Implicitly coupled finite volume algorithms

Uroić, Tessa

Doctoral thesis / Disertacija

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Mechanical Engineering and Naval Architecture / Sveučilište u Zagrebu, Fakultet strojarstva i brodogradnje**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:235:593963

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-07-17**

Repository / Repozitorij:

Repository of Faculty of Mechanical Engineering and Naval Architecture University of Zagreb

University of Zagreb
Faculty of Mechanical Engineering and Naval Architecture

Tessa Uroić

Implicitly Coupled Finite Volume Algorithms

DOCTORAL THESIS

Zagreb, 2019.

University of Zagreb
Faculty of Mechanical Engineering and Naval Architecture

Tessa Uroić

Implicitly Coupled Finite Volume Algorithms

DOCTORAL THESIS

Supervisor: prof. Hrvoje Jasak, PhD

Zagreb, 2019.

Sveučilište u Zagrebu
Fakultet strojarstva i brodogradnje

Tessa Uroić

Implicitno spregnuti algoritmi u metodi kontrolnih volumena

DOKTORSKI RAD

Mentor: prof. dr. sc. Hrvoje Jasak

Zagreb, 2019.

Dedicated to friends.

Acknowledgements

I would like to express my sincere gratitude to individuals who contributed to the quality and completion of this thesis.

Foremost, I thank my supervisor prof. Hrvoje Jasak, for sharing his knowledge and expertise in the field of computational fluid dynamics and finite volume method and for patiently reviewing my work. The debugging sessions of SAMG are also greatly appreciated.

I am grateful to Vuko Vukčević, who worked on the first versions of the implicitly coupled solver and has implemented the ILUC preconditioner into `foam-extend`.

Images which make this thesis easier to understand were generated with the help of my dear student Luka Balatinec, who I am indebted to.

There were also many others who (un)intendedly helped me in some way, and I dedicate the past 5 years to them.

Abstract

In this thesis a comparison of pressure–velocity coupling algorithms is conducted with special emphasis on the implicit coupling of momentum and pressure equations in a single linear system as an improvement on the conventional segregated methods. The research is done in the scope of the finite volume method, and carried out in `foam-extend`, a community driven fork of the open source computational fluid dynamics software `OpenFOAM`. We focus on the equations for steady–state, incompressible, single–phase turbulent flow. To counteract the zero block on the diagonal of the system, the pressure equation is derived as a Schur complement. A description of the `OpenFOAM` matrix format is given as well as the structure of the finite volume matrices which arise from the computational mesh connectivity. Contributions from finite volume equation discretisation to each term of the implicitly coupled block–matrix are illustrated. The computational effort is directed mostly to solving the pressure Poisson equation, thus it is very important to employ an efficient solver for elliptic equations, which is also effective for the hyperbolic momentum equation. An overview of linear solvers is given: fixed–point methods, algebraic multigrid and some versions of the Krylov subspace solvers are analysed in the context of Finite Volume Method. Two methods for constructing coarse matrices in algebraic multigrid are compared: additive correction method, which is the usual choice for the implicitly coupled pressure–velocity system in recent literature, and the newly implemented selection algebraic multigrid. Incomplete lower–upper factorisation based on Crout's algorithm is used as an error smoother with the algebraic multigrid method. Parallelisation issues regarding the selection algebraic multigrid are laid out, with additional comments on the possible parallelisation strategies. The performance of both segregated and implicitly coupled pressure–velocity solver is compared for multiple complex test cases including external aerodynamics cases (Formula 1 front wing, bluff body with a diffuser, submarine) and internal flow cases (cooling of an engine jacket, centrifugal pump, Francis turbine) both on structured and unstructured meshes. A comparison of convergence against computation

time and number of non–linear iterations is given, both in terms of field residuals as well as integral values. The influence of the linear solver on the convergence of implicitly coupled pressure–velocity solver is investigated, as well as the impact of different settings of selection algebraic multigrid on convergence rate and computation time.

Keywords:
Implicit coupling, Block matrix, Pressure–velocity coupling, Algebraic multigrid, Conjugate gradient method

Prošireni sažetak

I Sprega jednadžbi brzine i tlaka

Stacionarno, nestlačivo, jednofazno i turbulentno strujanje fluida opisano je jednadžbom očuvanja količine gibanja koja sadrži nelinearni konvekcijski član, difuzijski član te površinske sile u obliku gradijenta tlaka, i jednadžbom kontinuiteta, koja nameće uvjet konzervativnosti polja brzine u nestlačivom strujanju. Budući da ne postoji analitička metoda za rješavanje ovog sustava jednadžbi, koriste se iterativni algoritmi. U opisanom sustavu ne postoji eksplicitna jednadžba koja opisuje polje tlaka, što onemogućava korištenje klasičnih iterativnih metoda u kojima se neizbježno provodi dijeljenje dijagonalnim elementom matrice. Stoga se iz jednadžbe količine gibanja izvodi izraz za polje brzine te uvrštava u jednadžbu kontinuiteta kako bi se dobila jednadžba iz koje se može izračunati polje tlaka. Dobivena jednadžba tlaka ima oblik Poissonove jednadžbe, eliptičnog je karaktera te predstavlja najzahtjevniji dio rješenja linearne sprege brzine i tlaka.

Zbog ograničenih računalnih resursa, 70.–tih i 80.–tih godina prošlog stoljeća razvijaju se algoritmi koji jednadžbu količine gibanja i tlaka rješavaju u odvojenim linearnim sustavima, uvrštavajući prethodno izračunate vrijednosti druge varijable. Pristup razdvajanja jednadžbi linearnog sustava prisutan je do danas, a algoritmi SIMPLE i PISO te njihove inačice još se uvijek koriste u modernim kodovima za računalnu dinamiku fluida.

U ovom je radu predstavljen postupak implicitne sprege jednadžbe količine gibanja i tlaka u jedinstveni linearni sustav, koji je implementiran u program otvorenog koda za računalnu dinamiku fluida `OpenFOAM`. Matrica implicitno spregnutog sustava ima 4 puta veći broj redaka i stupaca u odnosu na matrice u SIMPLE algoritmu, dok su nepoznanice u vektoru poredane s obzirom na indeks kontrolnog volumena za koji se računaju. Zbog toga su matrični elementi koji odgovaraju jednadžbama za pojedini kontrolni volumen, i sami matrice dimenzije 4×4, a matrica implicitno spregnutog sustava naziva se blok–matricom. Budući da se u jednom sustavu javljaju jednadžbe različitih karaktera, hiperbolična jed-

nadžba količine gibanja te eliptična jednadžba tlaka, izbor algoritma za rješavanje linearnog sustava dodatno je otežan.

II Linearni algoritmi

Osnovni linearni algoritmi za rješavanje linearnih sustava su iteracije fiksne točke, tj. Jacobijeva i Gauss–Seidelova metoda. Na temelju spektralne analize matrice iteracije spomenutih metoda, uočeno je da konvergencija metoda usporava nakon što se eliminiraju greške čije komponente odgovaraju najvećim svojstvenim vrijednostima (visoke frekvencije), tj. preostala greška smatra se glatkom. Grešku je moguće ponovno učiniti oscilatornom zadržavajući samo komponente koje odgovaraju malim svojstvenim vrijednostima, što je uloga multigrid algoritma.

U ovom se radu koriste algebarske multigrid metode, koje ne trebaju informacije iz početne (fine) proračunske mreže, već za konstrukciju grubih razina koriste matrične elemente. Predstavljene su dvije metode: *additive correction* (dodane korekcije – AAMG) i *selection* (selekcijska – SAMG) metoda. AAMG metoda konstruira grubu matricu zbrajanjem elemenata fine matrice, što se opravdava konzervativnošću jednadžbi količine gibanja i kontinuiteta za kontrolni volumen. Kada bismo promatrali proračunsku mrežu, mreža grube razine AAMG metode dobila bi se grupiranjem volumena fine mreže u jedan veći volumen. Dodana korekcija odnosi se na član koji se dodaje izračunatom rješenju kako bi se kompenzirala pogreška koja se javlja zbog preslikavanja izračunatog rješenja s grubog na fini nivo. U SAMG metodi, matrica na gruboj razini konstruira se koristeći Galerkinov varijacijski princip, odnosno, u fiktivnoj proračunskoj mreži grube razine preostaje određeni broj odabranih kontrolnih volumena s fine razine. Budući da se u SAMG metodi koristi linearna interpolacija rješenja izračunatog na gruboj razini u linearni sustav na finoj razini, ostvaruje se bolja konvergencija rješenja u usporedbi s AAMG metodom. Multigrid metode optimalne su upravo za rješenje Poissonovog tipa jednadžbi jer osiguravaju efikasan prijenos lokalnih informacija na globalnoj razini sustava.

Poseban izazov predstavlja paralelizacija SAMG algoritma, budući da je proces odabira jednadžbi za grubu razinu u potpunosti sekvencijalan. Kako bi se očuvala efikasnost algoritma, uvode se sljedeća ograničenja:

- odabir jednadžbi grube razine provodi se nezavisno na svakom procesoru,
- nije moguća interpolacija korekcije sa susjednog procesora,
- matrični elementi koji se nalaze na granici procesora na gruboj razini računaju se također Galerkinovim principom pomoću filtrirane matrice interpolacije i restrikcije.

Budući da se u matricama grubih razina pojavljuju novi elementi, raste popunjenost matrice, a ujedno i broj elemenata na procesorskoj granici.

Kao dopunu kojom se efikasno rješava i hiperbolična jednadžba količine gibanja, odabrali smo algoritme koji komponente rješenja konstruiraju linearnom kombinacijom vektora iz Krylovljevih potprostora. U radu je detaljno opisan izvod metode konjugiranih gradijenata (CG), te inačica za nesimetrične matrice stabiliziranih bikonjugiranih gradijenata (BiCGStab). BiCGStab korišten je kao algoritam za efikasno rješavanje sustava fine razine multigrid algoritma, dok multigrid ciklus služi kao prekondicioniranje kojim se dobiva rješenje Poissonove jednadžbe tlaka. Budući da klasične iterativne metode fiksne točke nisu dale zadovoljavajuću konvergenciju za izglađivanje greške (ili su čak divergirale), implementirali smo nepotpunu LU faktorizaciju matrice temeljenu na Croutovom algoritmu (ILUC). Konvergencija ILU algoritma ovisi o stukturi matrice te je poželjno zadržati vrpčastu formu.

III Rezultati i zaključak

Konvergenciju SIMPLE i implicitno spregnutog algoritma usporedili smo na nizu slučajeva s kompleksnim turbulentnim strujanjem: opstrujavanje krila bolida Formule 1, analiza strujanja u pojednostavljenom difuzoru bolida, opstrujavanje modela podmornice BB2 na tri gustoće mreže, te slučajeve sa zonama rotacije u centrifugalnoj pumpi i Francisovoj turbini. Simulacije su provedene na strukturiranim i nestrukturiranim mrežama, veličine od 2 do 14 milijuna kontrolnih volumena. Na temelju rezultata simulacija, doneseni su sljedeći zaključci o algoritmima:

- implicitno spregnuti algoritam u usporedbi sa SIMPLE algoritmom konvergira s manjim oscilacijama reziduala i integralnih veličina,

- budući da se u implicitno spregnutom algoritmu ne podrelaksira polje tlaka, a jednadžba količine gibanja se podrelaksira minimalno, ostvaruje se brža konvergencija u pogledu broja iteracija, te ukupnog računalnog vremena,

- konvergencija implicitno spregnutog algoritma vrlo malo ili uopće ne ovisi o gustoći i tipu proračunske mreže, što nije slučaj kod SIMPLE algoritma,

- za proračune s implicitno spregnutim algoritmom potrebna je značajno veća količina radne memorije nego za SIMPLE algoritam, budući da je matrica sustava 16 puta veća.

Analizirali smo postavke i konvergenciju SAMG algoritma za implicitno spregnuti sustav te je primijećeno sljedeće:

- korištenje SAMG algoritma za rješenje linearnog implicitno spregnutog sustava jednadžbi količine gibanja i kontinuiteta daje manje oscilatornu konvergenciju reziduala u usporedbi s AAMG i BiCGStab algoritmima, no za jednu iteraciju SAMG–a potreban je veći broj operacija,

- SAMG često postiže teoretsku brzinu konvergencije (smanjuje rezidual jedan red veličine po iteraciji), ali na početku simulacije, dok se rješenje još uvijek značajno mijenja, potreban je veći broj iteracija kako bi se ostvarila željena konvergencija,

- najbolja konvergencija postiže se korištenjem Poissonove jednadžbe tlaka za računanje težinskih faktora interpolacije,

- odabir broja jednadžbi u matrici najgrublje razine, kao i odabir multigrid ciklusa utječu ne samo na konvergenciju nego i ukupno vrijeme proračuna: kako bi se smanjilo potrebno vrijeme, koristi se V–ciklus i veći broj jednadžbi grube razine,

- konvergencija ILUC algoritma za izglađivanje greške ovisi o strukturi matrice te se preporučuje i na grubim razinama očuvati vrpčastu strukturu postignutu optimalnim pobrojavanjem kontrolnih volumena u mreži,

- odabir jednadžbi koje će se rješavati na gruboj razini SAMG–a, ovisi o tipu kontrolnih volumena proračunske mreže: rješenje na mrežama s anizotropnim kontrolnim volumenima mogu lošije konvergirati od onih na mrežama s uniformnim volumenima,
- paralelizacija SAMG–a u kojoj se proces interpolacije ograničava na lokalnu procesorsku jezgru ne umanjuje značajno stopu konvergencije linearnog i nelinearnog algoritma.

Na temelju dosadašnjeg istraživanja, predlažemo sljedeće korake kao dopunu ili potencijalni nastavak istraživanja:

- izmjeriti performanse implicitno spregnutog algoritma polja tlaka i brzine na superračunalu s velikim brojem jezgara,
- proučiti moguće aproksimacije matrice konvekcije i difuzije čiji se inverz koristi kao difuzijski koeficijent u Poissonovoj jednadžbi tlaka,
- implementirati implicitne rubne uvjete (npr. rubni uvjet sa zadanim totalnim tlakom),
- proširiti implicitno spregnuti algoritam i za stlačiva strujanja, za čiju je stabilnost nužno implicitno tretirati rubne uvjete,
- razmotriti raspodjelu poslova između pojedinih procesora u paralelnim simulacijama sa SAMG algoritmom,
- implementirati *FLEX* multigrid ciklus, što može poboljšati konvergenciju linearnog algoritma (slično W–ciklusu): FLEX ciklus ima svojstvo samoregulacije, tj. rješenje se dinamički prebacuje s finog na grube razine i obrnuto u ovisnosti o vrijednosti reziduala,
- implementirati ILU algoritam s pivotiranjem kako bi se spriječila divergencija algoritma u slučaju matričnih elemenata različitih redova veličine.

Ključne riječi
Implicitno spregnuti algoritmi, blok–matrica, sprega polja brzine i tlaka, algebarski multigrid, metoda konjugiranih gradijenata

Contents

List of Figures

List of Tables

Nomenclature

A_f	face surface area	m^2
L_i	cluster i in additive correction multigrid	-
N	number of cells in computational mesh, dimension of the coefficient matrix	-
$\hat{\mathbf{u}}$	pseudo–velocity	m/s
$\mathbb{C}$	set of coarse equations in multigrid	-
$\mathbb{F}$	set of fine equations in multigrid	-
$\mathbb{N}$	set of neighbouring equations in multigrid	-
$\mathbf{A_{ij}}$	block matrix element	-
$\mathbf{A_u}$	convection–diffusion matrix	-
$\mathbf{D}$	diagonal matrix	-
$\mathbf{E_u}$	matrix with off–diagonal elements of the convection–diffusion matrix	-
$\mathbf{G}$	discretised gradient operator	-
$\mathbf{G}^T$	discretised divergence operator	-
$\mathbf{G}_{\mathrm{R}}$	Givens rotations matrix	-
$\mathbf{H}(\mathbf{u})$	matrix which contains only the off–diagonal part of the momentum matrix multiplied by the values of velocity	-
$\mathbf{H}$	Hessenberg matrix	-
$\mathbf{I}$	identity matrix	-
$\mathbf{L}$	lower triangular matrix	-
$\mathbf{M}$	multigrid matrix	-

$\mathbf{P_A}$	preconditioning matrix	-
$\mathbf{P}$	prolongation matrix in multigrid	-
$\mathbf{Q}$	unitary matrix	-
$\mathbf{R}$	restriction matrix in multigrid	-
$\mathbf{S}$	smoothing matrix in multigrid	-
$\mathbf{U}$	upper triangular matrix	-
$\mathbf{W}$	Hodge dual of angular velocity	-
$\mathbf{b}$	general right hand side vector	
$\mathbf{d}$	distance vector between two cell centres	m
$\mathbf{e}$	error	-
$\mathbf{k}$	non–orthogonal vector component in orthogonal correction	
$\mathbf{q}$	columns of the unitary matrix	-
$\mathbf{r_b}$	right hand side vector	-
$\mathbf{r}$	residual vector	-
$\mathbf{r}_\mathrm{d}$	distance from the axis of rotation	m
$\mathbf{s}_f$	surface normal face area vector	m^2
$\mathbf{u_R}$	relative velocity	m/s
$\mathbf{u}$	velocity	m/s
$\mathbf{u}*$	intermediate velocity field in SIMPLE	m/s
$\mathbf{v}$	eigenvector	-
$\mathbf{x}$	general unknown vector	
a_{ii}	diagonal matrix element	-

a_{ij}	off–diagonal matrix element	-
b	center of the boundary face	-
f	cell face centre	-
$f(\mathbf{x})$	quadratic function of the unknown vector $\mathbf{x}$	-
g_b	specified gradient value	-
k	turbulent kinetic energy	m^2/s^2
p	kinematic pressure	m^2/s^2
$p*$	intermediate pressure field in SIMPLE	m^2/s^2
w	weighting factor in selection multigrid	-
w_{CD}	interpolation weight Gauss linear scheme	-

Calligraphy letters

$\mathcal{D}$	Krylov subspace	-

Greek letters

α	length of the step in the direction of the residual in steepest descent	-
$\alpha_{\mathbf{u}}$	underrelaxation factor for the momentum equation	-
α_{MG}	strength of connection factor in agglomeration multigrid	-
β	projection operator in conjugate gradients	-
β_{MG}	strength of connection factor in selection multigrid	-
ϵ	rate of dissipation of turbulent kinetic energy	m^2/s^3
γ	scaling factor in selection multigrid	-
κ	matrix condition number	-
λ	eigenvalue	-

$\boldsymbol{\Delta}$	orthogonal vector component in orthogonal correction	
ν	kinematic viscosity	m^2/s
Ω	angular velocity	rad/s
ω	specific rate of dissipation of turbulent kinetic energy	s^{-1}
ω_{BCGS}	smoothing coefficient in biconjugate gradient stabilised	-
Φ	volumetric flux	m^3/s
ϕ	general scalar quantity	-
ϕ_{BCGS}	polynomial in biconjugate gradient stabilised	-
π	polynomial in biconjugate gradient stabilised	-
ψ	smoothing function in biconjugate gradient stabilised	-
ξ	residual normalisation factor	-
ζ	length of component of $\mathbf{e}$	-

Superscripts

(k)	current time step or iteration	-
T	transpose operation	-
C	value on the coarse level of multigrid	-
F	value on the fine level of multigrid	-
S	strong connection in selection algebraic multigrid	-
W	weak connection in selection algebraic multigrid	-

Subscripts

f	value which belongs to a face	-
i	value which belongs to the cell centre of cell i	-
j	value which belongs to the cell centre of neighbouring cell j	-

1. Introduction

This thesis is motivated by and aimed at the computational fluid dynamics (CFD) community, especially those using the `OpenFOAM` [1] CFD library, among whom treatment of non–linear and linear solvers as black–box tools is broadly spread. Many (more or less) experienced users and even developers tend to copy–paste the settings of the solution algorithms without taking into consideration the underlying physics, discrete equations and structure of the linear system. Since recent development in the scope of the finite volume method is dedicated to implicit coupling of equation sets, we have derived and implemented an implicitly coupled pressure–velocity system, which is the basis for the solution of the majority of problems in CFD. We have analysed the structure of the linearised pressure–velocity system and identified the appropriate solvers for the solution of the linear system. The mathematical background and nuances of each type of linear solvers are illustrated from an engineer's point of view, in hope that it will be useful to others when their simulations end with a floating point exception.

1.1. Previous and Related Studies

Since the 1960s, following the development of modern computers, there has been a continuous effort to develop a suitable, accurate and efficient solver for the Navier–Stokes equations in the scope of computational fluid dynamics and the finite volume method (FVM). The first step is the transformation of these non–linear partial differential equations into a linear system which will represent the continuous solution sufficiently well in a discrete manner. Even today, each discretisation technique - finite differences, finite elements, finite volumes - has its dedicated users, depending on the underlying physics of the problem they are trying to solve. Due to its conservation properties, the most popular method for the solution of turbulent fluid flow equations in practical applications is the FVM, which will be used in the scope of this thesis. We focus on the solution techniques for the steady–state, *incompressible*, single–phase and turbulent flow

equations. Here, incompressibility actually generates a constraint on the velocity field. That is, velocity field must be divergence free or - what comes in, must come out. The pressure field appears only in the momentum equation, under the gradient operator. This means that there doesn't exist a unique solution for the pressure field, since there is an infinite number of pressure values which could result in a certain pressure gradient. The issue is remedied by assigning a value of pressure somewhere in the domain, whether it is a single point or a set of points, e.g. on the outlet boundary.

The incompressibility constraint has led researchers to a natural choice of spatial discretisation - using staggered meshes, where the pressure values are stored in cell centres, while velocity components are stored at face centres. However, in `OpenFOAM`, collocated meshes are used, where all variables are calculated and stored at cell centres. This will cause certain issues which will be discussed in Chapter 2..

The solution of discretised Navier–Stokes equations using a direct linear solver such as Gaussian elimination [2] is formidable, since the application on sparse coefficient matrices would require a high amount of storage. Since there does not exist an equation for pressure, zero elements appear on the diagonal of the coefficient matrix. This is a serious obstacle for common iterative solution techniques in FVM CFD, since most of them include a division by the diagonal element. These issues motivated researchers to invent an equation for pressure and the limits of contemporary computers prompted the segregated solution techniques. Due to low memory capacities, the momentum and (invented) pressure equation were decoupled and solved sequentially in separate linear systems, using old values of the other variable, even though the coupling of velocity and pressure is linear. The most recognisable segregated solver is the SIMPLE algorithm (Semi IMplicit Pressure Linked Equations) [3] by Patankar and Spalding. The pressure equation is derived by combining the momentum and continuity equation and yielded a Poisson type equation for pressure. The Poisson equation for pressure is elliptic in nature, i.e. the value at each point in space is affected by values at other points. This effect was (unintentionally) beautifully described by Dostoevsky in his Brothers Karamazov: "For all is like an ocean, all flows and connects; touch it in one place and it echoes at the other end of the world". If only von

Neumann boundary conditions are used for pressure, the resulting coefficient matrix is diagonally equal, which causes convergence problems for iterative methods. In this case as well, assigning a value of pressure in a single point relieves the issue. Since the pressure field is used as a correction to achieve a divergence free velocity field, the pressure field requires substantial underrelaxation, i.e. using a certain fraction of old values to stabilise the solution in the following iteration. Other methods emerged from SIMPLE by adding corrections to certain assumptions which were made in the derivation of the algorithm. For example Patankar [4] proposed SIMPLER (SIMPLE Revised), where he tried to remedy the fact that a good velocity field (which is easier to guess) is often ruined by a bad pressure correction, and the rest of the iteration is trying to fix that. That is, he devised a procedure where values of the velocity are used to calculate the pressure field and a solution of an additional Poisson equation is required, which diminishes the overall efficiency. Another attempt was made by van Doormaal and Raithby [5] with their SIMPLEC (SIMPLE Consistent) algorithm, where they included a term containing the sum of off–diagonal matrix elements in the momentum equation, which was neglected in SIMPLE. It yielded a solver very similar to SIMPLE, but with no need for underrelaxation of the pressure field. Issa presented the PISO (Pressure Implicit by Splitting of Operators) [6] method where pressure was corrected twice in each non–linear iteration, which required the solution of a Poisson type equation twice.

Some researchers tried to avoid solving the pressure Poisson equation since it caused a great deal of difficulty in the solution procedure in terms of computational effort. Some authors even reported that it took up to 90% of overall computational time [7]. Raithby and Schneider [8] proposed a method which did not require a solution of the Poisson equation but it performed integration of the momentum equation along two paths, starting from a reference point where the value of pressure was set. Mazhar and Raithby [9] updated the method by adding additional integration paths. A different approach can be taken by stepping back from the primitive flow variables and solving the flow field using the velocity–vorticity [10] formulation or stream function–vorticity formulation. In [11], the flow field was solved using the mass fluxes through cell faces, rather than cell centred velocity values. The algorithm outperformed SIMPLE in terms of

the number of iterations and the convergence did not deteriorate with increase of mesh density. It was proposed that the method should be improved by investigating the properties of the corresponding linear system.

There were also attempts to retain the implicit linear coupling of pressure and velocity, instead of solving them as two separate systems. Zedan and Schneider [12, 13] tried an implicit solution for two–dimensional flows by arranging the momentum and pressure equation into a single linear system, but it underperformed compared to segregated systems. At the beginning of the new millennium, Mazhar [14, 15] using the natural form of two–dimensional equations, i.e. without inventing the pressure equation, and rearranged the pressure–velocity system. He ordered the unknowns by writing the two momentum equations as two consecutive blocks, followed by the continuity equation. Even though there was a zero matrix on the diagonal, a special incomplete factorisation was used to create nonzero elements on the diagonal. He reported a significant speedup of convergence.

The implicitly coupled algorithms have gained popularity over the last decade: there have been numerous publications by multiple authors. Darwish et al. [16, 17] have adopted a derivation of equations similar to SIMPLE algorithm, which they solve implicitly and have reported the performance for different applications: turbomachinery [18, 19], compressible flows [20, 21] and two–phase flows [22]. A similar solution technique was employed by Uroić et al. [23] for incompressible and compressible flow, as well as Chen [24] and Falk and Schäfer [25]. The procedure was even extended to non–Newtonian fluids: Fernandes et al. [26] implemented a block coupled algorithm for the solution of laminar, incompressible viscoelastic flow in `OpenFOAM`.

During our research, we have confirmed the findings reported in literature that the bottleneck of implicitly coupled pressure–velocity solvers is the efficient solution of the pressure Poisson equation, which counteracts a saddle point problem, i.e. the zero matrix on the diagonal. We have already discussed the global nature of the elliptic pressure equation and the mutual dependence of all points in the domain. Thus, a linear solver which could instantly deliver the information about variations of pressure in a certain point to other points would be highly efficient in solving the equation. Swift propagation of information is a characteristic of

the multigrid method. Since we are dealing with industrial applications, where computational meshes are often suboptimal in terms of structure and quality, the focus is on the algebraic multigrid method (AMG), which operates only on the linear algebraic equations, i.e. coefficient matrix and does not need any information about or from the mesh. It relies on the strategy from geometric multigrid method, where coarse level meshes can be selected using the Galerkin operator [27] and it can be constructed purely algebraically. The implementation in this thesis follows the work of Stüben and Ruge [28, 29] who described what is now known as the classic algebraic multigrid algorithm in the finite element method. The coarse levels are constructed by selecting equations from the previous fine level, while the transfer of residual and coarse level correction are done with restriction and prolongation matrices. The algorithm was extended for implicitly coupled systems by Clees and Stüben [30, 31] for a class of reaction–diffusion and drift–diffusion equations.

Other options for the construction of coarse levels are the smoothed aggregation algebraic multigrid techniques [32] by Vanka and the additive correction algebraic multigrid [33] by Hutchinson and Raithby. Both algorithms create coarse levels by grouping the cells, rather than selecting them. In smoothed aggregation a tentative interpolation operator is first created by assuming that the correction from coarse levels will be applied as an injection. However, a Jacobi smoothing sweep is performed on the interpolation operator to obtain a smoothed interpolation. Coarse level matrix is creted by Galerkin projection, as in the classic method. In the scope of this thesis, we did not investigate the smoothed aggregation multigrid. The additive correction multigrid does not exploit Galerkin coarse level matrices. Instead, it relies on geometric multigrid principles: a coarse level finite volume mesh can be constructed by grouping the fine level cells into clusters. Since the assembled linear equations are conservative (integral balance), the method retains this property by summing up the fine matrix elements which correspond to equations grouped into a single cluster. That is, boundary values are "absorbed" into the cluster centre. An additional correction is added to solution calculated on coarse level for each fine equation, to satisfy the integral balance. Since it arises from the finite volume method naturally, it is the most used algebraic multigrid metdod for the solution of Navier–Stokes equations. The

method was also applied to implicitly–coupled pressure–velocity system for two–dimensional flows by the same authors [34], later by Raw [35, 36] and adopted by Darwish et al. [20].

Multigrid methods are state–of–the–art iterative solver and are still being intensively developed as a part of several commercial codes for solving partial differential equations. The most recent reports are dedicated to parallelisation strategies, since some components of the algorithm are purely sequential. An overview of many parallelisation attempts is given in [37]. The core of the research is done by a group of authors who aspire to massively parallel multigrid algorithms. Some deal with challenges of efficient parallel scaling [38], some try to find optimal interpolation and relaxation methods for different types of equations [39] or explore aggresive coarsening techniques [40]. Multigrid is still an active topic of research and we hope that other users of `foam-extend` will help in the continuation of our efforts to efficiently apply it to general finite volume problems.

1.2. Present Contributions

The objective of this thesis is to establish a "best–practice" procedure for the solution of steady–state, single phase, incompressible and turbulent flows using implicit coupling of the momentum and continuity equations in the framework of arbitrary polyhedral finite volume method. The scientific contribution of the thesis can be summarised into the following statements:

1. Implicit coupling of the momentum and continuity equations is implemented in the framework of a publicly available open–source library for computational fluid dynamics `foam-extend`, which is a community driven fork of the `OpenFOAM` library. An overview of the derivation, discretisation and the underlying properties of the implicitly coupled system is presented in detail with additional comments on the implementation, structure and challenges stemming from the system in `foam-extend`. Such an extensive analysis cannot be found in present literature.

2. An overview of the state–of–the–art linear solvers, all available in `foam-extend`

is given from an engineer's point of view. Based on this overview, an appropriate linear algorithm is chosen for the solution of the linearised implicitly coupled pressure–velocity system: an algebraic multigrid method based on the selection of equations for construction of coarse levels, which was up to now, used in the finite element method. To the best of author's knowledge, there has been no report on the application or performance of the mentioned multigrid algorithm for the solution of the implicitly–coupled pressure–velocity system.

3. The block–selective algebraic multigrid, which did not exist in any publicly available open–source software until the implementation into `foam-extend` in the scope of this thesis, is adapted for arbitrary implicitly coupled equation sets, as well as scalar equations. It is now one of the standard linear solvers available in `foam-extend`, which allows further investigation for finite volume method applications. The algorithm is parallelised by modifying the coarsening process, which was also not investigated or reported in the scope of the finite volume method.

4. We have investigated the performance of the implicitly coupled pressure–velocity solver in comparison to the segregated algorithm based on the SIMPLE method for several test cases, using several linear solvers. After we concluded that block–selective algebraic multigrid provides superior performance, we investigated the influence of several algorithm parameters, which cannot be found in literature.

1.3. Thesis Outline

The remainder of the thesis is organised into the following chapters:

- In Chapter 2. the governing equations of steady–state, single phase, incompressible and turbulent flow are presented with emphasis on the pressure–velocity coupling issues and strategies, i.e. iterative procedures for the solution of the non–linear system are described. The conventional segregated approach is presented first, to familiarise the reader with the (currently)

most popular best–practice procedure for the solution of coupled equations. A derivation of the implicit solution technique, which is an improvement on the segregated algorithms, for the linearised momentum and continuity equation is given and the structure of the resulting block–coefficient matrix is laid out. In the last section of the chapter we focused on the well–known finite volume discretisation and the resulting structure and properties of the linear system.

- In Chapter 3. the attention is shifted onto iterative solution techniques for the linear system, obtained after linearisation of the momentum equation. Three classes of algorithms are presented: basic fixed–point iterations, algebraic multigrid methods and Krylov subspace algorithms. Since multigrid methods proved to be the most efficient option for the solution of the implicitly–coupled pressure velocity system, we describe the algorithm implemented into `foam-extend` in the scope of this thesis, which is based on the selection of equations as a coarsening technique. The challenges of the implementation, such as the choice of the block–element norm for the calculation of interpolation weights, choice of the optimal smoothing algorithm and parallelisation of the sequential setup phase are discussed in detail. Since Krylov subspace methods are an integral part in our multigrid algorithm, used as a method for obtaining the approximate fine level solution, an overview of the most frequently used methods is given: conjugate gradients, generalised minimal residual and biconjugate gradient stabilised. We also discuss the role and implementation of preconditioning techniques, specifically the incomplete lower upper factorisation.

- Chapter 4. is divided into two sections. The first section is dedicated to comparison of segregated and implicitly coupled non–linear solution techniques, where we illustrate the advantages and disadvantages of each solver. To be as close as possible to real engineering applications, we chose test cases of different complexities, mesh properties and densities: external flow around a Formula 1 front wing and a simplified diffuser section, flow around a BB2 submarine hull, internal flow inside a Francis turbine and centrifugal pump. The second section investigates the properties of linear solvers,

with emphasis on block–selection algebraic multigrid settings and resulting convergence rates. Three test cases were run: two–dimensional backward facing step, cooling of an engine jacket, a generic submarine model. A parallel scaling test is presented in this section as well to show the performance of the multigrid algorithm, since it has limitations in comparison to the same sequential algorithm.

- Chapter 5. contains the summary and conclusion drawn from this thesis. We have also given guidelines for future research.

2. Pressure–velocity system

2.1. Introduction

This chapter is dedicated to pressure–velocity coupling algorithms, i.e. resolving the linear coupling of the two fields. The non–linear part of the momentum equation (convection term) is eliminated by linearisation, thus non–linear iterative process will also be applied. Since there is no explicit equation for pressure, we show the derivation of the equation based on the Schur complement. Once the pressure equation is derived, there are two options for solving the system. In the segregated approach, reduced system of equations is being solved, i.e. the momentum and pressure equation are solved as two separate discrete systems, using the available values of the other variable. In the scope of this thesis we focus on an alternative approach, an implicitly coupled system in which both equations are solved simultaneously. We present the possible structures of the implicit block–matrix, as well as the relationship between the finite volume mesh and the structure of the matrix in `OpenFOAM`. The last section deals with various contributions to the linear system arising from the finite volume discretisation and the obtained linear equation properties. In the remainder of the thesis, we shall use symbolic notation to represent both continuous and discrete operators, which will be noted where necessary. Matrices and tensors are denoted using bold capitalised Latin letters, vectors are denoted with bold lower case Latin letters, while scalars are usually denoted by Greek letters.

2.2. Governing Equations

The governing equations of transient, incompressible, single–phase fluid flow are the continuity equation:

$$\underbrace{\nabla \cdot \mathbf{u}}_{\text{velocity divergence}} = 0, \tag{2.1}$$

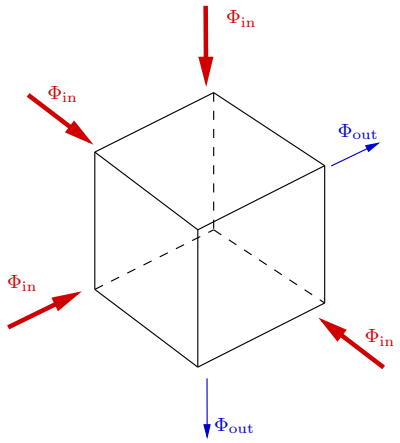
and the momentum equation:

$$\underbrace{\left(\frac{\partial \mathbf{u}}{\partial t}\right)^T}_{\text{local rate of change}} + \underbrace{\nabla \cdot (\mathbf{u}\mathbf{u}^T)}_{\text{convection}} - \underbrace{\nabla \cdot (\nu \nabla \mathbf{u}^T)}_{\text{diffusion}} = - \underbrace{(\nabla p)^T}_{\text{pressure gradient}}, \tag{2.2}$$

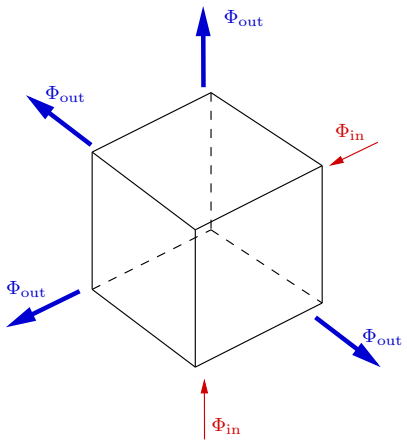
where $\mathbf{u}$ is the velocity, p is the kinematic pressure ($p = P/\rho$) and ν is the kinematic viscosity. In the scope of this thesis, turbulent flow was modelled using the Reynolds–averaged Navier–Stokes equations (RANS) in which the variables are decomposed into mean and fluctuating parts. The forces acting on the mean flow imposed by turbulent fluctuations (Reynolds stress tensor) are modeled using the two–equation turbulence models: k–ϵ [41] and k–ω–SST models [42], where k is the turbulent kinetic energy, ϵ is the rate of dissipation of k and similarly ω is the specific rate of dissipation. Eqn. (2.2) is non–linear due to the convection term $\nabla \cdot (\mathbf{u}\mathbf{u}^T)$. Since we want to avoid using non–linear solvers, the convection term in the momentum equation will be linearised which is shown in section 2.4.3. Finding an analytical solution of the continuity and momentum system of equations is impossible, except in cases of simplified, special types of flow. Instead, the equations are solved numerically, using iterative procedures, which is elaborated in section 2.3.

2.3. Pressure–Velocity Coupling Algorithms

A very important issue, which this section is dedicated to, is the mutual influence of pressure and velocity. The equation set consists of one vector equation (momentum) and one scalar equation (continuity), thus the system is closed. The problem is that the pressure does not appear in the continuity equation, Eqn. (2.1), i.e. we have a saddle point system of partial differential equations. Also, the pressure is present in the momentum equation, Eqn. (2.2), only as a gradient. Thus, the velocity is not affected by the absolute value of pressure, but rather by pressure differences. The pressure field can be determined to within an arbitrary constant. The methods for solving the pressure–velocity system rely on deriving an equation for the pressure from the continuity equation and using the calculated values of pressure to enforce the continuity of the velocity field. An illustration of asserting continuity for incompressible flow is given in Fig. 2.1: if



$\Phi_{in} > \Phi_{out}$
increase pressure in the cell to drive mass out.



$\Phi_{out} > \Phi_{in}$
decrease pressure in the cell to pull mass in.

Figure 2.1: A method for enforcing the mass continuity in incompressible flow is to manipulate the pressure field. If there is too much inflow, increase the pressure in the control volume to induce outflow. If there is too much outflow, decrease the pressure to turn the gradient in the opposite direction and pull the flow into the volume.

there is too much inflow, increase the pressure in the control volume to affect the pressure gradient and push the flow out, and vice versa. In the following sections, we shall present iterative algorithms for solving the pressure–velocity system in framework of the finite volume method.

2.3.1. SIMPLE Algorithm

The SIMPLE (Semi–Implicit Method for Pressure Linked Equations) algorithm [3] was conceived in 1972 by Suhas Patankar and his professor Brian Spalding, in a train somewhere between Wimbledon and Imperial College London [S. Patankar, private communication]. It is an iterative procedure for solving the steady–state pressure–velocity system which relies on decoupling the equations and solving them sequentially until the desired convergence criterion is reached. To define the steps of the SIMPLE algorithm, we shall use the discretised form of equations obtained from the finite volume method, written in terms of matrix elements, in line with [43]. The finite volume discretisation of the individual terms will be presented in section 2.4. The discretised momentum equation has the following

form:

$$\mathbf{A_u}\mathbf{u} = -\nabla p, \tag{2.3}$$

where $\mathbf{A_u}$ is a matrix which contains the discretised convection and diffusion operators. For a cell (control volume) i, the equivalent discretised momentum equation can be written as:

$$a_{ii}^{\mathbf{u}}\mathbf{u}_i + \sum_{j \neq i}^{N} a_{ij}^{\mathbf{u}}\mathbf{u}_j = \mathbf{r_b} - \nabla p, \tag{2.4}$$

where i denotes a value which belongs to the cell centre of cell i, while j denotes the values in the cell centres of neighbouring cells, with which cell i shares a face, see definitions in section 2.4.1. Thus, $a_{ii}^{\mathbf{u}}$ is the diagonal element and $a_{ij}^{\mathbf{u}}$ off–diagonal element of matrix $\mathbf{A_u}$. The vector $\mathbf{r_b}$ originates from $\mathbf{A_u}\mathbf{u}$ and it contains all the contributions which arise in the discretisation procedure and are chosen to be treated explicitly. For simplicity, Jasak [43] introduces a linear operator $\mathbf{H}$ which contains the off–diagonal part of the momentum matrix $\mathbf{A_u}$ and the right hand side vector $\mathbf{r_b}$:

$$\mathbf{H}(\mathbf{u}) = \mathbf{r_b} - \sum_{j \neq i}^{N} a_{ij}^{\mathbf{u}}\mathbf{u}_j. \tag{2.5}$$

The momentum equation can then be written as:

$$a_{ii}^{\mathbf{u}}\mathbf{u}_i = \mathbf{H}(\mathbf{u}) - \nabla p, \tag{2.6}$$

and $\mathbf{u}_i$ expressed as:

$$\mathbf{u}_i = (a_{ii}^{\mathbf{u}})^{-1}\left[\mathbf{H}(\mathbf{u}) - \nabla p\right]. \tag{2.7}$$

Substituting the expression for $\mathbf{u}_i$ into the continuity equation, $\nabla \cdot \mathbf{u} = 0$, yields the final form of the pressure equation:

$$\nabla \cdot [(a_{ii}^{\mathbf{u}})^{-1}\nabla p] = \nabla \cdot [(a_{ii}^{\mathbf{u}})^{-1}\mathbf{H}(\mathbf{u})]. \tag{2.8}$$

which is a variable coefficient Poisson equation. To solve the system, SIMPLE algorithm sequentially operates on Eqn. (2.4) and Eqn. (2.8). Here, a procedure implemented in `foam-extend` will be outlined. For clarity, we will simultaneously write both the differential (in symbolic notation) and discretised version of the governing equations.

1. In a single non–linear iteration k, the discretised momentum equation is solved to obtain the velocity field. The pressure gradient is explicitly calculated using the available values (guessed or calculated) of pressure, as indicated by the superscript $k-1$:

$$\nabla \cdot (\Phi^{(k-1)}\mathbf{u}) - \nabla \cdot (\nu \nabla \mathbf{u}) = -\nabla p^{(k-1)}. \tag{2.9}$$

Here, $\Phi^{(k-1)}$ is the *volumetric flux* from the previous iteration $k-1$, which appears in the linearisation of the convection term, section 2.4.3. It is defined on the cell face as the scalar product of the velocity $\mathbf{u}_f^{(k-1)}$ at the face, and the corresponding face normal vector $\mathbf{s}_f$, which has the magnitude equal to the face area ($\|\mathbf{s}_f\| = A_f$):

$$\Phi^{(k-1)} = \mathbf{s}_f^T \mathbf{u}_f^{(k-1)}. \tag{2.10}$$

The linear system obtained from the discretisation of Eqn. (2.2) also includes an implicit underrelaxation:

$$\frac{1}{\alpha_{\mathbf{u}}} a_{ii}^{\mathbf{u}} \mathbf{u}_i + \sum_{j \neq i}^{N} a_{ij}^{\mathbf{u}} \mathbf{u}_j = \mathbf{r_b} - \nabla p^{(k-1)} + \frac{1-\alpha_{\mathbf{u}}}{\alpha_{\mathbf{u}}} a_{ii}^{\mathbf{u}} \mathbf{u}_i^{(k-1)}, \tag{2.11}$$

where $0 < \alpha_{\mathbf{u}} \leq 1$ is the underrelaxation factor for the momentum equation. Implicit underrelaxation is built into the equation, rather then applied onto the solution, and it alleviates solving the linear system by increasing diagonal dominance of the equations, see Section 3.2.1. The underrelaxation is formulated so that once the final solution is reached, i.e. it stops changing in successive iterations, the terms affected by underrelaxation in the momentum equation, Eqn. (2.11), cancel out. The intermediate solution obtained from Eqn. (2.11), denoted as $\mathbf{u}^*$, does not satisfy the continuity equation!

2. After obtaining the solution of the momentum equation, i.e. the *intermediate velocity field* $\mathbf{u}^*$, the pressure equation is assembled in line with Eqn. (2.8):

$$\nabla \cdot [(a_{ii}^{\mathbf{u}})^{-1} \nabla p] = \nabla \cdot [(a_{ii}^{\mathbf{u}})^{-1} \mathbf{H}(\mathbf{u}^*)]. \tag{2.12}$$

It is important to notice that the effects of momentum underrelaxation are not included in the pressure equation, for consistency: the solution

should not be affected by the underrelaxation factor. The term under divergence on the right hand side can be interpreted as a *pseudo–velocity* $\hat{\mathbf{u}}$, and it also carries the boundary conditions from the momentum equation (e.g. Dirichlet at the inlet):

$$\hat{\mathbf{u}} = (a_{ii}^{\mathbf{u}})^{-1}\mathbf{H}(\mathbf{u}^*). \tag{2.13}$$

The pseudo–velocity $\hat{\mathbf{u}}$ is interpolated onto the cell faces and mulitiplied by the surface normal area vector to obtain the volumetric flux $\hat{\Phi}$:

$$\hat{\Phi} = \mathbf{s}_f^T \overline{\hat{\mathbf{u}}}, \tag{2.14}$$

where overline indicates the interpolation of $\hat{\mathbf{u}}$ from cell centres onto faces. The final form of the pressure equation is:

$$\nabla \cdot [(a_{ii}^{\mathbf{u}})^{-1}\nabla p] = \nabla \cdot \hat{\Phi}, \tag{2.15}$$

or in discrete form:

$$a_{ii}^p p_i + \sum_{j \neq i}^{N} a_{ij}^p p_j = \nabla \cdot \hat{\Phi}. \tag{2.16}$$

The solution of the pressure equation p^* contains both the physical pressure field which originates from the flow field, as well as the *correction part* which is responsible for enforcing the mass continuity and compensating the error in the initial pressure field. The corrected volumetric flux Φ for each face f, shared by cells i and j is calculated as:

$$\Phi^{(k)} = \hat{\Phi} - \sum_f a_{ij}^p (p_j^* - p_i^*). \tag{2.17}$$

The calculated pressure field is explicitly underrelaxed to counteract the unphysical component of the solution (implicit underrelaxation is not permitted as it would be inconsistent with the elliptic nature of the pressure equation):

$$p^{(k)} = (1 - \alpha_p)p^{(k-1)} + \alpha_p p^*, \tag{2.18}$$

where $0 < \alpha_p \leq 1$ is the pressure underrelaxation factor, p^* is the value of pressure calculated from the pressure equation and $p^{(k-1)}$ is the value of pressure which was used in the momentum equation. Thus, only α_p fraction of the new pressure field is taken into the final solution.

3. The velocity field $\mathbf{u}^*$ obtained from the momentum equation is corrected to satisfy the continutiy equation, using the underrelaxed values of the pressure field, and taking into account the implicit underrelaxation of the momentum equation:

$$\mathbf{u}^{(k)} = \alpha_{\mathbf{u}} \left(\hat{\mathbf{u}} - \frac{1}{a_{ii}^{\mathbf{u}}} \nabla p \right) + (1 - \alpha_{\mathbf{u}}) \, \mathbf{u}^*. \tag{2.19}$$

The sequence of these three steps is repeated until the linear and non–linear part of the solution converge to the desired tolerance. The repetition of this sequence is called the *outer* or *non–linear iteration*, while the solution of individual equations is the *inner* or *linear iteration*. Due to its age, the point of the SIMPLE algorithm is memory saving – once the momentum equation is solved, the same storage can be reused for the matrix elements of the pressure equation (the sparsity pattern of both matrices is identical). Thus, the memory peak of SIMPLE consists of a single matrix storage and working variables.

Rhie–Chow Correction

Since it causes confusion among `OpenFOAM` users, we shall discuss on the Rhie–Chow correction [44] in context of the presented implementation of the SIMPLE algorithm. The correction is necessary on collocated meshes, where all the unknowns are calculated at the centre of the cell. When calculating the pressure gradients for cell i in the momentum equation, i.e. the net pressure force acting on the faces of the cell, values of the pressure p_i cancel out due to the discretisation procedure (linear interpolation from cell centres). Since the pressure equation is derived by expressing the velocity from the momentum equation and inserting it into the divergence operator of the continuity equation, the connection between pressure values in neighbouring cells is lost here as well. This leads to oscillations in the solution of the pressure field, known as *pressure checkerboarding* [45]. Rhie–Chow interpolation provides a connection between the values of pressure in adjacent cells and smooths out the pressure field. The correction is applied when calculating the velocity at the face:

$$\mathbf{u}_f = \overline{\mathbf{u}}_f - \frac{1}{\overline{a}_{ii}^{\mathbf{u}}} (\nabla p_f - \overline{\nabla p}_f), \tag{2.20}$$

where overline indicates interpolation from cell centres onto faces. Face velocity defined by Eqn. (2.20) is used to calculate the volumetric flux in the continutiy equation, which takes the role of the pressure equation. Analysis of the SIMPLE algorithm in Section 2.3.1. reveals no explicit Rhie–Chow correction, since it is hidden by the $\mathbf{H}(\mathbf{u})$ operator. Recall the discretised momentum equation, Eqn. (2.4):

$$\begin{aligned} a_{ii}^{\mathbf{u}}\mathbf{u}_i + \sum_{j\neq i}^{N} a_{ij}^{\mathbf{u}}\mathbf{u}_j &= \mathbf{r_b} - \nabla p, \\ \underbrace{\mathbf{r_b} - \sum_{j\neq i}^{N} a_{ij}^{\mathbf{u}}\mathbf{u}_j}_{\mathbf{H}(\mathbf{u})} &= a_{ii}^{\mathbf{u}}\mathbf{u}_i + \nabla p, \\ \frac{1}{a_{ii}^{\mathbf{u}}}\mathbf{H}(\mathbf{u}) &= \mathbf{u}_i + \frac{1}{a_{ii}^{\mathbf{u}}}\nabla p. \end{aligned} \tag{2.21}$$

We shall now expand the Rhie–Chow correction, Eqn. (2.20):

$$\mathbf{u}_f = \overline{\mathbf{u}}_f - \frac{1}{\overline{a}_{ii}^{\mathbf{u}}}\nabla p_f + \frac{1}{\overline{a}_{ii}^{\mathbf{u}}}\overline{\nabla p}_f = \underbrace{\left(\overline{\mathbf{u}}_f + \frac{1}{\overline{a}_{ii}^{\mathbf{u}}}\overline{\nabla p}_f\right)}_{\overline{(a_{ii}^{\mathbf{u}})^{-1}\mathbf{H}(\mathbf{u})}} - \frac{1}{\overline{a}_{ii}^{\mathbf{u}}}\nabla p_f. \tag{2.22}$$

The bracketed term on the right hand side is the pseudo–velocity $\hat{\mathbf{u}}$ interpolated onto the cell faces, which was exactly done in the SIMPLE algorithm and inserted into the right hand side of the pressure equation, Eqn. (2.15). The second term is the pressure gradient which becomes the Laplacian of pressure when inserted into the divergence operator of the continuity equation. Thus, instead of introducing an artificial pressure term as proposed by Rhie and Chow, the same is achieved by using the face interpolation of the $\mathbf{H}(\mathbf{u})$ operator.

2.3.2. PISO Algorithm

The PISO algorithm (Pressure Implicit with Splitting of Operators) was developed by Raad Issa in 1986 [6]. It is similar to the SIMPLE algorithm with added pressure correction steps and it is usually used for simulation of transient flows. The idea behind multiple pressure correctors is to fully resolve the pressure–velocity coupling in a single timestep, thus eliminating the need for multiple

non–linear iterations. The algorithm presented in this section is a version implemented in `foam-extend`, with additional interpretation of certain segments. In a single timestep k, the following procedure is applied:

1. Solve the momentum equation using the pressure field from the previous timestep $k-1$, or guess the pressure field if there is none available:

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\Phi^{(k-1)}\mathbf{u}) - \nabla \cdot (\nu \nabla \mathbf{u}) = -\nabla p^{(k-1)}, \tag{2.23}$$

   where $\Phi^{(k-1)}$ is the *volumetric flux* from the linearisation of the convection term, calculated in the previous timestep. The linear system obtained after discretising the momentum equation has the following form:

$$\frac{\Delta \mathbf{u}}{\Delta t} + a_{ii}^{\mathbf{u}}\mathbf{u}_i + \sum_{j \neq i}^{N} a_{ij}^{\mathbf{u}}\mathbf{u}_j = \mathbf{r}_{\mathbf{b}} - \nabla p^{(k-1)}. \tag{2.24}$$

   Note that no underrelaxation is used, since the momentum equation is solved only once per timestep. The discretised inertial term $\Delta\mathbf{u}/\Delta t$ is written separately to emphasise the consistent derivation of the pressure equation in the next step of the algorithm. The solution of the momentum equation is the velocity field $\mathbf{u}^*$ which does not satisfy the continuity equation.

2. The pressure equation is assembled in the same way as in the SIMPLE algorithm, without taking into account the inertial effects of the time derivative in the momentum equation. This is important for consistency: the solution of the system should not depend on the timestep size [46]. Thus, the differential form of the pressure equation is:

$$\nabla \cdot [(a_{ii}^{\mathbf{u}})^{-1}\nabla p] = \nabla \cdot [(a_{ii}^{\mathbf{u}})^{-1}\mathbf{H}(\mathbf{u}^*)], \tag{2.25}$$

   where $\mathbf{H}(\mathbf{u}^*)$ contains the off–diagonal contributions from the momentum equation as well as any right hand side contributions, excluding the pressure gradient. For simplicity, the term on the right hand side is interpreted as a pseudo–velocity:

$$\hat{\mathbf{u}} = (a_{ii}^{\mathbf{u}})^{-1}\mathbf{H}(\mathbf{u}^*), \tag{2.26}$$

and it is interpolated onto the cell faces and mulitiplied by the surface normal area vector to obtain the volumetric flux $\hat{\Phi}$:

$$\hat{\Phi} = \mathbf{s}_f^T \, \overline{\hat{\mathbf{u}}}, \tag{2.27}$$

where overline indicates the interpolation of $\hat{\mathbf{u}}$ onto cell faces. The final form of the pressure equation is:

$$\nabla \cdot [(a_{ii}^{\mathbf{u}})^{-1} \nabla p] = \nabla \cdot \hat{\Phi}, \tag{2.28}$$

or in discrete form:

$$a_{ii}^p p_i + \sum_{j \neq i}^{N} a_{ij}^p p_j = \nabla \cdot \hat{\Phi}. \tag{2.29}$$

Once the solution of the pressure equation $p^{(k)}$ is obtained, the volumetric flux Φ which satisfies the continutiy equation for each face f, shared by cells i and j is calculated as:

$$\Phi^{(k)} = \hat{\Phi} - \sum_f a_{ij}^p (p_j^{(k)} - p_i^{(k)}). \tag{2.30}$$

The pressure field is not underrelaxed since the pressure equation will be solved multiple times.

3. The calculated velocity field is updated, taking into account the effects of time discretisation:

$$u^{(k)} = \frac{1}{a_{ii}^{\mathbf{u}} + \mathrm{diag}(\frac{\Delta \mathbf{u}}{\Delta t})} \left[a_{ii}^{\mathbf{u}} \hat{\mathbf{u}} - \nabla p^{(k)} + \mathrm{source}\left(\frac{\Delta \mathbf{u}}{\Delta t} \right) \right], \tag{2.31}$$

where $\mathrm{diag}(\Delta \mathbf{u}/\Delta t)$ is the diagonal contribution of the time discretisation and $\mathrm{source}(\Delta \mathbf{u}/\Delta t)$ is the right hand side contribution of the time discretisation. This is the point where the SIMPLE algorithm ends, while PISO returns to step 2, and solves the pressure equation again (until it reaches the user defined number of iterations). During the iterations, the updated velocity field affects the values of the pseudo–velocity $\hat{\mathbf{u}}$ and the volumetric flux $\hat{\Phi}$ on the right hand side of the pressure equation. It is possible to use PISO–like algorithm for steady–state flows, but the system is then stabilised using the underrelaxation of the momentum equation.

Both SIMPLE and PISO algorithms rely on decoupling the momentum and continutiy equations, i.e. a single unknown field is being solved while keeping the other constant, and successively repeating this procedure until convergence. Thus, SIMPLE and PISO belong to a group of *segregated algorithms*. The advantage of segregated algorithms is low memory peak during simulations (contains only one scalar matrix and solution field). Also, the solution of the linear system is somewhat easier since for each equation type (momentum – hyperbolic, pressure – elliptic) an appropriate and efficient linear solver can be chosen. The disadvantages of the segregated approach lie in the fact that the linear coupling of pressure and velocity has been broken by separating the equations into two independent systems. To prevent the divergence of the solution, underrelaxation is used in SIMPLE algorithm: implicit in the momentum equation and explicit for the pressure field. The PISO algorithm is more stable since the pressure equation is solved multiple times to counteract the non–physical, correction part of the solution, but requires small timestep sizes since the momentum equation is kept "frozen" during pressure iterations. Also, it has almost double memory peak compared to SIMPLE since both the momentum and pressure matrices must be kept in memory simultaneously.

2.3.3. Implicitly Coupled Pressure–Velocity System

SIMPLE and PISO algorithms were presented in sections 2.3.1. and 2.3.2. In segregated algorithms of this type, the pressure–velocity system is assembled by decoupling the momentum and pressure equations by treating the other unknown explicitly, using the value from the previous iteration, and solving the equations sequentially (Picard iterations [47, 48]). However, such treatment of the pressure–velocity system is unnatural since the connection between the two variables is linear and can be resolved simultaneously in a single linear system. The exact application of this procedure and the implications on the solution will be given in this section. Since the focus of this work was primarily on the linear system and the implementation and development of optimal linear solvers, steady–state problems shall be investigated and the time derivative in Eqn. (2.2) is omitted in the derivation of the implicitly coupled pressure–velocity algorithm. We shall

consider the steady–state versions of Eqn. (2.1) and Eqn. (2.2) ($\frac{\partial u}{\partial t} = 0$) in matrix format:

$$\begin{bmatrix} \mathbf{A_u} & \nabla \\ \nabla\cdot & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

where the linear combination of diffusion and linearised convection is replaced by a single term $\mathbf{A_u}$. Since the gradient operator is a transpose of the divergence operator:

$$\nabla = (\nabla\cdot)^T$$
$$\begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \end{bmatrix}^T,$$

we shall define $\nabla = \mathbf{G}$ to obtain:

$$\begin{bmatrix} \mathbf{A_u} & \mathbf{G} \\ \mathbf{G}^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{2.32}$$

The system defined in Eqn. (2.32) corresponds both to the differential and discretised form of equations but in the remainder of the section, we will assume that the system is discretised using the finite volume method, as described in section 2.3. Thus, $\mathbf{A_u}$ is the linear operator obtained by discretising convection and diffusion terms, while $\mathbf{G}$ and $\mathbf{G}^T$ are the discrete gradient and divergence operator, respectively, made implicit in the appropriate variable. Since there is a zero term on the diagonal, the solution of the system is a *saddle point* [47]. We shall use a preconditioning technique to obtain a stabilised version of the linear system by starting from a general linear system which contains two unknowns:

$$\mathbf{Ax} + \mathbf{By} = \mathbf{a} \tag{2.33}$$
$$\mathbf{Cx} + \mathbf{Dy} = \mathbf{b}, \tag{2.34}$$

where $\mathbf{A}_{m\times m}$, $\mathbf{B}_{m\times q}$, $\mathbf{C}_{q\times m}$ and $\mathbf{D}_{q\times q}$ are square matrices with dimensions m and q, as denoted in the indices, $\mathbf{x}$ and $\mathbf{y}$ are the unknown vectors, while $\mathbf{a}$ and $\mathbf{b}$ are the right hand side vectors. This system can be written using a block matrix $\mathbf{M}$:

$$\mathbf{M}_{(m+q)\times(m+q)} = \begin{bmatrix} \mathbf{A}_{m\times m} & \mathbf{B}_{m\times q} \\ \mathbf{C}_{q\times m} & \mathbf{D}_{q\times q} \end{bmatrix}, \tag{2.35}$$

to obtain:

$$\begin{bmatrix} \mathbf{A}_{m\times m} & \mathbf{B}_{m\times q} \\ \mathbf{C}_{q\times m} & \mathbf{D}_{q\times q} \end{bmatrix} \begin{bmatrix} \mathbf{x}_m \\ \mathbf{y}_q \end{bmatrix} = \begin{bmatrix} \mathbf{a}_m \\ \mathbf{b}_q \end{bmatrix}. \tag{2.36}$$

Assuming that $\mathbf{A}$ is invertible, we will express $\mathbf{x}$ from Eqn. (2.33) as:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{a} - \mathbf{A}^{-1}\mathbf{B}\mathbf{y}, \tag{2.37}$$

and substitute it into Eqn. (2.34) to obtain:

$$\begin{aligned} \mathbf{C}(\mathbf{A}^{-1}\mathbf{a} - \mathbf{A}^{-1}\mathbf{B}\mathbf{y}) + \mathbf{D}\mathbf{y} &= \mathbf{b} \\ \underbrace{(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})}_{\text{Schur complement}}\mathbf{y} &= \mathbf{b} - \mathbf{C}\mathbf{A}^{-1}\mathbf{a}. \end{aligned} \tag{2.38}$$

The first term in Eqn. (2.38) can be recognised as Schur complement of $\mathbf{A}$ in the block matrix $\mathbf{M}$ [49]. Using the Schur complement, it is possible to reduce the dimensions of the linear system: instead of solving the system with a $(m + q) \times (m + q)$ matrix $\mathbf{M}$, first find the solution $\mathbf{y}$ of Eqn. (2.38) ($q \times q$ problem) and then use $\mathbf{y}$ to calculate $\mathbf{x}$ ($m \times m$ problem), which is known as the Uzawa method for saddle point systems [48], and is equivalent to the SIMPLE algorithm. Comparison of Eqn. (2.32) and Eqn. (2.36) reveals the corresponding terms in the block pressure–velocity system:

$$\begin{aligned} &\mathbf{A} = \mathbf{A}_\mathbf{u}, \quad \mathbf{B} = \mathbf{G}, \quad \mathbf{C} = \mathbf{G}^T, \quad \mathbf{D} = 0, \\ &\mathbf{x} = \mathbf{u}, \quad \mathbf{y} = p, \quad \mathbf{a} = 0, \quad \mathbf{b} = 0. \end{aligned}$$

Thus, the Schur complement of the pressure–velocity system is obtained from Eqn. (2.38) by inserting the appropriate terms:

$$\begin{aligned} (\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})\mathbf{y} &= \mathbf{b} - \mathbf{C}\mathbf{A}^{-1}\mathbf{a} \\ -\mathbf{G}^T\mathbf{A}_\mathbf{u}^{-1}\mathbf{G}p &= 0. \end{aligned} \tag{2.39}$$

Using Eqn. (2.39) the block linear system 2.32 can be written as:

$$\begin{bmatrix} \mathbf{A}_\mathbf{u} & \mathbf{G} \\ 0 & -\mathbf{G}^T\mathbf{A}_\mathbf{u}^{-1}\mathbf{G} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{2.40}$$

The pressure equation is the Poisson equation, where the diffusion coefficient is equal to the inverse of the linearised convection–diffusion operator from the momentum equation:

$$-\nabla \cdot (\mathbf{A}_\mathbf{u}^{-1}\nabla p) = 0. \tag{2.41}$$

However, this formulation of the pressure equation is not used in the solution procedure due to numerical issues: inverting the linearised sparse convection–diffusion matrix $\mathbf{A_u}$ obtained from the finite volume discretisation is not trivial and produces a dense inverse which is too expensive to store! The remedy is to replace $\mathbf{A_u}$ with an approximation which is easy to invert. For example, it is possible to write $\mathbf{A_u}$ as a sum of diagonal, $\mathbf{D_u}$, and off–diagonal part, $\mathbf{E_u}$ and rewrite Eqn. (2.32):

$$\begin{bmatrix} \mathbf{D_u} & \mathbf{G} \\ \mathbf{G}^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} -(\mathbf{E_u})\mathbf{u} \\ 0 \end{bmatrix}. \tag{2.42}$$

The off–diagonal part of $\mathbf{A_u}$ is treated *explicitly*, i.e. it is moved into the right hand side vector, while the diagonal part is *implicit*, i.e. it remains in the coefficient matrix. Inserting the corresponding components of system 2.42 into Eqn. (2.38), yields the following form of the pressure equation:

$$\begin{aligned} (\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})\mathbf{y} &= \mathbf{b} - \mathbf{C}\mathbf{A}^{-1}\mathbf{a}, \\ (-\mathbf{G}^T\mathbf{D}_\mathbf{u}^{-1}\mathbf{G})p &= -\mathbf{G}^T\mathbf{D}_\mathbf{u}^{-1} \underbrace{[-(\mathbf{E_u})\mathbf{u}]}_{=(\mathbf{D_u})\mathbf{u}+\mathbf{G}p} \qquad (2.43) \\ &= -\mathbf{G}^T \underbrace{\mathbf{D}_\mathbf{u}^{-1}\mathbf{D_u}}_{\mathbf{I}} \mathbf{u} - \mathbf{G}^T\mathbf{D}_\mathbf{u}^{-1}\mathbf{G}p, \qquad (2.44) \end{aligned}$$

where $\mathbf{I}$ is the identity matrix and the inverse of the diagonal part of $\mathbf{A_u}$ is easy to calculate. Eqn. (2.43) can be identified in the SIMPLE algorithm as the pressure correction, Eqn. (2.8):

$$\begin{aligned} (-\mathbf{G}^T\mathbf{D}_\mathbf{u}^{-1}\mathbf{G})p &= -\mathbf{G}^T\mathbf{D}_\mathbf{u}^{-1}[-(\mathbf{E_u})\mathbf{u}] \\ \nabla \cdot \underbrace{\mathbf{D}_\mathbf{u}^{-1}}_{(a_{ii}^\mathbf{u})^{-1}} \nabla p &= \nabla \cdot [\underbrace{\mathbf{D}_\mathbf{u}^{-1}}_{(a_{ii}^\mathbf{u})^{-1}} \underbrace{(\mathbf{E_u})\mathbf{u}}_{\mathbf{H(u)}}]. \end{aligned} \tag{2.45}$$

We shall write Eqn. (2.44) using the gradient and divergence operators to obtain:

$$\nabla \cdot \mathbf{u} - \nabla \cdot (\mathbf{D}_\mathbf{u}^{-1}\nabla p) = -\nabla \cdot (\mathbf{D}_\mathbf{u}^{-1}\nabla p). \tag{2.46}$$

Eqn. (2.46) is the pressure equation which consists of the divergence of velocity term as well as two identical Laplacians of pressure field, where the diffusion coefficient is equal to the inverse of the diagonal part of the convection–diffusion matrix $\mathbf{D_u}$. The same equation for pressure would be obtained by inserting the

face velocity from the Rhie–Chow correction, Eqn. (2.20), into the continuity equation. The term on the right hand side was hidden in the SIMPLE algorithm by the interpolation of the $\mathbf{H}(\mathbf{u})$ operator. Thus, the saddle point problem is remedied by inserting a pressure Laplacian (symmetric positive definite matrix, Section 2.4.) in place of the zero term on the diagonal and counteracting it by adding the same term on the right hand side:

$$\begin{bmatrix} \mathbf{A_u} & \nabla \\ \nabla\cdot & -\nabla\cdot(\mathbf{D_u}^{-1}\nabla) \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} 0 \\ -\nabla\cdot(\overline{\mathbf{D_u}^{-1}\nabla p}) \end{bmatrix}, \tag{2.47}$$

where the explicit term $-\nabla\cdot(\overline{\mathbf{D_u}^{-1}\nabla p})$ is interpolated from cell centres to faces, in line with the Rhie–Chow correction. Thus, the solution procedure of the implicitly coupled pressure–velocity system consists of the following steps:

1. In a non–linear iteration k, the linear system which includes the linearised momentum equation and the continuity equation with the Rhie–Chow interpolation is assembled:

$$\begin{bmatrix} \mathbf{A_u} & \nabla \\ \nabla\cdot & -\nabla\cdot(\mathbf{D_u}^{-1}\nabla) \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} 0 \\ -\nabla\cdot(\overline{\mathbf{D_u}^{-1}\nabla p^{(k-1)}}) \end{bmatrix}.$$

   As denoted by the superscript $(k-1)$, for the explicit Rhie–Chow correction, old values of the pressure field are used. Also, the convection–diffusion matrix $\mathbf{A_u}$ contains the values of the volumetric flux $\Phi^{(k-1)}$ from the previous iteration. The momentum equation may be implicitly underrelaxed for stability (as in the SIMPLE algorithm), while there is no underrelaxation of pressure. The system is solved to obtain the values of pressure $p^{(k)}$ and velocity $\mathbf{u}^{(k)}$.

2. From the continuity equation, the new volumetric flux which will be used in the convection term in the following non–linear iteration is calculated:

$$\Phi^{(k)} = \mathbf{s}_f^T \mathbf{u}_f^{(k)} + \sum_f a_{ij}^p (p_i^{(k)} - p_j^{(k)}) + \overline{\mathbf{D_u}^{-1}\nabla p^{(k-1)}}. \tag{2.48}$$

   The non–linear iterative procedure is repeated until the desired convergence criterion is reached.

Since the majority of information in this approach is treated implicitly (put into the coefficient matrix), the convergence is more stable and the solution procedure requires very little or no underrelaxation. That is, there is no underrelaxation of the pressure since the equation is linear and the effects of the pressure gradient are implicitly included in the momentum equation. In the momentum equation, non–linear effects are present due to the convection term and some equation underrelaxation can be used, Eqn. (2.11). The finite volume discretisation of the momentum and continuity equation will be presented in the following section. The resulting block–matrix of the implicitly coupled system will be layed out to illustrate the various contributions of the discretisation technique.

2.4. Finite Volume Equation Discretisation

In this section the finite volume discretisation of the governing equations and show the contribution of individual terms to the coefficient matrix and the right hand side of the linear system will be laid out. The objective of this section is to clearly demonstrate the structure of the coefficient matrix of the implicitly coupled pressure–velocity system and subsequently justify the choice of the linear solver, which is covered in chapter 3.

2.4.1. Mesh and Matrix

The discretisation of equations will be carried out on a finite volume *mesh* which fills the solution domain, using the terminology and procedures introduced by Jasak in [43]. The mesh consists of non–overlapping three–dimensional *cells*, i.e. control volumes (Fig. 2.2), which are bounded by two–dimensional ("flat") surfaces called cell *faces*. V_i is the cell volume while $\mathbf{r}_i$ is the position of the cell centroid (*cell centre*). All variables are calculated (stored) at cell centres, which is a characteristic of the *collocated* finite volume method. Cells are distinguished by their index. If a mesh consists of n cells, the indices range from 0 to $n-1$. Each cell face has its surface area A_f, centre, normal vector $\mathbf{s}_f$ and it is shared between maximally two cells. In `OpenFOAM`, the cell with the lower index is called the *owner* of the face and the one with the higher index is the *neighbour*. The

origin of the face normal is at the face centre, it points outwards from the cell and the length (magnitude) is equal to the face area ($\|\mathbf{s}_f\| = A_f$). The terminology is general, even though the topology of cells can be arbitrary (hexahedra, tetrahedra, polyhedra, pyramids, prisms).

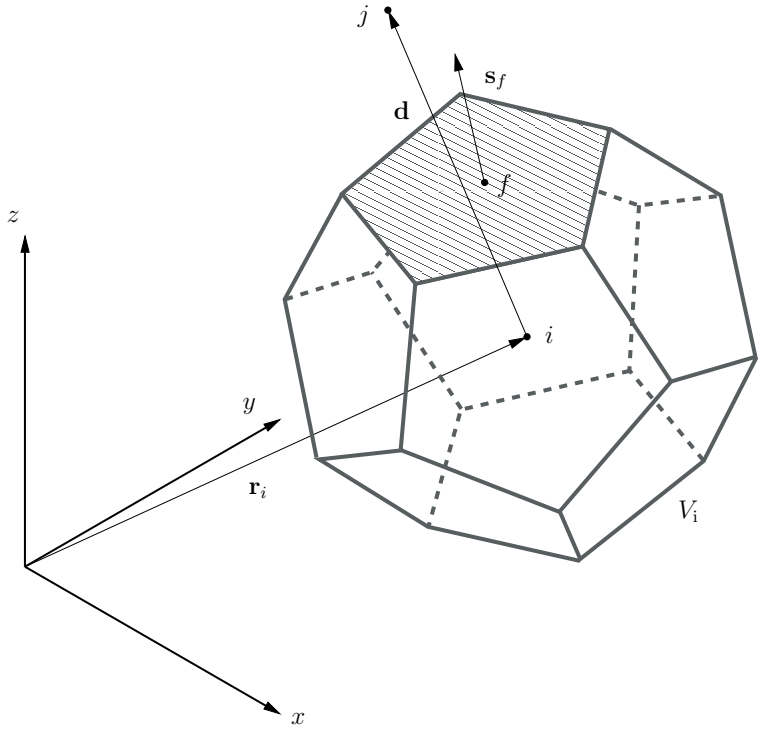


Figure 2.2: Arbitrary polyhedral finite volume cell with distinctive features: cell index i, position of the cell centre $\mathbf{r}_i$, face centre f, face area vector $\mathbf{s}_f$, distance $\mathbf{d}$ between cell centres of adjacent cells i and j.

An example of a two–dimensional finite volume mesh is shown in Fig. 2.3. The dimensions of the matrix $\mathbf{A}$ corresponding to a n–cell mesh are $n \times n$, i.e. a linear equation for a cell is represented in the matrix by a single row. Depending on the indices of the elements, the matrix is split into its *diagonal, upper and lower triangular* parts, shown in dark blue, light blue and grey in Fig. 2.4, respectively. The diagonal contains elements a_{ii} have equal row and column indices, i.e. they represent the influence of the cell i on itself. Off–diagonal elements a_{ij} represent the influence of neighbouring cells j onto cell i. The lower triangle contains elements which have column indices smaller than row indices ($j < i$), while the upper triangle has elements which have column indices larger than row indices

($j > i$). The matrices arising from finite volume discretisation are *sparse* (have more zero than non–zero elements).

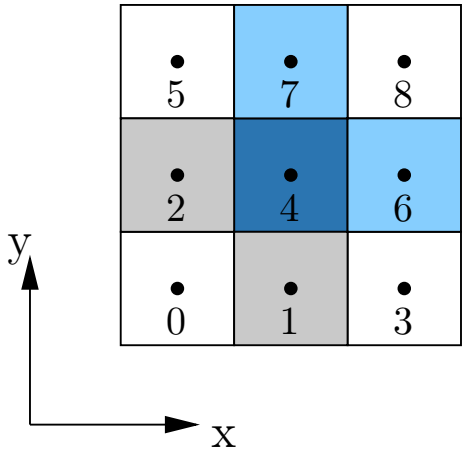


Figure 2.3: A two–dimensional finite volume mesh. Each cell is marked with an index, which corresponds to the matrix row, Fig. 2.4. "Onion" numbering is used: each new index is assigned to an unmarked cell which is a neighbour of the cell with the smallest index, similar to layers of an onion.

The sparsity pattern, i.e. the position of the non–zero elements in the matrix depends on mesh connectivity and numbering. As we shall see in the following sections, row i in the finite volume matrix will contain the diagonal element and as many off–diagonal elements as there are neighbours j of cell i. The tendency is to keep the off–diagonal elements as close as possible to the diagonal, to obtain a *banded matrix*. A banded matrix has a specific structure: apart from the main diagonal, it is possible to identify secondary diagonals above or below the main diagonal. Banded matrices are a result of optimisation algorithms for mesh numbering, e.g. the "onion" algorithm (reverse Cuthill–McKee, [50]), as in Fig. 2.3.

In `foam-extend` matrices can be distignuished by the type of elements: matrices with *scalar elements*, which correspond to a mesh where in each cell there is only one unknown, and *block–elements*, where there are multiple unknowns in a single cell. The reason for using block–elements is the efficiency of programming: the algorithms for scalar and block–matrices are equivalent since the addressing (sparsity pattern) of matrix elements is the same. For example, in SIMPLE algorithm the momentum and pressure equations are solved sequentially. Thus, two

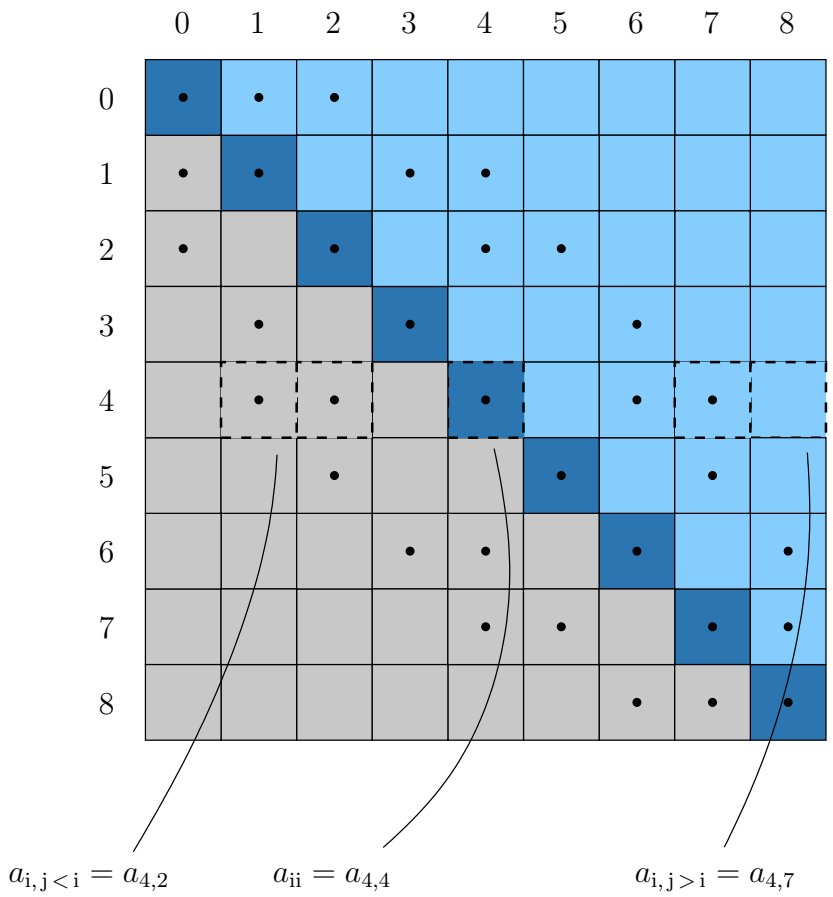


Figure 2.4: A finite volume matrix corresponding to a two–dimensional mesh shown in Fig. 2.3. Dark blue denotes the diagonal elements, light blue the upper triangle and grey lower triangle elements. Row corresponding to cell 4 is highlighted to ilustrate the addresing principles.

matrices are constructed: a matrix for the momentum equation and a matrix for the pressure equation. In SIMPLE, the pressure matrix is a scalar matrix as each cell i contains a single value of pressure p_i. Thus, the matrix elements are scalars a_{ii} and a_{ij}.

When solving the momentum equation, in a single cell i there exist three components of velocity, thus the unknown $\mathbf{x}_i$ is a vector:

$$\mathbf{x}_i = \begin{bmatrix} u_{xi} \\ u_{yi} \\ u_{zi} \end{bmatrix} .$$

The matrix elements in the momentum matrix should be 3×3 matrices themselves, but there is no coupling between the components of the velocity. Thus,

the solution procedure is simplified even more, by solving a scalar system for each component of the velocity.

When pressure and velocity are implicitly coupled, in a three–dimensional space four equations are being solved for each cell i. The system can be arranged in two ways:

- **variable–ordered system**, where in the unknown vector a single variable is written for all cells, followed by other variables in the same manner. For example, the unknown–ordered pressure–velocity system:

$$\begin{bmatrix} \mathbf{U_x} & \mathbf{U_{xy}} & \mathbf{U_{xz}} & \mathbf{U_{xp}} \\ \mathbf{U_{yx}} & \mathbf{U_y} & \mathbf{U_{yz}} & \mathbf{U_{yp}} \\ \mathbf{U_{zx}} & \mathbf{U_{zy}} & \mathbf{U_z} & \mathbf{U_{zp}} \\ \mathbf{P_x} & \mathbf{P_y} & \mathbf{P_z} & \mathbf{P} \end{bmatrix} \begin{bmatrix} \mathbf{u}_x \\ \mathbf{u}_y \\ \mathbf{u}_z \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_{u_x} \\ \mathbf{b}_{u_y} \\ \mathbf{b}_{u_z} \\ \mathbf{b}_p \end{bmatrix},$$

where block matrices have the dimension $n \times n$ ($\mathbf{U}$, $\mathbf{P} \in \mathbb{R}^{n \times n}$):

$$\mathbf{U_x} = \begin{bmatrix} a_{u_{x0},u_{x0}} & \dots & a_{u_{x0},u_{x_{n-1}}} \\ \vdots & \ddots & \vdots \\ a_{u_{x_{n-1}},u_{x0}} & \dots & a_{u_{x_{n-1}},u_{x_{n-1}}} \end{bmatrix}, \; \mathbf{U_{xy}} = \begin{bmatrix} a_{u_{x0},u_{y_0}} & \dots & a_{u_{x0},u_{y_{n-1}}} \\ \vdots & \ddots & \vdots \\ a_{u_{x_{n-1}},u_{y_0}} & \dots & a_{u_{x_{n-1}},u_{y_{n-1}}} \end{bmatrix},$$

$$\mathbf{P} = \begin{bmatrix} a_{p_0,p_0} & \dots & a_{p_0,p_{n-1}} \\ \vdots & \ddots & \vdots \\ a_{p_{n-1},p_0} & \dots & a_{p_{n-1},p_{n-1}} \end{bmatrix}, \; \mathbf{P_y} = \begin{bmatrix} a_{p_0,u_{y_0}} & \dots & a_{p_0,u_{y_{n-1}}} \\ \vdots & \ddots & \vdots \\ a_{p_{n-1},u_{y_0}} & \dots & a_{p_{n-1},u_{y_{n-1}}} \end{bmatrix},$$

and elements $a_{\cdot,\cdot}$ are scalars which describe the coupling between variables in adjacent cells. For example, $a_{u_{x0},u_{x_{n-1}}}$ describes the coupling of velocity component in x–direction in cells 0 and $n-1$, while $a_{p_{n-1},u_{y_{n-1}}}$ corresponds to the coupling of pressure and y–velocity component in cell $n-1$. When the usual linearisation of convection term is used, there is no coupling between components of velocity in different directions, i.e. matrices $\mathbf{U_{xy}}$, $\mathbf{U_{xz}}$, $\mathbf{U_{yx}}$, $\mathbf{U_{yz}}$, $\mathbf{U_{zx}}$ and $\mathbf{U_{zy}}$ are zero matrices. The structure of the system corresponds to the block matrix in Eqn. (2.47).

- **cell–ordered system**, where in the unknown vector all variables are written sequentially for a single cell, followed by other cells. The cell–ordered pressure–

velocity system has the following structure:

$$\begin{bmatrix} \mathbf{A_{0,0}} & \cdots & \mathbf{A_{0,n-1}} \\ \vdots & \ddots & \vdots \\ \mathbf{A_{n-1,0}} & & \mathbf{A_{n-1,n-1}} \end{bmatrix} \begin{bmatrix} \mathbf{x}_0 \\ \vdots \\ \mathbf{x}_{n-1} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_0 \\ \vdots \\ \mathbf{b}_{n-1} \end{bmatrix},$$

The unknown variable in each cell is a vector with 4 components ($\mathbf{x} \in \mathbb{R}^{4\times 1}$), while the matrix elements are 4×4 matrices ($\mathbf{A_{ij}} \in \mathbb{R}^{4\times 4}$):

$$\mathbf{x}_i = \begin{bmatrix} u_{xi} \\ u_{y_i} \\ u_{zi} \\ p_i \end{bmatrix}, \quad \mathbf{A_{ij}} = \begin{bmatrix} a_{u_{xi},u_{xj}} & a_{u_{xi},u_{y_j}} & a_{u_{xi},u_{zj}} & a_{u_{xi},p_j} \\ a_{u_{y_i},u_{xj}} & a_{u_{y_i},u_{y_j}} & a_{u_{y_i},u_{zj}} & a_{u_{y_i},p_j} \\ a_{u_{zi},u_{xj}} & a_{u_{zi},u_{y_j}} & a_{u_{zi},u_{zj}} & a_{u_{zi},p_j} \\ a_{p_i,u_{xj}} & a_{p_i,u_{y_j}} & a_{p_i,u_{zj}} & a_{p_i,p_j} \end{bmatrix}.$$

The diagonal element $\mathbf{A}_{ii}$ describes the mutual influence of velocity components and pressure in cell i, while the off–diagonal element $\mathbf{A}_{ij}$ contains the coupling of velocity components and pressure in adjacent cells i and j. The structure of the block–elements in the cell–ordered implicitly coupled pressure–velocity system emerges from the matrix form of the governing equations, Eqn. (2.47):

$$\begin{bmatrix} \mathbf{A_u} & \nabla \\ \nabla\cdot & -\nabla\cdot(\mathbf{D_u^{-1}}\nabla) \end{bmatrix} \equiv \begin{bmatrix} a_{u_x,u_x} & a_{u_x,u_y} & a_{u_x,u_z} & a_{u_x,p} \\ a_{u_y,u_x} & a_{u_y,u_y} & a_{u_y,u_z} & a_{u_y,p} \\ a_{u_z,u_x} & a_{u_z,u_y} & a_{u_z,u_z} & a_{u_z,p} \\ a_{p,u_x} & a_{p,u_y} & a_{p,u_z} & a_{p,p} \end{bmatrix}. \tag{2.49}$$

Thus, the upper 3×3 block (in red) corresponds to the discretisation of the convection–diffusion operator for the velocity, the blue column vector corresponds to the pressure gradient term, the green row vector is the discretised velocity divergence operator and the orange scalar is the discretised pressure Poisson equation. The elements which describe the coupling between different components of velocity (off–diagonal in the momentum matrix) are equal to 0.

In the scope of this thesis, we have chosen the latter approach of a cell–ordered system, which is more natural in the framework of `foam-extend` (mesh and matrix addressing presented in the following section) and the collocated finite volume method.

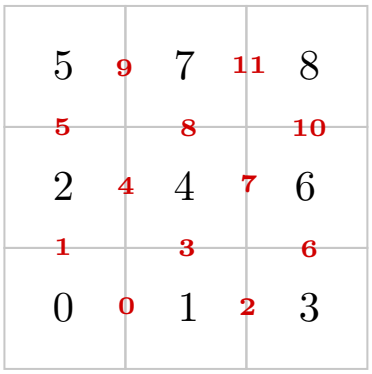


Figure 2.5: A simple 2D finite volume mesh. Cell indices are shown in black, while the face indices are shown in red.

OpenFOAM Matrix Format

The structure of matrices in OpenFOAM is directly related to the structure and numbering of the computational mesh. Since these matrices are sparse and often symmetric, an *arrow* format is used, where only the non–zero elements are stored. The matrix elements are divided into three arrays depending on the *address (row and column index)* in the matrix: *diagonal, lower and upper array*, and this format is called *LDU matrix* [H. Jasak, private communication]. For a symmetric matrix, it is only necessary to store the upper or lower triangle elements since $\mathbf{L} = \mathbf{U}^T$, where $\mathbf{L}$ is the lower triangular matrix and $\mathbf{U}^T$ is the transpose of the upper triangle. The addressing of matrix elements in OpenFOAM is colloquially known as *face addressing*, since the elements in the upper and lower array are sorted depending on whether they are the owner or a neighbour of a face. This type of addressing is not straightforward in terms of access to the elements at a certain position in the matrix (which is important for the implementation of linear solvers), but it is very convenient for efficient access when needed in the finite volume equation discretisation (e.g. interpolation).

The LDU matrix for a mesh shown in Fig. 2.5 is presented in Table 2.1. Face index f indicates the position in the array, e.g. face 3 is shared by cells 1 (owner) and 4 (neighbour), which are both stored in their corresponding array in position 3. The matrix elements obtained from the finite volume equation discretisation which correspond to communication between cells 1 and 4 are also located in position 3 of their corresponding arrays. The indexing of cells is done

Table 2.1: Arrays corresponding to LDU matrix format in OpenFOAM.

Face index f	**Owner cell** i	**Neighbour cell** j	**Upper element** a_{ij}	**Lower element** a_{ji}
0	0	1	$a_{0,1}$	$a_{1,0}$
1	0	2	$a_{0,2}$	$a_{2,0}$
2	1	3	$a_{1,3}$	$a_{3,1}$
3	1	4	$a_{1,4}$	$a_{4,1}$
4	2	4	$a_{2,4}$	$a_{4,2}$
5	2	5	$a_{2,5}$	$a_{5,2}$
6	3	6	$a_{3,6}$	$a_{6,2}$
7	4	6	$a_{4,6}$	$a_{6,4}$
8	4	7	$a_{4,7}$	$a_{7,4}$
9	5	7	$a_{5,7}$	$a_{7,5}$
10	6	8	$a_{6,8}$	$a_{8,6}$
11	7	8	$a_{7,8}$	$a_{8,7}$

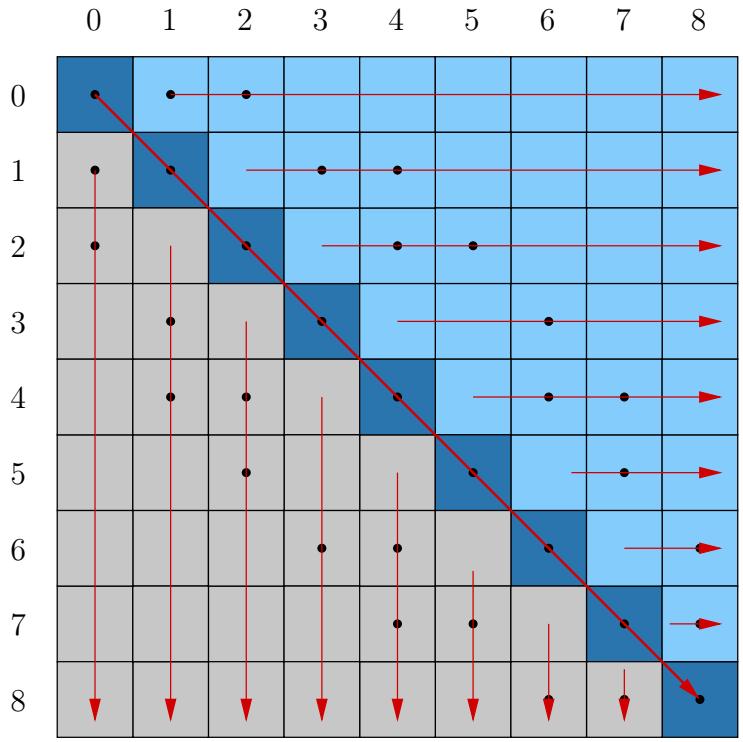


Figure 2.6: OpenFOAM LDU matrix format. Matrix elements are stored in three arrays: diagonal, lower and upper. Red arrows illustrate the order of writing the elements into these arrays. Non–zero elements in the upper triangle are stored row–wise, while the elements in lower triangle are stored column–wise (for a symmetric matrix $\mathbf{L} = \mathbf{U}^T$).

prior to indexing of faces: face indices are consecutively assigned to an unnamed face which has the owner and corresponding neighbour with the smallest indices. The lower array contains the column–wise sorted elements of the lower triangle, while the elements in the upper array are sorted row–wise (beneficial for storing symmetric matrices). Additionaly, for easier access to elements, an index pointing to the beginning of each row/column in the upper and lower array is uses, which is equivalent to compressed row matrix format.

2.4.2. Preliminaries for Spatial Terms

The discretisation procedure starts by integrating the differential equations over each control volume. Assuming that a general variable changes linearly through space and time (second order accuracy), the surface and volume integrals which arise from the governing equations will be evaluated using the Gauss–Ostrogradsky theorem for transformation of volume integrals into surface integrals, [43]. The integral of a scalar quantity ϕ over a control volume (cell) i is:

$$\int_{V_i} \phi(x)\, dV = \phi_i V_i, \tag{2.50}$$

where V_i is the volume of cell i. The integral over a cell of the divergence of a vector $\mathbf{a}$ is equal to:

$$\int_{V_i} \nabla \cdot \mathbf{a}\, dV = \oint \mathbf{n}^T \mathbf{a}\, dS = \sum_f \mathbf{s}_f^T \mathbf{a}_f = \sum_{f\,(j>i)} \mathbf{s}_f^T \mathbf{a}_f - \sum_{f\,(j<i)} \mathbf{s}_f^T \mathbf{a}_f, \tag{2.51}$$

where $\mathbf{s}_f$ is the normal face area vector and $\mathbf{a}_f$ is vector $\mathbf{a}$ evaluated at the cell face. The splitting of the sum is done according to the sign of $\mathbf{s}_f$: positive for owned faces ($j > i$) and negative for neighbour's faces ($j < i$), since only the normal from the owner side of the face is stored. Thus, divergence of a field can be replaced by a sum over the faces of a cell. The integral over a cell of the gradient of a scalar field ϕ is equal to:

$$\int_{V_i} \nabla\phi\, dV = \oint \mathbf{n}\phi\, dS = \sum_f \mathbf{s}_f \phi_f, \tag{2.52}$$

where ϕ_f is the value evaluated at the face. In the software implementation, face interpolation is achieved by a single face loop which handles both the owner and neighbour cell and it is extremely efficient.

2.4.3. Convection Term

The non–linear convection term appears in the momentum equation. The term is linearised since we want to avoid using non–linear solvers, usually by using the values from the previous iteration:

$$\mathbf{u}\mathbf{u}^T = \mathbf{u}^{(k-1)}(\mathbf{u}^{(k)})^T, \tag{2.53}$$

where k denotes the current iteration. The linearised term is then discretised for cell i:

$$\int_{V_i} \nabla \cdot [\mathbf{u}^{(k-1)}(\mathbf{u}^{(k)})^T] = \sum_f \mathbf{s}_f^T \mathbf{u}_f^{(k-1)}(\mathbf{u}_f^{(k)})^T = \sum_f \Phi_f \mathbf{u}_f^{(k)}, \tag{2.54}$$

where Φ_f is the volumetric flux, which in SIMPLE and PISO algorithm is calculated via Eqn. (2.17), while in the implicitly coupled solver Eqn. (2.48) is used. The value of velocity $\mathbf{u}_f^{(k)}$ on the cell face has to be estimated from the values of $\mathbf{u}^{(k)}$ in cell centres. This approximation should preserve the boundedness of the variable, since the convection operator is bounded. There are several options for convection discretisation:

1. **Central differencing (linear interpolation)** Central differencing is second order accurate [45], but boundedness cannot be guaranteed:

$$\mathbf{u}_f = w_{\mathrm{CD}}\, \mathbf{u}_i + (1 - w_{\mathrm{CD}})\, \mathbf{u}_j, \tag{2.55}$$

where w_{CD} is the weighting factor calculated as a ratio of the distance between the face centre and cell centre of j, and the distance $\|\mathbf{d}\|$ between cell centres of i and j. Central differencing for convection contributes to the linear system of the implicitly coupled equations in the coefficient matrix, both to diagonal and off–diagonal elements. The contribution for off–diagonal elements ($j \neq i$) is skew–symmetric due to the sign of the face normal vector, i.e. $\mathbf{A}_{ij} = -\mathbf{A}_{ji}$:

$$\mathbf{A}_{ij} = \begin{bmatrix} a_{u_{x_i},u_{x_j}} & a_{u_{x_i},u_{y_j}} & a_{u_{x_i},u_{z_j}} & a_{u_{x_i},p_j} \\ a_{u_{y_i},u_{x_j}} & a_{u_{y_i},u_{y_j}} & a_{u_{y_i},u_{z_j}} & a_{u_{y_i},p_j} \\ a_{u_{z_i},u_{x_j}} & a_{u_{z_i},u_{y_j}} & a_{u_{z_i},u_{z_j}} & a_{u_{z_i},p_j} \\ a_{p_i,u_{x_j}} & a_{p_i,u_{y_j}} & a_{p_i,u_{z_j}} & a_{p_i,p_j} \end{bmatrix}.$$

2. **Upwind differencing**

Upwind differencing guarantees boundedness of the solution, but causes the loss of accuracy for meshes which are not aligned with the flow, through numerical diffusion [43]. The value of velocity at the face depends on the direction of the flow, i.e. volumetric flux Φ. It is copied from the cell centre located upstream:

$$\mathbf{u}_f = \begin{cases} \mathbf{u}_i & \text{for } \Phi \geq 0, \\ \mathbf{u}_j & \text{for } \Phi < 0. \end{cases} \tag{2.56}$$

Upwind differencing for convection contributes to the linear system in the coefficient matrix, and the contribution is nonsymmetric, i.e. $\mathbf{A}_{ij} \neq \mathbf{A}_{ji}$:

$$\mathbf{A}_{ij} = \begin{bmatrix} a_{u_{xi},u_{xj}} & a_{u_{xi},u_{y_j}} & a_{u_{xi},u_{zj}} & a_{u_{xi},p_j} \\ a_{u_{y_i},u_{xj}} & a_{u_{y_i},u_{y_j}} & a_{u_{y_i},u_{zj}} & a_{u_{y_i},p_j} \\ a_{u_{zi},u_{xj}} & a_{u_{zi},u_{y_j}} & a_{u_{zi},u_{zj}} & a_{u_{zi},p_j} \\ a_{p_i,u_{xj}} & a_{p_i,u_{y_j}} & a_{p_i,u_{zj}} & a_{p_i,p_j} \end{bmatrix}.$$

3. **Blended differencing**

Blended differencing (by Perić [51]) is a hybrid between central and upwind differencing, a compromise between accuracy and boundedness. A constant blending factor γ is proposed for all faces:

$$\begin{aligned} \mathbf{u}_f &= (1-\gamma)\mathbf{u}_{\mathrm{UD}} + \gamma\,\mathbf{u}_{\mathrm{CD}} \\ &= \{(1-\gamma)\max[\mathrm{sign}(\Phi), 0] + \gamma w_{\mathrm{CD}}\}\ \mathbf{u}_i \\ &\quad + \{(1-\gamma)\min[\mathrm{sign}(\Phi), 0] + \gamma(1-w_{\mathrm{CD}})\}\ \mathbf{u}_j, \end{aligned} \tag{2.57}$$

where indices UD and CD denote the face velocity obtained by upwind and central differencing, respectively. The scheme reduces to upwind differencing for $\gamma = 0$, and to central differencing for $\gamma = 1$. The contribution of blended differencing to the linear system appears in the matrix elements, and the symmetry of the elements depends on the blending factor γ, but it is generally nonsymmetric ($\mathbf{A}_{ij} \neq \mathbf{A}_{ji}$):

$$\mathbf{A}_{ij} = \begin{bmatrix} a_{u_{xi},u_{xj}} & a_{u_{xi},u_{y_j}} & a_{u_{xi},u_{zj}} & a_{u_{xi},p_j} \\ a_{u_{y_i},u_{xj}} & a_{u_{y_i},u_{y_j}} & a_{u_{y_i},u_{zj}} & a_{u_{y_i},p_j} \\ a_{u_{zi},u_{xj}} & a_{u_{zi},u_{y_j}} & a_{u_{zi},u_{zj}} & a_{u_{zi},p_j} \\ a_{p_i,u_{xj}} & a_{p_i,u_{y_j}} & a_{p_i,u_{zj}} & a_{p_i,p_j} \end{bmatrix}.$$

4. **Higher order convection schemes**

An improvement in terms of accuracy and boundedness trade–off are blended schemes which try to locally determine the value of the blending factor γ, i.e. variants of *total variation diminishing (TVD)* [52] and *normalised variable diagram (NVD)* schemes [53]. *Gamma differencing scheme*, derived by Jasak [43] for unstructured meshes, is one of the variants of an NVD scheme: a boundedness criterion is defined using a non–dimensional transported variable $\tilde{\phi}$. Central differencing is used as a default discretisation scheme across the entire domain, except when the criterion is breached, i.e. when the boundedness of the variable is no longer preserved. In that case, a blending factor $0 \leq \gamma \leq 1$ is calculated and it depends on the local value of the normalised variable $\tilde{\phi}_i$ and a user defined constant of the scheme β_{m}. The scheme is shown in an NVD diagram, as well as other common blended schemes, Fig. 2.7. Details of other schemes and a comparison of TVD and NVD diagrams can be found in [54].

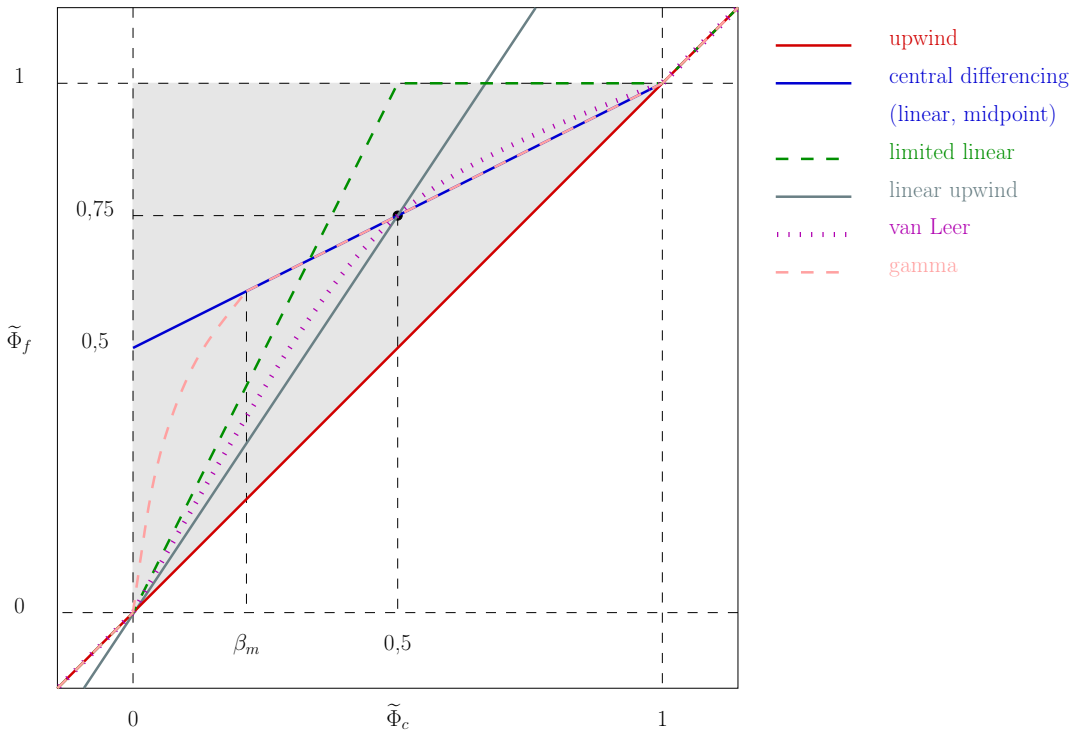


Figure 2.7: NVD diagram of blended convection schemes.

Deferred correction approach [55] can be applied for higher order schemes to

achieve the stability of upwind discretisation. The upwind part of the higher order scheme is always treated implicitly and inserted into the coefficient matrix, while the higher order term is calculated using the old values of $\mathbf{u}$ and inserted into the right hand side.

2.4.4. Velocity Diffusion Term

Disretisation of the diffusion term relies on turning the volume integral into a surface integral and finally, into a sum over cell faces:

$$\int_{V_i} \nabla \cdot (\nu \nabla \mathbf{u}) = \sum_f (\nu)_f \, \mathbf{s}_f^T (\nabla \mathbf{u})_f. \tag{2.58}$$

Diffusion coefficient on the face is calculated by linear interpolation. For meshes which are orthogonal, i.e. the face area vector $\mathbf{s}_f$ is parallel to the vector $\mathbf{d}$ which connects the two cell centres, the gradient on the face can be calculated using the cell centre values of $\mathbf{u}$ [56]:

$$(\nabla \mathbf{u})_f = \frac{\mathbf{u}_i - \mathbf{u}_j}{\|\mathbf{d}\|}. \tag{2.59}$$

Orthogonal meshes are very rare in practice. The measure of non–orthogonality is the angle formed by $\mathbf{s}_f$ and $\mathbf{d}$, $\measuredangle(\mathbf{d}, \mathbf{s}_f)$. Eqn. (2.59) is not used in this form if the mesh is non–orthogonal. Instead, the face area vector is split into an *orthogonal vector* $\boldsymbol{\Delta}$, which coincides with the distance vector $\mathbf{d}$, and *non–orthogonal* vector $\mathbf{k}$:

$$\mathbf{s}_f^T (\nabla \mathbf{u})_f = (\boldsymbol{\Delta} + \mathbf{k})^T (\nabla \mathbf{u})_f = \underbrace{\boldsymbol{\Delta}^T (\nabla \mathbf{u})_f}_{\text{orthogonal part}} + \underbrace{\mathbf{k}^T (\nabla \mathbf{u})_f}_{\text{non–orthogonal part}}. \tag{2.60}$$

The orthogonal part of Eqn. (2.60) is treated implicitly, i.e. put into the coefficient matrix, using Eqn. (2.59). The non–orthogonal part is treated explicitly and contributes to the right hand side vector $\mathbf{b}$. The interpolation of $(\nabla \mathbf{u})_f$ is done using the face values of $\mathbf{u}$ to calculate the cell centered gradient (Gaussian integration):

$$(\nabla \mathbf{u})_i = \frac{1}{V_i} \sum_f \mathbf{s}_f^T \mathbf{u}_f. \tag{2.61}$$

The gradient is then linearly interpolated to the face:

$$(\nabla \mathbf{u})_f = w_{\mathrm{CD}} \, (\nabla \mathbf{u})_i + (1 - w_{\mathrm{CD}}) \, (\nabla \mathbf{u})_j. \tag{2.62}$$

Thus, the contribution of the diffusion term to the linear system occurs in the matrix elements (orthogonal part, off–diagonal is symmetric $\mathbf{A}_{ij} = \mathbf{A}_{ji}$) and the right hand side vector (non–orthogonal part):

$$\mathbf{A}_{ij} = \begin{bmatrix} a_{u_{xi},u_{xj}} & a_{u_{xi},u_{y_j}} & a_{u_{xi},u_{zj}} & a_{u_{xi},p_j} \\ a_{u_{y_i},u_{xj}} & a_{u_{y_i},u_{y_j}} & a_{u_{y_i},u_{zj}} & a_{u_{y_i},p_j} \\ a_{u_{zi},u_{xj}} & a_{u_{zi},u_{y_j}} & a_{u_{zi},u_{zj}} & a_{u_{zi},p_j} \\ a_{p_i,u_{xj}} & a_{p_i,u_{y_j}} & a_{p_i,u_{zj}} & a_{p_i,p_j} \end{bmatrix}, \quad \mathbf{b}_i = \begin{bmatrix} b_{u_{xi}} \\ b_{u_{y_i}} \\ b_{u_{zi}} \\ b_{p_i} \end{bmatrix}.$$

To boost the convergence of the system, it is beneficial to have as much information in the coefficient matrix as possible. Having that in mind, the length of the orthogonal contribution can be defined in several ways:

- by keeping the non–orthogonal vector $\mathbf{k}$ as small as possible, i.e. ensuring that $\boldsymbol{\Delta}$ and $\mathbf{k}$ are orthogonal. This is the **minimum correction approach**. Orthogonal vector is calculated as:

$$\boldsymbol{\Delta} = \frac{\mathbf{d}^T \mathbf{s}_f}{\mathbf{d}^T \mathbf{d}} \mathbf{d}, \tag{2.63}$$

  while $\mathbf{k} = \mathbf{s}_f - \boldsymbol{\Delta}$. As non–orthogonality of the mesh increases, the contribution to the coefficient matrix decreases, which is detrimental for convergence of linear solvers. Meshes with non–orthogonality angles $\measuredangle(\mathbf{d}, \mathbf{s}_f) \geq 90°$ are invalid.

- By keeping the implicit contribution the same as on the orthogonal mesh, irrespective of the non–orthogonality. This is the **orthogonal correction approach**:

$$\boldsymbol{\Delta} = \frac{\mathbf{d}}{\|\mathbf{d}\|} \|\mathbf{s}_f\|, \tag{2.64}$$

$$\|\boldsymbol{\Delta}\| = \|\mathbf{s}_f\|. \tag{2.65}$$

- By increasing the magnitude of the implicit orthogonal contribution. This is the **overrelaxed approach**:

$$\boldsymbol{\Delta} = \frac{\mathbf{d}}{\mathbf{d}^T \mathbf{s}_f} \|\mathbf{s}_f\|^2. \tag{2.66}$$

The choice of the diffusion discretisation scheme depends on the mesh quality, i.e. the stability of the simulation. The application of non–orthogonal correction

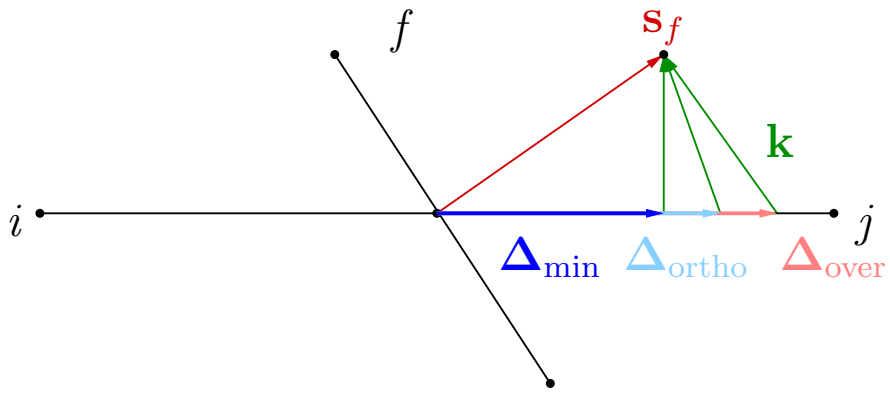


Figure 2.8: Non–orthogonal correction. When calculating the face gradient, take into account the non–orthogonality $\measuredangle(\mathbf{d}, \mathbf{s}_f)$ of the mesh and split the face area vector $\mathbf{s}_f$ into two parts: orthogonal component $\boldsymbol{\Delta}$ and non–orthogonal component $\mathbf{k}$. The magnitude of $\boldsymbol{\Delta}$ can vary depending on the splitting: $\boldsymbol{\Delta}_{\text{min}}$ for minimum correction, $\boldsymbol{\Delta}_{\text{ortho}}$ for orthogonal correction and $\boldsymbol{\Delta}_{\text{over}}$ for overrelaxed correction.

can cause unboundedness, and sometimes it has to be limited or not used at all. The limiter Υ is user defined, $0 \leq \Upsilon \leq 1$, where $\Upsilon = 1$ denotes no limiter (i.e. orthogonal correction approach is applied), and $\Upsilon = 0$ denotes that there is no orthogonal correction:

$$\Upsilon \underbrace{\|\mathbf{s}_f\|}_{\|\boldsymbol{\Delta}\|} \frac{\mathbf{u}_i - \mathbf{u}_j}{\|\mathbf{d}\|} > \mathbf{k}^T (\nabla \mathbf{u})_f. \tag{2.67}$$

2.4.5. Pressure Gradient

Pressure gradient which appears in the momentum equation is treated explicitly in SIMPLE and PISO algorithms, i.e. it is calculated from the available (old) values of pressure and put onto the right hand side of the linear system. However, in the implicitly coupled system, the effects of the pressure gradient are inserted into the coefficient matrix. In the scope of this thesis, the following gradient discretisation schemes were used:

1. **Gauss gradient** Discretisation begins by converting the volume integral into a surface integral, i.e. a sum over cell faces where the pressure gradient is replaced by the pressure at the face, as shown in Section 2.4.2.,

Eqn. (2.52):

$$\int_{V_i} \nabla p \, dV = \sum_f \mathbf{s}_f p_f.$$

Thus, the remaining task is to choose the appropriate interpolation technique for the value of the pressure at the face. The face value of pressure is calculated using the values from adjacent cells:

$$p_f = w_{\mathrm{CD}} \, p_i + (1 - w_{\mathrm{CD}}) \, p_j, \tag{2.68}$$

where w_{CD} is the weighting factor calculated as a ratio of the distance between the face centre and cell centre of j, and the distance $\|\mathbf{d}\|$ between cell centres of i and j, which yields:

$$(\nabla p)_i = \sum_f \mathbf{s}_f [w_{\mathrm{CD}} \, p_i + (1 - w_{\mathrm{CD}}) \, p_j]. \tag{2.69}$$

The contribution to the coefficient matrix in off–diagonal elements is skew–symmetric ($\mathbf{A}_{ij} = -\mathbf{A}_{ji}$):

$$\mathbf{A}_{ij} = \begin{bmatrix} a_{u_{xi},u_{xj}} & a_{u_{xi},u_{y_j}} & a_{u_{xi},u_{zj}} & a_{u_{xi},p_j} \\ a_{u_{y_i},u_{xj}} & a_{u_{y_i},u_{y_j}} & a_{u_{y_i},u_{zj}} & a_{u_{y_i},p_j} \\ a_{u_{zi},u_{xj}} & a_{u_{zi},u_{y_j}} & a_{u_{zi},u_{zj}} & a_{u_{zi},p_j} \\ a_{p_i,u_{xj}} & a_{p_i,u_{y_j}} & a_{p_i,u_{zj}} & a_{p_i,p_j} \end{bmatrix}.$$

2. **Least squares discretisation**

The order of accuracy of Gauss gradient diminishes in case of skewed meshes. Least squares discretisation is unconditionally second order accurate. The value in an adjacent cell j can be evaluated using the value in cell i and the gradient of the corresponding field. The error at cell j produced by extrapolation is:

$$e_j = p_j - [p_i + \mathbf{d}^T (\nabla p)_i], \tag{2.70}$$

where e_j is the error term in p_j and $\mathbf{d}^T(\nabla p)_i$ is the component of the gradient in the direction of the distance vector $\mathbf{d}$. In reverse, the value of the gradient can be evaluated by minimising the weighted error e_i [56], i.e. by minimising the sum of the squares of weighted errors e_j:

$$e_i^2 = \sum_j (w_{\mathrm{LSQ}} e_j)^2, \tag{2.71}$$

where the weighting factor is defined as $w_{\mathrm{LSQ}} = 1/\|\mathbf{d}\|$. Assuming that $e_i^2 = 0$ yields:

$$e_i^2 = \sum_j w_{\mathrm{LSQ}}^2 (p_j - [p_i + \mathbf{d}^T(\nabla p)_i])^2 = 0 \tag{2.72}$$

$$(\nabla p)_i = \sum_j \mathbf{G_{LSQ}}^{-1} \mathbf{d}\,(p_j - p_i). \tag{2.73}$$

$\mathbf{G_{LSQ}}$ is assumed to be an invertible 3×3 symmetric matrix:

$$\mathbf{G_{LSQ}} = \sum_j w_{\mathrm{LSQ}}^2 \mathbf{d}\mathbf{d}^T \tag{2.74}$$

$$= \sum_j \|\mathbf{d}\|^2 \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} \begin{bmatrix} d_x & d_y & d_z \end{bmatrix} \tag{2.75}$$

$$= \sum_j \|\mathbf{d}\|^2 \begin{bmatrix} d_x d_x & d_x d_y & d_x d_z \\ d_y d_x & d_y d_y & d_y d_z \\ d_z d_x & d_z d_y & d_z d_z \end{bmatrix}, \tag{2.76}$$

where d_x, d_y and d_z are the components of the distance vector $\mathbf{d}$ parallel to the corresponding coordinate axes. Matrix $\mathbf{G_{LSQ}}$ depends only on mesh topology and, for non–moving meshes, it can be calculated once and stored for future iterations. The value of the gradient for cell i is calculated as:

$$V_i(\nabla p)_i = \sum_j V_i \mathbf{G_{LSQ}}^{-1} \mathbf{d}\,(p_j - p_i), \tag{2.77}$$

where V_i is the volume of i, and it produces a symmetric contribution to

off–diagonal elements to the coefficient matrix ($\mathbf{A}_{ij} = \mathbf{A}_{ji}$):

$$\mathbf{A}_{ij} = \begin{bmatrix} a_{u_{xi},u_{xj}} & a_{u_{xi},u_{y_j}} & a_{u_{xi},u_{zj}} & a_{u_{xi},p_j} \\ a_{u_{y_i},u_{xj}} & a_{u_{y_i},u_{y_j}} & a_{u_{y_i},u_{zj}} & a_{u_{y_i},p_j} \\ a_{u_{zi},u_{xj}} & a_{u_{zi},u_{y_j}} & a_{u_{zi},u_{zj}} & a_{u_{zi},p_j} \\ a_{p_i,u_{xj}} & a_{p_i,u_{y_j}} & a_{p_i,u_{zj}} & a_{p_i,p_j} \end{bmatrix}.$$

2.4.6. Velocity Divergence

The divergence of velocity is treated implicitly in the pressure equation. From the structure of the saddle point system, it is equal to a transpose of the pressure gradient term. Thus, to preserve the symmetry the same discretisation procedure should be used for these two terms. Here, we shall present the central differencing discretisation:

$$\begin{aligned} \int_{V_i} \nabla \cdot \mathbf{u}\, dV &= \sum_f \mathbf{s}_f^T \mathbf{u}_f \\ &= \sum_f \mathbf{s}_f^T [w_{\mathrm{CD}} \mathbf{u}_i + (1 - w_{\mathrm{CD}}) \mathbf{u}_j], \end{aligned} \tag{2.78}$$

where w_{CD} is the central differencing weighting factor, equal to the ratio of the distance between the face centre and cell centre of j and the length of the distance vector $\|\mathbf{d}\|$. Velocity divergence contributes to the diagonal and off–diagonal elements of the coefficient matrix and the contribution to off–diagonal elements is skew–symmetric ($\mathbf{A}_{ij} = -\mathbf{A}_{ji}$):

$$\mathbf{A}_{ij} = \begin{bmatrix} a_{u_{xi},u_{xi}} & a_{u_{xi},u_{y_i}} & a_{u_{xi},u_{zi}} & a_{u_{xi},p_i} \\ a_{u_{y_i},u_{xi}} & a_{u_{y_i},u_{y_i}} & a_{u_{y_i},u_{zi}} & a_{u_{y_i},p_i} \\ a_{u_{zi},u_{xi}} & a_{u_{zi},u_{y_i}} & a_{u_{zi},u_{zi}} & a_{u_{zi},p_i} \\ a_{p_i,u_{xi}} & a_{p_i,u_{y_i}} & a_{p_i,u_{zi}} & a_{p_i,p_i} \end{bmatrix}.$$

2.4.7. Pressure Laplacian

Implicit term in the pressure equation is a Laplacian operator where the diffusivity is equal to the inverse of the diagonal element of the momentum matrix. It is discretised using the sum–over–the–faces procedure, as it was shown for the diffusion term in the momentum equation, section 2.4.4.:

$$\int_{V_i} \nabla \cdot (\mathbf{D}_{\mathbf{u}}^{-1} \nabla p)\, dV = \sum_f \left(a_{ii}{}^{-1}\right)_f \mathbf{s}_f^T (\nabla p)_f. \tag{2.79}$$

Interpolation of the pressure gradient to cell faces is done using central differencing, with or without non–orthogonal correction. Discretisation of pressure Laplacian produces a contribution to diagonal and off–diagonal matrix elements and to the source term, when non–orthogonal correction is applied:

$$\mathbf{A}_{ij} = \begin{bmatrix} a_{u_{x_i},u_{x_j}} & a_{u_{x_i},u_{y_j}} & a_{u_{x_i},u_{z_j}} & a_{u_{x_i},p_j} \\ a_{u_{y_i},u_{x_j}} & a_{u_{y_i},u_{y_j}} & a_{u_{y_i},u_{z_j}} & a_{u_{y_i},p_j} \\ a_{u_{z_i},u_{x_j}} & a_{u_{z_i},u_{y_j}} & a_{u_{z_i},u_{z_j}} & a_{u_{z_i},p_j} \\ a_{p_i,u_{x_j}} & a_{p_i,u_{y_j}} & a_{p_i,u_{z_j}} & a_{p_i,p_j} \end{bmatrix}, \quad \mathbf{b}_i = \begin{bmatrix} b_{u_{x_i}} \\ b_{u_{y_i}} \\ b_{u_{z_i}} \\ b_{p_i} \end{bmatrix}.$$

The contribution to off–diagonal elements is symmetric ($\mathbf{A}_{ij} = \mathbf{A}_{ji}$), and the sum of contributions in block–matrix elements in a single row (for internal cells) is equal to zero:

$$\sum_{j\ (i=\text{const})} a_{p_i,p_j} = 0. \tag{2.80}$$

Such matrix rows are called diagonally equal (sum of magnitudes of off–diagonal elements is equal to the magnitude of the diagonal element). The sign of diagonal and off–diagonal elements is opposite (negative diagonal, positive off–diagonal element).

2.4.8. Source Terms

The only source term, i.e. explicit right hand side contribution which appears in the system is the term in the pressure equation which corresponds to Rhie–Chow correction:

$$\int_{V_i} \nabla \cdot (\overline{\mathbf{D}_\mathbf{u}^{-1} \nabla p})\, dV = \sum_f \left((\overline{a_{ii}})^{-1} \right)_f \mathbf{s}_f^T \overline{(\nabla p)}_f. \tag{2.81}$$

The overline indicates interpolation from cell centres to cell faces. First, the cell centred pressure gradient is calculated, using the available (old, $k-1$) values of pressure:

$$(\nabla p)_i^{(k-1)} = \frac{\sum_f \mathbf{s}_f\, p_f^{(k-1)}}{V_i}, \tag{2.82}$$

where $p_f^{(k-1)}$ is calculated via central differencing, using the cell centre values of the pressure field. Then, the cell centred gradient is linearly interpolated on the

faces, as well as the inverse of the diagonal element a_{ii}^{-1}. The contribution of Rhie–Chow appears only in the right hand side vector:

$$\mathbf{b}_i = \begin{bmatrix} b_{u_{x\,i}} \\ b_{u_{y\,i}} \\ b_{u_{z\,i}} \\ b_{p_i} \end{bmatrix}.$$

2.4.9. Boundary Conditions

Boundary conditions are assigned on the borders of the computational domain, for the boundary faces. Boundary conditions can be divided into *numerical* and *physical boundary conditions*. Numerical boundary conditions are:

- *Dirichlet boundary condition*, where the value of the variable is prescribed on the boundary,
- *von Neumann boundary condition*, where the value of the gradient of a variable ($\nabla\phi$) is prescribed, and projected onto the boundary face area vector:

$$\frac{\mathbf{s}_f}{\|\mathbf{s}_f\|}\nabla\phi = g_b, \tag{2.83}$$

- *Robin (mixed) boundary condition*, where the linear combination of a fixed value and gradient is prescribed on the boundary.

Physical boundary conditions are impermeable walls, flow inlets and outlets, symmetry planes, cyclic and periodic planes, etc. and they can be represented by a set of numerical boundary conditions for each unknown variable. The topology of a boundary cell is shown in Fig. 2.9. Distance vector $\mathbf{d}$ points from the cell centre to the boundary face centre b. As shown in the figure, $\mathbf{d}$ and the face area vector $\mathbf{s}_f$ do not have to be parallel (similar to non–orthogonality between cells), and an orthogonal projection $\mathbf{d}_\mathrm{N}$ of vector $\mathbf{d}$ is defined as:

$$\mathbf{d}_\mathrm{N} = \frac{\mathbf{s}_f}{\|\mathbf{s}_f\|}\frac{\mathbf{d}^T\mathbf{s}_f}{\|\mathbf{s}_f\|}. \tag{2.84}$$

Since boundary cells are usually fully or nearly orthogonal, the non–orthogonal component $\mathbf{k}$ is not used. The common boundary conditions used for incompressible flows are:

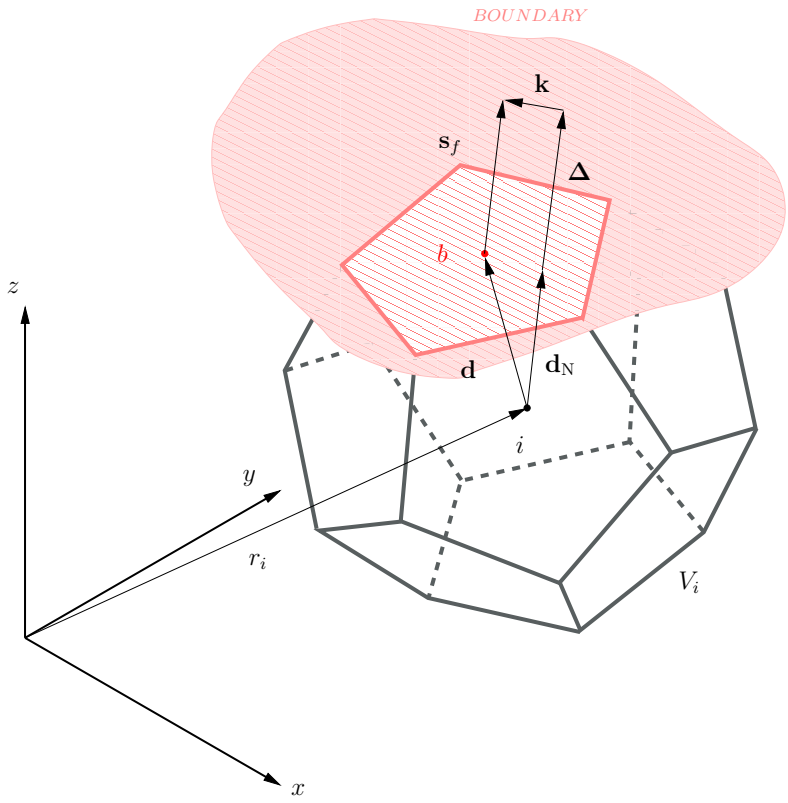


Figure 2.9: A finite volume cell at the boundary of the computational domain.

- *inlet boundaries*, where the velocity field is prescribed (Dirichlet), while the value of the pressure is extrapolated from the interior (pressure gradient set to zero);

- *outlet boundaries*, where the velocity is extrapolated from the interior (velocity gradient set to zero) and the value of static pressure is assigned. In some cases, it is possible to use a zero gradient for pressure as well, but then a value of pressure has to be set in a point (or cell) somewhere in the domain;

- *impermeable no–slip walls* which are modelled with a fixed value of fluid velocity at the wall. The pressure at the wall is extrapolated from the interior of the domain (pressure gradient equal to zero);

- *symmetry planes* which imply that the component of the gradient normal to the boundary is equal to zero. Components parallel to the boundary are

projected to the boundary face from the interior of the domain.

Boundary contributions to the linear system can again be analysed term by term for both equations:

- **Convection term** According to the discretisation, the face value of velocity $\mathbf{u}_f$ is needed:

$$\sum_f \Phi_f \mathbf{u}_f^{(k)}.$$

For a fixed value (Dirichlet) boundary condition, the value $\mathbf{u}_b$ is directly used and contributes to the right hand side vector:

$$\mathbf{b}_i = \begin{bmatrix} b_{u_{xi}} \\ b_{u_{yi}} \\ b_{u_{zi}} \\ b_{p_i} \end{bmatrix}. \tag{2.85}$$

For a fixed gradient (von Neumann) boundary condition, the value of velocity at the boundary face is extrapolated using the cell centre value and the specified gradient g_b of $\mathbf{u}$:

$$\mathbf{u}_b = \mathbf{u}_i + \|\mathbf{d}_\mathrm{N}\| g_b. \tag{2.86}$$

Fixed gradient for convection contributes to the diagonal element $\mathbf{A}_{ii}$ and the right hand side as in Eqn. (2.85):

$$\mathbf{A}_{ii} = \begin{bmatrix} a_{u_{xi},u_{xi}} & a_{u_{xi},u_{yi}} & a_{u_{xi},u_{zi}} & a_{u_{xi},p_i} \\ a_{u_{yi},u_{xi}} & a_{u_{yi},u_{yi}} & a_{u_{yi},u_{zi}} & a_{u_{yi},p_i} \\ a_{u_{zi},u_{xi}} & a_{u_{zi},u_{yi}} & a_{u_{zi},u_{zi}} & a_{u_{zi},p_i} \\ a_{p_i,u_{xi}} & a_{p_i,u_{yi}} & a_{p_i,u_{zi}} & a_{p_i,p_i} \end{bmatrix}. \tag{2.87}$$

- **Velocity diffusion term**

For velocity diffusion, the estimate of the boundary face gradient $(\nabla \mathbf{u})_f$ is needed:

$$\sum_f \nu_f \, \mathbf{s}_f^T (\nabla \mathbf{u})_f.$$

If a fixed value boundary condition is prescribed, the gradient is calculated using the face value $\mathbf{u}_b$:

$$(\nabla \mathbf{u})_f = \frac{\mathbf{u}_b - \mathbf{u}_i}{\|\mathbf{d}_\mathrm{N}\|}, \tag{2.88}$$

and a contribution to the diagonal element and right hand side emerges, Eqn. (2.87) and Eqn. (2.85). For a von Neumann boundary condition, the given value of the gradient g_b is directly used and inserted into the right hand side vector Eqn. (2.85). If a symmetry (Robin) boundary condition is used, cross–coupling elements between components of velocity appear. At the symmetry plane, normal component of the velocity is equal to zero, while the tangential component is equal to the projection of the velocity in the cell centre to the boundary. The face normal gradient of tangential velocity is equal to zero. Thus, there exists only the gradient of the normal component of velocity. The normal component can be expressed as:

$$\mathbf{u}_\mathrm{N} = (\mathbf{u}^T \mathbf{n})\, \mathbf{n} \tag{2.89}$$

$$= (u_x n_x + u_y n_y + u_z n_z)\, \mathbf{n} \tag{2.90}$$

$$= \begin{bmatrix} (u_x n_x + u_y n_y + u_z n_z) n_x \\ (u_x n_x + u_y n_y + u_z n_z) n_y \\ (u_x n_x + u_y n_y + u_z n_z) n_z \end{bmatrix}, \tag{2.91}$$

where $\mathbf{u}^T\mathbf{n}$ denotes the projection of velocity vector onto the face normal unit vector $\mathbf{n}$ and $\mathbf{n} = \mathbf{s}_f/\|\mathbf{s}_f\|$. Since the normal component of velocity on the boundary face is equal to zero ($(\mathbf{u}_\mathrm{N})_b = 0$), and there exists a symmetrical contribution from the "mirrored" side, the value of the velocity gradient on the face is equal to:

$$(\nabla \mathbf{u})_f = 2\frac{-\mathbf{u}_\mathrm{N}}{\|\mathbf{d}_\mathrm{N}\|}. \tag{2.92}$$

Thus, cross–coupling terms between different components of velocity appear due to the symmetry plane boundary condition:

$$\mathbf{A}_{ii} = \begin{bmatrix} a_{u_{x_i},u_{x_i}} & a_{u_{x_i},u_{y_i}} & a_{u_{x_i},u_{z_i}} & a_{u_{x_i},p_i} \\ a_{u_{y_i},u_{x_i}} & a_{u_{y_i},u_{y_i}} & a_{u_{y_i},u_{z_i}} & a_{u_{y_i},p_i} \\ a_{u_{z_i},u_{x_i}} & a_{u_{z_i},u_{y_i}} & a_{u_{z_i},u_{z_i}} & a_{u_{z_i},p_i} \\ a_{p_i,u_{x_i}} & a_{p_i,u_{y_i}} & a_{p_i,u_{z_i}} & a_{p_i,p_i} \end{bmatrix}. \tag{2.93}$$

- **Pressure gradient**

  If a fixed value of pressure p_b is prescribed at the boundary, the value is used instead of the value at the centre of adjacent cell j, i.e. $p_j = p_b$, to calculate the value of the gradient at the face in the case of central differencing scheme:

$$\sum_f \mathbf{s}_f p_f$$

$$p_f = \frac{p_i - p_b}{\|\mathbf{d}\|}.$$

  For least squares interpolation, the corresponding component is calculated using the boundary value:

$$(\nabla p)_i = \mathbf{G_{LSQ}}^{-1}\,\mathbf{d}(p_b - p_i).$$

  Fixed boundary value in the pressure gradient term contributes to the diagonal element and right hand side vector:

$$\mathbf{A}_{ii} = \begin{bmatrix} a_{u_{xi},u_{xi}} & a_{u_{xi},u_{y_i}} & a_{u_{xi},u_{zi}} & a_{u_{xi},p_i} \\ a_{u_{y_i},u_{xi}} & a_{u_{y_i},u_{y_i}} & a_{u_{y_i},u_{zi}} & a_{u_{y_i},p_i} \\ a_{u_{zi},u_{xi}} & a_{u_{zi},u_{y_i}} & a_{u_{zi},u_{zi}} & a_{u_{zi},p_i} \\ a_{p_i,u_{xi}} & a_{p_i,u_{y_i}} & a_{p_i,u_{zi}} & a_{p_i,p_i} \end{bmatrix}, \; \mathbf{b}_i = \begin{bmatrix} b_{u_{xi}} \\ b_{u_{y_i}} \\ b_{u_{zi}} \\ b_{p_i} \end{bmatrix}. \tag{2.94}$$

  If a pressure gradient g_b is assigned on the boundary, the same procedure as for the convection term is applied:

$$p_b = p_i + \|\mathbf{d}_{\mathrm{N}}\| g_b. \tag{2.95}$$

  Fixed gradient contributes to the diagonal element $\mathbf{A}_{ii}$ and the right hand side as in Eqn. (2.94).

- **Velocity divergence**

  When a fixed value of velocity is prescribed on the boundary, the face value in discretisation of velocity divergence

$$\sum_f \mathbf{s}_f^T \mathbf{u}_f$$

is replaced with the boundary value $\mathbf{u}_f$. The contribution goes into the right hand side vector:

$$\mathbf{b}_i = \begin{bmatrix} b_{u_{xi}} \\ b_{u_{y_i}} \\ b_{u_{zi}} \\ b_{p_i} \end{bmatrix}. \tag{2.96}$$

If there is a velocity gradient g_b imposed on the boundary, the value of velocity is calculated by extrapolation:

$$\mathbf{u}_b = \mathbf{u}_i + \|\mathbf{d}_\mathrm{N}\| g_b. \tag{2.97}$$

This produces a contribution to the diagonal element and source term as in Eqn. (2.96):

$$\mathbf{A}_{ii} = \begin{bmatrix} a_{u_{xi},u_{xi}} & a_{u_{xi},u_{y_i}} & a_{u_{xi},u_{zi}} & a_{u_{xi},p_i} \\ a_{u_{y_i},u_{xi}} & a_{u_{y_i},u_{y_i}} & a_{u_{y_i},u_{zi}} & a_{u_{y_i},p_i} \\ a_{u_{zi},u_{xi}} & a_{u_{zi},u_{y_i}} & a_{u_{zi},u_{zi}} & a_{u_{zi},p_i} \\ a_{p_i,u_{xi}} & a_{p_i,u_{y_i}} & a_{p_i,u_{zi}} & a_{p_i,p_i} \end{bmatrix}. \tag{2.98}$$

- **Pressure Laplacian**

  The boundary conditions for the Laplacian of pressure are treated in the same way as for the diffusion term in the momentum equation. The following equation was obtained from discretisation procedure:

$$\sum_f \left(\frac{1}{a_{ii}}\right)_f \mathbf{s}_f^T (\nabla p)_f.$$

  For a fixed value of pressure on the boundary, the gradient of pressure on the face is calculated using the assigned value:

$$(\nabla p)_f = \frac{p_b - p_i}{\|\mathbf{d}_\mathrm{N}\|}, \tag{2.99}$$

  and the contributions are inserted into the right hand side vector Eqn. (2.96), and the diagonal matrix element:

$$\mathbf{A}_{ii} = \begin{bmatrix} a_{u_{xi},u_{xi}} & a_{u_{xi},u_{y_i}} & a_{u_{xi},u_{zi}} & a_{u_{xi},p_i} \\ a_{u_{y_i},u_{xi}} & a_{u_{y_i},u_{y_i}} & a_{u_{y_i},u_{zi}} & a_{u_{y_i},p_i} \\ a_{u_{zi},u_{xi}} & a_{u_{zi},u_{y_i}} & a_{u_{zi},u_{zi}} & a_{u_{zi},p_i} \\ a_{p_i,u_{xi}} & a_{p_i,u_{y_i}} & a_{p_i,u_{zi}} & a_{p_i,p_i} \end{bmatrix}. \tag{2.100}$$

When there is an imposed value of pressure gradient g_b on the boundary, it is directly used as the value of the gradient on the face and it produces a term on the right hand side, Eqn. (2.96).

2.4.10. Overview of the Implicitly Coupled Pressure–Velocity System

In this section, a summary of this chapter will be given, with an emphasis to Section 2.3.3. in which the derivation of the implicitly coupled pressure–velocity system was given, and sections 2.4.3.–2.4.9. where all the contributions to the linear system from the finite volume discretisation were presented. The governing equations, namely the incompressible, steady–state, turbulent, single–phase momentum and continuity equation are written in a block form:

$$\begin{bmatrix} \mathbf{A_u} & \nabla \\ \nabla\cdot & -\nabla\cdot(\mathbf{D_u^{-1}}\nabla) \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} 0 \\ -\nabla\cdot(\overline{\mathbf{D_u^{-1}}\nabla p^{(k-1)}}) \end{bmatrix},$$

where:

- $\mathbf{A_u}$ is the momentum matrix, which consists of the discretised convection and diffusion terms,
- ∇ is the pressure gradient matrix in the momentum equation,
- $\nabla\cdot$ is the velocity divergence in the pressure equation,
- $-\nabla\cdot(\mathbf{D_u^{-1}}\nabla)$ is the Laplacian of the pressure, obtained as an approximation of the Schur complement,
- $-\nabla\cdot(\overline{\mathbf{D_u^{-1}}\nabla p^{(k-1)}})$ is the explicit Rhie–Chow correction for counteracting the oscillations of the pressure field.

The linear system is written in cell–ordered manner to keep the mesh and matrix indexing consistent, and the sparsity pattern of the matrix is a consequence of

mesh connectivity (cell–to–cell–communication):

$$\begin{bmatrix} \mathbf{A}_{0,0} & & \dots & & \mathbf{A}_{0,n-1} \\ & \ddots & & & \\ \vdots & & \mathbf{A}_{ii} & \mathbf{A}_{ij} & \vdots \\ & & & \ddots & \\ \mathbf{A}_{n-1,0} & & \dots & & \mathbf{A}_{n-1,n-1} \end{bmatrix} \begin{bmatrix} \mathbf{x}_0 \\ \vdots \\ \mathbf{x}_i \\ \vdots \\ \mathbf{x}_{n-1} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_0 \\ \vdots \\ \mathbf{b}_i \\ \vdots \\ \mathbf{b}_{n-1} \end{bmatrix}.$$

$$\mathbf{A}_{ij} = \begin{bmatrix} a_{u_{xi},u_{xj}} & a_{u_{xi},u_{yj}} & a_{u_{xi},u_{zj}} & a_{u_{xi},p_j} \\ a_{u_{yi},u_{xj}} & a_{u_{yi},u_{yj}} & a_{u_{yi},u_{zj}} & a_{u_{yi},p_j} \\ a_{u_{zi},u_{xj}} & a_{u_{zi},u_{yj}} & a_{u_{zi},u_{zj}} & a_{u_{zi},p_j} \\ a_{p_i,u_{xj}} & a_{p_i,u_{yj}} & a_{p_i,u_{zj}} & a_{p_i,p_j} \end{bmatrix}, \quad \mathbf{x}_i = \begin{bmatrix} u_{xi} \\ u_{yi} \\ u_{zi} \\ p_i \end{bmatrix}, \quad \mathbf{b}_i = \begin{bmatrix} b_{u_{xi}} \\ b_{u_{yi}} \\ b_{u_{zi}} \\ b_{p_i} \end{bmatrix}.$$

The four unknowns are assembled into vectors, and for each cell there exists a single unknown vector. Thus, each cell is represented by a single block–matrix row. Dimensions of the matrix are $n \times n$, where n is the number of cells, while elements are matrices with dimensions $l \times l$, where $l = 4$ is the number of unknowns per cell. The position of the element in the matrix is marked by row and column index. Cell number i corresponds to row index, while column indices j of non–zero elements correspond to indices of neighbouring cells. Matrix elements are divided into three sets: diagonal ($i = j$, dark blue), upper ($i < j$, light blue) and lower ($i > j$, grey) triangle elements. The off–diagonal elements $\mathbf{A}_{ij}$ describe the influence of neighbouring cells j onto the solution in cell i. The structure of the block–elements stemms from the structure of the block system of the governing equations. The contribution of the individual terms to parts of the block system is summed up in Table 2.2. In general, the pressure part of the system (4^{th} row in the block element) is symmetric in pressure since it contains a Laplacian term. Additionaly, Laplacian gives a positive definite matrix with diagonal and off–diagonal elements of opposite sign and the sum of row elements is equal to 0, except the rows wich contain boundary conditions (diagonally dominant). The velocity divergence part is skew–symmetric (upper and lower off–diagonal elements with opposite signs), as well as the pressure gradient in the momentum equation. If the same discretisation scheme is used for velocity divergence and pressure gradient, it gives equal elements, i.e. the block–element is symmetric. The convection–diffusion part of the system contains the sum of elements from

Table 2.2: Contribution of finite volume discretisation schemes and boundary conditions to diagonal and off–diagonal matrix elements, and right hand side vector of the implicitly coupled pressure–velocity system.

diagonal $\mathbf{A}_{ii}$	**off–diagonal $\mathbf{A}_{ij}$**	**right hand side $\mathbf{b}_i$**
CONVECTION		
- central differencing - upwind -von Neumann b.c.	- central differencing (skew–symmetric) - upwind	- Dirichlet b.c. - von Neumann b.c.
DIFFUSION		
- orthogonal part - Dirichlet b.c.	- orthogonal part (symmetric)	- non-orthogonal correction - Dirichlet b.c. - von Neumann b.c.
PRESSURE GRADIENT		
- central differencing - least squares interpolation - Dirichlet b.c. - von Neumann b.c.	- central differencing (skew–symmetric) - least squares (symmetric)	- Dirichlet b.c. - von Neumann b.c.
VELOCITY DIVERGENCE		
- central differencing - von Neumann b.c.	- central differencing (skew–symmetric)	- Dirichlet b.c. - von Neumann b.c.
PRESSURE LAPLACIAN		
- orthogonal part - Dirichlet b.c.	- orthogonal part (symmetric)	- non-orthogonal correction - Dirichlet b.c. - von Neumann b.c. - Rhie-Chow correction

unsymmetric convection term and symmetric diffusion. If the standard linearisation of convection is used, there is no coupling between components of velocity in different directions.

2.5. Closure

In this chapter we have presented the possible options for the solution of the linear coupling of incompressible, steady state, single–phase, turbulent pressure and velocity equations. The derivation of the pressure equation as a Schur complement of the convection–diffusion matrix has been shown, as well as segregated and im-

plicitly coupled solution technique for the resulting linear system. A connection between the `OpenFOAM` mesh and the structure of the coefficient matrix was shown, while the last section of the chapter was dedicated to various contributions of the finite volume discretisation to the linear system. Since the resulting implicitly coupled pressure velocity system contains both the hyperbolic momentum equation and elliptic pressure equation, choosing the appropriate and efficient linear solver is not straightforward. Thus, we shall present and analyse the possible algorithms for solving the linear pressure–velocity system in the following chapter.

3. Linear Solvers

3.1. Introduction

In the previous chapter we presented the iterative algorithms for the solution of the pressure–velocity system, whether in segregated or implicitly coupled form. The nature of the equations in question has brought doubts about the optimal algorithm for the solution of the linear system. In this chapter, the state–of–the–art linear algorithms will be laid out and discussed. We shall begin with the most basic, fixed–point methods in section, whose drawbacks will lead to the complementary algebraic multigrid algorithm (AMG). Another important class of algorithms which are based on the Krylov subspace will be described: conjugate gradients (CG) and its extensions biconjugate gradients (BiCG) and biconjugate gradient stabilised (BiCGStab), as well as the generalised minimal residual method (GMRES). The complement to these algorithms in terms of matrix preconditioning will also be covered.

3.2. Algebraic Multigrid

In this section a general overview of the Algebraic Multigrid (AMG) linear solver will be given. Unlike the geometric multigrid, AMG solver does not operate on the computational mesh, but uses only the information in the coefficient matrix **A** of the linear system. However, it is much easier to think about multigrid in terms of the computational mesh, and the analogy between mesh and matrix will be used at various points in this chapter.

Two AMG coarsening strategies will be presented in detail for the scalar and block matrix: additive correction method (AAMG), also known as the agglomerative algebraic multigrid, and the selection method (SAMG). A description of multigrid smoothers and cycles will be given, as well as the application of the coarsening strategies to the block matrix. In the last section, parallelisation

strategies for SAMG will be presented and discussed.

3.2.1. Basic iterative solvers

The simplest choice for solving a sparse linear system $\mathbf{Ax} = \mathbf{b}$ is a class of so called *fixed–point* iterative methods such as the Jacobi and Gauss–Seidel algorithms, [2]. To clarify the idea of multigrid methods, Gauss–Seidel algorithm will be briefly presented.

Gauss–Seidel relies on splitting the coefficient matrix $\mathbf{A}$ into a sum of diagonal and off–diagonal parts. Additionaly, the off–diagonal part is split into a lower and upper triangular matrix:

$$\mathbf{A} = \mathbf{D} + \mathbf{U} + \mathbf{L}. \tag{3.1}$$

The linear system can then be written in the following form, using the distributive properties of matrix–vector multiplication:

$$(\mathbf{D} + \mathbf{U} + \mathbf{L})\mathbf{x} = \mathbf{b} \tag{3.2}$$

$$\mathbf{Dx} + \mathbf{Ux} + \mathbf{Lx} = \mathbf{b}. \tag{3.3}$$

The iterative procedure begins by choosing a forward or backward solution technique. The forward technique propagates from the first to last component of the solution (top to bottom), i.e. the components x_i of the solution which multiply the upper triangular matrix $\mathbf{U}$ are treated explicitly, using the previously calculated values:

$$\mathbf{x}^{\text{new}} = \mathbf{D}^{-1}(b_i - \mathbf{Lx}^{\text{new}} - \mathbf{Ux}^{\text{old}}). \tag{3.4}$$

Here, superscripts *old* and *new* correspond to the values in the current and previous iteration, respectively, while index i denotes the component of the solution vector. In the remainder of the chapter, *old* and *new* will be replaced by a bracketed number or letter, denoting the iteration of the algorithm. Indices denoting the components of a vector will be written in subscript, while the powers of vectors and matrices will be written in superscript with no bracket.

A backward technique of Gauss–Seidel goes in the opposite direction, from last to first component of the solution (bottom to top). The components of the

solution which multiply the lower triangular matrix are treated explicitly:

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(b_i - \mathbf{L}\mathbf{x}^{(k)} - \mathbf{U}\mathbf{x}^{(k+1)}). \tag{3.5}$$

The two directions of calculating the components of the solution vector can be used in sequence (symmetric sweep) and multiple times per iteration, which will be discussed further in Section 3.2.3..

The criterion for declaring the solution approximation good enough is defined using the *residual*:

$$\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}, \tag{3.6}$$

where $\mathbf{r}^{(k)}$ is the vector of residuals and $\mathbf{x}^{(k)}$ is the approximation of the solution in interation k of the iterative sequence. The number of iterations for finding an acceptable solution approximation depends on the initial solution $\mathbf{x}^{(0)}$ and the spectral properties (eigenvalues, eigenvectors) of the matrix $\mathbf{A}$. The residual is the best available indication of how far the current solution is from the correct value, because the error is unknown:

$$\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}, \tag{3.7}$$

where $\mathbf{e}^{(k)}$ is the error and $\mathbf{x}$ the correct solution of the system. However, the residual can be a misleading indication of convergence if the matrix is ill–conditioned, which will be discussed in the following sections. The relation between the error and the residual is:

$$\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)} = \mathbf{A}\mathbf{x} - \mathbf{A}\mathbf{x}^{(k)} = \mathbf{A}(\mathbf{x} - \mathbf{x}^{(k)}) = \mathbf{A}\mathbf{e}^{(k)}. \tag{3.8}$$

Eqn. (3.8) can be used for calculating the correction of the solution based on the residual:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{e}^{(k)}. \tag{3.9}$$

It was proven that Gauss–Seidel converges for strictly diagonally–dominant matrices (sum of the magnitudes of off–diagonal elements in a single row is smaller than the magnitude of the diagonal element) or symmetric and positive definite matrices [2]. If the matrix is diagonally equal (sum of magnitudes of off–diagonal elements is equal to the magnitude of diagonal element), the method converges so slowly that it is not practical. That is, the method is efficient if the component

being solved for (multiplying the diagonal element) has the largest contribution to the right hand side.

There are methods for improving the convergence of fixed–point algorithms, such as the Successive Over–Relaxation (SOR) [2], where a relaxation factor ω is used, $0 < \omega < 2$, to increase or decrease the influence of off–diagonal contributions to the currently calculated component of the solution. It is not used in the scope of this thesis and will not be further examined.

Since it will be widely used in this chapter, the analogy between the computational mesh and matrix has to be established. A computational mesh in OpenFOAM consists of 3D cells which are connected to their neigbouring cells through cell faces. Since the discretisation of linear(ised) equations is done for each cell, every cell is represented in a coefficient matrix by a single row. The index of the row corresponds to the index of the cell. The diagonal element represents the local contribution in the cell, while off–diagonal elements correspond to contributions from neighbouring cells. The column of an off–diagonal element is the same as the index of the neighbouring cell, while the magnitude of the off–diagonal element determines the strength of connection between two equations, i.e. the influence of a field value from a neigbouring cell. It was shown in Section 2.4.1. that matrices arising from the Finite Volume discretisation are sparse, meaning that a single component of the solution depends only on a few other solution components. The effect can be easily visualised in a computational mesh: the value in a cell will only be affected by the values in the closest neighbouring cells. Thus, fixed–point methods such as Gauss–Seidel, use only *local* information when calculating the solution, which is efficient if the matrix is strongly diagonally dominant. If the matrix is not strongly diagonally dominant, the convergence of the method will decrease because there is a greater influence from the neighbourhood, while global propagation of information is not possible.

In the context of *algebraic* multigrid, only the information from the matrix is used. It was even proposed by multigrid experts to change the name of the method to *multilevel* since no mesh (grid) is used.

Multigrid methods enable global propagation of information by creating a hierarchy of coarse matrices. Here, coarse denotes matrices which have a smaller dimension (fewer rows and columns). Equations which had no communication

in the fine matrix (off–diagonal element equal to zero), can become neighbours in the coarse matrix (off–diagonal element is non–zero) and thus expand the contribution of their local solution. The propagation of information on coarse levels is done by applying the aforementioned fixed–point methods, which are called *smoothers* in this context. The smoothing property will be discussed in the next section as well as the motivation for multigrid methods, based on the convergence behaviour of fixed–point algorithms.

A basic outline of an algebraic multigrid algorithm is given here, to establish the terminology. For simplicity, this algorithm is presented with only two levels (one fine and one coarse), while in practice multiple coarse levels are constructed.

1. On the *fine level*, calculate the approximate solution of the system:

$$\mathbf{A}^{\mathrm{F}}_{n\times n}\bar{\mathbf{x}}^{\mathrm{F}} = \mathbf{b}, \tag{3.10}$$

   where $\mathbf{A}^{\mathrm{F}}_{n\times n}$ is the coefficient matrix on the fine level, $n \times n$ denotes the matrix dimensions, i.e. the number of equations which are being solved, $\bar{\mathbf{x}}^{\mathrm{F}}$ is the approximate solution on the fine level, and $\mathbf{b}$ is the right hand side vector.

2. Using the obtained approximate solution, calculate the fine level residual $\mathbf{r}^{\mathrm{F}}$:

$$\mathbf{r}^{\mathrm{F}} = \mathbf{b} - \mathbf{A}^{\mathrm{F}}_{n\times n}\bar{\mathbf{x}}^{\mathrm{F}}. \tag{3.11}$$

3. Based on some criterion, create a coarse matrix $\mathbf{A}^{\mathrm{C}}_{m\times m}$, where m is the number of equations on the coarse level, and $m < n$. The procedure for choosing the equations which are being solved on the coarse level is called *coarsening*.

4. Transform the residual from fine level $\mathbf{r}^{\mathrm{F}}$ to coarse level $\mathbf{r}^{\mathrm{C}}$, using the *restriction matrix* $\mathbf{R}_{m\times n}$:

$$\mathbf{r}^{\mathrm{C}} = \mathbf{R}_{m\times n}\,\mathbf{r}^{\mathrm{F}}. \tag{3.12}$$

5. On the *coarse level*, solve the correction equation Eqn. (3.8):

$$\mathbf{A}^{\mathrm{C}}_{m\times m}\mathbf{e}^{\mathrm{C}} = \mathbf{r}^{\mathrm{C}}, \tag{3.13}$$

   where $\mathbf{e}^{\mathrm{C}}$ is the coarse level error.

6. Transform the error from coarse level $\mathbf{e}^{\mathrm{C}}$ to fine level $\mathbf{e}^{\mathrm{F}}$, using the *prolongation matrix* $\mathbf{P}_{n\times m}$:

$$\mathbf{e}^{\mathrm{F}} = \mathbf{P}_{n\times m}\,\mathbf{e}^{\mathrm{C}}. \tag{3.14}$$

7. Use the error calculated on the coarse level to correct the solution on the fine level:

$$\bar{\mathbf{x}}^{\mathrm{F}} = \bar{\mathbf{x}}^{\mathrm{F}} + \mathbf{e}^{\mathrm{F}}. \tag{3.15}$$

In the following sections we shall elaborate and expand the key points of the presented basic algorithm. Since this algorithm has only two levels, one fine and one coarse, we shall present the effect of introducing multiple coarse levels which is governed by multigrid cycle, section 3.2.2. An overview of smoothing properties of fixed–point algorithms was presented in this section, while the concept of algebraic smoothness with respect to matrix elements will be presented in Section 3.2.3.. In sections 3.2.4.–3.2.5. we shall present two strategies for the calculation of coarse level matrices and the corresponding restriction and prolongation matrices. Sections 3.2.6. and 3.2.7. are dedicated to application of the presented algorithms onto block–matrices and to parallelisation of selection algebraic multigrid, respectively.

3.2.2. Multigrid Cycle

In this section, an overview of a two level multigrid will be given as well as the reasons for using the V– and W–cycle. First, a multigrid matrix will be derived to interpret the effect of a single two level V–cycle onto the error.

A V–cycle with only two levels, fine and coarse, is shown in Fig. 3.1 with all the algebraic operations denoted at the appropriate positions:

FINE LEVEL

- Use a fixed–point method to calculate the approximate solution $\bar{\mathbf{x}}^{\mathrm{F}}$ of the linear system $\mathbf{A}^{\mathrm{F}}\mathbf{x} = \mathbf{b}$.
- Calculate the residual: $\mathbf{r}^{\mathrm{F}} = \mathbf{b} - \mathbf{A}^{\mathrm{F}}\bar{\mathbf{x}}^{\mathrm{F}} = \mathbf{A}^{\mathrm{F}}\bar{\mathbf{e}}^{\mathrm{F}}$.
- Restrict the residual: $\mathbf{r}^{\mathrm{C}} = \mathbf{R}\mathbf{r}^{\mathrm{F}}$.

COARSE LEVEL

- Solve the residual equation $\mathbf{A}^{\mathrm{C}}\mathbf{e}^{\mathrm{C}} = \mathbf{r}^{\mathrm{C}}$ to obtain $\mathbf{e}^{\mathrm{C}} = (\mathbf{A}^{\mathrm{C}})^{-1}\mathbf{r}^{\mathrm{C}}$.
- Prolongate the correction: $\mathbf{e}^{\mathrm{F}} = \mathbf{P}\mathbf{e}^{\mathrm{C}}$.

FINE LEVEL

- Correct the solution: $\mathbf{x}^{\mathrm{F}} = \bar{\mathbf{x}}^{\mathrm{F}} - \mathbf{e}^{\mathrm{F}}$.
- "Smooth out" the solution using a fixed–point method.

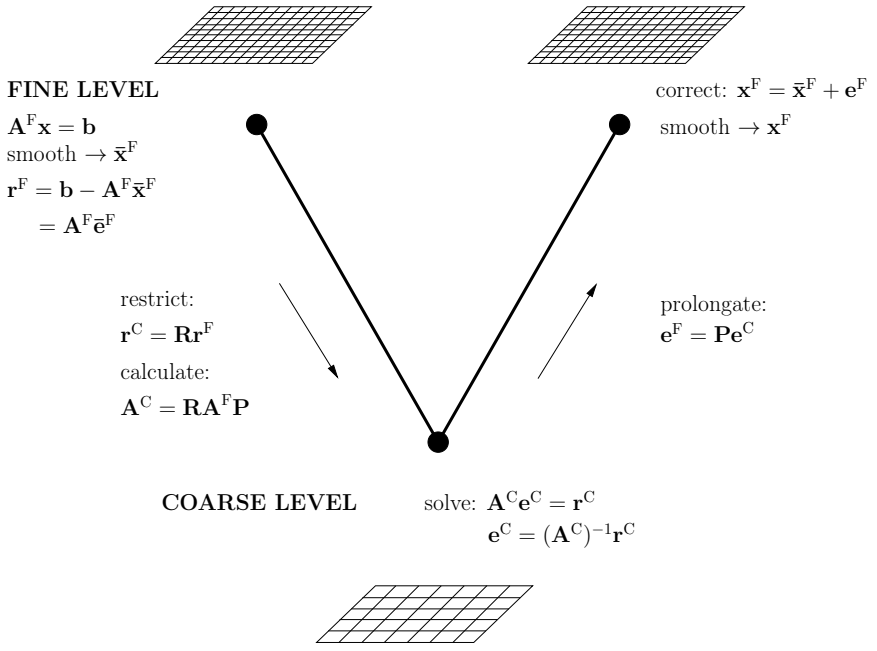


Figure 3.1: Two level multigrid V–cycle.

The multigrid matrix $\mathbf{M}$ can now be derived, using all the operations listed above, except the smoothing steps, [27]:

$$\underbrace{\underbrace{\underbrace{\mathbf{P}\,(\mathbf{A}^{\mathrm{C}})^{-1}\,\mathbf{R}}_{\text{prolongate correction}}\quad\underbrace{\underbrace{\mathbf{A}^{\mathrm{F}}\bar{\mathbf{e}}^{\mathrm{F}}}_{\text{fine level residual}}}_{\text{restrict residual}}}_{\text{solve coarse system}}}_{} = \mathbf{e}^{\mathrm{F}}. \tag{3.16}$$

Here, we can identify the multigrid matrix:

$$\mathbf{M} := \mathbf{P}(\mathbf{A}^{\mathrm{C}})^{-1}\mathbf{R}\mathbf{A}^{\mathrm{F}}, \tag{3.17}$$

where $\mathbf{A}^{\mathrm{C}}$ is the coarse level matrix defined as $\mathbf{A}^{\mathrm{C}} = \mathbf{R}\cdot\mathbf{A}^{\mathrm{F}}\cdot\mathbf{P}$. The correction equation in terms of the fine level error $\bar{\mathbf{e}}^{\mathrm{F}}$ is obtained from the correction of the solution:

$$\begin{aligned}
\mathbf{x}^{\mathrm{F}} &= \bar{\mathbf{x}}^{\mathrm{F}} - \mathbf{e}^{\mathrm{F}} \\
\cancel{\mathbf{x}^{\mathrm{F}}} &= \cancel{\mathbf{x}^{\mathrm{F}}} + \bar{\mathbf{e}}^{\mathrm{F}} - \mathbf{M}\bar{\mathbf{e}}^{\mathrm{F}} \\
\bar{\mathbf{e}}^{\mathrm{F}} - \mathbf{M}\bar{\mathbf{e}}^{\mathrm{F}} &= 0 \\
(\mathbf{I}-\mathbf{M})\bar{\mathbf{e}}^{\mathrm{F}} &= 0.
\end{aligned} \tag{3.18}$$

Thus, the correction term is scaled by matrix $\mathbf{M}$ and it is necessary to inspect its spectral properties to determine the effect of a single two level multigrid V–cycle on the convergence of the solution. The key is the definition of the coarse level matrix $\mathbf{A}^{\mathrm{C}}$ as a Galerkin matrix. If we insert it into the multigrid matrix:

$$\mathbf{M} = \mathbf{P}(\mathbf{R}\mathbf{A}^{\mathrm{F}}\mathbf{P})^{-1}\mathbf{R}\mathbf{A}^{\mathrm{F}},$$

and square it, the same multigrid matrix is obtained:

$$\begin{aligned}
\mathbf{M}^2 &= \mathbf{P}(\mathbf{R}\mathbf{A}^{\mathrm{F}}\mathbf{P})^{-1}\cancel{\mathbf{R}\mathbf{A}^{\mathrm{F}}\mathbf{P}(\mathbf{R}\mathbf{A}^{\mathrm{F}}\mathbf{P})^{-1}}\mathbf{R}\mathbf{A}^{\mathrm{F}} \\
&= \mathbf{P}(\mathbf{R}\mathbf{A}^{\mathrm{F}}\mathbf{P})^{-1}\mathbf{R}\mathbf{A}^{\mathrm{F}} \\
&= \mathbf{M}.
\end{aligned} \tag{3.19}$$

A conclusion can now be made about the eigenvalues of $\mathbf{M}$:

$$\begin{cases}
\mathbf{M}\mathbf{v} &= \lambda\mathbf{v} \\
\mathbf{M}^2\mathbf{v} &= \lambda^2\mathbf{v}.
\end{cases}$$

Since $\mathbf{M}^2 \equiv \mathbf{M}$:

$$\lambda\mathbf{v} = \lambda^2\mathbf{v}$$
$$\lambda^2 - \lambda = 0$$
$$(\lambda - 1)\lambda = 0.$$

The eigenvalues of matrix $\mathbf{M}$ are $\lambda_1 = 0$ and $\lambda_2 = 1$. According to Eqn. (3.18), the components of the error with the eigenvalue $\lambda_2 = 1$ will be completely eliminated, while multigrid will have no effect on the components with the eigenvalue $\lambda_1 = 0$. Looking at Eqn. (3.17) , it can be seen that the dimensions of $\mathbf{M}$ are the same as the dimension of the fine level matrix $\mathbf{A}^{\mathrm{F}}$. The rank of $\mathbf{M}$ (number of linearly independent columns, i.e. the number of non–zero rows in echelon form) is equal to the dimension of the coarse level matrix $\mathbf{A}^{\mathrm{C}}$. That is, the multiplicity of eigenvalue $\lambda_2 = 1$ is equal to the dimension of the coarse level matrix, as that is the number of error components which are completely eliminated on the fine level.

Since a single two level multigrid V–cycle has absolutely no effect on some components of the error, there is no point in doing multiple cycles. This is where fixed–point methods complement the multigrid two level cycle. Application of these methods, which smooth out the solution, is performed on the fine level system before the calculation of the residual (pre–smoothing) and after correction of the solution (post–smoothing) and the corresponding matrix is:

$$\underbrace{\mathbf{S}}_{\text{post–smoothing}} (\mathbf{I} - \mathbf{M}) \underbrace{\mathbf{S}}_{\text{pre–smoothing}}, \tag{3.20}$$

where $\mathbf{S}$ is the smoother. The smoothing step can be repeated multiple times, in attempt to further reduce the high–frequency components of the error. For example, the matrix for a multigrid two level V–cycle with two pre–smoothing and two post–smoothing sweeps is:

$$\mathbf{S}^2(\mathbf{I} - \mathbf{M})\mathbf{S}^2. \tag{3.21}$$

It is uncertain how many pre– and post–smoothing sweeps are optimal for convergence but it was shown that it is beneficial to recursively repeat the multigrid algorithm, i.e. create multiple coarse levels and move through them while repeating all the steps of a two level algorithm. The most used cycles are shown:

- V–cycle with several coarse levels, Fig. 3.2,
- W–cycle which is a V–cycle with multiple solutions on the coarser levels, Fig. 3.3,
- full multigrid cycle, which starts at the coarsest level and gradually moves towards the finest one, Fig. 3.4.

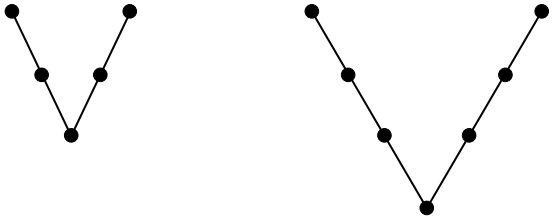


Figure 3.2: Multi–level multigrid V–cycle.

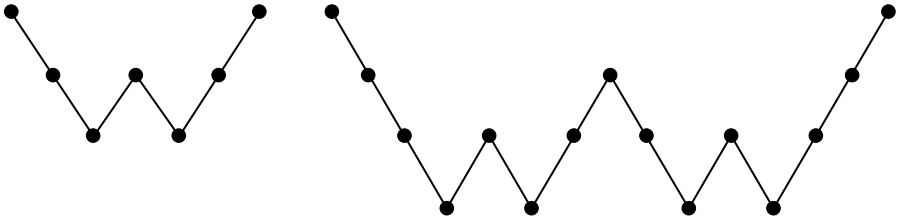


Figure 3.3: Multi–level multigrid W–cycle.

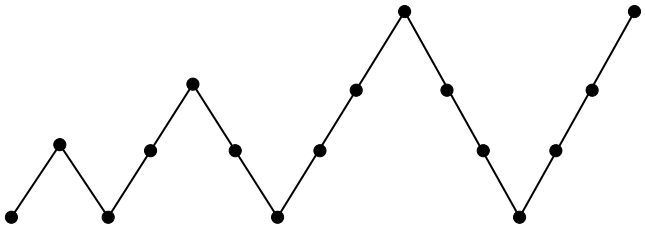


Figure 3.4: Full multigrid cycle.

In conclusion, the application of fixed–point methods to calculate the approximate solution of the linear system is limited by the eigenspectrum of the coefficient matrix. The error components which correspond to eigenvectors with very small eigenvalues are reduced very slowly, i.e. the residual is very small but the error is still quite large. Since these methods efficiently reduce some components of the error, and do not increase other components, the name smoothers

intuitively describes the effect. In the following section we shall describe how algebraic smoothness is related to application of fixed–point methods.

3.2.3. Algebraic Smoothness

In geometric multigrid, the error term on the fine mesh can be characterised as smooth if it can be well approximated on the coarse mesh (smoothness of a function). The definition of error *smoothness* in the context of algebraic multigrid is not the same, as the coarse mesh does not physically exist. In AMG, an error is called smooth if it cannot be eliminated, i.e. the chosen fixed–point method, see Section 3.2.1., stalls. The iterative solution of a linear system $\mathbf{Ax} = \mathbf{b}$ can be written in the following form:

$$\mathbf{x}^{(k+1)} = (\underbrace{\mathbf{I} - \mathbf{A}}_{\mathbf{S}})\mathbf{x}^{(k)} + \mathbf{b}, \tag{3.22}$$

where $\mathbf{I}$ is the identity matrix and $(\mathbf{I} - \mathbf{A})$ is the *iteration matrix* $\mathbf{S}$, i.e. the smoother. This equation represents a general fixed–point iteration: when the correct value of x is reached, term $-\mathbf{Ax} + \mathbf{b}$ vanishes and the solution does not change in the subsequent iteration. Thus, the correct solution of the system is called the *stationary point*. To analyse the effect of an iteration onto the solution, we shall write the current $\mathbf{x}^{(k+1)}$ and the previous solution $\mathbf{x}^{(k)}$ using the corresponding errors:

$$\begin{aligned}
(\mathbf{x} + \mathbf{e}^{(k+1)}) &= (\underbrace{\mathbf{I} - \mathbf{A}}_{\mathbf{S}})(\mathbf{x} + \mathbf{e}^{(k)}) + \mathbf{b} \\
&= \mathbf{x} + \mathbf{e}^{(k)} - \cancel{\mathbf{Ax}} - \mathbf{Ae}^{(k)} + \cancel{\mathbf{b}} \\
&= \mathbf{x} + \mathbf{e}^{(k)} - \mathbf{Ae}^{(k)} \\
&= \mathbf{x} + (\underbrace{\mathbf{I} - \mathbf{A}}_{\mathbf{S}})\mathbf{e}^{(k)}.
\end{aligned} \tag{3.23}$$

It can be seen from Eqn. (3.23) that the iteration matrix $\mathbf{S}$ does not affect the correct part of the solution $\mathbf{x}$, but only the error term $\mathbf{e}$. Finally, the fixed–point iteration expressed in terms of the solution error is:

$$\begin{aligned}
\mathbf{e}^{(k+1)} &= (\mathbf{I} - \mathbf{A})\mathbf{e}^{(k)} \\
&= \mathbf{e}^{(k)} - \mathbf{Ae}^{(k)}.
\end{aligned} \tag{3.24}$$

The condition for convergence of the fixed–point methods can be shown by assuming that the error term can be represented as a linear combination of eigenvectors of the coefficient matrix $\mathbf{A}$, [57]:

$$\mathbf{e} = \sum_{i}^{n} \zeta_i \cdot \mathbf{v}_i, \tag{3.25}$$

where subscript i denotes the i–th component of the error $\mathbf{e}$, ζ_i is a scalar constant describing the length of each component of $\mathbf{e}$, n is the dimension of the matrix and $\mathbf{v}_i$ is an eigenvector with the corresponding eigenvalue λ_i:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}. \tag{3.26}$$

The behaviour of eigenvectors is well–known, Fig. 3.5 and it can be used to analyse other vectors. For example, if a vector can be represented as a linear combination of eigenvectors, the component whose eigenvalue is larger than 1 will grow when the matrix is applied to it. If the corresponding eigenvalue of a component is smaller than 1, the repeated application of the matrix will "eliminate" it (it will become a null vector). Thus, to guarantee the convergence of a fixed–point iterative method, *all eigenvalues of the iteration matrix should be smaller than 1.* That is, the *spectral radius* $\rho(\mathbf{S})$ (the largest eigenvalue) of the iteration matrix should be smaller than 1. Looking at Eqn. (3.24), the goal is to eliminate the error to reach the correct solution, i.e. $\mathbf{e}^{(k+1)} = 0$, which gives:

$$\mathbf{e}^{(k)} - \mathbf{A}\mathbf{e}^{(k)} = 0. \tag{3.27}$$

Thus, the components of the error which have the corresponding eigenvalue closest to 1 will be eliminated by the fixed–point method the fastest. These error components are called high–frequency errors and were descibed in Section 3.2.1. as the components which can be eliminated locally (using the information from the neighbouring cells). The components which have small eigenvalues will be the slowest to converge, as the component $\zeta_j \cdot \mathbf{v}_j$ will not be reduced significantly by the corresponding scaled component of the term $\mathbf{A}\mathbf{e}^{(k)}$, and are called low–frequency errors. In the context of multigrid, methods which efficiently reduce the high–frequency components of the error are called *smoothers.* Smoothing can be visually explained for Jacobi and Gauss–Seidel: the error term is locally averaged using the available (local) information. Residual does not have to be a good

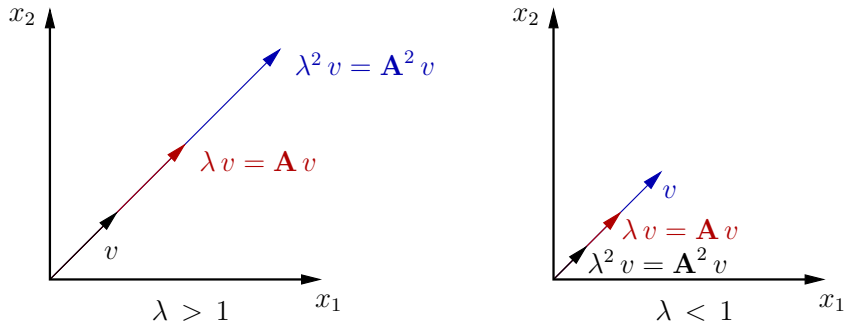


Figure 3.5: Scaling of eigenvectors with a matrix $\mathbf{A}$. The eigenvectors cannot rotate (except in the opposite direction, but they always lie on the same line), however, they can *contract* if the corresponding eigenvalue is smaller than 1, or *dilate* if the eigenvalue is larger than 1.

indication of the correct solution since a small residual doesn't necessarily imply a small error: we can recognise the residual as the second term in Eqn. (3.27), $\mathbf{r}^{(k)} = \mathbf{A}\mathbf{e}^{(k)}$. This is one of the definitions of the algebraically smooth error - the residual (error scaled by the coefficient matrix $\mathbf{A}$) is much smaller than the error itself [30].

The role of the algebraic multigrid algorithm is to identify the direction of low–frequency components of the error, turn them into high–frequency errors on the coarse mesh, and use the smoother to efficiently eliminate them.

To derive the starting point for the multigrid coarsening process, i.e. an equation for identification of the direction of low–frequency errors, recall Eqn. (3.4), describing the forward propagating iteration of the Gauss–Seidel method:

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(b_i - \mathbf{L}\mathbf{x}^{(k+1)} - \mathbf{U}\mathbf{x}^{(k)}),$$

which for each component x_i of the solution, can be written as:

$$\begin{aligned} x_i^{(k+1)} &= \frac{1}{a_{ii}}(b_i - \sum_{i \neq j} a_{ij} x_j^{(k)}) = \frac{1}{a_{ii}}(a_{ii} x_i^{(k)} + b_i - \sum_j a_{ij} x_j^{(k)}) \\ &= x_i^{(k)} + \frac{r_i^{(k)}}{a_{ii}}, \end{aligned} \tag{3.28}$$

where superscript $(k+1)$ indicates the solution in the current iteration, while other values are taken from previous k^{th} iteration. The equation can also be expressed for the corresponding error:

$$e_i^{(k+1)} = e_i^{(k)} - \frac{r_i^{(k)}}{a_{ii}}. \tag{3.29}$$

From the previous analysis, smooth (low–frequency) components of the error are characterised by slow reduction using the fixed–point method, i.e. the error does not significantly change between two iterations, $e_i^{(k+1)} \approx e_i^{(k)}$, and from that we can conclude that the second term in Eqn. (3.29) is negligible, i.e.:

$$\begin{aligned} |r_i| &\ll a_{ii}|e_i| \\ \left| a_{ii}e_i + \sum_{j \in \mathcal{N}_i} a_{ij}e_j \right| &\ll a_{ii}|e_i|. \end{aligned} \tag{3.30}$$

Eqn. (3.30) illustrates that the error can be approximated by the values of the error in the neighbouring cells:

$$a_{ii}e_i + \sum_{j \in \mathcal{N}_i} a_{ij}e_j = 0. \tag{3.31}$$

This equation is directly applied in the SAMG algorithm to describe the *algebraically smooth error* and derive the criterion for the coarsening process as well as the elements in restriction and prolongation matrices. The coherence between the multigrid algorithm and smoothers is based on Eqn. (3.31): since the equation originates from the limitations of the fixed–point methods, it is natural to exploit it for the definition of coarse levels, i.e. for the interpolation of the correction term from coarse level equations into the fine level equations.

Matrix Elements and Smoothness

Eqn. (3.27) describes the behavior of smooth errors: the corresponding residual is small, while the error itself remains large. Since the residual is actually the error transformed by the coefficient matrix $\mathbf{A}$, we shall introduce a unique measure of how the coefficient matrix affects vectors, i.e vector norms based on inner products [30]:

$$(\mathbf{u}, \mathbf{w})_{\mathbf{D}} = \mathbf{w}^T \mathbf{D} \mathbf{u}, \tag{3.32}$$

$$(\mathbf{u}, \mathbf{w})_{\mathbf{A}} = \mathbf{w}^T \mathbf{A} \mathbf{u}, \tag{3.33}$$

$$(\mathbf{u}, \mathbf{w})_{\mathbf{A}^T \mathbf{D}^{-1} \mathbf{A}} = (\mathbf{A}\mathbf{w})^T \mathbf{D}^{-1} \mathbf{A} \mathbf{u}, \tag{3.34}$$

where $\mathbf{u}$ an $\mathbf{w}$ are vectors, $(\bullet, \bullet)$ denotes a vector inner product, $\mathbf{A}$ is a positive definite matrix and $\mathbf{D}$ is the matrix containing only the diagonal of $\mathbf{A}$. The

corresponding vector norms are defined as:

$$\|\mathbf{z}\|_{\mathbf{D}} = (\mathbf{D}\mathbf{z}, \mathbf{z})^{\frac{1}{2}} = \sqrt{\mathbf{z}^T\mathbf{D}\mathbf{z}}, \tag{3.35}$$

$$\|\mathbf{z}\|_{\mathbf{A}} = (\mathbf{A}\mathbf{z}, \mathbf{z})^{\frac{1}{2}} = \sqrt{\mathbf{z}^T\mathbf{A}\mathbf{z}}, \tag{3.36}$$

$$\|\mathbf{z}\|_{\mathbf{A}^T\mathbf{D}^{-1}\mathbf{A}} = (\mathbf{D}^{-1}\mathbf{A}\mathbf{z}, \mathbf{A}\mathbf{z})^{\frac{1}{2}} = \sqrt{(\mathbf{A}\mathbf{z})^T\mathbf{D}^{-1}\mathbf{A}\mathbf{z}}. \tag{3.37}$$

The norm defined in Eqn. (3.36) is called the *energy norm* – a vector norm weighted by the coefficient matrix $\mathbf{A}$. It will also be significant in the context of Krylov subspace methods. The norm in Eqn. (3.35) is weighted by the diagonal of matrix $\mathbf{A}$, while the norm in Eqn. (3.37) is weighted by a scaled matrix $\mathbf{A}$, where all diagonal elements are equal to 1. A smooth error is not affected by the smoother, i.e. the energy norm of the error remains the same [2]:

$$\|\mathbf{S}\mathbf{e}\|_{\mathbf{A}} \approx \|\mathbf{e}\|_{\mathbf{A}}. \tag{3.38}$$

To derive an equation which defines the *smoothing property* of an iterative metod, we will use the set of norms in Eqn. (3.35)–Eqn. (3.37) on the eigenvectors $\mathbf{v}$ of the matrix $\mathbf{D}^{-1}\mathbf{A}$ [30], i.e. matrix $\mathbf{A}$ with diagonal elements equal to 1:

$$\mathbf{D}^{-1}\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

$$\|\mathbf{v}\|_{\mathbf{D}}^2 = \left(\sqrt{\mathbf{v}^T\mathbf{D}\mathbf{v}}\right)^2 = \mathbf{v}^T\mathbf{D}\mathbf{v}, \tag{3.39}$$

$$\|\mathbf{v}\|_{\mathbf{A}}^2 = \left(\sqrt{\mathbf{v}^T\mathbf{A}\mathbf{v}}\right)^2 = \mathbf{v}^T\mathbf{D}\underbrace{\mathbf{D}^{-1}\mathbf{A}\mathbf{v}}_{\lambda\mathbf{v}} = \lambda\|\mathbf{v}\|_{\mathbf{D}}^2, \tag{3.40}$$

$$\|\mathbf{v}\|_{\mathbf{A}^T\mathbf{D}^{-1}\mathbf{A}}^2 = \left(\sqrt{(\mathbf{A}\mathbf{v})^T\mathbf{D}^{-1}\mathbf{A}\mathbf{v}}\right)^2 = (\mathbf{A}\mathbf{v})^T\underbrace{\mathbf{D}^{-1}\mathbf{A}\mathbf{v}}_{\lambda\mathbf{v}} = \lambda\|\mathbf{v}\|_{\mathbf{A}}^2, \tag{3.41}$$

where λ is an eigenvalue of $\mathbf{D}^{-1}\mathbf{A}$ corresponding to eigenvector $\mathbf{v}$. From Eqn. (3.27) we have seen that the components of the error which correspond to the smallest eigenvalues ($\lambda \approx 0$) will be the slowest to converge. Comparing the three norms using Eqn. (3.39)–Eqn. (3.41) reveals that they are very different in size if the eigenvalues are close to 0:

$$\|\mathbf{v}\|_{\mathbf{A}^T\mathbf{D}^{-1}\mathbf{A}} \ll \|\mathbf{v}\|_{\mathbf{A}} \ll \|\mathbf{v}\|_{\mathbf{D}}. \tag{3.42}$$

On the contrary, when applied to a high–frequency error ($\lambda \approx 1$), the three norms are similar. A smoothing property of a matrix is defined using the behaviour of these norms for smooth and oscillatory error [30]:

Definition 3.1. *A smoothing matrix* $\mathbf{S}$ *satisfies a smoothing property with respect to a positive definite matrix* $\mathbf{A}$ *if for all vectors* $\mathbf{e}$

$$\|\mathbf{Se}\|_{\mathbf{A}}^2 \leq \|\mathbf{e}\|_{\mathbf{A}}^2 - \sigma\|\mathbf{e}\|_{\mathbf{A}^T\mathbf{D}^{-1}\mathbf{A}}^2 \tag{3.43}$$

where $\sigma > 0$*, and* σ *is independent of* $\mathbf{e}$*.*

To demonstrate how the connectivity affects the smoothness, we will rewrite the inner product defined in Eqn. (3.33) in terms of matrix elements [2]:

$$\begin{aligned}
(\mathbf{e},\mathbf{e})_{\mathbf{A}} &= \mathbf{e}^T\mathbf{Ae} \\
&= \sum_{i,j} a_{ij}e_ie_j \\
&= \frac{1}{2}\sum_{i,j} -a_{ij}\left[(e_i-e_j)^2 - e_i^2 - e_j^2\right] \\
&= \frac{1}{2}\sum_{i,j} -a_{ij}(e_i-e_j)^2 + \underbrace{\frac{1}{2}\sum_{i,j} a_{ij}\left(e_i^2+e_j^2\right)}_{(\mathbf{A}\text{ is symmetric!})} \\
&= \frac{1}{2}\sum_{i,j} -a_{ij}(e_j-e_i)^2 + \sum_i\left(\sum_j a_{ij}\right)e_i^2.
\end{aligned} \tag{3.44}$$

The inequality of norms from Eqn. (3.42) can be rewritten as:

$$\begin{aligned}
\|\mathbf{e}\|_{\mathbf{A}}^2 &= \epsilon\|\mathbf{e}\|_{\mathbf{D}}^2, \\
\mathbf{e}^T\mathbf{Ae} &= \epsilon\cdot\mathbf{e}^T\mathbf{De},
\end{aligned} \tag{3.45}$$

where $0 < \epsilon \ll 1$. If we consider a special case when the row sums of matrix $\mathbf{A}$ are 0 (the sign of the diagonal element is opposite to signs of off–diagonal elements and rows are diagonally equal), Eqn. (3.45) can be rewritten using Eqn. (3.44):

$$\frac{1}{2}\sum_{i,j}|a_{ij}|(e_j-e_i)^2 + \sum_i \underbrace{\left(\sum_j a_{ij}\right)}_{0} e_i^2 = \epsilon\sum_i a_{ii}e_i^2$$

$$\frac{1}{2}\sum_{i,j}|a_{ij}|(e_j-e_i)^2 = \epsilon\sum_i a_{ii}e_i^2 \tag{3.46}$$

$$\sum_i a_{ii}e_i^2\left[\sum_{j\neq i}\frac{|a_{ij}|}{a_{ii}}\left(\frac{e_i-e_j}{e_i}\right)^2 - 2\epsilon\right] = 0. \tag{3.47}$$

For Eqn. (3.47) to hold, the term in the square brackets must vanish, thus it is expected that the first part of the term is of order 2ϵ [2], that is:

$$\sum_{j \neq i} \frac{|a_{ij}|}{a_{ii}} \left(\frac{e_i - e_j}{e_i} \right)^2 \ll 1. \tag{3.48}$$

Thus, if $|a_{ij}|/a_{ii}$ is large, $(e_i - e_j)/e_i$ must be small. That is, the *components of the error change slowly in the direction of large off–diagonal elements with sign opposite to diagonal element.* To see the behaviour of the error for large off–diagonal elements with sign equal to the diagonal, a measure of diagonal dominance t_i, for each matrix row i is introduced [30]:

$$t_i = a_{ii} - \sum_{j \neq i} a_{ij}. \tag{3.49}$$

We shall reformulate Eqn. (3.44) using the measure of diagonal dominance to rewrite the i^{th} row sum

$$\sum_j a_{ij} = \underbrace{a_{ii} - \sum_{j \neq i} |a_{ij}|}_{t_i} + 2 \sum_{j \neq i} a_{ij}^+,$$

where a_{ij}^+ are elements with sign equal to the sign of the diagonal element ("positive"). Also, the first sum in Eqn. (3.44) can be split into sums of diagonal elements and "positive" and "negative" off–diagonal elements:

$$\begin{aligned} \sum_{i,j} -a_{ij}(e_j - e_i)^2 = & \underbrace{\sum_{i=j} -a_{ij}(e_j - e_i)^2}_{0} + \sum_i \sum_{j \neq i} -a_{ij}^+ (e_j - e_i)^2 \\ & + \sum_i \sum_{j \neq i} -a_{ij}^- (e_j - e_i)^2. \end{aligned} \tag{3.50}$$

Putting it all together yields:

$$\begin{aligned}
\mathbf{e}^T\mathbf{A}\mathbf{e} &= \frac{1}{2}\sum_{i,j} -a_{ij}(e_j - e_i)^2 + \sum_i \left(t_i + 2\sum_{j\neq i} a_{ij}^+\right) e_i^2 \\
&= \frac{1}{2}\left[\sum_i \sum_{j\neq i} -a_{ij}^-(e_j - e_i)^2 + \sum_i \sum_{j\neq i} -a_{ij}^+(e_j - e_i)^2\right] \\
&\quad + \sum_i \sum_{j\neq i} 2a_{ij}^+ e_j + \sum_i t_i e_i^2 \\
&= \frac{1}{2}\left[\sum_i \sum_{j\neq i} -a_{ij}^-(e_j - e_i)^2 + \sum_i \sum_{j\neq i} -a_{ij}^+(e_j - e_i)^2\right] \\
&\quad + \sum_i \sum_{j\neq i} a_{ij}^+ e_j^2 + a_{ij}^+ e_i^2 + \sum_i t_i e_i^2 \\
&= \frac{1}{2}\sum_i \sum_{j\neq i} -a_{ij}^-(e_j - e_i)^2 + \sum_i t_i e_i^2 \\
&\quad + \sum_i \sum_{j\neq i} \frac{1}{2}(-a_{ij}^+)(e_j - e_i)^2 + a_{ij}^+ e_j^2 + a_{ij}^+ e_i^2 \\
&= \frac{1}{2}\sum_i \sum_{j\neq i} -a_{ij}^-(e_j - e_i)^2 + \sum_i t_i e_i^2 + \frac{1}{2}\sum_i \sum_{j\neq i} a_{ij}^+(e_j + e_i)^2. \qquad (3.51)
\end{aligned}$$

The corresponding norm is equal to:

$$\|\mathbf{e}\|_{\mathbf{A}}^2 = \frac{1}{2}\sum_i \left[\sum_{j\neq i} |a_{ij}^-|(e_j - e_i)^2 + \sum_{j\neq i} a_{ij}^+(e_j + e_i)^2\right] + \sum_i t_i e_i^2. \qquad (3.52)$$

For a weakly diagonally dominant matrix ($t_i \approx 0$) comparison of norms according to Eqn. (3.42) yields:

$$\begin{aligned}
\|\mathbf{e}\|_{\mathbf{A}} &\ll \|\mathbf{e}\|_{\mathbf{D}}, \\
\sum_{j\neq i} |a_{ij}^-|(e_j - e_i)^2 + \sum_{j\neq i} a_{ij}^+(e_j + e_i)^2 &\ll a_{ii} e_i^2, \\
\sum_{j\neq i} \frac{|a_{ij}^-|}{a_{ii}} \frac{(e_j - e_i)^2}{e_i^2} + \sum_{j\neq i} \frac{a_{ij}^+}{a_{ii}} \frac{(e_j + e_i)^2}{e_i^2} &\ll 1. \qquad (3.53)
\end{aligned}$$

It follows from Eqn. (3.53): if a_{ij} is "positive" (has the sign equal to diagonal element), and if connection a_{ij}/a_{ii} is large, e_j will be close to $-e_i$, i.e. the error in that direction oscillates. Thus, the *algebraically smooth error tends to oscillate along strong positive connections* [30].

3.2.4. Additive Correction Algebraic Multigrid

Multigrid algorithms create a hierarchy of coarse matrices using the projection methods to efficiently eliminate components of the error corresponding to coarse level unknowns. Fixed–point methods complement the multigrid algorithm by reducing fine level components of the error and the compatibility is achieved through the construction of the prolongation matrix and the definition of the coarse level matrices as Galerkin matrices. For simplicity, the coarsening strategies presented in the current and the following section will be described in a cycle with only two multigrid levels (fine and coarse), although multiple coarse levels are usually used. In this section, we shall present a heuristic algorithm for construction of coarse level and transfer matrices.

It was shown in sections 3.2.2. and 3.2.2. that fixed–point iterative solvers reduce the high–frequency errors, i.e. the errors whose components correspond to eigenvectors with eigenvalues close to 1. That is, the difficulty in convergence arises from the non–uniformity of matrix elements. To illustrate, we shall use a non–uniform mesh shown in Fig. 3.6 on which we attempt to solve a discretised diffusion equation, where diffusivity is equal to 1:

$$\nabla_{\bullet}(\nabla\phi) = 0.$$

It was shown in Chapter 2. that the discretised diffusion operator is written as a sum over the faces of a finite volume cell and the equation elements are proportional to the face surface area. The cells at the fine level in Fig. 3.6 are elongated in x–direction, thus there appear matrix elements with different magnitudes, which stem from the difference in the face surface area. The solution from neighbouring cell j influences the solution in cell i proportionally to the matrix element a_{ij}. Thus, the influence from a greater face surface area is much stronger – in this case the strong influence exists in y–direction due to anisotropy of the cells, which is the direction in which the information spreads quickly during the fixed–point iteration.

The motivation for Additive Correction Algebraic Multigrid (AAMG), proposed by Hutchinson and Raithby [33] in the context of FVM, is to even out the face surface area, similar to principles in geometric multigrid, by grouping the fine level cells in the direction of strong connections to create cell *clusters*

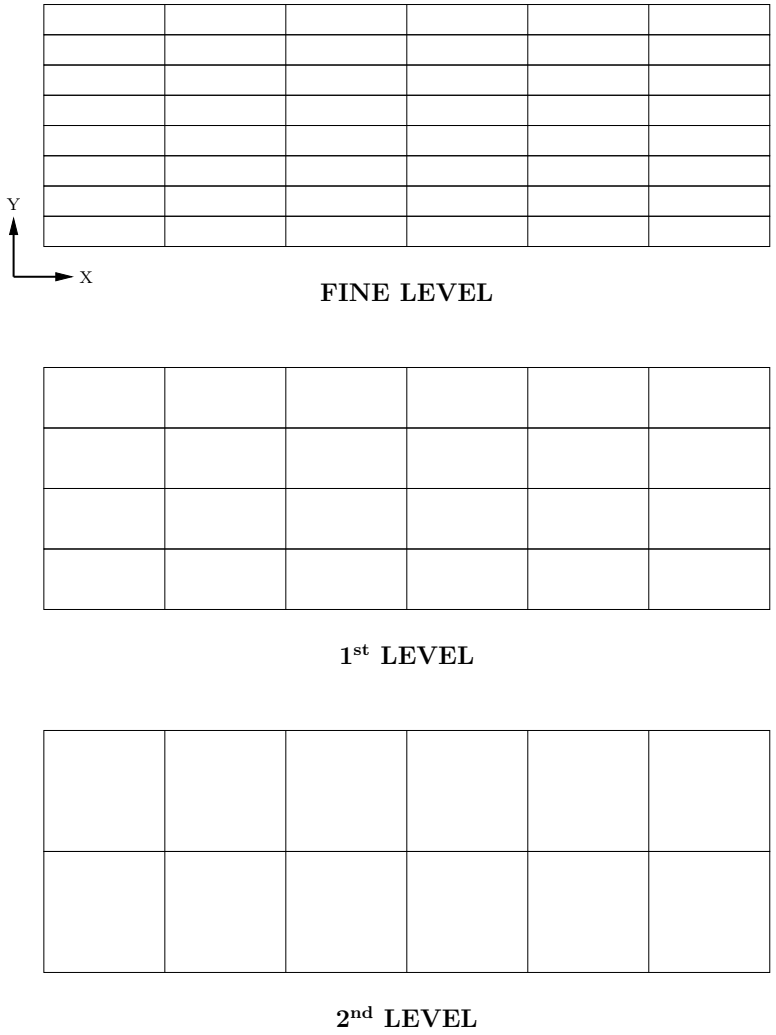


Figure 3.6: Coarsening in AAMG: grouping of the cells eliminates the anisotropy of the mesh.

which will represent coarse level cells. An example is shown in Fig. 3.6, where uniformity was achieved on the 2^{nd} multigrid level, after grouping two fine level cells into one coarse level cell.

Since the approach is purely algebraic, coarse level equations have to be constructed from the fine level matrix, without any knowledge about the computational mesh or corresponding boundary conditions. The construction of the coarse matrix is based on the conservativeness of the FVM equations.

Navier–Stokes equations are conservative by nature, i.e. the momentum in a finite volume cell can only be changed by the flow through the face or by the surface or volumetric forces acting on the cell. The mass conservation in the

continuity equation follows the same principles. The conservation property is valid for the FVM discretised equations as well. An integral over the whole domain is equal to the sum of integrals over the control volumes and it can be reduced to a sum over the surface of the domain. That is, if a property is conserved over a single finite volume cell, it has to be conserved over a group of finite volume cells. Thus, the correction calculated on the coarse level which will be applied onto fine level solution, must preserve the conservativeness over a cluster of cells. The derivation given in [33] starts from a general conservation law in algebraic form:

$$a_{ii}x_i + \sum_j a_{ij}x_j = b_i. \tag{3.54}$$

It is presumed that the final fine level solution is obtained after a correction from coarse level is applied:

$$x_i^{\mathrm{F}} = \tilde{x}_i^{\mathrm{F}} + e_{I/i}^{\mathrm{F}}, \tag{3.55}$$

where $\tilde{x}_i^{\mathrm{F}}$ is the uncorrected (initial) solution on the fine level, x_i^{F} is the fine level solution after correction, while $e_{I/i}^{\mathrm{F}}$ is the correction calculated for coarse cell I, which the fine level cell i was grouped into.

Since it is expected that the residual on the coarse level will be reduced to zero for each cluster, to keep the conservativeness on the fine level, the sum of the residual over a single cluster should be zero:

$$\sum_{i\,\mathrm{in}\,I} r_i^{\mathrm{F}} = 0, \tag{3.56}$$

where r_i^{F} is the residual obtained after correction on the fine level. The residual after correction is calculated as:

$$\begin{aligned} r_i^{\mathrm{F}} &= b_i - (a_{ii}x_i^{\mathrm{F}} + \sum_j a_{ij}x_j^{\mathrm{F}}) \text{ (insert expression for } x_i^{\mathrm{F}}), \\ r_i^{\mathrm{F}} &= b_i - \underbrace{(a_{ii}\tilde{x}_i^{\mathrm{F}} + \sum_j a_{ij}\tilde{x}_j^{\mathrm{F}})}_{\tilde{r}_i^{\mathrm{F}}} - (a_{ii}e_{I/i}^{\mathrm{F}} + \sum_j a_{ij}e_{I/j}^{\mathrm{F}}), \end{aligned} \tag{3.57}$$

where $\tilde{r}_i^{\mathrm{F}}$ is the uncorrected (initial) residual in the fine level matrix. Inserting Eqn. (3.57) into Eqn. (3.56) yields:

$$\sum_{i\,\mathrm{in}\,I} r_i^{\mathrm{F}} = \sum_{i\,\mathrm{in}\,I} \left[\tilde{r}_i^{\mathrm{F}} - (a_{ii}e_{I/i}^{\mathrm{F}} + \sum_j a_{ij}e_{I/j}^{\mathrm{F}}) \right] = 0,$$

$$\sum_{i \,\mathrm{in}\, I} (a_{ii} e^{\mathrm{F}}_{I/i} + \sum_{j} a_{ij} e^{\mathrm{F}}_{I/j}) = \sum_{i \,\mathrm{in}\, I} \tilde{r}^{\mathrm{F}}_{i}. \tag{3.58}$$

Eqn. (3.58) represents the coarse level equation in AAMG and it reveals that the coarse level matrices are obtained by summing the fine level matrix elements which correspond to a single cluster, while the right hand side is obtained by summing the fine level residual. Thus, addition of elements in the direction of strong connections is equivalent to enforcing the integral equation of a conservation law over a cluster which consists of the corresponding fine equations. After the addition of correction on the fine level, the conservativeness is satisfied globally (due to consistent sum of residuals). Local conservation (residual equal to zero in each cell which belongs to a cluster) is achieved by the fixed–point method, which efficiently eliminates the error inside a cluster.

As shown for the example in Fig. 3.6, the approach is straightforward for a structured computational mesh, and since the construction of coarse levels is purely algebraic, it can efficiently be applied to systems stemming from unstructured meshes.

AAMG algorithm implemented in `foam-extend` begins by determining the strength of connectivity in the fine level matrix, i.e. searching for large off–diagonal elements. This part of the algorithm is highly sequential. A strong connection between two equations exists, if the magnitude of the off–diagonal element squared, which represents the connection of two cells, is larger than the scaled product of the two diagonal elements. Thus, the criterion for declaring a strong connection between equations k and m is:

$$a^2_{km} > \alpha_{\mathrm{MG}} \cdot a_{kk} a_{mm}, \tag{3.59}$$

where a_{km} is the off–diagonal element representing the connection between equations k and m, and a_{kk} and a_{mm} are diagonal elements. α_{MG} is a chosen scaling factor. There are special cases when the equation has no strong connections. These equations are sorted into a so–called *zero cluster*, and their correction is calculated separately. After determining the strong connections for each equation, the first equation, called the *seed* is chosen based on the smallest index (lexicographically). Equations coupled to the seed are added to the cluster if the connection is strong enough, based on Eqn. (3.59). The size of a cluster is limited

by the user: a maximum number of equations which can be grouped into a single cluster is prescribed. If the cluster is not full after serching the coupled equations, couplings of already grouped coupled equations are then searched. After all the equations have been grouped into clusters, each cluster is marked with an index, in our case, in order of appearance of the cluster. The index of the cluster corresponds to the index of the equation in the coarse matrix.

As shown in Eqn. (3.58), the coarse level matrix is obtained by summing up the elements from the fine matrix.

The dimensions of the matrix for transferring the residual from fine level, i.e. restriction matrix $\mathbf{R}$ are equal to *total number of clusters* $\times$ *number of cells on the fine level*. The elements of the restriction matrix are either 0 or 1:

- if equation m belongs to a cluster I, the element in row I and column m of the restriction matrix is equal to 1,
- otherwise, the element in the restriction matrix is equal to 0.

There is no need for explicit formulation of the restriction matrix, since the residual on the coarse level is obtained as the sum of residuals on the fine level:

$$r_i^{\mathrm{C}} = \sum_{i \in I} r_i^{\mathrm{F}}. \tag{3.60}$$

Prolongation matrix $\mathbf{P}$ has the dimensions equal to *number of cells on the fine level* $\times$ *total number of clusters* and its elements are also either 0 or 1:

- if equation m belongs to a cluster I, the element in row m and column I of the prolongation matrix is equal to 1,
- otherwise, the element in the prolongation matrix is equal to 0.

Again, there is no explicit formation of the prolongation matrix, as the correction calculated for each cluster is directly applied to all the fine level equations which were grouped into that particular cluster:

$$x_i = \bar{x}_i^{\mathrm{F}} + e_{i \in I}^{\mathrm{C}}. \tag{3.61}$$

3.2.5. Selection Algebraic Multigrid

The second coarsening strategy is the Selection Algebraic Multigrid (SAMG) [28], in which the coarsening is achieved by selecting representative equations from the fine level matrix, i.e. choosing the representative cells from the fine mesh. On the coarse level, the equations are solved only for the selected cells. The selection of the cells is based on a heuristic procedure which, similar to AAMG, depends on the strength of connectivity in the coefficient matrix $\mathbf{A}$, i.e. the magnitude of off–diagonal elements. Here, in contrast to AAMG, the restriction and prolongation matrices are explicitly formed. For clarity, SAMG algorithm can be divided into separate phases:

1. *Selecting representative equations* in the fine level matrix;
2. Calculating the *scaling factors* to take into account the eliminated equations;
3. Calculating the *prolongation weights*;
4. Forming the *prolongation and restriction* matrices;
5. Calculating the *coarse level matrix*.

Each phase will be described in detail in the following sections.

Selection of representative equations

For easier understanding, the terminology used in this section will be introduced. For a single equation, the set of strongly coupled equations can be divided into two subsets, *influences* and *dependencies*, illustrated in Fig. 3.7:

- In a matrix row i, if there exists an off–diagonal element a_{ij} larger than some value, the equation with the column index j is an influence of equation i.
- In a matrix column j, if there exists an off–diagonal element a_{ij} larger than some value, equation with the row index i is a dependency of the equation with index of the column j.

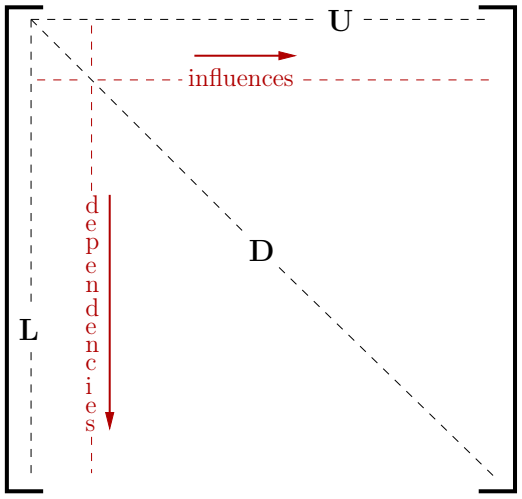


Figure 3.7: The direction of influences and dependencies in a coefficient matrix.

For a symmetric matrix, the number of influences and dependencies of an equation is equal. Also, for reasons mentioned in Section 3.2.3., it is important to distinguish between the signs of matrix elements:

- In a matrix row, if there exists an off–diagonal element with the sign *opposite* to the sign of the diagonal element in that row, the connection is called *negative*.
- In a matrix row, if there exists an off–diagonal element with the sign *equal* to the sign of the diagonal element in that row, the connection is called *positive*.

The task of the selection algorithm is to form a subset of *coarse level equations* from the set of *fine level equations*. The equations which were not selected into the coarse subset will receive the coarse correction by interpolation, which implies that the eliminated equations must be well represented by the chosen subset on the coarse level. To ensure optimal performance of the solver, the goal is to find the largest possible set of equations which will remain on the coarse level in such a way that they do not depend on each other. The algorithm for selection of the coarse subset is taken from Ruge and Stüben [28]. The first step is to find all negative elements a_{ij} in row i of the coefficient matrix. The element with the largest magnitude is chosen to be the *strongest* negative connection a_{ij}^S in row

i. All negative elements a_{ij}^{-} which are larger than the fraction of the strongest negative connection:

$$a_{ij}^{-} > \beta_{\mathrm{MG}} \cdot a_{ij}^{\mathrm{S}}, \text{ where } \beta_{\mathrm{MG}} < 1, \tag{3.62}$$

represent an influence from equation j onto equation i. Based on the same criterion, all negative connections a_{ji}^{-} in row j represent a dependency of equation i. After finding influences and dependencies for each row, an auxiliary matrix containing only these strong connections is formed. Based on the auxiliary matrix, the equations are ranked. Each equation is assigned a weight which is equal to the number of dependencies for that equation. The equation with the largest weight becomes coarse and all its dependencies are eliminated from the coarsening procedure, i.e. they only exist on the fine level. Weights of the unsorted equations are then updated: weights of neighbours of the selected coarse equation are decremented (they have less chance to become coarse). Weights of influences of the eliminated fine equations are incremented (to increase the chance of them becoming coarse). The process continues with the next unsorted equation which has the largest weight and is repeated until all the equations are either chosen as coarse or eliminated as fine.

The final division of equations should satisfy the condition that every fine equation has at least one influence in the coarse subset from which it will receive the coarse level correction. However, in some cases, there will be equations without any influences (for example, if all connections are positive). These equations will automatically be eliminated, and solved only on the fine level. It would be ideal if all the influences of a fine equation become coarse level equations (local maximum), but in general, this will not be the case. These eliminated influences will be treated in a special way which will be described in the following section.

Scaling factors and prolongation weights

After separating the equations into two sets, coarse and fine, it is necessary to define how the correction calculated on the coarse level will be applied to fine level equations. The idea is that every fine equation will receive a coarse level correction from its coarse influences (neighbours) through weighted interpolation. The interpolation weights will be assembled into the prolongation matrix. The

starting point for calculating the weights is the residual equation, Eqn. (3.8), for a single matrix row, split into diagonal and off–diagonal part, in which the residual is completely eliminated ($r_i = 0$), i.e. Eqn. (3.31):

$$a_{ii}e_i + \sum_{j \in \mathcal{N}_i} a_{ij}e_j = 0,$$

where a_{ii} is the diagonal element in row i, e_i is the error corresponding to row i, $\mathcal{N}_i$ is the set of all the equations coupled to equation i, a_{ij} is the off–diagonal element and e_j is the error corresponding to solution of equation j.

In Section 3.2.4., it was said that equations coupled to equation i can be either coarse or fine, and it would be ideal if all the influences of a fine equation are coarse. However, there will be some influences which are fine. These fine influences will be specially treated and put into a special subset. Thus, all equations j coupled to equation i in set $\mathcal{N}_i$ are divided into subsets:

- *Coarse influences* $\mathcal{C}_i$ of equation i;
- *Fine influences* $\mathcal{F}_i^{\mathrm{S}}$ of equation i. Superscript S denotes the strong connection of the two equations;
- *Other fine neighbours* $\mathcal{F}_i^{\mathrm{W}}$ of equation i. Superscript W denotes the weak connection between the two equations. Usually, only positive connections appear in this subset.

Using this classification, Eqn. (3.31) can be written in the following form:

$$a_{ii}e_i = -\sum_{j \in \mathcal{C}_i} a_{ij}e_j - \sum_{j \in \mathcal{F}_i^{\mathrm{S}}} a_{ij}e_j - \sum_{j \in \mathcal{F}_i^{\mathrm{W}}} a_{ij}e_j. \tag{3.63}$$

The subset of weak connections $\mathcal{F}_i^{\mathrm{W}}$ is collapsed and added to the diagonal element:

$$\left(a_{ii} + \sum_{j \in \mathcal{F}_i^{\mathrm{W}}} a_{ij}\right) e_i = -\sum_{j \in \mathcal{C}_i} a_{ij}e_j - \sum_{j \in \mathcal{F}_i^{\mathrm{S}}} a_{ij}e_j. \tag{3.64}$$

To derive the equation for calculating interpolation weights, consider an equation eliminated on the fine level which has only one coarse influence:

$$\left(a_{ii} + \sum_{j \in \mathcal{F}_i^{\mathrm{W}}} a_{ij}\right) e_i = -a_{ij}^{\mathrm{C}}e_j - \sum_{j \in \mathcal{F}_i^{\mathrm{S}}} a_{ij}e_j, \tag{3.65}$$

where a_{ij}^{C} is the off–diagonal matrix element connecting the fine equation i to coarse coupled equation j. There are multiple possibilities for the treatment of fine influences $\mathcal{F}_i^{\mathrm{S}}$ [27, 28].

- *Direct interpolation* in which only the coarse influences of an equation are used for interpolation. Fine influences are included in a *scaling factor*. We use this option in the scope of the thesis.

- *Standard interpolation* in which the contribution of fine influences is included through their coarse influences. The set of interpolatory variables is extended in comparison to standard interpolation. It is also possible to include the coarse influences of weak fine neighbours which further extends the number of variables used for interpolation.

- *Multi–pass interpolation* which is used in combination with aggressive coarsening procedure in which there is no guarantee that every fine equation will have a coarse connection. It is not considered in the scope of this work.

In line with direct interpolation, fine influences of an equation are included in the scaling factor γ:

$$\gamma = \frac{\sum_{j \in \mathcal{C}_i} a_{ij} + \sum_{j \in \mathcal{F}_i^S} a_{ij}}{\sum_{j \in \mathcal{C}_i} a_{ij}} \geq 1. \tag{3.66}$$

The scaling factor increases the contribution of coarse influences (it is always greater or equal to 1) used in the interpolation. Also, it ensures that the matrix row sum is preserved. Finally, the interpolation weight for a fine equation i from a coarse influence j:

$$w = -\gamma \cdot \frac{1}{a_{ii} + \sum_{j \in \mathcal{F}_i^W} a_{ij}} a_{ij}^{\mathrm{C}}. \tag{3.67}$$

Obviously, interpolation weights will only be calculated for the fine level equations which were not chosen into the coarse subset. Coarse equations will directly apply their correction calculated on the coarse level, thus the interpolation weight is equal to 1.

The dimensions of the prolongation matrix are equal to *number of fine level equations* $\times$ *number of equations in the coarse subset*. For a diagonally equal row in the fine matrix, sum of the weights in the row of the prolongation matrix is

equal to 1, i.e. the row sum is preserved, meaning that a constant will be exactly interpolated.

Restriction matrix and coarse level matrix

The restriction matrix is used to transfer the residual from fine to coarse level and it is defined consistently with a variational principle valid for a positive definite matrix $\mathbf{A}$ (Galerkin principle), as shown in section 3.2.2. The correction from the coarse level $\mathbf{P}\mathbf{e}^{\mathrm{C}}$ which minimises the $\mathbf{A}$–norm of the corrected error satisfies:

$$\mathbf{P}^T\mathbf{A}\mathbf{P}\mathbf{e}^{\mathrm{C}} = \mathbf{P}^T(\mathbf{b} - \mathbf{A}\mathbf{x}), \tag{3.68}$$

and then the restriction matrix is set to be the transpose of the prolongation matrix:

$$\mathbf{R} = \mathbf{P}^T. \tag{3.69}$$

The coarse level matrix is calculated as a matrix product of the restriction matrix, fine level matrix and prolongation matrix:

$$\mathbf{A}^{\mathrm{C}} = \mathbf{R}\mathbf{A}^{\mathrm{F}}\mathbf{P}. \tag{3.70}$$

If we have to calculate a product of the coarse level solution $\mathbf{x}^{\mathrm{C}}$ and coarse level matrix $\mathbf{A}^{\mathrm{C}}$, it is not necessary to explicitly calculate the coarse level matrix. Instead, calculate the fine level solution $\mathbf{x}^{\mathrm{F}}$ as a product of prolongation and coarse level solution:

$$\mathbf{x}^{\mathrm{F}} = \mathbf{P}\mathbf{x}^{\mathrm{C}}.$$

Then, we shall multiply the fine level solution with the fine level matrix:

$$\mathbf{A}^{\mathrm{F}}\mathbf{x}^{\mathrm{F}} = \mathbf{A}^{\mathrm{F}}\mathbf{P}\mathbf{x}^{\mathrm{C}},$$

and finally restrict the matrix–vector product onto the coarse level:

$$\mathbf{R}\mathbf{A}^{\mathrm{F}}\mathbf{x}^{\mathrm{F}} = \underbrace{\mathbf{R}\mathbf{A}^{\mathrm{F}}\mathbf{P}}\mathbf{x}^{\mathrm{C}} = \underbrace{\mathbf{A}^{\mathrm{C}}}\mathbf{x}^{\mathrm{C}}.$$

Thus, a coarse level vector–matrix product can be calculated without explicitly assembling the coarse level matrix. Since coarse level multiplications are conducted multiple times in a linear iteration, we want to avoid redundant operations with the eliminated rows of the fine level matrix $\mathbf{A}^{\mathrm{F}}$. Instead, coarse level

matrix can be explicitly calculated as a triple product $\mathbf{RA}^{\mathrm{F}}\mathbf{P}$, which eliminates unnecessary operations.

In conclusion, compared to additive correction method (AAMG), the computational effort is much larger for the selection method (SAMG). Both algorithms include two phases: *setup* and *solution* phase. This section has been dedicated only to the setup phase, while the solution phase is covered in Section 3.2.2. However, it is already evident that SAMG has a more complex setup phase compared to AAMG.

The first part of the algorithm for both methods is the construction of the coarse level (coarsening). Both methods use the strength of connectivity in the coefficient matrix, but in a different way. In AAMG, it is assumed that the connection between two equations is symmetric. If this connection is strong enough compared to the diagonal elements of both equations, the equations are grouped into a cluster which represents a single equation on the coarse level. In SAMG, there is no assumption about the symmetry of the matrix, although all the proofs regarding the performance of the solver found in literature are derived for a symmetric positive definite matrix. However, it is important to note that FVM matrices may have unsymmetric elements but the addressing is always symmetric. In SAMG, all off–diagonal elements are taken into consideration in order to determine the largest influence for each equation and use it as a criterion for determining other strong connections. The equations are then sorted into coarse and fine subsets.

The prolongation in AAMG is a simple injection and the prolongation matrix is not assembled explicitly. In SAMG, it is necessary to explicitly calculate the weights which will be assembled into the prolongation matrix for all equations belonging to the fine subset.

The restriction of fine level residual onto the coarse level in AAMG is done by summing the fine level residuals for all cells which belong to a single cluster. In SAMG, the restriction step is a matrix–vector product of the restriction matrix and fine level residual.

It can already be seen that there are two matrix–vector products in SAMG, while there are none in AAMG. However, the largest computational difference in the setup phase is the formation of the coarse level matrix. The formation

of the matrix in AAMG is done by simply summing the fine matrix elements. The coarse matrix in SAMG is calculated as a triple product of matrices which is computationally significantly more expensive.

3.2.6. AMG Solvers for Block–Matrices

The algorithms presented in sections 3.2.4. and 3.2.5. are described in terms of a matrix with scalar elements. However, the implicitly coupled pressure–velocity system is dicretised as a block–matrix and the matrix element is a 4×4 block, as described in Section 2.4.1. Block AMG methods have been analysed in literature: the most detailed classification and terminology has been given in [30], and will be used in this section. AAMG and SAMG can be used in the exact form that was described in previous sections by introducing a *primary matrix*. A primary matrix is a matrix with scalar elements which represents the block–matrix in some way. Obviously, it is important that the strength of connection in the block–matrix is well represented by the primary matrix. It is important also, especially for SAMG, to take into account the signs of the matrix elements. Since the equations are discretised on the same computational mesh, it is natural to create the same hierarchy of coarse levels for all unknowns.

To explain what lies under the term "same hierarchy", it is again practical to think about coarsening of the computational mesh rather than the matrix. In the mesh, each cell has its own set of unknown variables (three components of velocity and pressure). It is ill–advised to split the unknowns which belong to a single cell and do the coarsening procedure for each unknown separately. In that case, for a three–dimensional flow, four coarse level matrices would be created: one corresponding to each component of velocity and one corresponding to pressure. There would have to exist separate restriction and prolongation matrices for each unknown variable. Also, the variables would have to be decoupled on coarse levels (the cross–couplings between variables would have to be treated explicitly). This approach is called *variable–based AMG* in [30]. The approach used in this thesis is the *point–based AMG*, although in the framework of Finite Volume method, it would be more precise to call it a *cell–based AMG*. The starting point is the definition of the primary matrix. One of the possibilities is to

choose a single unknown variable and use its connectivity pattern to define the primary matrix. For example, one may choose the pressure equation and exclude the cross–couplings to velocity. Then, the sparsity pattern of the primary matrix is the same as the block–matrix, since the pressure equation is defined for each cell. The elements taken from the block–matrix are reduced to a scalar value in the primary matrix:

$$\mathbf{A}_{ij}^{block} = \begin{bmatrix} a_{u_{xi},u_{xj}} & a_{u_{xi},u_{yj}} & a_{u_{xi},u_{zj}} & a_{u_{xi},p_j} \\ a_{u_{yi},u_{xj}} & a_{u_{yi},u_{yj}} & a_{u_{yi},u_{zj}} & a_{u_{yi},p_j} \\ a_{u_{zi},u_{xj}} & a_{u_{zi},u_{yj}} & a_{u_{zi},u_{zj}} & a_{u_{zi},p_j} \\ a_{p_i,u_{xj}} & a_{p_i,u_{yj}} & a_{p_i,u_{zj}} & a_{p_i,p_j} \end{bmatrix} \rightarrow a_{ij}^{primary} = a_{p_i,p_j} \,. \tag{3.71}$$

The choice of the pressure equation as the primary matrix is appropriate considering its properties: the discretisation of the Laplacian produces a symmetric positive definite matrix, as required by AMG. Also, pressure is a scalar solution variable and it significantly (linearly) influences the velocity field. It is also convenient to create the primary matrix based on a norm of the block element:

$$a_{ij}^{primary} = \|\mathbf{A}_{ij}^{block}\|. \tag{3.72}$$

The primary matrix defined using the element–wise norm of block elements also has the same sparsity pattern and the scalar elements reflect the contribution of all unknown variables, depending on the definition of the norm. For example, using the Frobenius norm:

$$\|\mathbf{A}_{ij}^{block}\|_{\mathrm{F}} = \sqrt{\sum a_{\mathbf{u},\mathbf{u}}^2 + \sum a_{\mathbf{u},p}^2 + \sum a_{p,\mathbf{u}}^2 + \sum a_{p,p}^2}, \tag{3.73}$$

where $\|\quad\|_{\mathrm{F}}$ is the Frobenius norm, $a_{\mathbf{u},\mathbf{u}}$ denotes all the elements coupling the velocity components (convection, diffusion), $a_{\mathbf{u},p}$ the contribution of pressure in the momentum equation (pressure gradient), $a_{p,\mathbf{u}}$ the contribution of velocity in the pressure equation (velocity divergence) and $a_{p,p}$ the pressure Laplacian. Variable cross–coupling terms could be excluded from the norm and it would be calculated as:

$$\|\mathbf{A}_{ij}^{block}\|_{\mathrm{NCC}} = \sqrt{a_{\mathbf{u}_{xi},\mathbf{u}_{xj}}^2 + a_{\mathbf{u}_{yi},\mathbf{u}_{yj}}^2 + a_{\mathbf{u}_{zi},\mathbf{u}_{zj}}^2 + a_{p_i,p_j}^2}. \tag{3.74}$$

Since both the Frobenius norm and the norm without cross–couplings produce positive scalar elements, the criterion for the sign of off–diagonal elements is no

longer applicable. In [30], it is suggested that all off–diagonal scalar elements in the primary matrix obtained from the norm–based procedure, take the sign opposite to diagonal, thus all the connections are considered to be negative. It was reported that in the norm–based construction of the primary matrix, the choice of the norm did not affect the convergence significantly since usually the off–diagonal elements are dominated by a one or more large elements which largely contribute to different types of norms.

After the primary matrix has been constructed, the coarsening procedure is applied to it, in line what has already been described in previous sections: agglomeration of equations in clusters for AAMG and selection of coarse and fine equations in SAMG.

The construction of prolongation, restriction and coarse level matrices in AAMG is straightforward: prolongation is again an injection, restriction is done by summing the residuals and the coarse level matrix is constructed by summation of fine matrix elements. In contrast to the scalar version of AAMG, the unknown variables and the corresponding residuals and coarse level corrections are in vector form, e.g.:

$$\mathbf{e}^{\mathrm{C}} = \left[e_{\mathbf{u}_x}, e_{\mathbf{u}_y}, e_{\mathbf{u}_z}, e_p\right]^T. \tag{3.75}$$

In SAMG, the splitting into coarse and fine equations is based on the primary scalar matrix and it is the same for all unknowns. However, it is possible to define the prolongation and restriction based on the fine block–matrix for each variable separately, known as the *multiple–unknown interpolation* [30]. The corresponding matrices are block–matrices, which requires additional storage. Since the coupling between pressure and velocity is linear, and the momentum equation is highly anisotropic, the approach chosen in this work is the *single–unknown interpolation* [30]: the interpolation formulae are calculated based on the primary matrix and are applied to all variables belonging to a cell. Thus, the restriction and prolongation matrices can be compressed into scalar matrices which saves memory.

3.2.7. Parallelisation of AMG Solvers and Smoothers

In this section the parallelisation of algebraic multigrid will be discussed. To achieve textbook performance, it is desirable to stay as close as possible to the sequential outline of the algorithm. The challenge lies in the fact that the Galerkin triple matrix product which is used to create coarse level matrix in SAMG, produces additional off–diagonal elements, meaning that the matrices on coarse levels become more dense. This leads to high parallel communication costs on coarse levels. Also, there exists a problem of load balancing and the issue of possible idle processors.

As mentioned before, the multigrid algorithm can be divided into two phases: setup phase, which includes creating the hierarchy of the coarse level matrices, and the solution phase, which includes smoothing of the error and correction of the solution. Decision on the coarsening process in the setup phase will have an effect on the performance (convergence) in the solution phase. For easier understanding of the parallel algorithm, an illustration is given in Fig. 3.9: a mesh split between two processors, with the coresponding matrices belonging to each processor and off–diagonal elements located on processor boundaries illustrated by Eqn. (3.77).

For SAMG, the critical point is to decide whether to include the elements on the processor boundary in the strength of connectivity criterion. If these connections are included, the resulting coarsening should be identical to the coarsening produced by the sequential algorithm. However, this means that more information will have to be communicated between the two processors and it is possible that on the following coarse levels in the cycle, the communication costs will be greater than the calculation costs in the setup phase. Thus, a different strategy has to be employed in order to keep the algorithm efficient.

We have chosen to limit the coarsening process and make it local for each processor, similar to Ruge–Stüben coarsening described in [58]: the connections which are located on a neighbouring processor are not considered in the selection of influences and dependencies. Consequently, the subset of coarse equations will be different than in the sequential algorithm. Also, minimal number of coarse equations is defined to be the same for each processor and the number of coarse

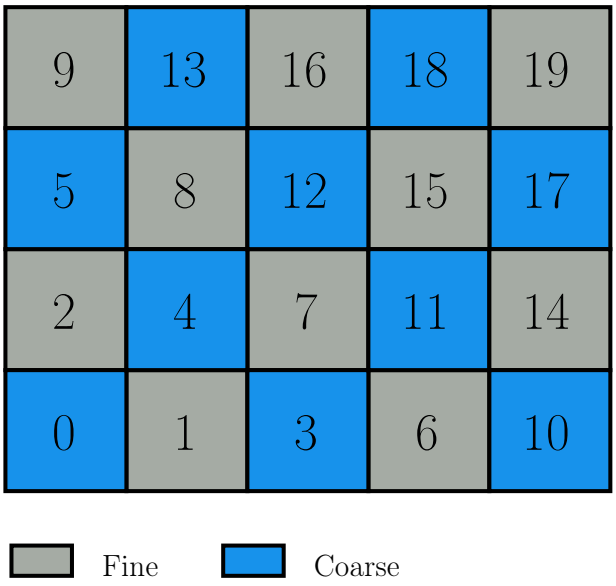


Figure 3.8: 2D mesh coarsening pattern produced by the sequential SAMG algorithm, for a Laplacian operator.

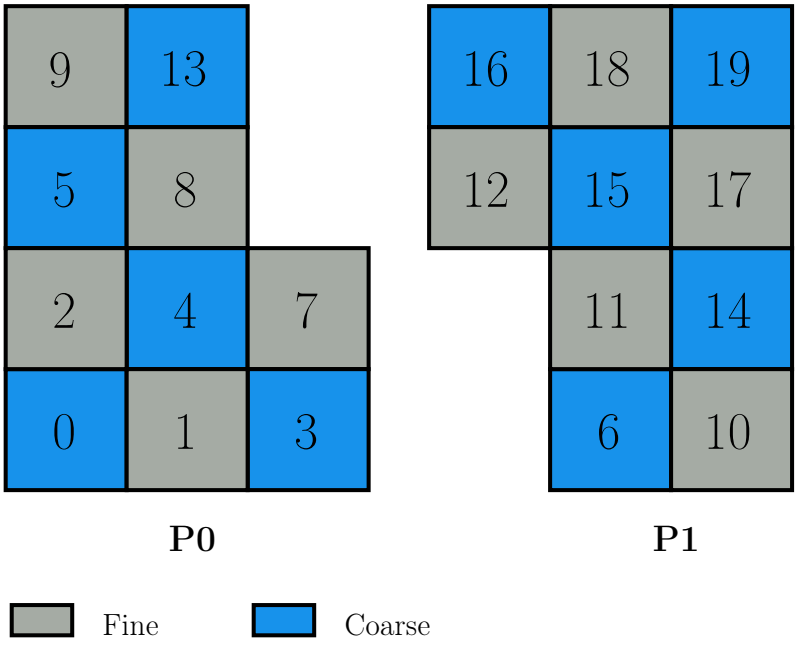


Figure 3.9: 2D mesh coarsening pattern produced by the parallel SAMG algorithm, for a Laplacian operator, on two neighbouring processors *P0* and *P1*.

levels is enforced to be the same. That is, there will be no idle processors and no mapping of data from one processor to another.

An example of sequential coarsening on a single processor is shown in Fig. 3.8: the coarsening pattern corresponds to a Laplacian operator discretised on a regular hexahedral mesh (all off–diagonal elements have the sign opposite to the diagonal element, all have the same magnitude, matrix rows are diagonally equal, except on the boundaries). Thus, the coarsening is expected to be similar to a red–black distribution of a chessboard. Since there are multiple equations with the same weight (same number of dependencies, equations 4, 7, 8, 11, 12, 15), the choice of the strongest equation is determined by the smallest cell index (equation 4). The remainder of the coarsening process is guided by the weights being incremented or decremented, as described in section 3.2.6. The correction from the coarse level is interpolated into fine equations following Eqn. (3.67), and copied into the corresponding coarse equations. All fine equations have multiple coarse neighbours, which is in accordance with the local maximum principle. However, the number of coarse equations could be smaller if a more aggressive coarsening algorithm is employed, where each fine equation has fewer coarse neighbours (some only one), which is not the case in this thesis. We found that the standard coarsening algorithm provides a satisfactory interpolation formula for the smooth error components. Aggressive coarsening could be a good choice if we used extended interpolation formulae, which take into account coarse neighbours of fine influences.

In the case of parallel multigrid, it can be seen in Fig. 3.9 that the coarsening algorithm chooses the same pattern of coarse equations on processor *P0*, which is the implication of the matrix chosen for this demonstration and the same hierarchy of cell indices. However, on processor *P1*, the pattern of coarse and fine cells is mirrored compared to the sequential algorithm. The reason lies in the fact that connections to cells across the processor boundary were not taken into account. Thus, the weighting factors of equations on processor *P1* were not updated as in the sequential algorithm and the first equation to become coarse was the one with the largest weight (equation 15). Weight updates on processor *P1* are performed independently of processor *P0* and the resulting subset of coarse equations is different from the sequential algorithm. The limitation of communication in the setup phase can have a negative effect on the convergence: the formulae in the prolongation matrix are somewhat truncated compared to

the sequential algorithm. The matrices corresponding to meshes in Figs. 3.8 and 3.9, respectively, have the following structure:

$$
\begin{array}{c|cccccccccccccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\
0 & a_{0,0} & a_{0,1} & a_{0,2} & & & & & & & & & & & & & & & & & \\
1 & a_{1,0} & a_{1,1} & & a_{1,3} & a_{1,4} & & & & & & & & & & & & & & & \\
2 & a_{2,0} & & a_{2,2} & & a_{2,4} & a_{2,5} & & & & & & & & & & & & & & \\
3 & & a_{3,1} & & a_{3,3} & & & a_{3,6} & a_{3,7} & & & & & & & & & & & & \\
4 & & a_{4,1} & a_{4,2} & & a_{4,4} & & & a_{4,7} & a_{4,8} & & & & & & & & & & & \\
5 & & & a_{5,2} & & & a_{5,5} & & & a_{5,8} & a_{5,9} & & & & & & & & & & \\
6 & & & & a_{6,3} & & & a_{6,6} & & & & a_{6,10} & a_{6,11} & & & & & & & & \\
7 & & & & a_{7,3} & a_{7,4} & & & a_{7,7} & & & & a_{7,11} & a_{7,12} & & & & & & & \\
8 & & & & & a_{8,4} & a_{8,5} & & & a_{8,8} & & & & a_{8,12} & a_{8,13} & & & & & & \\
9 & & & & & & a_{9,5} & & & & a_{9,9} & & & & a_{9,13} & & & & & & \\
10 & & & & & & & a_{10,6} & & & & a_{10,10} & & & & a_{10,14} & & & & & \\
11 & & & & & & & a_{11,6} & a_{11,7} & & & & a_{11,11} & & & a_{11,14} & a_{11,15} & & & & \\
12 & & & & & & & & a_{12,7} & a_{12,8} & & & & a_{12,12} & & & a_{12,15} & a_{12,16} & & & \\
13 & & & & & & & & & a_{13,8} & a_{13,9} & & & & a_{13,13} & & & a_{13,16} & & & \\
14 & & & & & & & & & & & a_{14,10} & a_{14,11} & & & a_{14,14} & & & a_{14,17} & & \\
15 & & & & & & & & & & & & a_{15,11} & a_{15,12} & & & a_{15,15} & & a_{15,17} & a_{15,18} & \\
16 & & & & & & & & & & & & & a_{16,12} & a_{16,13} & & & a_{16,16} & & a_{16,18} & \\
17 & & & & & & & & & & & & & & & a_{17,14} & a_{17,15} & & a_{17,17} & & a_{17,19} \\
18 & & & & & & & & & & & & & & & & a_{18,15} & a_{18,16} & & a_{18,18} & a_{18,19} \\
19 & & & & & & & & & & & & & & & & & & a_{19,17} & a_{19,18} & a_{19,19}
\end{array}
\tag{3.76}
$$

$$
\begin{array}{c|cccccccccccccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 7 & 8 & 9 & 13 & 6 & 10 & 11 & 12 & 14 & 15 & 16 & 17 & 18 & 19 \\
0 & a_{0,0} & a_{0,1} & a_{0,2} & & & & & & & & & & & & & & & & & \\
1 & a_{1,0} & a_{1,1} & & a_{1,3} & a_{1,4} & & & & & & & & & & & & & & & \\
2 & a_{2,0} & & a_{2,2} & & a_{2,4} & a_{2,5} & & & & & & & & & & & & & & \\
3 & & a_{3,1} & & a_{3,3} & & & a_{3,7} & & & & a_{3,6} & & & & & & & & & \\
4 & & a_{4,1} & a_{4,2} & & a_{4,4} & & a_{4,7} & a_{4,8} & & & & & & & & & & & & \\
5 & & & a_{5,2} & & & a_{5,5} & & a_{5,8} & a_{5,9} & & & & & & & & & & & \\
7 & & & & a_{7,3} & a_{7,4} & & a_{7,7} & & & & & & a_{7,11} & a_{7,12} & & & & & & \\
8 & & & & & a_{8,4} & a_{8,5} & & a_{8,8} & & a_{8,13} & & & & a_{8,12} & & & & & & \\
9 & & & & & & a_{9,5} & & & a_{9,9} & a_{9,13} & & & & & & & & & & \\
13 & & & & & & & & a_{13,8} & a_{13,9} & a_{13,13} & & & & & & & a_{13,16} & & & \\
6 & & & & a_{6,3} & & & & & & & a_{6,6} & a_{6,10} & a_{6,11} & & & & & & & \\
10 & & & & & & & & & & & a_{10,6} & a_{10,10} & & & a_{10,14} & & & & & \\
11 & & & & & & & a_{11,7} & & & & a_{11,6} & & a_{11,11} & & a_{11,14} & a_{11,15} & & & & \\
12 & & & & & & & a_{12,7} & a_{12,8} & & & & & & a_{12,12} & & a_{12,15} & a_{12,16} & & & \\
14 & & & & & & & & & & & & a_{14,10} & a_{14,11} & & a_{14,14} & & & a_{14,17} & & \\
15 & & & & & & & & & & & & & a_{15,11} & a_{15,12} & & a_{15,15} & & a_{15,17} & a_{15,18} & \\
16 & & & & & & & & & & a_{16,13} & & & & a_{16,12} & & & a_{16,16} & & a_{16,18} & \\
17 & & & & & & & & & & & & & & & a_{17,14} & a_{17,15} & & a_{17,17} & & a_{17,19} \\
18 & & & & & & & & & & & & & & & & a_{18,15} & a_{18,16} & & a_{18,18} & a_{18,19} \\
19 & & & & & & & & & & & & & & & & & & a_{19,17} & a_{19,18} & a_{19,19}
\end{array}
\tag{3.77}
$$

The matrix in Eqn. (3.77) illustrates the situation after the domain has been split onto two processors. It has the same number of elements as the matrix in the sequential algorithm, Eqn. (3.76), and they have the same value. Red elements correspond to the matrix on processor *P0*, blue elements represent the matrix elements on processor *P1*. Cell indices and the corresponding equation rows are renumbered locally for each processor, but we have kept the sequential numbering for a more clear comparison. The elements in black correspond to cell communication across processor boundaries and are stored and available on both processors. In case of a symmetric matrix (Laplacian), boundary elements are identical on both processors. It is clear now that on processor *P0*, only the red elements are used in the coarsening process, and on processor *P1*, only the blue elements are included in coarsening. However, it is very important to preserve matrix row sum when calculating the interpolation formulae, and the boundary elements need to be included in the scaling factor:

$$\gamma = \frac{\sum_{j\in\mathcal{C}_i} a_{ij} + \sum_{j\in\mathcal{F}_i^S} a_{ij} + \sum_{j\in\mathcal{B}_i} a_{ij}}{\sum_{j\in\mathcal{C}_i} a_{ij}} \geq 1, \tag{3.78}$$

where $\mathcal{B}$ is a set of boundary elements describing the influence of equation j, located on a neighbouring processor, onto equation i, located on the local processor. The calculated weighting factors are assembled into the local prolongation matrix:

$$\mathbf{P}_{P0} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 7 \\ 8 \\ 9 \\ 13 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 3 & 4 & 5 & 13 \end{array} \\ \left[\begin{array}{ccccc} w_{0,0}=1 & & & & \\ w_{1,0} & w_{1,3} & w_{1,4} & & \\ w_{2,0} & & w_{2,4} & w_{2,5} & \\ & w_{3,3}=1 & & & \\ & & w_{4,4}=1 & & \\ & & & w_{5,5}=1 & \\ & w_{7,3} & w_{7,4} & & \\ & & w_{8,4} & w_{8,5} & w_{8,13} \\ & & & w_{9,5} & w_{9,13} \\ & & & & w_{13,13}=1 \end{array}\right] \end{array}$$

$$\mathbf{P}_{P1} = \begin{array}{c} \\ 6 \\ 10 \\ 11 \\ 12 \\ 14 \\ 15 \\ 16 \\ 17 \\ 18 \\ 19 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 6 & & 14 & & 15 & & 16 & & 19 \end{array} \\ \begin{bmatrix} w_{6,6}=1 & & & & & & & & \\ w_{10,6} & & w_{10,14} & & & & & & \\ w_{11,6} & & w_{11,14} & & w_{11,15} & & & & \\ & & & & w_{12,15} & & w_{12,16} & & \\ & & w_{14,14}=1 & & & & & & \\ & & & & w_{15,15}=1 & & & & \\ & & & & & & w_{16,16}=1 & & \\ & & w_{17,14} & & w_{17,15} & & w_{17,16} & & \\ & & & & w_{18,15} & & w_{18,16} & & w_{18,19} \\ & & & & & & & & w_{19,19}=1 \end{bmatrix} \end{array}. \quad (3.79)$$

Restriction matrices are obtained by transposing the prolongation matrices:

$$\mathbf{R}_{P0} = \begin{array}{c} \\ 0 \\ 3 \\ 4 \\ 5 \\ 13 \end{array} \begin{array}{c} \begin{array}{ccccccccccc} 0 & & 1 & 2 & 3 & & 4 & 5 & 7 & 8 & 9 & & 13 \end{array} \\ \begin{bmatrix} w_{0,0}=1 & w_{0,1} & w_{0,2} & & & & & & & & & & \\ & w_{3,1} & & w_{3,3}=1 & & & & & w_{3,7} & & & & \\ & w_{4,1} & w_{4,2} & & & w_{4,4}=1 & & & w_{4,7} & w_{4,8} & & & \\ & & w_{2,5} & & & & & w_{5,5}=1 & & w_{5,8} & w_{5,9} & & \\ & & & & & & & & & w_{13,8} & w_{13,9} & & w_{13,13}=1 \end{bmatrix} \end{array} \quad (3.80)$$

$$\mathbf{R}_{P1} = \begin{array}{c} \\ 6 \\ 14 \\ 15 \\ 16 \\ 19 \end{array} \begin{array}{c} \begin{array}{ccccccccccc} 6 & & 10 & 11 & 12 & & 14 & 15 & 16 & 17 & 18 & & 19 \end{array} \\ \begin{bmatrix} w_{6,6}=1 & w_{6,10} & w_{6,11} & & & & & & & & & & \\ & w_{14,10} & w_{14,11} & & & w_{14,14}=1 & & & & w_{14,17} & & & \\ & & w_{15,11} & w_{15,12} & & & & w_{15,15}=1 & & w_{15,17} & w_{15,18} & & \\ & & & w_{16,12} & & & & & w_{16,16}=1 & w_{16,17} & w_{16,18} & & \\ & & & & & & & & & & w_{19,18} & & w_{19,19}=1 \end{bmatrix} \end{array}. \quad (3.81)$$

When calculating coarse level matrices on both processors, the coarse level processor boundary elements also need to be calculated. To set up the triple product on the processor boundary, the local prolongation matrices are filtered to include

only the rows for cells located on the processor boundary:

$$\mathbf{P}_{P0}^{filtered} = \begin{array}{c} \\ 3 \\ 7 \\ 8 \\ 13 \end{array} \begin{array}{c} \begin{array}{cccc} 3 & 4 & 5 & 13 \end{array} \\ \begin{bmatrix} w_{3,3}=1 & & & \\ w_{7,3} & w_{7,4} & & \\ & w_{8,4} & w_{8,5} & w_{8,13} \\ & & & w_{13,13}=1 \end{bmatrix} \end{array} \quad \mathbf{P}_{P1}^{filtered} = \begin{array}{c} \\ 6 \\ 11 \\ 12 \\ 16 \end{array} \begin{array}{c} \begin{array}{cccc} 6 & 14 & 15 & 16 \end{array} \\ \begin{bmatrix} w_{6,6}=1 & & & \\ w_{11,6} & w_{11,14} & w_{11,15} & \\ & & w_{12,15} & w_{12,16} \\ & & & w_{16,16}=1 \end{bmatrix} \end{array}. \quad (3.82)$$

The restriction matrices for the processor boundaries are created as a transpose of filtered prolongation matrices:

$$\mathbf{R}_{P0}^{filtered} = \begin{array}{c} \\ 3 \\ 4 \\ 5 \\ 13 \end{array} \begin{array}{c} \begin{array}{cccc} 3 & 7 & 8 & 13 \end{array} \\ \begin{bmatrix} w_{3,3}=1 & w_{3,7} & & \\ & w_{4,7} & w_{4,8} & \\ & & w_{5,8} & \\ & & w_{13,8} & w_{13,13}=1 \end{bmatrix} \end{array} \quad \mathbf{R}_{P1}^{filtered} = \begin{array}{c} \\ 6 \\ 14 \\ 15 \\ 16 \end{array} \begin{array}{c} \begin{array}{cccc} 6 & 11 & 12 & 16 \end{array} \\ \begin{bmatrix} w_{6,6}=1 & w_{6,11} & & \\ & w_{14,11} & & \\ & w_{15,11} & w_{15,12} & \\ & & w_{16,12} & w_{16,16}=1 \end{bmatrix} \end{array}. \quad (3.83)$$

Before calculating the triple product on the processor boundary, the local filtered prolongation matrix is sent to the neighbouring processor, and the neighbour's filtered prolongation matrix is received. The calculation of coarse processor boundary elements is done locally on each processor, following the Galerkin projection method. On processor *P0*, the coarse level boundary elements $\mathbf{B}_{P0}^{\mathrm{C}}$ which connect it to processor *P1* are calculated from the triple matrix product:

$$\mathbf{B}_{P0}^{\mathrm{C}} = \mathbf{R}_{P0}\mathbf{B}_{P0}^{\mathrm{F}}\mathbf{P}_{P1}, \quad (3.84)$$

where $\mathbf{B}_{P0}^{\mathrm{F}}$ is the submatrix of the fine matrix $\mathbf{A}^{\mathrm{F}}$ which contains the processor boundary elements, shown in Eqn. (3.77) as the black elements in the upper right part. Similarly, on processor *P1*, the coarse level boundary elements are calculated as:

$$\mathbf{B}_{P1}^{\mathrm{C}} = \mathbf{R}_{P1}\mathbf{B}_{P1}^{\mathrm{F}}\mathbf{P}_{P0}, \quad (3.85)$$

where $\mathbf{B}_{P0}^{\mathrm{F}}$ is the submatrix of the fine matrix $\mathbf{A}^{\mathrm{F}}$, shown in Eqn. (3.77) as the black elements in the lower left part.

Employing a standard coarsening algorithm which does not differentiate between interior equations and equations on the processor boundary (except that it does not allow interpolation across the boundary), means that the equations on the

boundary can be sorted into both the coarse and fine subset. To visually explain how the communication between equations on coarse level is established, we have to consider three situations which can occur in the coarsening process:

1. two coupled equations on the boundary of neighbouring processors are sorted into coarse subset,
2. one equation on the boundary is sorted into fine subset, while the coupled equation on the neighbouring processor is sorted into coarse subset,
3. two coupled equations on the boundary of neighbouring processors are sorted into fine subset.

The first case is the simplest one. If there are two equations which were sorted into the coarse subset, the communication remains the same on the coarse level, i.e. there are no additional off–diagonal elements. However, the off–diagonal element can increase in magnitude if there are common fine neighbours, which will be further discussed for the case with two coupled equations sorted into the fine subset. However, in the sequential algorithm, this can happen only under special circumstances as the coarsening algorithm should not produce a coarse subset containing two coupled equations. In parallel multigrid, this situation is common on shared processor boundaries since there are two coarsening algorithms which independently produce coarse subsets. An example is shown in Fig. 3.9: coarse cells 3 and 6 share a face, as well as cells 13 and 16. Here, there are no additional off–diagonal elements in the coarse level matrix in equations 3, 6, 13 or 16. When two coupled equations on the opposite sides of a processor boundary are sorted into the fine and coarse subset, it is possible that new off–diagonal elements will appear on the coarse level. For example, in Fig. 3.9 assume that equation which corresponds to cell 2 interpolates its coarse level correction from all its couplecd coarse equations: 0, 4 and 5, i.e. there exist weighting elements in the prolongation and restriction matrix. In the triple product, new off–diagonal elements will be created which means that the three coarse equations will be coupled through their common coupled fine equation 2. For example, the contributions to coarse

equation 0 are:

$$
\begin{aligned}
w^R_{0,2} \cdot a^{\mathrm{F}}_{2,0} \cdot w^P_{0,0} &\rightarrow a^{\mathrm{C}}_{0,0}, \\
w^R_{0,2} \cdot a^{\mathrm{F}}_{2,2} \cdot w^P_{2,0} &\rightarrow a^{\mathrm{C}}_{0,0}, \\
w^R_{0,2} \cdot a^{\mathrm{F}}_{2,2} \cdot w^P_{2,4} &\rightarrow a^{\mathrm{C}}_{0,4}, \\
w^R_{0,2} \cdot a^{\mathrm{F}}_{2,2} \cdot w^P_{2,5} &\rightarrow a^{\mathrm{C}}_{0,5}, \\
w^R_{0,2} \cdot a^{\mathrm{F}}_{2,4} \cdot w^P_{4,4} &\rightarrow a^{\mathrm{C}}_{0,4}, \\
w^R_{0,2} \cdot a^{\mathrm{F}}_{2,5} \cdot w^P_{5,5} &\rightarrow a^{\mathrm{C}}_{0,5}.
\end{aligned}
$$

The arrows denote the contribution to a element, i.e. the result of the expression on the left hand side is added to the element. The weights in the restriction and prolongation matrix are always positive, thus the sign of the matrix element is preserved. Thus, there is an increase of diagonal element $a^{\mathrm{C}}_{0,0}$ due to the communication of cells 0 and 2 on the fine level and the diagonal element of eqn. 2. New off–diagonal elements have appeared in the coarse matrix: $a^{\mathrm{C}}_{0,4}$ and $a^{\mathrm{C}}_{0,5}$. These new elements can visually be interpreted as new faces shared by cells 0 and 4, and 0 and 5, due to the "collapse" of cell 2 on the coarse level. The off–diagonal elements will also appear in rows 4 and 5 of the coarse matrix, and the matrix will remain symmetric.

The final possibility is that two coupled equations on the boundary of neighbouring processors are sorted into the fine subset. For example, equations corresponding to cells 8 and 12 in Fig. 3.9. Assume that equation 8 is strongly coupled to equations 4, 5 and 13, and equation 12 is strongly coupled to cells 15 and 16. On the coarse level, a connection will be established through fine equations 8 and 12, between equations 4, 5, 13, 15 and 16. Consider equation 4 and the contributions created through cell 8 on processor *P0*:

$$
\begin{aligned}
w^R_{4,8} \cdot a^{\mathrm{F}}_{8,4} \cdot w^P_{4,4} &\rightarrow a^{\mathrm{C}}_{4,4}, \\
w^R_{4,8} \cdot a^{\mathrm{F}}_{8,5} \cdot w^P_{5,5} &\rightarrow a^{\mathrm{C}}_{4,5}, \\
w^R_{4,8} \cdot a^{\mathrm{F}}_{8,8} \cdot w^P_{8,4} &\rightarrow a^{\mathrm{C}}_{4,4}, \\
w^R_{4,8} \cdot a^{\mathrm{F}}_{8,8} \cdot w^P_{8,5} &\rightarrow a^{\mathrm{C}}_{4,5}, \\
w^R_{4,8} \cdot a^{\mathrm{F}}_{8,8} \cdot w^P_{8,13} &\rightarrow a^{\mathrm{C}}_{4,13}, \\
w^R_{4,8} \cdot a^{\mathrm{F}}_{8,13} \cdot w^P_{13,13} &\rightarrow a^{\mathrm{C}}_{4,13}.
\end{aligned}
$$

These contributions are all a consequence of collapsing a single fine equation on the coarse level. However, looking at the situation on the processor boundary, new boundary elements also appear through fine equations 8 and 12:

$$w^R_{4,8} \cdot a^{\mathrm{F}}_{8,12} \cdot w^P_{12,15} \to a^{\mathrm{C}}_{4,15},$$
$$w^R_{4,8} \cdot a^{\mathrm{F}}_{8,12} \cdot w^P_{12,16} \to a^{\mathrm{C}}_{4,16}.$$

Thus, new processor boundary elements have appeared from collapsing two fine equations and connecting the coarse equations which are coupled to them. It is important to notice that the connection between two coarse equations, 13 and 16, becomes stronger, i.e. the off–diagonal elements $a^{\mathrm{C}}_{13,16}$ and $a^{\mathrm{C}}_{16,13}$ increase in magnitude in comparison to $a_{13,16}$ and $a_{16,13}$ in the fine level matrix, as there is a contribution from collapsed fine equations 8 and 12:

$$w^R_{13,8} \cdot a^{\mathrm{F}}_{8,12} \cdot w^P_{12,16} \to a^{\mathrm{C}}_{13,16},$$
$$w^R_{16,12} \cdot a^{\mathrm{F}}_{12,8} \cdot w^P_{8,13} \to a^{\mathrm{C}}_{16,13}.$$

The resulting coarse level submatrices have the following structure:

$$\mathbf{A}^{\mathrm{C}}_{P0} = \begin{array}{c} \\ 0 \\ 3 \\ 4 \\ 5 \\ 13 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 3 & 4 & 5 & 13 \end{array} \\ \begin{bmatrix} a^{\mathrm{C}}_{0,0} & a^{\mathrm{C}}_{0,3} & a^{\mathrm{C}}_{0,4} & a^{\mathrm{C}}_{0,5} & \\ a^{\mathrm{C}}_{3,0} & a^{\mathrm{C}}_{3,3} & a^{\mathrm{C}}_{3,4} & & \\ a^{\mathrm{C}}_{4,0} & a^{\mathrm{C}}_{4,3} & a^{\mathrm{C}}_{4,4} & a^{\mathrm{C}}_{4,5} & a^{\mathrm{C}}_{4,13} \\ a^{\mathrm{C}}_{5,0} & & a^{\mathrm{C}}_{5,4} & a^{\mathrm{C}}_{5,5} & a^{\mathrm{C}}_{5,13} \\ & & a^{\mathrm{C}}_{13,4} & a^{\mathrm{C}}_{13,5} & a^{\mathrm{C}}_{13,13} \end{bmatrix} \end{array} \quad \mathbf{A}^{\mathrm{C}}_{P1} = \begin{array}{c} \\ 6 \\ 14 \\ 15 \\ 16 \\ 19 \end{array} \begin{array}{c} \begin{array}{ccccc} 6 & 14 & 15 & 16 & 19 \end{array} \\ \begin{bmatrix} a^{\mathrm{C}}_{6,6} & a^{\mathrm{C}}_{6,14} & a^{\mathrm{C}}_{6,15} & & \\ a^{\mathrm{C}}_{14,6} & a^{\mathrm{C}}_{14,14} & a^{\mathrm{C}}_{14,15} & & a^{\mathrm{C}}_{14,19} \\ a^{\mathrm{C}}_{15,6} & a^{\mathrm{C}}_{15,14} & a^{\mathrm{C}}_{15,15} & a^{\mathrm{C}}_{15,16} & a^{\mathrm{C}}_{15,19} \\ & & a^{\mathrm{C}}_{16,15} & a^{\mathrm{C}}_{16,16} & a^{\mathrm{C}}_{16,19} \\ & a^{\mathrm{C}}_{19,14} & a^{\mathrm{C}}_{19,15} & a^{\mathrm{C}}_{19,16} & a^{\mathrm{C}}_{19,19} \end{bmatrix} \end{array} \tag{3.86}$$

$$\mathbf{B}^{\mathrm{C}}_{P0} = \begin{array}{c} \\ 0 \\ 3 \\ 4 \\ 5 \\ 13 \end{array} \begin{array}{c} \begin{array}{ccccc} 6 & 14 & 15 & 16 & 19 \end{array} \\ \begin{bmatrix} & & & & \\ a^{\mathrm{C}}_{3,6} & a^{\mathrm{C}}_{3,14} & a^{\mathrm{C}}_{3,15} & a^{\mathrm{C}}_{3,16} & \\ a^{\mathrm{C}}_{4,6} & a^{\mathrm{C}}_{4,14} & a^{\mathrm{C}}_{4,15} & a^{\mathrm{C}}_{4,16} & \\ & & a^{\mathrm{C}}_{5,15} & a^{\mathrm{C}}_{5,16} & \\ & & a^{\mathrm{C}}_{13,15} & a^{\mathrm{C}}_{13,16} & \end{bmatrix} \end{array} \quad \mathbf{B}^{\mathrm{C}}_{P1} = \begin{array}{c} \\ 6 \\ 14 \\ 15 \\ 16 \\ 19 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 3 & 4 & 5 & 13 \end{array} \\ \begin{bmatrix} & a^{\mathrm{C}}_{6,3} & a^{\mathrm{C}}_{6,4} & & \\ & a^{\mathrm{C}}_{14,3} & a^{\mathrm{C}}_{14,4} & & \\ & a^{\mathrm{C}}_{15,3} & a^{\mathrm{C}}_{15,4} & a^{\mathrm{C}}_{15,5} & a^{\mathrm{C}}_{15,13} \\ & a^{\mathrm{C}}_{16,3} & a^{\mathrm{C}}_{16,4} & a^{\mathrm{C}}_{16,5} & a^{\mathrm{C}}_{16,13} \\ & & & & \end{bmatrix} \end{array}. \tag{3.87}$$

Using these local and boundary matrices, the coarse level matrix can be represented as a block matrix:

$$\begin{bmatrix} \mathbf{A}^{\mathrm{C}}_{P0} & \mathbf{B}^{\mathrm{C}}_{P0} \\ \mathbf{B}^{\mathrm{C}}_{P1} & \mathbf{A}^{\mathrm{C}}_{P1} \end{bmatrix}, \tag{3.88}$$

although the processor boundary elements are not stored in a conventiontional LDU–matrix format.

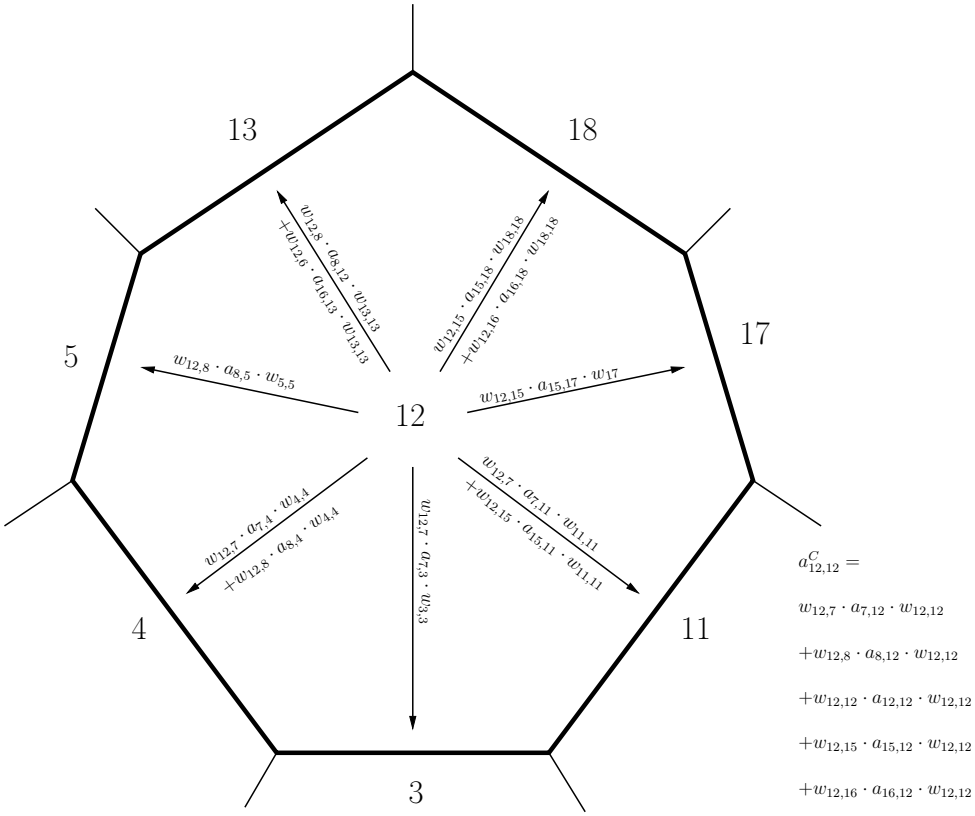


Figure 3.10: Coarse matrix connectivity of cell 12 in the sequential SAMG algorithm. The formulae on the arrows represent the off–diagonal matrix elements. The diagonal element is written out on the right.

It can be seen that the coarse level matrix has more non–zero entries per row, i.e. it becomes denser. The number of processor boundary elements also increases, and the rate depends on the domain decomposition method, and the initial coarsening pattern. The number of boundary elements increases if there are equations on the processor boundary identified as fine. This fact has contributed to development of parallel multigrid algorithms using the subdomain blocking principles,

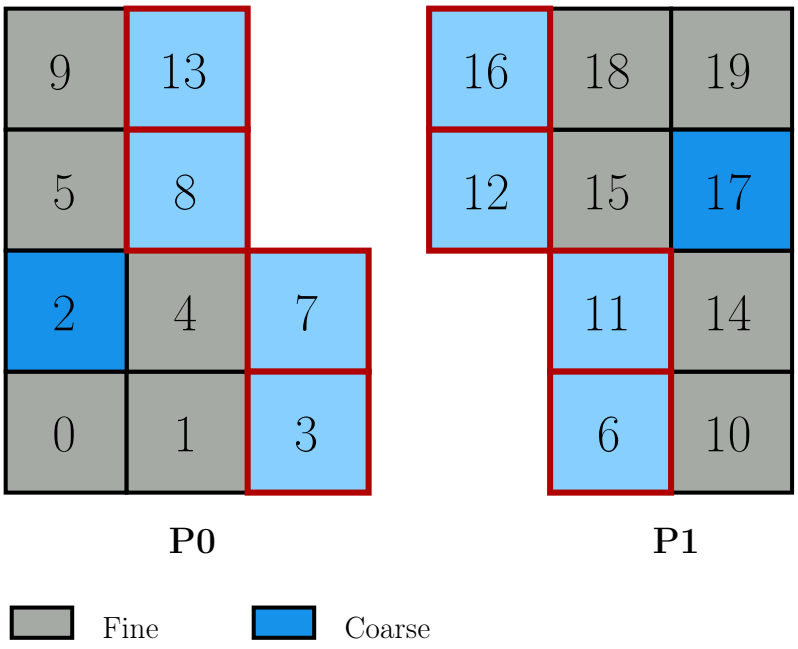


Figure 3.11: Full blocking parallel SAMG: there is a layer of only coarse cells on the processor boundary. There is no need for processor communication since the boundary matrix elements remain the same on all multigrid levels.

[59]. The idea is to prevent the coarse level matrices from penetrating into the neighbouring processor domains by modifying the coarsening algorithm. The full subdomain blocking, shown in Fig. 3.11, creates a layer of only coarse equations on the processor boundary. In this way, the communication between two processors, i.e. the number of processor boundary matrix elements remains the same, but the number of equations on coarse levels increases. This increased number of coarse level equations leads to better convergence, as reported in literature [59], at a cost of significantly denser coarse level matrices. The second approach, minimum subdomain blocking shown in Fig. 3.12, also relies on separating the layer of equations on processor boundaries, but it is not necessary to sort them all into the coarse subset. Instead, the coarsening algorithm is performed separately inside the layer: fine equations can only interpolate the correction from coarse equations in the layer. In this way, the matrices cannot penetrate into the interior of other processors, but there is some communication necessary for taking into account the collapsed fine elements on processor boundaries (strengthening the connection between coarse equations). The number of processor boundary

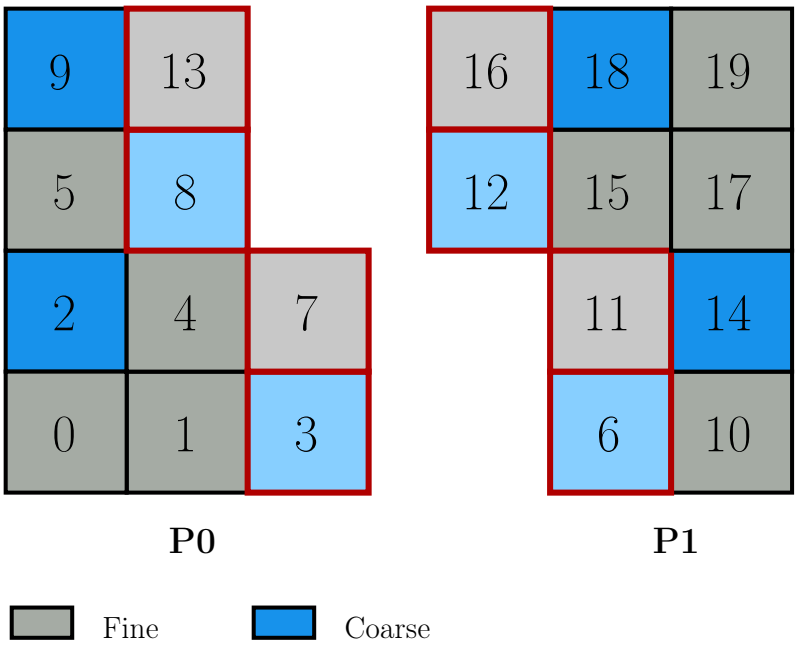


Figure 3.12: Minimum blocking parallel SAMG: there is a layer of cells on the processor boundary which is separated from interior cells in the coarsening process. There is some processor communication since the boundary matrix elements change based on the multigrid level.

matrix elements on the coarse level reduces.

There are also other methods for reducing computation and communication costs, which operate directly on the assembled interpolation and/or coarse level matrices: truncation of the prolongation matrix or sparsification of the coarse level matrix. Truncation of interpolation implies removing the elements in the prolongation matrix and rescaling the formula to preserve row sums, [27]. Sparsification of coarse level matrix eliminates entries in the calculated coarse matrix which does not preserve the properties of the Galerkin projection and can lead to serious convergence issues [60]. Both are based on reducing the density of the coarse level matrix.

3.3. Conjugate Gradient Method

Methods for solving a linear system based on the Krylov subspace will be presented in this section. First, the distinct property of the positive definite matrix will be illustrated as it is the basis for deriving the idea of the method of steepest descent and afterwards, Conjugate Gradients (CG). Section 3.3.3. deals with methods for matrix preconditioning and their impact on the convergence rate. Other CG–like methods for nonsymmetric matrices will be discussed.

3.3.1. Introduction to Conjugate Gradient Method

Derivation of the conjugate gradient method in this thesis is inspired and closely follows the publication of J. Shewchuk [57]. A linear system, $\mathbf{Ax} = \mathbf{b}$, can be reformulated into a minimisation problem. The objective is to find an extreme value (minimum or maximum) of the following function:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{b}^T\mathbf{x} + c, \tag{3.89}$$

where $\mathbf{x}$ is the vector of the unknowns, $\mathbf{A}$ is the coefficient matrix, $\mathbf{b}$ is the right–hand side vector and c is a constant (take $c = 0$). $f(\mathbf{x})$ is a *quadratic function* [57] of $\mathbf{x}$. For a system consisting of two equations, it is easy to draw the quadratic function: an example is shown in Fig. 3.13. $f(\mathbf{x})$ in the shape of a convex paraboloid belongs to a positive definite matrix, while a concave paraboloid corresponds to a quadratic function of a negative definite matrix. Both of these types of matrices have a quadratic function with a clear extremum, a minimum for a positive definite matrix and maximum for a negative definite matrix. The first term in Eqn. (3.89) corresponds to the inner product used in the energy norm defined in section 3.2.2., Eqn. (3.33) and Eqn. (3.36). Thus, for a positive definite matrix, finding the minimum of the quadratic function is equivalent to minimising the energy norm $\|\mathbf{e}\|_1$, i.e. finding the correct solution.

The quadratic function can be represented in two dimensions, using the isocontours which connect the points with the same value of $f(\mathbf{x})$, Fig. 3.14. Black arrows represent the eigenvectors of the matrix with the corresponding eigenvalues. The eigenvector with the larger eigenvalue coincides with the minor axis of

the elliptic isocontour, while the eigenvector with the smaller eigenvalue coincides with the major axis.

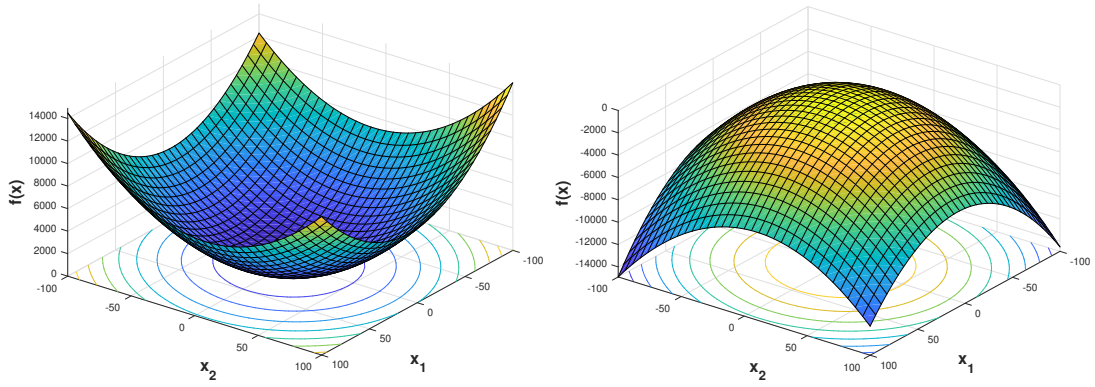


Figure 3.13: Quadratic function of a positive definite matrix has a shape of a convex paraboloid, shown on the left. A negative definite matrix is a negative positive definite matrix and quadratic function is a concave paraboloid, shown on the right.

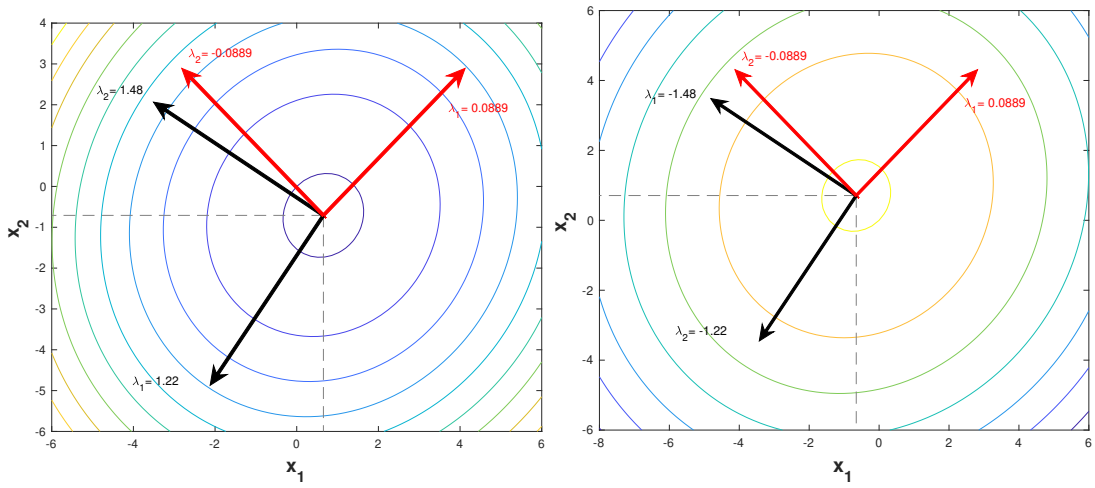


Figure 3.14: Isocontours of a quadratic function corresponding to a positive definite matrix (left) and a negative definite matrix (right). The black arrows correspond to the eigenvectors of matrix $\mathbf{A}$, while the red arrows represent the eigenvectors of a Jacobi preconditioned matrix $\mathbf{I} - \mathbf{D}^{-1}\mathbf{A}$.

At the extremum of the quadratic function, the gradient of the function is equal to 0:

$$f'(\mathbf{x}) = \frac{1}{2}\mathbf{A}^T\mathbf{x} + \frac{1}{2}\mathbf{A}\mathbf{x} - \mathbf{b} = 0. \tag{3.90}$$

For a symmetric ($\mathbf{A} = \mathbf{A}^T$) positive definite matrix, Eqn. (3.90) reduces to:

$$f'(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = 0, \tag{3.91}$$

which is equal to the initial linear system. Thus, the solution of the system is a minimum of the quadratic function. For every point

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

it is possible to calculate the value of the gradient of the quadratic function, which is a vector and it can be seen that for a symmetric positive definite matrix, it is equal to the negative residual:

$$f'(\mathbf{x}) = -(\mathbf{b} - \mathbf{A}\mathbf{x}) = -\mathbf{r}. \tag{3.92}$$

The gradient of the quadratic function points in the direction of the largest increase of the function, and the residual points in the opposite direction, that is, in the direction of the largest decrease of the function. The iterative method for solving a linear system as a minimisation problem, which uses the residual as a search direction is called *steepest descent*. Each iteration of steepest descent has the following form:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha \mathbf{r}^{(k)}, \tag{3.93}$$

where k denotes the value calculated in the previous iteration, $\mathbf{r}$ is the residual and α is the length of the step in the direction of the residual. α is calculated using a *line search* procedure. In Fig. 3.15, residual vectors are plotted on the contours of the quadratic function for several points. Each residual is orthogonal to the contour on which it lies. Also, if we extend the line in the direction in which the residual points, the line is tangent to the isocontour where the quadratic function begins to increase. This point is where the step should end, as the function begins to increase and we would be stepping away from the minimum. The point on the isocontour also corresponds to a residual, which is orthogonal to the isocontour, and consequently to the previous search direction (residual from the previous iteration), which is a tangent on the contour. To find α, a directional derivative

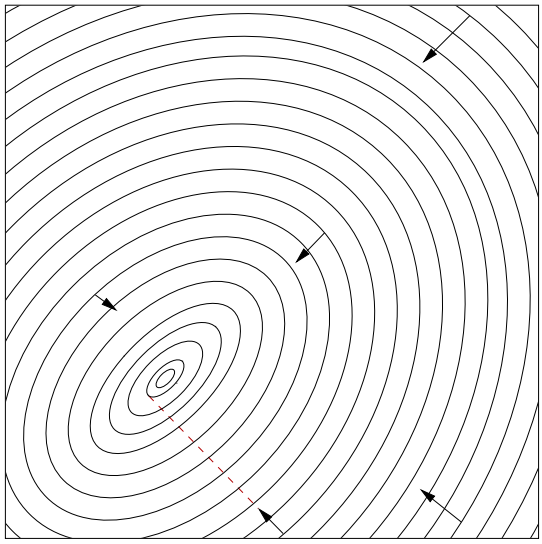


Figure 3.15: Residual vectors plotted on the isocontours of the quadratic function, pointing in the direction of the greatest decrease of the function.

of the quadratic function is calculated as:

$$\frac{d}{d\alpha}f(\mathbf{x}^{(1)}) = f'(\mathbf{x}^{(1)})^T\frac{d}{d\alpha}(\mathbf{x}^{(1)}) = f'(\mathbf{x}^{(1)})^T\frac{d}{d\alpha}(\mathbf{x}^{(0)} + \alpha\mathbf{r}^{(0)}) = f'(\mathbf{x}^{(1)})^T\mathbf{r}^{(0)}$$
$$= (\mathbf{r}^{(1)})^T\mathbf{r}^{(0)}. \tag{3.94}$$

A minimum of the function is found where the derivative is equal to 0, thus:

$$(\mathbf{r}^{(1)})^T\mathbf{r}^{(0)} = 0 \tag{3.95}$$
$$(\mathbf{b} - \mathbf{A}\mathbf{x}^{(1)})^T\mathbf{r}^{(0)} = 0 \tag{3.96}$$
$$(\mathbf{b} - \mathbf{A}(\mathbf{x}^{(0)} + \alpha\mathbf{r}^{(0)}))^T\mathbf{r}^{(0)} = 0 \tag{3.97}$$
$$\underbrace{(\mathbf{b} - \mathbf{A}\mathbf{x}^{(0)})^T}_{(\mathbf{r}^{(0)})^T}\mathbf{r}^{(0)} - \alpha(\mathbf{A}\mathbf{r}^{(0)})^T\mathbf{r}^{(0)} = 0 \tag{3.98}$$

$$\alpha = \frac{(\mathbf{r}^{(0)})^T\mathbf{r}^{(0)}}{(\mathbf{r}^{(0)})^T\mathbf{A}\mathbf{r}^{(0)}}. \tag{3.99}$$

The final algorithm of steepest descent consists of the following steps.

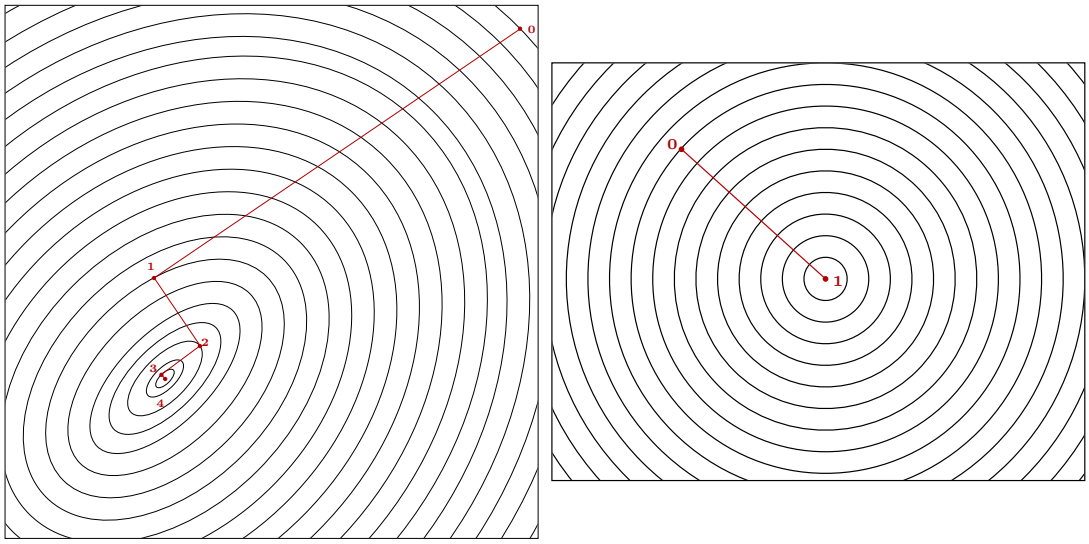


Figure 3.16: Convergence of steepest descent for a 2x2 matrix: left – matrix with two distinct eigenvalues, right – matrix with duplicate eigenvalues.

1. For the current solution $\mathbf{x}^{(k)}$, calculate the residual:
   $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$.

2. Calculate the length of the step in the direction of the residual $\mathbf{r}^{(k)}$:
   $\alpha^k = \frac{(\mathbf{r}^{(k)})^T \mathbf{r}^{(k)}}{(\mathbf{r}^{(k)})^T \mathbf{A}\mathbf{r}^{(k)}}$.

3. Update the solution $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{r}^{(k)}$.
   It is also possible to update the residual instead of the solution (to avoid one matrix–vector multiplication): $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha\mathbf{A}\mathbf{r}^k$.

4. Repeat the procedure from step 1 if the convergence criterion is not met.

An example of convergence of steepest descent for a 2×2 symmetric positive definite matrix is shown in Fig. 3.16. It is obvious that the method needs a large number of iterations, compared to the size of the system, to reach the correct solution. The reason can be found in the eigenspectrum of the coefficient matrix.

Similar to fixed–point methods, the problem appears if the magnitudes of the eigenvalues are different in size. An example is shown in Fig. 3.17.

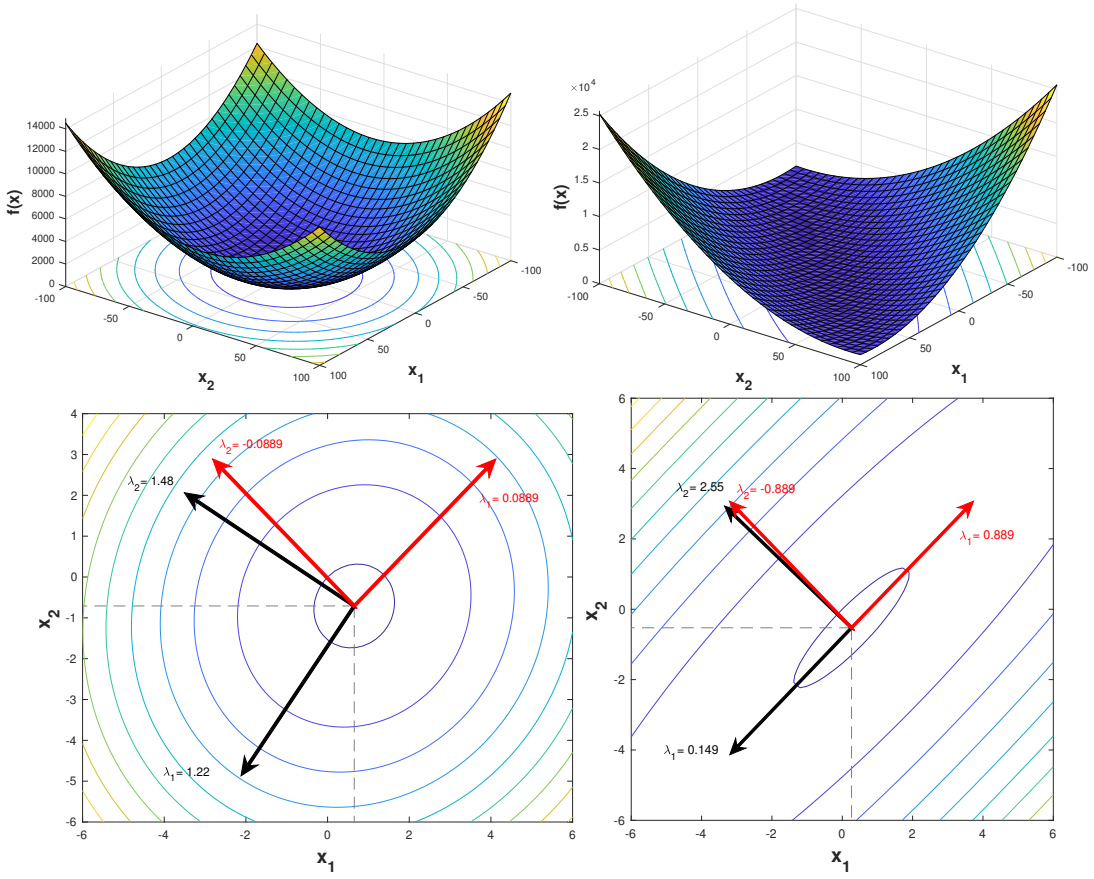


Figure 3.17: Quadratic functions of two symmetric positive definite matrices with eigenvalues of different magnitudes. The black arrows correspond to the eigenvectors of matrix $\mathbf{A}$, while the red arrows represent the eigenvectors of a Jacobi preconditioned matrix $\mathbf{I} - \mathbf{D}^{-1}\mathbf{A}$.

The quadratic function on the left belongs to the matrix $\begin{bmatrix} 1.4 & -0.12 \\ -0.12 & 1.3 \end{bmatrix}$, and it has two eigenvalues with similar magnitudes and both are positive (positive definite matrix). The shape of the isocontours is almost circular. The quadratic function on the left corresponds to matrix $\begin{bmatrix} 1.4 & -1.2 \\ -1.2 & 1.3 \end{bmatrix}$, and its shape is more stretched. The eigenvalues are of different order of magnitude and it can be seen that the larger eigenvalue belongs to the eigenvector which coincides with the minor axis of the elliptic isocontour. Steepest descent will quickly converge for the first shape of the quadratic function, since the residual at every point is directed towards the centre of the circle. To show the convergence limitations, the error is decomposed

into a sum of components which coincide with the eigenvectors of the coefficient matrix $\mathbf{A}$:

$$\mathbf{e}^{(k)} = \sum_{j=1}^{n} \zeta_j \mathbf{v}_j,$$

and the same can be done for the residual:

$$\mathbf{r}^{(k)} = \mathbf{A}\mathbf{e}^{(k)} = \sum_{j=1}^{n} \zeta_j \lambda_j \mathbf{v}_j,$$

where ζ_j is the length of the component in the direction of eigenvector $\mathbf{v}_j$ and λ_j are the eigenvalues corresponding to $\mathbf{v}_j$. The matrix is assumed to be symmetric, which means it has n linearly independent mutually orthogonal eigenvectors ($\mathbf{v}_i^T \mathbf{v}_j = 0$). The length of the residual is scaled with the eigenvalue: $\lambda_j \zeta_j$. All eigenvectors are normalized to have unit length ($\mathbf{v}_i^T \mathbf{v}_i = 1$). The iteration of steepest descent in terms of the error can be written as:

$$\begin{aligned}\mathbf{e}^{(k+1)} &= \mathbf{e}^{(k)} + \alpha^{(k)} \mathbf{r}^{(k)} = \mathbf{e}^{(k)} + \frac{(\mathbf{r}^{(k)})^T \mathbf{r}^{(k)}}{(\mathbf{r}^{(k)})^T \mathbf{A}\mathbf{r}^{(k)}} \mathbf{r}^{(k)} \\ &= \mathbf{e}^{(k)} + \frac{\sum_{j=1}^{n} \zeta_j^2 \lambda_j^2}{\sum_{j=1}^{n} \zeta_j^2 \lambda_j^3} \mathbf{r}^{(k)}. \end{aligned} \tag{3.100}$$

If all the eigenvalues are equal (circular isocontours of the quadratic function):

$$\mathbf{e}^{(k+1)} = \mathbf{e}^{(k)} + \frac{\lambda^2 \sum_{j=1}^{n} \zeta_j^2}{\lambda^3 \sum_{j=1}^{n} \zeta_j^2} (\lambda \mathbf{e}^{(k)}) = 0. \tag{3.101}$$

Thus, the error is eliminated in the first iteration and the convergence is instant. Another way to look at the convergence is to remember that the eigenvectors of the matrix do not rotate when multiplied by the matrix. If the initial solution of the system lies on an eigenvector, i.e. the axis of the elliptic isocontour, the error consists of a single eigenvector component. The convergence will also be instant, since the residual is a scaled eigenvector ($\mathbf{A}\mathbf{e} = \lambda \mathbf{e}$) and points to the correct solution. In general, if there are multiple components of the error with different eigenvalues, there is no such α which will eliminate all the error components at once. The priority in eliminating the error is given to longer components, which can be seen from the weighted average of $\frac{1}{\lambda_j}$ in Eqn. (3.100). The convergence depends on the ratio of eigenvalues, i.e. the condition number of the (symmetric, positive definite) matrix:

$$\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}. \tag{3.102}$$

With larger condition numbers, the quadratic function becomes more elongated. This is the cause of the main drawback od steepest descent: the method will do multiple steps in the same direction to reach the correct solution, shown in Fig. 3.16. To decrease the number of iterations, it is neccessary to do a single step in one direction and completely eliminate the corresponding error component. The application of this idea is described in the following section.

3.3.2. Conjugate Gradient Method

As described in the previous section, method of steepest descent is used for solving the linear system as a minimisation problem. The method utilises the residual, which is the direction of the largest decrease of the quadratic function, Eqn. (3.89). For a symmetric positive definite coefficient matrix, the solution of the linear system coincides with the minimum of the quadratic function. Using the residual as the search direction does not yield fast convergence, since it is neccesary to correct the solution in a single direction multiple times.

The idea is to use n mutually orthogonal search directions and eliminate each of the n components of the error in one step. This is achieved by locating the point on the line corresponding to the search direction which minimises the quadratic function. This is similar to line search, but using mutually orthogonal search directions means that the final solution will be reached in maximally n steps, because we cannot step in the same direction more than once. Using the directional derivative and setting it to zero yields:

$$\begin{aligned}\frac{d}{d\alpha}f(\mathbf{x}^{(1)}) &= f'(\mathbf{x}^{(1)})^T\frac{d}{d\alpha}(\mathbf{x}^{(1)}) = f'(\mathbf{x}^{(1)})^T\frac{d}{d\alpha}(\mathbf{x}^0+\alpha\mathbf{d}^0)\\ &= f'(\mathbf{x}^{(1)})^T\mathbf{d}^{(0)} = (\mathbf{r}^{(1)})^T\mathbf{d}^{(0)} = 0. \end{aligned} \tag{3.103}$$

Thus, the minimum is found where the residual is orthogonal to the search direction. If the residual is written in terms of the scaled error, the definition of *conjugate* or $\mathbf{A}$*–orthogonal* vectors appears:

$$\mathbf{d}^{(k)}\mathbf{A}\mathbf{e}^{(k+1)} = 0. \tag{3.104}$$

The search direction is orthogonal to the residual, and it is $\mathbf{A}$–orthogonal to the error. It can be shown that by defining the search directions to be mutually

conjugate, correct solution is also reached in n iterations. If the function is minimised at some point along the direction $\mathbf{d}^{(0)}$, the gradient of the function at that point is equal to zero. The search in the new direction should not affect the gradient of the previous step, since the function is already minimised in that previous direction. The change of the gradient of the function can be expressed as:

$$\begin{aligned}\Delta f'(\mathbf{x}) &= f'(\mathbf{x}^{(1)}) - f'(\mathbf{x}^{(0)}) = (\mathbf{A}\mathbf{x}^{(1)} - \mathbf{b}) - (\mathbf{A}\mathbf{x}^{(0)} - \mathbf{b}) \\ &= \mathbf{A}(\mathbf{x}^{(0)} + \mathbf{d}^{(0)}) - \mathbf{A}\mathbf{x}^{(0)} = \mathbf{A}\mathbf{d}^{(0)}.\end{aligned}$$

Since the step in the next direction $\mathbf{d}^{(1)}$ shouldn't affect the gradient in direction $\mathbf{d}^{(0)}$, the change of the gradient $\Delta f'(\mathbf{x})$ should be orthogonal to the new direction:

$$\mathbf{d}^{(1)}\mathbf{A}\mathbf{d}^{(0)} = 0, \tag{3.105}$$

which is identical to Eqn. (3.104). Thus, defining the search directions to be **A**–orthogonal, produces an iteration which will minimise the function (find the solution of the system) in n steps, since the minimum in one direction won't be compromised by the minimum in another direction.

To ensure that the method steps in a certain direction only once, the component of the error in that direction should be eliminated, i.e. equal to zero. Thus, the remaining error is **A**–orthogonal to the search direction, and the length of the next step can be obtained:

$$(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{e}^{(k+1)} = 0,$$

$$(\mathbf{d}^{(k)})^T\mathbf{A}(\mathbf{e}^{(k)} + \alpha^{(k)}\mathbf{d}^{(k)}) = 0,$$

$$\alpha^{(k)} = -\frac{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{e}^{(k)}}{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)}} = -\frac{(\mathbf{d}^{(k)})^T\mathbf{r}^{(k)}}{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)}}. \tag{3.106}$$

The construction of the sequence of mutually **A**–orthogonal vectors is done using the Gram–Schmidt process [2]:

- The process begins with a set of linearly independent vectors:

$$\mathbf{u}^{(0)}, \mathbf{u}^{(1)}, ..., \mathbf{u}^{(n-1)}.$$

- To obtain an orthonormal set, take the previous directions $\mathbf{d}^{(k)}$ and calculate the $\mathbf{A}$–orthogonal projection of the new vector $\mathbf{u}^{(i)}$ onto each previous direction
$$\frac{(\mathbf{u}^{(i)})^T\mathbf{A}\mathbf{d}^{(j)}}{(\mathbf{d}^{(j)})^T\mathbf{A}\mathbf{d}^{(j)}}\mathbf{d}^{(j)}. \tag{3.107}$$
- Subtract the components parallel to the previous search directions
$$\mathbf{d}^{(i)} = \mathbf{u}^{(i)} - \sum_{j=0}^{i-1}\frac{(\mathbf{u}^{(i)})^T\mathbf{A}\mathbf{d}^{(j)}}{(\mathbf{d}^{(j)})^T\mathbf{A}\mathbf{d}^{(j)}}\mathbf{d}^{(j)}. \tag{3.108}$$
- Normalise the new direction by dividing it with its magnitude.
$$\mathbf{d}^{(i)} = \frac{\mathbf{d}^{(i)}}{\|\mathbf{d}^{(i)}\|}. \tag{3.109}$$

In order to calculate the projection of the new search direction onto the old ones, it is necessary to store all the previous vectors in memory, and the complexity of the entire procedure is estimated at $\mathcal{O}(n^3)$ [57]. In order to improve the method, similar to steepest descent, use the residuals for the starting set of linearly independent vectors: $\mathbf{u}^{(i)} = \mathbf{r}^{(i)}$. The residuals have several useful properties:

- Residuals are orthogonal to previous search directions. From the fact that the error is reduced one by one component, and length of each component is $\alpha^{(k)}$:
$$\begin{aligned}
\mathbf{e}^{(i)} &= \sum_{k=i}^{n-1}\alpha^{(k)}\mathbf{d}^{(k)} \quad \backslash\cdot(\mathbf{d}^{(j)})^T\mathbf{A},\\
(\mathbf{d}^{(j)})^T\mathbf{A}\mathbf{e}^{(i)} &= \sum_{k=i}^{n-1}\alpha^{(k)}\underbrace{(\mathbf{d}^{(j)})^T\mathbf{A}\mathbf{d}^{(k)}}_{=0,\ \mathbf{A}\text{–orthogonality}},\\
(\mathbf{d}^{(j)})^T\mathbf{r}^{(i)} &= 0 \quad j<i.
\end{aligned} \tag{3.110}$$
- Residuals are orthogonal to the initial set of linearly independent vectors. From the Gram–Schmidt procedure:
$$\begin{aligned}
\mathbf{d}^{(i)} &= \mathbf{u}^{(i)} - \sum_{k=0}^{i-1}\frac{(\mathbf{u}^{(i)})^T\mathbf{A}\mathbf{d}^{(k)}}{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)}}\mathbf{d}^{(k)} \quad \backslash\cdot\mathbf{r}^{(j)},\\
\underbrace{\mathbf{d}^{(i)}r^{(j)}}_{=0} &= \mathbf{u}^{(i)}\mathbf{r}^{(j)} - \sum_{k=0}^{i-1}\frac{(\mathbf{u}^{(i)})^T\mathbf{A}\mathbf{d}^{(k)}}{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)}}\underbrace{\mathbf{d}^{(k)}\mathbf{r}^{(j)}}_{=0},\\
\mathbf{u}^{(i)}\mathbf{r}^{(j)} &= 0 \quad i<j.
\end{aligned} \tag{3.111}$$

If $i = j$:

$$\underbrace{\mathbf{d}^{(i)}\mathbf{A}\mathbf{e}^{(i)}}_{\neq 0} = \mathbf{u}^{(i)}\mathbf{A}\mathbf{e}^{(i)} - \sum_{k=0}^{i-1} \frac{(\mathbf{u}^{(i)})^T\mathbf{A}\mathbf{d}^{(k)}}{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)}} \underbrace{\mathbf{d}^{(k)}\mathbf{A}\mathbf{e}^{(i)}}_{=0}$$
$$\mathbf{d}^{(i)}\mathbf{r}^{(i)} = \mathbf{u}^{(i)}\mathbf{r}^{(i)}. \tag{3.112}$$

- Residual is a linear combination of the previous residual and $\mathbf{A}\mathbf{d}^{(k)}$:

$$\mathbf{r}^{(k+1)} = \mathbf{A}\mathbf{e}^{(k+1)} = \mathbf{A}(\mathbf{e}^{(k)} + \alpha^{(k)}\mathbf{d}^{(k)}) = \mathbf{r}^{(k)} + \alpha^{(k)}\mathbf{A}\mathbf{d}^{(k)}. \tag{3.113}$$

For each new direction, the approximation of the solution comes closer to the actual solution, i.e. the method always finds the optimal point in the space (combination of vectors) where it is possible to explore (new dimensions are added to the space with each new direction). Gram–Schmidt procedure guarantees that the initial set of vectors $\mathbf{u}^{(i)}$ spans the same space (hyperplane) as the orthonormal set of search directions $\mathbf{d}^{(i)}$. If the search direction is defined using the residuals, the space is defined as:

$$\mathcal{D}^{(k)} = \text{span}\{\mathbf{d}^{(0)}, \mathbf{d}^{(0)}, \mathbf{d}^{(2)}, \ldots, \mathbf{d}^{(k-1)}\} = \text{span}\{\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \mathbf{r}^{(2)}, \ldots, \mathbf{r}^{(k-1)}\},$$

where span denotes a vector space which contains all the linear combinations of vectors in the curly brackets.
Each new iteration (step in new direction) adds a new dimension to the solution space (hyperplane spanned by search directions $\mathbf{d}$). According to Eqn. (3.113), new residual $\mathbf{r}^{(k+1)}$ is a linear combination of the previous residual $\mathbf{r}^{(k)}$ and $\mathbf{A}\mathbf{d}^{(k)}$, which means the space for the approximation of the solution $\mathcal{D}^{(k+1)}$ can be written as a union of the previous space and the new search direction:

$$\mathcal{D}^{(k+1)} = \text{span}\{\mathcal{D}^{(k)}, \mathbf{r}^{(k+1)}\} = \text{span}\{\mathcal{D}^{(k)}, \underbrace{\mathbf{r}^{(k)}}_{\text{already in } \mathcal{D}^{(k)}} + \alpha^{(k)}\mathbf{A}\mathbf{d}^{k}\} = \text{span}\{\mathcal{D}^{(k)}, \mathbf{A}\mathbf{d}^{(k)}\}.$$

Since the residuals are chosen as the initial set of search vectors, and each new residual contains the old residual and a search direction multiplied by matrix $\mathbf{A}$, it can be shown by mathematical induction that each subsequent step adds a new dimension equal to the initial residual $\mathbf{r}^{(0)}$ multiplied by the power of the

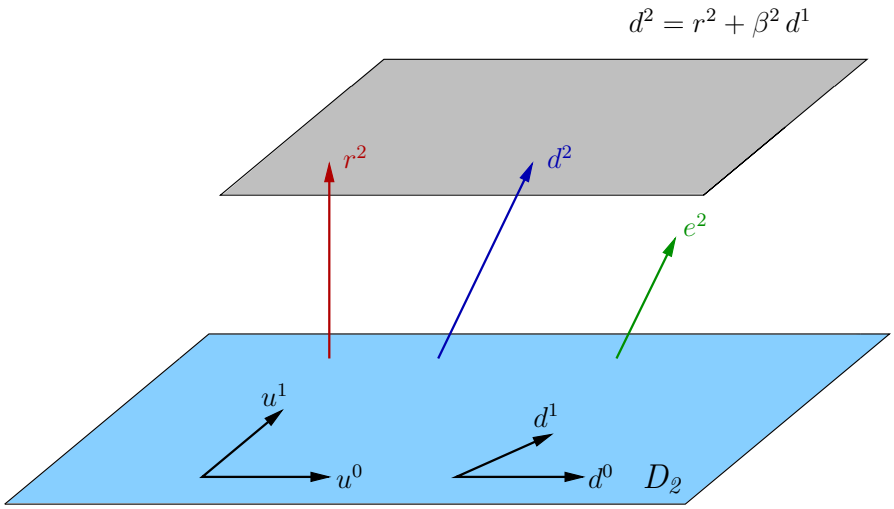


Figure 3.18: Subspace $\mathcal{D}^{(2)}$ for the approximation of the solution in the second iteration is spanned by the initial vectors $\mathbf{u}^{(0)}$ and $\mathbf{u}^{(1)}$. It is also spanned by $\mathbf{A}$–orthogonal vectors $\mathbf{d}^{(0)}$ and $\mathbf{d}^{(1)}$. The error $\mathbf{e}^{(2)}$ is $\mathbf{A}$–orthogonal to $\mathcal{D}^{(2)}$, while the residual $\mathbf{r}^{(2)}$ is orthogonal. The new search direction is a linear combination of $\mathbf{r}^{(2)}$ and $\mathbf{d}^{(1)}$ and it is $\mathbf{A}$–orthogonal to $\mathcal{D}^{(2)}$.

coefficient matrix $\mathbf{A}$. Thus, the approximation space is called the *Krylov subspace* due to the continuous repetition of multiplying a vector with the same matrix:

$$\mathcal{D}^{(k)} = \text{span}\{\mathbf{d}^{(0)}, \mathbf{A}\mathbf{d}^{(0)}, \mathbf{A}^{(2)}\mathbf{d}^{(0)}, \ldots, \mathbf{A}^{(k-1)}\mathbf{d}^{(0)}\} \tag{3.114}$$

$$= \text{span}\{\mathbf{r}^{(0)}, \mathbf{A}\mathbf{r}^{(0)}, \mathbf{A}^{(2)}\mathbf{r}^{(0)}, \ldots, \mathbf{A}^{(k-1)}\mathbf{r}^{(0)}\}. \tag{3.115}$$

The solution of the linear system, i.e. the minimum of the quadratic function is a linear combination of the vectors in space $\mathcal{D}^{(i)}$. The subspace has a property which makes storing all the search directions redundant: since the residual $\mathbf{r}^{(k+1)}$ is orthogonal to all the previous search directions, it is orthogonal to the whole space $\mathcal{D}^{(k+1)}$ spanned by them. As shown before, the space $\mathcal{D}^{(k+1)}$ contains:

$$\mathcal{D}^{(k+1)} = \text{span}\{\mathcal{D}^{(k)}, \mathbf{A}\mathcal{D}^{(k)}\}.$$

$\mathbf{r}^{(k+1)}$ is conjugate to $\mathcal{D}^{(k)}$ and all the search directions $\mathbf{d}^0, \ldots, \mathbf{d}^{(k-1)}$ it contains. It is only necessary to make $\mathbf{d}^{(k)}$ conjugate to $\mathbf{r}^{(k+1)}$, thus the projection is done

onto only one vector. The projection operator $\beta^{(k)}$ can be calculated as:

$$\begin{aligned}
\mathbf{r}^{(k+1)} &= \mathbf{r}^{(k)} + \alpha^{(k)}\mathbf{A}\mathbf{d}^{(k)} \quad \backslash \cdot \mathbf{r}^{(i)}, \\
\mathbf{r}^{(i)}\mathbf{r}^{(k+1)} &= \mathbf{r}^{(i)}\mathbf{r}^{(k)} + \alpha^{(k)}\mathbf{r}^{(i)}\mathbf{A}\mathbf{d}^{(k)}, \\
\alpha^{(k)}\mathbf{r}^{(i)}\mathbf{A}\mathbf{d}^{(k)} &= \mathbf{r}^{(i)}\mathbf{r}^{(k+1)} - \mathbf{r}^{(i)}\mathbf{r}^{(k)}, \\
\mathbf{r}^{(i)}\mathbf{A}\mathbf{d}^{(k)} &= \begin{cases} \frac{-1}{\alpha^{(k)}}\mathbf{r}^{(k)}\mathbf{r}^{(k)} & \text{if } i = k, \\ \frac{1}{\alpha^{(k)}}\mathbf{r}^{(k+1)}\mathbf{r}^{(k+1)} & \text{if } i = k+1, \\ 0 & \text{otherwise} \end{cases} \\
\frac{(\mathbf{u}^{(i)})^T\mathbf{A}\mathbf{d}^{(j)}}{(\mathbf{d}^{(j)})^T\mathbf{A}\mathbf{d}^{(j)}} &= \begin{cases} \frac{1}{\alpha^{(k)}}\frac{(\mathbf{r}^{(k+1)})^T\mathbf{r}^{(k+1)}}{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)}} & \text{if } i = k+1, \\ 0 & \text{if } i > k+1. \end{cases},
\end{aligned}$$

Inserting the expression for $\alpha^{(k)}$, Eqn. (3.106) into the projection operator yields:

$$\begin{aligned}
\beta^{(k)} &= \frac{1}{\alpha^{(k)}}\frac{(\mathbf{r}^{(k+1)})^T\mathbf{r}^{(k+1)}}{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)}} = \frac{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)}}{(\mathbf{d}^{(k)})^T\mathbf{r}^{(k)}} \cdot \frac{(\mathbf{r}^{k+1})^T\mathbf{r}^{(k+1)}}{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)}} \\
&= \frac{(\mathbf{r}^{(k+1)})^T\mathbf{r}^{(k+1)}}{(\mathbf{d}^{(k)})^T\mathbf{r}^{(k)}} = \frac{(\mathbf{r}^{(k+1)})^T\mathbf{r}^{(k+1)}}{(\mathbf{r}^{(k)})^T\mathbf{r}^{(k)}},
\end{aligned} \tag{3.116}$$

since $(\mathbf{d}^{(k)})^T\mathbf{r}^{(k)} = (\mathbf{u}^{(k)})^T\mathbf{r}^{(k)}$, and $\mathbf{u}^{(k)} = \mathbf{r}^{(k)}$. Finally, a single iteration of the conjugate gradient method can be written as:

1. In the beginning assume $\mathbf{x}^{(0)} = 0$, and then $\mathbf{d}^{(0)} = \mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$.

2. Calculate the length of the step in direction $\mathbf{d}^{(k)}$:
$$\alpha^{(k)} = \frac{(\mathbf{r}^{(k)})^T\mathbf{r}^{(k)}}{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)}}.$$

3. Calculate the new solution:
$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{d}^{(k)}.$$

4. Calculate the new residual:
$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} + \alpha^{(k)}\mathbf{A}\mathbf{d}^{(k)}.$$

5. Calculate the projection operator of the new residual onto the previous search direction:
$$\beta^{(k)} = \frac{(\mathbf{r}^{(k+1)})^T\mathbf{r}^{(k+1)}}{(\mathbf{r}^{(k)})^T\mathbf{r}^{(k)}}\mathbf{d}^{(k)}.$$

6. Calculate the new search direction by making it **A**–orthogonal to the new residual:
$$\mathbf{d}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta^{(k)}\mathbf{d}^{(k)}.$$

7. Return to step 2 if the convergence criterion is not satisfied.

The conjugate gradient method will converge to the correct solution in n iterations, where n is the dimension of the system. However, for applications in computational fluid dynamics, even this number is inadequate, since the dimension of the system depends on the number of cells and, for implicitly coupled systems, the number of physical variables. Usually, it is not necessary to conduct the maximum number of iterations, since, as it was seen with steepest descent, the largest components of the error will be eliminated first. A satisfactory approximation of the solution can be achieved in significantly less than n iterations. The convergence of the method can also be drastically improved by manipulating the shape of the quadratic function to be more spherical, i.e. closer to the quadratic function of the identity matrix. This transformation of the coefficient matrix **A** is called preconditioning and it is described in the following section.

3.3.3. Preconditioning

The idea of preconditioning is to transform the coefficient matrix of the linear system $\mathbf{Ax} = \mathbf{b}$ so that the system becomes easier to solve:

$$\mathbf{P}_\mathbf{A}^{-1}\mathbf{A}. \tag{3.117}$$

Matrix $\mathbf{P}_\mathbf{A}$ is called a *preconditioner*. The ideal preconditioner is identical to the coefficient matrix, $\mathbf{P}_\mathbf{A} = \mathbf{A}$:

$$\begin{aligned}\mathbf{Ax} = \mathbf{b} \rightarrow \mathbf{P}_\mathbf{A}^{-1}\mathbf{Ax} &= \mathbf{P}_\mathbf{A}^{-1}\mathbf{b}\\ \mathbf{A}^{-1}\mathbf{Ax} &= \mathbf{P}_\mathbf{A}^{-1}\mathbf{b}\\ \mathbf{Ix} &= \mathbf{P}_\mathbf{A}^{-1}\mathbf{b},\end{aligned}$$

because when multiplied by $\mathbf{P}_\mathbf{A}^{-1}$ the coefficient matrix reduces to the identity matrix **I** and the system can be solved in a single iteration. This ideal case is unfeasible since inverting the coefficient matrix is analogous to solving the

system. The desired property of the preconditioner is to be as close as possible to the original coefficient matrix $\mathbf{A}$, but that the construction and inversion of $\mathbf{P_A}$ is not computationally expensive. Also, to achieve an efficient solver, the system $\mathbf{P_A y} = \mathbf{z}$ should be much easier to solve than the original system [61]. The transformed matrix $\mathbf{P_A^{-1}A}$ will not be explicitly formed, except for some simple preconditioning techniques. Instead, two linear systems are solved:

$$\begin{aligned} & & \mathbf{P_A^{-1}Ax} &= \mathbf{P_A^{-1}b} \\ \text{substitute } \mathbf{Ax} = \mathbf{y},\ \mathbf{P_A^{-1}b} = \mathbf{z} \rightarrow\ & & \mathbf{P_A^{-1}y} &= \mathbf{z} \\ & & \mathbf{y} &= \mathbf{P_A z} \\ & & \mathbf{x} &= \mathbf{A^{-1}y}, \end{aligned}$$

where $\mathbf{A}^{-1}$ denotes (approximately) solving the system, rather than calculating the inverse of $\mathbf{A}$. Since the preconditioner is applied from the left side of the coefficient matrix, this is called *left preconditioning*. The application of left preconditioning affects the residual: $\mathbf{r} = \mathbf{P_A^{-1}}(\mathbf{b} - \mathbf{Ax})$ and the convergence criteria should be carefully formulated since the preconditioned residual may be very different from the true residual. *Right preconditioning* does not affect the right hand side of the system:

$$\mathbf{AP_A^{-1}y} = \mathbf{b},\ \text{where } \mathbf{P_A x} = \mathbf{y}.$$

However, the error in $\mathbf{y}$ is scaled by the preconditioner, $\|\mathbf{P_A^{-1}}(\mathbf{y} - \mathbf{y}_{\text{approx}})\|$, compared to the error in $\mathbf{x}$, $\|\mathbf{x} - \mathbf{x}_{\text{approx}}\|$. Left and right preconditioning are special cases of *two–sided preconditioning*, where the preconditioner is split into two factors:

$$\begin{aligned} \mathbf{P_A} &= (\mathbf{P_A})_1(\mathbf{P_A})_2 \rightarrow \\ (\mathbf{P_A})_1^{-1}\mathbf{A}(\mathbf{P_A})_2^{-1}\mathbf{y} &= \mathbf{P_A^{-1}b} \\ \text{where } (\mathbf{P_A})_2\mathbf{x} &= \mathbf{y}. \end{aligned}$$

To obtain left preconditioning, $(\mathbf{P_A})_2$ is equal to the identity matrix $\mathbf{I}$, while for right preconditioning, $(\mathbf{P_A})_1$ is equal to identity.

For different iterative algorithms, the desired effect of the preconditioner is not the same. For example, for fixed–point methods, the objective is to achieve

that the norm of the iteration matrix $\|\mathbf{I} - \mathbf{P}_\mathbf{A}^{-1}\mathbf{A}\|$ is much smaller than 1, which guarantees a fast reduction of the norm of the error. For conjugate gradients, it is desirable to transform the shape of the quadratic function of the coefficient matrix $\mathbf{A}$ to be less elongated in certain directions and the isocontours to be circular. This means that there are duplicated eigenvalues or the eigenvalues are very similar in magnitude (*clustered* together).

Jacobi and Gauss–Seidel splittings of matrix $\mathbf{A}$ are considered as preconditioners for a general fixed–point iteration, Eqn. (3.22):

$$\begin{aligned} \mathbf{x} &= (\mathbf{I} - \mathbf{A})\mathbf{x} + \mathbf{b} \\ \text{add a preconditioner: } \mathbf{P}_\mathbf{A}\mathbf{x} &= (\mathbf{P}_\mathbf{A} - \mathbf{A})\mathbf{x} + \mathbf{b} \\ \mathbf{x}^{(k)} &= (\mathbf{I} - \mathbf{P}_\mathbf{A}^{-1}\mathbf{A})\mathbf{x}^{(k-1)} + \mathbf{b}. \end{aligned}$$

The simplest choice of approximating $\mathbf{A}$ is to use the diagonal of the matrix $\mathbf{D}_\mathbf{A}$ as $\mathbf{P}_\mathbf{A}$. This is called *diagonal preconditioning* and it is equivalent to the Jacobi method. The inverse of the preconditioning matrix can be easily calculated, since it is a diagonal matrix: the elements on the diagonal are equal to reciprocal value of the diagonal elements of $\mathbf{A}$. Diagonal preconditioning is an anisotropic scaling along the coordinate axes $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$. In comparison, using the coefficient matrix $\mathbf{A}$ as a preconditioner is equivalent to scaling the quadratic function in the direction of the eigenvectors.

Fig. 3.19 on the left shows the quadratic function of a symmetric positive definite matrix $\begin{bmatrix} 1.4 & -1.2 \\ -1.2 & 10 \end{bmatrix}$ and the eigenvectors with the corresponding eigenvalues. The right part of Fig. 3.19 shows the same, diagonally preconditioned matrix $\begin{bmatrix} 1 & -0.857 \\ -0.12 & 1 \end{bmatrix}$ which is still positive definite, but no longer symmetric. Diagonal preconditioner is easy and inexpensive to compute, and it is usually directly applied to the coefficient matrix $\mathbf{A}$. But, the preconditioned matrix $\mathbf{D}_\mathbf{A}^{-1}\mathbf{A}$ does not have to be symmetric, as it depends on the magnitude of diagonal elements of $\mathbf{A}$. In the majority of cases, diagonal elements of the finite volume matrix will not have the same magnitude.

To demonstrate the effect of explicit diagonal preconditioning on the eigenspectrum, and generally estimating the range in which the eigenvalues lie, we shall use the *Gersghorin's theorem*. The theorem states that each row i of a matrix $\mathbf{A}$ can be represented in the form of a disc: the centre of the disc is equiv-

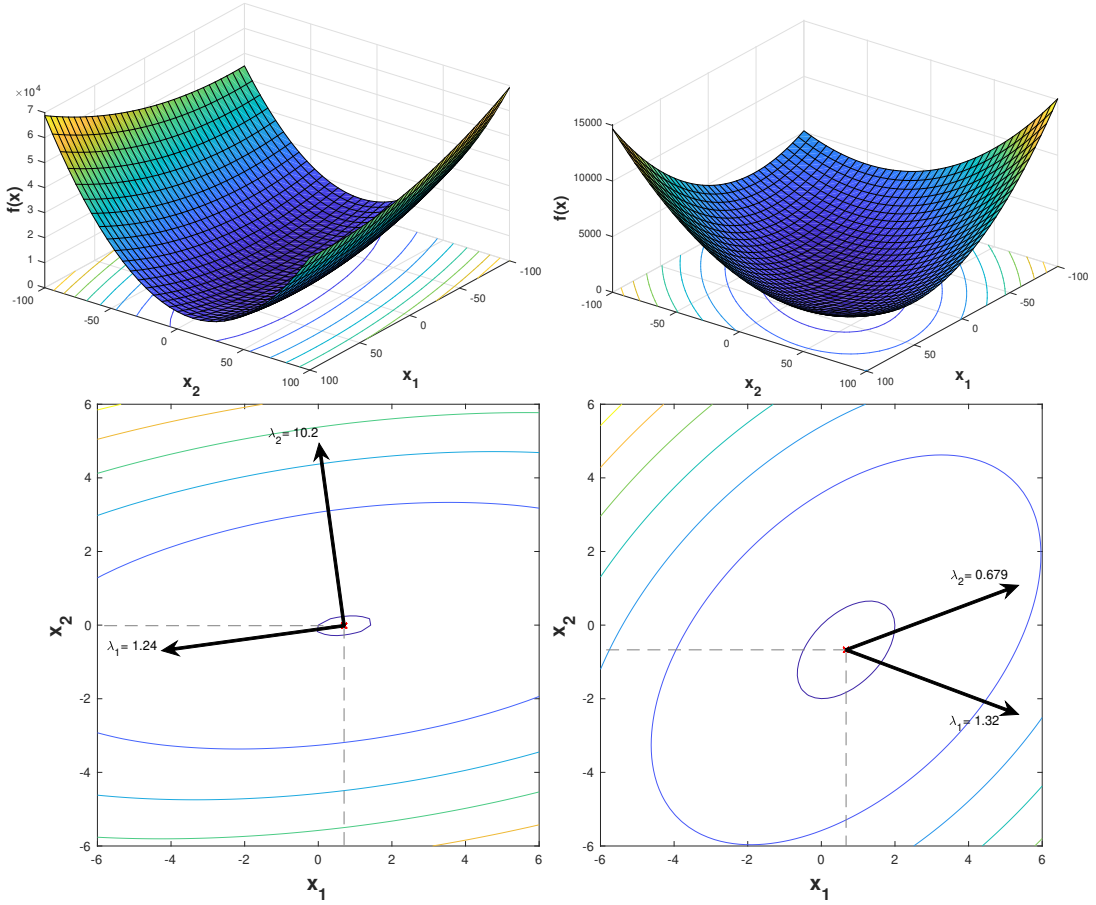


Figure 3.19: On the left: quadratic function of a symmetric positive definite matrix and eigenvectors with the corresponding eigenvalues. On the right: diagonally preconditioned matrix with the corresponding eigenvectors and eigenvalues, no longer symmetric.

alent to the diagonal element, $c_{\mathrm{G}} = a_{ii}$, while the radius of the disc is equal to the sum of magnitudes of off–diagonal elements in that row, $r_{\mathrm{G}} = \sum_{i \neq j} |a_{ij}|$. All eigenvalues of the matrix $\mathbf{A}$ lie within the set of n discs corresponding to the rows or columns of that matrix. Also, if there is a disjoint subset of Gershgorin discs, meaning that the discs in that subset do not intersect any of the discs outside the subset, and if there is r non–concentric discs in the subset, then there are at least r distinct eigenvalues. Two examples are shown in Fig. 3.20, for matrices with real elements. The top figure corresponds to a diagonally dominant symmetric matrix where all the elements are of the same order of magnitude: $\begin{bmatrix} 1.4 & -1.2 \\ -1.2 & 1.3 \end{bmatrix}$. Two black circles are Gershgorin discs constructed from the matrix rows, and two distinct eigenvalues (black dots λ_1, λ_2) lie outside the intersection area of the

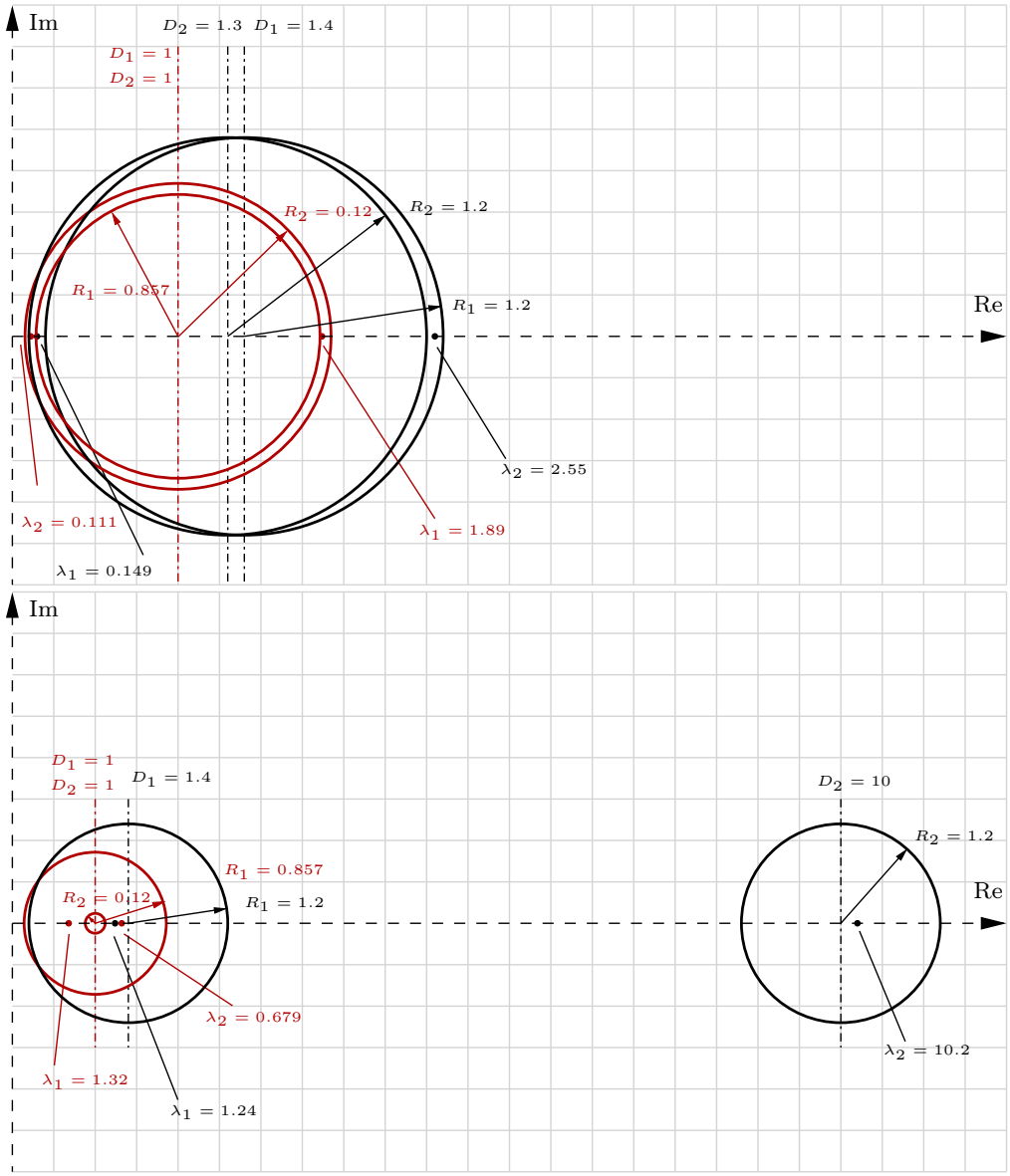


Figure 3.20: Demonstration of explicit diagonal preconditioning using the Gershgorin theorem: the eigenvalues of the preconditioned matrix (red) are clustered closer together in comparison to the eigenvalues of the original matrix (black).

discs. Red concentric circles are the discs of the diagonally preconditioned matrix. The centre of all discs of a diagonally preconditioned matrix are $c_G = 1$ and the radii are smaller compared to the discs of the original matrix. The clustering of the eigenvalues closer together is evident, even more so in the bottom part of Fig. 3.20, for the matrix with a strongly diagonally dominant row: $\begin{bmatrix} 1.4 & -1.2 \\ -1.2 & 10 \end{bmatrix}$. As

mentioned earlier, conjugate gradient method will converge in fewer iterations if there are multiple eigenvalues or if the eigenvalues are very similar in magnitude. However, conjugate gradient method works for symmetric positive definite matrices, and the preconditioned matrix should remain symmetric. To retain the symmetry of the preconditioned matrix, it is possible to factorise the symmetric positive definite matrix $\mathbf{P_A}$ as:

$$\mathbf{P_A} = \mathbf{E}\mathbf{E}^T.$$

Then matrices $\mathbf{P_A}^{-1}\mathbf{A}$ and $\mathbf{E}^{-1}\mathbf{A}(\mathbf{E}^T)^{-1}$ have the same eigenvalues λ, but the eigenvectors of $\mathbf{E}^{-1}\mathbf{A}(\mathbf{E}^T)^{-1}$ are scaled by $\mathbf{E}^T$ compared to eigenvectors v of $\mathbf{P_A}^{-1}\mathbf{A}$:

$$(\mathbf{E}^{-1}\mathbf{A}(\mathbf{E}^T)^{-1})(\mathbf{E}^T\mathbf{v}) = \mathbf{E}^T \underbrace{(\mathbf{E}^T)^{-1}\mathbf{E}^{-1}}\mathbf{A}\mathbf{v} = \mathbf{E}^T\mathbf{P_A}^{-1}\mathbf{A}\mathbf{v} = \mathbf{E}^T\lambda\mathbf{v}. \tag{3.118}$$

Since $\mathbf{E}^{-1}\mathbf{A}(\mathbf{E}^T)^{-1}$ is symmetric, the system can be transformed into:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \;\rightarrow\; \underbrace{\mathbf{E}^{-1}\mathbf{A}(\mathbf{E}^T)^{-1}}_{\tilde{\mathbf{A}}}\underbrace{\mathbf{E}^T\mathbf{x}}_{\tilde{\mathbf{x}}} = \underbrace{\mathbf{E}^{-1}\mathbf{b}}_{\tilde{\mathbf{b}}}, \tag{3.119}$$

and conjugate gradient can be applied to the transformed system to calculate $\tilde{\mathbf{x}}$. To avoid the explicit factorisation of $\mathbf{P}$, it is sufficient to introduce two substitutions:

$$\tilde{\mathbf{r}}^{(k)} = \mathbf{E}^{-1}\mathbf{r}^{(k)},$$
$$\tilde{\mathbf{d}}^{(k)} = \mathbf{E}^T\mathbf{d}^{(k)}$$

and insert them into the transformed algorithm:

1. Initialise the solution:

$$\begin{aligned} \tilde{\mathbf{x}}^{(0)} &= \mathbf{E}^T\mathbf{x}^{(0)} = 0, \\ \tilde{\mathbf{r}}^{(0)} &= \tilde{\mathbf{b}}, \\ \mathbf{E}^{-1}\mathbf{r}^{(0)} = \mathbf{E}^{-1}\mathbf{b} &\rightarrow \mathbf{r}^{(0)} = \mathbf{b}. \end{aligned} \tag{3.120}$$

2. Determine the initial search direction:

$$\begin{aligned} \tilde{\mathbf{d}}^{(0)} = \tilde{\mathbf{r}}^{(0)} &\rightarrow \mathbf{E}^T\mathbf{d}^{(0)} = \mathbf{E}^{-1}\mathbf{r}^{(0)}, \\ \mathbf{d}^{(0)} &= (\mathbf{E}^T)^{-1}\mathbf{E}^{-1}\mathbf{r}^{(0)} = \mathbf{P_A}^{-1}\mathbf{r}^{(0)}. \end{aligned} \tag{3.121}$$

3. Calculate the length of the step:

$$\begin{aligned}\tilde{\alpha}^{(k)} &= \frac{(\tilde{\mathbf{r}}^{(k)})^T\tilde{\mathbf{r}}^{(k)}}{(\tilde{\mathbf{d}}^{(k)})^T\tilde{\mathbf{A}}\tilde{\mathbf{d}}^{(k)}} = \frac{(\mathbf{E}^{-1}\mathbf{r}^{(k)})^T\mathbf{E}^{-1}\mathbf{r}^{(k)}}{(\mathbf{E}^T\mathbf{d}^{(k)})^T\mathbf{E}^{-1}\mathbf{A}(\mathbf{E}^T)^{-1}\mathbf{E}^T\mathbf{d}^{(k)}} \\ &= \frac{(\mathbf{r}^{(k)})^T\overbrace{(\mathbf{E}^T)^{-1}\mathbf{E}^{-1}}^{\mathbf{P}_\mathbf{A}^{-1}}\mathbf{r}^{(k)}}{(\mathbf{d}^{(k)})^T\underbrace{\mathbf{E}\mathbf{E}^{-1}}_{\mathbf{I}}\mathbf{A}\underbrace{(\mathbf{E}^T)^{-1}\mathbf{E}^T}_{\mathbf{I}}\mathbf{d}^{(k)}} = \frac{(\mathbf{r}^{(k)})^T\mathbf{P}_\mathbf{A}^{-1}\mathbf{r}^{(k)}}{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)}}.\end{aligned} \tag{3.122}$$

4. Update the solution:

$$\begin{aligned}\tilde{\mathbf{x}}^{(k+1)} &= \tilde{\mathbf{x}}^{(k)} + \tilde{\alpha}^{(k)}\tilde{\mathbf{d}}^{(k)}, \\ \rightarrow \mathbf{E}^T\mathbf{x}^{(k+1)} &= \mathbf{E}^T\mathbf{x}^{(k)} + \tilde{\alpha}^{(k)}\mathbf{E}^T\mathbf{d}^{(k)}, \\ \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \tilde{\alpha}^{(k)}\mathbf{d}^{(k)}.\end{aligned} \tag{3.123}$$

5. Calculate the residual:

$$\begin{aligned}\tilde{\mathbf{r}}^{(k+1)} &= \tilde{\mathbf{r}}^{(k)} + \tilde{\alpha}^{(k)}\tilde{\mathbf{A}}\tilde{\mathbf{d}}^{(k)}, \\ \rightarrow \mathbf{E}^{-1}\mathbf{r}^{(k+1)} &= \mathbf{E}^{-1}\mathbf{r}^{(k)} + \tilde{\alpha}^{(k)}\mathbf{E}^{-1}\mathbf{A}\underbrace{(\mathbf{E}^T)^{-1}\mathbf{E}^T}_{\mathbf{I}}\mathbf{d}^{(k)}, \\ \mathbf{r}^{(k+1)} &= \mathbf{r}^{(k)} + \tilde{\alpha}^{(k)}\mathbf{A}\mathbf{d}^{(k)}.\end{aligned} \tag{3.124}$$

6. Calculate the projection operator:

$$\begin{aligned}\tilde{\beta}^{(k)} &= \frac{(\tilde{\mathbf{r}}^{(k+1)})^T\tilde{\mathbf{r}}^{(k+1)}}{(\tilde{\mathbf{r}}^{(k)})^T\tilde{\mathbf{r}}^{(k)}} = \frac{(\mathbf{E}^{-1}\mathbf{r}^{(k+1)})^T\mathbf{E}^{-1}\mathbf{r}^{(k+1)}}{(\mathbf{E}^{-1}\mathbf{r}^{(k)})^T\mathbf{E}^{-1}\mathbf{r}^{(k)}} \\ &= \frac{(\mathbf{r}^{(k+1)})^T(\mathbf{E}^T)^{-1}\mathbf{E}^{-1}\mathbf{r}^{(k+1)}}{(\mathbf{r}^{(k)})^T(\mathbf{E}^T)^{-1}\mathbf{E}^{-1}\mathbf{r}^{(k)}} = \frac{(\mathbf{r}^{(k+1)})^T\mathbf{P}_\mathbf{A}^{-1}\mathbf{r}^{(k+1)}}{(\mathbf{r}^{(k)})^T\mathbf{P}_\mathbf{A}^{-1}\mathbf{r}^{(k)}}.\end{aligned} \tag{3.125}$$

7. Find the new search direction:

$$\begin{aligned}\tilde{\mathbf{d}}^{(k+1)} &= \tilde{\mathbf{r}}^{(k+1)} + \tilde{\beta}^{(k)}\tilde{\mathbf{d}}^{(k)}, \\ \mathbf{E}^T\mathbf{d}^{(k+1)} &= \mathbf{E}^{-1}\mathbf{r}^{(k+1)} + \tilde{\beta}^{(k)}\mathbf{E}^T\mathbf{d}^{(k)} \;\backslash\cdot(\mathbf{E}^T)^{-1}, \\ \mathbf{d}^{(k+1)} &= \mathbf{P}_\mathbf{A}^{-1}\mathbf{r}^{(k+1)} + \tilde{\beta}^{(k)}\mathbf{d}^{(k)}.\end{aligned} \tag{3.126}$$

8. Return to step 3 if the convergence criterion is not satisfied.

The additional cost of the preconditioned conjugate gradient method is the term $\mathbf{P}_\mathbf{A}^{-1}\mathbf{r}^{(k+1)}$ which is equivalent to solving a linear system. Thus, $\mathbf{P}_\mathbf{A}$ should be chosen in such a way that this system can be solved quickly and efficiently. Besides the aforementioned diagonal and Gauss–Seidel preconditioners, another popular method is the *incomplete lower–upper factorization, ILU* and it will be presented in the next section.

Incomplete Lower–Upper Preconditioning

Factoring a matrix into a product of lower triangular and upper triangular matrices is done using the *Gaussian elimination*. Gaussian elimination, Algorithm 3.2, is a direct solution method which eliminates the elements in the lower triangle of the matrix to obtain an upper triangular matrix, and the solution of the system is then trivial to calculate [2].

Algorithm 3.2. *Gaussian Elimination*

1: ***for*** $i = 2, \ldots, n$ ***do*** ▷ *Go into row i.*
2: ***for*** $k = 1, \ldots, i-1$ ***do*** ▷ *Access all previous rows k.*
3: $a_{ik} := \frac{a_{ik}}{a_{kk}}$ ▷ *Calculate the multiplier.*
4: ***for*** $j = k+1, \ldots, n$ ***do*** ▷ *Go through elements of row i.*
5: $a_{ij} := a_{ij} - a_{ik} \cdot a_{kj}$ ▷ *Update the element of row i with the contribution from row k.*
6: ***end for***
7: ***end for***
8: ***end for***

The multipliers used to eliminate the coefficents below the diagonal can be put into the corresponding positions of a lower triangular matrix which has a unit diagonal. Matrix $\mathbf{A}$ can then be written as a product of the obtained upper and lower triangular matrices:

$$\mathbf{A} = \mathbf{L}_\mathbf{A}\mathbf{U}_\mathbf{A}. \tag{3.127}$$

However, LU factorisation of a sparse matrix will generally produce dense factors, which is prohibitive in terms of storage. For preconditioning, it is sufficient to use an approximation $\tilde{\mathbf{A}}$ of the original matrix, thus it is not necessary to compute the exact factors and they can be truncated in some way. Thus, the product of

two factors $\tilde{\mathbf{A}}$ will not be the same as the original matrix $\mathbf{A}$, and there exists an error matrix $\mathbf{R}$:

$$\mathbf{A} = \underbrace{\tilde{\mathbf{A}}}_{\mathbf{LU}} - \mathbf{R}, \tag{3.128}$$

where $\mathbf{L}$ and $\mathbf{U}$ are approximations of $\mathbf{L_A}$ and $\mathbf{U_A}$, respectively. Incomplete lower–upper factorisation of a matrix is a modification of the Gaussian elimination where some of the off–diagonal elements in $\mathbf{L}$ and $\mathbf{U}$ are dropped, i.e. the factors have a different *fill–in* (structure or pattern). Factorisation by Gaussian elimination preserves the positive definiteness of the matrix, even when arbitrary off–diagonal elements are ignored [61]. There are two possibilities for deciding which elements will be dropped:

- using a threshold value (dynamic ILU), or
- using a predefined pattern of fill–in (static ILU).

The dynamic ILU uses a tolerance for deciding whether an element obtained in the factorisation is large enough to keep. In this method, the level of fill–in is difficult to control. This version is known as ILUT. In static ILU, the level of fill–in is usually defined using the pattern of the original matrix $\mathbf{A}$. For example, the factors will have non–zero elements only at the positions where $\mathbf{A}$ has non–zero elements. This algorithm is called ILU0 and the fill–in level is 0. Fill–in can be extended to a structure different than the structure of $\mathbf{A}$, and include more elements. The method is beneficial in terms of memory capacity, since it is easy to control the amount of stored data. Hybrid algorithms which employ the threshold strategy and predetermined fill–in of factors are also used. In the scope of this thesis, we decided to use a static ILU algorithm with arbitrary level of fill–in.

Algorithm 3.3. *Static ILU, and* $\mathbf{F}$ *is a set of locations where there is no fill*

1: ***for*** $i = 2, \ldots, n$ ***do***
2: ***for*** $k = 1, \ldots, i-1$ *and if* $(i,k) \notin \mathbf{F}$ ***do***
3: $a_{ik} := \frac{a_{ik}}{a_{kk}}$
4: ***for*** $j = k+1, \ldots, n$ *and for* $(i,j) \notin \mathbf{F}$ ***do***
5: $a_{ij} := a_{ij} - a_{ik} \cdot a_{kj}$
6: ***end for***

7: ***end for***

8: ***end for***

Algorithm 3.3 is the so–called IKJ (delayed–update) version of ILU, and the name comes from the ordering of the three for–loops. In this version, the rows of factors **L** and **U** are generated sequentially. When in row i, all previous rows k are accessed to use the diagonal element a_{kk} and eliminate the off–diagonal element in i which belongs to the lower triangle, as shown in centre of Fig. 3.21. In contrast, KIJ ordering of the loops would produce a computational pattern shown on the left in Fig. 3.21: multiple rows are modified using the elements from a single row.

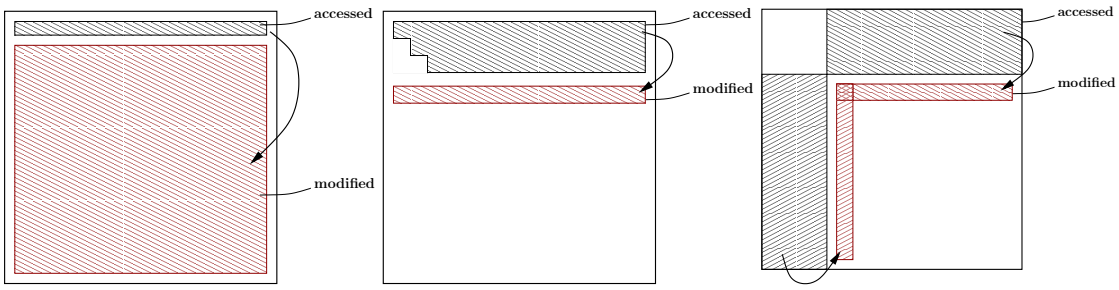


Figure 3.21: Versions of LU factorisation: KIJ (left), IKJ (center), Crout (right).

Depending on the matrix storage format and whether the static or dynamic variant of ILU is used, the general IKJ algorithm can be computationally very expensive. A more practical version of the IKJ algorithm is the Crout factorisation [62], Algorithm 3.4.

Algorithm 3.4. *Crout LU factorisation*

1: ***for*** $k = 1, \ldots, n$ ***do***

2: ***for*** $i = 1, \ldots, k-1$ *and if* $a_{ki} \neq 0$ ***do***

3: $a_{k,k:n} := a_{k,k:n} - a_{ki} \cdot a_{i,k:n}$

4: ***end for***

5: ***for*** $i = 1, \ldots, k-1$ *and if* $a_{ik} \neq 0$ ***do***

6: $a_{k+1:n,k} := a_{k+1:n,k} - a_{ik} \cdot a_{k+1:n,i}$

7: ***end for***

8: ***for*** $i = k+1, \ldots, n$ ***do***

9: $a_{ik} := \frac{a_{ik}}{a_{kk}}$
10: ***end for***
11: ***end for***

The computational pattern of Crout LU is shown on the right in Fig. 3.21. The algorithm can be viewed as an IJK variant for computing the upper triangle and a transpose of the same algorithm for calculating the lower triangle. The k^{th} step calculates the k^{th} row in **U** and k^{th} column of **L**. Thus, it is desirable to store the elements in **U** by rows and the elements in **L** by columns, as is the case with the LDU–matrix in OpenFOAM and this is a clear motivation for implementation.

In the scope of the thesis, a static Crout ILU algorithm is used: the sparsity of factors **L** and **U** is equivalent to the sparsity of matrix **A**. This is called fill–in level 0 (ILUC0), i.e. only the indices of columns belonging to first neighbours of the cell (sharing a common face) are non–zero. Fill–in level 1 extends the sparsity to include the positions in the matrix corresponding to neighbours of neighbours of the cell. One step further would be to include the indices of columns belonging to third neighbours (neighbours of neighbours of first neighbours of the cell), which is fill–in level 2. An illustration of how quickly the density of the factors increases is shown in Fig. 3.22: dark blue is the sparsity for fill–in level 0 (original matrix), dark blue and light blue combined correspond to fill–in level 1, while all three colours belong to the sparsity pattern of factors for fill–in level 2. However, the pattern shown in Fig. 3.22 is not the expected sparsity pattern of the ILU factorisation.

The result of an incomplete LU factorisation depends on the ordering of equations. If the matrix is banded, factors **L** and **U** will also be banded matrices, where the positions inside the outermost band will be filled in [27]. This implies that the structure of the matrix should be optimised using mesh renumbering techniques, in order to keep the bands as close to the diagonal as possible.

The incomplete Crout LU factorisation is done according to the predetermined sparsity pattern. Two temporary working variables are added in the algorithm: **z** corresponds to the active row **k** of the upper triangle, **w** corresponds to the active column **k** of the lower triangle, Algorithm 3.5 [62].

Algorithm 3.5. *Crout ILU factorisation*

1: ***for*** $k = 1, \ldots, n$ ***do***
2: $z_{1:k-1} := 0$, $z_{k:n} := a_{k,k:n}$
3: ***for*** $i = 1, \ldots, k-1$ *and if* $l_{ki} \neq 0$ ***do***
4: $z_{k:n} := z_{k:n} - l_{ki} \cdot u_{i,k:n}$
5: ***end for***
6: $w_{1:k} := 0$, $w_{k+1:n} := a_{k+1:n,k}$
7: ***for*** $i = 1, \ldots, k-1$ *and if* $u_{ik} \neq 0$ ***do***
8: $w_{k+1:n} := w_{k+1:n} - u_{ik} \cdot l_{k+1:n,i}$
9: ***end for***
10: $u_{k,:} = z$
11: $l_{:,k} = \frac{w}{u_{kk}}$, $l_{kk} = 1$
12: ***end for***

When the factorisation is completed, the preconditioning step consists of solving two systems by forward and back substitution, respectively:

$$\mathbf{Ly} = \mathbf{b}, \tag{3.129}$$

$$\mathbf{Ux} = \mathbf{y}. \tag{3.130}$$

It was proved by Wittum [63] that ILU has smoothing properties for symmetric matrices with elements of different magnitudes. Thus, in addition to using ILU as a preconditioner for conjugate gradients, it can also be used as a smoother in a multigrid cycle.

Parallelisation of ILU preconditioners is a challenging task. A matrix corresponding to a decomposed computational domain is shown in Eqn. (3.131). The red elements correspond to the local matrix on processor *P0*, while blue elements represent the local matrix on processor *P1*. Black elements in the upper right triangle belong to processor boundary *P0* and black elements in the lower left tringle belong to boundary *P1*. The factorisation can be performed independently on *P0* (and for the interior equations on *P1* also), since no elements from other processors need to be included in the elimination process. However, equations from *P0* need to be used for the elimination of elements on the processor boundary of *P1*. Thus, factored equations from *P0* have to be sent to *P1* after the elimination on *P0* has been completed.

$$
\begin{array}{c|cccccccccccccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 7 & 8 & 9 & 13 & 6 & 10 & 11 & 12 & 14 & 15 & 16 & 17 & 18 & 19 \\
0 & a_{0,0} & a_{0,1} & a_{0,2} & & & & & & & & & & & & & & & & & \\
1 & a_{1,0} & a_{1,1} & & a_{1,3} & a_{1,4} & & & & & & & & & & & & & & & \\
2 & a_{2,0} & & a_{2,2} & & a_{2,4} & a_{2,5} & & & & & & & & & & & & & & \\
3 & & a_{3,1} & & a_{3,3} & & & a_{3,7} & & & & a_{3,6} & & & & & & & & & \\
4 & & a_{4,1} & a_{4,2} & & a_{4,4} & & a_{4,7} & a_{4,8} & & & & & & & & & & & & \\
5 & & & a_{5,2} & & & a_{5,5} & & a_{5,8} & a_{5,9} & & & & & & & & & & & \\
7 & & & & a_{7,3} & a_{7,4} & & a_{7,7} & & & & & & a_{7,11} & a_{7,12} & & & & & & \\
8 & & & & & a_{8,4} & a_{8,5} & & a_{8,8} & & a_{8,13} & & & & a_{8,12} & & & & & & \\
9 & & & & & & a_{9,5} & & & a_{9,9} & a_{9,13} & & & & & & & & & & \\
13 & & & & & & & & a_{13,8} & a_{13,9} & a_{13,13} & & & & & & & a_{13,16} & & & \\
6 & & & & a_{6,3} & & & & & & & a_{6,6} & a_{6,10} & a_{6,11} & & & & & & & \\
10 & & & & & & & & & & & a_{10,6} & a_{10,10} & & & a_{10,14} & & & & & \\
11 & & & & & & & a_{11,7} & & & & a_{11,6} & & a_{11,11} & & a_{11,14} & a_{11,15} & & & & \\
12 & & & & & & & a_{12,7} & a_{12,8} & & & & & & a_{12,12} & & a_{12,15} & a_{12,16} & & & \\
14 & & & & & & & & & & & & a_{14,10} & a_{14,11} & & a_{14,14} & & & a_{14,17} & & \\
15 & & & & & & & & & & & & & a_{15,11} & a_{15,12} & & a_{15,15} & & a_{15,17} & a_{15,18} & \\
16 & & & & & & & & & & a_{16,13} & & & & a_{16,12} & & & a_{16,16} & & a_{16,18} & \\
17 & & & & & & & & & & & & & & & a_{17,14} & a_{17,15} & & a_{17,17} & & a_{17,19} \\
18 & & & & & & & & & & & & & & & & a_{18,15} & a_{18,16} & & a_{18,18} & a_{18,19} \\
19 & & & & & & & & & & & & & & & & & & a_{19,17} & a_{19,18} & a_{19,19}
\end{array}
\tag{3.131}
$$

After receiving the boundary equations from *P0*, elimination of (black) boundary elements on *P1* can be performed. Then factorisation for the boundary equations in blue can be completed. The procedure is applicable to any number of processors as long as there exists a global ordering in which the factorisation will be performed [2]. Also, the preconditioning sweeps can be parallelised by gathering the updated values of variables from neighbouring processors based on the global ordering. The local ordering of equations on each processor is even more important. The standard approach is to number the interior equations first, followed by the boundary equations which is natural in terms of the presented parallelisation approach. However, there are other equation ordering strategies whose overview is given by H. van der Vorst [61]. Different equation orderings produce different, but equally valid preconditioners.

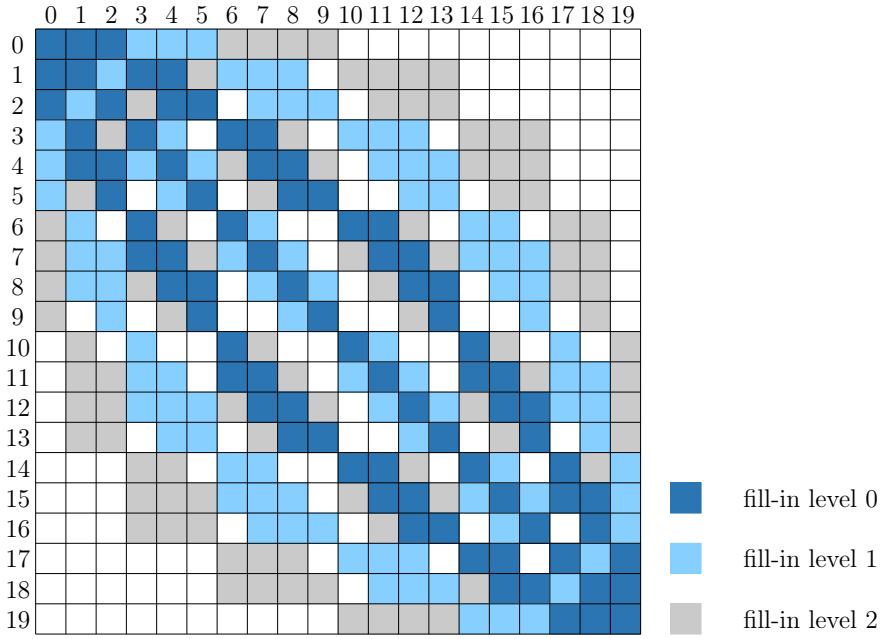


Figure 3.22: Sparsity pattern of a matrix, corresponding to mesh shown in Fig. 3.8, with extended addressing depending on the level of fill–in.

3.3.4. Krylov Subspace Methods for Nonsymmetric Matrices

Krylov subspace methods all explore the subspace spanned by a vector and the same vector multiplied by increasing powers of matrix $\mathbf{A}$. Each metod is connected to an optimality condition which guarantees finding the correct solution of the linear system. For example, conjugate gradient method constructs the solution $\mathbf{x}^{(k)}$ from the available components, so that the corresponding residual $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$ is orthogonal to the current Krylov subspace $\mathcal{D}^{(k)} = \text{span}\{\mathbf{r}^{(0)}, \mathbf{A}\mathbf{r}^{(0)}, \mathbf{A}^2\mathbf{r}^{(0)}, \ldots, \mathbf{A}^{(k-1)}\mathbf{r}^{(0)}\}$. According to the optimality condition, CG belongs to a family of *Ritz–Galerkin* methods [61]. Another large group of algorithms takes the *minimum norm residual approach* in which the solution is constructed so that the Euclidean norm of the residual $\|\mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}\|_2$ is minimal over $\mathcal{D}^{(k)}$, e.g. GMRES (Generalised Minimal Residual Method) [64].

For nonsymmetric matrices, the solution $\mathbf{x}^{(k)}$ is found so that the residual $\mathbf{r}^{(k)}$ is orthogonal to some other k–dimensional space, e.g. a space constructed using

the transpose of $\mathbf{A}$:

$$\mathcal{D}^{(k)} = \text{span}\{\mathbf{r}^{(0)}, \mathbf{A}^T\mathbf{r}^{(0)}, (\mathbf{A}^T)^2\mathbf{r}^{(0)}, \ldots, (\mathbf{A}^T)^{k-1}\mathbf{r}^{(0)}\}. \tag{3.132}$$

This approach is called *Petrov–Galerkin* [61] and BiCG (BiConjugate Gradient method) [65] is a method which employs it. The derivation of these methods is closely related to CG and it is beneficial to shift the focus from minimisation of the quadratic function to another viewpoint [66].

GMRES – Generalised Minimal Residual Method

Gram–Schmidt – procedure for $\mathbf{A}$–orthogonalising a sequence of vectors was presented in Section 3.3.2., as a part of the conjugate directions method. Another option for orthogonalisation is the *Arnoldi's method*, Algorithm 3.6 [67]. The method is equivalent to modified Gram–Schmidt procedure, as it remedies the problem of round–off errors which cause the loss of orthogonality of the search directions.

Algorithm 3.6. *Arnoldi's iteration*

1: *An initial vector* $\mathbf{q}^{(0)}$ *is given and* $\|\mathbf{q}^{(0)}\| = 1$.
2: ***for*** $j = 0, \ldots, n$ ***do***
3: $\quad \tilde{\mathbf{q}}^{(j+1)} = \mathbf{A}\mathbf{q}^{(j)}$
4: $\quad$ ***for*** $i = 0, \ldots, j$ ***do***
5: $\quad\quad h_{i,j} := \tilde{\mathbf{q}}^{(j+1)} \cdot \mathbf{q}^{(i)}$ ▷ *Projection of* $\tilde{\mathbf{q}}^{(j+1)}$ *onto* $\mathbf{q}^{(i)}$.
6: $\quad\quad \tilde{\mathbf{q}}^{(j+1)} := \tilde{\mathbf{q}}^{(j+1)} - h_{i,j} \cdot \mathbf{q}^{(i)}$ ▷ *Subtracting the parallel component.*
7: $\quad$ ***end for***
8: $\quad h_{j+1,j} = \|\tilde{\mathbf{q}}^{(j+1)}\|$
9: $\quad \mathbf{q}^{(j+1)} = \frac{\tilde{\mathbf{q}}^{(j+1)}}{h_{j+1,j}}$
10: ***end for***

A construction of an orthonormalised Krylov subspace by Arnoldi can be interpreted as a projection of matrix $\mathbf{A}$ onto the Krylov subspace. A similarity transformation of matrix $\mathbf{A}$ is defined as:

$$\mathbf{Q}^T\mathbf{A}\mathbf{Q} = \mathbf{H}. \tag{3.133}$$

where $\mathbf{Q}$ is the basis matrix. Matrices $\mathbf{A}$, $\mathbf{Q}$ and $\mathbf{H}$ are square with dimensions $n \times n$. Similarity transformation in Eqn. (3.133) can be reorganised and written

as a factorisation:

$$\mathbf{AQ} = \mathbf{QH}, \tag{3.134}$$

Arnoldi's iteration produces a part of the system Eqn. (3.134), i.e. the dimensions of the basis matrix $\mathbf{Q}$ and the r.h.s. matrix $\mathbf{H}$ are not $n \times n$, which is indicated by the overline:

$$\mathbf{A}_{n\times n}\mathbf{Q}_{n\times m} = \mathbf{Q}_{n\times(m+1)}\overline{\mathbf{H}}_{(m+1)\times m}, \tag{3.135}$$

$$\underbrace{\begin{bmatrix} & & \\ \mathbf{Aq}^{(0)} & \dots & \mathbf{Aq}^{(m-1)} \\ & & \end{bmatrix}}_{\mathbf{AQ}} = \underbrace{\begin{bmatrix} & & \\ \mathbf{q}^{(0)} & \dots & \mathbf{q}^{(m)} \\ & & \end{bmatrix}}_{\mathbf{Q}} \underbrace{\begin{bmatrix} h_{0,0} & h_{0,1} & \dots & h_{0,m-1} \\ h_{1,0} & h_{2,2} & \ddots & \vdots \\ 0 & \ddots & \ddots & \vdots \\ 0 & 0 & \ddots & h_{m-1,m-1} \\ 0 & 0 & 0 & h_{m,m-1} \end{bmatrix}}_{\overline{\mathbf{H}}},$$

where $m < n$. The columns of $\mathbf{Q}$ are orthonormal vectors $\mathbf{q}^{(j)}$, thus $\mathbf{Q}$ is unitary ($\mathbf{Q}^T = \mathbf{Q}^{-1}$, $\mathbf{QQ}^T = \mathbf{Q}^T\mathbf{Q} = \mathbf{I}$). $\overline{\mathbf{H}}$ is an upper Hessenberg matrix, i.e. matrix $\mathbf{A}$ projected onto the Krylov subspace by $\mathbf{Q}$.

If $\mathbf{A}$ is symmetric, Hessenberg matrix $\mathbf{H}$ is symmetric tridiagonal, i.e. it has two diagonals just above and below the main diagonal. CG actually solves a tridiagonal system $\mathbf{Hy} = \mathbf{f}$ [66], i.e. the original linear system projected by $\mathbf{Q}$ onto a Krylov subspace:

$$\mathbf{Ax} = \mathbf{b} \tag{3.136}$$

$$\underbrace{\mathbf{Q}^T\mathbf{AQ}}_{\mathbf{H}}\underbrace{\mathbf{Q}^T\mathbf{x}}_{\mathbf{y}} = \underbrace{\mathbf{Q}^T\mathbf{b}}_{\mathbf{f}}, \tag{3.137}$$

where $\mathbf{f} = [\|\mathbf{b}\|, 0, \dots, 0]^T$, since $\mathbf{q}^{(0)} = \frac{\mathbf{b}}{\|\mathbf{b}\|}$ and every next $\mathbf{q}^{(i)}$ is orthogonal to $\mathbf{q}^{(0)}$. The elements of the Hessenberg matrix are not explicitly calculated in CG, but they do appear in the calculation of the length of the step α and the projection of the current residual onto the previous one. An unsymmetric system cannot be solved by CG, as the $\mathbf{A}$–orthogonality of directions while calculating α is lost. However, the Hessenberg matrix approach can be used if the optimality criterion is altered. In GMRES, the goal is to minimise the norm of the residual:

$$\begin{aligned} \|\mathbf{r}^{(j)}\| &= \|\mathbf{b} - \mathbf{Ax}^{(j)}\| = \|\mathbf{b} - \mathbf{A}(\mathbf{x}^{(0)} + \mathbf{Q}^{(j)}\mathbf{y})\| \\ &= \|\underbrace{\mathbf{b} - \mathbf{Ax}^{(0)}} - \underbrace{\mathbf{AQ}^{(j)}}\mathbf{y}\| = \|\mathbf{r}^{(0)} - \mathbf{Q}^{(j+1)}\overline{\mathbf{H}}_{j+1,j}\mathbf{y}\|, \end{aligned} \tag{3.138}$$

where $\mathbf{x}^{(j)} = \mathbf{x}^{(0)} + \mathbf{Q}^{(j)}\mathbf{y}$ is the solution update in iteration j of GMRES. 2–norm is unitarily invariant, i.e. it does not change when multiplied by a unitary matrix $(\mathbf{Q}^{(j+1)})^T$:

$$\begin{aligned}\|\mathbf{r}^{(j)}\| &= \|\mathbf{r}^{(0)} - \mathbf{Q}^{(j+1)}\overline{\mathbf{H}}_{j+1,j}\mathbf{y}\| = \|(\mathbf{Q}^{(j+1)})^T\mathbf{r}^{(0)} - \underbrace{(\mathbf{Q}^{(j+1)})^T\mathbf{Q}^{(j+1)}}_{\mathbf{I}}\overline{\mathbf{H}}_{j+1,j}\mathbf{y}\| \\ &= \|\underbrace{(\mathbf{Q}^{(j+1)})^T\mathbf{r}^{(0)}}_{\mathbf{f}} - \overline{\mathbf{H}}_{j+1,j}\mathbf{y}\|, \end{aligned} \tag{3.139}$$

where $\mathbf{f} = (\mathbf{Q}^{(j+1)})^T\mathbf{r}^{(0)} = [\|\mathbf{b}\|, 0, \ldots, 0]^T$ when $\mathbf{x}^{(0)} = 0$. The minimisation of Eqn. (3.139) is an ordinary least squares problem where $\mathbf{y}$ is the minimiser, and it is found by factorising the Hessenberg matrix $\overline{\mathbf{H}}_{j+1,j}$ using Givens plane rotations [2]. Since $\overline{\mathbf{H}}_{j+1,j}$ has one diagonal below the main diagonal, only those elements need to be eliminated to achieve an upper triangular form and calculate $\mathbf{y}$ by back substitution. In a single iteration of GMRES, Givens rotation matrices have the dimension $(m+1)\times(m+1)$, where m is the number of columns of the Hessenberg matrix:

$$\mathbf{G}_{\mathrm{R}}^{(i)} = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & c^{(i)} & s^{(i)} & & \\ & & -s^{(i)} & c^{(i)} & & \\ & & & & \ddots & \\ & & & & & 1 \end{bmatrix}, \tag{3.140}$$

where $c^{(i)} = \cos\theta^{(i)}$ and $s^{(i)} = \sin\theta^{(i)}$ and $\theta^{(i)}$ is the angle of rotation. To eliminate the diagonal in the lower triangle, Hessenberg matrix and r.h.s. vector are multiplied from the left by a sequence of Givens matrices. An example given in [2] illustrates two steps in the elimination of the lower triangular elements of a 5×5 Hessenberg matrix:

$$\overline{\mathbf{H}}_{6,5} = \begin{bmatrix} h_{0,0} & h_{0,1} & h_{0,2} & h_{0,3} & h_{0,4} \\ h_{1,0} & h_{1,1} & h_{1,2} & h_{1,3} & h_{1,4} \\ & h_{2,1} & h_{2,2} & h_{2,3} & h_{2,4} \\ & & h_{3,2} & h_{3,3} & h_{3,4} \\ & & & h_{4,3} & h_{4,4} \\ & & & & h_{5,4} \end{bmatrix}, \quad \mathbf{f}^{(0)} = (\mathbf{Q}^{(6)})^T\mathbf{r}^{(0)} = \begin{bmatrix} \|\mathbf{b}\| \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

The following two multiplications will eliminate elements $h_{1,0}$ and $h_{2,1}$ from the lower triangle, and modify the r.h.s. vector:

$$\mathbf{G}_{\mathrm{R}}^{(2)}\mathbf{G}_{\mathrm{R}}^{(1)}\overline{\mathbf{H}}_{6,5},$$
$$\mathbf{G}_{\mathrm{R}}^{(2)}\mathbf{G}_{\mathrm{R}}^{(1)}\mathbf{f}^{(6)}.$$

Here, Givens plane rotation matrices are defined as:

$$\mathbf{G}_{\mathrm{R}}^{(1)} = \begin{bmatrix} c^{(1)} & s^{(1)} & & & & & \\ -s^{(1)} & c^{(1)} & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & & & & 1 & & \\ & & & & & 1 & \\ & & & & & & 1 \end{bmatrix}, \quad \mathbf{G}_{\mathrm{R}}^{(2)} = \begin{bmatrix} 1 & & & & & & \\ & c^{(2)} & s^{(2)} & & & & \\ & -s^{(2)} & c^{(2)} & & & & \\ & & & 1 & & & \\ & & & & 1 & & \\ & & & & & 1 & \\ & & & & & & 1 \end{bmatrix},$$

and the scalar elements s and c are calculated using the elements from the Hessenberg matrix:

$$s^{(1)} = \frac{h_{1,0}}{\sqrt{(h_{0,0}^{(0)})^2 + h_{1,0}^2}}, \; c^{(1)} = \frac{h_{0,0}^{(0)})}{\sqrt{(h_{0,0}^{(0)})^2 + h_{1,0}^2}},$$
$$s^{(2)} = \frac{h_{2,1}}{\sqrt{(h_{1,1}^{(1)})^2 + h_{2,1}^2}}, \; c^{(2)} = \frac{h_{1,1}^{(1)}}{\sqrt{(h_{1,1}^{(1)})^2 + h_{2,1}^2}}.$$

Upper triangular Hessenberg matrix and the modified r.h.s. vector are obtained after applying the final Givens matrix $\mathbf{G}_{\mathrm{R}}^{(5)}$:

$$\mathbf{G}_{\mathrm{R}}^{(5)}\mathbf{G}_{\mathrm{R}}^{(4)}\mathbf{G}_{\mathrm{R}}^{(3)}\mathbf{G}_{\mathrm{R}}^{(2)}\mathbf{G}_{\mathrm{R}}^{(1)}\overline{\mathbf{H}}^{(0)} = \begin{bmatrix} h_{0,0}^{(5)} & h_{0,1}^{(5)} & h_{0,2}^{(5)} & h_{0,3}^{(5)} & h_{0,4}^{(5)} \\ & h_{1,1}^{(5)} & h_{1,2}^{(5)} & h_{1,3}^{(5)} & h_{1,4}^{(5)} \\ & & h_{2,2}^{(5)} & h_{2,3}^{(5)} & h_{2,4}^{(5)} \\ & & & h_{3,3}^{(5)} & h_{3,4}^{(5)} \\ & & & & h_{4,4}^{(5)} \\ & & & & 0 \end{bmatrix}, \quad \mathbf{f}^{(5)} = (\mathbf{Q}^{(6)})^T\mathbf{r}^{(0)} = \begin{bmatrix} \gamma_0 \\ \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \\ \gamma_5 \end{bmatrix}.$$

The product of matrices $\mathbf{G}_{\mathrm{R}}^{(i)}$ is a unitary matrix:

$$\mathbf{V}_m = \mathbf{G}_{\mathrm{R}}^{(m)}\mathbf{G}_{\mathrm{R}}^{(m-1)}\ldots\mathbf{G}_{\mathrm{R}}^{(1)}. \tag{3.141}$$

Thus, the result after Givens rotations can be written as:

$$\mathbf{V}_m \overline{\mathbf{H}}^{(0)}_{m+1,m} = \overline{\mathbf{H}}^{(m)}_{m+1,m} = \overline{\mathbf{R}}^{(m)} \tag{3.142}$$

$$\mathbf{V}_m \mathbf{f} = \overline{\mathbf{g}}^{(m)} = [\gamma_0, \ldots, \gamma_m]^T. \tag{3.143}$$

2–norm is unitarily invariant, and the minimisation problem is reduced to:

$$\min \|\mathbf{f} - \overline{\mathbf{H}}_{j+1,j}\mathbf{y}\| = \min \|\overline{\mathbf{g}}^{(m)} - \overline{\mathbf{R}}^{(m)}\mathbf{y}\|. \tag{3.144}$$

The last row of $\overline{\mathbf{R}}^{(m)}$ contains only zero elements, i.e. the last component of $\overline{\mathbf{g}}^{(m)}$ is not relevant for the solution of the system. To prove that the solution of the upper triangular system produced by Givens rotations minimises the residual [2], use the fact that 2–norm is unitarily invariant and for any vector $\mathbf{y}$:

$$\begin{aligned} \|\mathbf{f} - \overline{\mathbf{H}}_{j+1,j}\mathbf{y}\|^2 &= \|\mathbf{V}_m(\mathbf{f} - \overline{\mathbf{H}}_{j+1,j}\mathbf{y})\|^2 \\ &= \|\overline{g}^{(m)} - \overline{\mathbf{R}}^{(m)}\mathbf{y}\|^2 \\ &= |\gamma_m|^2 + \|\mathbf{g}^{(m)} - \mathbf{R}^{(m)}\mathbf{y}\|^2, \end{aligned} \tag{3.145}$$

where γ_m is the last component of the r.h.s. vector $\overline{\mathbf{g}}^{(m)}$, $\mathbf{g}^{(m)}$ is equal to $\overline{\mathbf{g}}^{(m)}$ without the last component, $\mathbf{R}^{(m)}$ is equal to $\overline{\mathbf{R}}^{(m)}$ where the last zero row is deleted. The minimum of the l.h.s. is reached when $\|\mathbf{g}^{(m)} - \mathbf{R}^{(m)}y\|^2$ is equal to zero, i.e. solving the upper triangular system ($\mathbf{y} = \mathbf{R}^{(m)})^{-1}\mathbf{g}^{(m)}$) gives the solution of the minimisation problem.

Also, looking at Eqn. (3.139), if y nullifies all components of $\overline{\mathbf{g}}^{(m)}$ except the last one, the last component of $\overline{\mathbf{g}}^{(m)}$ corresponds to the norm of the residual produced by the solution $\mathbf{x}^{(m)} = \mathbf{x}^{(0)} + \mathbf{Q}^{(m)}\mathbf{y}$ [2]:

$$\begin{aligned} \|\mathbf{r}^{(m)}\|^2 &= \|\mathbf{f} - \overline{\mathbf{H}}_{m+1,m}\mathbf{y}\|^2 \\ &= |\gamma_m|^2 + \underbrace{\|\mathbf{g}^{(m)} - \mathbf{R}^{(m)}\mathbf{y}\|^2}_{0}. \end{aligned} \tag{3.146}$$

Thus, it is not necessary to explicitly calculate the solution after each iteration, but only when the residual $\mathbf{r}^{(m)} = |\gamma_m|$ is declared small enough.

The number of orthogonalised vectors in the Krylov subspace m, used for the construction of the approximate solution $\mathbf{x}^{(m)}$ is smaller than the dimension of the coefficient matrix n. When $m = n$, GMRES should converge to the exact

solution in maximally n steps (assuming no round–off errors). However, when n is very large, storage and computational requirements of GMRES grow linearly with each iteration [68]. To avoid high computational costs, *restarted versions* of GMRES are used, where after a chosen number of m iterations, the accumulated data is discarded and the intermediate data is used as the input in the next iteration. The restarted approach could cause the solution to stall as the old search directions are lost. The choice of optimal m is not trivial.

BiCGStab – Biconjugate Gradient Stabilised

It was shown in Section 3.3.2. that CG method converges for symmetric matrices, i.e. the solution of the symmetric system $\mathbf{Ax} = \mathbf{b}$ is a minimum of the quadratic function $f(\mathbf{x})$, Eqn. (3.89). However, if the coefficient matrix $\mathbf{A}$ is not symmetric, it is still possible to employ the CG algorithm by applying it to a *surrogate symmetric system*:

$$\begin{bmatrix} 0 & \mathbf{A} \\ \mathbf{A}^T & 0 \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{x}} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}, \tag{3.147}$$

where $\tilde{\mathbf{x}}$ is a solution of the system $\mathbf{A}^T\tilde{\mathbf{x}} = 0$. When applied to this system, CG method will produce two sequences of search directions and residuals, one corresponding to $\mathbf{A}$ and the other to $\mathbf{A}^T$. That is, two approximation spaces are created:

$$\mathcal{D}^{(k)} = \text{span}\{\mathbf{d}^{(0)}, \mathbf{A}\mathbf{d}^{(0)}, \mathbf{A}^2\mathbf{d}^{(0)}, \ldots, \mathbf{A}^{k-1}\mathbf{d}^{(0)}\}, \tag{3.148}$$

$$\mathcal{D}^{*(k)} = \text{span}\{\mathbf{d}^{*(0)}, \mathbf{A}^T\mathbf{d}^{*(0)}, (\mathbf{A}^T)^2\mathbf{d}^{*(0)}, \ldots, (\mathbf{A}^T)^{k-1}\mathbf{d}^{*(0)}\}, \tag{3.149}$$

where $\mathcal{D}^{(k)}$ is the Krylov subspace corresponding to matrix $\mathbf{A}$ in the k^{th} iteration, while $\mathcal{W}^{(k)}$ is the subspace obtained by repetitively multiplying vector $\mathbf{d}^*$ with the transpose of $\mathbf{A}$. Note that the superscript $(^*)$ does not denote the conjugate transpose, rather it symbolises the additional vectors and subspace, corresponding to $\mathbf{A}^T$. Since there are two subspaces created, the algorithm is called the *biconjugate gradient method (BiCG)* [65]. The aim of BiCG is to orthogonally project $\mathcal{D}^{(k)}$ onto $\mathcal{D}^{*(k)}$:

$$(\mathbf{r}^{*(i)})^T\mathbf{r}^{(j)} = 0 \text{ where } i \neq j, \tag{3.150}$$

$$(\mathbf{d}^{*(i)})^T\mathbf{A}\mathbf{d}^{(j)} = 0 \text{ where } i \neq j. \tag{3.151}$$

It uses the *Lanczos method* [2] for orthogonalisation. The algorithm resembles the CG algorithm with additional operations using the transpose of $\mathbf{A}$.

1. At the beginning assume $\mathbf{x}^{(0)} = 0$ and $\mathbf{d}^{(0)} = \mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$.
   Choose $\mathbf{r}^{*(0)} = \mathbf{d}^{*(0)}$ such that $(\mathbf{r}^{(0)})^T \mathbf{r}^{*(0)} \neq 0$.

2. Calculate the length of the step in direction of the search vectors:

$$\alpha^{(k)} = \frac{(\mathbf{r}^{*(k)})^T \mathbf{r}^{(k)}}{(\mathbf{d}^{*(k)})^T \mathbf{A}\mathbf{d}^{(k)}}. \tag{3.152}$$

3. Calculate the new solution:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{d}^{(k)}. \tag{3.153}$$

4. Update the residuals:

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} + \alpha^{(k)} \mathbf{A}\mathbf{d}^{(k)} \tag{3.154}$$

$$\mathbf{r}^{*(k+1)} = \mathbf{r}^{*(k)} + \alpha^{(k)} \mathbf{A}^T \mathbf{d}^{*(k)}. \tag{3.155}$$

5. Calculate the projection operator:

$$\beta^{(k)} = \frac{(\mathbf{r}^{(k+1)})^T \mathbf{r}^{*(k+1)}}{(\mathbf{r}^{(k)})^T \mathbf{r}^{*(k)}}. \tag{3.156}$$

6. Update the search directions:

$$\mathbf{d}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta^{(k)} \mathbf{d}^{(k)} \tag{3.157}$$

$$\mathbf{d}^{*(k+1)} = \mathbf{r}^{*(k+1)} + \beta^{(k)} \mathbf{d}^{*(k)}. \tag{3.158}$$

In BiCG, $\mathbf{A}^T$ must be explicitly multiplied with the previous search direction to update the residual, which is a drawback of BiCG compared to CG (parallelisation issues [67]). Also, oscillatory convergence was noticed in practice [68]. Contrary to GMRES, BiCG does not have any minimisation effects, and it is not necessary to store all previous search directions. However, there are additional vector–matrix products.

Multiplication with $\mathbf{A}^T$ in BiCG was remedied in the *conjugate gradient squared (CGS)* method (by Sonneveld [69]) which uses the fact that residuals and search directions are polynomial combinations of $\mathbf{A}$ and $\mathbf{r}^{(0)}$ [70]:

$$\mathbf{r}^{(k)} = (\phi_{\text{BCGS}})_k(\mathbf{A})\mathbf{r}^{(0)}, \tag{3.159}$$

$$\mathbf{d}^{(k)} = \pi_k(\mathbf{A})\mathbf{r}^{(0)}, \tag{3.160}$$

$$\mathbf{r}^{*(k)} = \phi_k(\mathbf{A}^T)\mathbf{r}^{*(0)}, \tag{3.161}$$

$$\mathbf{d}^{*(k)} = \pi_k(\mathbf{A}^T)\mathbf{r}^{*(0)}, \tag{3.162}$$

where $(\phi_{\text{BCGS}})_k$ and π_k are polynomials with degree k ($(\phi_{\text{BCGS}})_0(\mathbf{A}) = 1$, $\pi_0(\mathbf{A}) = 0$). These polynomials are presumed to act as contraction functions, since the residual vector shrinks in each subsequent iteration. Inserting the polynomial expressions into BiCG algorithm and manipulating the terms [70], produces an improvement in terms of eliminating the multiplication with $\mathbf{A}^T$. The "squared" in the name of the method comes from squaring the polynomials, i.e. applying the contraction function twice in a single iteration. However, the oscillatory behaviour of BiCG still remains.

Van der Vorst introduced a *stabilised version of BiCG (BiCGStab)* [71] by applying a smoothing function $\psi_k(\mathbf{A})$, which reduces the convergence oscillations of BiCG:

$$\mathbf{r}^{(k)} = \psi_k(\mathbf{A})\,(\phi_{\text{BCGS}})_k(\mathbf{A})\mathbf{r}^{(0)}, \tag{3.163}$$

$$\mathbf{d}^{(k)} = \psi_k(\mathbf{A})\,\pi_k(\mathbf{A})\mathbf{r}^{(0)}, \tag{3.164}$$

where

$$\psi_{k+1}(\mathbf{A}) = (\mathbf{I} - \omega^{(k)}_{\text{BCGS}}\mathbf{A})\,\psi_k(\mathbf{A}). \tag{3.165}$$

The new residual and search direction are defined as:

$$\mathbf{r}^{(k+1)} = \psi_k(\mathbf{A})\,(\mathbf{r}^{(k)} - \alpha^{(k)}\mathbf{A}\mathbf{d}^{(k)}) = (\mathbf{I} - \omega^{(k)}_{\text{BCGS}}\mathbf{A})(\mathbf{r}^{(k)} - \alpha^{(k)}\mathbf{A}\mathbf{d}^{(k)}), \tag{3.166}$$

$$\mathbf{d}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta^{(k)}(\mathbf{I} - \omega^{(k)}_{\text{BCGS}}\mathbf{A})\mathbf{d}^{(k)}. \tag{3.167}$$

BiCGStab is a very popular Krylov subspace method for solving large sparse systems. The efficiency and stability are achieved by applying a smoothing function ψ_k which, by ensuring $\mathbf{A}$–orthogonality between $\mathbf{r}^{(k+1)}$ and $\mathbf{r}^{(k)}$, locally reduces the 2–norm of the residual vector to improve convergence. The algorithm of BiCGStab consists of the following steps:

1. Assume $\mathbf{x}^{(0)} = 0$, calculate $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$, arbitrarily choose $\mathbf{r}^{*(0)}$. Set $\mathbf{d}^{(0)} = \mathbf{r}^{(0)}$.

2. Calculate the length of the step in direction of the search vectors:

$$\alpha^{(k)} = \frac{(\mathbf{r}^{*(0)})^T \mathbf{r}^{(k)}}{(\mathbf{r}^{*(k)})^T \mathbf{A}\mathbf{d}^{(k)}}. \tag{3.168}$$

3. Update the residual before orthogonalisation using an auxilliary variable $\mathbf{s}$:

$$\mathbf{s}^{(k)} = \mathbf{r}^{(k)} - \alpha^{(k)} \mathbf{A}\mathbf{d}^{(k)}. \tag{3.169}$$

   Check the stopping criterion. If it is satisfied, terminate the algorithm.

4. Calculate the smoothing coefficient:

$$\omega_{\mathrm{BCGS}}^{(k)} = \frac{(\mathbf{s}^{(k)})^T \mathbf{A}\mathbf{s}^{(k)}}{(\mathbf{A}\mathbf{s}^{(k)})^T \mathbf{A}\mathbf{s}^{(k)}}. \tag{3.170}$$

5. Calculate the new solution:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{d}^{(k)} + \omega_{\mathrm{BCGS}}^{(k)} \mathbf{s}^{(k)}. \tag{3.171}$$

6. Calculate the new residual:

$$\mathbf{r}^{(k+1)} = \mathbf{s}^{(k)} - \omega_{\mathrm{BCGS}}^{(k)} \mathbf{A}\mathbf{s}^{(k)}. \tag{3.172}$$

7. Calculate the projection operator:

$$\beta^{(k)} = \frac{(\mathbf{r}^{*(0)})^T \mathbf{r}^{(k+1)}}{(\mathbf{r}^{*(0)})^T \mathbf{r}^{(k)}} \frac{\alpha^{(k)}}{\omega_{\mathrm{BCGS}}^{(k)}}. \tag{3.173}$$

8. Establish the next search direction:

$$\mathbf{d}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta^{(k)} (\mathbf{d}^{(k)} - \omega_{\mathrm{BCGS}}^{(k)} \mathbf{A}\mathbf{d}^{(k)}). \tag{3.174}$$

9. Return to step 2 if the convergence criterion is not satisfied.

3.4. Closure

The main topic of this chapter were the efficient state–of–the–art algorithms for the solution of the linear system. Fixed–point methods were presented as the first step towards an algebraic multigrid algorithm. Algebraic multigrid aids the convergence of fixed–point methods by constructing a hierarchy of matrices with smaller dimensions, which enables the propagation of information carried by the fixed–point method. Two methods for construction of coarse levels were outlined: additive correction (AAMG) and selection method (SAMG). Selection method has a better compatibility between interpolation and smoothing, which should prove as an advantage in terms of convergence rate. A separate class of efficient solvers based on assembling the solution from the Krylov subspace was presented. First a derivation of the steepest descent method and its upgrade, the conjugate gradient (CG) method was given. CG is an alternative to the multigrid method for symmetric positive definite matrices, but it is inefficient for unsymmetric systems. Krylov subspace methods are often paired with preconditioning techniques which improve convergence: we presented the incomplete lower–upper (ILU) factorisation as an example, which can also be used as a smoother for multigrid. Since the block–matrix of the implicit pressure–velocity system is unsymmetric, it is necessary to employ appropriate Krylov subspace methods such as the generalised minimal residual (GMRES) or biconjugate gradient stabilised (BiCGStab) which were also described in this chapter. In the following chapter, we shall investigate the performance of these linear algorithms for the implicitly–coupled pressure velocity system, as well as highlight the advantages and disadvantages of the implicit coupling.

4. Case Studies

4.1. Introduction

In Chapter 2. we have presented the concept of segregated and implicitly coupled solution techniques for the pressure–velocity system as well as the finite volume discretisation and the resulting linear system. In chapter 3. different types of algorithms for the solution of linear system were presented: multigrid (selection – SAMG, and agglomeration – AAMG) and Krylov subspace (CG, GMRES, BiCGStab) solvers. Also, some techniques for acceleration of convergence were presented: preconditioners and smoothers (Jacobi, Gauss–Seidel, ILU).

In this chapter, we will try to resolve some uncertainties which arise when thinking about the appropriate choice of solution strategy and linear solver, by showing the simulation settings and results of several complex flow cases. Some of the questions we seek to answer are:

- Is it better to use segregated (SIMPLE) or implicitly coupled pressure–velocity solver, in terms of stability, robustness and speed of convergence?
- Which linear solver should we choose for the block–matrix of the implicitly coupled system?
- What is the effect of mesh density on the convergence of SIMPLE versus implicitly coupled solver?
- Is it better to treat inter–equation coupling terms implicitly or explicitly, considering the properties of the coefficient matrix?

Since the main topic of this thesis was the implementation of the selection algebraic multigrid algorithm (SAMG) we shall explore its performance on the implicitly–coupled system:

- Which is the optimal smoother?
- Which is the optimal multigrid cycle in terms of efficiency?

- Does the weighting factor, i.e. coarsening norm in SAMG have an effect on convergence?
- What is the influence of number of cells on the coarsest level? Is the number of levels important?
- What is the optimal criterion for determining a strong connection between two equations?
- How does the algorithm scale and perform on parallel high–performance computers in domain decomposition mode?

In the following sections we shall present observations and conclusions derived from several test cases. It is important to emphasise that these observations are not case specific, i.e. to keep the chapter concise we did not repeat the same conclusions multiple times, even though they are confirmed for multiple cases. Simulations were run on a high performance computer with the following specifications: 28 CPUs (Intel Xeon Processor E5-2637 v3 15M Cache) with a total of 112 cores (3.50 GHz per core). Most cases were run using 8 or 16 cores in domain decomposition mode, using a message passing interface (MPI).

4.2. Segregated vs. Implicitly Coupled Pressure–Velocity Solver

Since the equations in the coupled solver, presented in Section 2.3.3. are solved in a single linear system and include the cross–coupling terms implicitly in the coefficient matrix, it is expected that the solver will reach the desired convergence criterion (i.e. solution) in fewer non–linear iterations, compared to the segregated (SIMPLE) algorithm, Section 2.3.1. However, the implicitly–coupled linear system is more demanding in terms of *storage capacity* (16 times larger matrix), *choice of linear solver* (different types of equations are solved simultaneously, which requires employing complex preconditioners and solvers) and overall *computational time per iteration*. Thus, the implicitly coupled pressure–velocity solver may outperform the segregated solver based on the SIMPLE algorithm regarding the number of non–linear iterations, but it could be slower when com-

paring the overall execution time of the simulation.

In this chapter we shall illustrate the advantages and disadvantages of the coupled solver compared to the traditional segregated solver based on the SIMPLE algorithm, for several test cases: flow inside a Francis turbine and a centrifugal pump, external flow around a bluff body, a Formula 1 front wing and a BB2–submarine. To keep this concise, additional data for each case such as mesh statistics, flow features, boundary conditions are listed in Appendix A.

The main criterion for comparison is the rate of decrease of the normalised residual per iteration. Since it is expected that a single iteration of the implicitly–coupled solver requires more memory and CPU time than one iteration of the segregated solver, we shall also compare convergence against required computation time.

To be able to compare convergence of different cases, the residual $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$ is normalised by dividing it with a normalisation factor ξ [H. Jasak, private communication]:

$$\xi = \sum_{i}^{n} (|\mathbf{A}\mathbf{x} - \mathbf{A}\overline{\mathbf{x}}| + |\mathbf{b} - \mathbf{A}\overline{\mathbf{x}}|), \tag{4.1}$$

where $\mathbf{x}$ is the current value of the solution vector and every component of $\overline{\mathbf{x}}$ is equal to an average value of $\mathbf{x}$ over n (all) equations:

$$\overline{\mathbf{x}} = \frac{\sum_i^n x_i}{n}. \tag{4.2}$$

Note that the terms in Eqn. (4.1) are assembled to take into account two possibilities: if the solution vector $\mathbf{x}$ is equal to zero (at the beginning of the iteration process), or if the right hand side vector $\mathbf{b}$ is equal to zero (no source terms). Eqn. (4.1) returns a scalar normalisation factor for each unknown ($\xi_{\mathbf{U}_x}$, $\xi_{\mathbf{U}_y}$, $\xi_{\mathbf{U}_z}$, ξ_p). For the implicitly coupled solver, $|\bullet|$ denotes the absolute value of each component of the obtained vector rather than the Euclidean norm.

Thus, the normalised residual $\mathbf{r}_\mathrm{n}$ for the segregated and coupled solver is calculated as:

$$\mathbf{r}_\mathrm{n} = \left[\frac{\sum_i^n |r_{U_{x_i}}|}{\xi_{\mathbf{U}_x}}, \frac{\sum_i^n |r_{U_{y_i}}|}{\xi_{\mathbf{U}_y}}, \frac{\sum_i^n |r_{U_{z_i}}|}{\xi_{\mathbf{U}_z}}, \frac{\sum_i^n |r_p|}{\xi_p} \right]^T. \tag{4.3}$$

Non–linear and linear convergence is evaluated by monitoring the reduction of the 1–norm of the residual, i.e. a sum of magnitudes of residuals for all equations corresponding to a single variable. An absolute convergence criterion is used for the non–linear solver: if the residual drops below a prescribed value after the non–linear solution update, the solver has converged. Absolute and relative convergence criterion is used for the linear solver. Absolute criterion is similar to the non–linear solution, i.e. the iteration will end as soon as the residual reaches some predetermined value. A relative tolerance is defined as a ratio of the initial and final residual of the linear iterative process. For example, if a relative tolerance 10^{-3} is prescribed for a linear solver, the solution process will end as soon as the residual drops three orders of magnitude compared to the residual at the beginning of the linear iteration.

Bluff body

The first test case is a simulation of external turbulent flow around a bluff body. Mesh statistics are given in Table A7 while a slice through the volume mesh is shown in Fig. A11. A definition of a bluff body is given by Fackrell [72]: "A bluff body can be defined as a body of any shape, which experiences complete boundary layer separation before the trailing edge, due to large adverse pressure gradient set up over that part of the body behind the position of maximum thickness. This pressure gradient decelerates the slow moving fluid within the boundary layer near the surface and eventually causes a reversed flow and hence separation". In the scope of this work, we chose a simple geometry which resembles a bullet with an upswept surface at the back, enclosed with two flat plates. When placed near the ground, the upswept surface behaves as a diffuser used on Formula 1 racing cars to increase downforce (negative lift). The half–width of the diffuser is $d = 157\,\text{mm}$, length $l = 1315\,\text{mm}$, height $h = 326\,\text{mm}$ and the diffuser angle is $17°$.

The pattern of the flow is quite complex and it depends on the distance of the diffuser section from the ground (ride height): when the diffuser is sufficiently high, two vortices are formed along the two sideplates and additional (high energy) air is pulled in from the outside [73]. This causes a pressure drop at the start of the diffuser which pulls in even more airflow from the front section. The

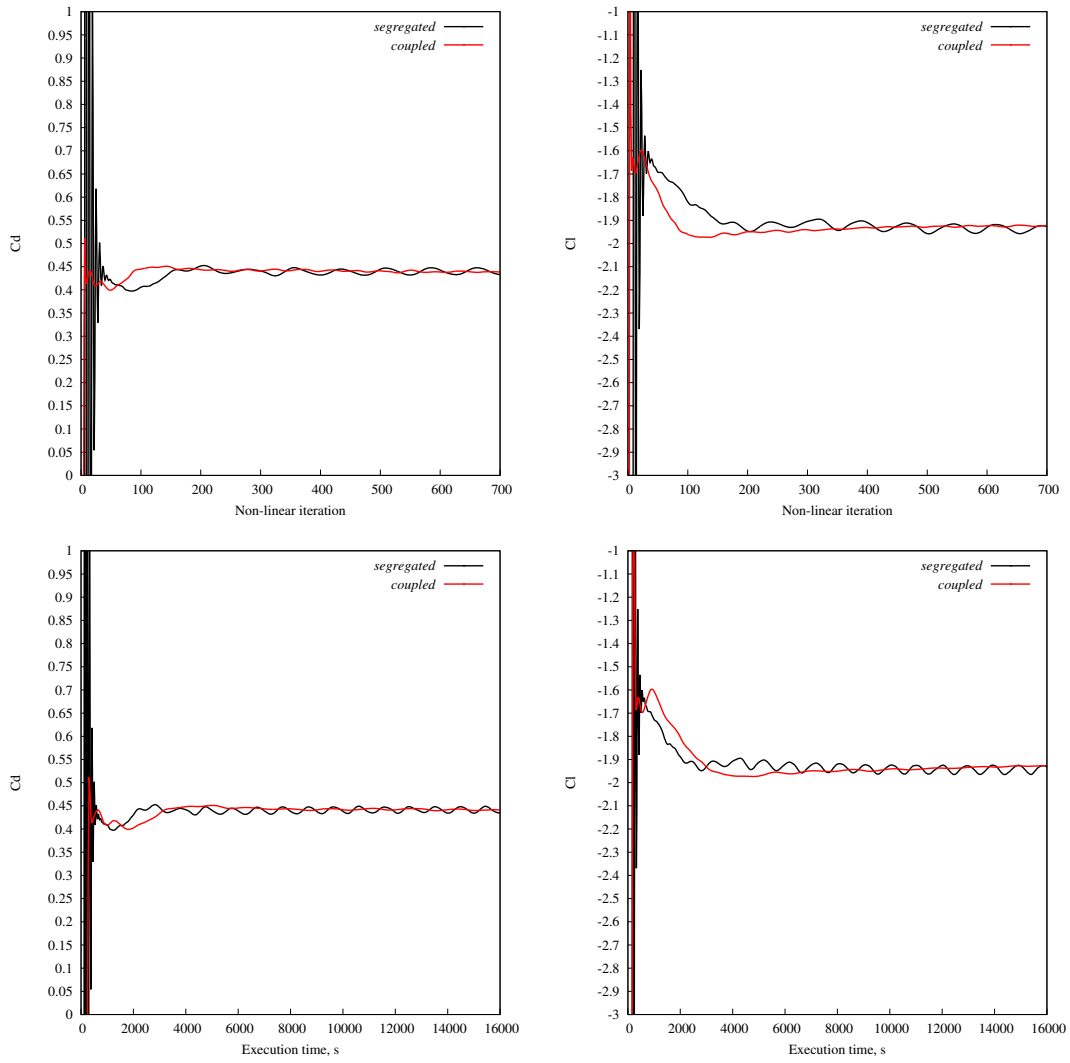


Figure 4.1: Bluff body: convergence of drag and lift coefficient against non–linear iterations and execution time for the segregated (SIMPLE) and implicitly coupled pressure–velocity solver.

flow is symmetrical and as the ride height is decreased, downforce increases while a separation bubble is formed at the back between the two vortices. At a critical (too low) ride height, one of the vortices will fall apart which will cause immediate drop of downforce. The two vortices at 40 mm ride height and 20 m/s freestream velocity are illustrated in Fig. 4.2. The pressure field acting on the surface of the body is shown in Fig. A12.

The two integral values of interest are the drag and lift coefficients, defined as:

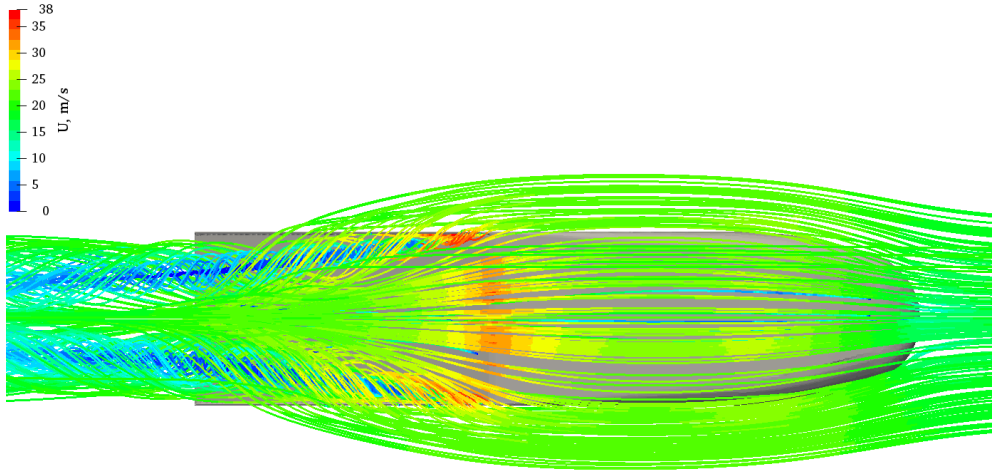


Figure 4.2: Bluff body: the pattern of the flow on the bottom surface.

$$C_{\mathrm{D}} = \frac{F_{\mathrm{D}}}{\frac{1}{2}\rho u^2 A}, \tag{4.4}$$

$$C_{\mathrm{L}} = \frac{F_{\mathrm{L}}}{\frac{1}{2}\rho u^2 A}, \tag{4.5}$$

respectively, where F_{D} is the drag force in the direction of the flow, F_{L} is the lift force, perpendicular to the direction of the flow, ρ is the fluid density, u is the freestream flow velocity and A is the representative surface area. The reported experimental values for this model and freestream velocity $u = 20\mathrm{m/s}$ [74] are $C_{\mathrm{D}} = 0.49$ and $C_{\mathrm{L}} = -1.90$.

The convergence of force coefficients is shown in Fig. 4.1 for the segregated and the implicitly coupled solver. Block–selection AMG was used for the coupled coefficient matrix, underrelaxation factor of the momentum equation was $\alpha_{\mathbf{u}} = 0.9$. The momentum equation in the segregated solver was solved using the BiCGStab solver preconditioned by ILU0, $\alpha_{\mathbf{u}} = 0.7$, while the pressure equation was solved using a scalar selection AMG solver, $\alpha_p = 0.3$. The equations of the k–ω–SST turbulence model in both cases were solved with BiCGStab preconditioned by Cholesky ILU. The setup of linear solvers for all cases mentioned in this section is listed in Table 4.1. Identical discretisation schemes were used in all cases for

the segregated and implicitly coupled solver. Note that two turbulence models were used: k-ω-SST [42] and an implicitly coupled version of the same model [75], where the equations for k and ω are solved in a single linear system, which improves the stability of convergence.

Table 4.1: Setup of linear solvers for segregated and implicitly coupled solver for all test cases.

	COUPLED	**SIMPLE**		**TURBULENCE**	
CASE	$\mathbf{u}$, p	$\mathbf{u}$	p	k	ω
Bluff body	SAMG, V–cycle, smoother ILUC0, 1 pre–sweep, 2 post–sweeps, no. coarse equations 10, relative tolerance 10^{-3}, convergence at 10^{-7} $\alpha_{\mathbf{u}} = 0.85$	BiCGStab, preconditioner ILU0, convergence at 10^{-6} $\alpha_{\mathbf{u}} = 0.7$	SAMG, V–cycle, smoother symmetric Gauss–Seidel, 1 pre–sweep, 3 post–sweeps, no. coarse equations 4, relative tolerance 10^{-4}, convergence at 10^{-6} $\alpha_p = 0.3$	coupled k–ω–SST BiCGStab, preconditioner Cholesky, convergence at 10^{-12} $(\alpha_k)_C = 0.9$ $(\alpha_\omega)_C = 0.9$ $(\alpha_k)_S = 0.7$ $(\alpha_\omega)_S = 0.7$	
Front wing	SAMG, V–cycle, smoother ILUC0, 1 pre–sweep, 2 post–sweeps, no. coarse equations 10, relative tolerance 10^{-3}, convergence at 10^{-7} $\alpha_{\mathbf{u}} = 0.85$	BiCGStab, preconditioner ILU0, convergence at 10^{-6} $\alpha_{\mathbf{u}} = 0.7$	SAMG, V–cycle, smoother symmetric Gauss–Seidel, 2 pre–sweep, 2 post–sweeps, no. coarse equations 4, relative tolerance 10^{-3}, convergence at 10^{-5} $\alpha_p = 0.3$	BiCGStab, preconditioner ILU0, convergence at 10^{-8} $(\alpha_k)_C = 0.9$ $(\alpha_k)_S = 0.7$	BiCGStab, preconditioner ILU0, convergence at 10^{-8} $(\alpha_\omega)_C = 0.9$ $(\alpha_\omega)_S = 0.7$
BB2 submarine	SAMG, W–cycle, smoother ILUC0, 1 pre–sweep, 2 post–sweeps, no. coarse equations 20, convergence at 10^{-7} $\alpha_{\mathbf{u}} = 0.95$	BiCGStab, preconditioner ILU0, convergence at 10^{-6} $\alpha_{\mathbf{u}} = 0.7$	SAMG, V–cycle, smoother symmetric Gauss–Seidel, 1 pre–sweep, 3 post–sweeps, no. coarse equations 4, relative tolerance 10^{-4}, convergence at 10^{-6} $\alpha_p = 0.3$	coupled k–ω–SST BiCGStab, preconditioner Cholesky, convergence at 10^{-8} $(\alpha_k)_C = 0.95$ $(\alpha_\omega)_C = 0.95$ $(\alpha_k)_S = 0.7$ $(\alpha_\omega)_S = 0.7$	
Francis turbine	SAMG, V–cycle, smoother ILUC0, 1 pre–sweep, 2 post–sweeps, no. coarse equations 20, relative tolerance 10^{-4}, convergence at 10^{-6} $\alpha_{\mathbf{u}} = 0.9$	BiCGStab, preconditioner ILU0, convergence at 10^{-6} $\alpha_{\mathbf{u}} = 0.7$	SAMG, V–cycle, smoother symmetric Gauss–Seidel, 1 pre–sweep, 3 post–sweeps, no. coarse equations 4, relative tolerance 10^{-4}, convergence at 10^{-7} $\alpha_p = 0.3$	coupled k–ω–SST BiCGStab, preconditioner Cholesky, convergence at 10^{-12} $(\alpha_k)_C = 0.95$ $(\alpha_\omega)_C = 0.95$ $(\alpha_k)_S = 0.7$ $(\alpha_\omega)_S = 0.7$	
Centrifugal pump	SAMG, V–cycle, smoother ILUC0, 1 pre–sweep, 2 post–sweeps, no. coarse equations 20, relative tolerance 10^{-4}, convergence at 10^{-7} $\alpha_{\mathbf{u}} = 0.95$	BiCGStab, preconditioner ILU0, relative tolerance 10^{-3}, convergence at 10^{-5} $\alpha_{\mathbf{u}} = 0.7$	SAMG, V–cycle, smoother symmetric Gauss–Seidel, 2 pre–sweep, 2 post–sweeps, no. coarse equations 10, relative tolerance 10^{-3}, convergence at 10^{-7} $\alpha_p = 0.3$	coupled k–ω–SST BiCGStab, preconditioner Cholesky, convergence at 10^{-12} $(\alpha_k)_C = 0.95$ $(\alpha_\omega)_C = 0.95$ $(\alpha_k)_S = 0.7$ $(\alpha_\omega)_S = 0.7$	

In this case, both solvers give approximately the same values of the force. The coupled solver is not faster in terms of execution time. However, the con-

vergence of the force coefficients demonstrates one of the distinct advantages of the implicitly coupled pressure–velocity solver, and that is stability. The value of drag and lift coefficients steadily converges (red line in Fig. 4.1), while the values calculated by the SIMPLE solver oscillate, quite heavily at the beginning of the simulation, until the amplitude decreases to reach a repetitive oscillation pattern. The oscillation is caused by the changes in the values of the pressure field, since the pressure component dominates the viscous component of the force. It can be suggested that the oscillation comes from the unsteady components of the solution. However, the oscillation of the force comes not only from the transient part of the solution, but also from the instabilities caused by decoupling the linearly coupled equations.

Front wing

In addition to the rear diffuser, an equally important aerodynamics component on a Formula 1 car is the front wing, which we have also simulated to compare the performance of the implicitly coupled and segregated solver. Front wing is an essential part, since it directs the flow towards the rear of the car where most of the downforce is generated. Thus, the geometry is very sensitive and it includes numerous small segments such as pylons, flaps, winglets and vanes, which poses a challenge in the meshing process. The cells which surround the surface of the wing must be very small to capture the smallest features of the geometry, while the overall number of cells must be as small as possible for efficiency reasons. Mesh statistics and boundary conditions are given in Table A6, while the slice through the volume mesh is shown in Fig. A8. The convergence of the residuals is shown in Fig. 4.4. Since we use consistent normalisation of the residuals, we can conclude that the implicitly coupled pressure velocity solver converges to a tighter tolerance than the SIMPLE algorithm.

However, convergence of the residual can sometimes be misleading regarding the behaviour of the flow field, i.e. the residual can locally be large, while the flow variables have converged and do not change anymore, or vice versa. For external aerodynamics, it is beneficial to monitor the convergence of an integral value, such as the drag force. The convergence of drag acting on the front wing is shown in Fig. 4.5. The superior stability of the coupled solver compared to the segregated solver is apparent, which also yields faster convergence both in terms of number of iterations and execution time. The pressure field acting on the surface of the wing is shown in Fig. A9. The vortices in the wake of the wing are shown in Fig. A10.

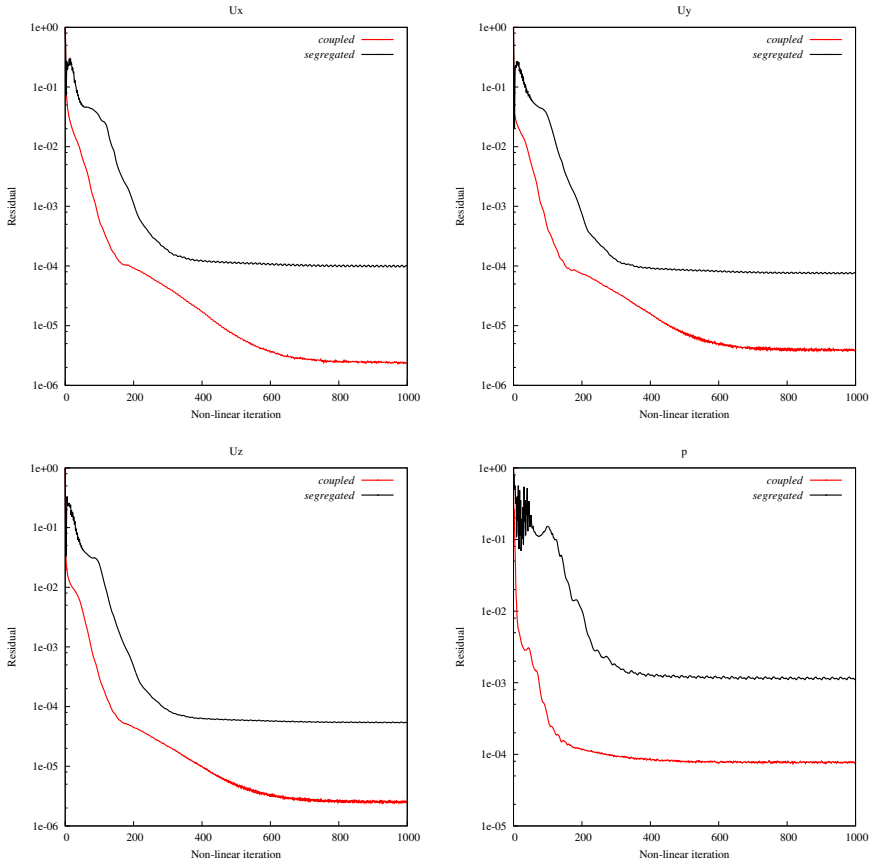


Figure 4.3: Front wing: convergence of the residual for segregated and coupled solver against the number of non–linear iterations.

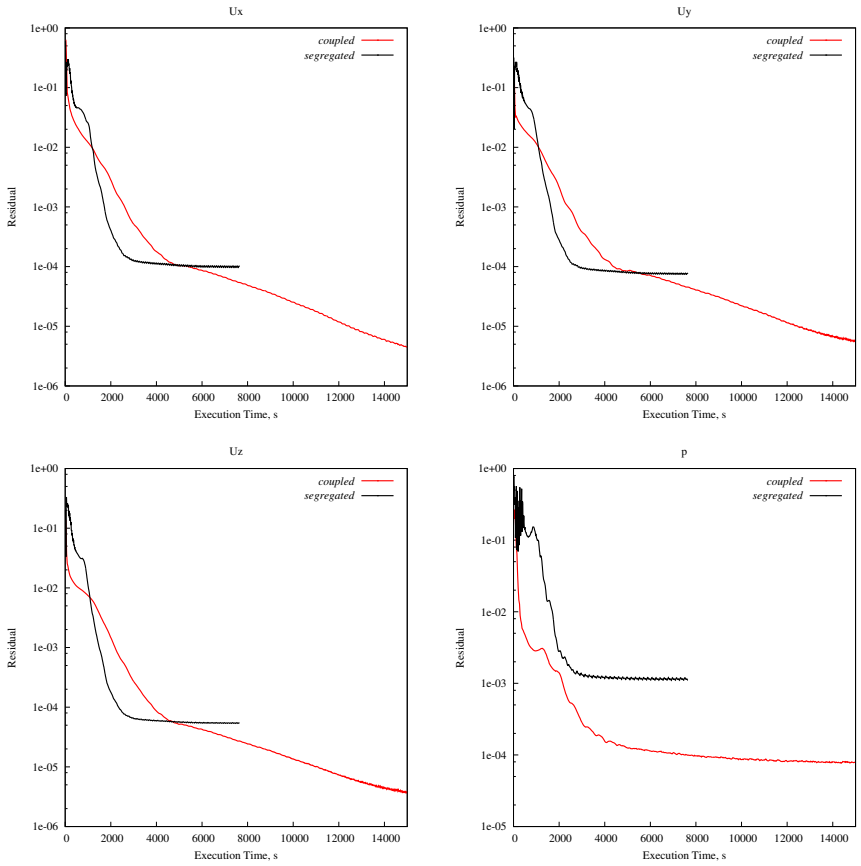


Figure 4.4: Front wing: convergence of the residual for segregated and coupled solver against execution time.

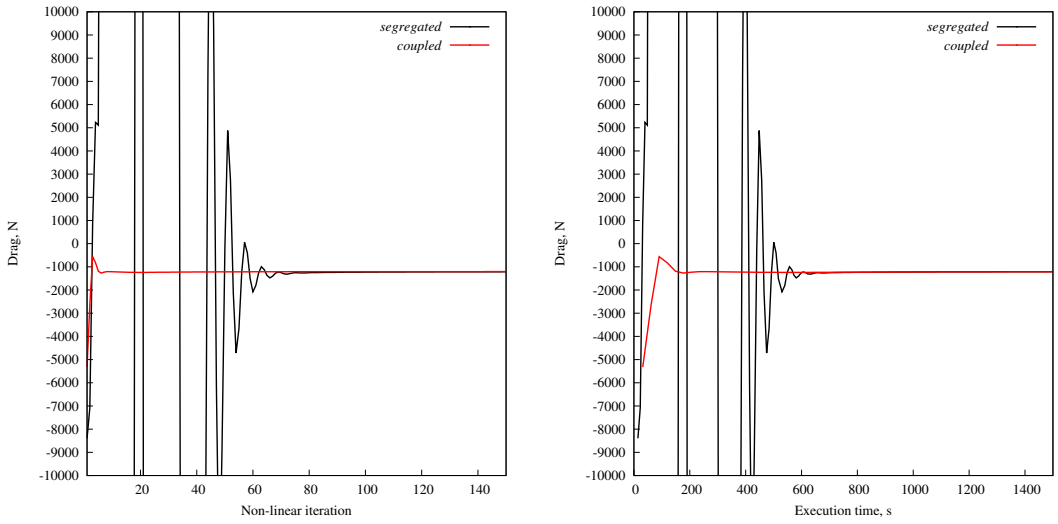


Figure 4.5: Front wing: convergence of the drag and lift force for segregated and coupled solver.

BB2–submarine

A mesh refinement study was conducted for the generic BB2 submarine test case, which is a collaborative project of several institutions, in order to generate a deeper understanding of the fluid dynamics processes and relative capabilities of the computational methods, by gathering and interchanging data among all participants.

Table 4.2: BB2 submarine: mesh properties and boundary conditions.

	BB2 SUBMARINE						
	Mesh data		**Boundary conditions**				
FINE	*No. cells*	14 357 865	**Boundary**	**u**	p	k	ω
	Cell type	hexahedral (structured)	*inlet*	Dirichlet (-3 0 0)	von Neumann (0)	Dirichlet (0.00135)	Dirichlet (112.5)
	Average non–orthogonality	10.95	*outlet*	von Neumann (0)	Dirichlet (0)	von Neumann (0)	von Neumann (0)
	Maximum non–orthogonality	82.30	*top wall, bottom wall*	Dirichlet (-3 0 0)	von Neumann (0)	wall function	wall function
	Maximum skewness	1.69	*front wall, back wall*	Dirichlet (-3 0 0)	von Neumann (0)	von Neumann (0)	von Neumann (0)
MEDIUM	*No. cells*	6 958 240	*submarine hull*	Dirichlet (0 0 0)	von Neumann (0)	wall function	wall function
	Cell type	hexahedral (structured)					
	Average non–orthogonality	11.0					
	Maximum non–orthogonality	79.61					
	Maximum skewness	1.78					
COARSE	*No. cells*	2 777 304					
	Cell type	hexahedral (structured)					
	Average non–orthogonality	11.12					
	Maximum non–orthogonality	80.16					
	Maximum skewness	2.53					

The length of the model in this study is $L_{\text{oa}} = 3.826\,\text{m}$. Straight flight conditions were simulated with freestream velocity u_∞ equal to $3\,\text{m/s}$. Three mesh densities were used for the simulations: statistics for each mesh are shown in Table 4.2, as well as boundary conditions. The three mesh densities are shown in Fig. A13. The comparison of the measured and simulated pressure coefficient and non–dimensional wall shear stress in x–direction, exhibits good quantitative agreement between the two data sets, Fig. 4.6.

The stability of the implicitly coupled solver is highlighted once more by the convergence of the force acting on the submarine hull, Fig. 4.7. Another

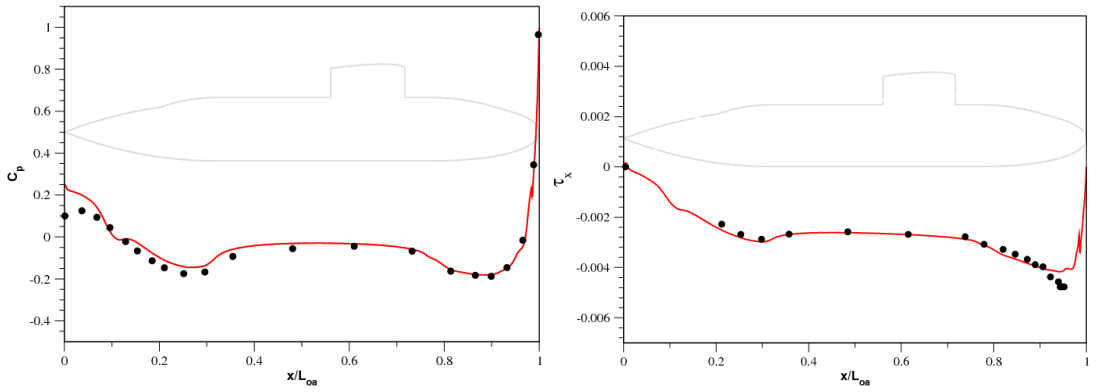


Figure 4.6: BB2 submarine: comparison of experimental data vs. data obtained from the simulation with implicitly coupled pressure–velocity solver. Pressure coefficient $c_p = \frac{p}{\frac{1}{2}\rho u_\infty^2}$ on the bottom of the hull, and non–dimensional wall shear stress in x–direction $\tau_x = \frac{\tau_x}{\frac{1}{2}\rho u_\infty^2}$ on the bottom of the hull.

important advantage of the coupled solver is evident - the convergence of the non–linear solver in terms of the number of iterations does not depend (or only slightly depends) on mesh density. We have also tested the segregated SIMPLE algorithm with two mesh densities, the coarse and the fine one. It can be seen that the convergence of the force for SIMPLE changes with the mesh density, i.e. it deteriorates as the number of cells increases.

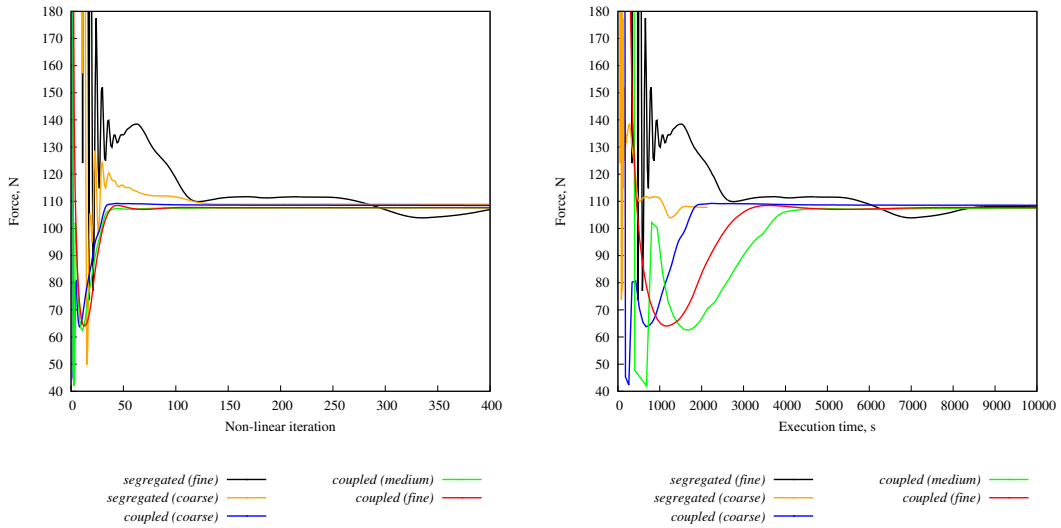


Figure 4.7: BB2 submarine: convergence of the total force onto the hull for three mesh densities, segregated and coupled solver.

However, as the number of cells and consequently size of the matrix increases, CPU time needed to reach convergence increases as well, i.e. each non–linear iteration becomes more time–consuming, which is expected due to a larger number of floating point operations. It is also interesting to observe the convergence of the residual, Fig. 4.8, especially the residual of the pressure equation. For coarse and medium mesh density, the coupled solver stalls at a higher value (10^{-4}), while it drops to 10^{-6} for the fine mesh. Since the mesh density was mainly increased in the submarine wake (the mesh close to the surface was constructed to satisfy the prerequisites of the turbulence wall functions), we can conclude that the coupled solver manages to "fully" resolve the flow only at the finest mesh. The SIMPLE algorithm however, requires a very large number of iterations to resolve the flow in the wake, which was not achieved within the maximally allowed number (600) of non–linear iterations. That is, the combination of the implicitly–coupled solver and AMG linear solver, whose convergence (in terms of the number of iterations) should be insensitive to changes in mesh density. In that case, the computational effort scales linearly.

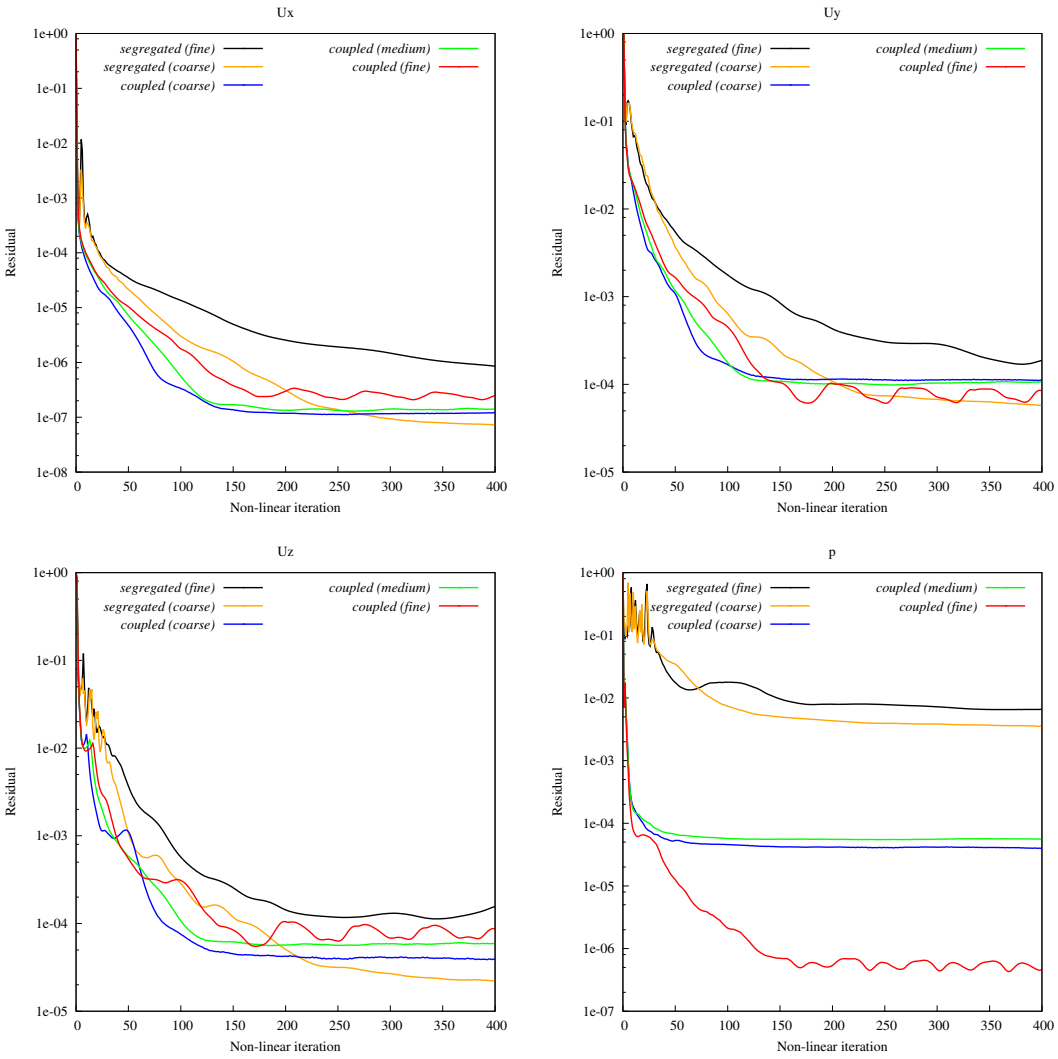


Figure 4.8: BB2 submarine: convergence of field variables for three mesh densities, segregated and coupled solver.

Francis turbine

The equations of basic, incompressible turbulent flow can be extended for applications in e.g. turbomachinery by adding the appropriate terms which take into account the effects of rotation. Since both rotating and stationary parts of the domain can be present, a *multiple reference frame (MRF)* approach is used [76], where the domain is divided into stationary and rotating zones. The equations are steady–state, thus a single position of the rotating part of the domain is simulated ("frozen rotor"). The equations in the stationary part of the domain remain the same (Eqn. (2.1), Eqn. (2.2)), expressed in terms of absolute (inertial) velocity. In the rotating part, equations are written via relative velocity. Coriolis and centrifugal forces are added and the equations are then cast into the inertial (absolute) coordinate system, which yields:

$$\nabla \cdot \mathbf{u} = 0, \tag{4.6}$$

$$\nabla \cdot (\mathbf{u_R u}) - \nabla \cdot (\nu \nabla \mathbf{u}) + \boldsymbol{\Omega} \mathbf{u}^T = -\nabla p, \tag{4.7}$$

where $\mathbf{u_R}$ is the relative velocity, $\boldsymbol{\Omega}$ is the angular velocity in the direction of the axis of rotation and $\mathbf{u} = \mathbf{u_R} + \boldsymbol{\Omega} \mathbf{r}_\mathrm{d}^T$. $\mathbf{r}_\mathrm{d}$ is the distance from the axis of rotation and it is orthogonal to the axis.

Most terms in equations corresponding to a relative reference frame, Eqn. (4.6) and Eqn. (4.7) are discretised in the same way as presented in Chapter 2., Section 2.4.. However, the volumetric flux in the convection term is expressed in terms of relative velocity $\mathbf{u_R}$:

$$\Phi_\mathrm{R} = \Phi - (\boldsymbol{\Omega} \mathbf{r}_\mathrm{d}^T)^T \, \mathbf{s}_f. \tag{4.8}$$

Additional term in the momentum equation $\boldsymbol{\Omega} \mathbf{u}^T$ can be expressed using the Hodge dual $\mathbf{W}$ of angular velocity:

$$\int \boldsymbol{\Omega} \mathbf{u}^T \, dV \; \rightarrow \; \int \mathbf{W} \mathbf{u} \, dV = V_i \mathbf{W} \mathbf{u}, \tag{4.9}$$

$$\mathbf{W} = \begin{bmatrix} 0 & -\Omega_z & \Omega_y \\ \Omega_z & 0 & -\Omega_x \\ -\Omega_y & \Omega_x & 0 \end{bmatrix}. \tag{4.10}$$

Thus, the term appears in the diagonal block–element $\mathbf{A}_{ii}$ of the momentum equation, it introduces the coupling between components of velocity in different directions, and the contribution is skew–symmetric:

$$\mathbf{A}_{ii} = \begin{bmatrix} a_{u_{xi},u_{xi}} & a_{u_{xi},u_{y_i}} & a_{u_{xi},u_{zi}} & a_{u_{xi},p_i} \\ a_{u_{y_i},u_{xi}} & a_{u_{y_i},u_{y_i}} & a_{u_{y_i},u_{zi}} & a_{u_{y_i},p_i} \\ a_{u_{zi},u_{xi}} & a_{u_{zi},u_{y_i}} & a_{u_{zi},u_{zi}} & a_{u_{zi},p_i} \\ a_{p_i,u_{xi}} & a_{p_i,u_{y_i}} & a_{p_i,u_{zi}} & a_{p_i,p_i} \end{bmatrix}.$$

It is possible to treat this term explicitly, i.e. calculate the outer product using the velocity from previous iteration and put the contribution into the right hand side:

$$\mathbf{b}_i = \begin{bmatrix} b_{u_{xi}} \\ b_{u_{y_i}} \\ b_{u_{zi}} \\ b_{p_i} \end{bmatrix}.$$

Flow inside a Francis turbine was simulated using a block–structured mesh built from three separate blocks. The case is distinctive since the non–matching blocks are connected using the Generalised Grid Interface (GGI) [76] compatible with selection AMG, and the Multiple Reference Frame (MRF) approach is used for the rotation zone near the rotor, with implicit tretment of the additional equation terms.

The positions of the GGI interface are shown in Fig. 4.9. Mesh statistics and boundary conditions are shown in Table A5, while the surface mesh of the rotor is shown in Fig. A6. Experimental data [77] are available for the best efficiency point, and we have used the corresponding inlet boundary conditions (mass flow rate and direction): the value of the measured head is 11.91 m, while the measured power is equal to 21.6 kW. Convergence of these integral values for the coupled and segregated solver is shown in Fig. 4.10.

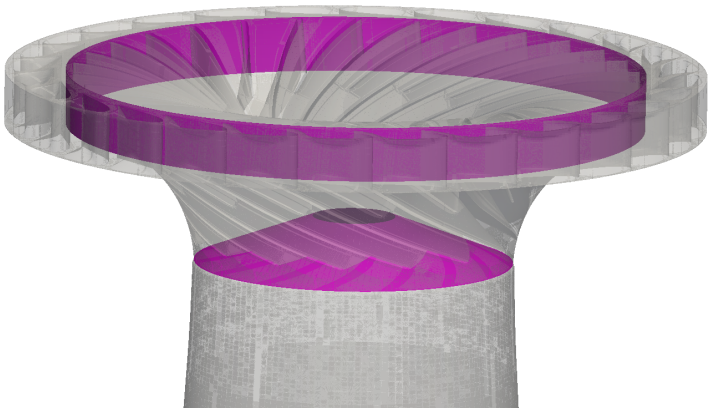


Figure 4.9: Francis turbine: GGI interfaces between the stay vanes and rotor, and between rotor and the draft tube.

The coupled solver exhibits a more stabile, faster convergence of both head and power. It is also important to notice that SIMPLE overshoots the stationary value achieved by the coupled solver. It is expected that the value will gradually drop to the same value as the coupled solver's, but a very large number of non–linear iterations is needed due to the low underrelaxation factor (0.3) used for the pressure field. A slice showing the velocity field arount stay vanes and impeller and the velocity field at the diffuser outlet are shown in Fig. A7.

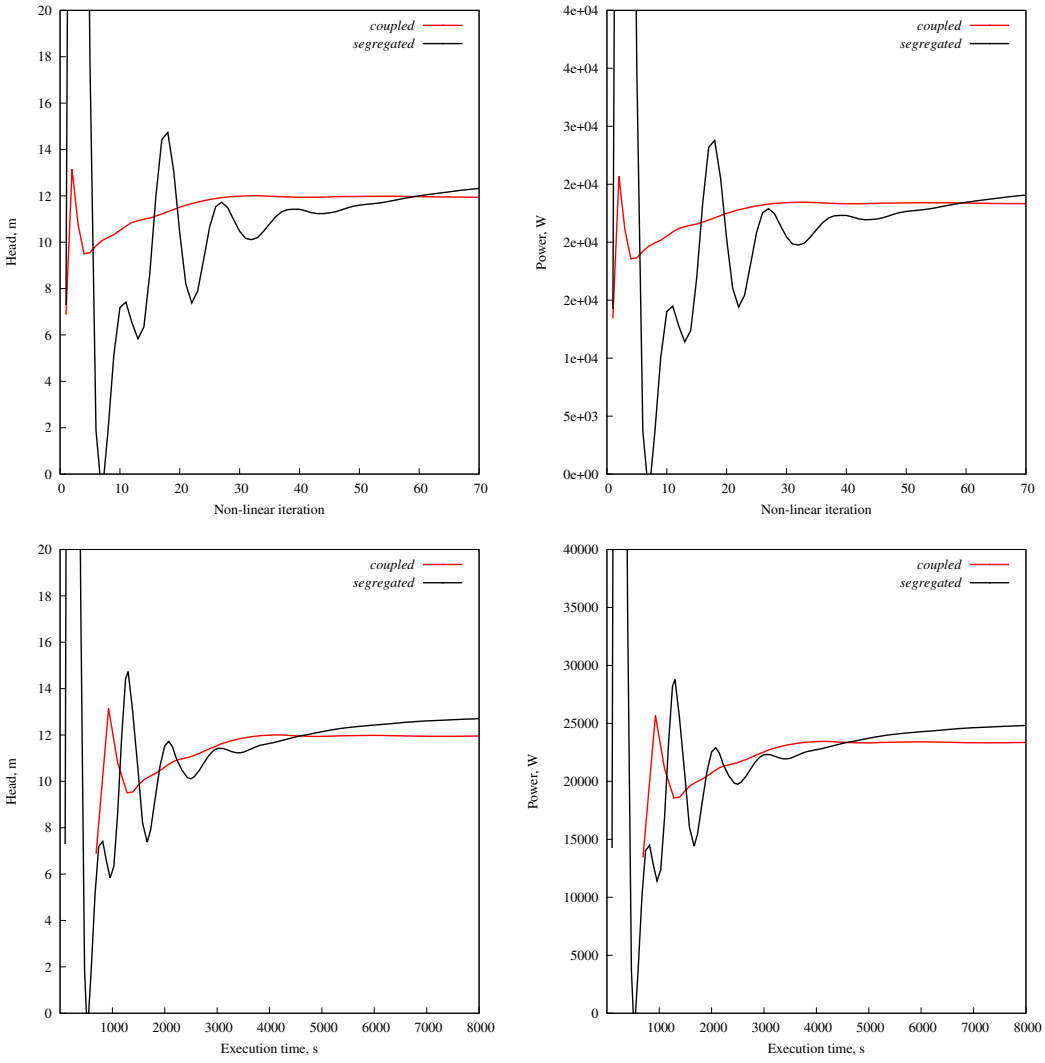


Figure 4.10: Francis turbine: convergence of turbine head and power for the implicitly coupled and SIMPLE algorithm.

Centrifugal pump

We were motivated to investigate another internal flow case using a more challenging case in terms of the computational mesh. A hybrid mesh consisting of 9 054 517 cells was created in Pointwise [78] for a centrifugal pump geometry. Block–structured hexahedral mesh was used in the rotating zone around the impeller, while a tetrahedral unstructured mesh was created in the spiral casing and the pipeline, Fig. 4.12. A GGI interface was used to connect the two sections of the mesh, Fig. 4.13. Mesh statistics and boundary conditions are given in Table A2. Also, the implicit and explicit treatment of MRF terms was tested on this case, as well as two linear solvers for the coupled system (block–selection AMG and BiCGStab). The fields obtained from the simulation with block–selection AMG are shown in Fig. A2.

Fig. 4.11 shows that the implicitly coupled solver converges in approx. 400 non–linear iterations while the segregated solver needs 1250 iterations. That is, implicitly coupled solver is always significantly faster in terms of the number of non–linear iterations. For this case, it is several times faster considering the execution time as well.

Even though a single non–linear solution of the implicitly coupled system is more expensive than a non–linear solution of the segregated system, minimal underrelaxation can be used for the momentum equation in the coupled solver. In this case, we used $\alpha_{\mathbf{u}} = 0.95$ to improve the convergence of the block–linear solver. In some cases, the linear solution of the block–system diverges if no underrelaxation is used for the momentum equation. In this case, the linear solver needs less iterations in a single non–linear iteration to reach the desired tolerance, if the underrelaxation is lowered to $\alpha_{\mathbf{u}} = 0.85$, see Table 4.3.

Table 4.3: Centrifugal pump: dependence of performance of linear and non–linear implicitly coupled solver on underrelaxation factor.

Linear solver	$\alpha_{\mathbf{u}}$	**Total number of linear iterations until non–linear iteration 1000**	**Execution time at non–linear iteration 1000, s**	**Converged at non–linear iteration**	**Execution time until convergence, s**
SAMG	0.85	3267	24 638.2	900	22 157.4
SAMG	0.95	6999	32 207.8	400	12 932.0

For example, in non–linear iteration 280 , there are 9 linear iterations if $\alpha_{\mathbf{u}} =$

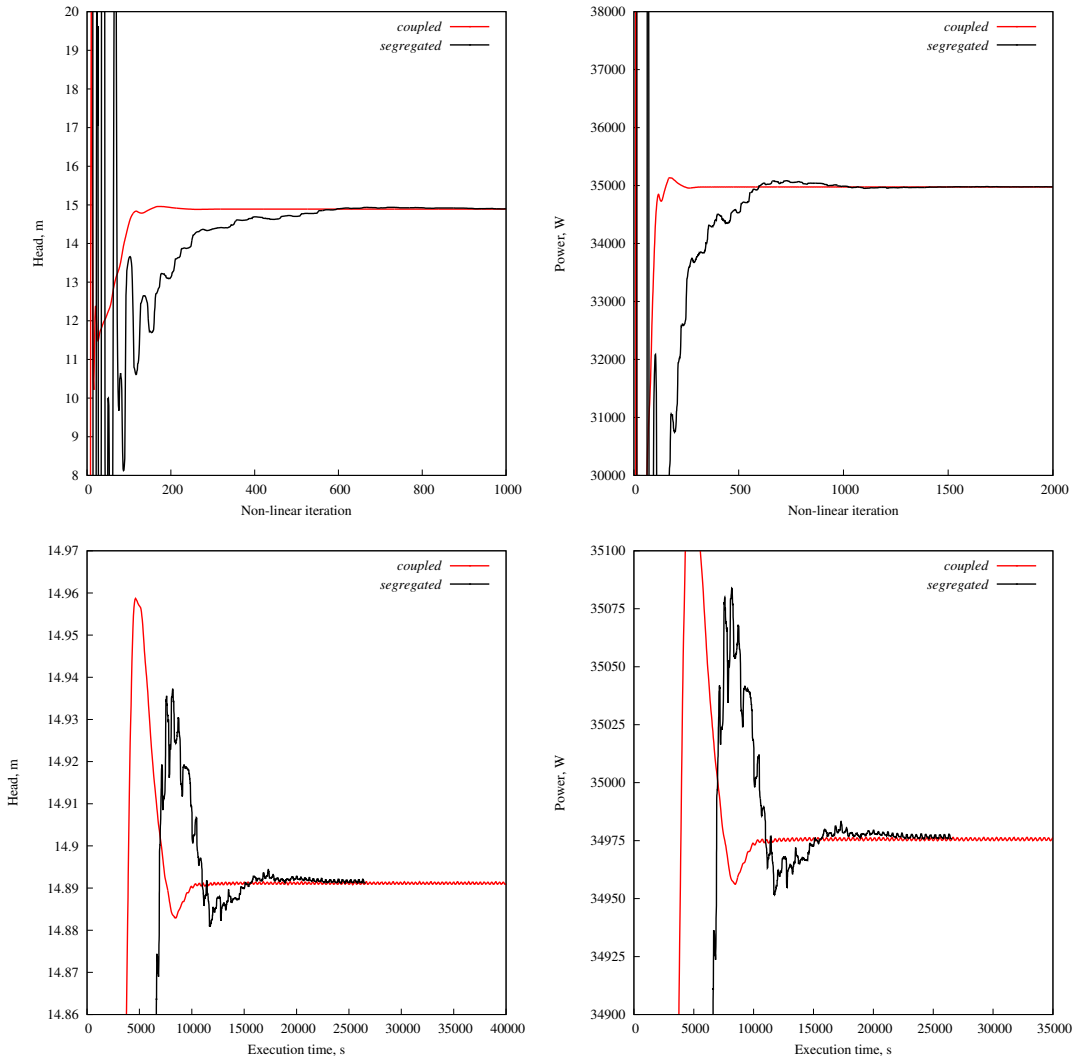


Figure 4.11: Centrifugal pump: convergence of pump head and power for the implicitly coupled and SIMPLE algorithm.

0.95 is used, while there are 3 linear iterations for $\alpha_{\mathbf{u}} = 0.85$. However, for the centrifugal pump, the best result in terms of execution time was achieved with minimal underrelaxation, even though the total number of linear iterations per non–linear iteration is higher. Momentum equation in the SIMPLE algorithm was underrelaxed with the usual $\alpha_{\mathbf{u}} = 0.7$, while the values of pressure were underrelaxed with $\alpha_p = 0.3$.

Regarding the performance of BiCGStab, Fig. 4.15 shows that the non–linear solver does not converge when using BiCGStab preconditioned by ILUC0 as

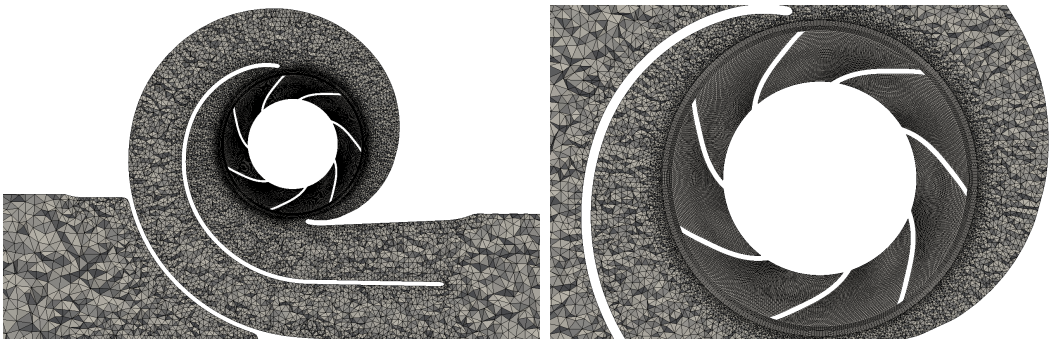


Figure 4.12: Centrifugal pump: slice showing a detail of a hybrid computational mesh.

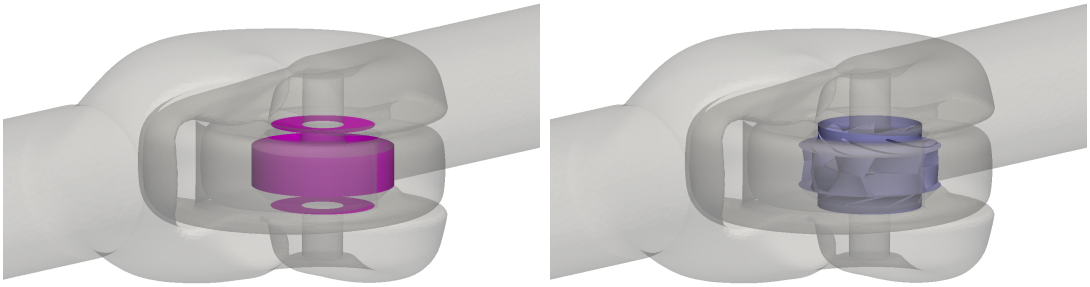


Figure 4.13: Centrifugal pump: GGI interface which connects the structured and unstructured section of the mesh (left), and the position of the impeller (right).

the linear solver. In this case, it is possible to achieve non–linear convergence by using a smaller underrelaxation factor, e.g. $\alpha_{\mathbf{u}} = 0.75$, which deteriorates non–linear convergence. Explicit treatment of MRF terms also gives bad convergence, i.e. requires modification of the non–linear solver parameters which slows down non–linear convergence. In conclusion, convergence of both the implicitly–coupled solver and SAMG is insensitive to cell type.

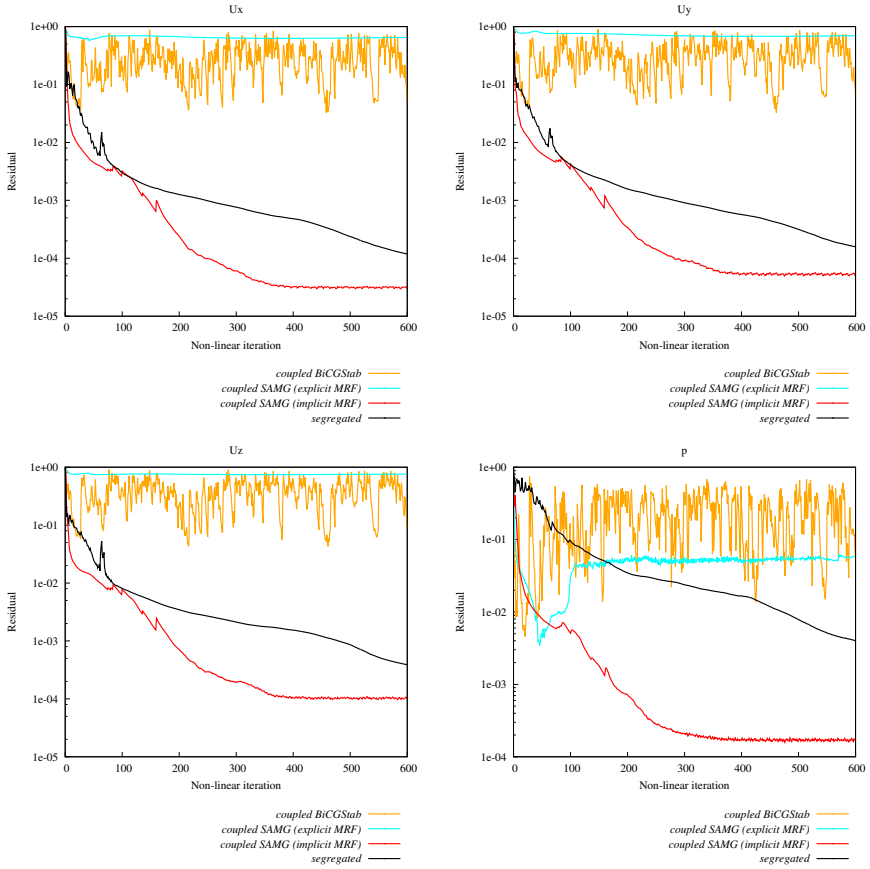


Figure 4.14: Centrifugal pump: convergence of field variables for the segregated and coupled solver against the number of non–linear iterations. Coupled solver was run with block–selection AMG and BiCGStab linear solvers, as well as explicit MRF terms.

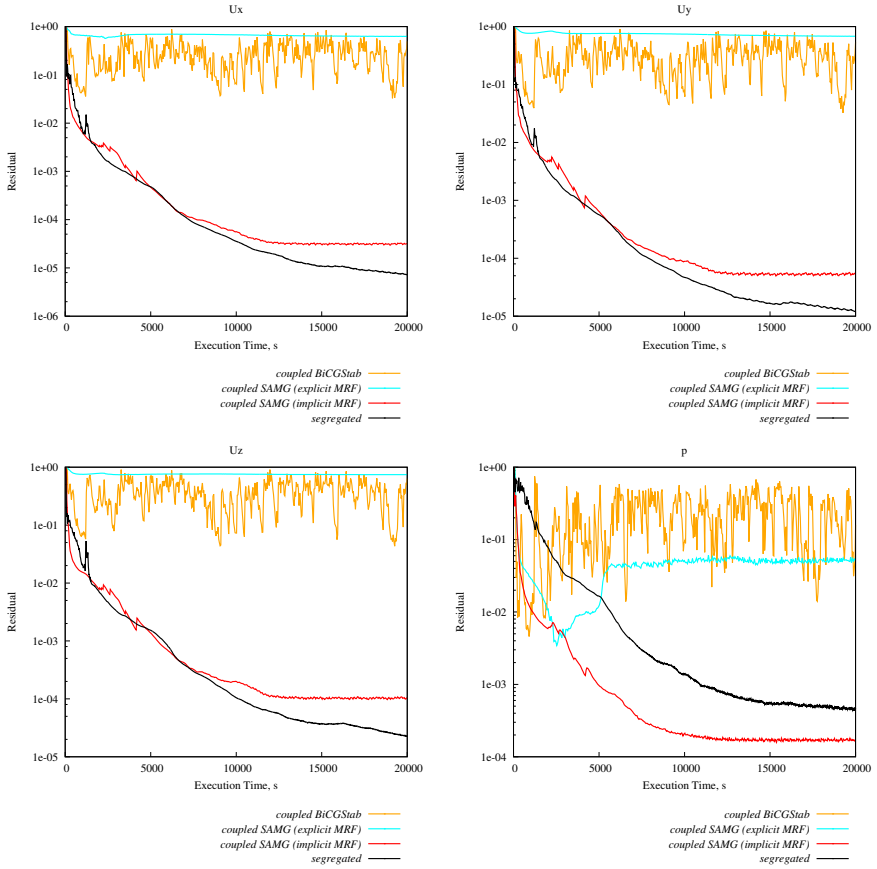


Figure 4.15: Centrifugal pump: convergence of field variables for the segregated and coupled solver against execution time. Coupled solver was run with block–selection AMG and BiCGStab linear solvers, as well as explicit MRF terms.

4.3. Performance of the Selection Algebraic Multigrid Algorithm

In this section we shall answer the questions regarding the setup of the block–selection algebraic multigrid solver: choice of the cycle, block–element norm, size of the strength factor for determining strong connections, the number of coarse levels, order of smoothing etc., as well as provide a comparison to other linear solvers.

It is important to emphasize the synergy of different components of linear solution procedure which we employ, all united under the name SAMG:

- The fine level solution, prior to running the multigrid cycle, is calculated using the biconjugate gradient stabilised solver (BiCGStab), preconditioned by the incomplete lower upper factorisation based on Crout's algorithm (ILUC0).
- The pre– and post–smoothing sweeps are done using the ILUC0 algorithm.
- We do not use a direct solver for the solution on the coarsest level. Instead, we define the dimension of the matrix on the coarsest level to be sufficiently small to ensure that ILUC0 smoother behaves almost like a direct solver.
- After running the multigrid cycle, which efficiently calculates the pressure field, fine level solution is again obtained using BiCGStab, which updates the velocity field.

The cases were run on a single CPU core, except for the BB2 parallel scaling test. The first case is the cooling of an engine jacket, for which we have compared the performance of different linear solvers. The mesh is unstructured, dominantly hexahedral with 156 739 cells, shown in Fig. A4. The statistics are shown in Table A4. The flow is illustrated in Fig. A5.

The following setup was used for specific runs:

- *SAMG*: selection AMG, smoother ILUC0, V-cycle, pressure norm, minimal number of coarse equations = 20, convergence tolerance = 10^{-5}, max

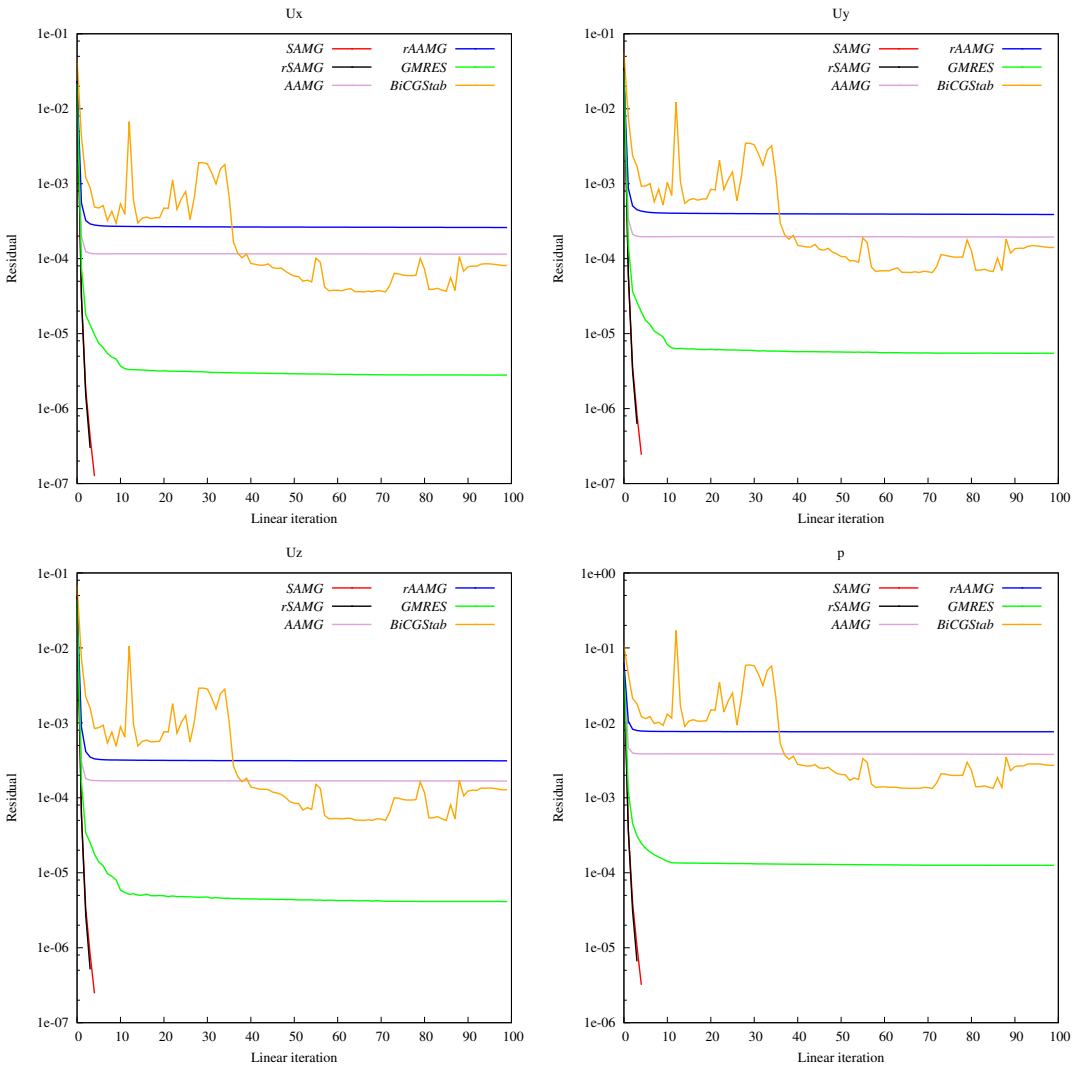


Figure 4.16: Engine cooling: convergence of field variables for different linear solvers in the 15^{th} non–linear iteration of the implicitly coupled pressure–velocity solver.

number of iterations = 100. The equations in coarse matrices are sorted according to the order of selection;

- *rSAMG*: setup is identical to SAMG, with a single difference: the order of equations on coarse levels is consistent with the finest level;
- *AAMG*: additive correction AMG, smoother ILUC0, V-cycle, pressure norm, cluster size = min 2 max 5, minimal number of coarse equations = 20, convergence tolerance = 10^{-5}, max number of iterations = 100. The equations

in coarse matrices are sorted according to the order of cluster formation;;

- *rAAMG*: setup is identical to AAMG, with a single difference: the order of equations on coarse levels is consistent with the finest level;
- *GMRES*: preconditioner ILUC0, number of directions = 10, convergence tolerance = 10^{-5}, max number of iterations = 100.
- *BiCGStab*: preconditioner ILUC0, convergence tolerance = 10^{-5}, max number of iterations = 100.

The conclusions drawn for this particular case were also noticed for numerous other cases with complex three–dimensional flow. Convergence of linear solvers in the 15^{th} non–linear iteration of the engine cooling simulation is shown in Fig. 4.16.

The advantage of any version of SAMG is evident: the residual drops 5 orders of magnitude in 5 or 6 iterations, i.e. SAMG achieves theoretical convergence of order of magnitude residual reduction per W–cycle, see Appendix C by A. Brandt in [27]. GMRES reduces the residual approximately 4 orders of magnitude but then stalls (possibly because of the limited number of search directions), while AAMG and BiCGStab reduce the residual 2-3 orders of magnitude. BiCGStab exhibits a highly oscillatory convergence. Regarding the order of equations on coarse levels, we examined whether keeping the same optimised ordering of equations on coarse levels is beneficial. Fine level matrix is banded and, even though additional off–diagonal elements appear on the coarse levels, retaining the same structure could prove to be valuable for the ILU smoother since there could be fewer elements which are ignored compared to the full factorisation. The alternative ordering of equations arises naturally from the coarsening process: the equations are sequentially written in the coarse level matrix in the order of appearance which is related to the order of selection, which equals the order of strength. In this ordering, the strongest equations with highest influence to other equations, are visited first. From Fig. 4.16 we can conclude that keeping the ordering of equations consistent between levels results in slightly better convergence, i.e. fewer iterations are necessary to achieve the desired tolerance.

To visualise the coarsening process using the computational mesh and compare the possible SAMG settings, we have chosen a traditional two–dimensional

Table 4.4: Backward–facing step: comparison of different settings for the multigrid solver (non–linear iteration 50).

ID	Solver	Norm	Order of equations on coarse levels	Strength of connection (factor)	Underrelaxation factor (U)	No. levels	No. iterations
1	AAMG	pressure	renumbered	0.2	0.99	9	92
2	SAMG	2–norm	renumbered	0.2	0.99	7	23
3	SAMG	pressure	order of appearance	0.2	0.95	7	4
4	SAMG	pressure	renumbered	0.1	0.98	6	3
5	SAMG	pressure	renumbered	0.25	0.98	7	4
6	SAMG	pressure	renumbered	0.3	0.99	7	7
7	SAMG	pressure	renumbered	0.2	0.99	7	4

Table 4.5: Backward–facing step: coarsening statistics for the 50^{th} non–linear iteration.

	No. equations on level									Average number of off–diagonal elements per row on level								
ID	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9
1	4800	2332	1144	547	247	115	50	20	9	4.0	3.9	4.2	4.6	4.9	5.1	4.9	4.3	3.6
2	4800	2400	730	190	56	18	5	-	-	4.0	7.3	9.2	9.9	9.2	5.7	2.8	-	-
3	4800	2400	811	238	69	20	7	-	-	4.0	7.3	9.5	11.8	10.4	6.6	3.7	-	-
4	4800	2400	739	199	50	14	-	-	-	4.0	7.4	9.8	11.5	9.2	5.3	-	-	-
5	4800	2400	838	255	74	20	8	-	-	4.0	7.2	9.5	11.9	10.3	5.7	3.5	-	-
6	4800	2400	852	273	77	24	7	-	-	4.0	7.2	9.3	11.6	10.1	6.8	3.4	-	-
7	4800	2400	814	241	68	23	6	-	-	4.0	7.3	10.0	12.3	10.1	7.3	3.0	-	-

test case of the flow inside a duct with a backward–facing step. The computational mesh is shown in Fig. A1, while the statistics are given in Table A1. The investigated settings are shown in Table 4.4. For all cases we have prescribed the same minimal number of coarse level equations (4), convergence tolerance (10^{-6}) and maximum number of iterations (100). Group size for AAMG ranges from 2 to 4 cells per cluster. Different settings produce different number of coarse levels, i.e. they change the coarsening pattern, which consequently affects the interpolation weights and the structure of the coarse level matrices.

From Fig. 4.17, convergence of linear solver shows that option 4, i.e. identifying strong equation couplings to be greater or equal to 10% of the strongest coupling gives significant advantage. The reason could lie in the fact that the cells in the mesh are anisotropic, i.e. they are elongated in direction of the flow. Since the discretisation of the pressure Laplacian contains the surface normal face area

vector, the off–diagonal elements are proportional to the area of the face which the corresponding cells share assuming constant diffusivity. Ideally, when looking at this two–dimensional structured mesh the coarsening process should produce an evenly distributed checkerboard pattern on the first coarse level, since the Laplacian is elliptic. However, anisotropic cells effect the strength of off–diagonal elements, and there appear equations which weakly depend on each other (small off–diagonal element) and both are declared coarse. This phenomenon is expected for meshes with highly anisotropic cells (e.g. boundary layers, farfield of structured meshes), and it can affect the number of coarse levels (increase in the number of levels increases CPU time) as well as the performance of the smoother. Thus, the best performance of SAMG would be achieved on uniform meshes, i.e. if all the cells were as close as possible to perfect cubes, with minimal anisotropy.

An overview of the coarsening process is given in Figs. 4.18–4.22 and it can be seen that each setup results in a different coarsening pattern. Defining a smaller strength factor reduces the occurance of two coarse equations which depend on each other, i.e. it enables more aggressive coarsening (fewer coarse levels) and improves convergence of the linear solver. Fig. 4.22 reveals that the most uniform distribution of coarse level equations throughout the domain is indeed achieved with the lowest strength factor. However, the underrelaxation of the momentum equation had to be lowered to 0.98 to prevent divergence of the linear solver which happened at the exact iteration in which the recirculation behind the step started to form (5^{th} non–linear iteration). Lowering the underrelaxation factor has a negative impact on the non–linear convergence, shown below in Fig. 4.17. The best performance was achieved with strength factor equal to 0.2, ID 7 in Table 4.4, which will be confirmed as the best value for a complex case as well.

Coarsening is conducted only once in each non–linear iteration and kept frozen during the linear iterations (V–cycle is repeated using the same prolongation, restriction and coarse level matrices). An important observation is that the coarsening pattern changes in each non–linear iteration if there are large variations in the solution (updates of the coefficient matrix). The large variations of the solution also cause a larger number of linear iterations in a single non–linear iteration in comparison to a stabilised solution when only a minimal number of linear iterations is conducted. Thus, a possible strategy suggested by Clees [30]

to reduce the cost of the setup phase by freezing and reusing the coarsening is generally not favourable.

Coarsening statistics for the 50^{th} non–linear iteration are given in Table 4.5. Fine level matrix has 4800 rows, marked as level 1 in the table. AAMG coarsening produced 8 coarse levels (average reduction factor 2.2), while SAMG coarsening (with the majority of settings) produced 7 levels (average reduction factor for case 7 is 3.11). Grey part of the table shows the average number of off–diagonal elements per row on the individual level. If we were to depict a mesh which corresponds to the connectivity of the coarse level matrices, AAMG would have a hexahedral mesh on coarse levels as well. Unlike AAMG, SAMG would have polyhedral coarse meshes, which imposes additional challenges on the smoother. That is, the bands of the matrix have broaden and there are more positions where an element would appear in the full factorisation.

Uniformity of the mesh, which is assocciated with the uniformity of the magnitude of matrix elements, is important for the ILU factorisation as well. If there are cells in a mesh with volumes of different order of magnitude, e.g. small cells in the refined wake zone or around sharp features of the geometry, and very large cells in distant regions of the farfield, the solution of the system might diverge. That is, during the ILU sequence, rows with very small elements might be used to eliminate the lower triangle elements which are large. The corresponding multiplier is very large and discarded factorisation elements might be of crucial importance for the solution of the system. This can be remedied by sorting the equations by the order of magnitude of diagonal elements (pivoting), which was not implemented in the scope of this thesis.

Another problem which may occur in some cases is the loss of weak diagonal dominance in coarse level matrices. For example, the Dirichlet boundary condition on the outlet section of the backward facing step case ($p_{\text{outlet}} = 0$) ensures that all equations which correspond to cells on the outlet patch, are diagonally dominant. To retain the diagonal dominance on the coarse level, a diagonally dominant equation should always be chosen as coarse, or the effect of diagonal dominance has to be transferred to its coarse neighbours through interpolation. This is not done by force, i.e. it is always achieved naturally through the coarsening process. A problem may occur if there is only a single diagonally dominant

equation on the fine level, i.e. if only von Neumann boundary conditions are used for pressure with a reference value at a point somewhere in the flowfield. It is very likely that diagonal dominance from a single equation will dissipate when calculating the coarse level matrices.

The third case for which we have examined the behaviour of linear solvers is the flow around a generic submarine with an unstructured mesh (1 939 796 cells), shown in Fig. A3. Mesh statistics and boundary conditions are shown in Table A3. The settings and corresponding results of all the tests we have conducted on this geometry are summarised in Tabs. 4.6 and 4.7, for non–linear iterations 3 and 10 respectively.

Table 4.6: Generic submarine: coarsening statistics for the 3$^{\text{rd}}$ non–linear iteration.

ID	Norm	Cycle	Order of eqns on coarse levels	Pre–sweeps	Post–sweeps	No. levels	No. coarse eqns on the last level	Strength of connection (factor)	No. iterations	CPU time, s
1	pressure	V–cycle	renumbered	2	2	7	323	0.2	9	124.3
2	pressure	V–cycle	renumbered	1	1	7	323	0.2	10	101.5
3	pressure	V–cycle	renumbered	1	2	7	323	0.2	9	111.2
4	pressure	V–cycle	renumbered	1	3	7	323	0.2	7	111.2
5	pressure	V–cycle	renumbered	2	1	7	323	0.2	11	123.7
6	pressure	V–cycle	renumbered	0	4	7	323	0.2	10	129.1
7	pressure	V–cycle	renumbered	3	1	7	323	0.2	9	127.6
8	2–norm	V–cycle	renumbered	2	2	13	107	0.2	100	765.5
9	pressure	V–cycle	renumbered	2	2	7	117	0.1	10	112.2
10	pressure	V–cycle	renumbered	2	2	7	1220	0.25	10	111.6
11	pressure	V–cycle	renumbered	2	2	7	1561	0.3	10	108.1
12	pressure	W–cycle	renumbered	2	2	7	323	0.2	7	223.1
13	pressure	V–cycle	order of appearance	2	2	7	379	0.2	9	141.0
14	pressure	V–cycle	renumbered	2	2	6	3279	0.2	9	122.1
15	pressure	V–cycle	renumbered	2	2	8	39	0.2	9	125.0

Table 4.7: Generic submarine: coarsening statistics for the 10$^{\text{th}}$ non–linear iteration.

ID	Norm	Cycle	Order of eqns on coarse levels	Pre–sweeps	Post–sweeps	No. levels	No. coarse eqns on the last level	Strength of connection (factor)	No. iterations	CPU time, s
1	pressure	V–cycle	renumbered	2	2	7	570	0.2	4	111.0
2	pressure	V–cycle	renumbered	1	1	7	570	0.2	5	108.0
3	pressure	V–cycle	renumbered	1	2	7	570	0.2	5	109.2
4	pressure	V–cycle	renumbered	1	3	7	570	0.2	4	111.3
5	pressure	V–cycle	renumbered	2	1	7	570	0.2	5	111.6
6	pressure	V–cycle	renumbered	0	4	7	570	0.2	4	111.0
7	pressure	V–cycle	renumbered	3	1	7	570	0.2	5	118.94
8	2–norm	V–cycle	renumbered	2	2	13	122	0.2	100	785.4
9	pressure	V–cycle	renumbered	2	2	6	1022	0.1	5	104.6
10	pressure	V–cycle	renumbered	2	2	7	648	0.25	5	102.1
11	pressure	V–cycle	renumbered	2	2	7	1262	0.3	5	98.4
12	pressure	W–cycle	renumbered	2	2	7	570	0.2	4	188.1
13	pressure	V–cycle	order of appearance	2	2	7	295	0.2	5	131.4
14	pressure	V–cycle	renumbered	2	2	6	3827	0.2	5	116.5
15	pressure	V–cycle	renumbered	2	2	8	46	0.2	4	109.8

In addition to settings which were investigated for the two–dimensional backward–

facing step case, we have also examined the number of pre– and post–smoothing sweeps (IDs 1, 2, 3, 4, 5, 6, 7), effect of the W multigrid cycle (ID 12) and maximum number of coarse level cells (IDs 14 and 15). The corresponding convergence of residuals is shown in Figs. 4.23 and 4.24.

In both non–linear iterations, the fewest linear iterations until the prescribed criterion was achieved with settings 12, 4 and 15, listed in Tables 4.6 and 4.7. Case 12 was run with the W–cycle, i.e. the equations were solved multiple times on coarser levels, which should result in better convergence. However, CPU time per non–linear iteration is much greater than for the V–cycle. Case 4 was run with 1 pre–smoothing sweep and 3 post–smoothing sweeps, which improved the convergence rate in comparison to the usual 2–2 setting. Thus, increasing the number of post–smoothing sweeps could improve convergence, but only up to a certain point. Running only post–smoothing (ID 6) gives one of the worst convergence rates. Another way to improve convergence is to have less coarse equations on the last coarse level, but this possibility also increases CPU time. Coarsening based on the 2–norm of block–elements (ID 8) gives bad convergence, mainly due to nonoptimal interpolation weights. Using less than 2 post–smoothing sweeps (IDs 2, 5) is one of the worst options. Setting the strength of connection strength factor to 0.1 (ID 9) does not work as well as for the anisotropic mesh of the backward–facing step case. On the contrary, it is one of the three worst rates of convergence in the 10^{th} non–linear iteration, Fig. 4.24.

The final question regarding the performance of block–selection AMG is whether our strategy for parallelisation, in which interpolation across processor boundaries is not allowed as well as serial smoothing with ILUC0, impairs linear and non–linear convergence. Therefore we have run the finest mesh of the BB2 submarine on our high performance computer to compare the performance with respect to the number of CPU cores. The maximum number of cores at our disposal at the time was 112 and we did a strong scaling test, using the same mesh density for all simulations. The simulation was run on 1, 2, 4, 6, 8, 16, 32, 64, 96 and 112 cores. Non–linear convergence was not reached when only 1 processor was used as the simulation was force stopped due to extremely slow calculations. The number of linear iterations per non–linear iteration at the beginning of each simulation is shown in Fig. 4.25.

As mentioned before, at the beginning of the simulation, when the solution is still oscillating, a higher number of linear iterations is needed to reach the linear convergence criterion, which can again be noticed for this case. When comparing the number of linear iterations, there is no significant difference between cases run on either number of cores. Non–linear convergence is also unaffected by the increasing number of cores, i.e. the same number of non–linear iterations was needed to reach convergence criterion. Regarding the scalability of the simulation, parallel efficiency is shown in Fig. 4.26, calculated as:

$$\eta_P = \frac{t_1}{N \cdot t_N} \cdot 100\%,$$

where t_1 is the amount of time for 1 processing unit to complete the work, N is the number of processing units and t_N the amount of time for N processing units to complete the work. Since the calculations with 1 core were extremely slow, we compared the total time necessary to complete 10 non–linear iterations, shown in Table 4.8. The speedup from 1 to 4 processors is super–linear, i.e. parallel efficiency is greater than 100%. The overall solution procedure is memory–intensive (fine and coarse level block–matrices are kept in memory during the solution procedure) and multiple processors have more cache (fast) memory available, which significantly reduces the computation time. Thus, we have also calculated parallel efficiency and speedup compared to case with 4 cores, shown in Table 4.8.

Table 4.8: BB2 submarine: statistics of parallel simulation tests.

No. CPUs	Time (iter 10), s	Parallel efficiency, % (compared to 1 CPU)	Speedup (compared to 1 CPU)	Time (iter 200), s	Parallel efficiency, % (compared to 4 CPUs)	Speedup (compared to 4 CPUs)
1	13013.4	100	1	-	-	-
2	5049.8	128.9	2.6	64205.9	-	-
4	2242.9	145.1	5.8	30402	100	1
6	1753.7	123.7	7.4	23240.1	87.2	1.3
8	1369.8	118.7	9.5	18984.6	80.1	1.6
16	755.7	107.6	17.2	10231.3	74.3	3.0
32	407.2	99.9	32.0	5551.1	68.5	5.5
64	216.9	93.7	60.0	3122.1	60.9	9.7
96	206.8	65.5	62.9	2348.1	53.9	12.9
112	195.6	59.4	66.5	2015.7	53.9	15.1

An important aspect for running a simulation in parallel is the domain decom-

position method. Decomposition algorithms try to minimise the number of cells on processor interfaces without taking into consideration the underlying physics. Some decompositions may cause divergence of linear solvers, especially if continuous transfer of information is crucial for the solution. An obvious example is decomposing the backward facing step vertically, i.e. perpendicular to the direction of the flow, which works very well; or decomposing it horizontally, parallel to direction of the flow, which causes the simulation to diverge.

Since the coarsening process is not parallelised, i.e. it is done separately on each processor, the reduction of global number of variables is slower compared to the sequential algorithm. At some level, continuing parallel coarsening becomes inefficient. Thus, joining a number of processes onto one processing unit is proposed, which could also prove to be beneficial for the convergence of ILUC0 algorithm. However, determining the number of coarse variables which would act as a switching criterion for joining the processes is not clear. Load balancing should be investigated for highly parallel SAMG, but was not considered in the scope of this thesis.

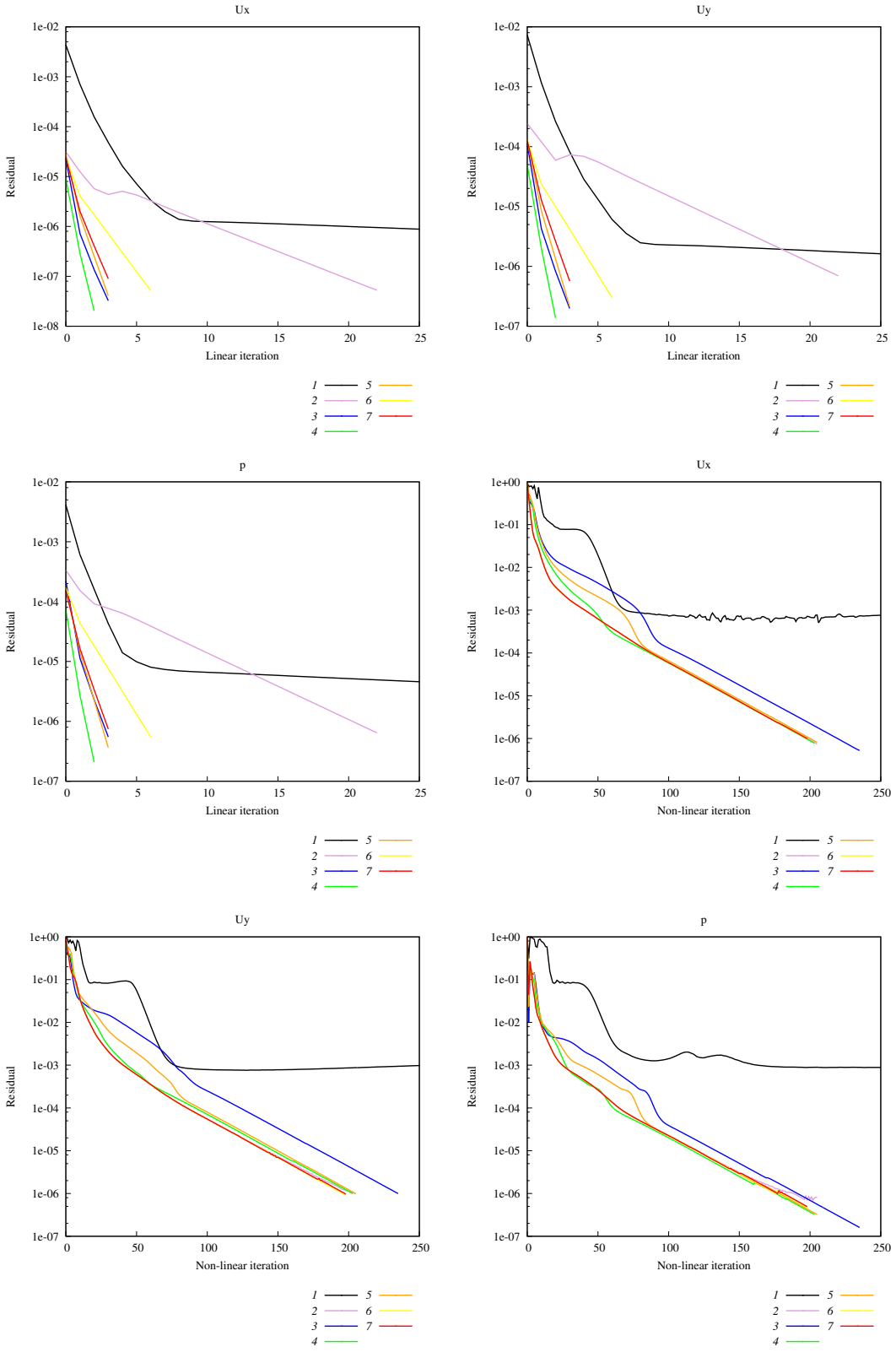


Figure 4.17: Backward–facing step: convergence of field variables for different settings (IDs in Table 4.4) of linear solver in the 50^{th} non–linear iteration of the implicitly coupled pressure–velocity solver and non–linear convergence for the same settings.

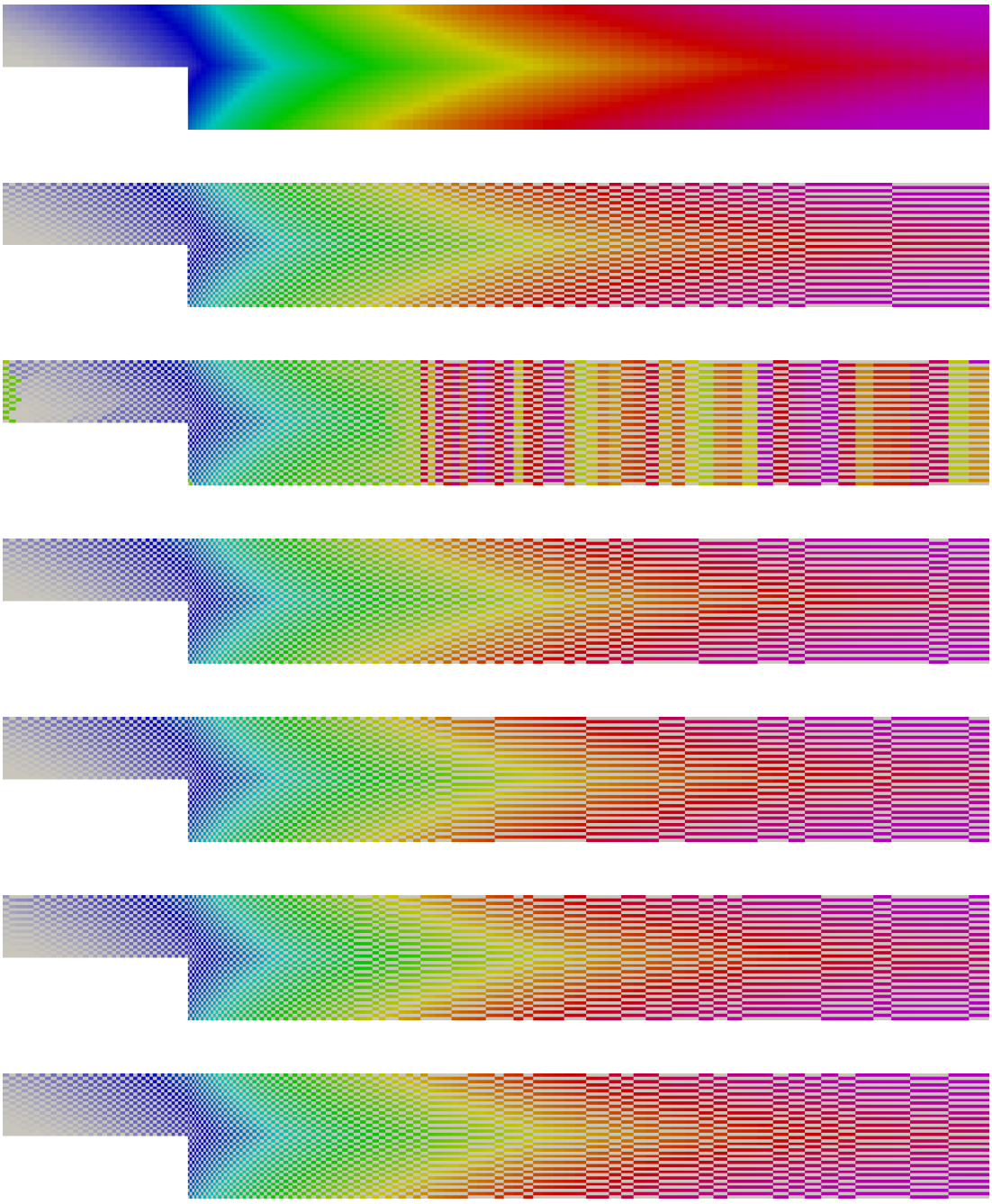


Figure 4.18: First coarse level for 7 cases in Table 4.4, top to bottom image corresponds to ascending ID. Colours of the rainbow denote the order of selection or cluster formation, blue first to magenta last, grey cells are eliminated as fine.

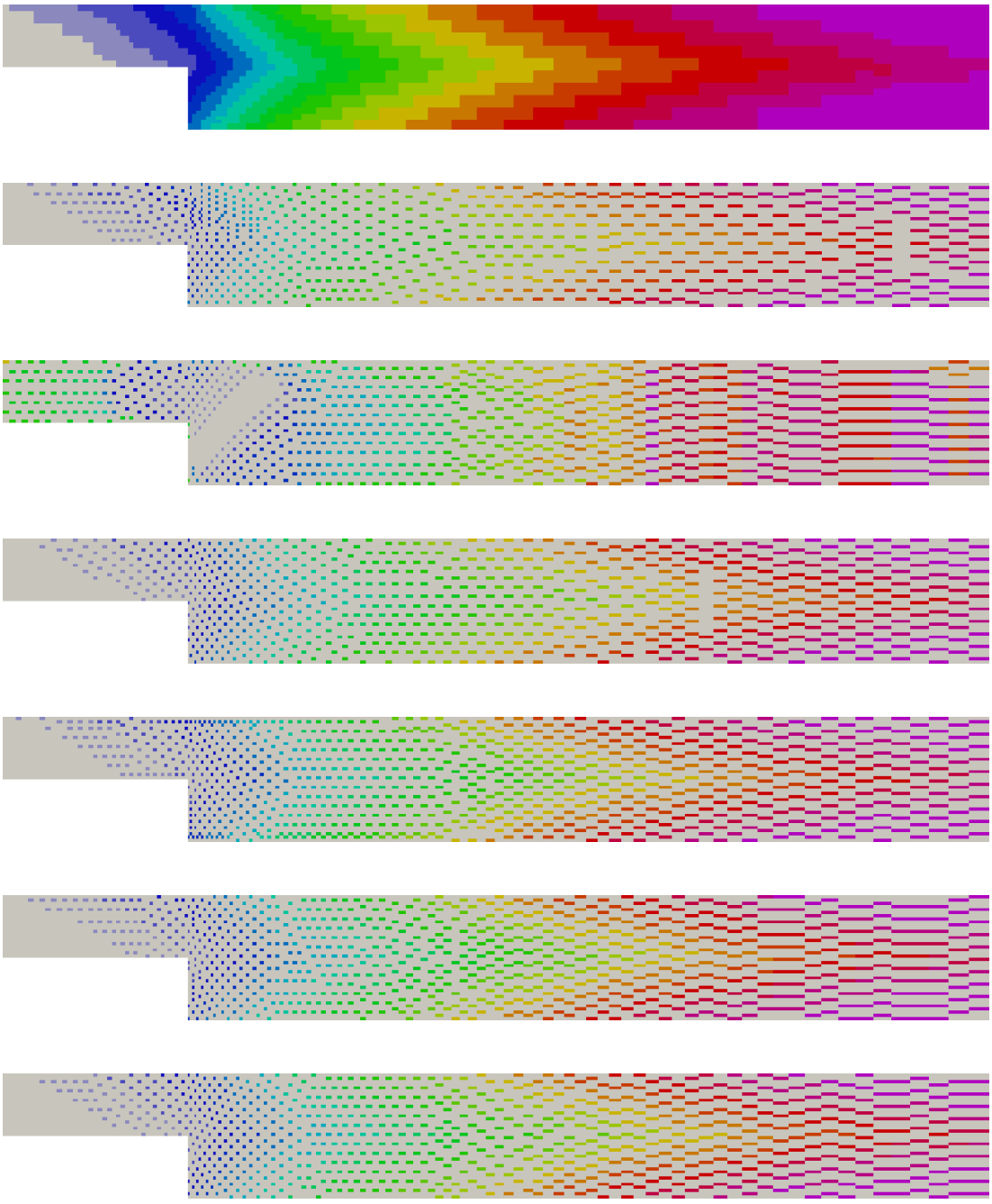


Figure 4.19: Second coarse level for 7 cases in Table 4.4, top to bottom image corresponds to ascending ID. Colours of the rainbow denote the order of selection or cluster formation, blue first to magenta last, grey cells are eliminated as fine.

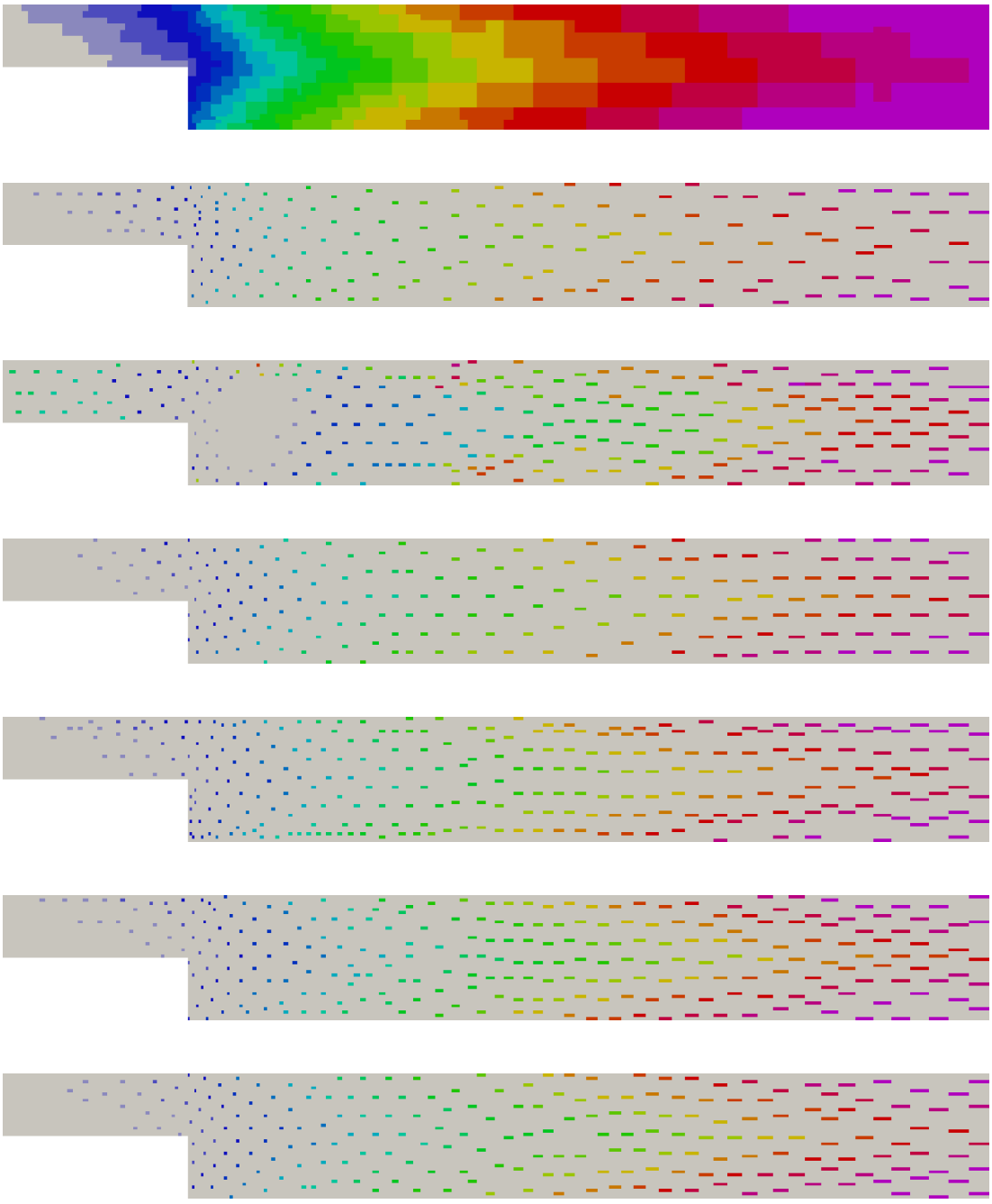


Figure 4.20: Third coarse level for 7 cases in Table 4.4, top to bottom image corresponds to ascending ID. Colours of the rainbow denote the order of selection or cluster formation, blue first to magenta last, grey cells are eliminated as fine.

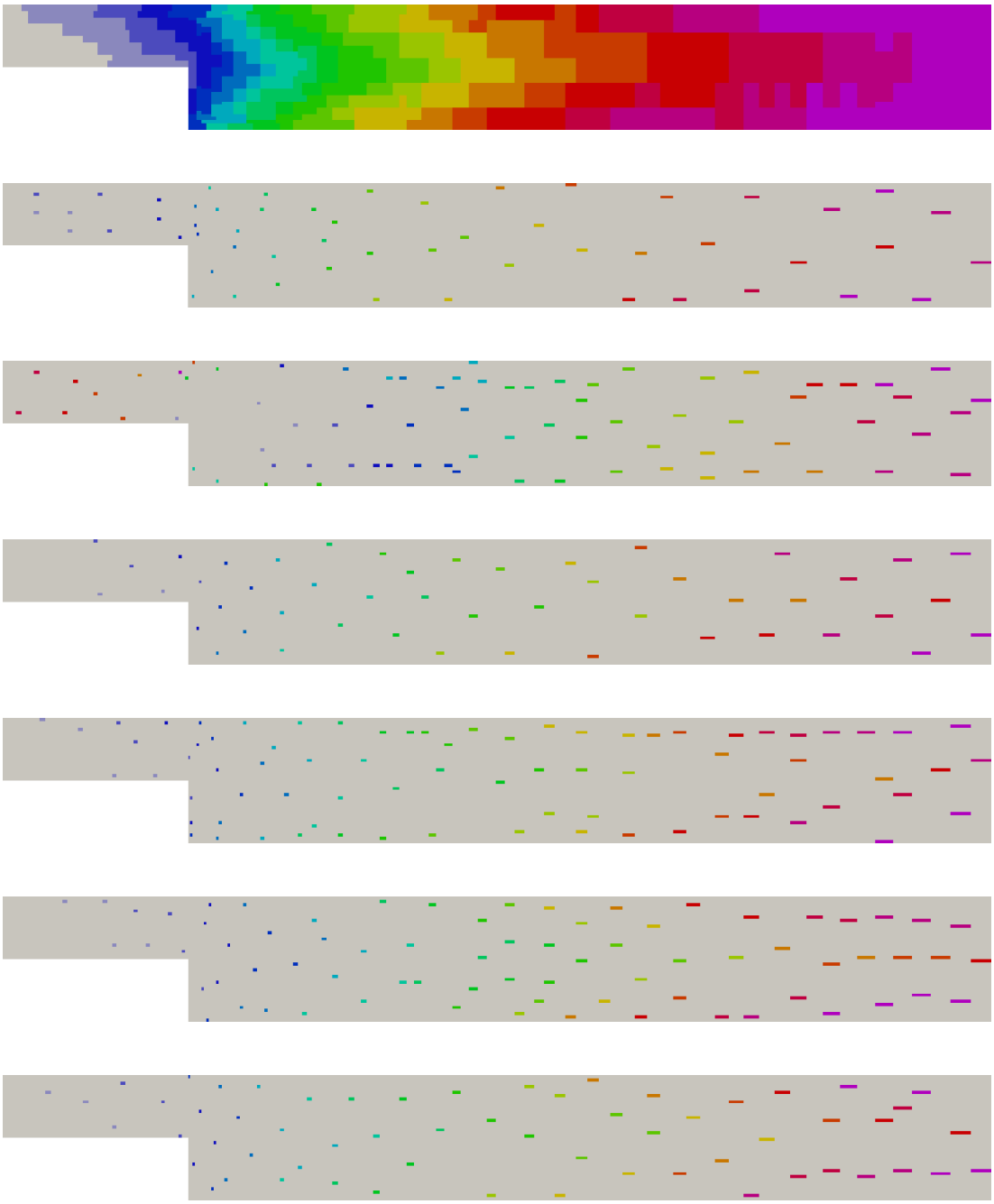


Figure 4.21: Fourth coarse level for 7 cases in Table 4.4, top to bottom image corresponds to ascending ID. Colours of the rainbow denote the order of selection or cluster formation, blue first to magenta last, grey cells are eliminated as fine.

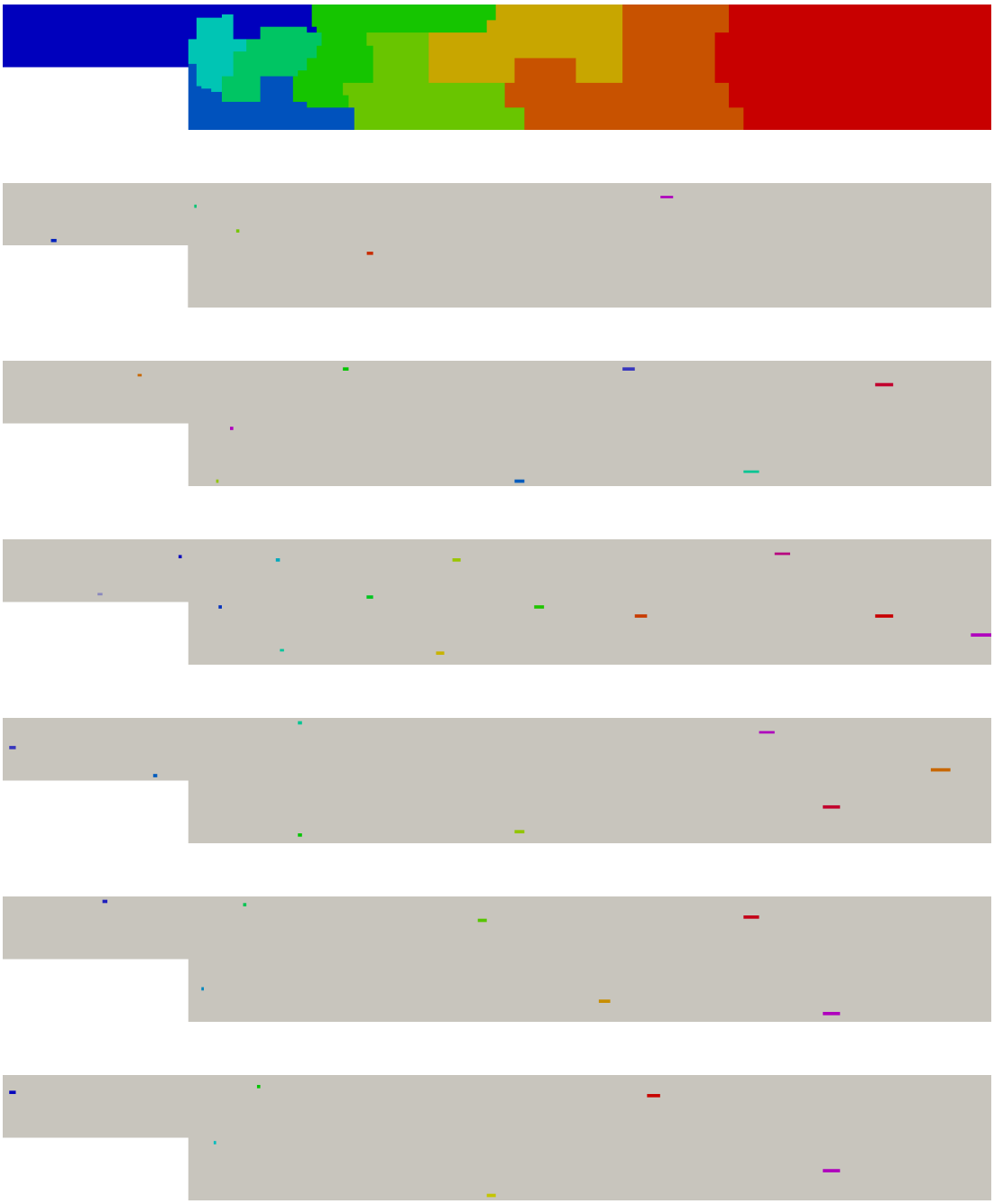


Figure 4.22: Last coarse level for 7 cases in Table 4.4, top to bottom image corresponds to ascending ID. Colours of the rainbow denote the order of selection or cluster formation, blue first to magenta last, grey cells are eliminated as fine.

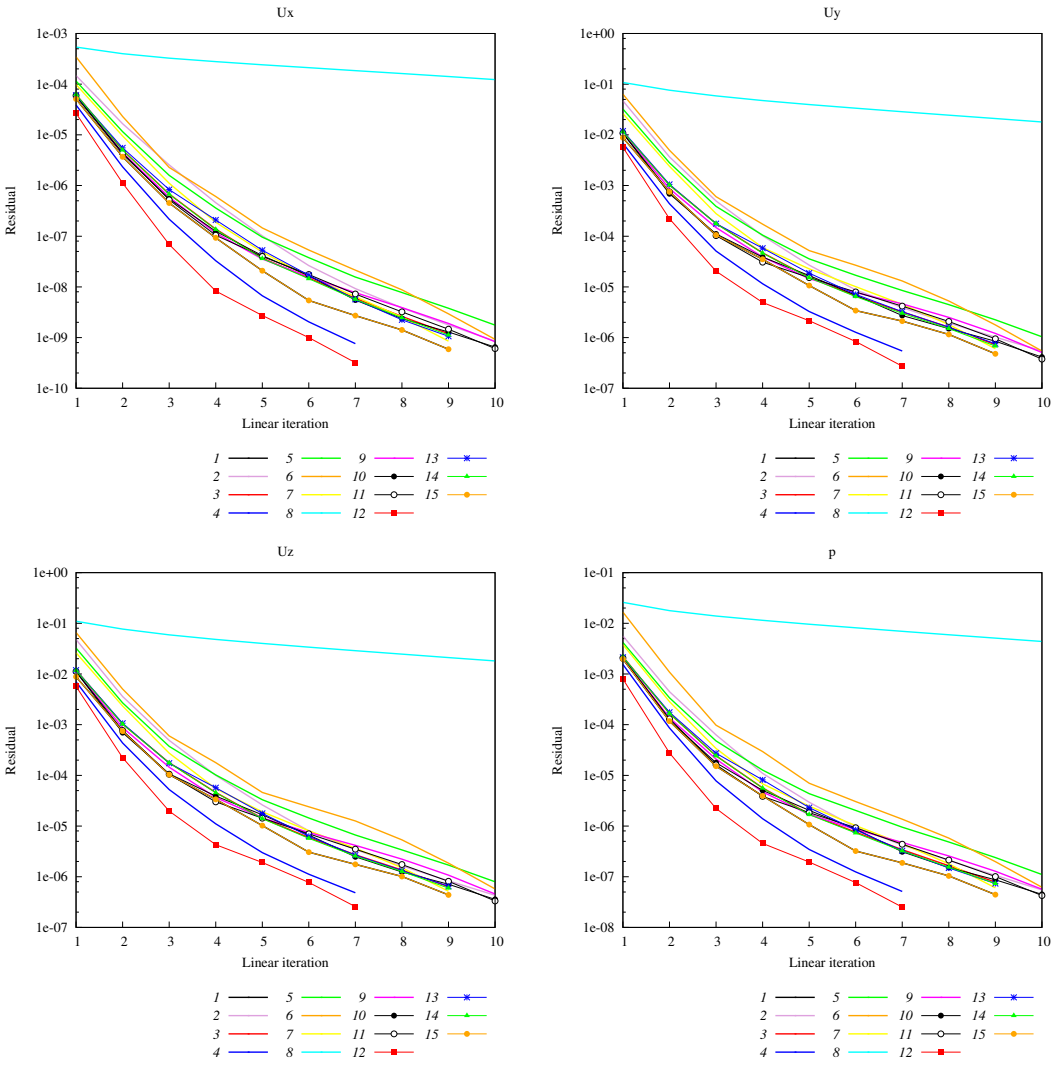


Figure 4.23: Generic submarine: convergence of linear solver residuals for cases with corresponding settings presented in Table 4.6, non–linear iteration 3.

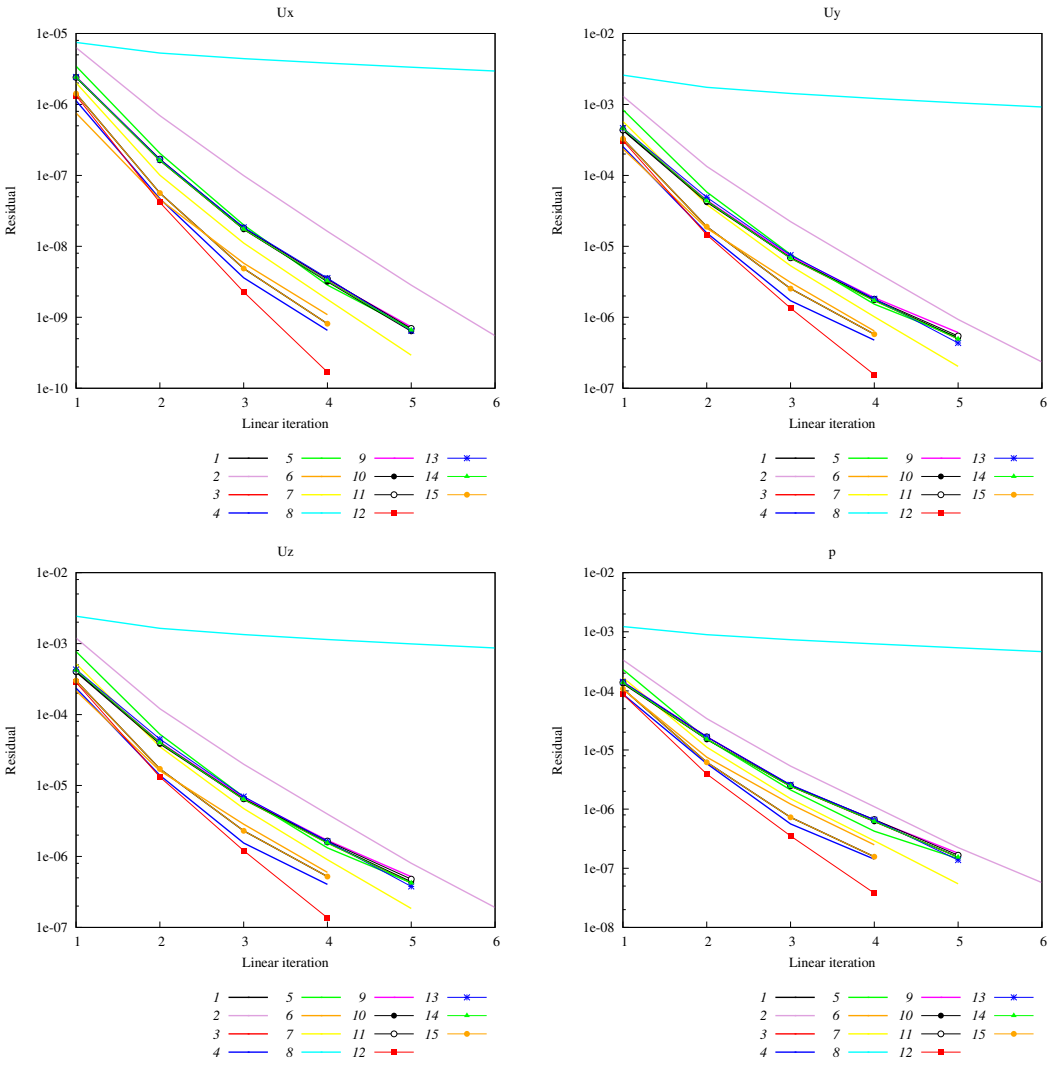


Figure 4.24: Generic submarine: convergence of linear solver residuals for cases with corresponding settings presented in Table 4.7, non–linear iteration 10.

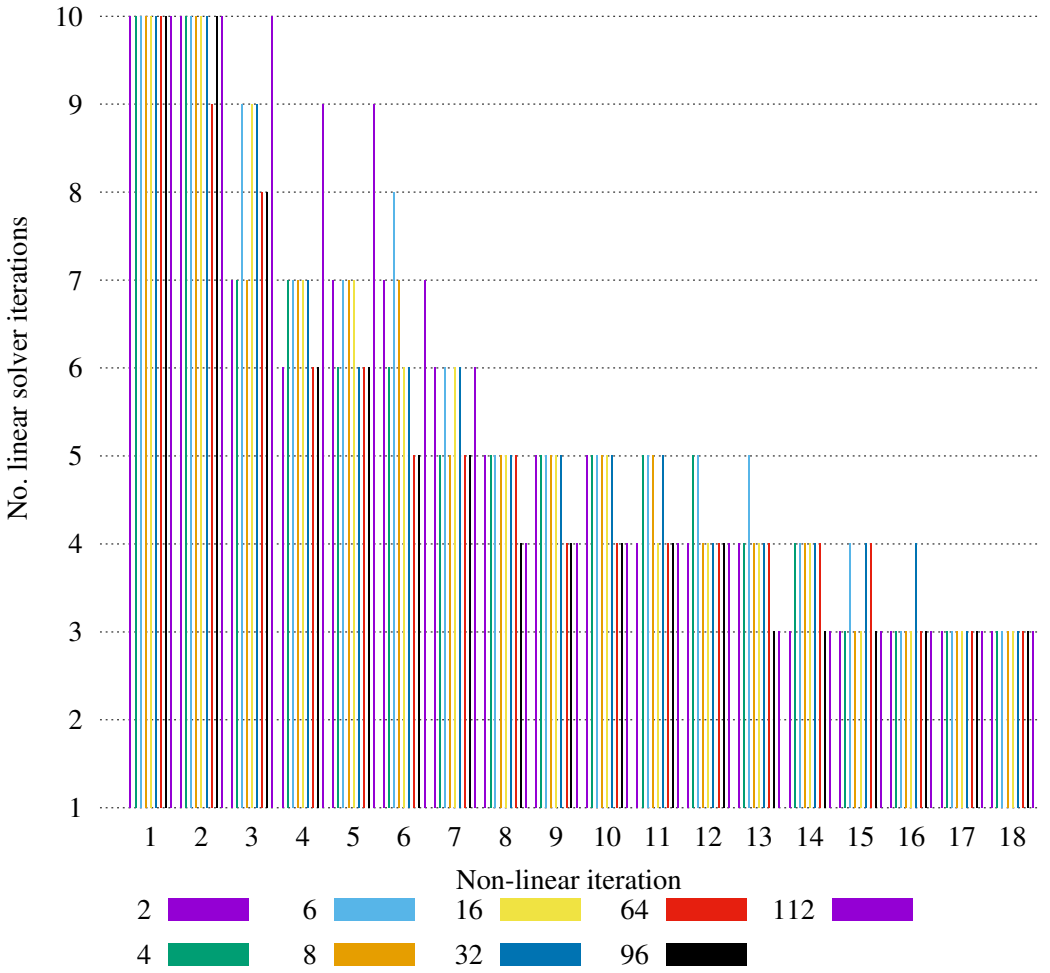


Figure 4.25: BB2 submarine: number of linear iterations per non–linear iteration depending on the number of CPU cores.

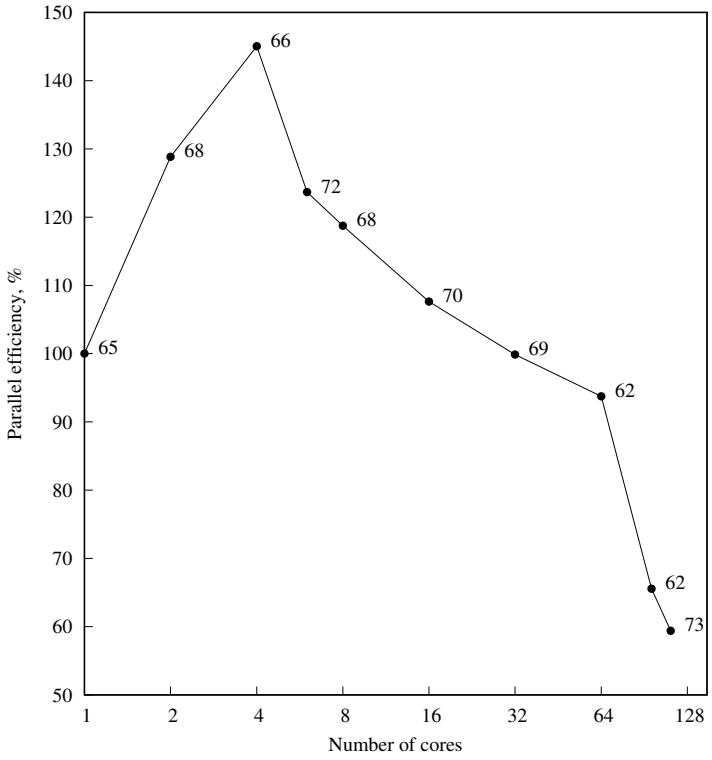


Figure 4.26: BB2 submarine: parallel efficiency calculated for the 10^{th} non–linear iteration of the implicitly coupled pressure–velocity solver. The total number of linear iterations for 10 non–linear iterations is annotated for each number of processors.

4.4. Closure

In this chapter we presented the results of several test cases with steady–state, single–phase, incompressible and turbulent flow: backward–facing step, cooling of an engine jacket, BB2 submarine, centrifugal pump, Francis turbine, generic submarine, front wing and bluff body. We selected each case to demonstrate certain observations and conclusions about the behaviour and performance of the implicitly coupled pressure–velocity solver and block–selective algebraic multigrid algorithm. The most important pieces of information derived from the study are listed in the following chapter.

5. Conclusions and Future Work

The focus of this thesis are the implicitly coupled pressure–velocity system for steady–state, incompressible, single–phase, turbulent flow and the choice of the corresponding algorithm for the solution of the linear system. Based on findings from numerous simulations of complex flows using the implicitly coupled pressure–velocity algorithm, we have decided to implement a parallel selective algebraic multigrid algorithm to improve the convergence, stability and robustness of the solution procedure.

Conclusions

Findings regarding the implicitly coupled pressure–velocity solver presented in chapter 4. can be summarised as follows:

- Implicitly coupled pressure–velocity solver always converges in fewer non–linear iterations than the segregated counterpart, due to implicit treatment of cross–coupling terms (pressure gradient, velocity divergence) and consequently no need for underrelaxation of the pressure field. However, the benefit when comparing the overall CPU time depends on the complexity of the case: coupled solver has a clear advantage for more complex flows.

- A disadvantage of implicit coupling of the pressure–velocity system is the dimension of the coefficient matrix which imposes signficant memory requirements in comparison to the segregated algorithm. This limits the applicability of the solver, since the calculation slows down if there is not enough fast (cache, or even RAM) memory available.

- The second disadvantage of implicit coupling is the choice of the optimal linear solver. For segregated systems, it is possible to select specific linear algorithms adapted for a certain type of problem (e.g. convection dominated or diffusion dominated problems). For coupled system, a more advanced (complex) approach is necessary, leading to a more expensive

procedure regarding the number of operations. We have showed that the optimal choice is using a block–selective multigrid algorithm with an ILU smoother based on Crout's algorithm, supported by a biconjugate gradient stabilised algorithm as the fine level solver. The multigrid part accelerates the convergence of (elliptic) pressure equation, while BiCGStab deals with the (unsymmetric) momentum equation. All disadvantages contribute to a longer execution time per non–linear iteration in comparison to segregated solvers. Thus, the convergence of the implicitly coupled pressure–velocity solver must be superior in terms of the number of non–linear iterations until convergence, to be competitive.

- Convergence of the residual in the implicitly coupled solver is more stable compared to the segregated solver, especially regarding the observed integral values for specific cases, which highly depend on the values of the pressure field. Various intergal values (lift and drag coefficients, head and efficiency in turbomachinery) tend to heavily oscillate with very large amplitudes at the beginning of simulations with the segregated solver. At some point the amplitude of the oscillations reduces and achieves a constant frequency. Integral values with coupled solver oscillate at the beginning, but with much smaller amplitudes and frequencies and for true steady state cases they settle at a constant value without any additional oscillation.

- The density (number of finite volume cells) of the computational mesh does not, or only slightly affects the convergence of the implicitly coupled solver, regarding the number of iterations until meeting the convergence criterion. Convergence of the segregated solver is greatly influenced by the changes in the mesh density.

- Flows which have additional physical effects, such as rotating or porous zones, greatly benefit in terms of stability and convergence if those terms are treated implicitly after discretisation, and their contribution is included in the coefficient matrix.

Findings regarding the block–selection algebraic multigrid (SAMG) can be summarised into the following statements and guidelines:

- Applying SAMG for the solution of the implicitly coupled block system provides superior convergence rates in comparison to additive correction multigrid (AAMG) or preconditioned biconjugate gradient stabilised solver. However, SAMG is more expensive in terms of CPU time per iteration compared to BiCGStab.

- SAMG often achieves theoretical rate of convergence (reduction of residual one order of magnitude per iteration). However, at the beginning of the simulation, when the error is still large, there is always a greater number of linear iterations per non–linear iteration than towards the end of the simulation when the solution is already stabilised. Also, implicit underrelaxation of the momentum equation aids (ensures) linear convergence. For some cases, the momentum equation can be underrelaxed only for the first couple of non–linear iterations. When the oscillation of the solution reduces, the underrelaxation factor can be increased.

- Using the 2–norm of the block–coefficients to create a primary matrix for coarsening does not give good convergence, mainly due to nonoptimal interpolation weights. The best convergence is achieved using the pressure part of the continutiy equation as a primary matrix.

- The number of coarse levels created during the setup phase of SAMG as well as the multigrid cycle, affect not only the convergence rate but the overall execution time as well. Prescribing a small number of coarsest level equations implies a direct solution on the coarsest level, however it increases the computational cost of a single linear iteration. W–cycle provides a better convergence rate compared to V–cycle, but it does not have crucial effect on overall convergence to justify the increase of CPU time.

- Since the conventional smoothers (Jacobi, Gauss–Seidel) did not converge for any of our cases run with coupled solver, we opted for the incomplete lower–upper factorisation with no fill–in based on Crout's algorithm (ILUC0). The convergence of ILUC0 depends on the structure of fine and coarse level coefficient matrices, i.e. it is important to preserve the band (structure) of the matrix to avoid ignoring large factors which can be omit-

ted due to the lowest level of fill–in. Some authors state that smoothing the equations in decreasing order of strength (i.e. smoothing the equations in order of selection) may improve convergence, which proved to be true for some cases, but at a risk that the solver won't converge at all due to divergence of the ILU smoother.

- Regarding the number of smoothing sweeps: we found that using a total of 4 sweeps per level provides the best convergence with respect to computational costs. Increasing the number of post–sweeps can improve convergence, thus the recommended settings are 1 pre–sweep and 3 post–sweeps, or 2 pre–sweeps and 2 post–sweeps.

- Computational mesh, i.e. the type of cells and cell aspect ratio affects the convergence of SAMG. Since off–diagonal matrix coefficients are proportional to the face surface area, the shape of the cells affects the strength of connection in the coarsening process. Changing the strength of connection criterion may improve convergence for meshes which are dominantly anisotropic in a certain direction, but this is not a trivial and straightforward procedure.

- Our parallelisation of SAMG where interpolation is not allowed across processor boundaries does not diminish linear or overall convergence in any of our test cases.

Future work

There are several directions for additional improvement and extension of the implicitly coupled pressure–velocity system:

- further investigate the parallel performance of the entire solution procedure on high performance computers with large number of cores,

- investigate alternative approximations of the convection–diffusion matrix to be used as a diffusion coefficient in the pressure Poisson equation,

- implement implicit boundary conditions (e.g. total pressure boundary condition),

- extend the solver for compressible flows, for which the implicitness of physical boundary conditions is necessary,
- consider load balancing during parallel simulations with SAMG for the solution of coarse levels,
- implement *FLEX* (F) multigrid cycle which could improve convergence of linear solver similar to W–cycle, but with a reduced computational cost: F–cycle is a self–controlling cycle, i.e. it switches between fine and coarse levels according to some convergence criterion,
- implement ILU smoother with pivoting to counteract the different magnitudes of matrix coefficients and prevent divergence.

Appendices

A Mesh Statistics and Images, Boundary Conditions, Flow Field Images

Table A1: Backwardfacing step: mesh statistics and boundary conditions.

BACKWARD FACING STEP						
Mesh data		**Boundary conditions**				
No. cells	4800	**Boundary**	**u**	p	k	ϵ
Cell type	hexahedral (structured)	*inlet*	Dirichlet (1 0 0)	von Neumann (0)	Dirichlet (0.00135)	Dirichlet (0.0001)
Average non–orthogonality	0	*outlet*	von Neumann (0)	Dirichlet (0)	von Neumann (0)	von Neumann (0)
Maximal non–orthogonality	0	*top wall, bottom wall*	Dirichlet (0 0 0)	von Neumann (0)	wall function	wall function
Maximal skewness	0.0002					

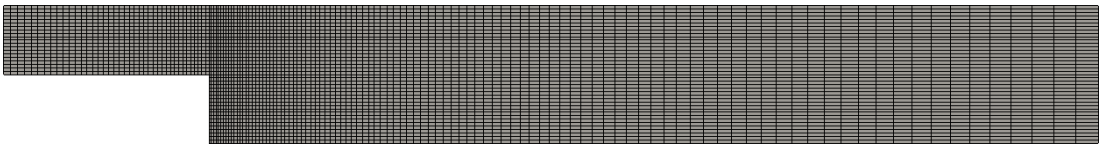


Figure A1: Backward facing step: two–dimensional finite volume mesh.

Table A2: Centrifugal pump: mesh statistics and boundary conditions.

CENTRIFUGAL PUMP						
Mesh data		**Boundary conditions**				
No. cells	9 054 517	**Boundary**	**u**	p	k	ω
Cell type	unstructured: 59% hexahedral, 41% tetrahedral	*inlet*	Dirichlet (-4.78 0 0)	von Neumann (0)	Dirichlet (0.026)	Dirichlet (16.62)
Average non–orthogonality	39.95	*outlet*	von Neumann (0)	Dirichlet (0)	von Neumann (0)	von Neumann (0)
Maximal non–orthogonality	88.13	*casing*	Dirichlet (0 0 -60)	von Neumann (0)	wall function	wall function
Maximal skewness	1.61	*rotor*	Dirichlet $\Omega = 136.14\text{s}^{-1}$	von Neumann (0)	wall function	wall function

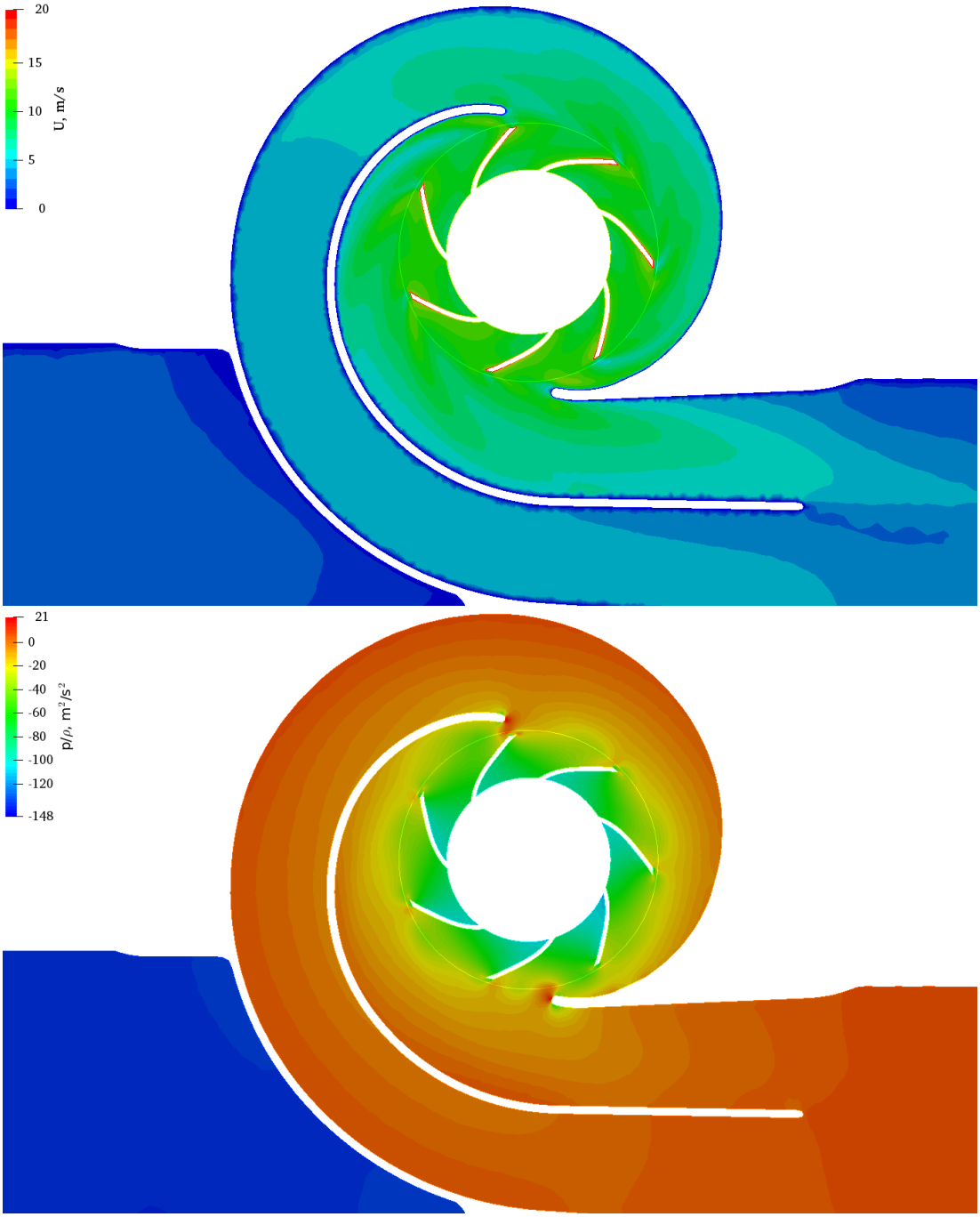


Figure A2: Centrifugal pump: velocity and pressure field on a slice through the impeller.

Table A3: Generic submarine: mesh statistics and boundary conditions.

GENERIC SUBMARINE						
Mesh data		**Boundary conditions**				
No. cells	1 939 796	**Boundary**	**u**	p	k	ω
Cell type	unstructured: 95% hexahedral, 5% polyhedral	*inlet*	Dirichlet (35 0 0)	von Neumann (0)	Dirichlet (0.18)	Dirichlet (180)
Average non–orthogonality	5.66	*outlet*	von Neumann (0)	Dirichlet (0)	von Neumann (0)	von Neumann (0)
Maximal non–orthogonality	61.51	*top wall, bottom wall, side walls*	von Neumann (0)	von Neumann (0)	von Neumann (0)	von Neumann (0)
Maximal skewness	7.65	*submarine hull*	Dirichlet (0 0 0)	von Neumann (0)	wall function	wall function

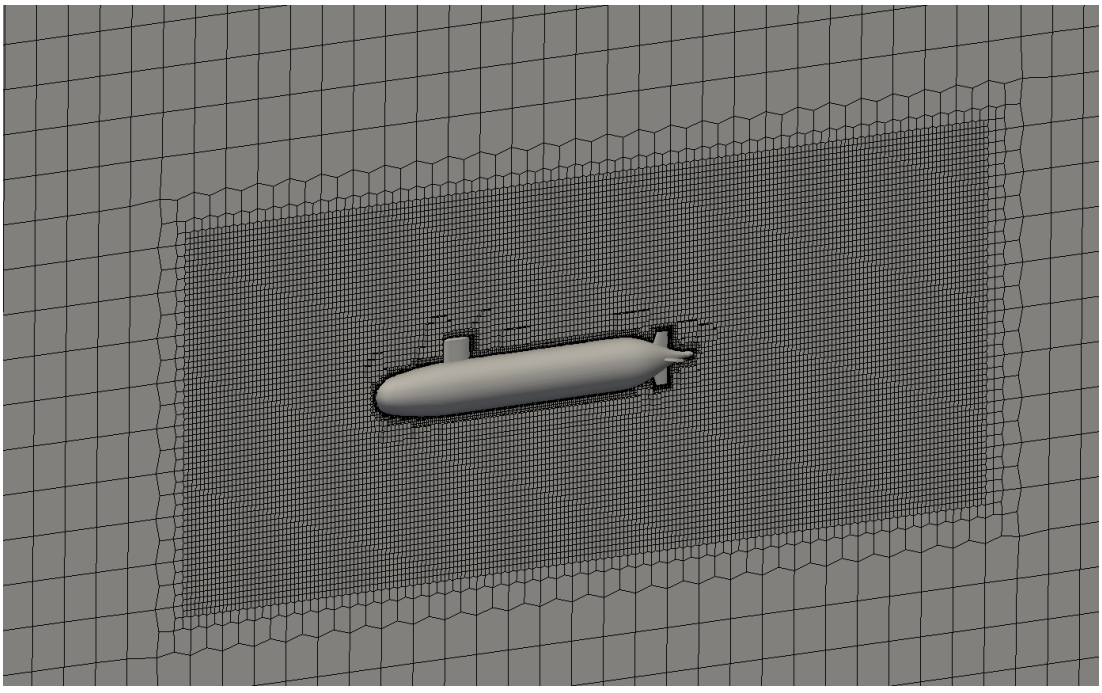


Figure A3: Generic submarine: a slice through the finite volume mesh.

Table A4: Engine cooling: mesh statistics and boundary conditions.

ENGINE COOLING						
Mesh data		**Boundary conditions**				
No. cells	156 739	**Boundary**	**u**	p	k	ϵ
Cell type	unstructured: 99.9% hexahedral, 0.1% prisms	*inlet*	Dirichlet (0 10 0)	von Neumann (0)	Dirichlet (0.0375)	Dirichlet (14.86)
Average non–orthogonality	22.28	*outlet*	von Neumann (0)	Dirichlet (0)	von Neumann (0)	von Neumann (0)
Maximal non–orthogonality	82.19	*casing*	Dirichlet (0 0 0)	von Neumann (0)	wall function	wall function
Maximal skewness	6.60					

Figure A4: Engine cooling: finite volume mesh, inlet surface is purple, outlet is yellow.

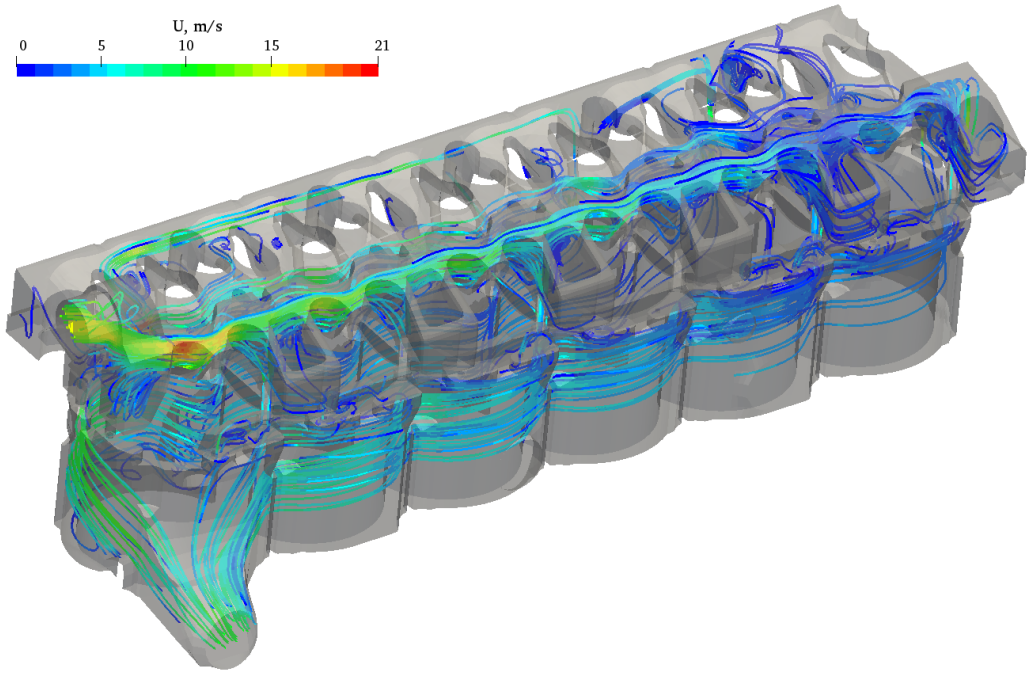


Figure A5: Engine cooling: streamlines couloured by the values of velocity.

Table A5: Francis turbine: mesh statistics and boundary conditions.

FRANCIS TURBINE						
Mesh data		**Boundary conditions**				
No. cells	6 242 679	**Boundary**	**u**	p	k	ω
Cell type	hexahedral (structured)	*inlet*	Dirichlet (-1.41 -2.12 0)	von Neumann (0)	Dirichlet (0.01)	Dirichlet (25.54)
Average non–orthogonality	25.59	*outlet*	von Neumann (0)	Dirichlet (0)	von Neumann (0)	von Neumann (0)
Maximal non–orthogonality	67.95	*casing*	Dirichlet (0 0 0)	von Neumann (0)	wall function	wall function
Maximal skewness	3.46	*impeller*	Dirichlet $\omega = 34.82\text{s}^{-1}$	von Neumann (0)	wall function	wall function
		stator	Dirichlet (0 0 0)	von Neumann (0)	wall function	wall function

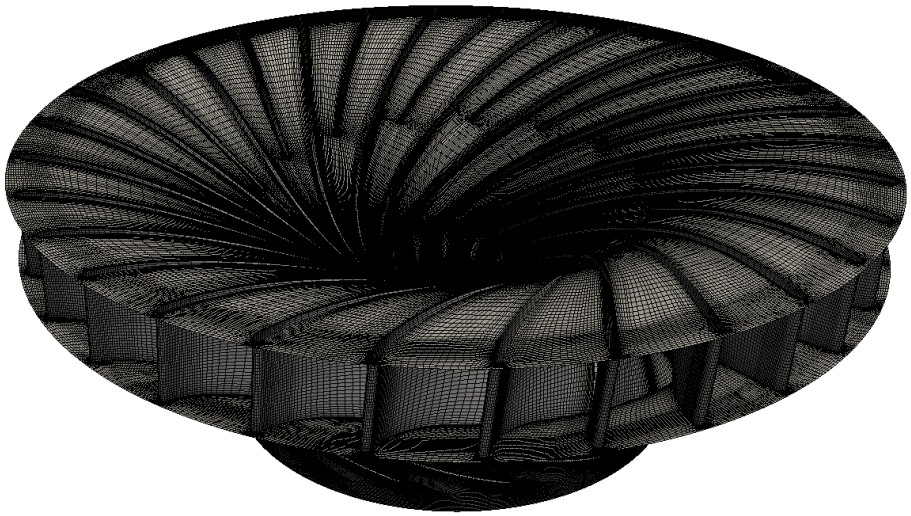


Figure A6: Francis turbine: impeller surface mesh.

Table A6: Front wing: mesh statistics and boundary conditions.

FRONT WING						
Mesh data		**Boundary conditions**				
No. cells	6 575 126	**Boundary**	**u**	p	k	ω
Cell type	unstructured: 97% hexahedral, 3% polyhedral	*inlet*	Dirichlet (0 0 -60)	von Neumann (0)	Dirichlet (0.135)	Dirichlet (100)
Average non–orthogonality	3.58	*outlet*	von Neumann (0)	Dirichlet (0)	von Neumann (0)	von Neumann (0)
Maximal non–orthogonality	68.74	*ground*	Dirichlet (0 0 -60)	von Neumann (0)	wall function	wall function
Maximal skewness	4.88	*top wall, side walls*	Robin (symmetry)	von Neumann (0)	Robin (symmetry)	Robin (symmetry)
		front wing surface	Dirichlet (0 0 0)	von Neumann (0)	wall function	wall function

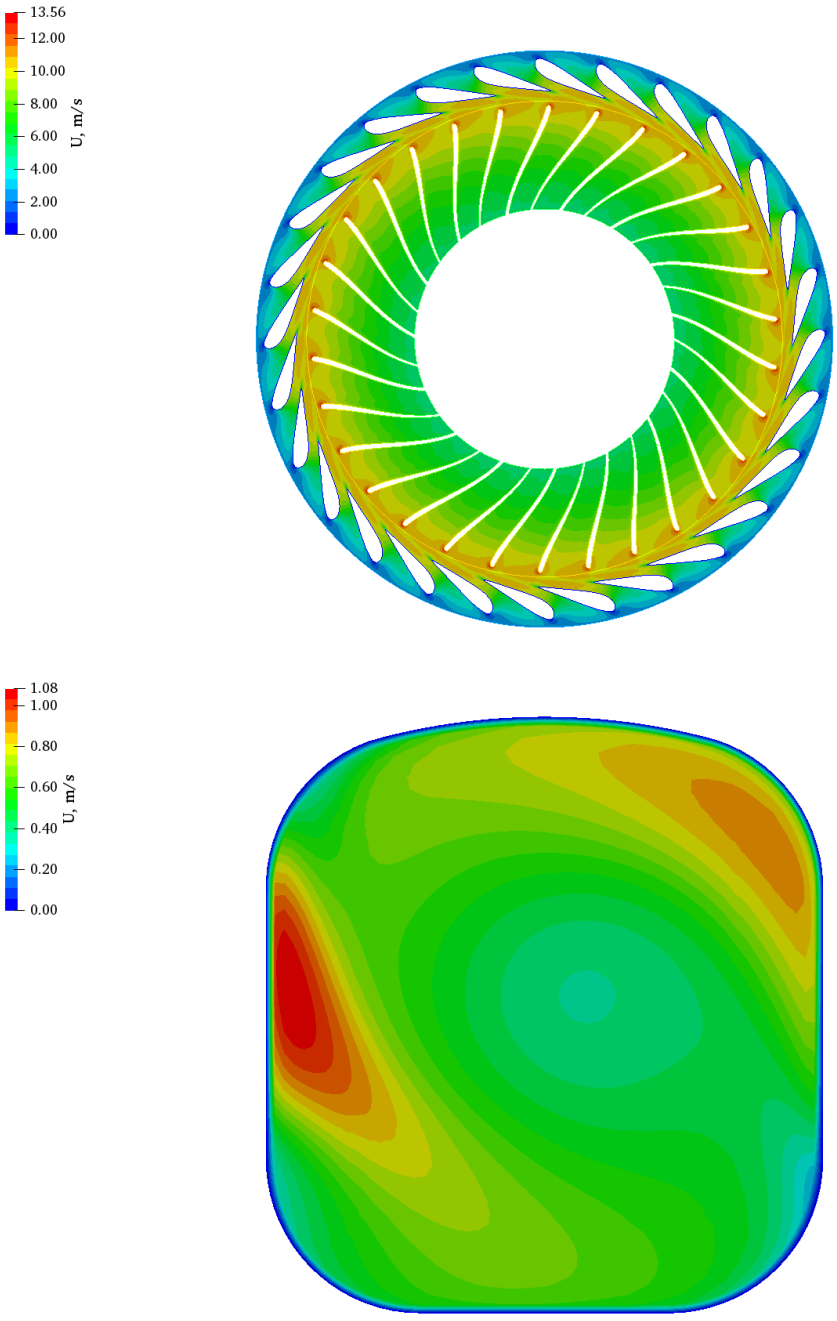


Figure A7: Francis turbine: slice showing the velocity field arount stay vanes and impeller (top), velocity field at the diffuser outlet (bottom).

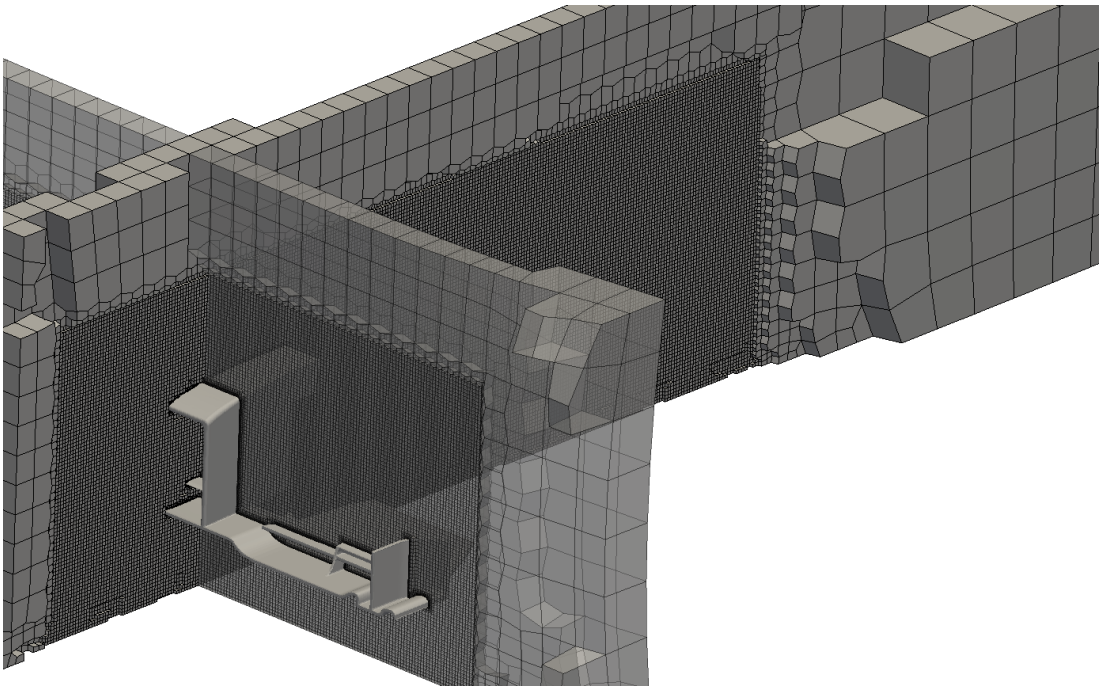


Figure A8: Front wing: crinkled slices through the finite volume mesh.

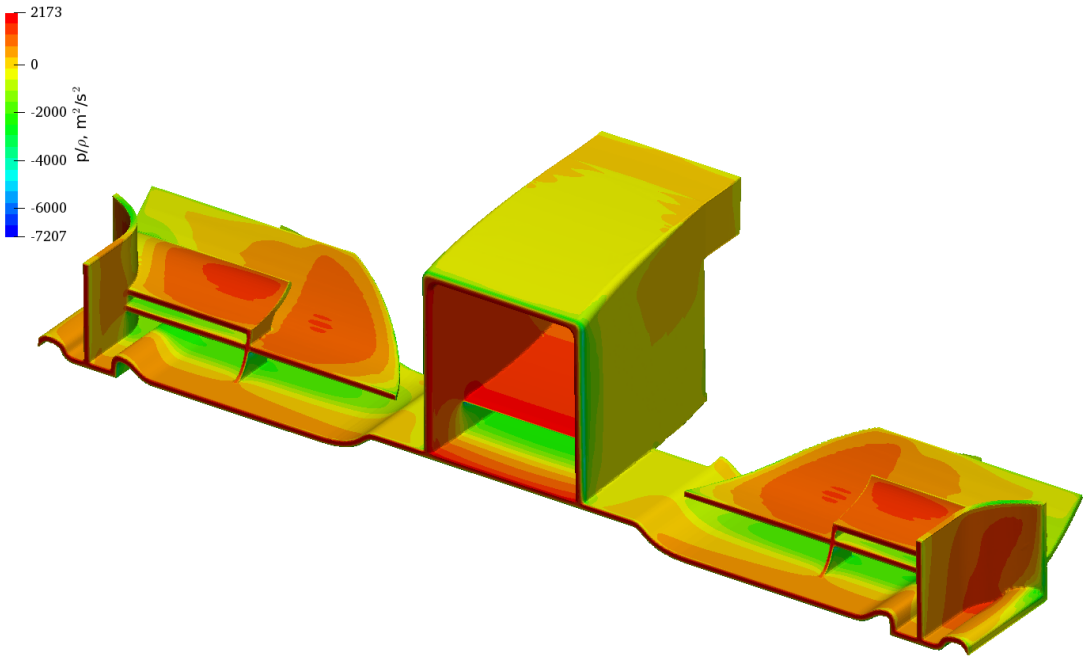


Figure A9: Front wing: pressure on the surface of the wing.

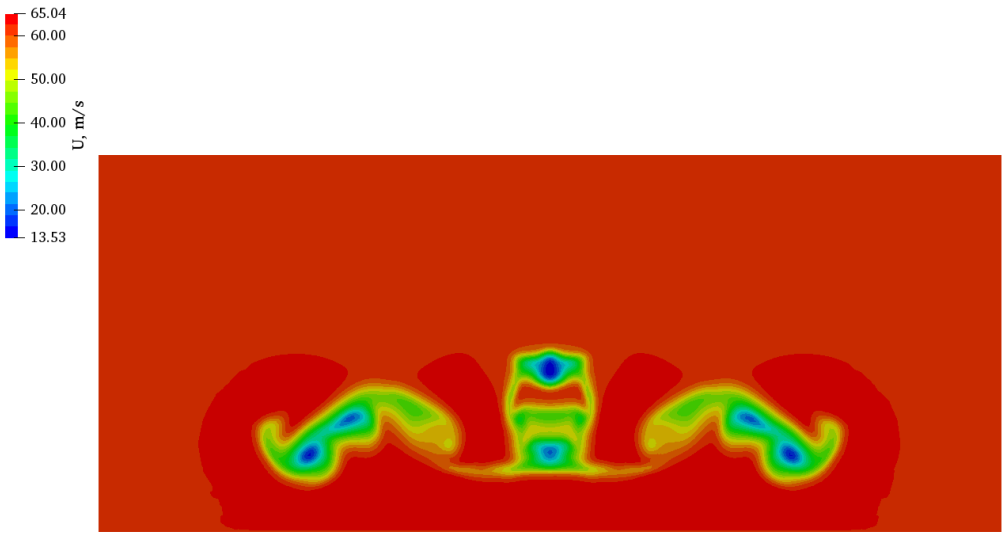


Figure A10: Front wing: vortices in the wake of the wing coloured by the values of velocity.

Table A7: Bluff body: mesh statistics and boundary conditions.

BLUFF BODY						
Mesh data		**Boundary conditions**				
No. cells	9 269 072	**Boundary**	**u**	p	k	ω
Cell type	unstructured: 99% hexahedral, 1% polyhedral	*inlet*	Dirichlet (20 0 0)	von Neumann (0)	Dirichlet (0.0024)	Dirichlet (13.33)
Average non–orthogonality	1.56	*outlet*	von Neumann (0)	Dirichlet (0)	von Neumann (0)	von Neumann (0)
Maximal non–orthogonality	41.63	*ground*	Dirichlet (20 0 0)	von Neumann (0)	wall function	wall function
Maximal skewness	1.50	*top wall, front wall, back wall*	Robin (symmetry)	von Neumann (0)	von Neumann (0)	von Neumann (0)
		bluff body surface	Dirichlet (0 0 0)	von Neumann (0)	wall function	wall function

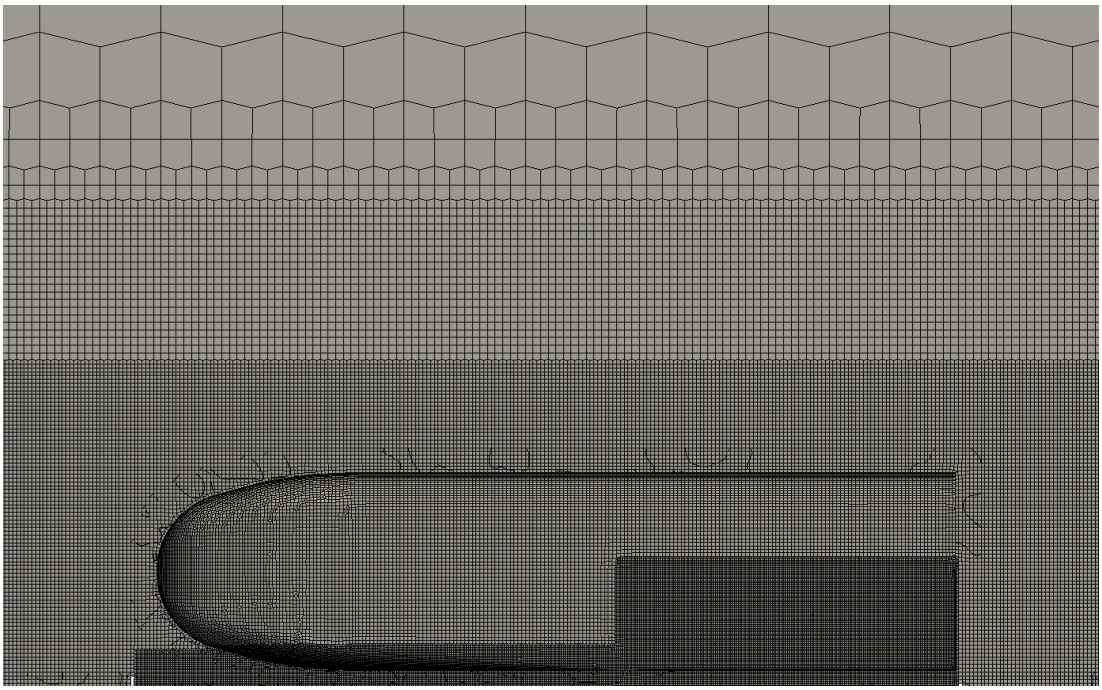


Figure A11: Bluff body: slice through the finite volume mesh.

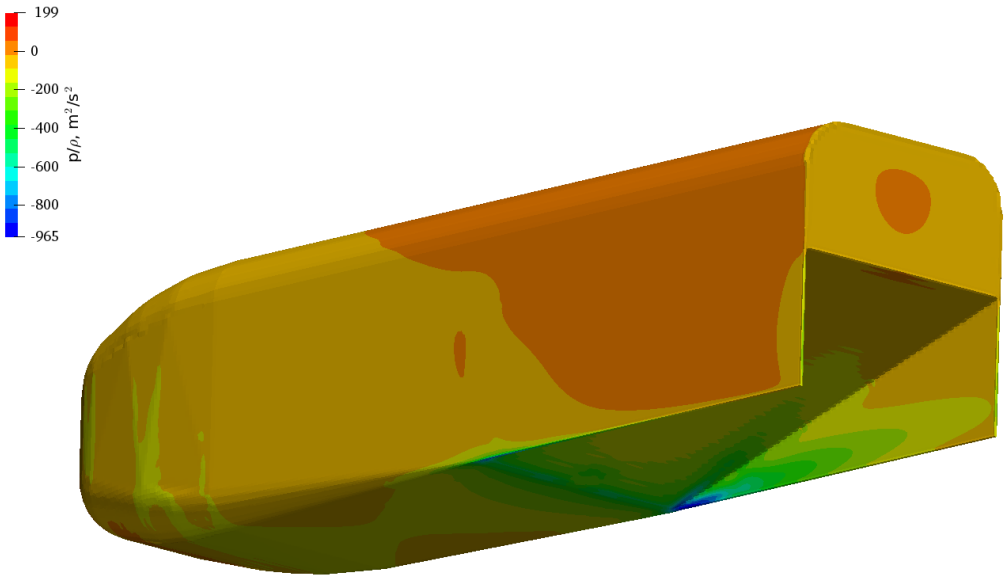


Figure A12: Bluff body: pressure on the surface of the body.

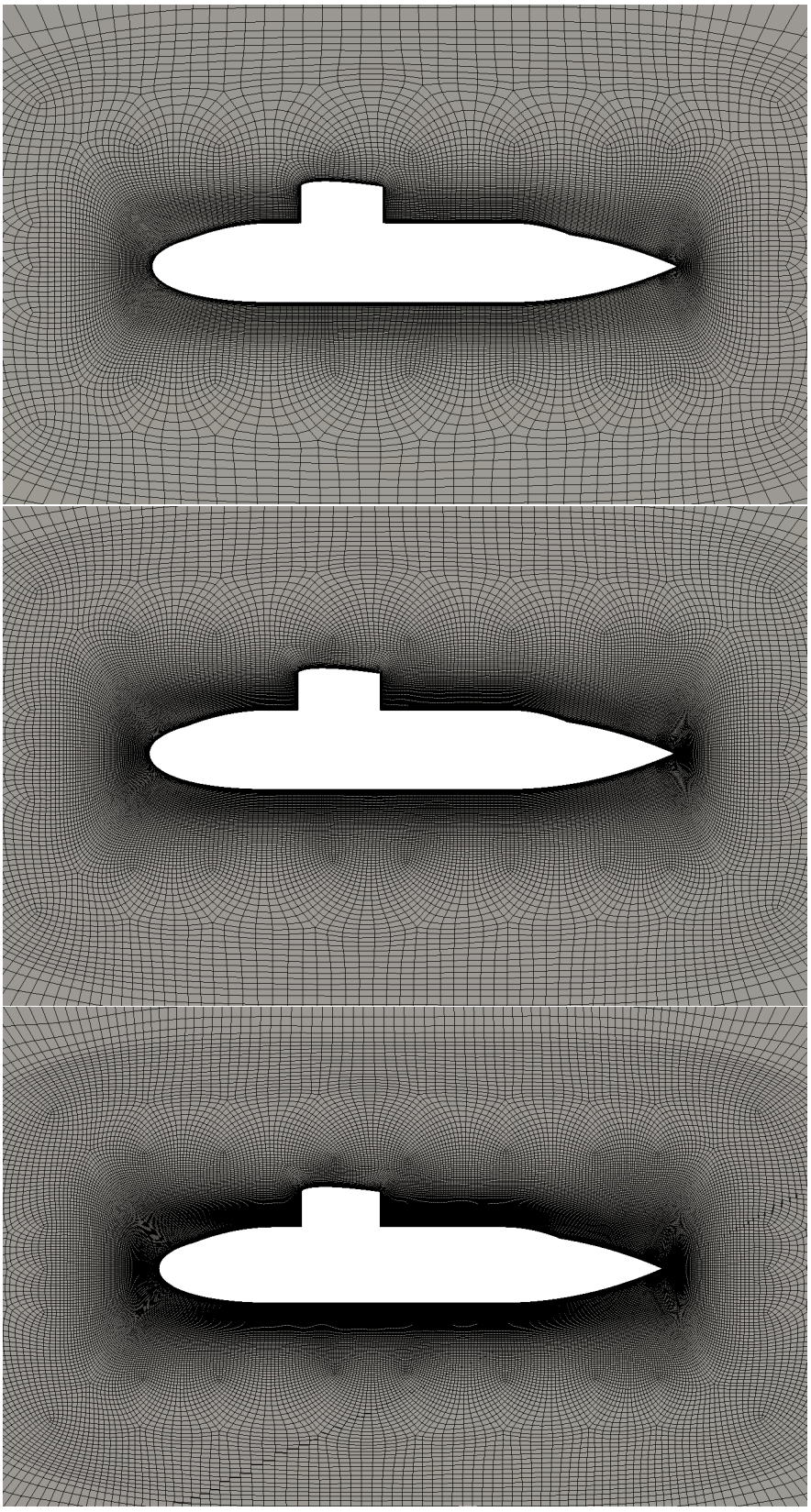


Figure A13: BB2 submarine: slices through the finite volume mesh showing three densities - coarse to fine, from top to bottom.

Bibliography

[1] H. Weller, G. Tabor, H. Jasak, C. Fureby, A tensorial approach to computational continuum mechanics using object-oriented techniques, Computers in Physics 12 (1998) 620–631.

[2] Y. Saad, Iterative methods for sparse linear systems, SIAM, 2000.

[3] S. Patankar, D. Spalding, A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows, International journal of heat and mass transfer 15 (1972) 1787–1806.

[4] S. V. Patankar, Numerical heat transfer and fluid flow, Taylor and Francis, 1980.

[5] J.P. van Doormaal and G.D. Raithby, Enhancements of the simple method for predicting incompressible fluid flows, Numerical heat transfer 7 (1984) 147–163.

[6] R. Issa, Solution of the implicitly discretized fluid flow equations by operator-splitting, Journal of Computational Physics 62 (1986) 40–65.

[7] Z. Mazhar, Fully implicit, coupled procedures in computational fluid dynamics, Springer International Publishing, 2016.

[8] G. Raithby, G. Schneider, Numerical solution of problems in incompressible fluid flow: treatment of the velocity-pressure coupling, Numerical heat transfer 2:4 (1979) 417–440.

[9] Z. Mazhar, G. Raithby, A refined pumpin (pressure update by multiple path integration) method for updating pressures in the numerical solution of the incompressible fluid flow equations, in: Proceedings 2nd international conference on numerical methods in laminar and turbulent flow, Pineridge Press, Swansea, 1981, pp. 255–266.

[10] C. Davies, P. Carpenter, A novel velocity-vorticity formulation of the navier-stokes equations with applications to boundary layer disturbance evolution, Journal of computational physics 172 (2001) 119–165.

[11] S. Krizmanić, Novi algoritam za povezivanje polja brzine i tlaka, PhD Thesis, Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb, 2010.

[12] M. Zedan, G. Schneider, A strongly implicit simultaneous variable solution procedure for velocity and pressure in fluid flow problems, in: AIAA 18th Thermophysics Conference, Montreal, Canada, 1983.

[13] M. Zedan, G. Schneider, A coupled strongly implicit procedure for velocity and pressure computation in fluid flow problems, Numerical heat transfer 8:5 (1985) 537–557.

[14] Z. Mazhar, A procedure for the treatment of the velocity-pressure coupling problem in incompressible fluid flow, Numerical Heat Transfer 39 (2001) 91–100.

[15] Z. Mazhar, An enhancement to the block implicit procedure for the treatment of the velocity-pressure coupling problem in incompressible fluid flow, Numerical Heat Transfer 41 (2002) 493–500.

[16] M. Darwish, I. Sraj, F. Moukalled, A coupled finite volume solver for the solution of incompressible flows on unstructured grids, Journal of Computational Physics 228 (2009) 180–201.

[17] M. Darwish, I. Sraj, F. Moukalled, A coupled incompressible flow solver on structured grids, Numerical heat transfer, Part B: Fundamentals 52 (2007) 353–371.

[18] L. Mangani, M. Buchmayr, M. Darwish, Development of a novel fully coupled solver in OpenFOAM: steady state incompressible turbulent flows in rotational reference frames, Numerical heat transfer, Part B: Fundamentals 66 (2014) 526–543.

[19] M. Darwish, F. Moukalled, A fully coupled Navier-Stokes solver for fluid flow at all speeds, Numerical Heat Transfer Fundamentals 65 (2014) 410–444.

[20] L. Mangani, M. Buchmayr, M. Darwish, Development of a novel fully coupled solver in OpenFOAM: stady–state incompressible turbulent flows, Numerical Heat Transfer Fundamentals 66 (2014) 1–20.

[21] L. Mangani, M. Darwish, F. Moukalled, Development of a pressure-based coupled CFD solver for turbulent and compressible flows in turbomachinery applications, in: ASME TURBO EXPO 2014, Düsseldorf, Germany, 2014.

[22] M. Darwish, A. A. Aziz, F. Moukalled, A coupled pressure–based finite volume solver for incompressible two–phase flow, Numerical heat transfer, Part B: Fundamentals 67 (2015) 47–74.

[23] T. Uroic, H. Jasak, H. Rusche, Implicitly coupled pressure–velocity solver, in: H. J. J. M. Nobrega (Ed.), OpenFOAM: Selected papers of the 11th Workshop, Springer, 2017.

[24] Z. Chen, A. Przekwas, A coupled pressure-based computational method for incompressible/compressible flows, Journal of Computational Physics 229 (2010) 9150–9165.

[25] U. Falk, M. Schäfer, A fully coupled finite volume solver for the solution of incompressible flows on locally refined non–matching block–structured grids, in: VI International Conference on Adaptive Modeling and Simulation ADMOS, Lisbon, Portugal, 2013.

[26] C. Fernandes, V. Vukčević, T. Uroić, R. Simoes, O. Carneiro, H. Jasak, J. Nobrega, A coupled finite volume solver flow solver for the solution of incompressible viscoelastic flows, Journal of non–Newtonian fluid mechanics 265 (2019) 99–115.

[27] U. Trottenberg, C. Oosterlee, A. Schüller, Multigrid, Elsevier, Academic Press, 2001.

[28] J. W. Ruge, K. Stüben, Algebraic multigrid, SIAM - Frontiers in Applied Mathematics: Multigrid methods.

[29] K. Stüben, A review of algebraic multigrid, Journal of Computational and Applied Mathematics 128 (2001) 281–309.

[30] T. Clees, AMG strategies for PDE systems with applications in industrial semiconductor simulation, PhD Thesis, University of Cologne, Germany, 2004.

[31] T. Füllenbach, K. Stüben, Algebraic Multigrid for Selected PDE Systems.

[32] P. Vanek, J. Mandel, M. Brezina, Algebraic multigrid on unstructured meshes, Tech. rep., Denver, CO, USA (1994).

[33] B. Hutchinson, G. Raithby, A multigrid method based on the additive correction strategy, Numerical heat transfer 9 (1986) 511–537.

[34] B. Hutchinson, P. Galpin, G. Raithby, Application of additive correction multigrid to the coupled fluid flow equations, Numerical heat transfer 13 (1988) 133–147.

[35] M. Raw, A coupled algebraic multigrid method for the 3D Navier-Stokes Equations, Fast solvers for flow problems 49 (1995) 204–215.

[36] M. Raw, Robustness of coupled algebraic multigrid for the Navier-Stokes equations, in: AIAA 34th Aerospace Sciences Meeting and Exhibit, Reno, NV, USA, 1996.

[37] E. Chow, R. Falgout, J. Hu, R. Tuminaro, U. Yang, A survey of parallelization techniques for multigrid solvers, in: H. S. M.A. Heroux, P. Raghavan (Ed.), Parallel Processing for Scientific Computing, SIAM Series on Software, Environments, and Tools, 2006.

[38] A. Baker, T. Gamblin, M. Schulz, U. Yang, Challenges of scaling algebraic multigrid across modern multicore architectures, in: 25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS, Anchorage, AK, USA, 2011.

[39] J. Brannick, F. Cao, K. Kahl, R. Falgout, X. Hu, Optimal interpolation and compatible relaxation in classical algebraic multigrid, SIAM Journal on Scientific Computing 40 (2018) 1473–1493.

[40] U. Yang, On long-range interpolation operators for aggressive coarsening, Numerical linear algebra with applications 17 (2010) 453–472.

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

[42] F. Menter, Zonal two equation k-ω turbulence models for aerodynamic flows, in: Proceedings of 24$^{\text{th}}$ Fluid Dynamics Conference, Orlando, USA, 1993.

[43] H. Jasak, Error analysis and estimation for the finite volume method with applications to fluid flows, Ph.D. thesis, Imperial College of Science, Technology & Medicine, London (1996).

[44] C. Rhie, W. Chow, A numerical study of the turbulent flow past an isolated airfoil with trailing edge separation, AIAA Journal 21 (1983) 1525–1532.

[45] J. Ferziger, M. PeriÄ‡ (Eds.), Computational methods for fluid dynamics, Springer, 2002.

[46] Ž. TukoviÄ‡, M. PeriÄ‡, H. Jasak, Consistent second–order time–accurate non–iterative PISO–algorithm, Computers and Fluids 166 (2018) 78–85.

[47] M. Benzi, G. Golub, J. Liesen, Numerical solution of saddle point problems, Acta numerica 14 (2005) 1–137.

[48] H. C. Elman, D. J. Silvester, A. J. Wathen, Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics, Numerical Mathematics and Scientific Computation, Oxford University Press, 2005.

[49] F. Zhang (Ed.), The Schur complement and its applications, Numerical Methods and Algorithms, Springer, 2005.

[50] E. Cuthill, J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: ACM '69 Proceedings of the 1969 24th national conference, New York, NY, USA, 1969.

[51] M. PeriÄ‡, A finite volume method for the prediction of three–dimensional fluid flow in complex ducts, Ph.D. thesis, Imperial College, University of London (1985).

[52] P. Sweby, High resolution schemes using flux limiters for hyperbolic conservation laws, SIAM Journal of Numerical Analysis 21 (1984) 995–1011.

[53] P. Gaskell, A. Lau, Curvature–compensated convective transport: SMART, a new boundedness–preserving transport algorithm, International Journal for Numerical Methods in Fluids 8 (1988) 617–641.

[54] F. Moukalled, L. Mangani, M. Darwish, The Finite Volume Method in Computational Fluid Dynamics, Fluid Mechanics and Its Applications, Springer, 2016.

[55] P. Khosla, S. Rubin, A diagonally dominant second–order accurate implicit scheme, ComputersFluids 2 (1974) 207–209.

[56] H. Jasak, H. Weller, Application of the finite volume method and unstructured meshes to linear elasticity, International journal for numerical methods in engineering 48 (2000) 267–287.

[57] J. Shewchuk, An introduction to the Conjugate Gradient method without the agonizing pain, School of Computer Science, Pittsburgh, 1994.

[58] V. E. Henson, U. M. Yang, BoomerAMG: A parallel algebraic multigrid solver and preconditioner, Applied Numerical Mathematics 41–1 (2002) 155–177.

[59] A. Krechel, K. Stüben, Parallel algebraic multigrid based on subdomain blocking, Parallel Computing 27 (2001) 1009–1031.

[60] A. Bienz, R. Falgout, W. Gropp, L. Olson, J. Schroeder, Reducing parallel communication in algebraic multigrid through sparsification, SIAM Journal on Scientific Computing 38 (2016) 332–357.

[61] H. van der Vorst, Iterative Krylov methods for large linear systems, Cambridge University Press, 2003.

[62] N. Li, Y. Saad, E. Chow, Crout versions of ILU for general sparse matrices, SIAM Journal on Scientific Computing 25 (2003) 716–728.

[63] G. Wittum, Linear iterations as smoothers in multigrid methods: theory with applications to incomplete decompositions, Impact of computing in science and engineering 1 (1989) 180–215.

[64] Y. Saad, M. Schultz, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM Journal on Scientific and Statistical Computing 7 (1986) 856–869.

[65] R. Fletcher, Conjugate gradient methods for indefinite systems, Numerical analysis (1976) 73–89.

[66] G. Strang, Lecture in Mathematical Methods for Engineers II (Spring 2006).

[67] N. Bosner, Iterative methods for solving linear systems (in Croatian), Department of Mathematics, University of Zagreb, 2001.

[68] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. V. der Vorst, Templates for the solution of linear systems: building blocks for iterative methods, SIAM, 1994.

[69] P. Sonneveld, CGS, a fast Lanczos type solver for nonsymmetric linear systems, SIAM Journal on Scientific and Statistical Computing 10 (1989) 36–52.

[70] Y. Vinay, S. Mojdeh, L. Ash, Algebraic multigrid on unstructured meshes, Tech. rep., Ontario, Canada (2016).

[71] H. van der Vorst, Bi-CGStab: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, SIAM Journal on Scientific and Statistical Computing 13 (1992) 631–644.

[72] J. Fackrell, The aerodynamics of an isolated wheel rotating in contact with the ground, Ph.D. thesis, Faculty of Engineering, University of London (1974).

[73] W. Toet, How do motorsport diffusers work? (2017).
URL `https://www.motorsport.com/`

[74] A. Senior, The aerodynamics of a diffuser equipped bluff body in ground effect, Ph.D. thesis, School of Engineering Sciences, University of Southampton, Southampton (2002).

[75] R. Keser, Block-coupled solution algorithms for 2-equation turbulence models, Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb, 2016.

[76] H. Jasak, M. Beaudoin, OpenFOAM turbo tools: From general purpose CFD to turbomachinery simulations, in: ASME-JSME-KSME 2011 Joint Fluids Engineering Conference: Volume 1, Symposia – Parts A, B, C, and D, ASME, 2011. doi:10.1115/ajk2011-05015.
URL https://doi.org/10.1115/ajk2011-05015

[77] N. H. Centre, Experimental study of Francis 99 turbine (2014).
URL https://www.ntnu.edu/nvks/f99-test-case1

[78] Pointwise, the choice for CFD meshing (2019).
URL https://www.pointwise.com/

Curriculum Vitae

Tessa Uroić was born in Kutina on 2 October 1990 where she attended elementary and secondary school. During the undergraduate and graduate studies in power engineering at the Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb, she was awarded numerous prizes and acknowledgements for the results accomplished during the studies. After finishing her undergraduate and graduate studies with highest honours, she started working as a teaching assistant at the Faculty of Mechanical Engineering and Naval Architecture, and has since been teaching courses: Practical Finite Volume Method, Jet Engines 1 and 2, Windturbines and Hydraulic Turbomachinery. Her fields of interest include high quality CFD meshing, applied numerical linear algebra and algebraic multigrid. She has been dancing ballet in her hometown since the age of 9 and still perseveres to improve her dancing skills as well as carry her passion over to younger generations.

Declaration

Parts of the work presented in this thesis have been published in scientific journals.