text{in}} \cdot \mathbf{u}^{\text{in}} \cdot \mathbf{n}_x$.
- On Γ_{Inl} , $\widehat{f}_d(\mathbf{x}, \mathbf{u}^{\text{in}}, \mathbf{n}_x) = w_d \cdot \mathbf{w} \cdot \mathbf{n}_x$.

§90: Remark - DG discretization of viscous terms: Since the Laplacian of a vector field fully decouples for the components of \mathbf{u} , i.e.

$$\Delta \mathbf{u} = (\Delta u_1, \dots, \Delta u_D)^T,$$

its discretization can be done component-by-component.

The viscous operator for the d -th velocity component is given by

$$H^2(\Omega) \ni u_d \mapsto \frac{-1}{\text{Re}} \Delta u_d \in L^2(\Omega),$$

together with Dirichlet boundary conditions on Inlet and Wall, and Neumann boundary conditions on Outlet- and 'Pressure Outlet' - regions, i.e.

$$\begin{cases} u_d|_{\Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}}} = [\mathbf{u}_{\text{modDir}}(t, -)]_d & \text{on } \Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}} \\ (\mathbf{n} \cdot \nabla u_d)|_{\Gamma_{\text{Out}} \cup \Gamma_{\text{POlt}}} = 0 & \text{on } \Gamma_{\text{Out}} \cup \Gamma_{\text{POlt}} \end{cases} .$$

for some $t > 0$. Refer to Eq. 5.10 for the definition of $\mathbf{u}_{\text{modDir}}$.

We notate its consistent spatial discretization as

$$DG_p(\mathfrak{K}) \ni \underline{u}_d \mapsto \mathbf{D}_{\frac{-1}{\text{Re}}\Delta} \underline{u}_d \in DG_p(\mathfrak{K})$$

and use the interior penalty method, see §51 and §52; in the notation of these paragraphs, $\Gamma_{\text{Diri}} := \Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}}$, $\Gamma_{\text{Neu}} := \Gamma_{\text{Out}} \cup \Gamma_{\text{POlt}}$ and $\nu := \frac{-1}{\text{Re}}$.

Because of its linearity, the operator can be expressed as a matrix that acts on the DG-coordinates of some DG-field. It should be noted that (1st) the Dirichlet boundary condition depends on time but this affects only the affine offset of the operator, not the operator matrix itself; and that (2nd) the matrix of the operator does not depend on the spatial component index d , while the affine vector does.

This is important for the efficiency of the implementation: it is only necessary to do the expensive assembly of the operator matrix once, namely at start-up. The same matrix can be used for all velocity components. Only the affine part of the operator needs to be recomputed in every timestep, for each velocity component.

Hence, the d -th component of the discretized viscous operator will be denoted as

$$\tilde{\mathbf{D}}_{\frac{-1}{\text{Re}}\Delta} \tilde{u}_d = \mathcal{M}_{\frac{-1}{\text{Re}}\Delta} \cdot \tilde{u}_d + b_d(t)$$

within the remains of this chapter.

§91: Notation - standard-basis-vector: $\mathbf{e}_d := [\delta_{d,j}]_{j=1,\dots,D}$

§92: Remark - DG discretization of the pressure gradient: The pressure gradient operator, written in the quite unusual form

$$H^1(\Omega) \ni \Psi \mapsto [\text{div}(\mathbf{e}_d \cdot \Psi)]_{d=1,\dots,D} \in \left(L^2(\Omega)\right)^D$$

can be discretized using central differences. This "conservative" form of the operator helps to directly apply the framework of §49. In the notation introduced there, for the flux $\mathbf{f}_d(\Psi) = \mathbf{e}_d \cdot \Psi$ the Riemannians are chosen as

- $\widehat{f}_d(\mathbf{x}, \Psi^{\text{in}}, \Psi^{\text{out}}, \mathbf{n}) = \frac{1}{2}(\Psi^{\text{in}} + \Psi^{\text{out}}) \cdot \mathbf{e}_d \cdot \mathbf{n}$ on interior edges and
- $\widehat{f}_d(\mathbf{x}, \Psi^{\text{in}}, \mathbf{n}) = \Psi^{\text{in}} \cdot \mathbf{e}_d \cdot \mathbf{n}$ on edges on the non-Dirichlet region $\partial\Omega \setminus \Gamma_{\text{POlt}}$ and
- $\widehat{f}_d(\mathbf{x}, \Psi^{\text{in}}, \mathbf{n}) = (\Psi^{\text{in}} + p_{\text{POut}}) \cdot \mathbf{e}_d \cdot \mathbf{n}$ on edges on the Dirichlet-pressure region Γ_{POlt} .

This numerical gradient, written as

$$DG_l(\mathfrak{K}) \ni \underline{\Psi} \mapsto \mathbf{G}_\nabla(\underline{\Psi}) \in DG_p(\mathfrak{K})^D$$

is affine linear and its matrix and affine vector are denoted as

$$\widetilde{\mathbf{G}}_\nabla(\check{\Psi}) = \mathcal{M}_\nabla \check{\Psi} + c.$$

§93: Remark - DG discretization of velocity divergence operator: For the velocity divergence operator

$$H^1(\Omega)^D \ni \mathbf{u}^* \mapsto \text{div}(\mathbf{u}^*) \in L^2(\Omega)$$

its numerical discretization

$$DG_p(\mathfrak{K})^D \ni \underline{\mathbf{u}}^* \mapsto \mathbf{div}(\underline{\mathbf{u}}^*) \in DG_l(\mathfrak{K})$$

is computed by taking the local divergence in each cell and a penalization of jumps between cells. The reason for this choice is given in §94. Linear and affine part of \mathbf{div} are denoted by

$$\widetilde{\mathbf{div}}(\tilde{\mathbf{u}}^*) = \mathcal{M}_{\text{div}} \cdot \tilde{\mathbf{u}}^* + d(\mathbf{u}_{\text{Bnd}})$$

for some boundary velocity $\mathbf{u}_{\text{Bnd}} \in L^2(\Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}} \cup \Gamma_{\text{Out}})^D$ on which the affine part and only the affine part depends on. Up to now, \mathbf{u}_{Bnd} remains unspecified. It will be given in §98. It should be noted that $d(\mathbf{u}_{\text{Bnd}})$ depends linearly on \mathbf{u}_{Bnd} .

In detail the operator is fully characterized by

$$\left[\widetilde{\mathbf{div}}(\tilde{\mathbf{u}}^*) \right]_{\text{map}(0,j,m)} = \int_{K_j} \text{div}(\mathbf{u}) \cdot \Phi_{j,m} \, d\mathbf{x} - \int_{\partial K_j} \mathbf{n} \cdot \mathbf{f} \cdot \Phi_{j,m} \, dS,$$

where the function \mathbf{f} – which causes the penalization of velocity jumps in between the cells – is given by

- $\mathbf{f} = \mathbf{f}(\mathbf{u}^{\text{in}}, \mathbf{u}^{\text{out}}) = \frac{1}{2}(\mathbf{u}^{\text{in}} - \mathbf{u}^{\text{out}})$ on interior edges,
- $\mathbf{f} = \mathbf{f}(\mathbf{u}^{\text{in}}) = (\mathbf{u}^{\text{in}} - \mathbf{u}_{\text{Bnd}})$ on edes in $\Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}} \cup \Gamma_{\text{Out}}$ and

- $\mathbf{f} = 0$ on edges in Γ_{POut} .

It should be noted that \mathbf{f} is not a Riemannian or a flux in the sense of §49, because $\mathbf{f} \cdot \mathbf{n}$ does not fulfill the symmetry property of a proper Riemannian. In fact, the non-flux $\mathbf{f} \cdot \mathbf{n}$ is counted with opposite sign for two cells sharing some edge.

In BoSSS, this operator could be realized e.g. by a combination of ⁷ or ⁸ and ⁹.

§94: Corollary : For the symbols of §92 and §93, one gets the equality:

$$\mathcal{M}_{\text{div}} = (-\mathcal{M}_{\nabla})^T$$

Proof: The statement is quite obvious if the so-called *primal flux formulation* of the DG method, e.g. used in (Arnold et al. 2002, Shahbazi et al. 2007) is used.

The NSE2b - application offers two different discretizations of the Laplace operator for the Poisson equation, Eq. 5.11. The first is the Central-Difference discretization of §95, which is the exact composition of numerical divergence and gradient. The second option is an Interior penalty discretization, see §96.

§95: Corollary - Central difference discretization of the Laplace operator:

The operator

$$DG_l(\mathfrak{K}) \ni \underline{\Psi} \mapsto \mathbf{CD}_{\Delta}(\underline{\Psi}) \in DG_l(\mathfrak{K})$$

defined by

$$\widetilde{\mathbf{CD}}_{\Delta}(\tilde{\Psi}) = \mathcal{M}_{\text{div}} \cdot \mathcal{M}_{\nabla} \cdot \tilde{\Psi} + \mathcal{M}_{\text{div}} \cdot c + d(\mathbf{u}_{\text{Bnd}})$$

with symbols and definitions of §92 and §93, is a consistent discretization of the Laplace operator $\Psi \mapsto \Delta\Psi$ and is consistent with the boundary conditions

$$\left\{ \begin{array}{ll} (\nabla\Psi \cdot \mathbf{n})|_{\Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}} \cup \Gamma_{\text{Out}}} = \mathbf{u}_{\text{Bnd}} \cdot \mathbf{n} & \text{on } \Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}} \cup \Gamma_{\text{Out}} \\ \Psi|_{\Gamma_{\text{POut}}} = p_{\text{POut}} & \text{on } \Gamma_{\text{POut}} \end{array} \right. .$$

Furthermore, the matrix of the operator, $\mathcal{M}_{\text{div}} \cdot \mathcal{M}_{\nabla}$, is symmetric and coercive, i.e. $\|\mathcal{M}_{\text{div}} \cdot \mathcal{M}_{\nabla} \cdot x\| \rightarrow \infty$ if $\|x\| \rightarrow \infty$.

⁷BoSSS.Foundation.IDualValueFlux

⁸BoSSS.Foundation.ILinearDualValueFlux

⁹BoSSS.Foundation.IDualValueFlux

§96: Definition - Interior Penalty discretization of the Poisson equation operator: The interior penalty discretization of $\Psi \mapsto \Delta\Psi$, with boundary conditions

$$\begin{cases} (\nabla\Psi \cdot \mathbf{n})|_{\Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}} \cup \Gamma_{\text{Out}}} = 0 & \text{on } \Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}} \cup \Gamma_{\text{Out}} \\ \Psi|_{\Gamma_{\text{POut}}} = p_{\text{POut}} & \text{on } \Gamma_{\text{POut}} \end{cases}$$

is notated for the remaining chapter as

$$DG_l(\mathfrak{K}) \ni \underline{\Psi} \mapsto \mathbf{IP}_\Delta(\underline{\Psi}) \in DG_l(\mathfrak{K})$$

and defined like in §52. In the notation of §52, $g_{\text{Diri}} := p_{\text{Out}}$, $g_{\text{Neu}} = 0$ and $\Gamma_{\text{Diri}} = \Gamma_{\text{POut}}$ while $\Gamma_{\text{Neu}} = \Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}} \cup \Gamma_{\text{Out}}$.

Matrix and affine part of \mathbf{IP}_Δ are denoted as

$$\widetilde{\mathbf{IP}}_\Delta(\tilde{\Psi}) = \mathcal{M}_{ip} \cdot \tilde{\Psi} + c_{ip}.$$

As already discussed in §84, the Neumann boundary condition for the Poisson problem has to fulfil a compatibility condition (see Eq. 5.6 resp. 5.7) if it is used on the whole domain boundary, without any Dirichlet region. The easiest – and most precise – way of implementing it can be derived from the discrete analogy of the compatibility condition:

§97: Corollary : Let be $A \in \mathbb{R}^{n \times n}$ with rank $n - 1$; Further given is $v \in \mathbb{R}^n$ so that $v^T \cdot A = 0$, i.e. v is an Eigenvector for A^T for Eigenvalue 0. Then the system

$$A \cdot x = r$$

has a solution if $v^T \cdot r = 0$.

Proof: see (Pozrikidis 2001).

Up to now, all ingredients – discretization of pressure gradient, velocity divergence and Poisson operator – have been laid out, so the projection step can be formulated.

§98: Corollary - Discrete Projection: The discretized version of the Poisson equation Eq. 5.11 for projection is

$$\mathcal{M}_\Delta \tilde{\Psi} + c' = \frac{1}{\Delta t} \cdot \widetilde{\mathbf{div}}(\tilde{\mathbf{u}}^*).$$

As already mentioned, the NSE2b - application offers two options for the discretization of the Laplace operator, \mathcal{M}_Δ :

- Option CDP (Central Difference Poisson) With definitions from §93 and §92, set

$$\mathcal{M}_\Delta := \mathcal{M}_{\text{div}} \cdot \mathcal{M}_\nabla \quad \text{and} \quad c' = \mathcal{M}_{\text{div}} \cdot c.$$

- Option IPP (Interior Penalty Poisson): With definitions from §96, set

$$\mathcal{M}_\Delta := \mathcal{M}_{ip} \quad \text{and} \quad c' = c_{ip}.$$

Then, the “final” velocity field $\mathbf{u}^1 \in DG_p(\mathcal{K})^D$ is given by

$$\tilde{\mathbf{u}}^1 = \tilde{\mathbf{u}}^* - \Delta \cdot \tilde{\mathbf{G}}_\nabla(\tilde{\Psi}). \quad (5.12)$$

For the CDP option, the divergence-free condition is exactly fulfilled, i.e. $\mathbf{div}(\tilde{\mathbf{u}}^1) = 0$; for the IPP option, this only holds approximately¹⁰.

It remains to specify $\mathbf{u}_{\text{Bnd}} : \Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}} \cup \Gamma_{\text{Out}} \rightarrow \mathbb{R}$ from §93, which the affine part of \mathbf{div} depends on. Since on inlet and wall the Dirichlet boundary condition for the velocity, $\mathbf{u}_{\text{Inl,Wall}} : \Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}} \rightarrow \mathbb{R}$ is given it is natural to set $(\mathbf{u}_{\text{Bnd}})|_{\Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}}} := \mathbf{u}_{\text{Inl,Wall}}$. For the remaining part $(\mathbf{u}_{\text{Bnd}})|_{\Gamma_{\text{Out}}}$ two cases must be considered:

- *Case 1:* $\Gamma_{\text{Out}} \neq \{\}$, i.e. the boundary $\partial\Omega$ contains some region in which a Dirichlet boundary condition for the pressure is given. In this case, the compatibility condition applies neither on discrete (§97) nor on continuous level (Eq. 5.6). \mathcal{M}_Δ has full rank, and one choice for $\mathbf{u}_{\text{Bnd}}|_{\Gamma_{\text{Out}}}$ is to set

$$\mathbf{u}_{\text{Bnd}}|_{\Gamma_{\text{Out}}} = (\mathbf{u}^* - \Delta t \cdot g_{\text{Neu}} \cdot \mathbf{n})|_{\Gamma_{\text{Out}}}.$$

- *Case 2:* $\Gamma_{\text{POut}} = \{\}$: here, the rank of \mathcal{M}_Δ is $(\dim(DG_l(\mathcal{K})) - 1)$, i.e. there is one linear dependent row in \mathcal{M}_Δ . One sets

$$\mathbf{u}_{\text{Bnd}}|_{\Gamma_{\text{Out}}} = (\mathbf{u}^* - \Delta t \cdot g_{\text{Neu}} \cdot \mathbf{n} + \alpha \cdot \mathbf{1}_{\Gamma_{\text{Out}}})|_{\Gamma_{\text{Out}}}.$$

with a constant $\alpha \in \mathbb{R}$ so that the discrete compatibility condition is fulfilled. The computation of α is given within the proof, Eq. 5.13.

Partial Proof/Discussion: $\mathbf{div}(\mathbf{u}^1) = 0$ (*) for the CDP option: this follows directly from inserting Eq. 5.12 into (*):

$$\begin{aligned} (*) \quad & \mathcal{M}_{\text{div}} \cdot \tilde{\mathbf{u}}^1 + d(\mathbf{u}_{\text{Bnd}}) = 0 \\ (P) \quad & \tilde{\mathbf{u}}^1 = \tilde{\mathbf{u}}^* - \Delta t \cdot (\mathcal{M}_\nabla \tilde{\Psi} + c) \end{aligned}$$

¹⁰ The idea of “approximating” $\mathcal{M}_{\text{div}} \cdot \mathcal{M}_\nabla$ by \mathcal{M}_{ip} is already presented in (Shahbazi et al. 2007). What this means in terms of accuracy is, to our knowledge, unknown.

Inserting (P) into (*) yields:

$$\mathcal{M}_{\text{div}} \cdot \mathcal{M}_{\nabla} \tilde{\Psi} + \mathcal{M}_{\text{div}} \cdot c = \frac{1}{\Delta t} \cdot (\mathcal{M}_{\text{div}} \cdot \tilde{\mathbf{u}}^* + d(\mathbf{u}_{\text{Bnd}}))$$

Choice of parameter α : For the right-hand-side of Eq. 5.12, one gets the decomposition:

$$\begin{aligned} \widetilde{\text{div}}(\tilde{\mathbf{u}}^*) &= \underbrace{\mathcal{M}_{\text{div}} \cdot \tilde{\mathbf{u}}^* + d(\mathbf{u}_{\text{Bnd}})}_{=:r} \\ &= \underbrace{\mathcal{M}_{\text{div}} \cdot \tilde{\mathbf{u}}^* + d(\mathbf{u}_{\text{Inl,Wall}} + (\mathbf{u}^* - \Delta t \cdot \mathbf{p})|_{\Gamma_{\text{Out}}})}_{=:r_1} + \alpha \cdot \underbrace{d(\mathbf{1}_{\Gamma_{\text{Out}}})}_{=:r_2} \end{aligned}$$

Define v as the projection of a constant pressure of 1: $v = Proj_l(1)$. If \mathcal{M}_{Δ} is a consistent discretization of the operator $\Psi \mapsto \Delta \Psi$ with homogeneous Neumann boundary condition, it must hold that $\mathcal{M}_{\Delta} \cdot v = 0$. Because of the symmetric structure of \mathcal{M}_{Δ} , also $v^T \cdot \mathcal{M}_{\Delta} = 0$. Recalling from §97 that the pressure equation $\mathcal{M}_{\Delta} \cdot \tilde{\Psi} = r$ (Note that in case 2 $\Gamma_{\text{POut}} = \{\}$ and therefore $c' = 0$.) has a solution if $v^T \cdot r = 0$, one gets

$$\alpha = \frac{-v^T \cdot r_1}{v^T \cdot r_2}. \quad (5.13)$$

(End of partial Proof.)

It remains to explain “what has happened” to the inhomogeneous Neumann boundary condition $\nabla \Psi \cdot \mathbf{n} = g_{\text{Neu}}$ on the Neumann region $\partial\Omega \setminus \Gamma_{\text{POut}}$, since its value g_{Neu} is, at least not explicitly, present in §98. The answer is that contribution of the velocity jump at the boundary, between the intermediate velocity \mathbf{u}^* and the velocity boundary condition \mathbf{u}_{Bnd} to the pressure equation (Eq. 98) is the same as the contribution of the inhomogeneous that the contribution of the Neumann boundary condition for Ψ would be:

§99: Remark - Relation between Neumann boundary condition and intermediate velocity jump: Recall that the domain of the Neumann boundary condition is $\Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}} \cup \Gamma_{\text{Out}}$, and that, “after the viscous+transport integrator”,

$$\begin{aligned} \mathbf{u}^*|_{\Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}}} &= \mathbf{u}_{\text{Inl,Wall}} + \Delta t \cdot \mathbf{p}_{\text{Neu}} && \text{on inlet and wall and} \\ \mathbf{u}^*|_{\Gamma_{\text{Out}}} &\approx \mathbf{u}^0 + \Delta t \cdot \mathbf{p}_{\text{Neu}} && \text{on the outflow-domain.} \end{aligned} \quad (5.14)$$

W.l.o.g. within this paragraph there is no Dirichlet region for the pressure, i.e. $\Gamma_{\text{Polt}} = \{\}$. For $\mathbf{div}(\mathbf{u}^*)$, consider the following unique decomposition:

$$\begin{aligned} \left[\widetilde{\mathbf{div}}(\tilde{\mathbf{u}}^*) \right]_{\text{map}(0,j,m)} = & \underbrace{\left(\int_{K_j} \% d\mathbf{x} - \int_{\partial K_j \setminus \partial\Omega} \% dS \right)}_{=:d_{\text{internal}}} \\ & + \underbrace{\left(- \int_{\partial K_j \setminus \partial\Omega} (\mathbf{u}^* - \mathbf{u}_{\text{Bnd}}) \cdot \mathbf{n} \cdot \phi_{j,m} dS \right)}_{=:d_{\text{boundary}}} \end{aligned}$$

Together with Eqs. 5.14, this yields

$$d_{\text{boundary}} \approx - \int_{\partial K_j \setminus \partial\Omega} g_{\text{Neu}} \cdot \phi_{j,m} dS$$

(the “ \approx ” - sign is even exact on $\Gamma_{\text{Inl}} \cup \Gamma_{\text{Wall}}$). This is exactly the same contribution as an inhomogeneous Neumann boundary condition $\nabla\Psi \cdot \mathbf{n} = g_{\text{Neu}}$ would have in discrete Poisson equation (Eq. 98) for both, the CDP and the IP discretization of Δ , see §95 and §52.

5.4 Temporal discretization of the predictor

Within the previous section, all ingredients for the algorithm presented in paragraph 86, have been laid out, except the temporal discretization of the predictor, in order to get from an initial value \mathbf{u}^0 to an intermediate solution \mathbf{u}^* , i.e.

$$\mathbf{u}^0 \mapsto \mathbf{u}^*.$$

This will be the issue of the current section. Three variants will be presented. Two of them are splitting schemes, where the evolution of the nonlinear terms are performed by an explicit Runge-Kutta scheme, while the linear viscous terms are treated by an implicit scheme, e.g. Crank-Nicolson. The third scheme is a combination of Addams-Bashforth for nonlinear and Crank-Nicolson for viscous terms.

§100: Corollary - Lie splitting for the predictor: A comprehensive overview about splitting methods can be found in (McLachlan & Quispel

2002). In DG - coordinates, the Lie-splitting approximation for the predictor (Eq. 5.9) is given by

$$\tilde{\mathbf{u}}^0 \xrightarrow{\mathcal{E}\left\{\frac{\Delta t}{2}, \tilde{\mathbf{N}}_{\mathbf{u}|_{\text{Inl}}}\right\}} \tilde{\mathbf{u}}^{\text{int}} \xrightarrow{\mathcal{E}\left\{\Delta t, \left[\tilde{u}_d \mapsto \mathcal{M}_{\frac{-1}{\text{Re}\Delta}} \cdot \tilde{u}_d + b_d(t)\right]_{d=1, \dots, D}\right\}} \tilde{\mathbf{u}}^*.$$

Note that the time-dependent velocity boundary condition $\mathbf{u}_{\text{modDir}}$ (see Eq. 5.10) is applied in the viscous operator. During the time-integration of the nonlinear terms, we keep the velocity on the inlet constant. We admit that this may be controversial.

As the time accuracy of the overall Lie splitting is of first order, it is sufficient to use an implicit Euler scheme for the time-integration of the viscous part:

$$\frac{1}{\Delta t} (\tilde{u}_d^* - \tilde{u}_d^{\text{int}}) + \mathcal{M}_{\frac{-1}{\text{Re}\Delta}} \cdot \tilde{u}_d^* + b_d(\Delta t) = 0.$$

For time integration of the nonlinear parts (which are, in terms of computational cost, much cheaper than the viscous part) we use a 3-stage, 3rd order TVD diminishing Runge-Kutta scheme, as presented in (Gottlieb & Shu 1998).

§101: Corollary - Strang splitting for the predictor: Analogous to paragraph 100, the Strang splitting could be defined as:

$$\tilde{\mathbf{u}}^0 \xrightarrow{\mathcal{E}\left\{\frac{\Delta t}{2}, \tilde{\mathbf{N}}_{\mathbf{u}|_{\text{Inl}}(0, -)}\right\}} \tilde{\mathbf{u}}^{\text{int1}} \xrightarrow{\mathcal{E}\left\{\Delta t, \left[\tilde{u}_d \mapsto \mathcal{M}_{\frac{-1}{\text{Re}\Delta}} \cdot \tilde{u}_d + b_d(t)\right]_{d=1, \dots, D}\right\}} \tilde{\mathbf{u}}^{\text{int2}} \xrightarrow{\mathcal{E}\left\{\frac{\Delta t}{2}, \tilde{\mathbf{N}}_{\mathbf{u}_{\text{modDir}}(\Delta t, -)}\right\}} \tilde{\mathbf{u}}^*.$$

For the Strang splitting, the same discussion about boundary conditions applies.

As the scheme is expected to be of order 2, a second-order integrator for the middle part is required, we suggest to use a Crank-Nicolson scheme:

$$\frac{1}{\Delta t} (\tilde{u}_d^{\text{int2}} - \tilde{u}_d^{\text{int1}}) + \frac{1}{2} \cdot \mathcal{M}_{\frac{-1}{\text{Re}\Delta}} \cdot (\tilde{u}_d^{\text{int2}} + \tilde{u}_d^{\text{int1}}) + \frac{1}{2} \cdot (b_d(\Delta t) + b_d(0)) = 0.$$

Again, for time integration of the nonlinear parts (which are, in terms of computational cost, much cheaper than the viscous part) we use a 3-stage, 3rd order TVD diminishing Runge-Kutta scheme, as presented in (Gottlieb & Shu 1998).

§102: Corollary - 2-stage Adams-Bashforth/Crank Nicolson – scheme (AB2/CrnkNic) for the predictor: Providing one previous timestep, \mathbf{u}^{-1} at $t = -\Delta t$, (Kim & Moin 1985) suggest the scheme:

$$\begin{aligned} \frac{1}{\Delta t}(\tilde{\mathbf{u}}^* - \tilde{\mathbf{u}}^0) + \frac{3}{2}\tilde{\mathbf{N}}_{\mathbf{u}_{\text{Inl}}(0,-)}(\tilde{\mathbf{u}}^0) - \frac{1}{2}\tilde{\mathbf{N}}_{\mathbf{u}_{\text{Inl}}(-\Delta t,-)}(\tilde{\mathbf{u}}^{-1}) \\ + \frac{1}{2} \left[\mathcal{M}_{\frac{-1}{\text{Re}}\Delta} \cdot \delta_{i,j} \right]_{\substack{i=1,\dots,D \\ j=1,\dots,D}} \cdot (\tilde{\mathbf{u}}^* - \tilde{\mathbf{u}}^0) + \frac{1}{2} [b_d(\Delta t) - b_d(0)]_{d=1,\dots,D} = 0. \end{aligned}$$

5.5 Numerical Results

We give four examples for the incompressible Navier-Stokes solver; At first, the differences between the Central-Difference and the Interior-Penalty projection are demonstrated in a laminar 2D channel (§103). Second, a 2D turbulent channel at $\text{Re}_\tau = 180$ is shown (§104). Third, an instationary flow around a 2D-cube at $\text{Re} = 100$. This could be verified against experimental and other numerical data. Finally, a 3D wall-mounted-cube with approximately 65'000 tetrahedral cells is presented (§106). This is actually the largest incompressible simulation that has been done with BoSSS until now. However, the Reynolds number of 350 is still a magnitude lower than with typical numerical or experimental setups that can be found in literature.

Within this work, no verification of the Navier-Stokes solver is given. In fact, these tasks which include comparison with exact solutions and measurement of convergence order, have been performed by co-workers N. Emamy, B. Klein, R. Ashoori, R. Mousavi and Z. Nirobash.

§103: Example and Discussion - Comparison of CDP and IPP discretization: The same easy example as in §85 is considered. Given is a non-periodic channel flow, $\Omega = (0, 10) \times (-1, 1)$, with inlet $\Gamma_{\text{Inl}} = \{0\} \times (-1, 1)$ where $\mathbf{u}_{\text{Inl}} = (1 - y^2, 0)^T$, walls $\Gamma_{\text{Wall}} = (0, 10) \times \{-1, 1\}$ where $\mathbf{u}_{\text{Wall}} = 0$ and a pressure outlet $\Gamma_{\text{POut}} = \{10\} \times (-1, 1)$ where $p_{\text{POut}} = 0$. The Reynolds number is set to 10, giving the steady-state – solution $\mathbf{u} = (1 - y^2, 0)^T$.and $p = 2 - \frac{-2 \cdot x}{10}$

The domain is discretized using 128×32 Cartesian equidistant cells, the DG polynomial degree for both, \mathbf{u} and p is 2. As an initial value

$$\mathbf{u}^0 = \left((1 - y^2) \cdot \left(1 - \frac{x}{10} \right), 0 \right)^T$$

is given, which obviously violates the continuity equation, i.e. $\text{div}(\mathbf{u}^0) \neq 0$.

Detailed comparison for Poisson solver runtime, and the evolution of $\|\mathbf{div}(\mathbf{u}^n)\|_2$ for the timesteps $n = 1, \dots, 10$ are given in tables 5.1 and 5.2. For the Poisson solver, the Conjugate-Gradient with Local-Block-Elimination – preconditioning (see §111) was used. The initial guess for the Poisson solver, for all iterations had been set to 0. The residual threshold was set to 1.0^{-6} , using the monkey – implementation (see chapter 4) with GPU acceleration, using a Nvidia GTX 470 GPU.

Overall, the following statements can be made:

- For the CDP, $\|\mathbf{div}(\mathbf{u}^n)\|_2 \approx 0$ and only determined by numerical round-off – errors. For the IPP, $\|\mathbf{div}(\mathbf{u}^n)\|_2$ is several magnitudes higher. The effect on the accuracy of the overall simulation is, to our knowledge, unknown.
- In all investigated cases, some of which are not discussed within this manuscript, the CDP required a significantly lower number of (pre-conditioned) CG iterations to reach a given residual threshold.
- It is obvious that the CDP discretization has a larger stencil of dependence than the IPP discretization, and therefore, a higher number of non-zeros within the matrix and requires more memory. Therefore, the sparse matrix-vector product, which is the kernel of the CG algorithm, is computationally more expensive. Overall, CDP is still faster, because of the lower number of iterations.
- The initialization cost for the CDP, to compute the sparse matrix-matrix product $\mathcal{M}_{\text{div}} \cdot \mathcal{M}_{\nabla}$ is much higher than for the IPP:

§104: Example - A 2D channel with $\text{Re}_\tau = 180$: As a representative for the periodic channel $\mathbb{R}/2\pi\mathbb{Z} \times (-1, 1)$ the domain $\Omega = (-\pi, \pi) \times (-1, 1)$ was chosen, discretized by 256×144 Cartesian cells. In x - direction, the grid spacing is equidistant and in y - direction an exponential is used. The distribution of nodes in y - direction is given in figure 5.1. As a initial value, to trigger the initialisation of turbulence, the sum of a parabolic pro-

n	IPP		CDP	
	$t_{\text{wall}}, [\text{sec}]$	No.Of.Iter.	$t_{\text{wall}}, [\text{sec}]$	No.Of.Iter.
1	0.36	1082	0.295	701
2	0.345	1036	0.25	581
3	0.34	1003	0.28	660
4	0.345	1029	0.295	662
5	0.345	1011	0.275	656
6	0.335	997	0.25	562
7	0.315	893	0.235	545
8	0.32	977	0.255	563
9	0.325	981	0.25	559
10	0.33	952	0.25	533

Table 5.1: Comparison of Poisson solver performance for Interior Penalty (IPP) versus Central Difference Poisson discretization. Wall clock time t_{Wall} and number of iterations for preconditioned CG sparse solver are tabulated.

n	IPP	CDP
	$\ \mathbf{div}(\mathbf{u}^n)\ _2$	$\ \mathbf{div}(\mathbf{u}^n)\ _2$
1	$5.679 \cdot 10^{-3}$	$7.464 \cdot 10^{-7}$
2	$8.317 \cdot 10^{-3}$	$7.355 \cdot 10^{-7}$
3	$4.753 \cdot 10^{-3}$	$7.316 \cdot 10^{-7}$
4	$2.113 \cdot 10^{-3}$	$7.289 \cdot 10^{-7}$
5	$1.457 \cdot 10^{-3}$	$7.396 \cdot 10^{-7}$
6	$1.169 \cdot 10^{-3}$	$7.447 \cdot 10^{-7}$
7	$6.528 \cdot 10^{-4}$	$7.487 \cdot 10^{-7}$
8	$4.2 \cdot 10^{-4}$	$7.595 \cdot 10^{-7}$
9	$4.558 \cdot 10^{-4}$	$7.662 \cdot 10^{-7}$
10	$4.526 \cdot 10^{-4}$	$7.794 \cdot 10^{-7}$

Table 5.2: Comparison of Interior Penalty (IPP) versus Central Difference Poisson discretization, regarding their effect on velocity divergence.

file (i.e. the laminar solution) and three vortices around the points $(-2, 0)$, $(0, 0)$ and $(2, 0)$ was chosen:

$$\begin{aligned}
u(x, y) &= 90 \cdot (1 - y^2 + H(0.9 - \sqrt{x^2 + y^2}) \cdot (-y) \\
&\quad + H(0.9 - \sqrt{(x - 2)^2 + y^2}) \cdot (-y) \\
&\quad + H(0.9 - \sqrt{(x + 2)^2 + y^2}) \cdot (-y)) \\
v(x, y) &= 90 \cdot (H(0.9 - \sqrt{x^2 + y^2}) \cdot x \\
&\quad + H(0.9 - \sqrt{(x - 2)^2 + y^2}) \cdot (x - 2) \\
&\quad + H(0.9 - \sqrt{(x + 2)^2 + y^2}) \cdot (x + 2)).
\end{aligned}$$

Further properties of the numerical simulation were:

- For time discretization, the 2-stage Adams-Bashforth/Crank Nicolson – scheme presented in §102 with a timestep $\Delta t = 2.5 \cdot 10^{-6}$ was used.
- DG polynomial degree is 3 for u , v and p ;
- Simulation was performed on GPU cluster FUCHS (for details on the system, see §73) using 4 MPI processes and 4 GPU's.
- The Poisson equation was discretized by the central difference discretization, see §95.
- A local-block-elimination, see §111 was used for preconditioning of the Poisson equation.

Snapshots of vorticity at different timesteps can be found in figures 5.2 and 5.3.

The following example (§105) was already used by (Shahbazi et al. 2007) to verify their DG implementation.

§105: Example - A 2D flow around a box at $Re = 100$: The 2D domain $\Omega = (-16, 25) \times (-22, 22) \setminus (-1/2, 1/2)^2$ is discretized by an unstructured triangular mesh with 2'092 cells¹¹, as seen in figure 5.4. The boundary is partitioned into

$$\Gamma_{\text{Inl}} = -16 \times (-22, 22) \quad \text{and} \quad \Gamma_{\text{POut}} = \partial\Omega \setminus \Gamma_{\text{Inl}}.$$

As boundary conditions,

$$\begin{aligned}
\mathbf{u}_{\text{Wall}} &= (1, 0)^T && \text{on } \Gamma_{\text{Inl}} \text{ and} \\
p_{\text{POut}} &= 0 && \text{on } \Gamma_{\text{POut}}
\end{aligned}$$

¹¹ The grid was created by Mrs. Zarah Niroobakhsh using the Pointwise V16 software package.

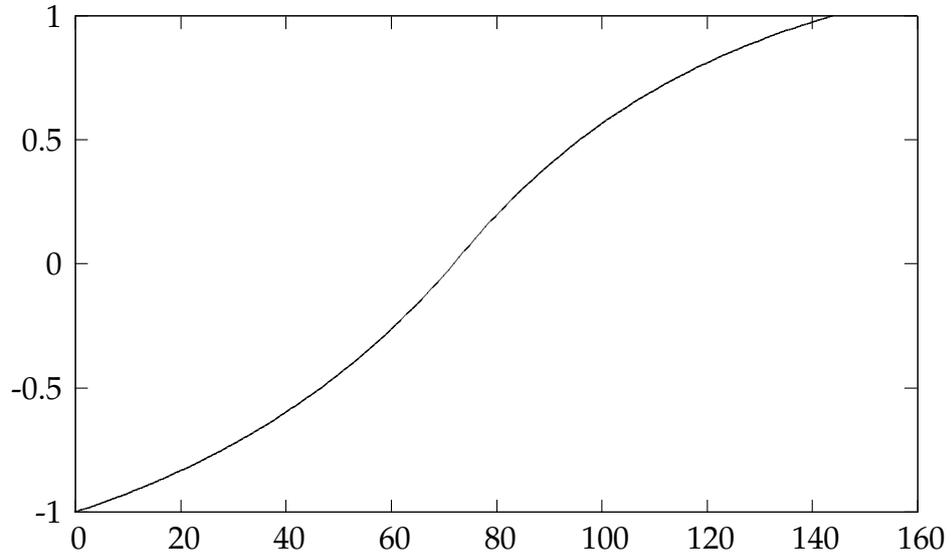


Figure 5.1: Distribution of grid-nodes in y -direction for the $\text{Re}_\tau = 180$ - 2D channel

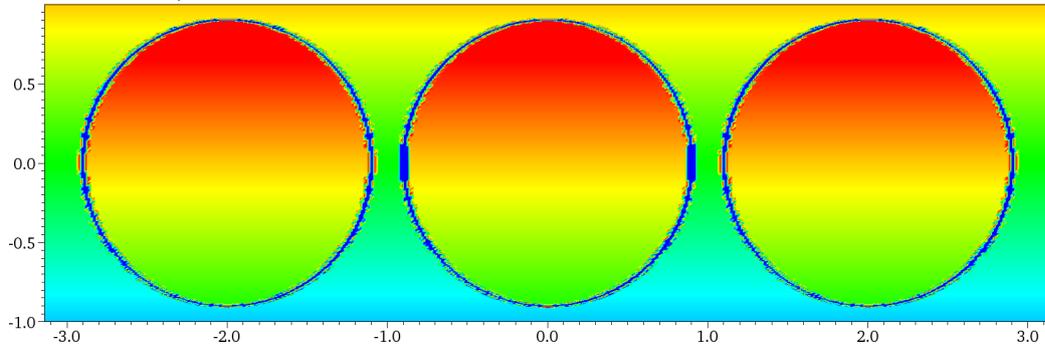
are given. Further, $\text{Re} = 100$. As an initial condition, to trigger vortex shading, the constant velocity of 1 in x - direction is overlapped by an artificial vortex centred at $(-3, 0.25)$ with radius $\sqrt{3}$:

$$\begin{aligned}
 u(x, y) &= H(3 - (x + 3)^2 - (y - 0.25)^2) \cdot \sqrt{(x + 3)^2 + (y - 0.25)^2} \\
 &\quad \cdot (-(y - 0.25)) + H((x + 3)^2 + (y - 0.25)^2 - 3) \\
 v(x, y) &= H(3 - (x + 3)^2 - (y - 0.25)^2) \cdot \sqrt{(x + 3)^2 + (y - 0.25)^2} \cdot (x + 3)
 \end{aligned}$$

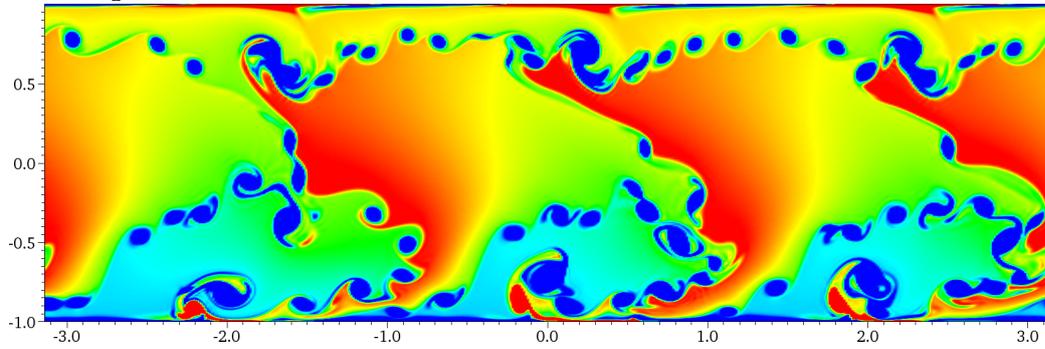
Contour plots of velocity magnitude can be found in figures 5.5 and 5.6. Further properties of the numerical simulation were:

- For time discretization, the 2-stage Adams-Bashforth/Crank Nicolson - scheme presented in §102 with a timestep $\Delta t = 0.5 \cdot 10^{-3}$ was used.
- DG polynomial degree is 4 for u , v and p ;
- The Poisson equation was discretized by the central difference discretization, see §95.
- The direct solver PARDISO (see (Schenk et al. 2000, Schenk 2002, Schenk 2004, Schenk & Gärtner 2006)) was used to solve the Poisson equation.

initial value, $t = 0$:



timestep 8'000, $t = 0.02$:



timestep 16'000, $t = 0.04$:

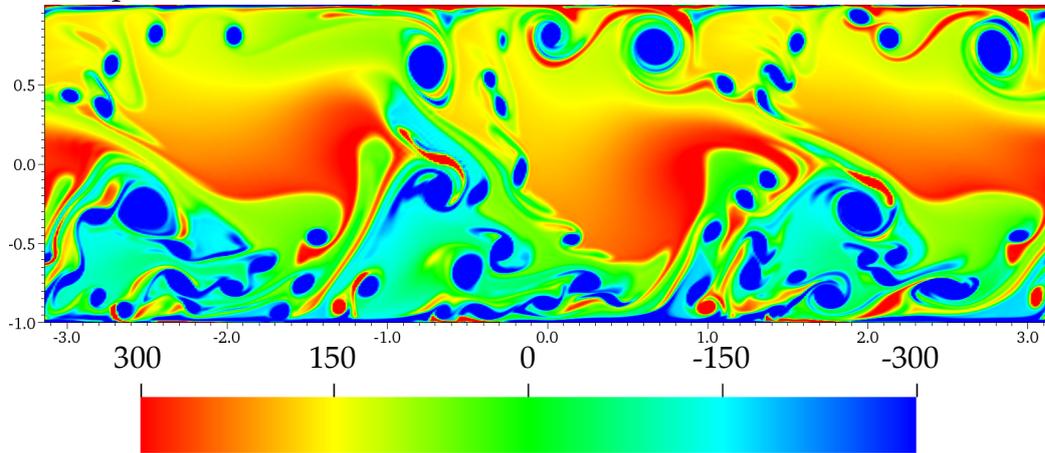
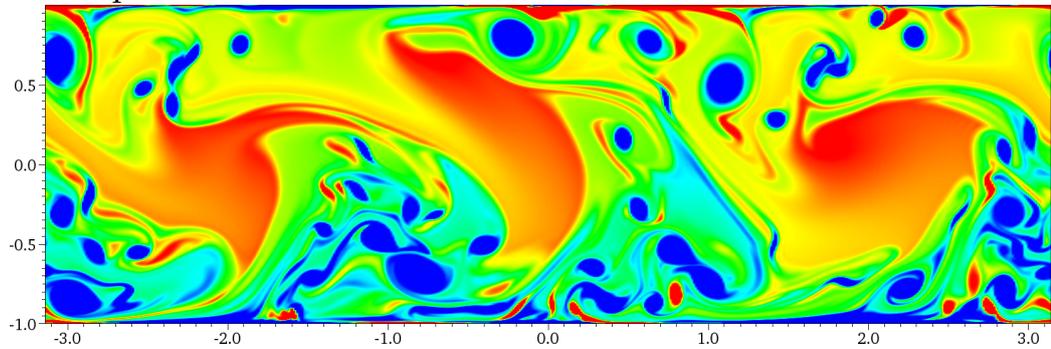
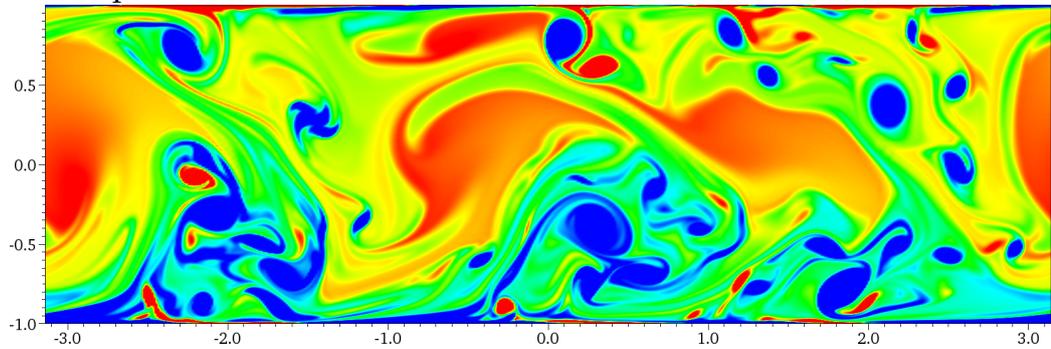


Figure 5.2: Vorticity ($\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$) contour plot for the $Re_\tau = 180$ - 2D channel at different time steps

timestep 24'000, $t = 0.06$:



timestep 32'000, $t = 0.08$:



timestep 40'000, $t = 0.1$:

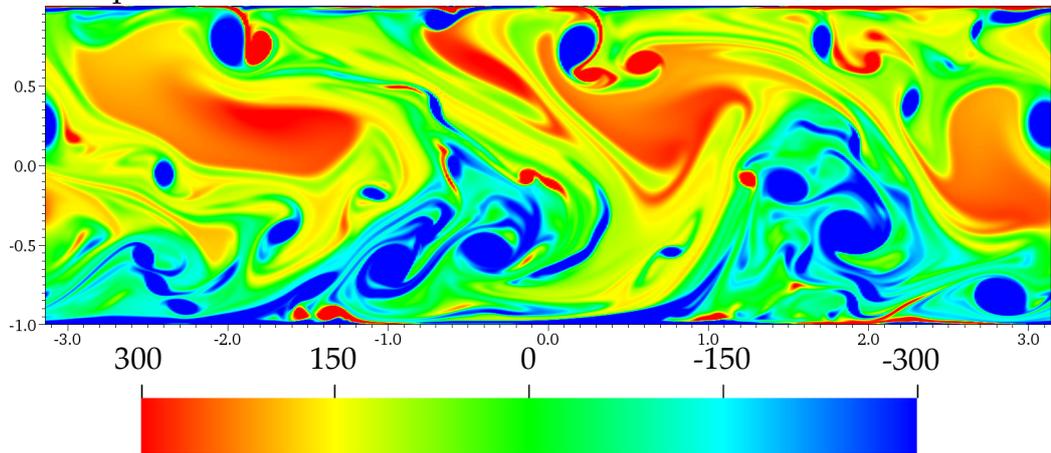


Figure 5.3: Vorticity ($\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$) contour plot for the $Re_\tau = 180$ - 2D channel at different time steps, part 2

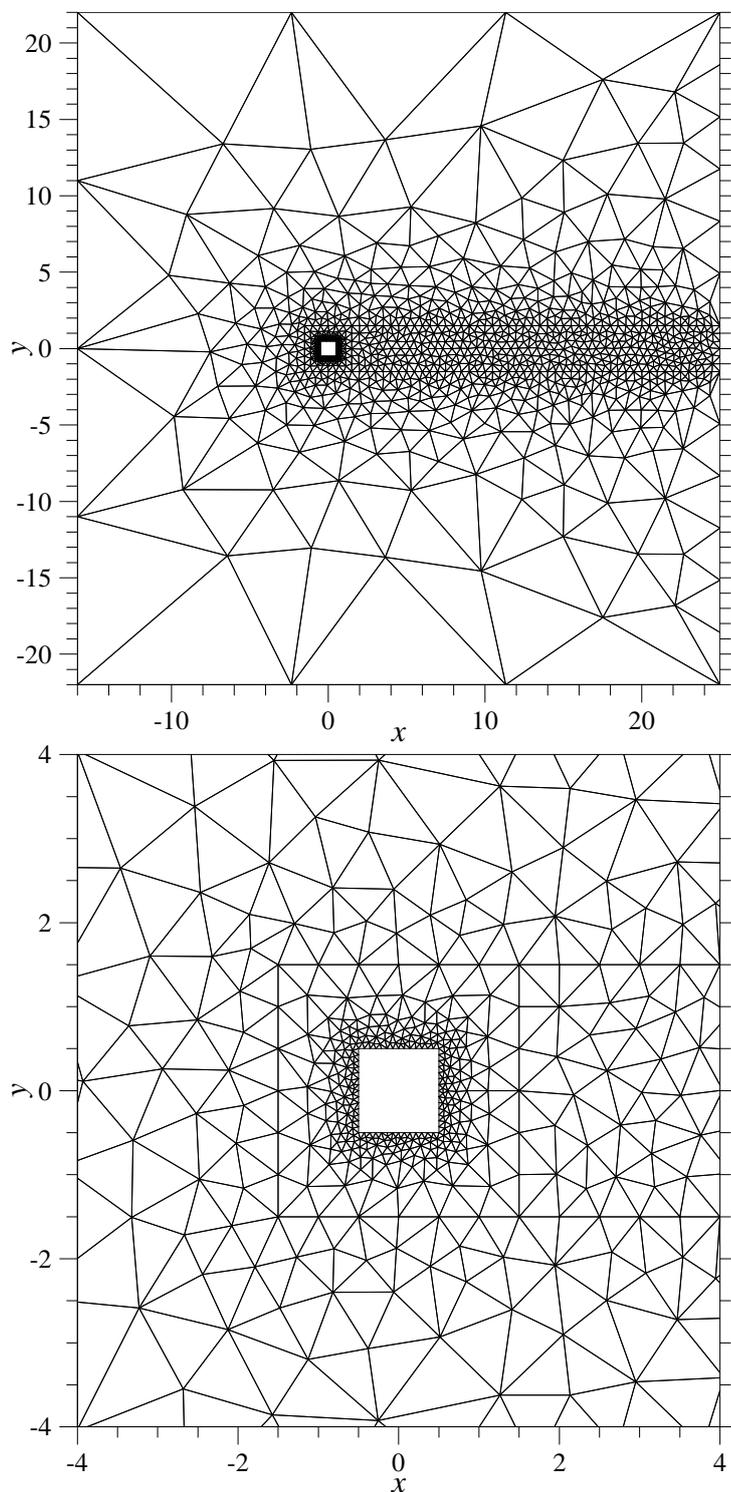


Figure 5.4: Grid used in example §105; Full domain (top) and zoom around the box $(-0.5, 0.5)^2$.

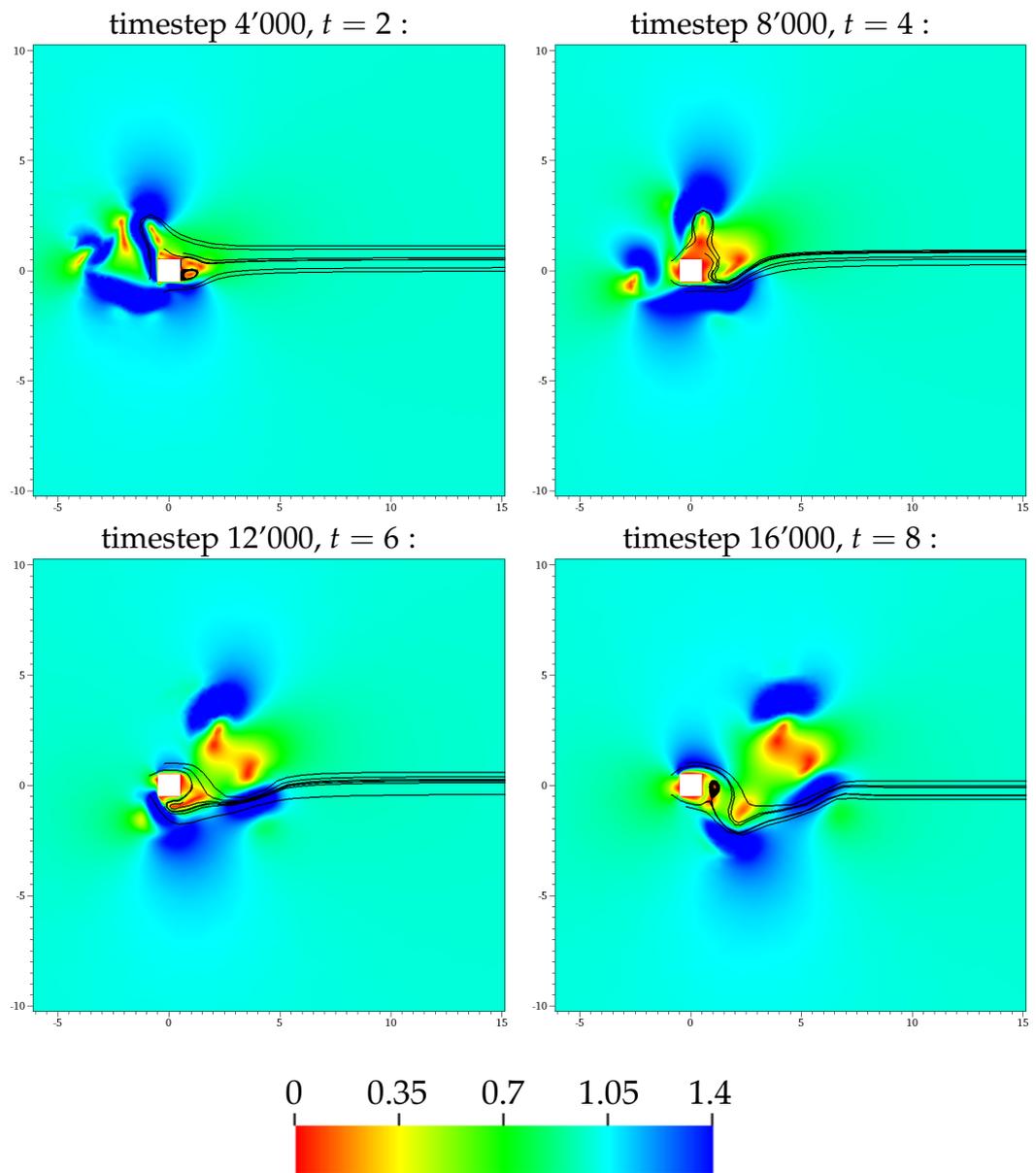


Figure 5.5: Velocity magnitude ($\|\mathbf{u}\|_2$) contour plot and streamlines for flow around an obstacle at $Re = 100$.

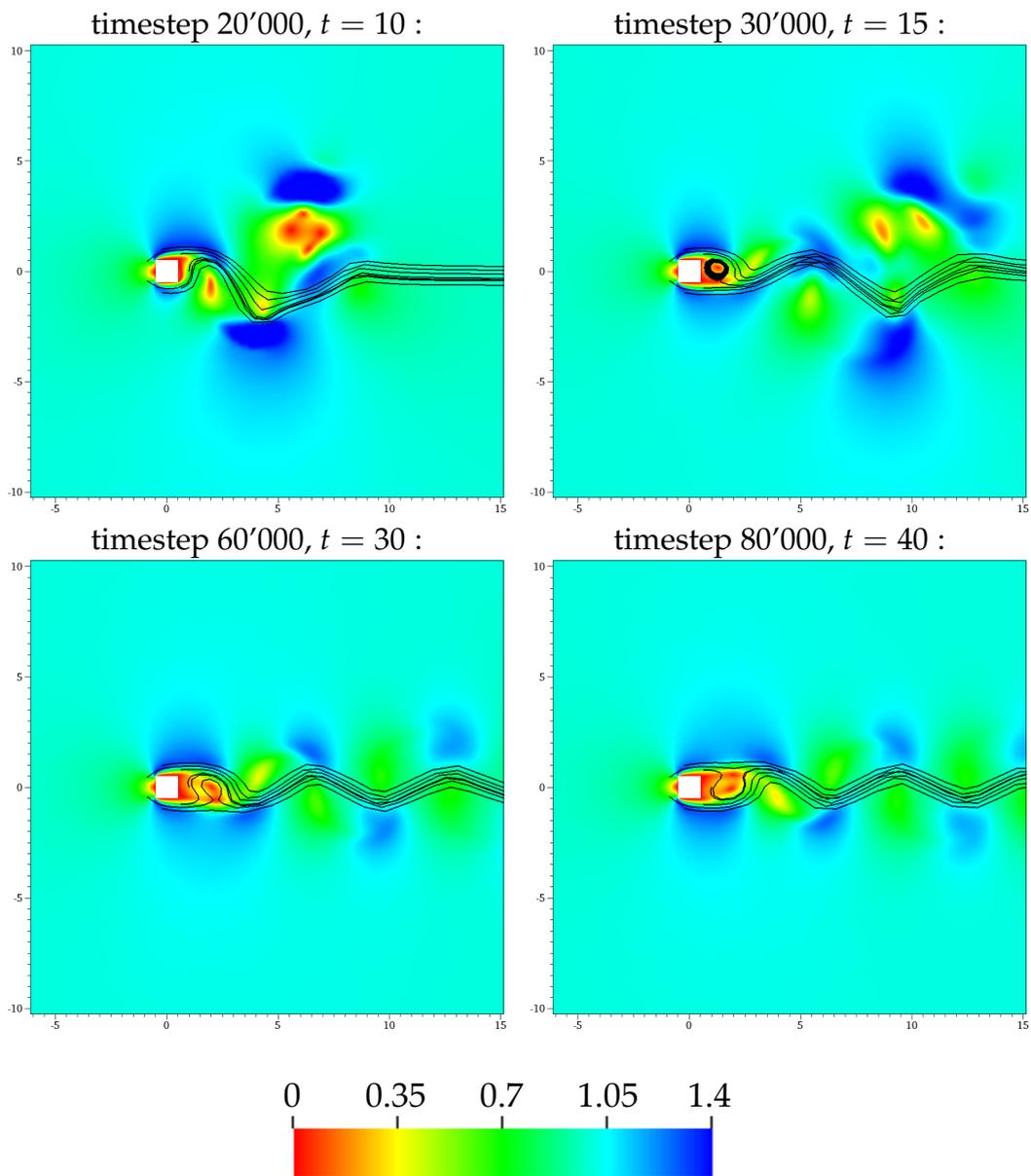


Figure 5.6: Velocity magnitude ($\|\mathbf{u}\|_2$) contour plot and streamlines for flow around an obstacle at $Re = 100$.

§106: Example - A 3D flow around a wall-mounted cube at $Re = 350$:
The 3D domain

$$\Omega = (-3.5, 11.5) \times (-3.2, 3.2) \times (0, 3) \setminus (-1/2, 1/2) \times (-1/2, 1/2) \times (0, 1)$$

is discretized by an unstructured tetrahedral mesh with 63'438 cells¹². The diameter of the smallest cell, located on the surface of the cube is approximately 0.037, while the diameter of the largest cell, located on Γ_{POut} is approximately 1.44.

The Reynolds number is set to 350.

The boundary is partitioned into

$$\begin{aligned}\Gamma_{Inl} &= -3.5 \times (-3.2, 3.2) \times (0, 3), \\ \Gamma_{POut} &= \{(x, y, z); z = 3 \text{ or } y = \pm 3.2 \text{ or } x = 11.5\} \text{ and} \\ \Gamma_{Wall} &= \partial\Omega \setminus (\Gamma_{Inl} \cup \Gamma_{POut}).\end{aligned}$$

So, Γ_{Wall} corresponds to what one usually would consider as “floor and cube surface”, Γ_{POut} corresponds to “ceiling, left, right and back face” and Γ_{Inl} may be called “the front” of the computational domain. As boundary conditions,

$$\begin{aligned}\mathbf{u}_{Wall} &= (0, 0)^T && \text{on } \Gamma_{Wall} \text{ and} \\ p_{POut} &= 0 && \text{on } \Gamma_{POut}\end{aligned}$$

are given.

As an initial and inlet condition, a Blasius profile in z - direction is used. The thickness of this particular Blasius boundary layer is given by

$$z_{Blasius}(x) = 5 \cdot \sqrt{\frac{x + 4}{Re}},$$

i.e. its thickness grows in x -direction. This implies that the thickness of the Blasius profile is approximately 0.189 at the inlet, i.e. at $x = -3.5$ and 0.5 at the beginning of the cube, i.e. at $x = -0.5$. The velocity profile of the initial and inlet condition, at three different x - positions, is shown in figure 5.7.

Velocity streamline plots can be found in figure 5.8; Further properties of the numerical simulation were:

- For time discretization, the 2-stage Adams-Bashforth/Crank Nicolson – scheme presented in §102 with a timestep $\Delta t = 0.5 \cdot 10^{-3}$ was used.

¹² The grid was created using the Pointwise V16 software package.

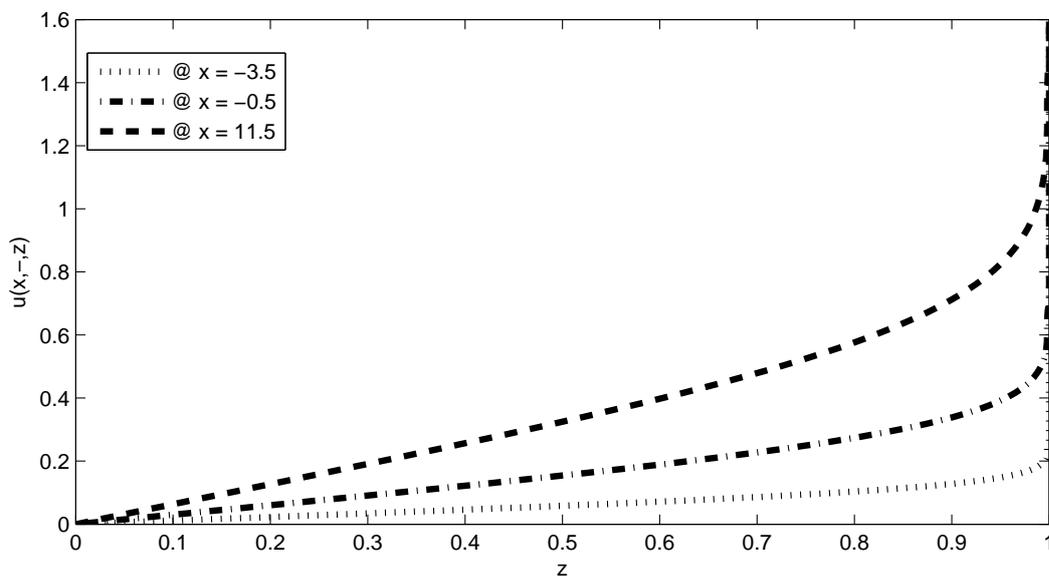


Figure 5.7: Initial and inlet value of the u - velocity component, that is constant in y - direction but depends on the distance from the wall, i.e. the z - axis as well as the distance from the inlet, i.e. the x - axis. Three different profiles are shown, one at the inlet ($x = -3.5$), one at the front edge, in streamwise direction, of the cube ($x = -0.5$) and one at the outlet ($x = 11.5$).

- DG polynomial degree is 2 for u, v, w and p ;
- The Poisson equation was discretized by the interior penalty discretization, see §96.
- Simulation was performed on GPU cluster FUCHS (for details on the system, see §73) using 6 MPI processes and 6 GPU's.
- All 6 MPI processes used approximately 15.5 Gigabyte memory in total.
- A local-block-elimination, see §111, was used for preconditioning of the Poisson equation.

5.6 Basic preconditioning techniques

For the unsteady, incompressible Navier-Stokes solver presented in this chapter, linear systems are usually assembled once, at start-up, and are solved hundred times or even more often, with different right-hand-side. Thus, it pays off to invest computational time into explicit precondition-

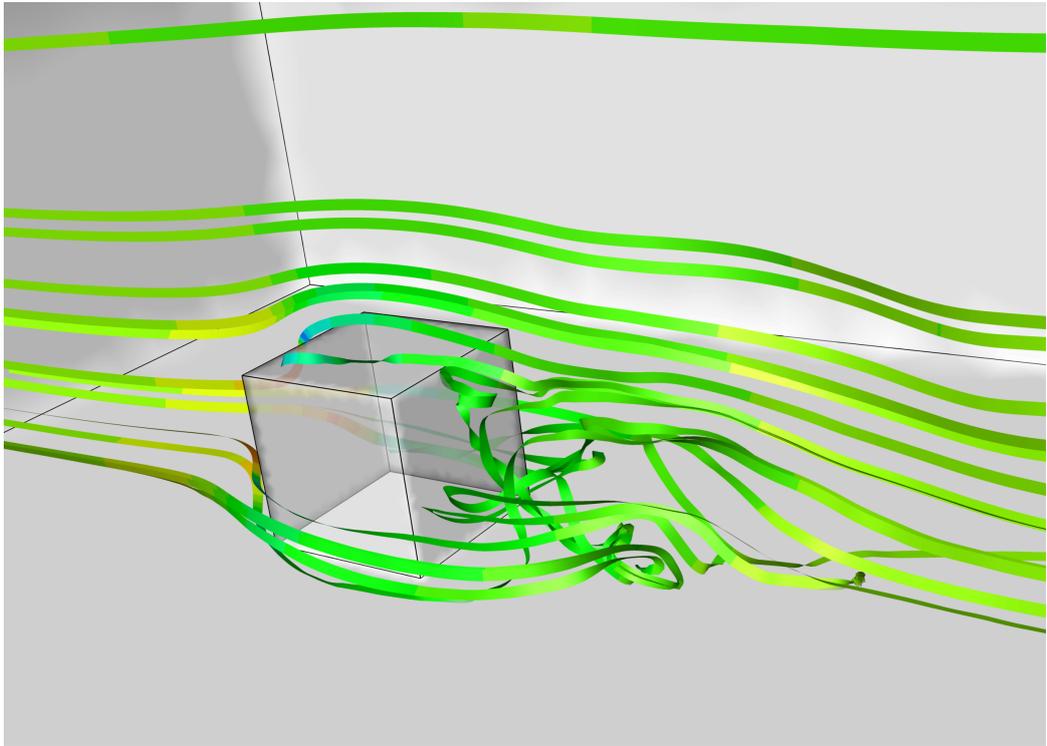


Figure 5.8: Streamlines around a wall mounted cube, at timestep 5'040, i.e. $t = 2.5$.

ers. These transform the matrix itself and have a quite high computational cost at start-up, but produce almost no additional cost during an iterative solver run. If these systems would be solved only once, their application would not be beneficial, but since the systems are solved many times, they are.

§107: Notation : Within this section let $\mathcal{M} \in \mathbb{R}^{J \cdot N \times J \cdot N}$ be some invertible matrix, usually the matrix \mathcal{M}_Δ from §98. The goal is to solve the system

$$\mathcal{M} \cdot x = r. \quad (5.15)$$

Within this section, only preconditioning for symmetric matrices is considered. There it is desirable that the preconditioned system is itself symmetric. For a more detailed review, refer to (Meister 2004).

§108: Definition : For an invertible matrix $P \in \mathbb{R}^{J \cdot N \times J \cdot N}$, the preconditioned equivalent of Eq. 5.15 is given by

$$\mathcal{M}^P \cdot x^P = r^P \quad (5.16)$$

with

$$\mathcal{M}^P = \alpha \cdot P \cdot \mathcal{M} \cdot P^T,$$

with $\alpha \in \mathbb{R}_{\neq 0}$ and

$$r^P = \alpha \cdot P \cdot r.$$

§109: Remark : From §108 it immediately follows that

$$x = P^T \cdot x^P.$$

§110: Definition - Condition Number: $Cond_a(\mathcal{M}) := \|\mathcal{M}\|_a \cdot \|\mathcal{M}^{-1}\|_a.$

Usually, the convergence rate of iterative sparse solvers depends on the condition number. Therefore searched is for P with $Cond_a(\mathcal{M}^P) < Cond_a(\mathcal{M})$. The local-block-elimination, presented in subsequence, is a solution that helps with the dramatic grow of the condition number for the Interior-Penalty – discretization of the Poisson equation on stretched grids (see tables 5.3 and 5.4).

§111: Definition - local-block-elimination: Given is the notation of §108. Let the regular matrix $\mathcal{M} \in \mathbb{R}^{J \cdot N \times J \cdot N}$ be either positive or negative definite

and the constant $\alpha \in \{-1, 1\}$. P is called a local-block-elimination preconditioner for \mathcal{M} , if the $N \times N$ - diagonal blocks of $\mathcal{M}^P = \alpha \cdot P \cdot \mathcal{M} \cdot P^T$ are the identity matrix, i.e.

$$\forall 0 \leq j < J : \forall j \cdot N < n, m < (j + 1 \cdot N) : [\mathcal{M}]_{nm} = \delta_{nm}.$$

§112: Remark - on local-block-elimination: The local-block-elimination - matrix P , together with constant α is uniquely defined by the characteristic property given in the previous paragraph (§111); it can be found by the use of Gaussian elimination. Remember that a multiplication of \mathcal{M} from left by a regular matrix corresponds to some row operation, while a multiplication from the right corresponds to a column operation. Because \mathcal{M} is multiplied from the left by P and from the right by P^T , this induces that – in contrast to the normal Gaussian elimination – each operation is carried out symmetrically, e.g. if row i is multiplied by some number $c \in \mathbb{R}$, then column i is multiplied with the same number c as well.

§113: Example - Performance gain by Local-Block-Elimination: The investigated problem is an Interior Penalty discretization (as used by the pressure projection, see §52, see also (Shahbazi 2005)) of the 2D Poisson problem

$$\begin{cases} \Delta T = 1 & \text{in } \Omega := (0, 10) \times (-1, 1) \\ T = 0 & \text{on } \Gamma_{\text{Diri}} := \{\mathbf{x} \in \partial\Omega; x = 0\} \\ \nabla T \cdot \mathbf{n} = 0 & \text{on } \partial\Omega \setminus \Gamma_{\text{Diri}} \end{cases}$$

with the exact solution

$$T(x, y) = 50 - \frac{x^2}{2}.$$

The domain is discretized by a Cartesian grid with 128×32 cells, equidistant in x - direction. In y - direction the grid is stretched, with each cell closer to $+y$ being α times “wider” than the next neighbouring cell in $-y$ - direction. Stretching factors $\alpha \in \{1, 1.05, 1.1, 1.15\}$ were investigated. The DG polynomial degree of the T was 2.

The effect of Local-Block-Elimination on matrix condition number and Conjugate-Gradient solver runtime are shown in tables 5.3 and 5.4. Especially for “highly” stretched grids, Local-Block-Elimination is absolutely necessary to get acceptable solver performance. On equidistant grids, the performance gain is still in the region of 30 %.

Stretch factor. α	no precondition. matrix condition no.	Local-Block-Elimination matrix condition no.
1	$8.2 \cdot 10^6$	$3 \cdot 10^6$
1.05	$3.4 \cdot 10^7$	$3.4 \cdot 10^6$
1.1	$1.9 \cdot 10^8$	$5 \cdot 10^6$
1.15	$1.3 \cdot 10^9$	$9.2 \cdot 10^6$

Table 5.3: Effect of Local-block-Elimination on Conjugate Gradient – solver performance, in dependence of grid stretching, computed with MATLAB function `condest(...)`, which gives an approximation to the condition number based on the 1-norm. We agree that the condition number of the problem may be considered as “horrible”, but with Local-Block-elimination it is less “horrible”. This result shows, that for the Poisson equation of the Projection scheme for the incompressible NSE, work on preconditioning still needs to be done.

Stretch factor. α	no precondition.		Local-Block-Elimination	
	t_{wall} , [sec]	No.Of.Iter.	t_{wall} , [sec]	No.Of.Iter.
1	2.3	1'570	1.6	1'094
1.05	5.6	3'589	2.8	1'012
1.1	14.9	9'819	4.5	2'035
1.15	51.7	34'890	6.4	3'332

Table 5.4: Effect of Local-block-Elimination on Conjugate Gradient – solver (CG) performance, in dependence of grid stretching. Results were achieved using the Hypr - implementation of CG, using an absolute residual threshold of 10^{-7} .

6 The extended DG method and the Level-Set - framework

Our motivation for singular partial differential equations is the treatment of immiscible two-phase flows. If physical properties, such as density and viscosity, jump at the interface between the two fluids, the pressure and velocity field will contain at least kinks. For material interfaces, additionally – singular – models for surface tension will induce a jump in the pressure field. Solutions for non-material interfaces contain jumps in velocity and pressure field, even without any surface tension models.

These equations – together with the jump conditions – are called singular, because the jump conditions can be translated to singular distributions and vice-versa. The relation between jumps and Delta - distributions are given in §151 and 152.

The numerical treatment of singular equations requires two things. At first, the position of singularity – i.e. the position of the “front”, or the “interface” – must be described, which is done by the well-known Level Set method. Second, the singularities – or discontinuities – must be represented in the numerics. This is achieved by the modulation of the DG basis by the characteristic function of the subdomains, or phases in §126.

This chapter is organized as follows: within the first two sections (6.1 and 6.2) the Extended DG (XDG) method is introduced and formally constructed.

After defining the XDG method, the question is how the additional degrees-of-freedom, which were introduced, could be determined. This is demonstrated for linear, scalar steady-state and transient examples. For such problems, to fix the additional degrees-of-freedom, we propose what we call the “patched-decomposition” - approach.

This approach is introduced for the Poisson equation with jump, which stands as a prototype for a steady-state problem (section 6.3).

Subsequently, this ideas are extended to transient equations. In order to explain the effect of the interface movement onto the $\frac{\partial}{\partial t}$ - operator, some results from Distribution theory are summed up (section 6.4). Given that,

the patched-decomposition approach is extended to a singular Heat equation (section 6.5).

6.1 The Level-Set – framework

§114: Notation - Problem Setting: For the usual computational domain $\Omega \in \mathbb{R}^D$ the time-dependent disjoint decomposition

$$\mathbb{R}_{\geq 0} \ni t \mapsto (\mathfrak{A}(t), \mathfrak{I}(t), \mathfrak{B}(t)) \in \text{Pot}(\Omega)^3 \quad \text{with} \quad \mathfrak{A}(t) \cup \mathfrak{I}(t) \cup \mathfrak{B}(t) \quad \forall t$$

is given. The sets \mathfrak{A} and \mathfrak{B}^1 are open, not necessarily connected, but each decomposes into a finite disjoint decomposition of simply connected parts, with D – dimensional volume of each part greater than 0. The set $\mathfrak{I}(t) = \overline{\mathfrak{A}(t)} \cap \overline{\mathfrak{B}(t)}$ is called the interface between \mathfrak{A} and \mathfrak{B} and is a $(D - 1)$ – dimensional, at least C^0 – manifold. The decomposition is continuous in time, i.e.

$$\left\| \mathbf{1}_{\mathfrak{A}(t')} - \mathbf{1}_{\mathfrak{A}(t)} \right\|_2 \leq L \cdot |t' - t|$$

with a Lipschitz constant $L > 0$ for all $t \geq 0$. If required, other types of continuity may be defined in analogue manner. This setup is illustrated in figure 6.1. The normal field on \mathfrak{I} , $\mathbf{n}_{\mathfrak{I}} : \mathfrak{I} \rightarrow \mathcal{S}^D$ is oriented so that “it points from \mathfrak{A} to \mathfrak{B} ”, i.e. $\mathbf{n}_{\mathfrak{I}} = \mathbf{n}_{\mathfrak{A}}$ for the outer normal field $\mathbf{n}_{\mathfrak{A}}$ of \mathfrak{A} .

For transient problems, it is more convenient to notate the time-dependent phases in a space-time fashion. Therefore we introduce the space-time – versions of the phases $\mathfrak{A}(t)$ and $\mathfrak{B}(t)$ and of the interface $\mathfrak{I}(t)$:

§115: Definition - time-space formulation: We define

$$\begin{aligned} \Omega^\times &:= \mathbb{R}_{>0} \times \Omega, \\ \mathfrak{A}^\times &:= \{(t, \mathbf{x}) \in \Omega^\times; \mathbf{x} \in \mathfrak{A}(t)\}, \\ \mathfrak{I}^\times &:= \{(t, \mathbf{x}) \in \Omega^\times; \mathbf{x} \in \mathfrak{I}(t)\}, \\ \mathfrak{B}^\times &:= \{(t, \mathbf{x}) \in \Omega^\times; \mathbf{x} \in \mathfrak{B}(t)\}. \end{aligned}$$

§116: Definition - Level-Set - function: A function²

$$\mathbb{R}_{\geq 0} \times \Omega \ni (t, \mathbf{x}) \mapsto \varphi(t, \mathbf{x}) \in \mathbb{R}$$

¹ $\mathfrak{A}(t)$ may be denoted just as \mathfrak{A} , if t can be considered fixed, e.g. in a steady-state problem.

² In everyday speaking, the Level-Set – function is often referred to as the “Level-Set”.

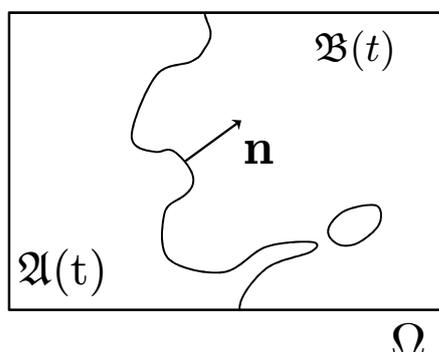


Figure 6.1: Illustration of multiphase problem setting; phase $\mathfrak{A}(t)$, phase $\mathfrak{B}(t)$ and the interface $\mathfrak{I}(t)$. $\Omega = \mathfrak{A}(t) \cup \mathfrak{I}(t) \cup \mathfrak{B}(t)$ for all $t \geq 0$.

in $\mathcal{C}^0(\mathbb{R}_{\geq 0} \times \Omega)$ is called *Level-Set - function* for the setting from §114, if

$$\begin{cases} \mathfrak{A}(t) = \{\mathbf{x}; \varphi(t, \mathbf{x}) < 0\} \\ \mathfrak{I}(t) = \{\mathbf{x}; \varphi(t, \mathbf{x}) = 0\} \\ \mathfrak{B}(t) = \{\mathbf{x}; \varphi(t, \mathbf{x}) > 0\} \end{cases}$$

and φ fulfils the signed-distance property, i.e.

$$|\varphi(t, \mathbf{x})| = \min\{|\mathbf{x} - \mathbf{y}|; \mathbf{y} \in \mathfrak{I}(t)\}.$$

Consequently,

$$\text{sign}(\varphi(t, \mathbf{x})) = \begin{cases} -1 & \mathbf{x} \in \mathfrak{A}(t) \\ 1 & \mathbf{x} \in \mathfrak{B}(t) \end{cases}.$$

The theory of Level-Set - functions can be considered general knowledge, see e.g. (Sethian 2001, Sethian & Smereka 2003) or the textbooks (Sethian 1996, Sethian 1999, Osher & Fedkiw 2002).

The evolution of $\mathfrak{I}(t)$ in time could be described by defining the “speed” of \mathfrak{I} in its normal direction. As long as the normal field $\mathbf{n}_{\mathfrak{I}}$ could be defined everywhere, the situation is quite easy. In general, this is however not the case.

At first, we give a characteristic property of the surface normal speed s for the evolution of $\mathfrak{I}(t)$.

§117: Definition - level set evolution: Let all $\mathcal{I}(\tau)$ be \mathcal{C}^1 - manifolds for $\tau \in [t^0, t]$. If, for all $\mathbf{x}_0 \in \mathcal{I}(t^0)$ the points $\mathbf{x}(t)$, defined by the integral equation

$$\mathbf{x}(t) = \int_{\tau=t^0}^t \mathbf{x}_0 + s(\tau, \mathbf{x}(\tau)) \cdot \mathbf{n}(\tau, \mathbf{x}(\tau)) d\tau$$

are in $\mathcal{I}(t)$, the function $s \in \mathcal{C}^0(\mathcal{I}^\times) \cap L^\infty(\mathcal{I}^\times)$ is called the *surface speed in normal direction* or just *surface speed* of \mathcal{I} , resp. \mathcal{I}^\times .

§118: Remark - Level-Set – equation: For a signed-distance Level-Set function φ , on \mathcal{I}^\times the equation

$$\frac{\partial}{\partial t} \varphi + s = 0. \quad (6.1)$$

holds. Trivially, $\mathbf{n}_{\mathcal{I}} = \nabla \varphi$.

Rationale: $0 = \frac{d\varphi}{dt} = \frac{\partial}{\partial t} \varphi + \underbrace{\nabla \varphi \cdot \mathbf{n}_{\mathcal{I}}}_{=1} \cdot s$. Note that because of the signed-distance property of φ , $|\nabla \varphi| = 1$ and $\mathbf{n}_{\mathcal{I}} = \nabla \varphi$. End of rationale.

Typically, the surface speed s is defined by a physical model which is only reasonable on the front \mathcal{I}^\times itself. A good example for this may be the modelling of two-phase flows with non-material interface like e.g. evaporation. It is obvious that a model for evaporation speed can only be given at the interface itself, and not within the gas or liquid phase. So, at first the Level-Set evolution equation (Eq. 6.1) is only defined on the interface itself.

For a numerical method that computes the evolution of the Level Set function φ , it is of course necessary to know an evolution speed on the whole domain, or at least in an not-too-small vicinity around the interface, since the numerical algorithm “cannot work on just the interface itself”. Therefore it is numerically necessary to define a continuation of s from the interface \mathcal{I} to the whole computational domain Ω , in a way that does not influence the propagation of the interface itself. One solution, proposed by the Extension-Velocity method (Sethian & Smereka 2003) is to continue s constant along lines perpendicular to $\mathcal{I}(t)$. This will preserve the signed-distance property. Formally, the Extension-Velocity method could be written down by the use of the closest-point - operator.

§119: Definition - closest point: For some point in $\mathbf{x} \in \Omega$ we define the closest point (if it exists) within some set $X \subset \Omega$:

$$Cp(\mathbf{x}, X) := \operatorname{minarg}_{\mathbf{y} \in X} |\mathbf{x} - \mathbf{y}|$$

§120: Remark - closest point: (i) It is plain to see that the line between any point \mathbf{x} in the domain and its closest point on the interface is perpendicular to the interface, i.e.

$$(\mathbf{x} - Cp(\mathbf{x}, \mathcal{I})) \cdot \mathbf{v} = 0$$

for all vectors \mathbf{v} in the tangential space of \mathcal{I} at the point $Cp(\mathbf{x}, \mathcal{I})$.

(ii) For a signed-distance Level-Set φ , almost everywhere the closest point on $\mathcal{I}(t)$ is given by

$$Cp(\mathbf{x}, \mathcal{I}(t)) = \mathbf{x} - \varphi(\mathbf{x}) \cdot (\nabla \varphi)(\mathbf{x}).$$

As already mentioned, the goal is to find a continuation of the surface speed s – that is only defined on the interface – to the whole domain. So, using the closest-point - operator one defines

$$s(t, \mathbf{x}) = \underbrace{s(t, Cp(\mathbf{x}, \mathcal{I}(t)))}_{\text{on } \mathcal{I}(t) \text{ therefore def. by phys. model}} \quad (6.2)$$

for points \mathbf{x} outside of \mathcal{I} .

The concept of surface speed is – up to now – only defined for smooth interfaces \mathcal{I} , and the Level-Set – equation is only defined on the interface \mathcal{I} itself. Especially the evolution of \mathcal{C}^0 - interfaces is still undefined.

§121: Corollary - evolution of \mathcal{C}^0 - interfaces: Given is a field $s \in \mathcal{C}^0(\Omega^\times) \cap L^\infty(\Omega^\times)$ and an initial Level Set function φ^0 ; assume that φ is given by the evolution equation

$$\frac{\partial}{\partial t} \varphi(t, \mathbf{x}) + s(t, Cp(\mathbf{x}, \{\mathbf{y}; \varphi(t, \mathbf{y}) = 0\})) = 0 \quad (6.3)$$

with initial value $\varphi(t = 0, -) = \varphi^0$. Then, $\varphi(t, -)$ is a signed-distance Level-Set function in the sense of §116 and the zero Level-Sets $\mathcal{I}(t)$ are \mathcal{C}^0 - manifolds for all $t > t^0$.

(without proof.)

A very convenient property of the closest-point formulation of the Level Set evolution (Eq. 6.3) is that it is able to treat singularities on the interface, i.e. regions where the interface normal cannot be defined, in a way that is reminiscent of the Huygens-Fresnel principle, see figure 6.2.

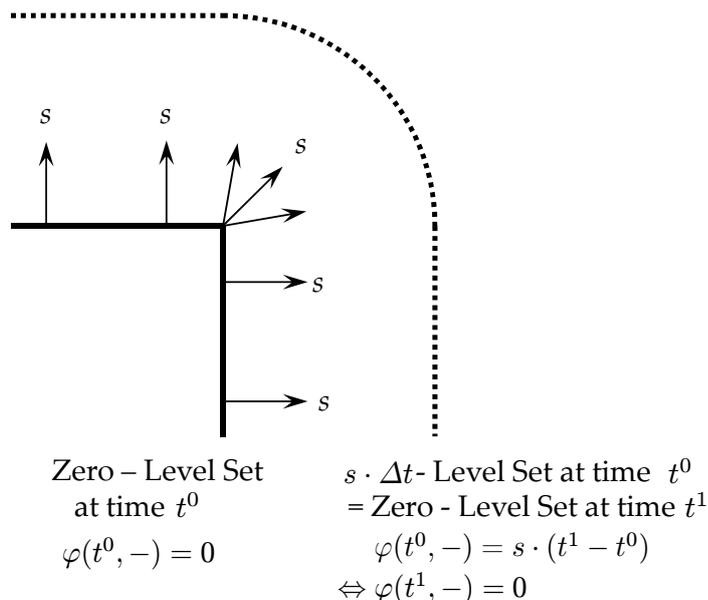


Figure 6.2: Illustration of the “Huygens-Fresnell - principle” for Level Sets: Note that for a surface with a singularity, i.e. an kink where no normal can be defined, the concept of speed-in-normal-direction makes no sense. The evolution of such non- C^0 - surfaces can be closed in satisfying manner if the above-mentioned principle is applied. Mathematically, this can be achieved by the closest-point formulation of the Level Set evolution equation.

Numerous methods for computing the evolution of the Level-Set have been developed. A numerical implementation of Eq. 6.3 is obviously very difficult, and maybe impossible to achieve by using only classical numerical techniques for PDE's. The nature of Eq. 6.3, as already discussed, is that the evolution speed s somewhere outside of \mathcal{J} is determined by the closest point on the interface. The extension-velocity method uses the so-called fast marching algorithm (Sethian 2001) to construct an s - field with this property. Theoretically, the closest-point formulation completely eliminates any need for reinitialisation. Practically this implies for the ex-

tension velocity method that the Reinitialisation procedure is only rarely required. However, it should be noted that the construction of the extended velocity in the sense of Eq. 6.2 is as expensive or complex as a complete Reinitialisation, because both problems are very closely related. It is therefore only affordable if a very fast algorithm, like fast-marching, is available for the extension of the velocity field.

Other algorithms split into Level-Set evolution and Reinitialisation (see §122): Given an initial signed-distance Level-Set function $\varphi'(t^0, -)$, the equation

$$\frac{\partial}{\partial t} \varphi' + \nabla \varphi' \cdot \frac{\nabla \varphi'}{|\nabla \varphi'|} \cdot s = 0.$$

is integrated up to some time $t^1 > t^0$. For s , some reasonable but easy-to-get extension outside of \mathcal{J} is used. It is obvious that the signed-distance property does not hold for $t > 0$, therefore the normal vector $\mathbf{n}_{\mathcal{J}}$ is computed as $\frac{\nabla \varphi'}{|\nabla \varphi'|}$. A Reinitialisation – algorithm is used to replace $\varphi'(t^1, -)$ by some level set function that fulfills the signed-distance property.

§122: Definition - Reinitialisation: The Reinitialisation is defined as a mapping

$$\mathcal{C}^0(\Omega) \ni \varphi' \mapsto \varphi \in \mathcal{C}^0(\Omega)$$

so that

- $\text{sign}(\varphi'(\mathbf{x})) = \text{sign}(\varphi(\mathbf{x}))$ for all $\mathbf{x} \in \Omega$
- The zero - Level-Sets of φ' and φ are equal
- φ fulfills the signed-distance property.

Since Level-Set algorithms are not the scope of this work we refer to the already cited textbooks for further reading. For a DG-specific implementation of the algorithm – very coarsely – discussed above, we refer to (Gross & Hesthaven 2006).

Supplement: For operations with the time-space interface it will be necessary to know its time-space normal which is only reasonable in an dimensionless setting; it could be found from the surface speed s and the space-normal $\mathbf{n}_{\mathcal{J}}$. This could be argued from basic geometric considerations.

§123: Remark - normal field on \mathcal{J}^\times : is given by

$$\mathbf{n}^\times := (-s, \mathbf{n}_{\mathcal{J}}) \cdot \frac{1}{\sqrt{1 + s^2}}$$

6.2 The XDG framework

Ideas similar to the ones presented below have been proposed by many authors in the context of Finite Element methods, summarized under the term “extended Finite Element”, or XFEM. Consequently, the method presented here is called “extended DG” or XDG.

§124: Definition - Cut, Near and Far cells: Given the definitions from §114, for a Level Set function $\mathbf{x} \mapsto \varphi(\mathbf{x})$, i.e. at a fixed timestep $t \geq 0$, the set of *cut cells* is defined as:

$$Cut(\varphi) := Near(\varphi, 0) := \{K \in \mathfrak{K}; K \cap \mathfrak{I} \neq \{\}\}.$$

The sets *near-cells* of distance e , for $e \in \mathbb{Z}_{\neq 0}$, are recursively defined as

$$Near(\varphi, e) := \left\{ K \in \mathfrak{K}; \exists K' \in Near(\varphi, e - \text{sign}(e)) : \bar{K} \cap \bar{K}' \neq \{\} \right. \\ \left. \text{and } \text{sign}(e) \cdot \varphi|_K > 0 \right\}.$$

Given a certain near-region width $n_w \in \mathbb{N}_{>0}$, one defines the positive and negative Far-region, i.e.

$$Far_+(\varphi) := \bigcup_{i=n_w+1}^{\infty} Near(\varphi, i) \quad \text{and} \quad Far_-(\varphi) := \bigcup_{i=n_w+1}^{\infty} Near(\varphi, -i),$$

respectively.

All the sets \mathfrak{I} , \mathfrak{A} and \mathfrak{B} used in the definition are fully determined by φ , see §116. The definition of cut cells with respect to the interface \mathfrak{I} , i.e. the set $Cut(\varphi)$ is obvious. The layers of near-cells around the cut cells are defined recursively. The set $Near(\varphi, 1)$ contains all cells that share at least one point with the cut cells and where additionally $\varphi > 0$. $Near(\varphi, 2)$ contains all cells that share some point with $Near(\varphi, 1)$ where $\varphi > 0$, and so on. In analogue fashion one defines $Near(\varphi, -1), Near(\varphi, -2), \dots$ for the region where the Level Set $\varphi < 0$. Since it is technically not necessary to know e.g. $Near(\varphi, 321)$, the – expensive – construction of the Near-sets is only done for a low number of layers n_w with e.g. $n_w = 2$. The Near-layers above this number are collected in the $Far_+(\varphi)$ and $Far_-(\varphi)$ sets. For the construction of the near-layers, it is important to notice the difference to the usual meaning of ‘neighbouring cell’. In the typical finite volume or DG context, two cells are considered as neighbours if they share a common edge³. For the construction of Near-cells two are considered as

³ Rem.: the term “edge” is used generally for the boundary of the element K , i.e. for the faces of K in 3D and for the edges of K in 2D.

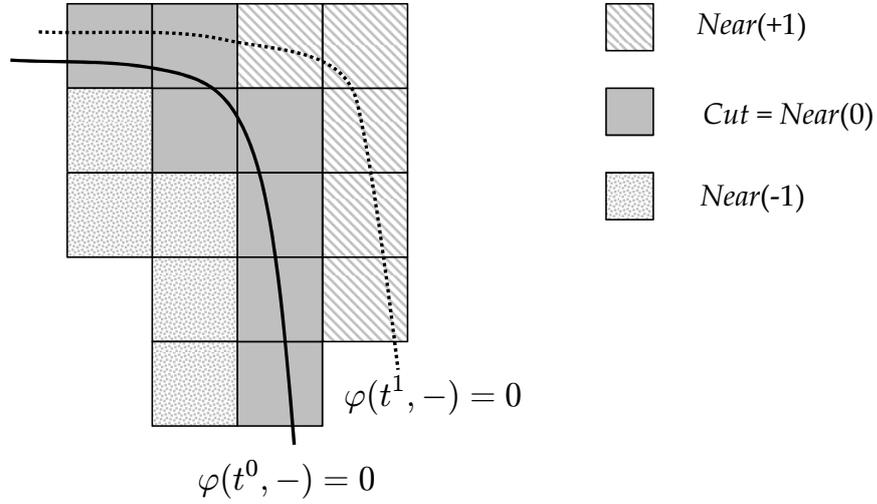


Figure 6.3: Illustration of cut cells, as well as $Near(\varphi(t^0, -), +1)$ and $Near(\varphi(t^0, -), -1)$ sets. Note that some cell is in the near-region not only if it shares an edge with some cut cell, but also if it shares at least one point with some cut cell, e.g. like illustrated by the cell on the upper right corner of the sketch.

neighbours if they share even a single point, i.e. $K \in \mathfrak{K}$ is a neighbour of $L \in \mathfrak{K}$ if $\bar{K} \cap \bar{L} \neq \{\}$, like illustrated in figure 6.3.

§125: Remark - a CFL- condition for the Level-Set: In time dependent problems, the level-set $\mathcal{I}(t)$ moves across cells. Consider some kind of time-stepping procedure, that evolves an initial Level Set field φ^0 at t^0 to a new solution φ^1 at time t^1 ; By some CFL-criterion, one usually requires that the interface \mathcal{I} has not moved by more than one cell, i.e.

$$Cut(\varphi^1) \subset \left(Near(\varphi^0, -1) \cup Cut(\varphi^0) \cup Near(\varphi^0, 1) \right). \quad (6.4)$$

With the usual definition of neighbourhood, it is clear that Eq 6.4 would not hold, e.g. as is illustrated in figure 6.3. The Level-Set is expected to cross corners of a cut cell, and therefore can enter cells that do not share an edge with the cut cell of the old timestep t^0 .

But instead, if two cells are already considered as neighbours when they share at least one point, these problems can be omitted and Eq. 6.4 gives a well-defined and usable CFL criterion for the movement of $\mathcal{I}(t)$.

§126: Definition - XDG space: The extended DG space, for al Level Set field $\mathbf{x} \mapsto \varphi(\mathbf{x})$ is defined as

$$XDG_p(\varphi, \mathfrak{K}) := DG_p(\mathfrak{K}) \oplus \bigoplus_{\substack{j \in J_C \\ n \in N_S}} \mathbf{1}_{\mathfrak{A}(t)} \cdot \phi_{j,n} \cdot \mathbb{R}$$

where the set $N_S \subset \{0, \dots, N_p - 1\}$ is the set of separate degrees-of-freedom (DOF) for the cut-cells and the indices of all cut cells $J_C = \{j; K_j \in \text{Cut}(\varphi)\}$. The integer N_p is the number of DOF in an uncut cell, i.e. $N_p = \dim(DG_p(\{K_{\text{Ref}}\}))$.

§127: Definition and Remark - a basis for the extended DG space: A basis for $XDG_p(\varphi, \mathfrak{K})$ is given by

$$\begin{aligned} \Phi^X := & \left\{ \phi_{j,n}; K_j \notin \text{Cut}(\varphi), 0 \leq n < N_p \right\} && \text{un-cutted cells} \\ \cup & \left\{ \phi_{j,n} \cdot \mathbf{1}_{\mathfrak{A}(t)}, \phi_{j,n} \cdot \mathbf{1}_{\mathfrak{B}(t)}; K_j \in \text{Cut}(\varphi), n \in N_S \right\} && \text{separate DOF} \\ \cup & \left\{ \phi_{j,n}; K_j \in \text{Cut}(t), 0 \leq n < N_p, n \notin N_S \right\}, && \text{common DOF} \end{aligned}$$

where $N_S \subset \{0, \dots, N_p - 1\}$ denotes the indices of the degrees-of-freedom for species \mathfrak{A} and \mathfrak{B} . Here, N_p denotes the number of elements in the polynomial basis of degree p .

§128: Remark - on common and separate DOF's in the extended DG space: The set N_S , i.e. the indices of the separate DOF's of $XDG_p(\varphi, \mathfrak{K})$ in one cell may be chosen problem - dependent:

- the most general choice is $N_S := \{0, \dots, N_p - 1\}$.
- to enforce continuity of higher derivatives: continuity of $\partial^\alpha (u|_K)$, for $\underline{u} \in XDG_p(\varphi, \mathfrak{K})$ and the cut cell $K \in \text{Cut}(\varphi)$, for a multiindex $\alpha = (\alpha_1, \dots, \alpha_D)$ with $|\alpha| \geq p'$, can be achieved by setting $N_S = \{0, \dots, N'\}$ with $N' = \max\{n; \text{deg}(\phi_n) < p'\}$.
- to enforce continuity: if e.g. from theory it is known that the problem features only a jump in gradient, one may set $N_S = \{n; \text{deg}(\phi_n) = 1\}$. This option is currently not supported by the BoSSS framework, but could be easily added.

§129: Definition and Remark - order of the extended DG basis: It remains to define some order for the elements in the Cut-Cell basis Φ^X , in order to define the coordinate mapping for a cut-cell DG field.

In some cut cell $K_j \in \text{Cut}(\varphi)$, the basis functions are sorted in the sequence: “separate DOF for species \mathfrak{A} ” – “separate DOF for species \mathfrak{B} ” – “common DOF”; For a cut cell,

$$\left[\phi_{j,n}^X \right]_{n=0,\dots,(N_p+\#N_S-1)} = \begin{bmatrix} \phi_{j,0} \cdot \mathbf{1}_{\mathfrak{A}(t)} \\ \vdots \\ \phi_{j,\#N_S-1} \cdot \mathbf{1}_{\mathfrak{A}(t)} \\ \phi_{j,0} \cdot \mathbf{1}_{\mathfrak{B}(t)} \\ \vdots \\ \phi_{j,\#N_S-1} \cdot \mathbf{1}_{\mathfrak{B}(t)} \\ \phi_{j,\#N_S} \\ \vdots \\ \phi_{j,N-1} \end{bmatrix}^T \quad \left. \begin{array}{l} \left. \begin{array}{l} \text{separate DOF's of } \mathfrak{A} \\ \text{separate DOF's of } \mathfrak{B} \end{array} \right\} \cdot \\ \text{common DOF's} \end{array} \right\}$$

For an uncut cell, as usual

$$\left[\phi_{j,n}^X \right]_{n=0,\dots,N_p-1} = \begin{bmatrix} \phi_{j,0} \\ \vdots \\ \phi_{j,N_p-1} \end{bmatrix}^T .$$

The number of DOF's per cell is given by

$$N_{\text{DOF}}(j) = \begin{cases} N_p & \text{if } K_j \notin \text{Cut}(\varphi) \\ N_p + \#N_S & \text{if } K_j \in \text{Cut}(\varphi) \end{cases} .$$

Given those definitions,

$$\Phi^X := \{ \phi_{j,n}^X; 0 \leq j < J, 0 \leq n < N_{\text{DOF}}(j) \} .$$

§130: Remark - non-orthogonality of the extended basis: It is quite clear that, within a cut cell K_j , the basis functions $\phi_{j,n}^X$ are not orthogonal any more, i.e. $\langle \phi_{j,n}^X, \phi_{j,m}^X \rangle_{K_j} \neq \delta_{n,m}$.

Numerically, even the linear independence may be “very weak”, i.e. the determinant of the matrix $\left[\langle \phi_{j,n}^X, \phi_{j,m}^X \rangle_{K_j} \right]_{\substack{n=1,\dots,N_{\text{DOF}}(j) \\ m=1,\dots,N_{\text{DOF}}(j)}}$ may be close to singular, especially if either $\mathfrak{A} \cap K_j$ or $\mathfrak{B} \cap K_j$ get very small.

E.g. consider a quadratic cell $K_j = (-1, 1)^2$ in 2D; two orthonormal basis functions are $\phi_{j,0} = \frac{1}{2}$ and $\phi_{j,1} = \frac{x \cdot \sqrt{3}}{2}$. Further, let \mathfrak{B} be some rather narrow strip on one boundary, e.g. $\mathfrak{B} = \{(x, y); x > 0.9999\}$. It is obvious that, with a limited numerical precision, it is very difficult to distinguish between $\phi_{j,0} \cdot \mathbf{1}_{\mathfrak{B}}$ and $\phi_{j,1} \cdot \frac{2}{\sqrt{3}} \cdot \mathbf{1}_{\mathfrak{B}}$, because the variation of $\phi_{j,1}$ in x - direction, within the region of $(0.9999, 1)$ is very small.

§131: Definition - Jump-Operator: For a function that is continuous everywhere but on \mathfrak{J} , i.e. $f \in C^0(\Omega \setminus \mathfrak{J})$ and $\mathbf{x} \in \mathfrak{J}$ let be

$$f_{\mathfrak{A}}(\mathbf{x}) := \lim_{\substack{\xi \rightarrow \mathbf{x} \\ \xi \in \mathfrak{A}}} f(\xi) \text{ and } f_{\mathfrak{B}}(\mathbf{x}) := \lim_{\substack{\xi \rightarrow \mathbf{x} \\ \xi \in \mathfrak{B}}} f(\xi)$$

and

$$[[f]](\mathbf{x}) := f_{\mathfrak{B}}(\mathbf{x}) - f_{\mathfrak{A}}(\mathbf{x}).$$

By means of the trace operator (see §46), these definitions can be extended onto the Sobolev spaces $H^n(\Omega \setminus \mathfrak{J})$ resp. $H^n(\Omega^\times \setminus \mathfrak{J}^\times)$.

§132: Definition - species: For a extended DG field $\underline{f} \in XDG_p(\varphi, \mathfrak{K})$, we define “species \mathfrak{A} of \underline{f} ” and “species \mathfrak{B} of \underline{f} ” as

$$\underline{f}_{\mathfrak{A}} := \underline{f} \cdot \mathbf{1}_{\mathfrak{A}} \text{ and } \underline{f}_{\mathfrak{B}} := \underline{f} \cdot \mathbf{1}_{\mathfrak{B}}.$$

We further define that on \mathfrak{J} the definition of $\underline{f}_{\mathfrak{A}}$ and $\underline{f}_{\mathfrak{B}}$ that is given in §131 is still valid. $\underline{f}_{\mathfrak{A}}$ is called “species \mathfrak{A} ” and $\underline{f}_{\mathfrak{B}}$ is called “species \mathfrak{B} ” of the extended DG field \underline{f} .

Within the domain of cut cells it is quite easy to formulate a continuation of species \mathfrak{A} or \mathfrak{B} : E.g. let $\sum_{n=0}^{N-1} \phi_{j,n} \cdot \mathbf{1}_{\mathfrak{A}} \cdot \tilde{f}_{j,n}$ be the representation of the species \mathfrak{A} of some cut cell field $\underline{f} \in XDG_p(\varphi, \mathfrak{K})$ in the cut cell $K_j \in \mathfrak{K}$, i.e. $\underline{f}_{\mathfrak{A}} = \sum_{n=0}^{N-1} \phi_{j,n} \cdot \mathbf{1}_{\mathfrak{A}} \cdot \tilde{f}_{j,n}$ in cell K_j . Then a continuation of $\underline{f}_{\mathfrak{A}}$ into domain \mathfrak{B} , but only within cut cells is canonically defined by just removing the modulation by $\mathbf{1}_{\mathfrak{A}}$, i.e. $Cont_{\mathfrak{A}}(\underline{f}) = \sum_{n=0}^{N-1} \phi_{j,n} \cdot \tilde{f}_{j,n}$ in cell K_j . $Cont_{\mathfrak{A}}(\underline{f})$ and $Cont_{\mathfrak{B}}(\underline{f})$ can be defined by their characteristic property:

§133: Definition - continuation of an extended DG field within $Cut(\varphi)$: The “continuation of $\underline{f}_{\mathfrak{A}}$ onto \mathfrak{B} ”, for some XDG - field $\underline{f} \in XDG_p(\varphi, \mathfrak{K})$ is a mapping

$$XDG_p(\varphi, \mathfrak{K}) \ni \underline{f} \mapsto Cont_{\mathfrak{A}}(\underline{f}) \in DG_p(\mathfrak{K})$$

with the property

$$Cont_{\mathfrak{A}}(\underline{f}) \cdot \mathbf{1}_{\mathfrak{A}} = \underline{f}_{\mathfrak{A}}$$

and

$$Cont_{\mathfrak{A}}(\underline{f}) = 0 \text{ in } \bigcup_{\substack{K \in \mathfrak{K} \\ Vol_D(K \cap \mathfrak{A}) = 0}} K. \quad (6.5)$$

The continuation of species \mathfrak{B} , $Cont_{\mathfrak{B}}$ is defined in analogue fashion.

It is easy to see that the definition of $Cont_{\mathfrak{A}}(\underline{f})$ and $Cont_{\mathfrak{B}}(\underline{f})$ is unique: Because $Cont_{\mathfrak{A}}(\underline{f}) \in DG_p(\mathfrak{K})$, i.e. $Cont_{\mathfrak{A}}(\underline{f})$ must be represented by DG polynomials, the choice is unique in cells K where $\underline{f}_{\mathfrak{A}} \neq 0$. Eq. 6.5 just states that $Cont_{\mathfrak{A}}(\underline{f}) = 0$ in all other cells. In BoSSS, this continuation could be acquired by ⁴.

6.3 Poisson equation

§134: Notation - restricted differential operator: For an open set $X \subset \Omega$, a function $f \in H^n(\Omega)$ we define, for a differential operator ∂^α ,

$$\partial^\alpha|_X f := \begin{cases} \partial^\alpha(f|_X) & \text{in } X \\ 0 & \text{in } \Omega \setminus X \end{cases}$$

the restriction of ∂^α onto X .

§135: Example : Let H be the Heaviside function. Then $\frac{\partial}{\partial x} \Big|_{\mathbb{R} \setminus \{0\}} H(x) = 0$, while $\frac{\partial}{\partial x} H(x)$ is not defined at $x = 0$, at least not in the classical sense.

The Poisson problem with jump, specified in detail within §136, may stand as a prototype problem for a linear PDE with jumps. It is assumed that the problem is well-posed.

§136: Definition - Poisson equation with a jump: The full problem is given as

$$\left\{ \begin{array}{ll} \Delta|_{\Omega \setminus \mathfrak{J}} \Psi = f & \text{in } \Omega \\ \llbracket \Psi \rrbracket = g_1 & \text{on } \mathfrak{J} \\ \llbracket \nabla \Psi \cdot \mathbf{n}_{\mathfrak{J}} \rrbracket = g_2 & \text{on } \mathfrak{J} \\ \Psi = g_{\text{Diri}} & \text{on } \Gamma_{\text{Diri}} \\ \nabla \Psi \cdot \mathbf{n}_{\partial\Omega} = g_{\text{Neu}} & \text{on } \Gamma_{\text{Neu}} \end{array} \right. , \quad (6.6)$$

⁴BoSSS.Foundation.LevelSet.CutCellField.GetSpeciesShadowField

with the disjoint decomposition $\partial\Omega = \Gamma_{\text{Diri}} \cup \Gamma_{\text{Neu}}$ of the boundary into Dirichlet- and Neumann-region.

One solution to this problem may be a grid that coincides with the interface \mathcal{I} . In certain situations this may be not applicable, e.g. when the Problem has to be solved multiple times with a time-dependent interface $\mathcal{I}(t)$. As an example for a situation like this, consider an incompressible multi-phase Navier-Stokes - solver, where the Poisson solver is an essential operation for the pressure correction.

The solution we come up with contains two ingredients that we refer to as “jump subtraction” and “patching”. For jump subtraction, we assume to know an Ansatz function $\Psi_A \in H^2(\Omega \setminus \mathcal{I})$ that fulfills the jump condition, i.e.

$$\begin{cases} \llbracket \Psi_A \rrbracket = g_1 & \text{on } \mathcal{I} \\ \llbracket \nabla \Psi_A \cdot \mathbf{n}_{\mathcal{I}} \rrbracket = g_2 & \text{on } \mathcal{I} \end{cases}$$

and is sufficiently smooth everywhere else. Then, instead of solving Eq. 6.6, one could define the decomposition

$$\Psi = \Psi_S + \Psi_A,$$

which immediately yields $\Psi_S \in H^2(\Omega)$, and solve the equation

$$\begin{cases} \Delta \Psi_S = f - \Delta|_{\Omega \setminus \mathcal{I}} \Psi_A & \text{in } \Omega \\ \Psi_S = g_{\text{Diri}} - \Psi_A & \text{on } \Gamma_{\text{Diri}} \\ \nabla \Psi_S \cdot \mathbf{n}_{\partial\Omega} = g_{\text{Neu}} - \nabla \Psi_A \cdot \mathbf{n}_{\partial\Omega} & \text{on } \Gamma_{\text{Neu}} \end{cases} \quad (6.7)$$

The remaining problem is the construction of the Ansatz function Ψ_A ; while this is a challenging problem in the whole domain Ω it seems, assuming the presence of a signed-distance Level-Set, quite easy in a close neighbourhood of \mathcal{I} , e.g. within the cells in $\text{Cut}(\varphi)$. An example for such a construction is given in §142. So, within this neighbourhood the decomposed equation, Eq. 6.7, is discretized. Everywhere outside the interface neighbourhood, where Ψ_A is unknown, Eq. 6.6 is discretized. These two domains are patched together, the details are outlined in subsequence.

§137: Notation - Laplace operator: Subsequently, the discretization of the Laplace operator is denoted by

$$DG_p(\mathfrak{K}) \ni \underline{\Psi} \mapsto \mathbf{L}_{\Delta}(\underline{\Psi}) \in DG_p(\Omega)$$

and given as

$$\tilde{\mathbf{L}}_{\Delta}(\tilde{\Psi}) := \mathcal{M}_{\Delta} \cdot \tilde{\Psi} + c'.$$

The operators matrix \mathcal{M}_Δ and its affine part c' may be defined as in §95 or §96. Note that it is defined and required only on the standard, not-extended DG space $DG_p(\mathfrak{K})$.

§138: Notation : Subsequently, we define:

- the patching domain $K_C = \bigcup_{K \in \text{Cut}(\varphi)} K$.
- the decomposition $\underline{\Psi} = \underline{\Psi}_S + \underline{\Psi}_A$, with $\underline{\Psi}, \underline{\Psi}_A \in XDG_p(\varphi, \mathfrak{K})$ and $\underline{\Psi}_S \in DG_p(\mathfrak{K})$.
- the patched decomposition $\underline{\Psi}_+ := \underline{\Psi} - \mathbf{1}_{K_C} \cdot \underline{\Psi}_A$.

An explicit form of the patched discretization is difficult and lengthy to write down; therefore we define it by its characteristic property.

§139: Corollary and Definition - characteristic property of patching decomposition: For an Ansatz function $\underline{\Psi}_A \in XDG_p(\varphi, \mathfrak{K})$ with no jump and no kink on the boundary of the patching domain, there exists a vector $c_+(\underline{\Psi}_A \cdot \mathbf{1}_{K_C}) \in \mathbb{R}^{J \cdot N_p}$ so, that for all $\underline{\Psi}_S \in DG_p(\mathfrak{K})$ the equation

$$\mathcal{M}_\Delta \tilde{\Psi}_+ + c_+(\underline{\Psi}_A \cdot \mathbf{1}_{K_C}) = \mathcal{M}_\Delta \tilde{\Psi}_S + \sum_{S \in \{\mathfrak{A}, \mathfrak{B}\}} \mathbf{1}_{S \setminus K_C} \cdot \mathcal{M}_\Delta \cdot \text{Cont}_{S \setminus K_C}(\underline{\Psi}_A). \quad (6.8)$$

holds and that

$$\text{supp}(c_+(\underline{\Psi}_A \cdot \mathbf{1}_{K_C})) = \bigcup_{\substack{K \in \mathfrak{K} \\ \text{Vol}_{D-1}(\partial K_C \cap K) \neq 0}} K, \quad (6.9)$$

where $c_+(\underline{\Psi}_A \cdot \mathbf{1}_{K_C}) := \sum_{\substack{0 \leq j \leq J-1 \\ 0 \leq n \leq N-1}} \phi_{j,n} \cdot [c_+(\%)]_{\text{map}(0,j,n)}$ Subsequently, c_+ will be referred to as the *patching-correction* - vector.

Rationale: existence follows because $\underline{\Psi} \in DG_p(\mathfrak{K}) + \underline{\Psi}_A$.

Eq. 6.8 could be interpreted as

$$\begin{aligned} \mathbf{L}_\Delta(\underline{\Psi}_+) + c_+(\underline{\Psi}_A \cdot \mathbf{1}_{K_C}) &\approx \Delta \Psi_S + \sum_{S \in \{\mathfrak{A}, \mathfrak{B}\}} \mathbf{1}_{S \setminus K_C} \cdot \Delta \Psi_A \\ &= \Delta \Psi_S + \mathbf{1}_{\Omega \setminus K_C} \cdot \Delta \Psi_A \\ &= \begin{cases} \Delta \Psi_+ & \text{in } K_C \\ \Delta \Psi & \text{elsewhere, i.e. in } \Omega \setminus K_C \end{cases} \end{aligned}$$

Here, it is assumed that

$$\mathbf{1}_{S \setminus K_C} \cdot (\mathbf{L}_\Delta(\text{Cont}_S(\underline{\Psi}_A)) - \mathbf{L}_\Delta(0)) \approx \mathbf{1}_{S \setminus K_C} \cdot \Delta \Psi_A,$$

i.e. that the left-hand-side of the “equation” above is, for species S , an approximation to the Laplacian of the Ansatz function outside of the patching domain. Therefore it is required that the stencil of the operator \mathbf{L}_Δ is just one cell: since the continuation $Cont_{\mathfrak{A}}(\underline{\Psi}_A)$ of the Ansatz function will contain on $\partial K_C \cap \mathfrak{B}$, the approximation of Laplacian in all adjacent cells will be bad, due to the penalization of the jump. Therefore, within the patching domain the approximation will be bad, but due to the 1-cell stencil the approximation $\mathbf{L}_\Delta(Cont_S(\underline{\Psi}_A)) \approx \Delta \Psi_A$ holds with the same accuracy as the classical Interior Penalty method on species- \mathfrak{A} cells outside of the patching domain, i.e. for cells within $(\Omega \setminus K_C) \cap \mathfrak{A}$.

Eq. 6.9 describes that the support of \underline{c}_+ is limited to cells that share an edge with the boundary of the patching domain K_C .

In BoSSS, the vector c_+ , for an Interior Penalty discretization of the Laplace operator, can be computed by ⁵.

It is still an issue to approximate $\Delta|_{\Omega \setminus \mathfrak{J}} \Psi_A$ on the patching domain K_C , since Ψ_A must be assumed to be unknown outside of K_C .

§140: Definition - restricted evaluation of \mathbf{L}_Δ : (i) For $\underline{\Psi}_A \in DG_p(\mathfrak{K})$, we define

$$\mathbf{L}_{rst}(\underline{\Psi}_A \cdot \mathbf{1}_{K_C}) := \mathbf{L}_\Delta(\underline{\Psi}_A \cdot \mathbf{1}_{K_C}) + \underline{c}_+(\underline{\Psi}_A \cdot \mathbf{1}_{K_C}).$$

(ii) For $\underline{\Psi}_A \in XDG_p(\varphi, \mathfrak{K})$, we define

$$\mathbf{L}_{rst}(\underline{\Psi}_A \cdot \mathbf{1}_{K_C}) := \sum_{S \in \{\mathfrak{A}, \mathfrak{B}\}} \mathbf{1}_S \cdot \mathbf{L}_{rst}(Cont_S(\underline{\Psi}_A) \cdot \mathbf{1}_{K_C})$$

Interpretation: The purpose of $\mathbf{L}_{rst}(\underline{\Psi}_A \cdot \mathbf{1}_{K_C})$ is to be an approximation to $\Delta|_{\Omega \setminus \mathfrak{J}} \Psi_A$ on the domain K_C , although $\underline{\Psi}_A$ is practically not known outside of K_C , i.e. only $\underline{\Psi}_A \cdot \mathbf{1}_{K_C}$ is known. We illustrate that by the diagram:

$$\begin{array}{ccc} \underline{\Psi}_A & \xrightarrow{\text{patch.}} & \underline{\Psi}_A \cdot \mathbf{1}_{K_C} \\ \mathbf{L}_\Delta \downarrow & & \downarrow \mathbf{L}_{rst} \\ \mathbf{L}_\Delta(\underline{\Psi}_A) - \mathbf{L}_\Delta(0) & \longmapsto & (\mathbf{L}_\Delta(\underline{\Psi}_A) - \mathbf{L}_\Delta(0)) \cdot \mathbf{1}_{K_C} \approx \mathbf{L}_{rst}(\underline{\Psi}_A \cdot \mathbf{1}_{K_C}) \end{array}$$

This diagram is commutative, i.e. the \approx in the lower right corner turns into an equality, if the jump of $\underline{\Psi}_A$ and $\nabla \underline{\Psi}_A$ on ∂K_C are zero.

⁵BoSSS.Solution.MultiphaseZoo.ipLaplacePatched.ComputeLaplaceAffine2

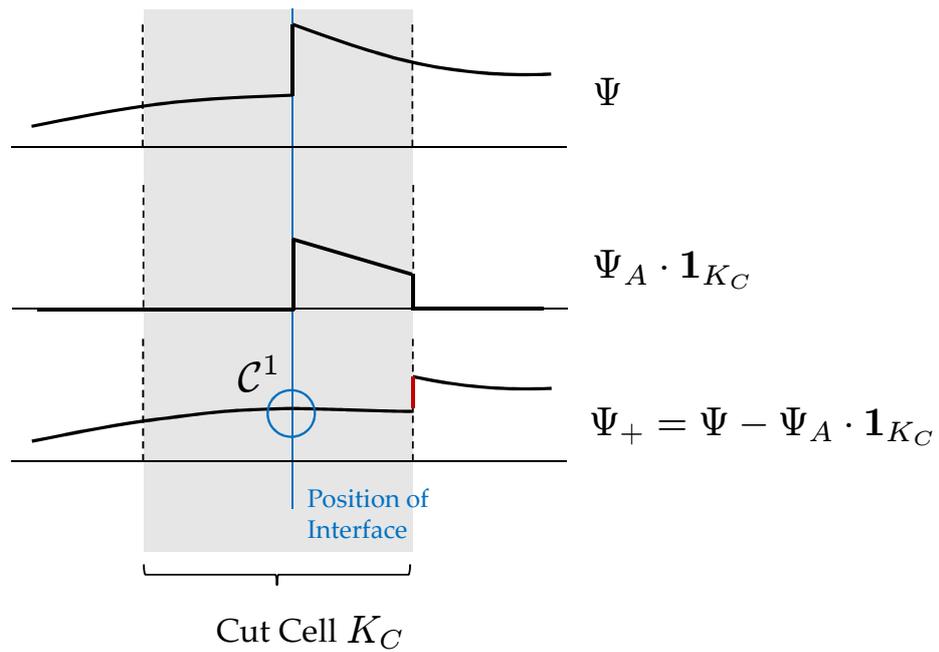


Figure 6.4: Illustration of patching, 1D example. One may assume that the solution of some singular problem looks like Ψ . When the patched Ansatz $\Psi_A \cdot \mathbf{1}_{K_C}$ – which is 0 outside of the patching domain K_C – is subtracted from Ψ , two things happen: at first, jump and kink at \mathcal{I} is removed, i.e. Ψ_+ and its first derivative are continuous. Second, another artificial jump in Ψ_+ arises, which is located exactly at a cell boundary. The idea of patching is to correct this jump in the flux functions, by the patching-correction vector c_+ .

Right now we are ready to formulate the central result of this section, the patched-decomposition scheme for the Poisson equation; it is illustrated in figure 6.6.

§141: Corollary - XDG scheme for a Poisson equation with an additive jump: Given is a discretization $\underline{\Psi}_A \cdot \mathbf{1}_{K_C} \in XDG_p(\varphi, \mathfrak{K})$ of the Ansatz function Ψ_A within the patching domain. A consistent discretization of the discontinuous Poisson problem defined in §136 in the sense of §47 is given by

$$\mathcal{M}_\Delta \cdot \tilde{\Psi}_+ + c' + c_+(\underline{\Psi}_A \cdot \mathbf{1}_{K_C}) = \langle \underline{\Phi}, Proj_p(f) - Proj_p(\mathbf{L}_{rst}(\underline{\Psi}_A \cdot \mathbf{1}_{K_C})) \rangle_\Omega. \quad (6.10)$$

The numerical solution $\underline{\Psi} \in XDG_p(\varphi, \mathfrak{K})$ of the discontinuous Poisson problem is given as

$$\underline{\Psi} = \underline{\Psi}_+ + \underline{\Psi}_A \cdot \mathbf{1}_{K_C}.$$

Rationale: Given is the Poisson equation

$$\Delta|_{\Omega \setminus \mathcal{J}} \Psi = f.$$

Jump-subtraction yields:

$$\Delta \Psi_S = f - \Delta|_{\Omega \setminus \mathcal{J}} \Psi_A.$$

The DG discretization can be written as

$$\mathcal{M}_\Delta \tilde{\Psi}_S + c' = Proj_p(f - \Delta|_{\Omega \setminus \mathcal{J}} \Psi_A)$$

and by applying patching one gets

$$\mathcal{M}_\Delta \tilde{\Psi}_S + c' + c_+ = Proj_p(f - \mathbf{1}_{K_C} \cdot \Delta|_{\Omega \setminus \mathcal{J}} \Psi_A).$$

Finally, one approximates

$$\mathbf{1}_{K_C} \cdot \Delta|_{\Omega \setminus \mathcal{J}} \Psi_A \approx Proj_p(\mathbf{L}_{rst}(\underline{\Psi}_A \cdot \mathbf{1}_{K_C}))$$

to get Eq. 6.10. End of Rationale.

§142: Remark - Construction of Ansatz function: Given are $g_1, g_2 \in L^2(\mathcal{J})$; It is further assumed that the curvature of \mathcal{J} is bounded, which is the case if $\Delta\varphi$ is bounded. Then an Ansatz Ψ_A function with

$$\begin{cases} [[\Psi_A]] = g_1 & \text{on } \mathcal{J} \\ [[\nabla \Psi_A \cdot \mathbf{n}_{\mathcal{J}}]] = g_2 & \text{on } \mathcal{J} \end{cases}$$

is given in a neighbourhood of \mathcal{I} by

$$\Psi_A := \begin{cases} 0 & \text{for } \mathbf{x} \in \mathfrak{A} \\ \varphi(\mathbf{x}) \cdot g_2(\mathbf{x}_0) + g_1(\mathbf{x}_0) & \text{for } \mathbf{x} \in \mathfrak{B} \end{cases}$$

with $\mathbf{x}_0 = \mathbf{x} - \varphi(\mathbf{x}) \cdot (\nabla \varphi)(\mathbf{x})$ for a signed-distance Level-Set – function φ . An implementation of this can be found at ⁶.

§143: Example - Poisson equation with additive jump condition: Consider the notation of §141 and §136. Given is a domain $\Omega = (0, 10)^2$ discretized by a Cartesian equidistant grid with 64×64 equidistant cells. The polynomial degree of the DG interpolation is 2 and there are no common modes for both phases; the phases are explicitly given as

$$\mathfrak{A} = \{\mathbf{x} \in \Omega; |\mathbf{x} - (5, 5)| > 3\}, \mathcal{I} = \partial\mathfrak{A}, \text{ and } \mathfrak{B} = \Omega \setminus (\mathfrak{A} \cup \mathcal{I}).$$

The Ansatz function is explicitly given as

$$\Psi_A(\mathbf{x}) = \begin{cases} 0 & \text{if } \mathbf{x} \in \mathfrak{A} \\ \frac{x}{5} & \text{if } \mathbf{x} \in \mathfrak{B}' \end{cases},$$

which coincides with the jump conditions

$$\begin{aligned} [[\Psi]] &= \frac{x}{5} \\ [[\mathbf{n}_{\mathcal{I}} \cdot \nabla \Psi]] &= \frac{x-5}{17.5}. \end{aligned}$$

On $\partial\Omega$, homogeneous Dirichlet boundary conditions are assumed; the right-hand-side of the Poisson problem is given as

$$f = \begin{cases} -1 & \text{if } |x - 2| < 2 \text{ and } |y - 5| < 2 \\ 0 & \text{otherwise} \end{cases}.$$

Numerical results are shown in figure 6.5, the Ansatz function and the Patching are illustrated in figure 6.6.

Notes on Performance: As already mentioned, the presented XDG method may pay of most in the case of moving interfaces, i.e. if the Poisson problem is solved multiple times with a different interface position at each time. In this case the matrix of the system stays the same and no re-meshing is necessary when the interface position is changed. Exactly the opposite would be the case, if the grid would be re-meshed and adapted to the interface position every time it changes.

⁶BoSSS.Solution.MultiphaseZoo.SomeTools.AnsatzConstruction

The XDG performance is compared to a Poisson problem without any jump condition, i.e. $\Delta\Psi = f$ with homogeneous Dirichlet boundary conditions on $\partial\Omega$, using the same grid as for the Cut-Cell problem.

A conjugate gradient (CG) algorithm (see (Meister 2004), page 120), without any preconditioning was used for solving the linear system. The termination criterion is set to an absolute residual 2-norm of 10^{-7} , i.e. the algorithm terminates if $\|M \cdot x - b\|_2 < 10^{-7}$ for the linear system $M \cdot x = b$. One CPU core (AMD Phenom II X4 940 3 GHz) was used to run the solver, using the monkey-implementation presented in chapter 4.

The overhead, in terms of CPU time, can be broken down into three individual parts:

- the computation of the Ansatz function Ψ_A . In this example, it is explicitly known, but this may vary from case to case; therefore, the effect on runtime is not further investigated for this example.
- the computation of the patching-correction vector c_+ : runtime is 0.015 seconds. It should be noted that there is a lot of software optimization potential.
- the solution of the linear system: runtime is 3.6 seconds for 1'978 iterations.

The runtime of the matrix assembly – which is exactly the same for both, the discontinuous and the smooth problem – is approximately 0.8 seconds, but that may heavily depend on problem dimension (2D, 3D), DG polynomial degree and many other factors. The “cost” of computing c_+ seems to be quite small in comparison to matrix assembly and negligible in comparison to the solver runtime.

The solver runtime heavily depends on preconditioning, but due to a lack of effective preconditioning techniques this was not further investigated. For the smooth problem the solver runtime, for the same residual threshold is 2.9 seconds for 1'604 CG iterations, which is approximately 20 % faster than for the discontinuous problem.

6.4 Brief overview about theory of distributions

For analysing discontinuous PDE's we found the theory of distributions very useful, because it allows to apply the differential form of a conservation law onto problems that are not differentiable in the classical sense.

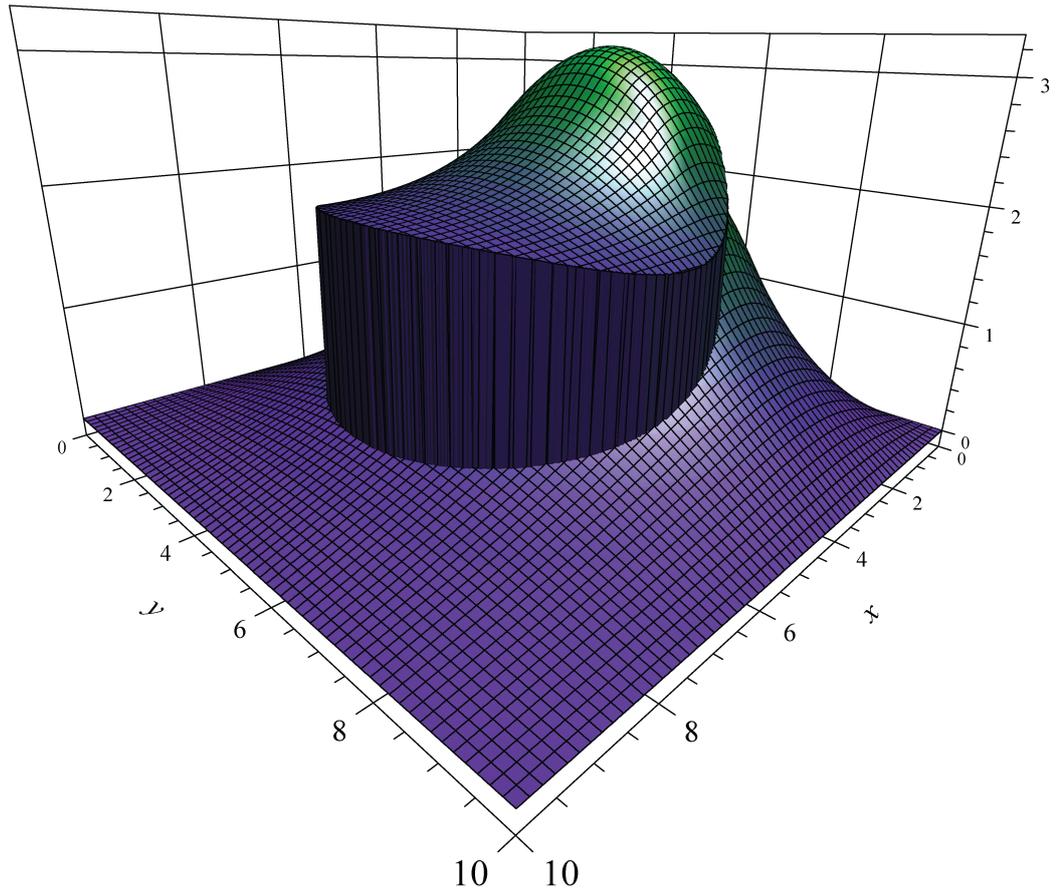


Figure 6.5: Poisson problem $\Delta|_{\Omega \setminus \mathcal{I}} \Psi = f$ with jump $[[\Psi]] = \frac{x}{5}$ and kink $[[\mathbf{n}_{\mathcal{I}} \cdot \nabla \Psi]] = \frac{x-5}{17.5}$ on the interface \mathcal{I} , which is a circle around point $(5, 5)$. The right-hand-side f is equal to -1 for $0 < x < 4$ and $3 < y < 7$. On the boundary of the domain $(0, 10)^2$, discretized by 64×64 equidistant cells, homogeneous Dirichlet boundary conditions are used.

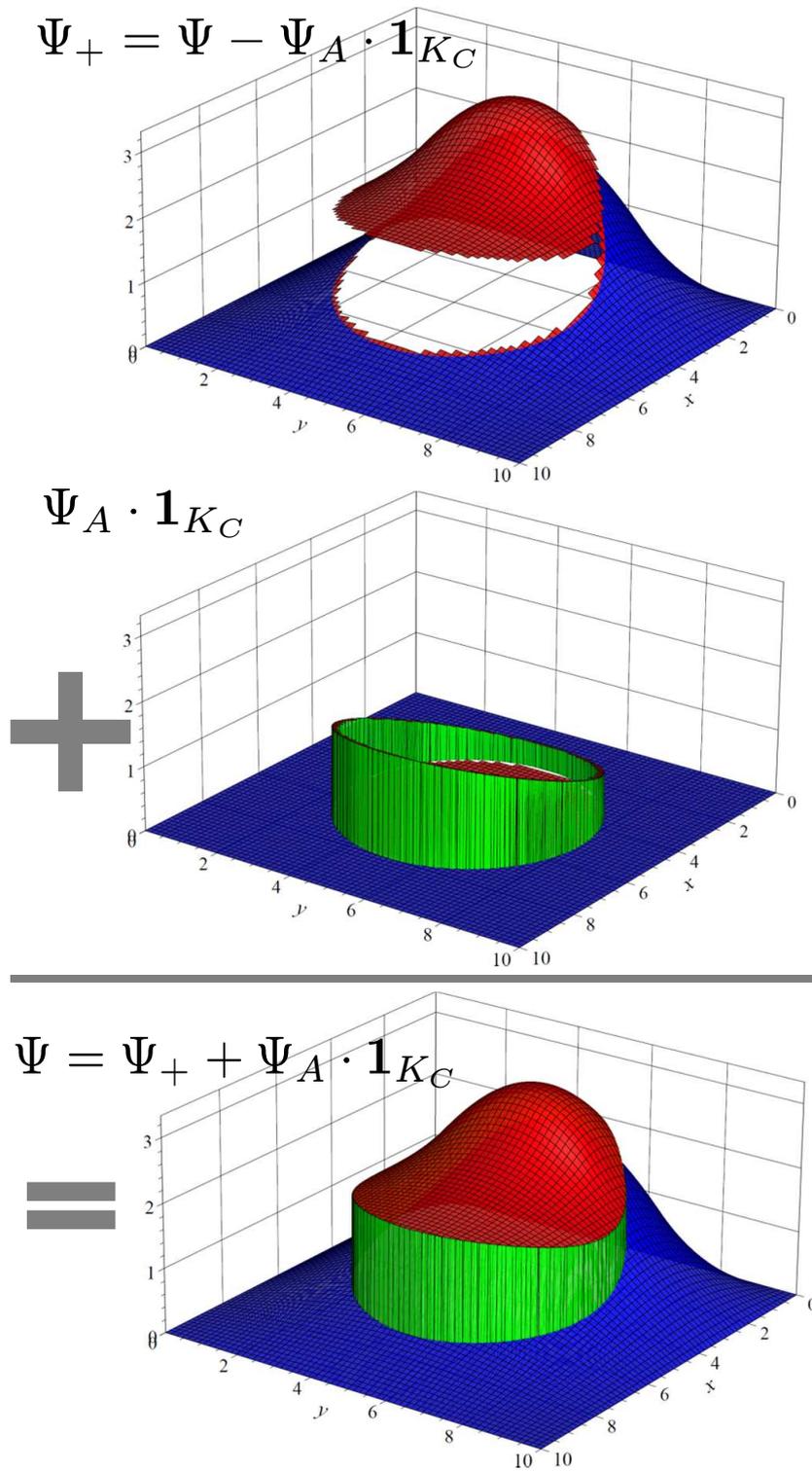


Figure 6.6: Illustration of patching, 2D example: The solution Ψ_+ of the patched-decomposed equation (Eq. 6.10) shows an “artificial” jump on the boundary of the patching domain, which compensates with the jump in the patched Ansatz $\Psi_A \cdot \mathbf{1}_{K_C}$. In sum, they give the numerical solution of the singular problem.

E.g. the jump conditions for the incompressible multiphase Navier-Stokes problem can be derived from the differential form of the Navier-Stokes equation by the means of distributions. The notation is usually more compact than the integral form of the conservation law.

Distribution theory was developed by Laurent Schwartz in 1944 (Schwartz 1966) who himself was heavily influenced by the works of Jacques Hadamard, Paul Dirac and Sergei Lwowitsch Sobolew. In the year 1951, Schwartz received the Fields medal for his work.

For the rather small subset of the theory that is presented within this section, we follow the textbooks (Walter 1973) and (Triebel 1980).

At first, the basic definition of distributions, their associated test-function space and the distributional derivative should be recalled.

§144: Definition/Notation - Test Functions: In the sense of distributional theory the space of test functions on Ω is defined:

$$\mathcal{D}(\Omega) := \{\mathcal{K} \in \mathcal{C}^\infty(\Omega); \text{supp}(\mathcal{K}) \text{ is compact, i.e. closed and bounded}\}$$

§145: Definition - convergence of Test Functions: A series $(\mathcal{K}_1, \mathcal{K}_2, \dots)$ in $\mathcal{D}(\Omega)^{\mathbb{N}}$ is called to be convergent against $\mathcal{K} \in \mathcal{D}(\Omega)$, i.e. $\mathcal{K} = \lim_{i \rightarrow \infty} (\mathcal{K}_i)$, if, and only if

$$\left\{ \begin{array}{l} \exists K \subset \Omega \text{ compact} : \forall k : \text{supp}(\mathcal{K}_k) \subset K \\ \forall \alpha \in \mathbb{N}^D : \text{uniform convergence of } \partial^\alpha \mathcal{K}_k \rightarrow \partial^\alpha \mathcal{K} \end{array} \right.$$

§146: Definition - distributions: A linear functional T , notated as

$$\mathcal{D}(\Omega) \ni \mathcal{K} \mapsto (T, \mathcal{K}) \in \mathbb{R}$$

is called to be a distribution if for all series $(\mathcal{K}_i)_{i=0, \dots, \infty}$ with $\mathcal{K}_i \xrightarrow{\text{in } \mathcal{D}} 0$ also $(T, \mathcal{K}_i) \xrightarrow{\text{in } \mathbb{R}} 0$. The set of all distributions is denoted as $\mathcal{D}'(\Omega)$.

§147: Definition - support of a distribution: For $T \in \mathcal{D}'(\Omega)$, one defines the support of T ,

$$\text{supp}(T) := \Omega \setminus \{\mathbf{x} \in \Omega; \exists \epsilon > 0 : \forall \mathcal{K} \in \mathcal{D}(\Omega) \text{ with } \text{supp}(\mathcal{K}) \subset \{\mathbf{y} : |\mathbf{x} - \mathbf{y}| < \epsilon\} : (T, \mathcal{K}) = 0\}.$$

One – almost magical – property of the space $\mathcal{D}'(\Omega)$ is, that it not only contains classical functions, embedded as regular distributions, but also so-called singular distributions, like the delta-distribution that represent derivatives of functions that are not differentiable in a classical sense, like the Heaviside-function.

§148: Definition - special distributions: • *regular distributions:* functions $f \in L^1_{\text{loc}}(\Omega)$ (see ⁷) are called *regular distributions* and embedded into $\mathcal{D}'(\Omega)$ by

$$(f, \mathcal{K}) := \int_{\Omega} f \cdot \mathcal{K} \, d\mathbf{x}.$$

- *the (Dirac-) delta - distribution:* for $\mathbf{x}_0 \in \Omega$, one defines

$$(\delta_{\mathbf{x}_0}, \mathcal{K}) := \mathcal{K}(\mathbf{x}_0).$$

Further, for the $(D - 1)$ – dimensional manifold \mathfrak{J} and $f \in L^1_{\text{loc}}(\Omega)$, one defines

$$(\delta_{\mathfrak{J}} \cdot f, \mathcal{K}) := \int_{\mathfrak{J}} f \cdot \mathcal{K} \, dS.$$

In general, the product of two distributions is not defined. There is e.g. no well-defined expression for $\delta \cdot \delta$. A definition for the product is only available as long as one of the factors is a smooth, i.e. C^∞ - function. (“The Space of distributions is an C^∞ - module.”)

§149: Definition - multiplication distributions with functions in $C^\infty(\Omega)$: The product of $f \in C^\infty(\Omega)$ and $T \in \mathcal{D}'(\Omega)$ is given by

$$(f \cdot T, \mathcal{K}) := (T, f \cdot \mathcal{K})$$

for a test function $\mathcal{K} \in \mathcal{D}(\Omega)$.

The main idea for defining derivatives of discontinuous functions like e.g. the Heaviside-function is to “shift” the derivative to the test function, which is C^∞ :

§150: Definition - derivative of a distribution: A derivative $\partial^\alpha T$, for a multi-index $\alpha \in \mathbb{N}^D$, of a distribution T is given by

$$(\partial^\alpha T, \mathcal{K}) := (-1)^{|\alpha|} \cdot (T, \partial^\alpha \mathcal{K}).$$

⁷ $L^1_{\text{loc}}(\Omega) := \{g : \Omega \rightarrow \mathbb{R}; g \text{ is Lebesgue measurable and } \forall K \subset \Omega \text{ compact : } \int_K |g| \, d\mathbf{x} < \infty\}$

It can be shown that, for regular distributions that are differentiable in the classical sense, the distributional derivative coincides with the classical one. We refer to the cited textbooks for more detailed information on that.

For the problems investigated in this work, i.e. certain PDE's with jumps at the interface, we have to calculate distributional derivative of functions with jumps at the interface \mathfrak{J} . The following two paragraphs create a direct link between the jump operator and the δ - distribution, resp. its derivative. It is not within the scope of this work, but it should be mentioned that this relation can be used in the scope of two-phase flows to translate so-called tow-fluid models with jump conditions into distributional one-fluid models and vice-versa.

§151: Corollary - derivative of functions in $\mathcal{C}^1(\Omega \setminus \mathfrak{J})$: For $u \in \mathcal{C}^1(\Omega \setminus \mathfrak{J})$

$$\frac{\partial}{\partial x_i} u = \frac{\partial}{\partial x_i} \Big|_{\Omega \setminus \mathfrak{J}} u + \delta_{\mathfrak{J}} \cdot \mathbf{n} \cdot \mathbf{e}_i \cdot \llbracket u \rrbracket$$

Proof:

$$\begin{aligned} \left(\frac{\partial}{\partial x_i} u, \mathfrak{K} \right) & \stackrel{\text{Def.}}{=} \left(u, \frac{\partial}{\partial x_i} \mathfrak{K} \right) \\ & = - \int_{\text{supp}(\mathfrak{K}) \cap \mathfrak{A}} u \cdot \text{div}(\mathbf{e}_i \mathfrak{K}) \, d\mathbf{x} - \int_{\text{supp}(\mathfrak{K}) \cap \mathfrak{B}} \% \, d\mathbf{x} \\ & \stackrel{\text{Gauss-f-thm.}}{=} \int_{\text{supp}(\mathfrak{K}) \cap \mathfrak{A}} \nabla u \cdot \mathbf{e}_i \cdot \mathfrak{K} \, d\mathbf{x} + \int_{\text{supp}(\mathfrak{K}) \cap \mathfrak{B}} \% \, d\mathbf{x} \\ & \quad - \underbrace{\int_{\partial \text{supp}(\mathfrak{K}) \cap \mathfrak{A}} \mathbf{e}_i \cdot \mathbf{n} u \cdot \mathfrak{K} \, dS}_{=0} - \underbrace{\int_{\partial \text{supp}(\mathfrak{K}) \cap \mathfrak{B}} \% \, dS}_{=0} \\ & \quad + \int_{\mathfrak{J}} \mathbf{e}_i \cdot \mathbf{n}_{\mathfrak{J}} (u_{\mathfrak{B}} - u_{\mathfrak{A}}) \cdot \mathfrak{K} \, dS \\ & = \left(\frac{\partial}{\partial x_i} \Big|_{\Omega \setminus \mathfrak{J}} u, \mathfrak{K} \right) + (\delta_{\mathfrak{J}} \cdot \mathbf{n} \cdot \mathbf{e}_i \cdot \llbracket u \rrbracket, \mathfrak{K}) \end{aligned}$$

End of Proof.

§152: Corollary - Laplacian of a jumping function: For $u \in \mathcal{C}^1(\Omega \setminus \mathfrak{J})$

$$\text{div}(\nabla u) = \Delta|_{\Omega \setminus \mathfrak{J}} u + \delta_{\mathfrak{J}} \cdot \llbracket \nabla u \cdot \mathbf{n}_{\mathfrak{J}} \rrbracket + \underbrace{\sum_{d=1}^D \frac{\partial}{\partial x_d} (\delta_{\mathfrak{J}} \llbracket u \cdot \mathbf{n}_{\mathfrak{J}} \cdot \mathbf{e}_d \rrbracket)}_{=:\text{div}(\delta_{\mathfrak{J}} \llbracket u \cdot \mathbf{n}_{\mathfrak{J}} \rrbracket)}.$$

The proof is a straightforward application of §151.

For the construction of timestepping schemes, one usually has to integrate over time, e.g. the equation $\frac{\partial}{\partial t}u = Op(u)$ is integrated over the interval $(0, \Delta t)$ to gain an implicit Euler scheme $u(\Delta t) - u(0) \approx \Delta t \cdot Op(u(\Delta t))$. Now that has to be done for distributions.

§153: Definition - definite integrals of distributions: (i) Given is a distribution $T \in \mathcal{D}'(\Omega)$; for a Lebesgue - measurable, bounded set $K \subset \Omega$ let be

$$\int_K T d\mathbf{x} := \lim_{i \rightarrow \infty} (T, \mathfrak{K}_i)$$

for a sequence $(\mathfrak{K}_1, \mathfrak{K}_2, \dots) \in \mathcal{D}(\Omega)^{\mathbb{N}}$ with $\mathfrak{K}_i \xrightarrow{\text{in } L^2(\Omega)} \mathbf{1}_K$.

(ii) Given is a distribution $T \in \mathcal{D}'(\Omega \times \Omega_2)$; for a Lebesgue - measurable, bounded set $K \subset \Omega_2$ the distribution (!) $\int_K T d\mathbf{y} \in \mathcal{D}'(\Omega)$ is defined

$$\left(\int_K T d\mathbf{y}, \mathfrak{K} \right) := \lim_{i \rightarrow \infty} (T, (\mathbf{x}, \mathbf{y}) \mapsto \mathfrak{K}(\mathbf{x}) \cdot \mathfrak{z}_i(\mathbf{y}))$$

for a test function $\mathfrak{K} \in \mathcal{D}(\Omega)$ and a sequence $(\mathfrak{z}_1, \mathfrak{z}_2, \dots) \in \mathcal{D}(\Omega)^{\mathbb{N}}$ with $\mathfrak{z}_i \xrightarrow{\text{in } L^2(\Omega_2)} \mathbf{1}_K$.

Because of the relation

$$\frac{\partial}{\partial t} \Big|_{\Omega \times \mathcal{J}^\times} u = \frac{\partial}{\partial t} u - \delta_{\mathcal{J}} \llbracket u \rrbracket \cdot \frac{-s}{\sqrt{1+s^2}}$$

one can argue that

$$\int_{t=t^0}^{t^1} \frac{\partial}{\partial t} \Big|_{\Omega \times \mathcal{J}^\times} u dt = u(t^1, -) - u(t^0, -) - \int_{t=t^0}^{t^1} \delta_{\mathcal{J}^\times} \llbracket u \rrbracket \cdot \frac{-s}{\sqrt{1+s^2}} dt. \quad (6.11)$$

This relation is used in the rationale of §157, in order to construct an implicit Euler scheme for the Heat equation with a jump.

It should be mentioned that it is important for the construction of the numerical schemes to understand the difference between Eq. 6.11 and

$$\int_{t=t^0}^{t^1} \frac{\partial}{\partial t} u dt = u(t^1, -) - u(t^0, -).$$

Whichever of them applies depends on the problem that should be solved.

6.5 Heat equation

§154: Definition - Heat equation with a jump: The full initial-boundary value problem with $u \in H^2(\Omega^\times)$ is given as

$$\left\{ \begin{array}{ll} \frac{\partial}{\partial t} u \Big|_{\Omega^\times \setminus \mathcal{J}^\times} - \nu \cdot \Delta \Big|_{\Omega^\times \setminus \mathcal{J}^\times} u = 0 & \text{in } \Omega^\times \\ \llbracket u \rrbracket = g_1 & \text{on } \mathcal{J}^\times \\ \llbracket \nabla u \cdot \mathbf{n}_{\mathcal{J}} \rrbracket = g_2 & \text{on } \mathcal{J}^\times \\ u = g^{\text{Diri}} & \text{on } \mathbb{R}_{>0} \times \Gamma^{\text{Diri}} \\ \nabla u \cdot \mathbf{n}_{\partial\Omega} = g^{\text{Neu}} & \text{on } \mathbb{R}_{>0} \times \Gamma^{\text{Neu}} \\ u = u^0 & \text{on } \{0\} \times \Omega \end{array} \right. ,$$

with the disjoint decomposition $\partial\Omega = \Gamma^{\text{Diri}} \cup \Gamma^{\text{Neu}}$ of the boundary into Dirichlet- and Neumann-region and a function $\nu \in C^\infty(\Omega)$.

§155: Notation - multiplication: For some $\nu \in C^\infty(\Omega)$ we notate the matrix of the linear mapping

$$DG_p(\mathfrak{K}) \ni f \mapsto \text{Proj}_p(\nu \cdot f) \in DG_p(\mathfrak{K})$$

as \mathcal{M}_ν .

§156: Remark : For $0 \leq j < J$ and $0 \leq n, m < N_p$

$$[\mathcal{M}_\nu]_{\text{map}(0,j,m), \text{map}(0,j,n)} = \langle \nu \cdot \phi_{j,n}, \phi_{j,m} \rangle.$$

All other entries of \mathcal{M} are 0, therefore \mathcal{M} is of block-diagonal shape with $N_p \times N_p$ - blocks and therefore, *numerically easy to invert*, since N_p is usually small (below 100).

Rationale: $\langle \text{Proj}_p(\nu \cdot f), \phi_{j,m} \rangle = \langle \nu \cdot \left(\sum_{j,n} \tilde{f}_{j,n} \cdot \phi_{j,n} \right), \phi_{j,m} \rangle = \sum_{j,n} \tilde{f}_{j,n} \cdot \langle \nu \cdot \phi_{j,n}, \phi_{j,m} \rangle$.

Right now, the foundation for the formulation of an implicit Euler scheme for a scalar Heat equation with jump has been laid out; a rationale for its construction – which is not a proof for its consistency – is given in subsequence to the statement. It is important to notice that the initial value \underline{u}^0 at time t^0 and the numerical solution \underline{u}^1 at time t^1 are in different XDG spaces, since the jump has moved, i.e. $\underline{u}^1 \in XDG_p(\varphi^1, \mathfrak{K})$ but $\underline{u}^0 \in XDG_p(\varphi^0, \mathfrak{K})$. In consequence, almost any linear combination $\alpha \cdot \underline{u}^1 + \beta \cdot \underline{u}^0$ of \underline{u}^1 and \underline{u}^0 has *two* jumps, at \mathcal{J}^1 and at \mathcal{J}^0 and is therefore neither a member of

$XDG_p(\varphi^1, \mathfrak{K})$ nor $XDG_p(\varphi^0, \mathfrak{K})$. Since an implicit Euler scheme requires linear combinations between initial and new solution, this is “lifted” by the projection onto the standard DG space, i.e. by taking linear combinations of the form $\alpha \cdot \underline{u}^1 + \beta \cdot Proj_p(\underline{u}^0) \in XDG(\mathfrak{J}^1, \mathfrak{K})$.

§157: Corollary - Implicit Euler scheme for a scalar Heat equation with additive jump condition: Given is the initial-boundary value problem defined in §154; w.l.o.g. let be $t^0 = 0$ and $t^1 > 0$. Then we define/recall, for $i \in \{0, 1\}$...

- the initial value at time t^0 and the solution at time t^1 : $u^i = u(t^i, -)$.
- an Ansatz function $u_A \in H^2(\Omega^\times \setminus \mathfrak{J}^\times)$ that fulfils the jump condition, especially $u_A^i \in H^2(\Omega \setminus \mathfrak{J}(t^i))$.
- the decomposition into smooth part and Ansatz, i.e. $u_S^i = u^i - u_A^i$.
- the DG discretizations $\underline{u}^i, \underline{u}_A^i = Proj(u_A^i) \in XDG_p(\varphi, \mathfrak{K})$ of u^i and u_A^i , as well the discretization $\underline{u}_S^i \in DG_p(\mathfrak{K})$ of u_S^i .
- the Patching domain $K_C := \left(\bigcup_{K \in Cut(\varphi^0)} K \right) \cup \left(\bigcup_{L \in Cut(\varphi^1)} L \right)$, consisting of the cut cells at times t^0 and t^1 .
- the patched decomposition: $\underline{u}_+^i = \underline{u}^i - \mathbf{1}_{K_C} \cdot \underline{u}_A^i$
- a discretization of the Laplace operator, like in §137, represented by the matrix \mathcal{M}_Δ and affine vector c' , consistent with boundary conditions $u = g_{\text{Diri}}(t^1, -)$ on Γ_{Diri} and $\nabla u \cdot \mathbf{n} = g_{\text{Neu}}(t^1, -)$ on Γ_{Neu} in the sense of §47.
- the patching correction vector $c_+(\underline{u}_A^1 \cdot \mathbf{1}_{K_C})$ in the sense of §139, for the patching domain K_C and Ansatz function \underline{u}_A^1 at time t^1 .

An implicit Euler discretization is given by

$$\begin{aligned} & \left(\frac{1}{\Delta t} \cdot \mathcal{M}_v^{-1} - \mathcal{M}_\Delta \right) \cdot \tilde{u}_+^1 = c' + c_+(\underline{u}_A^1 \cdot \mathbf{1}_{K_C}) \\ & \quad + \left\langle \underline{\Phi}, Proj_p \left(\mathbf{L}_{rst}(\underline{u}_A^1 \cdot \mathbf{1}_{K_C}) \right) \right\rangle_\Omega + \\ & \mathcal{M}_v^{-1} \cdot \left(\frac{1}{\Delta t} \tilde{u}_+^0 - \left\langle \underline{\Phi}, Proj_p \left(\frac{\mathbf{1}_{K_C}}{\Delta t} \cdot (\underline{u}_A^1 - \underline{u}_A^0) - \int_{t^0}^{t^1} \delta_{\mathfrak{J}^\times} \cdot \frac{-s}{\sqrt{1+s^2}} \llbracket \underline{u}_A \rrbracket dt \right) \right\rangle_\Omega \right) \end{aligned} \quad (6.12)$$

If \mathcal{M}_Δ is symmetric, also the matrix of the linear system, i.e. $\frac{1}{\Delta t} \mathcal{M}_v^{-1} - \mathcal{M}_\Delta$ is symmetric.

Rationale: Given is the Heat equation, i.e.

$$\frac{\partial}{\partial t} \Big|_{\Omega^\times \setminus \mathcal{J}^\times} u = \nu \cdot \Delta|_{\Omega^\times \setminus \mathcal{J}^\times} u.$$

Integration of the left-hand-side over the interval (t^0, t^1) and an implicit Euler Ansatz for the right-hand-side yield:

$$u^1 - u^0 - \int_{t^0}^{t^1} \delta_{\mathcal{J}^\times} \cdot \frac{-s}{\sqrt{1+s^2}} \llbracket u \rrbracket dt = \Delta t \cdot \nu \cdot \Delta|_{\Omega^\times \setminus \mathcal{J}^\times} u^1. \quad (6.13)$$

Note that $\int_{t^0}^{t^1} \frac{\partial}{\partial t} u dt = u^1 - u^0$, but since $\frac{\partial}{\partial t} \Big|_{\Omega^\times \setminus \mathcal{J}^\times} u = \frac{\partial}{\partial t} u - \delta_{\mathcal{J}^\times} \cdot \frac{-s}{\sqrt{1+s^2}} \llbracket u \rrbracket$ one gets $\int_{t^0}^{t^1} \frac{\partial}{\partial t} u \Big|_{\Omega^\times \setminus \mathcal{J}^\times} dt = u^1 - u^0 - \int_{t^0}^{t^1} \delta_{\mathcal{J}^\times} \cdot \frac{-s}{\sqrt{1+s^2}} \llbracket u \rrbracket dt$.

Inserting the decomposition $u^i = u_S^i + u_A^i$ into Eq. 6.13 yields

$$u_S^1 - u_S^0 + u_A^1 - u_A^0 - \int_{t^0}^{t^1} \delta_{\mathcal{J}^\times} \cdot \frac{-s}{\sqrt{1+s^2}} \llbracket u \rrbracket dt = \Delta t \cdot \nu \cdot \left(\Delta u_S^1 + \Delta|_{\Omega^\times \setminus \mathcal{J}^\times} u_A^1 \right). \quad (6.14)$$

A DG discretization of Eq. 6.14 yields

$$\begin{aligned} & \left(\frac{1}{\Delta t} I - \mathcal{M}_\nu \cdot \mathcal{M}_\Delta \right) \cdot \tilde{u}_S^1 = \\ & \mathcal{M}_\nu \cdot \left(c' + c_+ (\underline{u}_A^1 \cdot \mathbf{1}_{K_C}) + \left\langle \underline{\Phi}, Proj_p \left(\Delta|_{\Omega^\times \setminus \mathcal{J}^\times} u_A^1 \right) \right\rangle_\Omega \right) + \\ & \left. \frac{1}{\Delta t} \tilde{u}_S^0 - \left\langle \underline{\Phi}, Proj_p \left(\frac{1}{\Delta t} \cdot (\underline{u}_A^1 - \underline{u}_A^0) - \int_{t^0}^{t^1} \delta_{\mathcal{J}^\times} \cdot \frac{-s}{\sqrt{1+s^2}} \llbracket \underline{u}_A \rrbracket dt \right) \right\rangle_\Omega \right) \end{aligned} \quad (6.15)$$

Note that, the matrix of the linear system to solve, in Eq. 6.15 is not necessarily symmetric; to get a symmetric system, Eq. 6.15 is multiplied by \mathcal{M}_ν^{-1} . Afterwards, patching may be applied and the term $Proj_p \left(\Delta|_{\Omega^\times \setminus \mathcal{J}^\times} u_A^1 \right)$ may be approximated by the same method that was used for the Poisson equation, see §142. This finally gives Eq. 6.12. End of rationale.

§158: Remark - Simplified implicit Euler scheme for a scalar Heat equation with additive jump condition: If the time-dependent Ansatz function of §157 can be written as

$$u_A(t, \mathbf{x}) = \begin{cases} u_1(\mathbf{x}) & \text{if } \mathbf{x} \in \mathfrak{A}(t) \\ u_2(\mathbf{x}) & \text{if } \mathbf{x} \in \mathfrak{B}(t) \end{cases}$$

with *time-independent* functions u_1 and u_2 . Then Eq. 6.12 simplifies to

$$\begin{aligned} \left(\frac{1}{\Delta t} \cdot \mathcal{M}_v^{-1} - \mathcal{M}_\Delta \right) \cdot \tilde{u}_+^1 &= c' + c_+(\underline{u}_A^1 \cdot \mathbf{1}_{K_C}) \\ &+ \left\langle \underline{\Phi}, \text{Proj}_p \left(\mathbf{L}_{rst}(\underline{u}_A^1 \cdot \mathbf{1}_{K_C}) \right) \right\rangle_\Omega + \mathcal{M}_v^{-1} \cdot \left(\frac{1}{\Delta t} \tilde{u}_+^0 \right). \end{aligned} \quad (6.16)$$

Rationale: Because $\frac{\partial}{\partial t} \Big|_{\Omega^\times \setminus \mathfrak{J}^\times} u_A = 0$, one gets $\frac{\mathbf{1}_{K_C}}{\Delta t} \cdot (\underline{u}_A^1 - \underline{u}_A^0) - \int_{t^0}^{t^1} \delta_{\mathfrak{J}^\times} \cdot \frac{-s}{\sqrt{1+s^2}} \llbracket \underline{u}_A \rrbracket dt = \int_{t^0}^{t^1} \frac{\partial}{\partial t} \Big|_{\Omega^\times \setminus \mathfrak{J}^\times} u_A dt = 0$.

§159: Example - Scalar Heat equation with additive jump condition: Consider the notation of §157 and §158. Given is a domain $\Omega = (-1, 1)^2$ discretized by a Cartesian equidistant grid with 64×64 cells. The polynomial degree of the DG interpolation is 2 and there are no common modes for both phases, which are explicitly given as

$$\mathfrak{A}(t) = \{\mathbf{x} \in \Omega; |\mathbf{x}| < 0.25 + t\}, \mathfrak{J}(t) = \partial\mathfrak{A}(t), \text{ and } \mathfrak{B}(t) = \Omega \setminus (\mathfrak{A}(t) \cup \mathfrak{J}(t)).$$

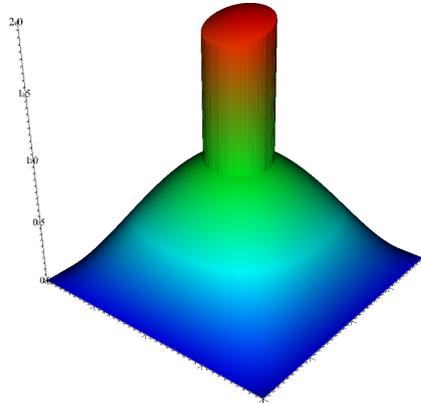
The Ansatz function is explicitly given as

$$u_A(t, \mathbf{x}) = \begin{cases} 1 + 0.3 \cdot x & \text{if } \mathbf{x} \in \mathfrak{A}(t) \\ 0 & \text{if } \mathbf{x} \in \mathfrak{B}(t) \end{cases}.$$

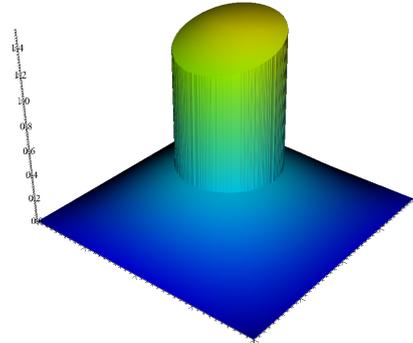
On $\partial\Omega$, homogeneous Dirichlet boundary conditions are assumed; the initial value is given as

$$u^0(\mathbf{x}) = (1 - x^2) \cdot (1 - y^2) + u_A(0, \mathbf{x}).$$

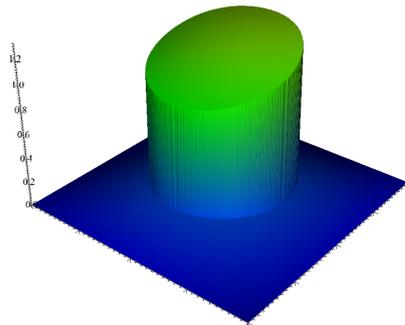
Because of the structure of the Ansatz function, the discretization presented in §158 can be used. Numerical results are shown in figure 6.7.



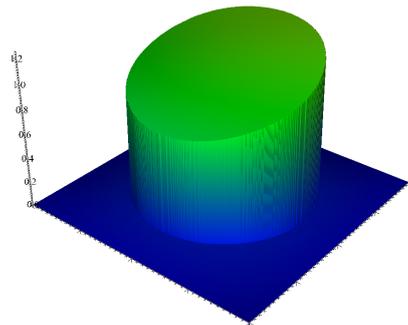
timestep 0, $t = 0$



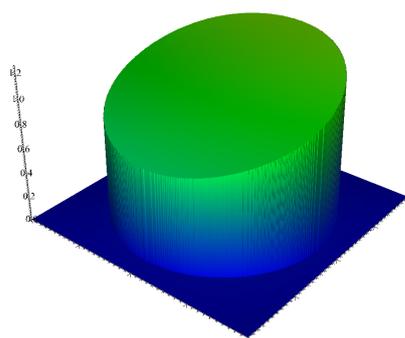
timestep 8, $t = 0.15$



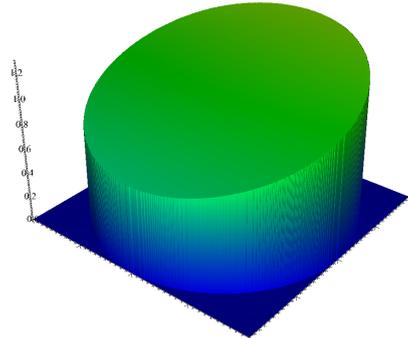
timestep 16, $t = 0.3$



timestep 24, $t = 0.45$



timestep 32, $t = 0.6$



timestep 40, $t = 0.75$

Figure 6.7: Scalar heat equation with time-dependent jump.

6.6 Summary and Outlook

Up to this point, schemes for the solution of singular linear, scalar problems like the Poisson (§141) and the Heat equation (§157) have been developed. It should be mentioned that for those problems only affine jump conditions in the form $[[u]] = g$ have been investigated. By such jump conditions, it is clear that the solution is in an affine linear manifold within the XDG space, i.e. $\underline{u} \in \underline{u}_A + DG_p(\mathcal{R})$, for an Ansatz function \underline{u}_A with $[[\underline{u}_A]] \approx g$.

More complicated jump conditions would be affine-linear ones like e.g.

$$[[\varpi \cdot u]] = g \text{ with } [[\varpi]] \neq 0 \text{ on } \mathcal{J}$$

or, even more complicated, general nonlinear jump conditions:

$$Y(\mathbf{x}, u_{\mathcal{A}}, u_{\mathcal{B}}) = 0 \text{ on } \mathcal{J}$$

for a suitable nonlinear function Y . First experiments show that such Jump conditions could be approximated in an iterative procedure as follows. Given is a problem which formally reads as

$$\begin{cases} Op u = f & \text{in } \Omega \setminus \mathcal{J} \\ \beta \cdot u_{\mathcal{B}} + \alpha \cdot u_{\mathcal{A}} = \gamma & \text{on } \mathcal{J} \end{cases} \quad (6.17)$$

with a linear operator Op . This problem may be substituted by a *sequence* of problems with affine jump conditions

$$\begin{cases} Op v^n = f & \text{in } \Omega \setminus \mathcal{J} \\ [[v^n]] = c^n & \text{on } \mathcal{J} \end{cases} \quad n = 1, 2, 3, \dots$$

which can be solved in the way that we propose for the Poisson equation (see section 6.3) and which converges to the solution of the original problem (Eq. 6.17), i.e.

$$\begin{cases} v^n \rightarrow u & \text{for } n \rightarrow \infty. \\ \beta \cdot v_{\mathcal{B}}^n + \alpha \cdot v_{\mathcal{A}}^n \rightarrow \gamma \end{cases}$$

Here, the relation between the real jump condition and the c^n is

$$c_n = \frac{-\gamma + v_{\mathcal{A}}^{n-1} \cdot (\beta + \alpha) \cdot (1 - \vartheta) + v_{\mathcal{B}}^{n-1} \cdot (\beta + \alpha) \cdot \vartheta}{-\beta \cdot (1 - \vartheta) \cdot \alpha \vartheta}$$

with a scalar $\vartheta \in (0, 1)$. The basic idea behind that formula is the assumption that

$$\begin{aligned} v_{\mathcal{A}}^n &\approx \vartheta \cdot v_{\mathcal{A}}^{n-1} + (1 - \vartheta) \cdot v_{\mathcal{B}}^{n-1} - \vartheta \cdot c^n \\ v_{\mathcal{B}}^n &\approx \vartheta \cdot v_{\mathcal{A}}^{n-1} + (1 - \vartheta) \cdot v_{\mathcal{B}}^{n-1} + (1 - \vartheta) \cdot c^n \end{aligned} \quad \text{on } \mathcal{J}.$$

As this is still work-in-progress, it will not be discussed further.

7 Conclusion, final remarks and outlook

Single-Phase solvers. In the present implementation status of BoSSS, it is possible to perform direct numerical simulation of single-phase - setups (i.e. constant density and viscosity) with up to approximately 10^6 degrees-of-freedom per variable, as it was demonstrated in the wall-mounted-cube configuration (§106). Currently, in these simulations the Poisson solver, used in the projection step (aka. ‘pressure-correction’), is dominating the runtime of the algorithm. The only option to overcome this limitation in BoSSS is currently the use of GPU’s and, but only for small systems the use of the direct solver PARDISO (Schenk et al. 2000, Schenk 2004, Schenk & Gärtner 2006).

Within this work, the classical Projection method was used. In (Klein 2011), the SIMPLE algorithm (Patankar & Spalding 1972) was implemented in DG. In order to perform large timesteps, or to compute steady-state solutions, these methods perform a linearisation of the nonlinear transport operator of the Navier-Stokes equation in each timestep and each iteration. Actually, the linearisation, i.e. the matrix assembly in BoSSS is not optimized for runtime, because originally it was considered that it is only performed once, at the initialisation of an application. For efficient SIMPLE algorithms, the matrix assembly should be further optimized.

All the simulations in this manuscript – and also those done by co-workers – have been executed on a low number of CPU cores, most of the time located on one compute node. It should be a future goal to expand the BoSSS code into regions that can really be considered as High-Performance-Computing (HPC). There, the usage of GPU’s will be of major importance, since this technology seems to be very suitable for DG because the number of arithmetic operations is high in comparison to the number of operands.

GPU’s. We implemented and tested GPU-based sparse solvers which also work on GPU clusters, i.e. GPU’s which are distributed over more than one compute node. These show a speedup factor, in comparison to the CPU, in the range of 5 to 20. In fact, the GPU’s are only used in the

so-called ‘accelerator-pattern’: a sub-algorithm, which usually consumes a big share of the overall runtime, is put onto an accelerator device. While the accelerator is working, the CPU is in idle mode. It is obvious that this approach is limited by Amdahl’s law¹. It also seems to be sub-optimal to have always one of the two, CPU or GPU in idle mode. For DG it would look attractive to compute e.g. edge integrals on the CPU, while volume integration is performed by the GPU.

Further problems are introduced by technical realities. In a typical cluster, such as SCOUT², the GPU-to-CPU ratio lies in the range between 2 and 8. A GPU should not be shared among more than one MPI process, since that would degrade performance. The consequence is e.g. on a cluster with 4 GPU’s per node, that only 4 MPI processes can be run per node. However, since these nodes contain between 8 and 32 CPU cores, according to the GPU-to-CPU ratios mentioned above, each MPI process should use more than one CPU. Therefore, multi-threading (‘OpenMP - parallelism’) is necessary, which is currently not available in BoSSS.

Apart from that, it is still unclear whether CUDA or OpenCL will be the technology of the future. A decision between the two will be very difficult, since CUDA is significantly faster than OpenCL (Karimi, Dickson & Hamze 2011), but CUDA is only available on NVIDIA GPU’s and therefore less portable.

Therefore, further enhancement of GPU support in BoSSS should have the following goals:

- multi-threading support for the CPU-parts of BoSSS.
- GPU ports of further parts of BoSSS, which are currently executed on CPU.
- simultaneous use of GPU and CPU

Fortunately, because of the vectorized design pattern that is used in the quadrature kernels, the porting of these kernels to GPU’s will be quite straightforward.

Modelling. Another future direction may be the addition of turbulence models for the Reynolds Averaged Navier-Stokes (RANS) equations, for URANS and for Large Eddy Simulation (LES). In order to integrate LES

¹http://en.wikipedia.org/wiki/Amdahl's_law, 8th of dec. 2011

²<http://csc.uni-frankfurt.de/index.php?id=48&L=2>, 8th of dec. 2011

models into BoSSS, a first investigation has been done in the diploma thesis by Roozbeh Ashoori (Ashoori 2010).

Since turbulence models are typically defined via an artificial turbulent viscosity, which varies in space and time, also here the need for a high-performance matrix assembly is given, as it is the case with the SIMPLE - algorithm.

The XDG method and multiphase flows. In a future project, it is planned to use the methods presented in chapter 6 as building-blocks for an incompressible two-phase Navier Stokes solver. In single-phase settings, as presented in chapter 5, it is commonly accepted that fractional-step - approaches like the Projection method, but also approaches like Pressure correction, SIMPLE or PISO offer better performance than ‘overall’-schemes, which assemble a large nonlinear, differential-algebraic system from the Navier-Stokes equations. From this point of view, it seems promising to extend the ideas of such schemes to multiphase solvers. The challenge seems to be how the jump and kink conditions that can be derived from conservation laws and additional constitutive laws, see e.g. (Wang & Oberlack 2011), can be integrated into the fractional-step schemes. For the Extended Finite Element method (XFEM), see (Moës et al. 1999), this has successfully been done in (Chen et al. 2004). There, modified jump and kink conditions, which imply the original ones, but not vice-versa (!) are used. The mathematical consequences of this – more strict – jump condition are left open. At least, because of the similarities between XFEM and XDG, it should be possible to apply these ideas to XDG.

To conclude, fractional-step-XDG schemes seem to be within reach. Whether the building blocks that were presented here, i.e. the Poisson and Heat equation are sufficient, or if additional ideas are required, has to be worked out in detail.

8 Bibliography

- ALBAHARI, J., ALBAHARI, B. (2010): *C# 4.0 in a Nutshell: The Definitive Reference*. O'Reilly Media.
- ARNOLD, D. N. (1982): An Interior Penalty Finite Element Method with Discontinuous Elements. *SIAM Journal on Numerical Analysis* 19, 4, 742.
- ARNOLD, D. N., BREZZI, F., COCKBURN, B., MARINI, L. D. (2002): Unified Analysis of Discontinuous Galerkin Methods for Elliptic Problems. *SIAM Journal on Numerical Analysis* 39, 5, 1749–1779.
- ASHOORI, R. (2010): *Numerical Solution of the Navier-Stokes Equations Using Discontinuous Galerkin Method*. Master's thesis, TU Darmstadt, Fachgebiet für Strömungsdynamik, Petersenstraße 30, 64287 Darmstadt, Germany.
- BOLZ, J., FARMER, I., GRINSPUN, E., SCHRÖDER, P. (2003): Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM TRANSACTIONS ON GRAPHICS* 22, 917 – 924.
- BULL, J. M., SMITH, L. A., POTTAGE, L., FREEMAN, R. (2001): Benchmarking Java against C and Fortran for scientific applications. In: *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande - JGI '01*, Palo Alto, California, United States, 97–105.
- CHEN, T., MINEV, P. D., NANDAKUMAR, K. (2004): A projection scheme for incompressible multiphase flow using adaptive Eulerian grid. *International Journal for Numerical Methods in Fluids* 45, 1–19.
- COCKBURN, B., KANSCHAT, G., SCHÖTZAU, D. (2005): A Locally Conservative LDG Method for the Incompressible Navier-Stokes Equations. *Mathematics of Computation* 334, 251, 1067–1095.
- COCKBURN, B., KARNIADAKIS, G. E., SHU, C. (2000): The development of Discontinuous Galerkin Methods. In: *Discontinuous Galerkin methods : theory, computation and applications*, B. Cockburn, G. E. Karniadakis, C. Shu, eds., Springer, Berlin, no. 11 in Lecture Notes in Computational Science and Engineering, 3 – 52.

- COCKBURN, B., SHU, C. (1989): TVB Runge-Kutta Local Projection Discontinuous Galerkin Finite Element Method for Conservation Laws II: General Framework. *Mathematics of Computation* 52, 186, 411.
- COCKBURN, B., SHU, C. (1991): The P1-RKDG Method for Two-dimensional Euler Equations of Gas Dynamics. In: *NASA Contractor Report 187542*, ICASE Report 91-32, 12.
- COCKBURN, B., SHU, C. (1998): The Runge-Kutta Discontinuous Galerkin Method for Conservation Laws V. *Journal of Computational Physics* 141, 2, 199–224.
- E, W., LIU, J. (1995): Projection Method I: Convergence and Numerical Boundary Layers. *SIAM Journal on Numerical Analysis* 32, 4, 1017.
- ESSER, P., GRANDE, J., REUSKEN, A. (2010): An extended finite element method applied to levitated droplet problems. *International Journal for Numerical Methods in Engineering* 84, 7, 757–773.
- FALGOUT, R., JONES, J., YANG, U. (2006): The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners. In: *Numerical Solution of Partial Differential Equations on Parallel Computers*, T. J. Barth, M. Griebel, D. E. Keyes, R. M. Nieminen, D. Roose, T. Schlick, A. M. Bruaset, A. Tveito, eds., Springer Berlin Heidelberg, vol. 51 of *Lecture Notes in Computational Science and Engineering*, 267–294. 10.1007/3-540-31619-1_8.
- GIRAULT, V., RIVIÈRE, B., WHEELER, M. F. (2004): A discontinuous Galerkin method with nonoverlapping domain decomposition for the Stokes and Navier-Stokes problems. *Math. Comp* , 53–84.
- GIRAULT, V., RIVIÈRE, B., WHEELER, M. F. (2005): A splitting method using discontinuous Galerkin for the transient incompressible Navier-Stokes equations. *ESAIM: Mathematical Modelling and Numerical Analysis* 39, 6, 1115–1147.
- GOTTLIEB, S., SHU, C. (1998): Total Variation Diminishing Runge-Kutta Schemes. *Mathematics of Computation* 67, 221, 73–85.
- GROSS, J., HESTHAVEN, J. (2006): A level set discontinuous Galerkin method for free surface flows. *Computer Methods in Applied Mechanics and Engineering* 195, 25-28, 3406–3429.
- GROSS, S., REUSKEN, A. (2007): An extended pressure finite element space for two-phase incompressible flows with surface tension. *Journal of Computational Physics* 224, 1, 40–58.

- HESTHAVEN, J. S., WARBURTON, T. (2008): *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Texts in Applied Mathematics , Vol. 54, Springer-Verlag.
- JAVA GRANDE FORUM (1998): Java Grande Forum Report: Making Java Work for High-End Computing.
- KARIMI, K., DICKSON, N., HAMZE, F. (2011): A Performance Comparison of CUDA and OpenCL. Preprint.
- KHRONOS GROUP (2011): OpenCL. <http://www.khronos.org/opencvl>.
- KIM, J., MOIN, P. (1985): Application of a fractional-step method to incompressible Navier-Stokes equations. *Journal of Computational Physics* 59, 2, 308–323.
- KLEIN, B. (2011): *Implementation of the SIMPLE algorithm for solving the steady incompressible Navier-Stokes equations discretized by the discontinuous Galerkin method*. Master’s thesis, TU Darmstadt, Fachgebiet für Strömungsdynamik, Petersenstraße 30, 64287 Darmstadt, Germany.
- KLÖCKNER, A., WARBURTON, T., BRIDGE, J., HESTHAVEN, J. (2009): Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics* 228, 21, 7863–7882.
- KRÜGER, J., WESTERMANN, R. (2003): Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics* 22, 3, 908 – 916.
- LATTNER, C., ADVE, V. (2004): LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, IEEE Computer Society, Washington, DC, USA, CGO ’04, 75–103.
- LEVEQUE, R. (2002): *Finite volume methods for hyperbolic problems*. Cambridge University Press, Cambridge New York.
- LEVEQUE, R. J., OLIGER, J. (1983): Numerical Methods Based on Additive Splittings for Hyperbolic Partial Differential Equations. *Mathematics of Computation* 40, 162, 469–497.
- MARCHANDISE, E., GEUZAIN, P., CHEVAUGEON, N., REMACLE, J.-F. (2007): A stabilized finite element method using a discontinuous level set approach for the computation of bubble dynamics. *J. Comput. Phys.* 225, 949–974.

- MARCHANDISE, E., REMACLE, J.-F. (2006): A stabilized finite element method using a discontinuous level set approach for solving two phase incompressible flows. *Journal of Computational Physics* 219, 2, 780–800.
- MCLACHLAN, R. I., QUISPEL, G. R. W. (2002): Splitting methods. *Acta Numerica* , 341–434.
- MEISTER, A. (2004): *Numerik linearer Gleichungssysteme: Direkte und iterative Verfahren (Springer-Lehrbuch)*. Springer.
- MOËS, N., DOLBOW, J., BELYTSCHKO, T. (1999): A finite element method for crack growth without remeshing. *International Journal for Numerical Methods in Engineering* 46, 131–150.
- NVIDIA CORPORATION (2010): CUDA Programming Guide 3.1. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf.
- NVIDIA CORPORATION (2011): Cuda. <http://developer.nvidia.com/category/zone/cuda-zone>.
- OSHER, S. J., FEDKIW, R. P. (2002): *Level Set Methods and Dynamic Implicit Surfaces*. Springer.
- PATANKAR, S. V., SPALDING, D. B. (1972): A Calculation Procedure for Heat, Mass and Momentum Transfer in Three-Dimensional Parabolic Flows. *Int. J. Heat Mass Transfer* 15, 1787–1972.
- POZRIKIDIS, C. (2001): A Note on the Regularization of the Discrete Poisson–Neumann Problem. *Journal of Computational Physics* 172, 2, 917–923.
- REED, W. H., HILL, T. R. (1973): Triangular mesh methods for the neutron transport equation. In: *National topical meeting on mathematical models and computational techniques for analysis of nuclear systems*, Los Alamos Scientific Lab., N.Mex. (USA), CONF-730414–2; LA-UR–73-479, 23.
- RENARDY, M. (2004): *An introduction to partial differential equations*. Springer, New York, Second edn..
- SCHENK, O. (2002): Two-level dynamic scheduling in PARDISO: Improved scalability on shared memory multiprocessing systems. *Parallel Computing* 28, 2, 187–197.

- SCHENK, O. (2004): Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems* 20, 3, 475–487.
- SCHENK, O., GÄRTNER, K. (2006): On Fast Factorization Pivoting Methods for Sparse Symmetric Indefinite Systems. *Electronic Transactions on Numerical Analysis* 23, 158–179.
- SCHENK, O., GÄRTNER, K., FICHTNER, W. (2000): Efficient Sparse LU Factorization with Left-Right Looking Strategy on Shared Memory Multiprocessors. *BIT Numerical Mathematics* 40, 1, 158–176. 10.1023/A:1022326604210.
- SCHWARTZ, L. (1966): *Théorie des distributions*. Hermann.
- SETHIAN, J. A. (1996): *Level Set Methods: Evolving Interfaces in Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press.
- SETHIAN, J. A. (1999): *Level Set Methods and Fast Marching Methods*. Cambridge University Press.
- SETHIAN, J. A. (2001): Evolution, Implementation, and Application of Level Set and Fast Marching Methods for Advancing Fronts. *Journal of Computational Physics* 169, 2, 503–555.
- SETHIAN, J. A., SMEREKA, P. (2003): Level Set Methods for Fluid Interfaces. *Annual Review of Fluid Mechanics* 35, 341–372.
- SHAHBAZI, K. (2005): An explicit expression for the penalty parameter of the interior penalty method. *Journal of Computational Physics* 205, 401–407.
- SHAHBAZI, K., FISCHER, P., ETHIER, C. (2007): A high-order discontinuous Galerkin method for the unsteady incompressible Navier-Stokes equations. *Journal of Computational Physics* 222, 1, 391–407.
- TRIEBEL, H. (1980): *Höhere Analysis*. Verlag Harri Deutsch.
- VOGELS, W. (2003): Benchmarking the CLI for high performance computing. *IEE Proceedings - Software* 150, 5, 266.
- WAFEL, A. (2009): *Sparse Matrix-Vector Multiplications on Graphics Processors*. Master's thesis, Universität Stuttgart, Institut für parallele und verteilte Systeme, Universitätsstraße 38, 70569 Stuttgart, Germany.

WALTER, W. (1973): *Einführung in die Theorie der Distributionen*. BI Wissenschaftsverlag.

WANG, Y., OBERLACK, M. (2011): A thermodynamic model of multiphase flows with moving interfaces and contact line. *Continuum Mechanics and Thermodynamics* 23, 409–433.

9 Curriculum Vitae

Florian Kummer

Personal Data:

Date of Birth: 14th of July, 1981
Place of Birth: Innsbruck, Austria
Nationality: Austria

Education:

Nov. 2011: Defence of PhD thesis
Dec. 2009 – Dec. 2011: Associated student at the “Graduate School of Computational Engineering”, TU Darmstadt
Sept. 2006 – Dec. 2011 PhD student at Chair of Fluid Dynamics, TU Darmstadt
2000 – 2002, 2003 – 2006: Studies of Technical Mathematics at Leopold-Franzens – University, Innsbruck, Austria; Master thesis: “Operator splitting for parabolic PDE’s”
1994 – 2000: Federal Secondary College of Engineering, specialized in communications engineering
1986 – 1994: Primary school, secondary school

Miscellaneous :

2002 – 2003: military duty service

Employment:

2006 – now: Scientific Staff, Chair of Fluid Dynamics, TU Darmstadt