PARALLEL BLOCK PRECONDITIONING OF THE INCOMPRESSIBLE NAVIER-STOKES EQUATIONS WITH WEAKLY IMPOSED BOUNDARY CONDITIONS

A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy in the Faculty of Engineering and Physical Sciences

2016

Raymon White
School of Computer Science
Contents

Abstract ............................................ 12

Declaration ........................................ 13

Copyright Statement .............................. 14

Acknowledgements ................................ 15

1 Introduction ...................................... 16
   1.1 Computational Modelling ....................... 16
   1.2 Numerical Solution of the Navier-Stokes Equations .................. 19
      1.2.1 Discretisation ................................ 20
   1.3 Parallel Computing ............................. 22
      1.3.1 Problem Partitioning ......................... 26
      1.3.2 Classical Parallel Computer Hardware Architecture ............. 26
      1.3.3 Parallel Computer Memory Architecture ....................... 28
      1.3.4 Communication Models ........................... 30
      1.3.5 Designing Efficient Parallel Algorithms ....................... 31
   1.4 Modelling Software ............................. 33
   1.5 Thesis Overview ................................ 35
   1.6 Main Project Contributions ..................... 37

2 The Incompressible Navier-Stokes Equations ... 39
   2.1 Derivation and Assumptions ..................... 39
   2.2 The Boundary Value Problem ..................... 44
      2.2.1 Non-dimensional Form .......................... 45
      2.2.2 Boundary Conditions ........................... 46
      2.2.3 Initial Conditions ............................. 48
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3</td>
<td>Spatial Discretisation</td>
<td>49</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Weak Formulation</td>
<td>51</td>
</tr>
<tr>
<td>2.4</td>
<td>Finite Element Discretisation</td>
<td>59</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Domain Discretisation</td>
<td>60</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Isoparametric Representation and Numerical Integration</td>
<td>66</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Stability of the Mixed Approximation</td>
<td>73</td>
</tr>
<tr>
<td>2.5</td>
<td>Parallel Implementation of a Finite Element System Assembly</td>
<td>74</td>
</tr>
<tr>
<td>2.6</td>
<td>Summary</td>
<td>78</td>
</tr>
<tr>
<td>3</td>
<td>Solution of Sparse Linear Systems</td>
<td>79</td>
</tr>
<tr>
<td>3.1</td>
<td>Direct Solvers for Dense Linear Systems</td>
<td>80</td>
</tr>
<tr>
<td>3.2</td>
<td>Sparse Direct Solvers</td>
<td>81</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Matrix Re-ordering Strategies</td>
<td>82</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Supernodal LU Solvers</td>
<td>82</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Frontal Solvers</td>
<td>82</td>
</tr>
<tr>
<td>3.3</td>
<td>Iterative Solvers</td>
<td>84</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Relaxation Methods</td>
<td>85</td>
</tr>
<tr>
<td>3.3.1.1</td>
<td>Rate of Convergence of the Relaxation Methods</td>
<td>87</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Multigrid Methods</td>
<td>90</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Projection Methods</td>
<td>94</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Krylov Methods</td>
<td>95</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Generalized Minimal Residual Method (GMRES)</td>
<td>95</td>
</tr>
<tr>
<td>3.3.6</td>
<td>Other Krylov Methods</td>
<td>98</td>
</tr>
<tr>
<td>3.4</td>
<td>Preconditioners</td>
<td>101</td>
</tr>
<tr>
<td>3.4.1</td>
<td>General Preconditioners</td>
<td>103</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Block Preconditioners</td>
<td>105</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Preconditioners for the Navier-Stokes System</td>
<td>106</td>
</tr>
<tr>
<td>3.4.3.1</td>
<td>Pressure Schur Complement Preconditioners</td>
<td>106</td>
</tr>
<tr>
<td>3.4.3.2</td>
<td>Augmented Lagrangian Preconditioners</td>
<td>111</td>
</tr>
<tr>
<td>3.5</td>
<td>Parallel Sparse Solvers</td>
<td>112</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Parallel Sparse Direct Solvers</td>
<td>113</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Parallel Iterative Solvers</td>
<td>114</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Parallel Preconditioners</td>
<td>116</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Domain Decomposition Preconditioners</td>
<td>118</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Parallel Coarsening Algorithms for AMG</td>
<td>120</td>
</tr>
</tbody>
</table>
6.3.1 Flow through a Unit Cube Rotated by an Arbitrary Angle . . . 218
6.3.2 Radial Flow through a Quarter Annulus . . . . . . . . . . . . 237
6.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 243
6.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 245

7 Parallel Performance Evaluation 248
7.1 Parallel Performance Review of the Initial BPF . . . . . . . . . . . 249
7.2 Performance Analysis of the get_block(...) Function . . . . . . . 250
  7.2.1 Weak Scaling for the get_block(...) Function . . . . . . . . . 251
  7.2.2 Strong Scaling for the get_block(...) Function . . . . . . . . . 254
7.3 Performance Analysis of the Functions get_block_vector(...) and
  return_block_vector(...) . . . . . . . . . . . . . . . . . . . . . . . . . . 255
  7.3.1 Weak Scaling for the Vector Functions . . . . . . . . . . . . . . 256
  7.3.2 Strong scaling for vector functions . . . . . . . . . . . . . . . . 257
7.4 Parallel Experiments: Flow through a Quarter Annulus in Three Spatial
  Dimensions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 258
  7.4.1 Weak Scaling Evaluation . . . . . . . . . . . . . . . . . . . . . . 259
  7.4.2 Strong Scaling Evaluation . . . . . . . . . . . . . . . . . . . . . 265
7.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 268

8 Conclusions and Future Work 270
8.1 Project Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 270
8.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 272
  8.2.1 Extensions to the Preconditioner Analysis and Implementation . 272
  8.2.2 Extensions to Numerical Simulations . . . . . . . . . . . . . . . . 273
  8.2.3 Extensions for the Software Implementation . . . . . . . . . . . 274

A Supporting Numerical Results 276
A.1 Numerical Results for 6.2.1 . . . . . . . . . . . . . . . . . . . . . . . . . 276
  A.1.1 AMG Strength of Dependence Parameter . . . . . . . . . . . . . . 276
  A.1.2 Damping Parameter for the Jacobi Smoother . . . . . . . . . . . 278

Word count: 69885
List of Tables

2.1 One-dimensional Gauss quadrature rules ...................................... 72

6.1 Numerical eigenvalue bounds of (4.6.8) with Re = 0 ......................... 185
6.2 Numerical eigenvalue bounds of (4.6.2) with Re = 0 ......................... 185
6.3 Iteration counts for $\hat{P}_{LSC}$ with different block approximations of $\hat{F}$ ............................................. 187
6.4 Execution times for $\hat{P}_{LSC}$ with different block approximations for $\hat{F}$ ............................................. 187
6.5 Iteration counts for $\hat{P}_{LSC}$ with varying $\theta$ using SVT-NSE .......... 189
6.6 Iteration counts for $\hat{P}_{LSC}$ with varying $\theta$ using SDVT-NSE .......... 189
6.7 Preconditioner setup times for $\hat{P}_{LSC}$ with varying $\theta$ using SVT-NSE .... 189
6.8 Preconditioner setup times for $\hat{P}_{LSC}$ with varying $\theta$ using SDVT-NSE .... 190
6.9 Solution times for $\hat{P}_{LSC}$ with varying $\theta$ using SVT-NSE ............. 190
6.10 Solution times for $\hat{P}_{LSC}$ with varying $\theta$ using SDVT-NSE ............. 190
6.11 Iteration counts for different smoothing strategies using SVT-NSE ...... 193
6.12 Iteration counts for different smoothing strategies using SDVT-NSE ...... 194
6.13 Execution times for different smoothing strategies using SVT-NSE ...... 194
6.14 Execution times for different smoothing strategies using SDVT-NSE ...... 194
6.15 Iteration counts for $P_E$ applied to Example 6.2.1 ......................... 198
6.16 Execution times for $P_E$ applied to Example 6.2.1 ......................... 198
6.17 Comparison of iteration counts for $P_{LSC}$ and $P_{NSLSC}$ .................. 200
6.18 Comparison of execution times for $P_{LSC}$ and $P_{NSLSC}$ .................. 201
6.19 Iteration counts for $P_{LSC}$ applied to Example 6.2.1 ......................... 201
6.20 Execution times for $P_{LSC}$ applied to Example 6.2.1 ......................... 201
6.21 Comparison of iteration counts for $\hat{P}_{LSC}$ and $\hat{P}_{NSLSC}$ ............... 203
6.22 Comparison of execution times for $\hat{P}_{LSC}$ and $\hat{P}_{NSLSC}$ ............... 204
6.23 Iteration counts for $\hat{P}_{LSC}$ applied to Example 6.2.1 ......................... 205
6.24 Execution times for $\hat{P}_{LSC}$ applied to Example 6.2.1 ......................... 205
6.25 Comparison of $P_{LSC}$ and $\hat{P}_{LSC}$ applied to Example 6.2.1 ......................... 209
6.26 Iteration counts for $\mathcal{P}_E$ applied to Example 6.2.3 . . . . . . . . . . . . . 212
6.27 Execution times for $\mathcal{P}_E$ applied to Example 6.2.3 . . . . . . . . . . . . 212
6.28 Iteration counts for $\mathcal{P}_{LSC}$ applied to Example 6.2.3 . . . . . . . . . . . 214
6.29 Execution times for $\mathcal{P}_{LSC}$ applied to Example 6.2.3 . . . . . . . . . . . 214
6.30 Iteration counts for $\hat{\mathcal{P}}_{LSC}$ applied to Example 6.2.3 . . . . . . . . . . . 215
6.31 Execution times for $\hat{\mathcal{P}}_{LSC}$ with different block approximations of $\hat{F}$ . . 223
6.32 Iteration counts for $\hat{\mathcal{P}}_{LSC}$ with varying $\theta$ using SVT-NSE . . . . . . . 223
6.33 Iteration counts for $\hat{\mathcal{P}}_{LSC}$ with varying $\theta$ using SDVT-NSE . . . . . . 223
6.34 Execution times for $\hat{\mathcal{P}}_{LSC}$ with varying $\theta$ using SVT-NSE . . . . . . . 223
6.35 Execution times for $\hat{\mathcal{P}}_{LSC}$ with varying $\theta$ using SDVT-NSE . . . . . . 223
6.36 Execution times for different smoothing strategies using SVT-NSE . . . . . . . . 227
6.37 Execution times for different smoothing strategies using SDVT-NSE . . . . . . . 228
6.38 Iteration counts for $\mathcal{P}_E$ applied to Example 6.3.1 . . . . . . . . . . . . . 230
6.39 Iteration counts for $\mathcal{P}_{LSC}$ applied to Example 6.3.1 . . . . . . . . . . . 231
6.40 Iteration counts for $\hat{\mathcal{P}}_{LSC}$ applied to Example 6.3.1 . . . . . . . . . . . 233
6.41 Iteration counts for $\mathcal{P}_E$ applied to Example 6.3.2 . . . . . . . . . . . . . 240
6.42 Iteration counts for $\mathcal{P}_{LSC}$ applied to Example 6.3.2 . . . . . . . . . . . 241
6.43 Iteration counts for $\hat{\mathcal{P}}_{LSC}$ applied to Example 6.3.2 . . . . . . . . . . . 242
7.1 Comparison of problem sizes for matrix concatenation . . . . . . . . . . . . . . 251
7.2 Weak scaling for $\text{get\_block}(...)$ . . . . . . . . . . . . . . . . . . . . . . 252
7.3 Strong scaling for $\text{get\_block}(...)$ . . . . . . . . . . . . . . . . . . . . . 255
7.4 Weak scaling for vector functions . . . . . . . . . . . . . . . . . . . . . . . . . . . 256
7.5 Strong scaling for vector functions . . . . . . . . . . . . . . . . . . . . . . . . . . 257
7.6 Comparison of problem sizes for 3D quarter annulus . . . . . . . . . . . . . . 259
7.7 Weak scaling for 3D quarter annulus . . . . . . . . . . . . . . . . . . . . . . . . . . 260
7.8 Weak scaling for preconditioner setup . . . . . . . . . . . . . . . . . . . . . . . . . 260
7.9 Weak scaling for SpGEMM and SpMV . . . . . . . . . . . . . . . . . . . . . . . . . 262
7.10 Weak scaling for preconditioner solve . . . . . . . . . . . . . . . . . . . . . . . . . 264
7.11 Strong scaling for 3D quarter annulus . . . . . . . . . . . . . . . . . . . . . . . . . 265
7.12 Strong scaling for preconditioner setup . . . . . . . . . . . . . . . . . . . . . . . . . 266
7.13 Strong scaling for preconditioner solve . . . . . . . . . . . . . . . . . . . . . . . . . 266
A.1 Iteration counts for $\hat{\mathcal{P}}_{LSC}$ with varying $\theta$ using SVT-NSE . . . . . . . 277
A.2 Iteration counts for $\hat{\mathcal{P}}_{LSC}$ with varying $\theta$ using SDVT-NSE . . . . . . 277
A.3 Preconditioner setup times for $\hat{P}_{LSC}$ with varying $\theta$ using SVT-NSE . 277
A.4 Preconditioner setup times for $\hat{P}_{LSC}$ with varying $\theta$ using SDVT-NSE . 277
A.5 Solution times for $\hat{P}_{LSC}$ with varying $\theta$ using SVT-NSE . . . . . . . . . 277
A.6 Solution times for $\hat{P}_{LSC}$ with varying $\theta$ using SDVT-NSE . . . . . . . . 278
A.7 Iteration counts for $\hat{P}_{LSC}$ with Jacobi smoothing using SVT-NSE . . . . . . . . 278
A.8 Iteration counts for $\hat{P}_{LSC}$ with Jacobi smoothing using SDVT-NSE . . . . . . . . 279
A.9 Execution times for $\hat{P}_{LSC}$ with Jacobi smoothing using SVT-NSE . . . . . . . . 279
A.10 Execution times for $\hat{P}_{LSC}$ with Jacobi smoothing using SDVT-NSE . . . . . . . . 279
# List of Figures

1.1 Fluid dynamics subdivision with respect to the Knudsen number ............................................. 18  
1.2 Clock frequency of Intel CPUs 1971–2014 ................................................................................... 23  
1.3 Flynn’s taxonomy ......................................................................................................................... 27  
1.4 Shared UMA architecture ............................................................................................................ 28  
1.5 Shared memory NUMA architecture ............................................................................................ 29  
1.6 Distributed memory architecture .................................................................................................. 29  
1.7 Hybrid distributed-shared memory architecture .............................................................................. 30  
2.1 Typical configurations of boundary problems ................................................................................ 46  
2.2 Structured grid discretisation of a domain ....................................................................................... 60  
2.3 Triangular reference element .......................................................................................................... 62  
2.4 $P_1$ and $P_2$ reference elements .................................................................................................. 63  
2.5 Quadrilateral reference element .................................................................................................... 63  
2.6 $Q_1$ and $Q_2$ reference elements .................................................................................................. 65  
2.7 $P_1$ and $P_2$ tetrahedral elements .................................................................................................. 65  
2.8 $Q_1$ and $Q_2$ brick elements .......................................................................................................... 66  
2.9 Geometric representation of the determinants ................................................................................ 69  
2.10 Displacement of node 4 of the $Q_1$ quadrilateral element ............................................................ 70  
2.11 Quadrilateral Taylor-Hood ($Q_2$ − $Q_1$) element ........................................................................ 74  
2.12 Domain decomposition via METIS ............................................................................................... 76  
3.1 Eigenvalues of the damped Jacobi iteration matrix .......................................................................... 90  
4.1 Strongly imposed boundary conditions .......................................................................................... 122  
4.2 Imposing boundary conditions on non-axes aligned boundaries .................................................. 123  
4.3 Imposing parallel outflow through a curvilinear boundary $\partial\Omega_e$ ........................................... 123  
4.4 Penalised flow along a boundary .................................................................................................... 124  
4.5 Matching and non-matching meshes ............................................................................................. 129
6.22 Comparison of different solver times for Example 6.2.3 . . . . . . . . . 217
6.23 Configuration of boundary conditions for Example 6.3.1 . . . . . . . . . 219
6.24 Solution plot of unsteady flow through a unit cube rotated by $\theta = 30^\circ$ . 220
6.25 Iteration counts for $\hat{P}_{LSC}$ with different block approximations of $\hat{F}$ . . 221
6.26 Execution times for $\hat{P}_{LSC}$ with different block approximations of $\hat{F}$ . . 222
6.27 Iteration counts for AMG strength of dependence study . . . . . . . . . 224
6.28 Execution times for AMG strength of dependence study . . . . . . . . . 225
6.29 Iteration counts for different smoothing strategies using SVT-NSE . . . . . 226
6.30 Execution times for different smoothing strategies using SVT-NSE . . . . . 226
6.31 Iteration counts for different smoothing strategies using SDVT-NSE . . . . . 227
6.32 Execution times for different smoothing strategies using SDVT-NSE . . . . . 228
6.33 Execution times for $P_E$ applied to Example 6.3.1 . . . . . . . . . . . . 230
6.34 Iteration counts for $P_{LSC}$ applied to Example 6.3.1 . . . . . . . . . . . . 231
6.35 Execution times for $P_{LSC}$ applied to Example 6.3.1 . . . . . . . . . . . . 232
6.36 Plot of scaled perturbation values with iteration counts . . . . . . . . . . 235
6.37 Iteration counts for $\hat{P}_{LSC}$ applied to Example 6.3.1 . . . . . . . . . . . . 236
6.38 Execution times for $\hat{P}_{LSC}$ applied to Example 6.3.1 . . . . . . . . . . . . 236
6.39 Comparison of different solver times for Example 6.3.1 . . . . . . . . . . . 237
6.40 Solution plot of uniform radial flow through a 3D quarter annulus . . . . . 239
6.41 Execution times for $P_E$ applied to Example 6.3.2 . . . . . . . . . . . . 240
6.42 Execution times for $P_{LSC}$ applied to Example 6.3.2 . . . . . . . . . . . . 241
6.43 Execution times for $\hat{P}_{LSC}$ applied to Example 6.3.2 . . . . . . . . . . . . 242
6.44 Comparison of different solver times for Example 6.3.2 . . . . . . . . . . . 243

7.1 Contribution of preconditioner setup operations under weak scaling . . . . 263
7.2 Relative contribution of solve phases under weak scaling . . . . . . . . . . 265
7.3 Relative contribution of setup phases under strong scaling . . . . . . . . . 267
7.4 Relative contribution of solve phases under strong scaling . . . . . . . . . 268
This project is concerned with the development and implementation of a novel preconditioning method for the iterative solution of linear systems that arise in the finite element discretisation of the incompressible Navier-Stokes equations with weakly imposed boundary conditions.

In this context we studied an augmented approach where the Schur complement associated with the momentum block of the Navier-Stokes equations has special sparse structure. We follow the standard \textit{inf-sup} stable method of discretising the Navier-Stokes equations by the Taylor-Hood elements with the Lagrange multiplier constraints discretised using the same order approximation on matching grids. The resulting system of nonlinear equations is solved iteratively by Newton’s method.

The spectrum of the linearised Oseen’s problem, preconditioned by the exact augmentation preconditioner was analysed. Then we developed inexact versions of the preconditioner aimed at achieving optimal scaling of the algorithm in terms of computational resources and wall-clock times.

The experimental evaluation of the methodology involve a number of benchmark problems in two and three spatial dimensions. The obtained results demonstrate efficiency, robustness and almost optimal scaling of the solution algorithm with respect to the discrete problem size.

We used \texttt{oomph-lib} as a testbed for our experiments. The preconditioning strategies were implemented using \texttt{oomph-lib}’s Parallel Block Preconditioning Framework. The initial version of the software was significantly upgraded during the course of this project with newly implemented functionalities to facilitate the rapid development of sophisticated hierarchical design of parallel block preconditioners. Parallel performance analysis of the newly introduced functionalities demonstrate negligible overhead in terms of wall-clock time and the framework demonstrate good weak and strong parallel scaling.
Declaration

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright Statement

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s Policy on Presentation of Theses.
Acknowledgements

I would like to express my special appreciation to my supervisor Dr. Milan D. Miha-
jlović for the tremendous support throughout my PhD study. His encouragement and
enthusiastic supervision was motivational, and made my research a stimulating and
fulfilling experience. His contributions of time, ideas and knowledge have been pivotal
to the success of this research project and in writing this thesis.

I am grateful to Prof. Matthias Heil for the many helpful and insightful con-
versations during my PhD. His suggestions and advice was vital for the fruition of
this project. A very special thanks goes out to the Research Infrastructure team, in
particular, George Leaver and Pen Richardson, for their unparalleled support and as-
sistance in using the Computational Shared Facility (Danzek). I wish to acknowledge
the generous financial support provided by the ESPRC and the school of Computer
Science.

I would like to make a special mention to the NHS, especially Dr. Janet Suckley,
for the continued medical treatment of the chronic nerve damage in my hands. This
had been a key factor which allowed me to complete this thesis.

I would like to show gratitude to Prof. Thomas Thomson, the NEST group and
my co-workers for making my time at the University of Manchester a very rewarding
and enjoyable experience. I would also like to extend my gratitude to all my friends
and family for their support and encouragement.

Last, but not least, I make a special thank you to Johanna Keljo. Thank you for
all your help and keeping me sane.
Chapter 1

Introduction

1.1 Computational Modelling

The research project discussed in this thesis fits into the broad area of numerical modelling and simulation. The aim of numerical modelling is to express the behaviour of physical and engineering systems in many areas of science through a mathematical formalism, thus allowing their study, and the design and optimisation of novel products using computers. This area of research sits in between analytical methods (i.e. the study of numerical and asymptotic analysis of the models of system behaviours) and experimental sciences. Combined together, the three approaches form a powerful research framework for tackling challenging problems in modern science.

Mathematical models of physical systems aim to accurately capture the behaviour of physical phenomena. These models are frequently expressed in terms of differential equations (DE), which give the relation between the unknown physical quantities and their derivatives. Often these expressions are too complex to solve analytically. DEs can be solved analytically only in a few special cases. For a broad treatment of important types of DEs, with an emphasis on analytical techniques, see [265, 153]. The alternative is to use numerical techniques for their approximate solution and implement these methods on a computer. DEs represent continuous models involving continuous functions. Numerical techniques for their solution replace this problem by a discrete counterpart, making it suitable for computer implementation. The combination of analytical tools and numerical simulations allow for a more in-depth understanding and comprehensive treatment of various important models in modern science.

In the past several decades most scientific fields have adopted computer simulations. It is possible to find a corresponding computational branch for many major fields in
science and engineering such as: astrophysics, particle physics, chemical engineering, continuum mechanics (including structural and fluid), electro-magnetism, systems in biology, social sciences and economics. In the sequel we provide a few examples.

In computational astrophysics, computational methods are developed to study problems arising from general relativity [49] (referred to as numerical relativity) and to model the motions of stars, galaxies and dark matter [47, 249] (referred to as celestial dynamics). Computational chemistry incorporate models of theoretical chemistry into efficient computer programs. They range from ab initio quantum chemistry methods, where the methods are derived from theoretical principles of atomic laws with no inclusion of experimental data [160, p. 480] to molecular dynamics where parameters in the equations describing the systems must be obtained from experimental data. In molecular dynamics, the theories of molecular mechanics are applied to molecular systems, examples include modelling graphene [223, 267] and carbon nanotubes [91, 155]. This research area overlaps with computational biology and modelling of biological systems, where molecular dynamics models are used to study protein and cell functions, such as protein folding [164] and simulations of cell membrane systems [199]. In quantitative finance, computational methods are used to model a wide range of social and economical phenomena such as modelling the price of financial derivatives (using the Black-Scholes partial differential equation [187]) and risk analysis (which typically involve statistical and Monte Carlo methods [7]).

Numerical simulations of fluid dynamics are referred to as computational fluid dynamics (CFD). In this context, numerical models are employed to study and analyse problems involving fluid flows. The most suitable approach depends on the type of fluid motion (the flow). A finite volume of fluid is composed of molecules which collide with each other or with solid objects (domain boundaries). If we assume that the fluid consists of a single species of atom/molecule which can be described using constant temperature, then we can typically classify three states of fluid: liquid such as water, gas such as air and plasma such as ionised gas. We must decide in each case whether it is appropriate to model the fluid as the discrete particle system or to approximate it as a continuous medium. The latter case assumes that the set of particles in the fluid is dense enough to be considered continuous, this is known as the continuum assumption. In the first instance, we can quantify the type of fluid flow by defining its Knudsen number [218, p. 23], $\text{Kn} = \frac{\lambda}{\mathcal{L}}$, where $\lambda$ is the mean free path - the average distance travelled by a particle before collision with another particle and $\mathcal{L}$ is the characteristic
length, which defines the scale of the physical problem (e.g. in a channel flow, $\mathcal{L}$ may be taken as the width of the channel). The Knudsen number is useful for the classification of fluid flows and in determining which formulation of fluid dynamics, either continuum mechanics or statistical mechanics, is most suitable to describe the fluid motion [191, p. 717]. As depicted in Figure 1.1, for the continuum mechanics formulation of fluid dynamics we can use models such as the Navier-Stokes equations (NSE) [191, p. 125], leading to the Euler equations [191, p. 411] in the limiting case as $Kn \rightarrow 0$. In this thesis we model incompressible fluids which are continuous, i.e. the Knudsen number satisfies $Kn \ll 1$ [218, p. 23].

When the continuum assumption does not hold, discrete particle models such as the Boltzmann equations should be used, leading to free molecular equations (such as the collisionless Boltzmann equations [45, pp. 347–357]) as $Kn \rightarrow \infty$. The classification of various flow regimes for Knudsen numbers close to or greater 1 is discussed in [15]. It is possible to extend the NSE to Knudsen number $Kn \approx 1$. This is typically done because kinetic-theory based numerical schemes such as Direct Simulation Monte Carlo (DSMC) are computationally expensive compared to numerical schemes developed for solving the NSE [208, 64, 108]. The Boltzmann equation satisfies the NSE in the continuum limit [93, 217], thus can be used to model fluid flows under the continuum assumption. The most suitable approach is problem dependent. For example, for problems involving multiphase flow (e.g. fluid consisting of materials with different states) and different characteristic length scales, the Lattice-Boltzmann method (a numerical scheme for the discrete Boltzmann equations) is more suitable since the Boltzmann equation is valid over a wider range of Knudsen numbers [191, p. 721].

The Lattice-Boltzmann method covers large spatial scales and is inherently transient, thus its solution for a steady state problem can be very expensive when compared to a NSE solution [87]. An alternative approach in modelling fluid flows with a wide

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure11.png}
\caption{Formulations of fluid dynamics as a function of the Knudsen number $Kn$.}
\end{figure}
range of Kn (i.e. covering details in a wide range of spatio-temporal scales) is to couple together existing continuum and molecular-dynamics models. This framework is often referred to as multiscale modelling [100]. This is a very complex approach which require advances in numerical mathematics, continuum mechanics, algorithms and software engineering. Under the continuum assumption, the NSE approach is the most frequently used and well researched method employed to model single phase fluid flows.

In this project we use the NSE to model incompressible viscous fluid flows. There is a vast number of applications involving fluid flows, such as aircraft design in aerodynamics [144], construction of heat exchange systems in thermodynamics [62], weather prediction in climatology [266] and oceanic modelling [54]. The target applications that motivate our research come from the area of biomechanical modelling, but the methodologies developed in this thesis are generic and thus applicable to other areas involving fluid flows. Modelling of fluids has important applications in biomechanical engineering, e.g. modelling of the blood flow through the vessels [39, 173] and the air flow through the bronchials [190, 133]. The main issue is that in vivo experiments for such problems are difficult or unethical to perform, thus numerical modelling can provide an invaluable insight into the physics of certain processes, such as the first breath scenario, and highlight the behaviour in certain abnormal or diseased situations e.g. the formation of liquid bridges blocking the bronchials [127], or air streams which are significantly altered by surface abnormalities and tubular constrictions [79]. Analytical and numerical sciences, combined with carefully crafted experiments from medical and physiological sciences, are a more suitable approach to study such scenarios. The interdisciplinary collaboration of analytical, computational and experimental sciences is vital to the continual development and success in modelling realistic problems.

1.2 Numerical Solution of the Navier-Stokes Equations

The primary formulation of the NSE is a system of two nonlinear partial differential equations (PDEs) in unknown velocity field and pressure. These two equations represent fundamental laws of physics applied to the fluid motion, namely, the conservation of momentum and the conservation of mass. The NSE can be solved analytically in only a few very special cases, such as the Couette flow (laminar flow of a viscous
fluid in the space between two parallel plates), Poiseuille flow (laminar flow through a straight pipe of constant profile induced by parabolic inflow), and the oscillatory Stokes boundary layer (boundary layer close to a solid wall in oscillatory flow of a viscous fluid), details can be found in [254]. Analytical solutions (for special cases, such as the planar flow) can also be derived from the vorticity formulation of the NSE [99, pp. 363–364]. For this reason, finding effective approximation methods to solve the NSE numerically continues to be a very fertile area of research.

1.2.1 Discretisation

Numerical solution of the NSE assumes some form of discretisation. In this project we use the finite element method (FEM) [99, pp. 438–662, 68, pp. 313–338]. The domain of interest is subdivided into a set of disjoint sub-domains (each being a simple polygon, such as a quadrilateral or hexahedron) referred to as elements, this subdivision forms a mesh. Two neighbouring elements can share only a common edge or a common node referred as the vertex. The discrete finite element (FE) solution is computed at the nodes of the mesh and then interpolated across the elements. The quality of the approximation is determined by the order of the interpolation polynomial (this defines the order of the approximation) and the distribution (size) of elements. The FE discretisation of the incompressible NSE and the choice of elements are discussed in Chapter 2. We review other popular spatial discretisation schemes for the NSE: finite difference (FD) methods [232], finite volume (FV) methods [248] and spectral methods [42].

FD methods are based on the FD approximation of the derivatives in the PDEs. The approximation of the derivative is based on Taylor’s series expansion and is often represented as a stencil which shows the relation between a derivative at a particular point and the value of the function at that and the neighbouring points. This decomposes the domain into a set of discrete points. FD methods are simple to implement in the case of regular tensor product spatial domains, but this can be more difficult for general complex domains. The marker and cell (MAC) method [109] is a FD based method originally developed for the free surface transient NSE. In this approach, marker particles are distributed in the fluid and move along with the velocity field. The marker particles are used to determine which cells contain fluid. We discuss the advantages and disadvantages of FD based methods in Section 2.3.
FV approximation of PDEs possesses the property of local conservation of the relevant quantities. The domain of interest is partitioned into a number of non-overlapping volumes (referred to as control volumes), each of which containing a node in the mesh. The PDE is transformed into a weak formulation involving volume integrals. Evaluation of these integrals over the control volume ensures the conservation of fluxes between the neighbouring volumes (in the case of the NSE, the mass of fluid in each control volume is conserved).

In spectral methods, the approximation at a given node depends on all other nodes in the domain. This is known as a global method and is achieved using approximation functions (referred to as basis functions) with global support. The set of basis functions is usually chosen to be either harmonic or polynomial functions. In higher dimensions, if a tensor-product domain is considered, the basis functions are obtained as tensor-products of the lower-dimensional basis sets. In practice, the method suffers the same problems as the FD method: it is not straightforward to model more complex domains and complex boundary conditions (BCs) because we need to construct a global orthogonal basis set on that particular domain that satisfies the BCs. The upshot is that in order to achieve the level of accuracy comparable to the methods with local basis sets (such as FE or FD methods), we need considerably fewer basis functions. Consequently, the resulting linear systems are much smaller (typically 10000s of equations).

The FE discretisation of the NSE results in large, sparse systems of nonlinear algebraic equations that need to be solved. There exists a number of different iterative schemes for the solution of nonlinear systems, such as the fixed-point iteration [75, p. 126], Picard’s iteration [55, pp. 182–184] and Newton’s method [143, pp. 97–99]. In this project, we use Newton’s method to linearise our system of nonlinear equations.

Linearisation of these problems require repeated solution of systems of linear algebraic equations. Therefore effective algorithms for the solution of linear systems are the key for constructing effective numerical DE solvers. Such systems can be solved effectively using iterative solution techniques. The ultimate goal in the development of iterative linear solvers for sparse linear systems obtained from FE discretisations is to create an optimal solver. In this context, an optimal solver is one where the computational cost, expressed as the number of floating point operations (flops) or the wall-clock time, and storage requirements, expressed as the size of the required

---

1The number of nonzero entries in the coefficient matrix associated with the discretised NSE is proportional to the size of the matrix. We quantify the notion of a sparse matrix in Chapters 2 and 3.
memory, scale linearly with the problem size (the number of unknowns in the system). However, commonly used iterative solvers have poor convergence or may not converge at all when applied to discrete FE problems. The convergence of iterative solvers is accelerated using a technique referred to as \textit{preconditioning} [251, pp. 173–204]. Preconditioning transforms the original system into one with the same solution, but which is much easier to solve by iterative methods. The introduction of preconditioners increase the computational cost and memory requirements of each linear iteration significantly. Thus preconditioners need to be efficient, so that their use leads to a considerable reduction in the overall wall-clock time, compared to an unpreconditioned case, and the memory requirement should not be prohibitive. The aim in developing preconditioners for a particular problem or a class of problems is to achieve optimal scaling of the resulting linear solver.

Our research is concerned with the development and evaluation of novel preconditioning techniques for the solution of linear systems obtained from the discretisation of the NSE with non-standard boundary conditions imposed through Lagrange multipliers [99, p. 375]. Such scenarios arise in modelling of fluid flows with imposed constraints, examples include blood flow through a blood vessel junction [115] or fluid structure interaction problems [181]. To achieve acceptable execution time for the simulation of realistic, large-scale problems, one also needs to consider parallelisation of the developed efficient solvers. In the next section we review parallel programming models.

\section{Parallel Computing}

Traditionally, to solve a problem on a computer, an algorithm is constructed which maps the solution procedure to a discrete set of instructions. These instructions are executed on a single computing resource, for example, a central processing unit (CPU), of a computer. If a given architecture has one CPU and no other hardware accelerators, only one stream of instructions may be executed. This is referred to as \textit{serial} computation. The clock frequency of a CPU (also referred as clock rate or just frequency) gives an indication of the processor’s speed and is measured in SI unit hertz (Hz). A CPU can be made faster by increasing the clock frequency, this process is known as frequency scaling. From the mid-1980s to 2004, this was the primary technique used
1.3. PARALLEL COMPUTING

to increase the computational speed. During this time, the frequency of CPUs approximately doubled every two years. This is referred to as Moore’s law, named after Gordon E. Moore who described this trend in [178]. From 2004 to 2014 there is relatively little increase in the CPU frequency compared to previous years, see Figure 1.2. There are two main reasons for the lack of growth. Firstly, the signals in electronic circuits cannot propagate faster than the speed of light (a fundamental physical constraint). Secondly, the miniaturisation of the electronic components (e.g. transistors) cannot go indefinitely without a fundamental change in technology. Miniaturisation faces two fundamental obstacles: packing too many components closely together will cause their interaction through parasitic capacitances and leakage currents (which grow significantly with miniaturisation), thus increasing the energy consumption and compromising the circuit reliability; dense packaging also reduces the volume of the chip over which thermal energy is dissipated. This affects both the reliability of the electronic components and the power supply networks. The CPU power consumption is given by the equation $P \approx CV^2f$, where $C$ is the capacitance (a function of wire length and transistor size), $V$ is the voltage, and $f$ is the frequency [74, p. 5], [193, p. 23]. This increase in power consumption and hence high heat emission is the most important limit in increasing CPU’s frequency [83]. The trend now is to increase performance by using multiple processing units such as multi-core and multi-processor
computer architectures. This lead to the concept of parallel computing. Parallel computing is the simultaneous use of more than one computing resource to solve a problem by constructing a parallel algorithm which subdivides the problem into discrete parts in which the instructions can be executed concurrently on multiple computational resources. The aim is to shorten the time required to solve the problem or to solve a problem of larger size in the same time required to solve a smaller problem on a single processor.

Other common forms of parallel architectures include, but not limited to, field-programmable gate arrays (FPGA) [216, pp. 1–3], and graphical processing units (GPU) [193, pp. 288–312]. FPGAs contain an array of programmable logic blocks, these are integrated circuits which can be customised by the user after manufacturing to perform specific logical functions. FPGAs have a reconfigurable computing architecture with the ability to make substantial changes to the dataflow (the ‘building blocks’ of a processing unit), thus high performance computing can be achieved. However, we need to ensure that data management is implemented appropriately (according to the re-configured architecture) and exploit the reuse of data once it has been fetched from memory. That may require new sparse data formats and a rewrite of known sparse algorithms (e.g. communication avoiding algorithms [61]). GPUs are optimised for vector operations with typically a few hundred parallel floating-point units operating at a lower frequency than a typical CPU. The two most common programming languages for GPU programming are Nvidia’s CUDA and OpenCL, a vendor-independent language. Generally, FPGAs are more energy efficient (in terms of performance per watt) than GPUs and GPUs are in turn more energy efficient than CPUs [177]. It is possible to have CPUs with a GPU accelerator on the same chip, this is referred to as CPU-GPU and is an example of a heterogeneous (hybrid) system architecture, for example, the AMD’s Fusion APU [3]. However, GPUs may have poor performance for algorithms with a lot of conditional branching and communication intensive procedures [193, pp. 302–303].

Although the power of desktop computers are increasing year by year (it is not uncommon to have 4 to 8 cores and 8 to 16 gigabytes (GB) of random access memory (RAM) on a typical desktop computer), the problem size in numerical simulations, such as those presented in this thesis, can become so large that running them on a desktop computer becomes infeasible. One solution to this is to use computer clusters.

\(\text{A processing unit for floating point computations.}\)
1.3. PARALLEL COMPUTING

connected by high speed interconnects. This is the most common architecture used in modern high performance computing (HPC). The term supercomputers is used to describe systems with significantly greater level of computational power compared to a typical general purpose computer. The TOP500 project publishes a list of the top 500 supercomputers biannually, they are considered the world’s most powerful commercially available computers. Their performance is ranked in terms of flops per second using the peak performance achieved with the LINPACK Benchmark (HPL) and their theoretical peak performance. The current systems in the 10 top are capable of performing teraflops per second\(^3\). The Collaboration of Oak Ridge, Argonne, and Livermore (CORAL) is expecting to deliver two petascale supercomputers\(^4\) by 2017.

In the sequel, a (computing) process refers to an instance of independent sequences of execution. A processing element/unit is the physical hardware for executing the process, they can be referred to as processors (in the case of a single-core processors) or cores (in the case of multi-core processors). Typically, a unique process is associated with each processing element. A CPU may have a single core or multiple cores. A computer (referred to as a node) may contain a single CPU or multiple CPUs (referred to as a multi-processor system). The main memory (which is typically RAM) may be shared among the CPUs or distributed, the practical implications of shared/distributed memory architectures are discussed in Section 1.3.3. Nodes can be connected together with networks to form a cluster.

In designing parallel programs, in the first instance, we must consider how to partition the problem into discrete parts to be executed on each core and how the cores communicate with each other (the communication model). The partitioning is usually problem-dependent, this is discussed in Section 1.3.1. The choice of communication model is largely dependent on the type of parallel computer architecture. In this regard, we first introduce different types of parallel computing architectures in terms of instruction and data streams (see Section 1.3.2), and in terms of memory architectures (see Section 1.3.3). Then models of communication between processes are discussed in Section 1.3.4. Finally, we consider aspects of designing efficient parallel algorithms in Section 1.3.5.

---

3The current rank 1 system, the Tianhe-2, measures 33,862 teraflops per second using the LINPACK Benchmark. One teraflop is \(10^{12}\) flops.

4One petaflop is \(10^{15}\) flops.
1.3.1 Problem Partitioning

Problem partitioning strategies can be classified into domain decomposition or functional decomposition [152], [193, p. 9].

In the domain decomposition approach, the data set associated with the problem is decomposed into discrete subsets, which can then be operated on simultaneously. This is referred to as data-level parallelism (DLP). FE computation typically employ this method of problem partitioning (e.g. in the assembly phase, the spatial domain is divided into sub-domains among the processes and in the solution phase, the coefficient matrix is divided into sub-matrices).

In functional decomposition, the instruction set associated with the problem is partitioned into discrete sets of instructions, followed by concurrent execution of these sets of instructions by independent processes. This is also referred to as task-level parallelism (TLP). This type of parallelism can be exploited in multiscale simulations, such as modelling multiphase flows [220]. These two problem partitioning strategies can be combined, e.g. in multi-scale and/or multi-physics simulations [19]. Data-level and task-level parallelism can be exploited by different types of parallel architectures, this discussed in the next section.

1.3.2 Classical Parallel Computer Hardware Architecture

We introduce the classification of classical parallel computer architectures (referred to as Flynn’s taxonomy [81]) into four categories. These are based upon different combination of instruction and data streams, see Figure 1.3.

**Single Instruction stream, Single Data stream (SISD):** This is a serial computer (a classical von Neumann model). The single core CPU acts on one instruction stream at any one time and only one data stream is used as an input, see Figure 1.3(a). Examples include single-core, single-processor computers and old generation mainframes. Modern processors have multi-stage instruction pipelines [193, pp. C-6 – C-11] which can perform multiple instructions concurrently (also referred to as multiple-issue processors). This type of architecture exploits instruction-level parallelism (ILP), where the stream of instructions is re-ordered and partitioned into groups which can be processed in parallel.

**Single Instruction stream, Multiple Data stream (SIMD):** This architecture model exploits multiple data streams by operating on different data elements processed
1.3. PARALLEL COMPUTING

(a) Single Instruction stream, Single Data stream (SISD) architecture.

(b) Single Instruction stream, Multiple Data stream (SIMD).

(c) Multiple Instruction stream, Single Data stream (MISD).

(d) Multiple Instruction stream, Multiple Data stream (MIMD).

Figure 1.3: Four classical parallel computing architectures (Flynn’s taxonomy).\footnote{Figures have been adapted with permission from \cite{17}.}

by a single instruction stream, that is, all processing units execute the same instruction, see Figure 1.3(b). Examples include vector architectures (such as array processors) and GPUs. This architecture naturally exploits DLP.

**Multiple Instruction stream, Single Data stream (MISD):** Multiple processing units independently operate on a single data stream using separate and independent instruction streams, see Figure 1.3(c). This is a rarely used architecture organisation.

**Multiple Instruction stream, Multiple Data stream (MIMD):** Multiple processing units independently execute different instructions on different data streams simultaneously, see Figure 1.3(d) For example, nodes connected by a network may issue different instructions independently of other nodes, with each node operating on its own data set. Thus this type of system primarily targets task-level parallelism.

Most modern parallel architectures are a hybrid of SISD, SIMD, and MIMD classifications, for example, the nodes (in a MIMD configuration) may contain GPUs. A large
proportion of computers in the TOP500 list are inevitably hybrid and many utilise GPU accelerators to achieve energy-efficient high computational power [177].

1.3.3 Parallel Computer Memory Architecture

The cache is a much smaller and faster memory located close to the processing unit. In modern architectures there exist a cache hierarchy consisting of several levels of different sizes and speeds (with the speed inversely proportional to the size). Data which is frequently used by the CPU is copied from the main memory (typically RAM) to reduce access time. For an overview of how CPU cache works, see [193, Appendix B]. The program and its data are stored in main memory during its execution. Each discrete memory location is accessed by a memory address and the range of memory addresses is called the address space. In a parallel machine the memory is either (logically) shared or distributed.

In a shared memory architecture, all processors have access to the entire memory space by the means of a global address space. We discuss two shared memory architectures, Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA) [193, pp. 346–348]. Architectures in which all processors are identical and have equal access (in terms of bandwidth and access time) to each location in memory are classified as UMA systems and are typically represented by symmetric multiprocessors (SMP) [193, pp. 346–347]. In such as systems, two or more identical processors have access to a centralised shared memory as shown in Figure 1.4.

![Figure 1.4: Shared memory UMA architecture. A symmetric multiprocessor. The letter P represent processors and I/O represents the input/output unit of the system. All processors have uniform access time to the entire shared memory.](image)

Computer architectures where not all processors have uniform access time to the main memory are referred to as NUMA systems, see Figure 1.5. An example being a distributed shared memory (DSM) multiprocessor [193, pp. 347–348] where the memory is physically distributed. The use of a global address space means that the programmer does not need to specify the location of the data. The overhead due to communication is typically much lower for shared memory than for distributed
1.3. PARALLEL COMPUTING

memory architectures. However, it is difficult to achieve good scalability of shared memory architectures. Adding more processing units causes two main issues: the cache coherency problem [193, p. 352] and performance degradation due to the contention for shared memory resources (i.e. the increase in traffic among CPU-memory paths when multiple processors access the shared resources simultaneously) [26, 204, p. 77]. As a result, it can become expensive to design and produce scalable shared memory machines with a large number of processors.

In a distributed memory parallel machine, each processor has its own local memory and thus local address space as shown in Figure 1.6. There is no mapping of memory addresses from one processor to another. When data from one processor is required by another, it must be explicitly communicated. This is usually handled by the programmer using calls to the communication libraries (see Section 1.3.4). The interconnecting network in distributed memory architectures vary, for example, it could be Gigabit ethernet or Infiniband [139, 193, pp. F-77–F-80]. Increasing the number of processors will increase the memory proportionately, thus it is easily scalable with the number of processors. Each processor has its local memory address space which means that there is significantly less contention for shared resources and there is no overhead due to maintaining a global memory address space. The components required for putting together such an architecture can be off-the-shelf, thus increasing the number of processors is cost effective. However, communication is significantly slower than in
shared memory architectures and non uniform memory access times may cause synchronisation to be orders of magnitude slower than in shared memory machines. In distributed memory machines the decomposition of the problem (both domain and functional decomposition) has to be explicitly handled by the programmer.

Hybrid distributed-shared memory architectures consists of a network of shared memory nodes as shown in Figure 1.7. The vast majority of high performance computers fall into this category.

![Network](image)

Figure 1.7: Hybrid distributed-shared memory example. Memory is distributed across the three nodes. Each node have shared memory architecture (consisting of four processors denoted by P, with shared memory denoted by M). The nodes are connected by a network, memory is local to each node and access to data between nodes require explicit communication calls and a transfer across the network.

### 1.3.4 Communication Models

The most appropriate choice of a communication model largely depends on the underlying memory architecture of the system, although we stress that communication models and parallel programming models are an abstraction above the hardware and memory architectures. The most common forms of communication model for parallel programming are shared memory and message passing [149], [193, pp. I-4–I-6].

**Shared Memory Communication Model:** Processors share a single global address space and data is communicated using the shared memory which they read from and write to asynchronously. This is handled by the processor with load and store operations. From a programmer perspective, communication is implicit as the data’s physical location is unspecified by the model, however the programmer must use synchronisation mechanisms such as semaphores, locks, and barriers to ensure the correctness of the program execution [209, pp. 145–147]. Shared memory communication model can be implemented on a distributed memory architecture. Advantages of shared memory communication include lower communication overhead, especially when communicating small items, and ease of programming when the communication pattern is complex. Most implementations of shared memory model involve threads, one of the most commonly used standard is Open Multi-Processing (OpenMP) [188].
This is an application programming interface (API) which provides the specifications for a set of environment variables, library routines and compiler directives that can be used to specify multi-platform shared-memory parallel programming in C/C++ and Fortran.

**Message Passing Communication Model:** Each processor has its own memory, and communication between the processors is explicit: the data exchange between the processors require explicit messages to be passed from those who have the data to those who require the data. These communications can be synchronous or asynchronous. For example, all collective communications such as Reduce, Scatter and Gather involve *global communication* and the processes are synchronised. However, global communications do not have good parallel scaling and can become a bottleneck in numerical algorithms. Asynchronous communications allow processes to overlap communication with computation, this is often the key to scalable parallel numerical algorithms.

Hardware for message passing model is simpler and more cost effective (see Section 1.3.3), especially compared to scalable shared memory. Explicit communications make it easier to identify where communication is happening, resulting in a code that may be simpler to understand and with a better control of granularity (the ratio of computation to communication, see Section 1.3.5). By contrast, in the shared memory model it can be difficult to identify parallel regions in the code. The most common standard used in message passing parallel programming is the Message Passing Interface (MPI) [179]. This library is considered the industry standard.

It is possible to combine the two models of communication, for example, by using MPI for communication between nodes with OpenMP for communication between cores on the same node, this is suitable for a hybrid shared/distributed architecture.

All computations performed in this thesis is done on the Computational Shared Facility (aka Danzek or CSF), a High Performance Computing (HPC) cluster at the University of Manchester [235]. The Danzek consists of Nvidia GPU, AMD and Intel nodes, connected with high speed InfiniBand. Each node is a SMP system (UMA), with NUMA across nodes. Both MPI and OpenMP are supported.

### 1.3.5 Designing Efficient Parallel Algorithms

In this section we discuss a few aspects to be considered when designing parallel algorithms.
Load balancing: Refers to the distribution of work among multiple computing resources. The aim is to optimise resource use by keeping all processes busy during the program execution. This can be achieved with equal distribution of the tasks either by uniform domain decomposition among the processors (if the amount of work required to process each subset of data is roughly the same), or by dynamic work scheduling, where tasks are assigned to processes at run time (e.g. the graph-based dynamic task scheduling used in the matrix algebra library MAGMA [32]).

Granularity: The ratio of computation to communication. There are two cases: a low computation to communication ratio is referred to as fine-grain parallelism, while a high ratio is referred to as coarse-grain parallelism. Fine-grain parallelism facilitates load balancing, which can be harder to achieve with coarse-grain parallelism. However, if the granularity of parallelism is too fine, then the overhead of process synchronisation and communication may outweigh the benefits of load balancing resulting in an increase of the overall execution time.

Scalability: In the context of parallel programming, scalability refers to the performance of a program when executed in a parallel setting with an increasing number of execution units. A common measure of parallel performance is the speedup, defined as a ratio of the wall-clock time of the serial execution to the wall-clock time for parallel execution. If one can identify the parallel and serial portions of the code, then the theoretical speedup of an algorithm is given by Amdahl’s law [193, p. 47]. However, in this thesis, we use the notion of parallel efficiency, which can be derived from either strong scalability or weak scalability. A program is strongly scalable if the wall-clock time decreases linearly with the increase of execution units for a fixed problem size. Likewise, a program is weakly scalable if the execution time remains constant as the number of execution units is increased while keeping a fixed problem size per execution unit. We formally define these quantities in Chapter 7.

Efficient parallel implementation assumes a good match between the underlying algorithm and the target parallel architecture. For shared memory architectures, data transfer between processes is relatively cheap but the contention for shared resources can negatively impact the algorithm performance. In the case of FE computations (assuming domain decomposition of the problem), the performance of the assembly phase may be negatively impacted if the ratio of memory access to computation is large, causing the issue of shared resource contention. The performance may be improved on NUMA microprocessors, but for large scale simulations, the use of main
memory is inevitable. In the solution phase, the use of preconditioned iterative linear solvers typically involve vector inner products (IP), sparse matrix-vector products (SpMV) and possibly sparse matrix-matrix multiplications (SpGEMM), which require communication between processes. These operations can easily be implemented on SMP systems. The performance of numerical algorithms on SMP systems can be significantly improved with cache-oblivious algorithms \[198\], this promotes a more effective use of CPU cache by restructuring the algorithms.

Due to (hardware) scalability and economical implications discussed in Section 1.3.3, distributed memory systems with accelerators (i.e. GPU) are the trend in HPC for the foreseeable future. Thus, most highly scalable software are developed for this type of architecture to allow researchers to harness HPC resources. This typically involve restructuring algorithms to reduce global communications and using non-blocking asynchronous communication. This is not a simple task and remains to be a very active and fertile area of research. For large scale FE simulations, the majority of the execution time is spent in the iterative linear solver, where the main computational tasks are IPs, SpMV products and SpGEMM multiplication. Thus, for scalable FE software, asynchronous algorithms and communication avoiding algorithms for large sparse systems must be developed. In Section 3.5 we discuss some of the techniques used for the development of highly scalable parallel solvers.

### 1.4 Modelling Software

We review some software commonly used in CFD modelling. We begin with general CFD modelling software capable of simulating a wide range of physical problems. Then we introduce software developed for particular classes of problems.

One of the most widely used commercial CFD modelling software is ANSYS Fluent \[6\]. It is primarily based on FV methods and is capable of tackling a wide range of physical problems on complex geometries, such as fluid-structure interaction problems, multiphase flows, turbulent modelling and particle tracking. The graphical user interface and the use of computer aided design (CAD) facilitates the generation of complex geometries and three-dimensional hybrid meshes (such as combining hexahedral and tetrahedron elements). Parallelism is achieve with both MPI and OpenMP, along with GPU acceleration support. A free and open source alternative to ANSYS Fluent with the same capabilities is OpenFoam \[257\]. The code uses the FV method
for continuous problems, while for particle tracking it deploys DSMC and molecular dynamics (MD) methods. It is written in C++ and parallelised with MPI, with GPU support integrated using the cufflink-library \cite{52}.

Now we present some efficient and highly scalable software of solving particular class of fluid mechanic problems. We stress that this is not an exhaustive list, nor do we claim that they are ‘the best’ software for their particular application, but they are all open source, actively maintained and well documented. The LB3D \cite{158} is a parallel implementation of the Lattice-Boltzmann method for the simulation of amphiphilic fluid dynamics \cite{186}. It was developed by University College London (CCS UCL headed by P. V. Coveney), University of Stuttgart and Eindhoven University of Technology. It is written in Fortran 90 and parallelised using MPI. The code demonstrated linear speedup up to 294,000 cores \cite{101}. LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) \cite{197} is a molecular dynamics simulator maintained at the Sandia National Laboratories. It is primarily written in C++ (with some Fortran subroutines), uses MPI for parallel communication and have GPU support for many subroutines. For spectral element method, Nektar++ \cite{40} is a spectral/hp element framework designed to facilitate the development of high performance scalable solvers for a wide range of PDEs including (but not limited to) the compressible and incompressible NSE, reaction-diffusion-advection equations \cite{41} and shallow water equations \cite{76}. It is implemented in C++ and parallelised with MPI.

All computations presented in this thesis were implemented in oomph-lib \cite{122}, a free and open source parallel, object-oriented, multi-physics FE library written in C++. It includes a large number of examples in structural and fluid mechanics, as well as multi-physics problems in fluid-structure interaction. The library implements various types of finite elements, which can be easily extended. It implements a state-of-the-art numerical methodology for parallel simulation of physical systems using a fully coupled (monolithic \cite{129}) discretisation by the FEM, adaptive time stepping methods, adaptive grid refinement, moving meshes, and efficient preconditioned iterative solvers.

There are many other, possibly hundreds, alternative FEM libraries. We discuss the capabilities and limitations of only a few, selected due to their rich features and well written documentation.

The FEniCS Project \cite{2}, is a collection of free software developed for the automated solution of DEs by the FEM \cite{170}. To achieve this, several core components were developed alongside FEniCS, for example, their UFL (Unified Form Language) is used
for specifying the element discretisation and DEs. The UFL code is then compiled by their FFC (FEniCS Form Compiler), generating the low-level C++ functions referred to as UFC (Unified Form-assembly Code), which can be interpreted using Instant, a Python module for inlining C and C++ code in Python. Currently the only available elements are line intervals, triangles, tetrahedra. Generic preconditioners are implemented, but there is currently no functionality for block preconditioning.

Elmer [73] is a software suite for multi-physics simulations, it is implemented in Fortran 2003 with MPI for parallelisation. Models of structural mechanics, electromagnetics, fluid dynamics, heat transfer and acoustics are included, with over 400 case studies [200]. Generic preconditioners exist, and block preconditioning has been demonstrated for the Stokes equation [171], but is built into their ParStokes module, a dedicated solver for ice flow modelling.

deal.II [14] is implemented in C++ and parallelised using MPI. The features and capabilities of deal.II are most similar to that of oomph-lib. It includes functionalities for block preconditioning which is decouple from the PDEs and solvers, but seems to suffer from flexibility issues which we will explore in Chapter 5. We stress that our experience with deal.II is very limited. Unlike oomph-lib, it only support line intervals, triangles and tetrahedra.

1.5 Thesis Overview

In Chapter 2 we introduce the Navier-Stokes equations starting with fundamental principles of physics and provide background on their finite element discretisation which results in a sparse nonlinear system we aim to solve iteratively.

Chapter 3 discusses different methods of solving linear systems. We motivate the use of iterative methods by discussing the direct solution approach and highlight the major drawbacks of sparse direct solvers when applied to the factorisation of large sparse coefficient matrices that arise in finite element method discretisation. We then discuss a particular class of iterative solvers for sparse linear systems - the Krylov subspace methods. We examine the reasons for their inefficiencies when applied to the solution of finite element discretisations. This motivates the concept of preconditioning as a remedy and we introduce block preconditioners as an efficient tool for improving the convergence characteristics of Krylov solvers in cases when systems of partial differential equations are solved. We review the existing efficient block preconditioning
techniques for the Navier-Stokes equations for problems with standard Dirichlet and Neumann boundary conditions.

In Chapter 4 we focus on a particular class of flow problems where the boundaries are not aligned with the Cartesian coordinates and impose tangential flow or parallel outflow boundary conditions using Lagrange multipliers. This leads to linear systems with an augmented block structure compared to a standard problem discussed in Chapter 2. We present a novel robust preconditioner for such problems. The preconditioner we developed is a special case of the augmentation preconditioning methodology described in [97]. We perform the analysis of the spectrum of the preconditioned augmented system, showing tight clustering of eigenvalues in the exact case. We then discuss the simplifications that lead to a nearly-optimal implementation in terms of the execution time. This implementation is based on a black-box optimal preconditioners for the standard Navier-Stokes problem and black-box solvers for scalar problems (such as AMG).

The new preconditioning algorithms were implemented using oomph-lib’s parallel block preconditioning framework (BPF), a middleware software which facilitates the efficient and rapid development of parallel block preconditioning algorithms for finite element discretisations. Owing to the complexity of the block structure of the new augmentation preconditioner described in Chapter 4, the initial version of the BPF [180, pp. 200–203] had to be significantly upgraded to give the functionality required to efficiently implement the new parallel block preconditioning algorithms. An overview of the development of the block preconditioning framework, both past and present, is the topic of Chapter 5.

Numerical results are presented in Chapter 6 for a range of benchmark test problems, demonstrating robust behaviour in cases where theoretical analysis was not possible, confirming effectiveness of a preconditioned Krylov solver in cases of practical interest. We also present the results for an inexact version of the preconditioner which leads to a nearly-optimal execution time scaling of the iterative solver. The parallel scalability of the new version parallel block preconditioning framework is demonstrated in Chapter 7 for three-dimensional transient problems with an inexact version of the preconditioner. The new version of the framework allow the parallel implementation of preconditioners that exhibit good weak and strong scaling.
1.6 Main Project Contributions

The main contributions of this research project can be broadly categorised into three sections, summarised below:

**Novel preconditioning methodology and analysis**

In this work we have developed a novel and robust preconditioner for the incompressible Navier-Stokes equations with non-standard (weakly imposed) boundary conditions through Lagrange multipliers. Such boundary conditions for the incompressible Navier-Stokes equations are important as they facilitate the finite element discretisation of problems posed over arbitrarily shaped domains, thus making it applicable to real-life problems.

We have demonstrated analytically a good clustering of eigenvalues for the preconditioned operator in the exact case. This result is supported by numerical evidence from the calculation of eigenvalues of the preconditioned operator.

Our novel preconditioning approach is based on a general augmentation preconditioning applied to problems with constraints imposed by Lagrange multipliers. For this reason, the preconditioning methodologies documented in this thesis may be extended to other types of constrained PDEs with constraints imposed with Lagrange multipliers. We hope that the approach developed in this research will be generalised for other similar problems.

**Development of an optimal inexact solver**

Serial numerical computations show that an inexact version of the new preconditioner have nearly-optimal cost in terms of wall-clock execution time. To achieve this, we have obtained experimentally the sets of robust parameters for the inexact solver which work well for two and three spatial dimensions on a wide range of problems.

**Significant upgrade of the parallel Block Preconditioning Framework (BPF)**

The initial version of oomph-lib’s parallel BPF was developed by Muddle [180, pp. 94–100]. The aim was to develop a general purpose software middleware to enable rapid development of parallel block preconditioners within oomph-lib. The initial version is shown in [180, pp. 101–109] to have good weak scalability.

However, additional functionalities were required for the rapid implementation of the augmentation preconditioner developed in this project. There were two main challenges in this context: the new functionalities must not significantly impact the
performance (both in serial and parallel); and they need to retain backward compatibility with the initial version of the BPF. The modification were not trivial and required the partial reimplementaion of the whole framework. However, the flexibility and functionality of the BPF have been significantly upgraded. Furthermore, the new functionalities exhibit negligible overhead in terms of wall-clock time.

Parallel computations with the newly implemented block preconditioning framework demonstrate good weak and strong scaling. In particular, we have shown that the overhead in terms of execution times from using the block preconditioning framework is either negligible or have good parallel scaling.
Chapter 2

The Incompressible Navier-Stokes Equations

In this chapter we provide a background survey of the incompressible Navier-Stokes equations and introduce their numerical solution by the Finite Element Method.

2.1 Derivation and Assumptions

The Navier-Stokes equations (NSE) model fluid motion as a continuum, where the physical properties (such as velocity and pressure) have finite values and are distributed throughout space, otherwise statistical molecular approaches may be more appropriate. We assume that the thermodynamic state of the fluid is at equilibrium and can be described using constant temperature, i.e. we assume that the fluid is isothermal (that is, as the system evolves, the temperature remains constant in time and homogeneous in space). Under these assumptions, conservation of energy is not required. In this work we are only concerned with the conservation of momentum and the conversation of mass.

For the remainder of this section we derive the incompressible NSE and state some assumptions and simplifications. By considering a physical quantity of a fluid flow field $\varphi$ (such as density) defined over a control volume $\Omega \in \mathbb{R}^d$, (where $d$ is the spatial dimension of the control volume) the Reynolds transport theorem [205, pp. 12–13] states that the sum of the changes of the physical quantity must be equal to the inflow/outflow of that quantity through the boundary $\partial \Omega$ of $\Omega$ plus the amount
CHAPTER 2. THE INCOMPRESSIBLE NAVIER-STOKES EQUATIONS

created by sources or consumed by sinks in $\Omega$. This can be expressed as

$$\frac{d}{dt} \int_{\Omega} \varphi \, d\Omega = - \int_{\partial \Omega} \varphi \mathbf{u} \cdot \mathbf{n} \, dS - \int_{\Omega} s \, d\Omega, \quad (2.1.1)$$

where $\mathbf{u}$ is the velocity flow field of the fluid, $\mathbf{n}$ is the outwards unit normal vector at each point of the boundary $\partial \Omega$, and $s$ represents total sources/sinks in $\Omega$. By applying the divergence theorem \[231, p. 1061\], which transforms the surface integral into a volume integral, (2.1.1) becomes

$$\frac{d}{dt} \int_{\Omega} \varphi \, d\Omega = - \int_{\Omega} \nabla \cdot (\varphi \mathbf{u}) \, d\Omega - \int_{\Omega} s \, d\Omega.$$

We use Leibniz’s rule \[261, p. 186\] to move the time derivative inside the integral and by combining all the integrals we have

$$\int_{\Omega} \left( \frac{\partial \varphi}{\partial t} + \nabla \cdot (\varphi \mathbf{u}) + s \right) \, d\Omega = 0. \quad (2.1.2)$$

The integral in (2.1.2) being equal to zero for any (infinitely small) control volume $\Omega$ implies that the integrand must be equal to zero everywhere, therefore we have

$$\frac{\partial \varphi}{\partial t} + \nabla \cdot (\varphi \mathbf{u}) + s = 0. \quad (2.1.3)$$

Equation (2.1.3) is a general continuity equation from which we can precisely formulate the conservation of mass and momentum.

Conservation of mass is expressed by substituting in density $\rho$ as the physical quantity of interest in (2.1.3) and set $s = 0$, we obtain

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \quad \text{in } \Omega. \quad (2.1.4)$$

In this work, we assume that each volume of fluid have the same density. This may not always be the case, in stratified flows such as in oceans (where temperature and salt content is a function of depth) or mixtures of different fluids (of different density), the density of adjacent volumes of fluid may vary. The characteristics of an incompressible flow is related to the Mach number, $Ma \equiv |\mathbf{u}|/c$, where $|\mathbf{u}|$ is the fluid’s speed and $c$ is the speed of sound in the fluid. The main criterion for an incompressible flow is that
Mach number is low (Ma → 0) \[191\), p. 205\]. If Ma = 0 then the density is constant within each volume of fluid, hence

\[ \nabla \cdot \mathbf{u} = 0 \quad \text{in } \Omega. \]  

(2.1.5)

In real fluids, the Mach number is never zero, however if Ma ≤ 0.3 then the changes in density are negligible and the incompressibility assumption \[2.1.5\] is a good approximation to model a compressible flow \[99\), pp. 3–4\]. In this work we model fluid flows under the assumption \[2.1.5\] i.e. in an incompressible fluid the amount of fluid that enters any volume is equal to the amount that exits it. Where there is no ambiguity, we will omit the word ‘incompressible’ when referring to the NSE.

Conservation of momentum may also be considered by applying \[2.1.3\] to the physical quantity mass flux $\rho \mathbf{u}$ (a product of the density $\rho$ and flow velocity $\mathbf{u}$ \[205\), p. 19\]), we obtain

\[ \frac{\partial}{\partial t}(\rho \mathbf{u}) + \nabla \cdot (\rho \mathbf{u} \mathbf{u}) = \mathbf{s}, \]  

(2.1.6)

where $\mathbf{s}$ represent the forces acting on $\Omega$ (which we will define later in this section), it is positive due to Newton’s second law applied to the fluid in $\Omega$, which states: *The rate of change of momentum of the fluid in $\Omega$ is equal to the sum of the forces on $\Omega$. Expanding the derivatives in \[2.1.6\] by the chain rule and after rearrangement we have

\[ \mathbf{u} \left( \frac{\partial \rho}{\partial t} + \mathbf{u} \cdot \nabla \rho + \rho \nabla \cdot \mathbf{u} \right) + \rho \left( \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = \mathbf{s}. \]  

(2.1.7)

Noting that $\nabla \cdot (\rho \mathbf{u}) = \mathbf{u} \cdot \nabla \rho + \rho \nabla \cdot \mathbf{u}$, we can rewrite \[2.1.7\] as

\[ \mathbf{u} \left( \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) \right) + \rho \left( \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = \mathbf{s}. \]  

(2.1.8)

By mass conservation \[2.1.4\] the first term in \[2.1.8\] is equal to zero, therefore

\[ \rho \left( \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = \mathbf{s}. \]  

(2.1.9)

The expression in \[2.1.9\] is known as the material derivative (also called the substantial derivative) and is usually denoted by

\[ \frac{D(\cdot)}{Dt} \equiv \frac{\partial(\cdot)}{\partial t} + (\mathbf{u} \cdot \nabla)(\cdot). \]  

(2.1.10)

\[1\] If $\varphi$ in \[2.1.3\] is a vector, $\varphi$, then the vector-vector product in the second term will be an outer product, i.e. $\varphi \mathbf{u} = \varphi \otimes \mathbf{u} = \varphi \mathbf{u}^T$. 
The material derivative expresses the rate of change of a quantity in time as we follow a selected particle in a velocity field, therefore (2.1.9) represents Newton’s second law (mass × acceleration = force). There are two kinds of forces acting on a control volume, body forces which acts on the bulk of the fluid in the control volume (such as gravity) and surface forces which acts on the boundary surface (surface stresses). By examining the forces acting on a small cube of fluid, see for example [191, pp. 78 – 82], it can be shown that

\[ \rho \left( \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} \right) = \nabla \cdot \mathbf{\sigma} + \mathbf{f}, \]

(2.1.11)

where \( \mathbf{f} \) represents the body force and \( \mathbf{\sigma} \) is the Cauchy stress tensor, given in three spatial dimensions as

\[
\mathbf{\sigma} = \begin{bmatrix} \sigma_1 & \tau_{12} & \tau_{13} \\ \tau_{21} & \sigma_2 & \tau_{23} \\ \tau_{31} & \tau_{32} & \sigma_3 \end{bmatrix},
\]

(2.1.12)

where \( \sigma_i \) are the normal stresses and \( \tau_{ij} \) are the shear stresses. We can split up the stress tensor into two terms by subtracting out the pressure as shown in [191, p. 83],

\[
\mathbf{\sigma} = -\begin{bmatrix} \rho & 0 & 0 \\ 0 & \rho & 0 \\ 0 & 0 & \rho \end{bmatrix} + \begin{bmatrix} \sigma_1 + p & \tau_{12} & \tau_{13} \\ \tau_{21} & \sigma_2 + p & \tau_{23} \\ \tau_{31} & \tau_{32} & \sigma_3 + p \end{bmatrix} = -p \mathbf{I} + \mathbf{\tau},
\]

(2.1.13)

where \( \mathbf{I} \) is the \( 3 \times 3 \) identity matrix and \( \mathbf{\tau} \) the viscous stress tensor (also referred to as the deviatoric stress tensor [191, p. 113]). Substituting (2.1.13) into (2.1.11) yields a general form of the momentum equations:

\[
\rho \left( \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} \right) = -\nabla p + \nabla \cdot \mathbf{\tau} + \mathbf{f}.
\]

(2.1.14)

A Newtonian fluid is usually characterized by the fact that the stress tensor \( \mathbf{\sigma} \) is a linear function of the strain rate tensor \( \mathbf{\epsilon} \), this is known as Newton’s viscosity law [191, pp. 111–114]. By definition the strain rate tensor is

\[
\mathbf{\epsilon} = \frac{1}{2} (\nabla \mathbf{u} + (\nabla \mathbf{u})^T) = \frac{1}{2} \begin{bmatrix} 2\frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} + \frac{\partial u_2}{\partial x_1} & \frac{\partial u_1}{\partial x_3} + \frac{\partial u_3}{\partial x_1} \\ \frac{\partial u_2}{\partial x_1} + \frac{\partial u_1}{\partial x_2} & 2\frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} + \frac{\partial u_3}{\partial x_2} \\ \frac{\partial u_3}{\partial x_1} + \frac{\partial u_1}{\partial x_3} & \frac{\partial u_3}{\partial x_2} + \frac{\partial u_2}{\partial x_3} & 2\frac{\partial u_3}{\partial x_3} \end{bmatrix}.
\]

(2.1.15)
In this thesis, only Newtonian fluids are considered, for which the viscous stress tensor is defined by

\[ \tau = \lambda \text{Trace}(\varepsilon) + 2\mu \varepsilon, \]  

(2.1.16)

where \( \lambda \) and \( 2\mu \) are referred to as the second and first viscosity coefficients, respectively. The second viscosity coefficient is associated with viscous effects due to changes in the volume of the fluid, this is not relevant for incompressible fluids since \( \text{Trace}(\varepsilon) = \nabla \cdot \mathbf{u} = 0 \). In this thesis we refer to \( \mu \) as the dynamic viscosity. Assuming that \( \mu \) is constant, the divergence of the viscous stress tensor is

\[ \nabla \cdot \tau = \mu \nabla \cdot \varepsilon = \mu \begin{bmatrix} \nabla^2 u_1 + \frac{\partial}{\partial x_1} (\nabla \cdot \mathbf{u}) \\ \nabla^2 u_2 + \frac{\partial}{\partial x_2} (\nabla \cdot \mathbf{u}) \\ \nabla^2 u_3 + \frac{\partial}{\partial x_3} (\nabla \cdot \mathbf{u}) \end{bmatrix} = \mu \nabla^2 \mathbf{u}. \]  

(2.1.17)

We can now write (2.1.14) as

\[ \rho \left( \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f}. \]  

(2.1.18)

From (2.1.18) we see that fluid moves from areas of high pressure to areas of low pressure, following \(-\nabla p\). The body force \( \mathbf{f} \) acts on every volume of fluid. This is commonly due to gravitational forces but it can also represent electromagnetic fields or variable body forces. In viscous fluids, each fluid particle is also acted on by shear stress due to surrounding particles moving at different speeds, hence the viscosity effect is proportional to \( \nabla^2 \mathbf{u} \). The strain rate describes the rate of change of deformation (strain) of the liquid. The resistance to deformation is measured by the viscosity (for example, honey has a much higher viscosity than water). The ratio of the dynamic viscosity and the density of the fluid is referred to as the kinematic viscosity (also called momentum diffusivity [20, p. 59]), denoted by \( \nu = \mu/\rho \). In Newtonian fluids, the relationship between the strain rate and viscous stress is linear (common examples include air and water). Fluids where this relationship is not linear are termed non-Newtonian. Examples include ketchup where the viscosity decreases as the shear stress increases, so the fluid resists motion at slow rates of deformation but will flow freely at high rates of deformation. Another common example is oobleck where the fluid resists flow at high rates of deformation but flows freely at low rates, for example custard or corn starch in water, it is possible for a person to walk across a bath tub of custard if the individual walks quickly enough to provide each step with enough force to cause the fluid to thicken.
CHAPTER 2. THE INCOMPRESSIBLE NAVIER-STOKES EQUATIONS

2.2 The Boundary Value Problem

In this section we introduce the NSE (2.1.5) and (2.1.18) in Cartesian index notation and define an appropriate set of boundary conditions (BCs) (described in Section 2.2.2). The discretisation using the finite element method (FEM) (described in Section 2.3) leads to the natural component-wise blocking of the resulting linear algebra problem. We will use symbolic notation where it is advantageous to do so (i.e. for notational simplicity). In their dimensional form (denoted by the asterisk superscripts) the NSE in Cartesian coordinates \((x^*_i; i = 1, 2, 3)\) are given by the momentum equations

\[
\rho \left( \frac{\partial u^*_i}{\partial t^*} + u^*_j \frac{\partial u^*_i}{\partial x^*_j} \right) = -\frac{\partial p^*}{\partial x^*_i} + B^*_i(x^*_j, t^*) + \rho G^*_i + \mu \frac{\partial}{\partial x^*_j} \left( \frac{\partial u^*_i}{\partial x^*_j} + \frac{\partial u^*_j}{\partial x^*_i} \right),
\] (2.2.1)

and the continuity equation

\[
\frac{\partial u^*_j}{\partial x^*_j} = 0,
\] (2.2.2)

where we have used the Einstein summation convention for the repeated index \(j\) (see [191, pp. 28–42] and [116] for this notation usage). In (2.2.1) (2.2.2) \(u^*_i\) denotes the Cartesian components of velocity vector, \(p^*\) denotes the pressure field, \(t^*\) is the time, \(\rho\) the fluid density and \(\mu\) is the dynamic viscosity (sometimes referred to as the molecular viscosity). The body force (denoted by \(f\) in (2.1.18)) is split into two components: a constant vector \(\rho G^*_i\) which typically represent the gravitational force, and \(B^*_i(x^*_j, t^*)\) is a variable body force (e.g. external forces acting on the fluid such as shaking). In this work we assume zero body forces, therefore the momentum equations (2.2.1) become

\[
\rho \left( \frac{\partial u^*_i}{\partial t^*} + u^*_j \frac{\partial u^*_i}{\partial x^*_j} \right) = -\frac{\partial p^*}{\partial x^*_i} + \mu \frac{\partial^2 u^*_i}{\partial x^*_j^2}.
\] (2.2.3)

From (2.1.17) we see that the assumption \(\nabla \cdot \mathbf{u} = 0\) allows us to further simplify the viscous terms of (2.2.3) to

\[
\rho \left( \frac{\partial u^*_i}{\partial t^*} + u^*_j \frac{\partial u^*_i}{\partial x^*_j} \right) = -\frac{\partial p^*}{\partial x^*_i} + \mu \frac{\partial^2 u^*_i}{\partial x^*_j^2}.
\] (2.2.4)

We refer to the pair (2.2.3) and (2.2.2) as the stress divergence viscous term (SDVT) of the NSE (denoted by SDVT-NSE), and the pair (2.2.4) and (2.2.2) as the simple viscous term (SVT) of the NSE (denoted by SVT-NSE). The SDVT-NSE form is required for
problems involving traction BCs \[117\] or problems with free-surfaces \[113\].

### 2.2.1 Non-dimensional Form

The advantage of introducing the non-dimensional form of the equations is in replacing a number of dimensional parameters by a smaller number of dimensionless parameters which connect the intrinsic physical quantities in the system, making their form simpler in the process.

We non-dimensionalise the NSE using problem-specific reference quantities for the velocity, \( U \), length \( L \), and time \( T \), to obtain

\[
\begin{align*}
u^* &= U u^i, \\
x^* &= L x^i, \\
t^* &= T t \\
p^* &= \frac{\mu_{ref} U}{L} p,
\end{align*}
\]

where the pressure is non-dimensionalised on the viscous scale. The parameters \( \mu_{ref} \) and \( \rho_{ref} \) (used below) are reference values for the dynamic viscosity and density of the fluid, respectively. For incompressible single-fluid flows (Ma \( \rightarrow 0 \)), the reference parameters \( \rho_{ref} \) and \( \mu_{ref} \) are identical to the density \( \rho \) and dynamic viscosity \( \mu \) of the fluid. The non-dimensional form of the NSE is then given by

\[
R_p \rho \text{Re} \left( S t \frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} \right) = - \frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j} R_\mu \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right),
\]

where

\[
\text{Re} = \frac{UL \rho_{ref}}{\mu_{ref}} \quad \text{and} \quad \text{St} = \frac{L}{UL}
\]

are the Reynolds number [191, p. 202] and the Strouhal number [191, p. 351], respectively. The parameters \( R_p \) and \( R_\mu \) are ratios of the fluid’s density and its dynamic viscosity relative to the reference density and viscosity values used to form the non-dimensional parameters. Since both fluid’s density and the dynamic viscosity are constant and \( \rho = \rho_{ref}, \mu = \mu_{ref} \), we have \( R_p = R_\mu = 1 \). It is shown in [191, pp. 201–207] that in the limit Ma \( \rightarrow 0 \), the parameters related to the transport properties are constant and the continuity and the momentum equations are decoupled from the energy equations. This means we may solve for velocity and pressure unknowns without involving the temperature or the energy equations. The Reynolds number is a dimensionless parameter defined as the ratio of inertial to viscous forces. It plays a key
role in incompressible flows, as it is often used to characterise different flow regimes and dynamic similarities between different cases of fluid flows. For Stokes flow, Re = 0. For high Reynolds number the flow is dominated by inertial forces and is likely to be turbulent. Flows in this regime often produce chaotic eddies and vortices, an example is the model of wind through a wind tunnel [201]. Low Reynolds number will often produce laminar flows, where viscous forces are dominant. Flows in this regime are characterised by constant and smooth fluid motion. Example includes blood flow in a blood vessels [269, 239] and microfluidic flows [233, p. 92]. The Reynolds number governing the transition between laminar and turbulent flow is called the critical Reynolds number [210]. The critical Reynolds number is different for different geometries and different flows. In this project we only model laminar flows. The Strouhal number is a dimensionless parameter typically associated with vortex shredding, see [191, p. 356] and [260].

2.2.2 Boundary Conditions

In order to obtain unique solution of the NSE posed on a two- or three-dimensional domain \( \Omega \subset \mathbb{R}^d \) \((d = 2, 3)\) from a set of infinitely many functions that satisfy the original problem (2.2.5) and (2.2.6), but differ by a constant, we need to supplement the NSE with a suitable set of BCs. To review the BCs that are commonly used, we introduce two main types of fluid problems: the flow-through problems, see Figure 2.1 (left), and the enclosed flow problems, see Figure 2.1 (right).

In flow-through problems the boundary \( \partial \Omega \) of \( \Omega \) consists of three disjoint parts: the inflow boundary \( \partial \Omega_I \), the outflow boundary \( \partial \Omega_O \) and the characteristic boundary \( \partial \Omega_C \). In the enclosed flow problems the entire boundary is characteristic. At the inflow and characteristic boundaries, Dirichlet BCs are typically imposed (i.e. the velocity is prescribed). On \( \partial \Omega_I \), a nonzero inflow is prescribed in a physically realistic manner, for
2.2. THE BOUNDARY VALUE PROBLEM

For example, the parabolic inflow profile: if $\Omega$ is a unit square $[0, 1]^2$ with the configuration as in Figure 2.1 (left), the appropriate BCs on $\partial \Omega_I$ would be $u_1 = x_2(1 - x_2)$, $u_2 = 0$. If a time-dependent problem is considered, care must be taken when imposing the inflow BC. One needs to avoid imposing an impulsive start which can trigger non-physical oscillations in the numerical solution [71]. On the characteristic boundary $\partial \Omega_C$ we typically prescribe the no-slip BC, given by $u_i = 0$, $i = 1, 2$. Other types of Dirichlet BCs on $\partial \Omega_C$ are also possible, for example the no-penetration BC is obtained when setting $\mathbf{u} \cdot \hat{n} = 0$, where $\hat{n}$ is the outward unit normal vector on $\partial \Omega_C$. In enclosed flow problems the application of Dirichlet BCs leads to pressure being unique up to a constant [68, pp. 223–224]. At the outflow boundary $\partial \Omega_O$ the most common approach is to apply the ‘natural’ BCs which appears in the formulation of the NSE upon the application of the divergence theorem to the momentum equations (2.2.5) [99, p. 381]:

$$- p \hat{n}_i + \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \hat{n}_j = T_i \quad \text{on} \quad \partial \Omega_O, \quad (2.2.7)$$

In (2.2.7) $T_i$ is the $i$th component of the traction force vector, $\mathbf{T} = \sigma \hat{n}$, applied to the fluid at the boundary $\partial \Omega_O$, and $\hat{n}_i$ is the $i$th component of the outward unit normal vector $\hat{n}$ to the boundary $\partial \Omega_O$. In practice, the true values of $T_i$ are not known and it can be a difficult to apply suitable outflow BCs due to the tight coupling of the normal velocity component and pressure, which satisfies the incompressibility constraint (2.1.5) at the outlet. Techniques for appropriate natural outflow BCs are discussed in [99, pp. 286–289] and [219]. The most established and commonly used natural BC is the so-called ‘do-nothing’ BC, given by $T_i = 0$ (i.e. the homogeneous natural BC). However, naïve application of this approach may result in undesirable and non-physical behaviour of the computed flow across $\partial \Omega_O$, such as inwards bending velocity see [86, pp. 292–293]. Furthermore, it can be shown that the problem is not well-posed [34]. For straight/planar outflow boundaries which are aligned with the Cartesian coordinates, this non-physical flow behaviour can be remedied by applying mixed BCs, where only some of the velocity components are prescribed. At each point on $\partial \Omega_O$, the velocity components orthogonal to the desired flow direction are set to zero, and the do-nothing condition is applied for the remaining velocity component(s). In the normal direction ($x_i$ to $\partial \Omega_O$), where the do-nothing BC is applied, (2.2.7) reduces to $2 \frac{\partial u_i}{\partial x_i} - p = 0$. In oomph-lib, this is referred to as the parallel, axially-traction-free outflow condition [110] and is well-posed [219]. In order to generalise this approach, the constraint $\mathbf{u} \cdot \hat{t}_s = 0$, for $s = 1, 2$, is imposed on $\partial \Omega_O$, where $\hat{t}_s$ are
the unit tangent vector(s) at specified points of \( \partial \Omega_O \), notice that two tangent vectors are required for problems in three spatial dimensions. By contrast, an impermeable boundary condition can be imposed by the constraint \( \mathbf{u} \cdot \mathbf{n} = 0 \). In this thesis we refer to \( \mathbf{u} \cdot \mathbf{n} = 0 \) as the no-penetration constraint and \( \mathbf{u} \cdot \mathbf{t}_s = 0 \) as the parallel flow constraint. We discuss further the BCs imposed by this technique in Chapter 4.

In practice it is possible to enforce Dirichlet outflow BCs as well, however, if the volumetric fluxes at the inflow and the outflow do not match, the incompressibility condition \((2.2.2)\) will be violated. The do-nothing outflow condition automatically satisfies the incompressibility constraint. There exists a number of attempts in the literature to improve the accuracy of the outflow condition \((2.2.7)\) (at increased complexity), see for example [99, pp. 380–403].

To summarise, denote the boundary of \( \Omega \) by \( \partial \Omega \), and the set of \( i \)th velocity component of the boundary by \( \partial \Omega_i \) for \( i = 1, 2, 3 \). The \( i \)th component of velocity at a boundary \( \partial \Omega \) can have only one type of BC. Then we have \( \partial \Omega_i = \partial \Omega_{Di} \cup \partial \Omega_{Ni} \) and \( \partial \Omega_{Di} \cap \partial \Omega_{Ni} = \emptyset \), where \( \partial \Omega_{Di} \) and \( \partial \Omega_{Ni} \) represent the Dirichlet and Neumann part of the boundary for the \( i \)th velocity component, respectively. Then \( \bigcup_{i=1}^{d} \partial \Omega_{Di} = \partial \Omega_D \) is the Dirichlet boundary and \( \bigcup_{i=1}^{d} \partial \Omega_{Ni} = \partial \Omega_N \) is the Neumann boundary, where \( d = 2 \). For Dirichlet BCs we have

\[
\mathbf{u}_i = w_i, \quad \text{on} \quad \partial \Omega_{Di}, \quad \text{for} \quad i = 1, 2, 3, \quad (2.2.8)
\]

where \( w_i \) is the \( i \)th component of the prescribed velocity profile \( (w_i = 0 \text{ for the homogeneous case}) \). For natural (Neumann) BCs we have

\[
-p\mathbf{n}_i + \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \mathbf{n}_j = T_i, \quad \text{on} \quad \partial \Omega_{Ni}, \quad \text{for} \quad i = 1, 2, 3. \quad (2.2.9)
\]

For the do-nothing BC (no applied traction force) we have \( T_i = 0 \).

### 2.2.3 Initial Conditions

To complete the formulation of the problem, we need to set the initial conditions (ICs) for the transient case. We use the standard ICs for the NSE detailed in [99, pp.
for which the initial velocity must be incompressible everywhere,

\[ \frac{\partial u_i^0}{\partial x_j} = 0, \quad \text{in } \Omega \]  

(2.2.10)

and

\[ \hat{n}_i \cdot u_i^0 = \hat{n}_i \cdot w_i^0, \quad \text{on } \partial \Omega_D, \quad \text{for } i = 1, 2, \ldots, 3, \]  

(2.2.11)

where \( u_i \) is the \( i \)th component of a prescribed velocity profile and the superscript zero denotes the initial time. Equations (2.2.5) and (2.2.6) along with BCs (2.2.8) and (2.2.9) and ICs (2.2.10) and (2.2.11) form the boundary value problem we want to solve. The prescribed velocity profile in (2.2.11) should not involve sudden changes (for example, compulsive start defined by Heaviside step function) to trigger numerical instabilities in a time integrator [71].

### 2.3 Spatial Discretisation

In order to solve (2.2.5)–(2.2.6), together with BCs (2.2.8)–(2.2.9) and ICs (2.2.10)–(2.2.11) numerically, we need to derive a discrete form of it. This process of replacing the continuous problem by it’s discrete counterpart is called discretisation. Two main groups of methods for the numerical solution of partial differential equations (PDEs) are the finite difference methods (FDMs) and the Galerkin methods. Here we highlight some key points of these two groups, an in-depth discussion on numerical methods for PDEs can be found in [156, 102]. The comparative performance of the two groups of numerical methods is very problem specific. Comparative studies in terms of computational efficiency and numerical stability and accuracy of these two classes of numerical methods can be found in [207, 56, 194] and [85].

The FDM is based on local approximation of the partial derivatives in the PDE by finite differences obtained from the truncation of Taylor series, that is, we obtain an approximation to the PDE in the strong form. We highlight the following features of the FDM:

- The FDM is easy to implement — finite differences for higher order terms are relatively easy to derive. There exists many different well documented variations of the FDM, see [272, pp. 661–668] and [159].
- There is no mesh in the FDM. The approximate solution is obtained via a set of
discrete points; these points are usually spaced uniformly (tensor-product structure), as finite difference stencils are easy to obtain in such cases. This makes the FDM attractive for problem with simple domains. However, FDM can be implemented on non uniform stencils \cite{159, pp. 31–36} and cylindrical coordinates to cope with more complicated domains \cite{16}. One of the most popular FDM for the NSE is the marker and cell (MAC) method \cite{109}.

- The immersed interface method (IIM) is a modification of the FDM for PDEs involving interfaces (internal boundaries) and irregular domains, for more details see \cite{165}.
- For scalar elliptic problems, finite difference discretisation often give rise to banded matrices with constant coefficients. Such linear systems are easy to solve, e.g. fast Poisson solvers can solve them in logarithmic time \cite{8}, i.e. in the order of $O(N \log N)$, where $N$ is the size of the linear system.

By contrast, the Galerkin method discretises an integral formulation of the PDE known as the weak form. There are a few things to note here:

- The integral formulation does not depend on the mesh structure, the problem can be easily posed over an arbitrary domain $\Omega$, giving it greater flexibility than the basic FDM with little extra work.
- The integral formulation allows for a more natural treatment of Neumann BCs.
- The weak form is less restrictive in terms of admissible data due to the reduced requirement on the smoothness of the solution, thus could offer solutions to problems where there is no classical solution (solution of the pair \cite[(2.2.5)]{2.2.6}–\cite[(2.2.6)]{2.2.6}), for example, see \cite[p. 14]{68}.

Examples of the Galerkin method:

- Galerkin method of weighted residuals, this leads to the FEM \cite{270} and will be described in the sequel.
- Spectral element methods are similar to FEMs but uses a higher order degree of approximation. This leads a much smaller (but dense) linear system than that of FEMs and better accuracy is observed for comparable FEM computations. However, the assembly process, such as defining an isoparametric mapping (described in Section 2.4.2), is non-trivial. More details can be found in \cite{192}.
- Collocation methods discretise the domain into a finite number of collocation points. Then an approximate solution to a PDE is sought with the requirement that the equation is satisfied exactly at the collocation points. For more details see \cite[p. 217]{156}.  

2.3. SPATIAL DISCRETISATION

- Finite volume methods (FVMs) \[156, p. 219\] are similar to FEMs but have conservation properties which may be hard to achieve with the FEM, such as conservation of flux across finite volumes. This is a result of the construction of the method (both the test and the solution spaces are spanned by piecewise constant functions).

In the sequel, we replace the continuous problem (2.2.5)-(2.2.6) with BCs (2.2.8)-(2.2.9) and ICs (2.2.10)-(2.2.11) by a discrete counterpart using the Galerkin method.

2.3.1 Weak Formulation

We seek the weak formulation of the problem in a residual form. To do this, we move all terms in (2.2.5) to the left hand side

\[
\text{Re} \left( \text{St} \frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} \right) + \frac{\partial p}{\partial x_i} - \frac{\partial}{\partial x_j} \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) = 0, \quad \text{for } i = 1, 2 [, 3],
\]

(2.3.1)

multiply (2.3.1) by a suitable test function \( \psi_i \) in the \( i \)th coordinate direction, which we shall define formally later, and integrate over the domain \( \Omega \) to form the weighted residual:

\[
r_i^{(u)} = \int_\Omega \left( \text{Re} \left( \text{St} \frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} \right) - \frac{\partial \sigma_{ij}}{\partial x_j} \right) \psi_i \, d\Omega = 0,
\]

(2.3.2)

where

\[
\sigma_{ij} = \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - p \delta_{ij},
\]

(2.3.3)

is the stress tensor. Note that in (2.3.2), there is no implied summation over the repeated index \( i \), however, there is over \( j \). We adopt this convention in the sequel. In equation (2.3.3) \( \delta_{ij} \) is the Kronecker delta, defined as

\[
\delta_{ij} = \begin{cases} 
1, & \text{if } i = j \\
0, & \text{if } i \neq j
\end{cases}
\]

(2.3.4)

By applying the divergence theorem \[231, p. 1061\] to (2.3.2) we have

\[
r_i^{(u)} = \int_\Omega \left[ \text{Re} \left( \text{St} \frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} \right) \psi_i + \sigma_{ij} \frac{\partial \psi_i}{\partial x_j} \right] \, d\Omega - \int_{\partial \Omega} \sigma_{ij} \hat{n}_j \psi_i \, dS = 0,
\]

(2.3.5)
Equation (2.3.5) represents the weak form for the momentum equations. Likewise, the weak form for the continuity equation (2.2.6) is given by

$$ r^{(p)} = - \int_{\Omega} \frac{\partial u_j}{\partial x_j} \varphi \, d\Omega = 0, \tag{2.3.6} $$

where $\varphi$ is a suitable test function. In (2.3.6) the minus sign is commonly introduced to reflect the term $\int_{\Omega} -p \frac{\partial \psi_i}{\partial x_i} \, d\Omega$ in (2.3.5) to get a symmetric block form of the coefficient matrix. The pair (2.3.5), (2.3.6) represents weak form for the NSE.

We restrict the velocity test function $\psi_i$ to be zero on Dirichlet boundaries, $\Omega_{D_i}$. This means that on Dirichlet boundaries the term $\int_{\partial \Omega} \sigma_{ij} \hat{n}_j \psi_i \, dS$ vanishes from equation (2.3.5). In the case of Neumann BCs, we observe that (2.2.9) is already incorporated into the weak form, namely

$$ \sigma_{ij} \hat{n}_j = -p \hat{n}_i + \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \hat{n}_j, \tag{2.3.7} $$

enters directly into the weak form as

$$ \int_{\partial \Omega} \left( -p \hat{n}_i + \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \hat{n}_j \right) \psi_i \, dS. \tag{2.3.8} $$

For natural outflow boundaries, the traction-free (do-nothing) BC

$$ \sigma_{ij} \hat{n}_j = 0 \quad \text{on} \quad \partial \Omega_{N_i}, \tag{2.3.9} $$

is applied and the velocity components on those boundaries are regarded as unknown. Formally, the BCs are

$$ u_i = w_i \quad \text{on} \quad \partial \Omega_{D_i}, \tag{2.3.10} $$

and

$$ T_i = -p \hat{n}_i + \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \hat{n}_j = 0, \quad \text{on} \quad \partial \Omega_{N_i}, \tag{2.3.11} $$

for $i = 1, 2 [, 3]$ . In (2.3.10) $w_i$ is a prescribed velocity profile in the $i$th Cartesian direction. The ICs which corresponds to (2.2.10) and (2.2.11) are:

$$ \frac{\partial u_i^{[0]}}{\partial x_j} = 0, \quad \text{in} \quad \Omega, \tag{2.3.12} $$

$$ \hat{n}_i \cdot u_i^{[0]} = \hat{n}_i \cdot w_i^{[0]}, \quad \text{on} \quad \partial \Omega_{D_i}, \quad \text{for} \quad i = 1, 2 [, 3]. \tag{2.3.13} $$
To complete our discussion of the weak formulation, we formally define our test and solution function spaces. For (2.3.5) to be well defined, it is sufficient that \( u_i \) (\( i = 1, 2, 3 \)) and its first derivatives are square integrable over the domain \( \Omega \). Define the space of square-integrable functions in \( \Omega \subset \mathbb{R}^d \) (\( d = 2, 3 \))

\[
L^2(\Omega) = \left\{ u : \Omega \to \mathbb{R} \ \left| \int_{\Omega} u^2 \, d\Omega < \infty \right. \right\}, \tag{2.3.14}
\]

and the Sobolev space \([206, \text{p. 52}]\)

\[
H^m(\Omega) = \left\{ u : \Omega \to \mathbb{R} \ \left| D^\alpha u \in L^2(\Omega), \forall |\alpha| \leq m \right. \right\}, \tag{2.3.15}
\]

where

\[
D^\alpha u = \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \ldots \partial x_d^{\alpha_d}}, \quad |\alpha| = \alpha_1 + \ldots + \alpha_d,
\]

are the derivatives taken in the sense of distributions. For example, in 2D the \( H^1(\Omega) \) space is \([68, \text{p. 16}]\)

\[
H^1(\Omega) = \left\{ u : \Omega \to \mathbb{R} \ \left| u, \frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2} \in L^2(\Omega) \right. \right\}. \tag{2.3.16}
\]

Recall that the velocity test functions are zero at the Dirichlet boundaries. Thus, appropriate velocity solution and test spaces are the restriction of the general Sobolev space \([2.3.16]\)

\[
\mathbf{H}^1_{E} = \left\{ \mathbf{u} \in [H^1(\Omega)]^d \mid \mathbf{u} = \mathbf{w} \text{ on } \partial \Omega_D \right\} \quad \text{and} \quad \mathbf{H}^1_0 = \left\{ \psi \in [H^1(\Omega)]^d \mid \psi = \mathbf{0} \text{ on } \partial \Omega_D \right\}, \tag{2.3.17}
\]

respectively, where \( d \) is the spatial dimension. Since there are no pressure derivatives nor BCs in \([2.3.5]\), \( p \) just needs to be square integrable, therefore the appropriate pressure solution and test space is \( L^2(\Omega) \). Now the weak formulation of the NSE can be stated as:

Find \( \mathbf{u} = [u_i]_{i=1}^d \in \mathbf{H}^1_{E} \) and \( p \in L^2 \) such that

\[
r_i^{(u)} = \int_{\Omega} \left[ \text{Re} \left( \frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} \right) \psi_i + \sigma_{ij} \frac{\partial \psi_i}{\partial x_j} \right] \, d\Omega = 0, \quad \text{for } i = 1, 2, 3, \tag{2.3.19}
\]

and

\[
r^{(p)} = -\int_{\Omega} \frac{\partial u_i}{\partial x_j} \varphi \, d\Omega = 0, \tag{2.3.20}
\]
satisfying BCs

\[ u_i = w_i, \quad \text{on } \partial \Omega_D, \quad \text{for } i = 1, 2, 3, \]  
(2.3.21)

and

\[ T_i = -p \hat{n}_i + \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \hat{n}_j = 0, \quad \text{on } \partial \Omega_N, \]  
(2.3.22)

with ICs

\[ \frac{\partial u_j}{\partial x_j} = 0, \quad \text{in } \Omega, \]  
(2.3.23)

and

\[ \hat{n}_i \cdot u_i^0 = \hat{n}_i \cdot w_i^0, \quad \text{on } \partial \Omega_D, \]  
(2.3.24)

\[ \forall \psi = [\psi_i]_{i=1}^d \in H^1_0 \quad \text{and} \quad \forall \varphi \in L^2. \]

Note that there may be a weak solution for (2.3.19)–(2.3.20), which is not smooth enough to be a classical solution for (2.2.5)–(2.2.6). However, if a classical solution does exist, then (2.2.5)–(2.2.6) is equivalent to (2.3.19)–(2.3.20) and the classical and the weak solution coincide. Existence and uniqueness of the weak solution for the NSE is discussed in [68, pp. 319–320]. A sufficient condition that ensures the existence of a solution in (2.3.19)–(2.3.20) is the so-called inf-sup condition. For the case of enclosed flow, it reads [68, p. 225]: There exists a constant \( \gamma > 0 \) such that

\[
\inf_{\varphi \neq \text{const}} \sup_{\psi \neq 0} \frac{|(\varphi, \nabla \cdot \psi)|}{\|\psi\|_{1, \Omega} \|\varphi\|_{0, \Omega}} \geq \gamma,
\]  
(2.3.25)

where \( \|\psi\|_{1, \Omega} = \left( \int_{\Omega} \psi \cdot \psi + \nabla \psi : \nabla \psi \right)^{1/2} \) is a norm for functions in \( H^1_0 \), and \( \|\varphi\|_{0, \Omega} = \|\varphi - (1/|\Omega|) \int_{\Omega} \varphi\| \) is the quotient space norm. Here \( \nabla u : \nabla v \) represents the component-wise scalar product (also known as double dot product), in two dimensions we have \( \nabla u_x \cdot \nabla v_x + \nabla u_y \cdot \nabla v_y \). If (2.3.25) is satisfied, the solution is unique up to a constant. A discrete version of the inf-sup condition is presented in Section 2.4.3 which has to be satisfied for the corresponding finite-dimensional subspaces of \( H^1_0 \) and \( L^2 \) (introduced later in this section).

Now we can form discrete weak approximation of the solution. Discrete weak formulation is obtained by restricting (2.3.19)–(2.3.20) to finite-dimensional subspaces
2.3. SPATIAL DISCRETISATION

\( X^h_0 \subset H^1, X^h_E \subset H^1_E \) and \( M^h \subset L^2(\Omega) \). Then the discrete formulation of (2.3.19)–(2.3.20) is: find \( u_h = [u_{h,i}]_{i=1}^d \in X^h_E \) and \( p_h \in M^h \) such that

\[
\begin{align*}
\{r_i^{(u_h)} &= \int_{\Omega} \left[ \Re \left( \text{Re} \left( \frac{\partial u_{h,i}}{\partial t} + \frac{\partial u_{h,j}}{\partial x_j} \right) \psi_{h,i} \right) 
- p_h \frac{\partial \psi_{h,i}}{\partial x_i} + \frac{\partial u_{h,i}}{\partial x_j} \frac{\partial \psi_{h,j}}{\partial x_j} + \frac{\partial u_{h,j}}{\partial x_i} \frac{\partial \psi_{h,i}}{\partial x_j} \right] \, d\Omega = 0, \quad \text{for } i = 1, 2 \quad [3], \\
\{r^{(p_h)} &= -\int_{\Omega} \frac{\partial u_{h,j}}{\partial x_j} \varphi_h \, d\Omega = 0, \quad \forall \varphi_h \in M^h. 
\end{align*}
\]

Here the index \( h \) refers to a to a discrete approximation associated with a mesh that facilitates the subdivision of \( \Omega \) (domain discretisation is discussed in Section 2.4.1). Equations (2.3.26)–(2.3.27) are nonlinear residuals. Defining appropriate basis for the discrete (finite-dimensional) spaces leads to a nonlinear system of algebraic equations. To solve this system, we need some form of iterative linearisation method, which involves the solution of a linear algebra problem at each iteration, examples include Picard’s method [68, p. 326, 151, pp. 66-68] or Newton’s method [68, p. 325, 151, pp. 71–73]. We use Newton’s method as this is the default nonlinear solver in oomph-lib. In this case, the solution is successively updated as \( \bar{X}^{[k+1]} := \bar{X}^{[k]} + \delta \bar{X} \), with the correction \( \delta \bar{X} \) computed by solving the linear system \( \mathcal{F}_{ns}^{[k]} \delta \bar{X} = -\bar{r}^{[k]} \), where \( \mathcal{F}_{ns}^{[k]} \) is the Jacobian matrix with coefficients \( \mathcal{F}_{ns,ij}^{[k]} = \frac{\partial r_i(\bar{X}^k)}{\partial \bar{X}_j} \).

This process is repeated until the norm of the nonlinear residual \( \bar{r}^{[k]} \) is reduced below a user defined tolerance \( \varepsilon_N \). The nonlinear residual \( \bar{r} \) vanishes at the exact solution. For a good initial approximation \( \bar{X}^{[0]} \), Newton’s method converges quadratically towards the exact solution. For linear problems, Newton’s method converges in one iteration. Thus, if a good initial guess can be provided, for example the solution from a previous time step or the solution from a simpler problem (as with the case of high Reynolds numbers, where the solution of the problem with a lower Reynolds number is used as the initial guess), Newton’s method is a robust general purpose nonlinear solver. For the initial approximation in the first time step, the initial guess of \( \bar{X}^{[0]} = 0 \) is found to be good enough. We describe Newton’s linearisation of the nonlinear term in (2.3.19).

Consider a nonlinear function \( uv \), let \( u^{[k]}v^{[k]} \) be its current approximation at the \( k \)th iteration. We can expand it in Taylor series about the current value and truncate after
the first derivative terms:

\[
\begin{align*}
\frac{\partial u_i}{\partial x_i} & = 0, \\
\text{then according to (2.3.28), we can linearise } \frac{\partial (u_i u_j)^{[k+1]}}{\partial x_j} & \text{ as }
\end{align*}
\]

\[
\begin{align*}
\frac{\partial (u_i u_j)^{[k+1]}}{\partial x_j} & \approx \\
& = \frac{\partial u_i^{[k+1]}}{\partial x_j} u_j^{[k]} + u_i^{[k]} \frac{\partial u_j^{[k+1]}}{\partial x_j} + \frac{\partial u_i^{[k+1]}}{\partial x_j} u_j^{[k]} + u_i^{[k]} \frac{\partial u_j^{[k+1]}}{\partial x_j} - \frac{\partial u_i^{[k]}}{\partial x_j} u_j^{[k]} - u_i^{[k]} \frac{\partial u_j^{[k]}}{\partial x_j}.
\end{align*}
\]

(2.3.29)

Let \( v_{h,i} \), for \( i = 1, 2, 3 \), be the current estimate of the discrete velocity, then the linearised momentum residuals corresponding to (2.3.26) is given by

\[
\begin{align*}
r_i^{(u_h)} = \int_{\Omega} \left[ \text{Re} \left( \frac{\partial u_{h,i}}{\partial t} + \frac{\partial u_{h,i}}{\partial x_j} v_{h,j} + \frac{\partial v_{h,i}}{\partial x_j} u_{h,j} - \frac{\partial v_{h,i}}{\partial x_j} v_{h,j} \right) \psi_{h,i} \\
- p_h \frac{\partial \psi_{h,i}}{\partial x_i} + \frac{\partial u_{h,i}}{\partial x_j} \frac{\partial \psi_{h,i}}{\partial x_j} + \frac{\partial u_{h,i}}{\partial x_i} \frac{\partial \psi_{h,i}}{\partial x_j} \right] d\Omega = 0.
\end{align*}
\]

(2.3.30)

We introduce a set of vector-valued global basis functions \( X_E^h = \text{span}\{\psi_{ij}\}_{i=1,j=1}^{N_i} \), where \( d = 1, 2, 3 \) is the spatial dimension and \( N_i \) is the number of velocity unknowns in the \( i \)th direction and define the discrete approximation of the velocity components \( u_{h,i} \), (for \( i = 1, 2, 3 \)) as

\[
u_{h,i} = \hat{U}_i + \sum_{k=1}^{N_i} U_{ik} \psi_{ik},
\]

(2.3.31)

where \( \hat{U}_i = \sum_{k=1}^{N_{D_i}} w_{ik} \psi_{ik} \) interpolates the Dirichlet boundary data \( w_i \) on \( \partial \Omega_{D_i} \). We also introduce a set of global basis functions \( M^h = \text{span}\{\varphi_k\}_{k=1}^{N_p} \) for the pressure, where \( N_p \) is the number of pressure unknowns, then we have

\[
p_h = \sum_{k=1}^{N_p} P_k \varphi_k.
\]

(2.3.32)
Similarly, the set of basis functions for the velocity and pressure test space are

\[ \psi_{h,i} = \sum_{k=1}^{N_i} \Phi_{ik} \psi_{ik} \quad \text{and} \quad \varphi_h = \sum_{k=1}^{N_p} \chi_k \varphi_k, \]  

(2.3.33)

respectively.

Replacing \( u_{h,i}, p_h, \psi_{h,i} \) and \( \varphi_h \) in \( (2.3.27) \) and \( (2.3.30) \) with \( (2.3.31)–(2.3.33) \) and after rearrangement yields

\[
\begin{align*}
 r_i^{(u_h)} &= \frac{\partial U_i}{\partial t} \int_{\Omega} \text{Re} \, \text{St} \, \psi_{ik} \psi_{il} \, d\Omega + \sum_{k=1}^{N_i} \left\{ U_{ik} \left[ \int_{\Omega} \text{Re} \frac{\partial \psi_{ik}}{\partial x_j} v_j \psi_{il} + \frac{\psi_{ik}}{\partial x_j} \frac{\partial \psi_{il}}{\partial x_j} \, d\Omega \right] \right\} \\
 &\quad + \sum_{k=1}^{N_j} \left\{ U_{jk} \left[ \int_{\Omega} \text{Re} \frac{\partial v_i}{\partial x_j} \psi_{jk} \psi_{il} + \frac{\partial \psi_{jk}}{\partial x_i} \frac{\partial \psi_{il}}{\partial x_j} \, d\Omega \right] \right\} - \sum_{k=1}^{N_p} \left\{ P_k \int_{\Omega} \varphi_k \frac{\partial \psi_{il}}{\partial x_i} \, d\Omega \right\} \\
 &\quad + \int_{\Omega} \text{Re} \left( \text{St} \frac{\partial U_i}{\partial t} + \frac{\partial U_i}{\partial x_j} v_j + \frac{\partial v_i}{\partial x_j} \hat{U}_j - \frac{\partial v_i}{\partial x_j} \hat{v}_j \right) \psi_{il} + \frac{\partial U_i}{\partial x_j} \frac{\partial \psi_{il}}{\partial x_j} + \frac{\partial \hat{U}_j}{\partial x_i} \frac{\partial \psi_{il}}{\partial x_j} \, d\Omega = 0,
\end{align*}
\]

(2.3.34)

and

\[
\begin{align*}
 r_l^{(p_h)} &= - \int_{\Omega} \sum_{k=1}^{N_i} \left\{ U_{jk} \frac{\partial \psi_{jk}}{\partial x_j} \varphi_l \right\} + \frac{\partial \hat{U}_j}{\partial x_i} \varphi_l \, d\Omega = 0,
\end{align*}
\]

(2.3.35)

Equations \( (2.3.34)–(2.3.35) \) represent a system of differential algebraic equations (DAE). For unsteady problems, the semi-discretised problem \( (2.3.34)–(2.3.35) \) still requires time integration since there is a derivative of the unknown \( U_i \) with respect to time in the first term of \( (2.3.34) \). The process where all spatial dimensions of a PDE are discretised but the time dimension is left continuous leading to a system of initial value problems (IVPs) which is solved by numerical methods for initial value ordinary equations is often referred to as the method of lines \[221\]. There are two main groups of methods for time integration, explicit methods such as the Adams-Bashforth family of methods \[99, \text{pp. } 645–646\], and implicit methods such as backwards Euler (BDF-1), Trapezoidal rule (TR), second-order backwards-difference method (BDF-2) and the implicit midpoint rule (IMR) \[99, \text{pp. } 646–650\]. In systems of initial value problems obtained from spatial discretisation of PDEs, stiffness has two sources: multiple temporal scales in the solution of the physical problem and spatial discretisation \[225\]. Explicit time integration performs poorly for stiff problems such as the system \( (2.3.34)–(2.3.35) \) requiring very small time steps \( \Delta t \). Implicit methods are unconditionally stable with respect to the time step size. For the purpose of this work
we use BDF-2 as the time integrator in unsteady problems. In this scheme the time derivative is approximated by [99, p. 265]

\[ \frac{dU_i^{k+1}}{dt} \approx \frac{3U_i^{k+1} - 4U_i^k + U_i^{k-1}}{2\Delta t}. \]  

(2.3.36)

Based on (2.3.34)–(2.3.35) the Jacobian matrix is given by

\[ F_{ns} = \frac{\partial F}{\partial \bar{X}_k} = \begin{bmatrix} \frac{\partial r^{(u_h)}}{\partial U_{ik}} & \frac{\partial r^{(u_h)}}{\partial P_k} \\ \frac{\partial r^{(p_h)}}{\partial U_{ik}} & \frac{\partial r^{(p_h)}}{\partial P_k} \end{bmatrix}, \]  

(2.3.37)

where \( \bar{r} = [r_i^{(u_h)}; r_i^{(p_h)}] \in \mathbb{R}^{dN_u+N_p} \) are the discrete linear residuals corresponding to (2.3.34)–(2.3.35) and \( \bar{X} = [U_{ik}; P_k] \in \mathbb{R}^{dN_u+N_p} \) represents velocity and pressure unknowns. The Newton corrections \( \delta \bar{X} = [\delta U_{ik}; \delta P_k] \in \mathbb{R}^{dN_u+N_p} \) are computed by solving the linear algebraic system \( F_{ns} \delta \bar{X} = -\bar{r}, \) referred to as the discrete Newton problem.

In two spatial dimensions, Newton linearisation applied to (2.3.26)–(2.3.27) give rise to a linear system of algebraic equations which has the following block structure:

\[
\begin{bmatrix}
A + N + G_{11} + W_{11} + M_u & G_{12} + W_{12} & B_1^T \\
G_{21} + W_{21} & A + N + G_{22} + W_{22} + M_u & B_2^T \\
B_1 & B_2 & O
\end{bmatrix}
\begin{bmatrix}
\delta \bar{U}_1 \\
\delta \bar{U}_2 \\
\delta \bar{P}
\end{bmatrix}
= -
\begin{bmatrix}
\bar{r}_1 \\
\bar{r}_2 \\
\bar{r}_p
\end{bmatrix},
\]

(2.3.38)

where \( N_{u_1} + N_{u_2} \) is the number of nodal velocity degrees of freedom (DOFs), and \( N_p \) is the number of pressure DOFs. The block vectors \( \delta \bar{U}_1, \delta \bar{U}_2 \) and \( \delta \bar{P} \) are the Newton corrections for the two velocity components and the pressure DOFs, respectively (i.e. the vector \( \delta \bar{X} \)). On the right hand side, \( \bar{r}_1, \bar{r}_2 \) and \( \bar{r}_p \) are the discrete residuals for the momentum and continuity equation. For notational simplicity, assume that the number of nodal velocity DOF is the same in all directions such that \( N_u = N_{u_1} = N_{u_2} \), then coefficients in the block matrices in (2.3.38) are calculated by:

- The matrix \( A \) is the scalar-Laplacian matrix with entries:

\[ a_{ij} = \int_\Omega \left( \frac{\partial \psi_j}{\partial x_1} \frac{\partial \psi_i}{\partial x_1} + \frac{\partial \psi_j}{\partial x_2} \frac{\partial \psi_i}{\partial x_2} \right) d\Omega, \quad i, j = 1, \ldots, N_u. \]  

(2.3.39)
2.4. **FINITE ELEMENT DISCRETISATION**

In this section we first outline the FEM framework for the NSE, discussing stability and implementation details. The FEM approximation discretises the domain \( \Omega \) into a finite number of non-overlapping sub-domains \( \Omega_e \) called elements, this results in

- The matrix \( B = [B_1, B_2] \) is the scalar divergence matrix, for example, the elements of \( B_1 \) are:

\[
b_{1,ij} = - \int_{\Omega} \frac{\partial \psi_j}{\partial x_1} \varphi_i \, d\Omega, \quad i = 1, \ldots, N_p, \quad j = 1, \ldots, N_u. \tag{2.3.40}
\]

- Define by \( v_h = [v_{h,1}, v_{h,2}]^T \) the current approximation of the velocity. Then \( N \) is the scalar-convection matrix with entries:

\[
n_{ij} = \Re \int_{\Omega} \left( \frac{\partial \psi_j}{\partial x_1} v_{h,1} + \frac{\partial \psi_j}{\partial x_2} v_{h,2} \right) \psi_i \, d\Omega, \quad i,j = 1, \ldots, N_u. \tag{2.3.41}
\]

- The matrices \( W_{11}, W_{12}, W_{21} \) and \( W_{22} \) represent the weak derivatives of the current velocity in the \( x_1 \) and \( x_2 \) directions, and arise from the discretisation of the nonlinear term. For example, the block \( W_{12} \) has entries:

\[
w_{12,ij} = \Re \int_{\Omega} \frac{\partial v_{h,1}}{\partial x_2} \psi_j \psi_i \, d\Omega, \quad i,j = 1, \ldots, N_u. \tag{2.3.42}
\]

- The matrices \( G_{11}, G_{12}, G_{21} \) and \( G_{22} \) exist when the stress divergence form (2.2.3)–(2.2.2) of the NSE is used. For example, the block \( G_{12}, \) has the following entries:

\[
g_{12,ij} = \int_{\Omega} \frac{\partial \psi_i}{\partial x_1} \frac{\partial \psi_j}{\partial x_2} \, d\Omega, \quad i,j = 1, \ldots, N_u. \tag{2.3.43}
\]

- The matrix \( M_u \) is a scaled velocity mass matrix and is only present in unsteady computations, i.e. when discretising the acceleration term \( \frac{\partial U}{\partial t} \). For a generic time discretisation method with time step \( \Delta t \), its elements are:

\[
m_{u,ij} = \frac{C \Re \St}{\Delta t} \int_{\Omega} \psi_j \psi_i \, d\Omega, \quad i,j = 1, \ldots, N_u, \tag{2.3.44}
\]

where \( C \) is a constant depending on the time stepping scheme employed. For BDF-2, \( C = \frac{3}{2} \) [99, p. 265].

### 2.4 Finite Element Discretisation

In this section we first outline the FEM framework for the NSE, discussing stability and implementation details. The FEM approximation discretises the domain \( \Omega \) into a finite number of non-overlapping sub-domains \( \Omega_e \) called elements, this results in
a mesh. The mesh can be structured (e.g. a tensor product grid over a box-like domain) or unstructured. The key feature of the FEM is that the basis functions have finite support. We first discuss domain discretisation in two spatial dimensions with triangular and quadrilateral finite elements (FEs), then extend the discussion to three-dimensions with tetrahedral and hexahedral (brick) FEs.

### 2.4.1 Domain Discretisation

For simplicity, we assume that $\Omega \subset \mathbb{R}^2$ is polygonal. Then we are able to tessellate the domain with a set of straight-edged triangles or quadrilaterals $\Omega_k$, $k = 1, \ldots, K$, defining the triangulation $T_h$ or quadrilateralisation $Q_h$, respectively. These shapes are often referred to as finite elements. This assumes that vertices of neighbouring elements coincide and that $\bigcup_k \Omega_k = \bar{\Omega} = \Omega \cup \partial \Omega$ and $\Omega_\ell \cap \Omega_m = \emptyset$ for $\ell \neq m$. The points where the vertices meet are called nodes. Surrounding any node is a patch of triangles or quadrilaterals that have that node as a vertex. If we label the nodes $j = 1, \ldots, N$, as shown in Figure 2.2 for quadrilateral elements, then for each $j$ we can define a global basis function $\phi_j$ that is nonzero only at that patch.

Each global basis function $\phi_j$ consists of the local (elemental) basis functions $\phi_{j,local}^{(e)}$ incident to the global node $j$, and $j_{local}$ is local node number within an element $\Omega_e$, i.e. for quadrilaterals, $\phi_j = \sum_{e \in Q_h} \phi_{j,local}^{(e)}$, where $Q_h^j$ is the patch in the quadrilateralisation corresponding to the

![Figure 2.2: Structured quadrilateralisation of a domain $\Omega$ into 16 elements with 25 nodes.](image-url)
node $j$. The global basis functions must satisfy the interpolation condition:

$$
\phi_j(\text{node } i) = \begin{cases} 
1, & \text{at node } j \\
0, & \text{at all other nodes}
\end{cases}
$$  (2.4.1)

The interpolation condition (2.4.1) ensures that the Jacobian matrix in (2.3.38) has a well-defined sparse structure and that the unknown coefficients (e.g. in (2.3.31)–(2.3.32)) are equal to the FEM solutions of the continuous quantities. For example, the scalar-Laplacian matrix $A$ is sparse because its entries

$$
a_{ij} = \int_{\Omega} \left( \frac{\partial \psi_j}{\partial x_1} \frac{\partial \psi_i}{\partial x_1} + \frac{\partial \psi_j}{\partial x_2} \frac{\partial \psi_i}{\partial x_2} \right) d\Omega, \quad i, j = 1, \ldots, N_u,
$$

are nonzero only when both the basis functions $\psi_i$ and $\psi_j$ are nonzero simultaneously, this means that the contributions to the entry $(i, j)$ in the Jacobian matrix (2.3.38) arise only from the elements that have $i$ and $j$ as vertices. Therefore the number of nonzero entries per row in the linear system (2.3.38) is constant as the problem size increases. We define a sparse matrix as one where the number of nonzero entries per row is of order $O(1)$. Furthermore, the set of basis functions must $\{\phi_j\}, j = 1, \ldots, N$ must define a partition of unity, $\sum_{j=1}^{N} \phi_j = 1$, so that the global basis functions can accurately capture rigid body movements (i.e. constant functions) [99, p. 46].

The idea is to choose basis functions for the FE subspaces $X_0^h$, $X_E^h$ and $M^h$ so that the interpolation condition (2.4.1) is satisfied, and any order of accuracy can be achieved by increasing the dimensions of the subspaces. Moreover, solving the resulting system of equations should not be prohibitively expensive.

A smooth function can often be approximated to arbitrary accuracy using piecewise polynomials. This can be achieved either by global polynomials of increasing degree (which leads to spectral methods) or by having polynomials of fixed degree defined on decreasing support (i.e. over arbitrarily small elements). In this context, we now introduce piecewise polynomial local basis functions defined on a reference element $\Omega_*$ with local coordinates $s_i$ ($i = 1, 2$). Any arbitrary triangular element $\Omega_e \in \mathbb{R}^2$ can be mapped to a reference element $\Omega_*$ with $s_1, s_2 \in [0,1]$ with isoparametric transformation, described in Section 2.4.2, as depicted in Figure 2.3.

For triangles, the simplest choice leading to conforming approximation is the piecewise linear basis set defined on the reference element. We denote this approximation $P_1$. The local nodes $p_i$, ($i = 1, 2, 3$) are located at local coordinates $(s_1, s_2) =$
(0,0), (1,0), and (0,1), respectively, as depicted in Figure 2.4 (left). The local basis functions are given by

\[
\phi_1^{(e)}(s_1, s_2) = 1 - s_1 - s_2, \quad \phi_2^{(e)}(s_1, s_2) = s_1, \quad \text{and} \quad \phi_3^{(e)}(s_1, s_2) = s_2.
\]

The functions \(\phi_j\) are continuous on \(\Omega\) but are not differentiable across the element boundaries. Nevertheless, they are smooth enough that \(\phi_j \in H^1(\Omega)\) and this choice leads to a conforming approximation space \(X^h_0 = \text{span}(\phi_1, \phi_2, \ldots, \phi_N)\), where \(\phi_j = \{(\phi_j, 0)^T, (0, \phi_j)^T\}\). Being able to locally define the basis functions on a reference element is important for element-by-element assembly of the Jacobian in (2.3.38) to be easily automated. For piecewise quadratic approximation defined on a triangular element (denoted by \(\mathbf{P}_2\)), we introduce additional nodes at the midpoint of each edge. The three corner nodes and three mid-edge nodes are associated with basis functions which take the value one at the node and zero at all other nodes. In the reference element depicted in Figure 2.4 (right), we have local nodes \(p_i\), \((i = 1, \ldots, 6)\), with local coordinates \((s_1, s_2) = (0,0), (1,0), (0,1), (0.5,0), (0.5,0.5)\) and \((0,0.5)\), respectively. The local basis functions are defined by

\[
\begin{align*}
\phi_1^{(e)}(s_1, s_2) &= (1 - s_1 - s_2)(1 - 2s_1 - 2s_2), \quad \phi_2^{(e)}(s_1, s_2) = s_1(2s_1 - 1), \\
\phi_3^{(e)}(s_1, s_2) &= s_2(2s_2 - 1), \quad \phi_4^{(e)}(s_1, s_2) = 4s_1(1 - s_1 - s_2), \\
\phi_5^{(e)}(s_1, s_2) &= 4s_1s_2, \quad \text{and} \quad \phi_6^{(e)}(s_1, s_2) = 4s_2(1 - s_1 - s_2).
\end{align*}
\]

Along each edge there is a unique univariate quadratic function that has given values at three points \(p_i\), thus continuity across the edges is guaranteed and the \(\mathbf{P}_2\) triangle is in \(H^1\). Higher order triangular elements are denoted by \(\mathbf{P}_m\) \((m \geq 3)\) and
2.4. **FINITE ELEMENT DISCRETISATION**

have basis functions taking the form of a bivariate \( m \)th degree polynomial.

Similarly, any arbitrary straight-edged quadrilateral can be mapped onto a reference element via isoparametric transformation with local coordinates \( s_i \in [-1, 1] \), \( (i = 1, 2) \) as shown in Figure 2.5.

The basis functions for the reference quadrilateral element can be constructed by taking the tensor product of appropriate one-dimensional FE basis functions. Consider a one-dimensional element with the local coordinate \( s \in [-1, 1] \). For a linear approximation we require two nodes \( p_i \ (i = 1, 2) \) located at \( s = -1, 1 \), with the associated local basis functions

\[
\phi_1^{(e)}(s) = \frac{1}{2}(1 - s) \quad \text{and} \quad \phi_2^{(e)}(s) = \frac{1}{2}(1 + s),
\]

respectively. For a quadratic approximation we require three nodes \( p_i \ (i = 1, 2, 3) \)
located at \( s = -1, 0, 1 \), with the associated local basis functions

\[
\phi_1^{(e)}(s) = \frac{1}{2} s(s - 1), \quad \phi_2^{(e)}(s) = 1 - s^2, \quad \text{and} \quad \phi_3^{(e)}(s) = \frac{1}{2} s(s + 1),
\]

(2.4.3) respectively.

The simplest conforming approximation for quadrilateral element is the bilinear \( Q_1 \) element. On a rectangle, a bilinear function takes the form \( c_1 + c_2 s_1 + c_3 s_2 + c_4 s_1 s_2 \). If we consider the standard rectangular element \( \Omega \), see [99, pp. 88–93] for more details.

Observe that, for example, \( \phi_4^{(e)} \) associated with node \( (s_1, s_2) = (-1, 1) \), can be constructed by taking the tensor product of the appropriate linear polynomials from (2.4.2) in \( s_1 \) and \( s_2 \) respectively, \( \phi_4^{(e)}(s_1, s_2) = \phi_1^{(e)}(s_1) \otimes \phi_2^{(e)}(s_2) \). The \( Q_1 \) quadrilateral element is in \( H^1 \) and so leads to a conforming approximation space for \( X_0^h \). For piecewise biquadratic approximation on rectangles (denoted \( Q_2 \)), we introduce four additional mid-edge nodes and an additional node at the centroid. If the nodes \( p_i \) (\( i = 1, 2, \ldots, 9 \)) are located at local coordinates as shown in Figure 2.6 (right), then the local basis functions are given by

\[
\begin{align*}
\phi_1^{(e)}(s_1, s_2) & = \frac{1}{4} s_1 s_2 (s_1 - 1)(s_2 - 1), \\
\phi_2^{(e)}(s_1, s_2) & = \frac{1}{4} s_1 s_2 (s_1 + 1)(s_2 - 1), \\
\phi_3^{(e)}(s_1, s_2) & = \frac{1}{4} s_1 s_2 (s_1 + 1)(s_2 + 1), \\
\phi_4^{(e)}(s_1, s_2) & = \frac{1}{4} s_1 s_2 (s_1 - 1)(s_2 + 1), \\
\phi_5^{(e)}(s_1, s_2) & = \frac{1}{2} s_2 (1 - s_1^2)(s_2 - 1), \\
\phi_6^{(e)}(s_1, s_2) & = \frac{1}{2} s_1 (s_1 + 1)(1 - s_2^2), \\
\phi_7^{(e)}(s_1, s_2) & = \frac{1}{2} s_2 (1 - s_1^2)(s_2 + 1), \\
\phi_8^{(e)}(s_1, s_2) & = \frac{1}{2} s_1 (s_1 - 1)(1 - s_2^2), \quad \text{and} \\
\phi_9^{(e)}(s_1, s_2) & = (1 - s_1^2)(1 - s_2^2).
\end{align*}
\]

Higher order quadrilateral elements are denoted by \( Q_m \) (\( m \geq 3 \)) and have continuous \( m \)th degree bivariate approximations. Other combinations are also possible, for example, members of the serendipity element exclude the centroid node, then the construction of a basis function includes all terms in \( Q_2 \) except for the \( s_1^2 s_2^2 \) term, see [99, pp. 88–93] for more details.
Three-dimensional FE s are defined analogously to their two-dimensional counterparts. Any polyhedral domain $\Omega \subset \mathbb{R}^3$ can be completely filled with disjoint tetrahedra or bricks (hexahedrons). For each node $j$ in a tessellation of $\Omega$, we define the global shape function $\phi_j(x_1,x_2,x_3)$ which satisfy the interpolation condition (2.4.1).

For the $P_1$ tetrahedral element we define $\phi_j$ to be a linear function (i.e. of the form $c_1 + c_2x_1 + c_3x_2 + c_4x_3$). Then we define the vector basis functions $\Phi_j = \{(\phi_j,0,0)^T,(0,\phi_j,0)^T,(0,0,\phi_j)^T\}$. Each basis function $\phi_j$ is continuous, ensuring a conforming approximation space $X_h^0 = \text{span}(\phi_1,\ldots,\phi_N) \subset H^1_0$, where $N$ is the number of nodes. On any particular tetrahedral element there are four local basis functions $\phi_{j_{\text{local}}}(s_1,s_2,s_3)$ associated with four vertices, as shown in Figure 2.7 (left), which satisfy the interpolation condition (2.4.1). The $P_2$ element has ten nodes as depicted in Figure 2.7 (right), each matching the ten coefficients needed to define a trivariate quadratic polynomial of the form $c_1 + c_2x_1 + c_3x_2 + c_4x_3 + c_5x_1^2 + c_6x_2^2 + c_7x_3^2 + c_8x_1x_2 + c_9x_1x_3 + c_{10}x_2x_3$. 

Similarly to the two-dimensional case, higher order approximations on tetrahedral elements are denoted by $P_m$ ($m \geq 3$).

The node configurations for brick elements are illustrated in Figure 2.8 for $Q_1$.
2.4.2 Isoparametric Representation and Numerical Integration

An arbitrary domain in two or three spatial dimensions cannot be modelled only by right-angled rectangles or triangles, thus the use of distorted FE s is common when
modelling realistic problems. Isoparametric representation of a FE is typically employed so that we can use the local basis functions defined in Section 2.4.1 to easily automate the assembly of the Jacobian. In this project we only use quadrilateral/brick elements, thus in the sequel we discuss isoparametric mapping for quadrilaterals only. For a treatment of triangular elements see [68, pp. 28–31]. For simplicity, we restrict ourselves to the two-dimensional case. Extension to three dimensions and different element types follow the same computational procedures and concepts, only these are more technically involving and lengthy. Let \( j = J(j_{\text{local}}, e) \) \((j = 1, 2, \ldots, N, \ j_{\text{local}} = 1, 2, \ldots, N_e)\) represent the mapping between the global node number \( j \) and the local node number \( j_{\text{local}} \) within an element \( \Omega_e \), where \( N \) is the number of global nodes and \( N_e \) is the number of nodes within \( \Omega_e \). Within this scheme, the \( i \)th component of the velocity approximation \( u_i(x_1, x_2) \) \((i = 1, 2 \text{ for the two velocity components})\) in an isoparametric element is defined by

\[
    u_i(x_1, x_2) = \sum_{j_{\text{local}}=1}^{N_e} U_{i,j} J(j_{\text{local}}, e) \varphi_{j_{\text{local}}} (s_1, s_2), \tag{2.4.4}
\]

where \( U_{i,j} J(j_{\text{local}}, e) \) \((i = 1, 2)\) is the \( i \)th component of the approximation at local node \( j_{\text{local}} \) in the element \( \Omega_e \). The global coordinates \( x_i \) \((i = 1, 2)\) within an isoparametric element \( \Omega_e \) can be written as

\[
    x_i(s_1, s_2) = \sum_{j_{\text{local}}=1}^{N_e} X_{i,j} J(j_{\text{local}}, e) \varphi_{j_{\text{local}}} (s_1, s_2), \tag{2.4.5}
\]

where \( X_{i,j} J(j_{\text{local}}, e) \) \((i = 1, 2)\) represents the \( i \)th coordinate of the nodal point \( j = J(j_{\text{local}}, e) \) in the reference system \((s_1, s_2)\). This is known as an isoparametric mapping because the same basis functions are used to interpolate both the unknown function and the global coordinates.

The Jacobian matrix of the global coordinates \((x_1, x_2)\) with respect to the local coordinates \((s_1, s_2)\) is given by

\[
    \mathbf{J} = \begin{bmatrix} \frac{\partial x_1}{\partial s_1} & \frac{\partial x_2}{\partial s_1} \\ \frac{\partial x_1}{\partial s_2} & \frac{\partial x_2}{\partial s_2} \end{bmatrix} = \begin{bmatrix} \hat{J}_{11} & \hat{J}_{12} \\ \hat{J}_{21} & \hat{J}_{22} \end{bmatrix}. \tag{2.4.6}
\]

The inverse of \((2.4.6)\) is the Jacobian matrix of the local coordinates \((s_1, s_2)\) with
Differentiating with respect to the local coordinates \((s_1, s_2)\) and is defined by

\[
\hat{J}^{-1} = \frac{\partial(s_1, s_2)}{\partial(x_1, x_2)} = \begin{bmatrix} \frac{\partial s_1}{\partial x_1} & \frac{\partial s_1}{\partial x_2} \\ \frac{\partial s_2}{\partial x_1} & \frac{\partial s_2}{\partial x_2} \end{bmatrix} = \frac{1}{\det(\hat{J})} \begin{bmatrix} \hat{J}_{22} & -\hat{J}_{12} \\ -\hat{J}_{21} & \hat{J}_{11} \end{bmatrix},
\]

(2.4.7)

where \(\det(\hat{J}) = \hat{J}_{11}\hat{J}_{22} - \hat{J}_{12}\hat{J}_{21}\) is the determinant of \(\hat{J}\). The relations (2.4.6) and (2.4.7) serves to transform one coordinate system into another, we demonstrate this in the sequel. The local basis functions defined in Section 2.4 are expressed in terms of local coordinates \(s_1\) and \(s_2\). However, the global basis functions in (2.3.39)–(2.3.44) are expressed in terms of global coordinates \(x_1\) and \(x_2\). We can transform the derivatives that appear in the integrals associated with the weak form (2.3.39)–(2.3.44) using the linear relations

\[
\begin{bmatrix} \frac{\partial}{\partial s_1} \\ \frac{\partial}{\partial s_2} \end{bmatrix} = \begin{bmatrix} \frac{\partial x_1}{\partial s_1} & \frac{\partial x_2}{\partial s_1} \\ \frac{\partial x_1}{\partial s_2} & \frac{\partial x_2}{\partial s_2} \end{bmatrix} \begin{bmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \end{bmatrix} = \hat{J} \begin{bmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \end{bmatrix},
\]

(2.4.8)

and

\[
\begin{bmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \end{bmatrix} = \begin{bmatrix} \frac{\partial s_1}{\partial x_1} & \frac{\partial s_2}{\partial x_1} \\ \frac{\partial s_1}{\partial x_2} & \frac{\partial s_2}{\partial x_2} \end{bmatrix} \begin{bmatrix} \frac{\partial}{\partial s_1} \\ \frac{\partial}{\partial s_2} \end{bmatrix} = \hat{J}^{-1} \begin{bmatrix} \frac{\partial}{\partial s_1} \\ \frac{\partial}{\partial s_2} \end{bmatrix}.
\]

(2.4.9)

The entries of \(\hat{J}\) are computed using the relation (2.4.5) in two dimensions we have

\[
x_1(s_1, s_2) = \sum_{j=1}^{N_e} X_1\mathcal{J}(j,e) \phi_j^{(c)}(s_1, s_2) \text{ and } x_2(s_1, s_2) = \sum_{j=1}^{N_e} X_2\mathcal{J}(j,e) \phi_j^{(c)}(s_1, s_2).
\]

Differentiating with respect to the local coordinates \(s_1\) and \(s_2\) we obtain

\[
\hat{J} = \begin{bmatrix} \sum_{j=1}^{N_e} X_1\mathcal{J}(j,e) \frac{\partial \phi_j^{(c)}}{\partial s_1} & \sum_{j=1}^{N_e} X_2\mathcal{J}(j,e) \frac{\partial \phi_j^{(c)}}{\partial s_1} \\ \sum_{j=1}^{N_e} X_1\mathcal{J}(j,e) \frac{\partial \phi_j^{(c)}}{\partial s_2} & \sum_{j=1}^{N_e} X_2\mathcal{J}(j,e) \frac{\partial \phi_j^{(c)}}{\partial s_2} \end{bmatrix}.
\]

The geometry of the element in global coordinates \((x_1, x_2)\) should be such that the determinant of the Jacobian matrix (2.4.6) is positive everywhere (i.e. \(\det(\hat{J}(s_1, s_2)) > 0\) for all points \((s_1, s_2)\) in the reference element \(\hat{\Omega}_e\)). This is the requirement for the inverse mapping \(\hat{J}^{-1}\) to be well defined. The relation between the differentials of \(x_1\) and \(x_2\) with \(s_1\) and \(s_2\) is given by

\[
dx_1 = \frac{\partial x_1}{\partial s_1} ds_1 + \frac{\partial x_1}{\partial s_2} ds_2 \text{ and } dx_2 = \frac{\partial x_2}{\partial s_1} ds_1 + \frac{\partial x_2}{\partial s_2} ds_2.
\]
Then we have

\[
d\Omega_e = dx_1 dx_2
\]

\[
= \left( \frac{\partial x_1}{\partial s_1} ds_1 + \frac{\partial x_1}{\partial s_2} ds_2 \right) \left( \frac{\partial x_2}{\partial s_1} ds_1 + \frac{\partial x_2}{\partial s_2} ds_2 \right)
\]

\[
= \frac{\partial x_1}{\partial s_1} \frac{\partial x_2}{\partial s_1} ds_1 ds_1 + \frac{\partial x_1}{\partial s_1} \frac{\partial x_2}{\partial s_2} ds_1 ds_2 + \frac{\partial x_1}{\partial s_2} \frac{\partial x_2}{\partial s_1} ds_2 ds_1 + \frac{\partial x_1}{\partial s_2} \frac{\partial x_2}{\partial s_2} ds_2 ds_2.
\]

(2.4.10)

Since \(ds_1\) and \(ds_2\) are differential forms, we have \(ds_1 ds_1 = ds_2 ds_2 = 0\) and \(ds_2 ds_1 = -ds_1 ds_2\), then (2.4.10) becomes

\[
\frac{\partial x_1}{\partial s_1} \frac{\partial x_2}{s_2} ds_1 ds_2 - \frac{\partial x_1}{\partial s_2} \frac{\partial x_2}{s_2} ds_1 ds_2 = \left( \frac{\partial x_1}{\partial s_1} \frac{\partial x_2}{s_2} - \frac{\partial x_1}{\partial s_2} \frac{\partial x_2}{s_2} \right) ds_1 ds_2
\]

\[
= \det(\mathbf{J}) ds_1 ds_2.
\]

Thus we have the relationship

\[
d\Omega_e = dx_1 dx_2 = \det(\mathbf{J}) ds_1 ds_2.
\]

(2.4.11)

From (2.4.11) we see that \(\det(\mathbf{J})\) characterises the local metric of the element in local coordinates, this relationship is depicted in Figure 2.9 we see that (2.4.11) defines the signed area of a parallelogram.

\[\text{Figure 2.9: Geometric representation of equation (2.4.11) for a } Q_1 \text{ quadrilateral element. The signed area of the differential parallelogram is } d\Omega_e = \det(\mathbf{J}) ds_1 ds_2.\]

Now consider the effect of gradually displacing node 4 from Figure 2.9, causing the element to go from a convex shape to a non-convex shape as depicted in Figure 2.10.

As node 4 is shifted towards node 2, the metric near it becomes badly distorted. When
the shape becomes a triangle we have \( \det(\hat{J}(s_{14}, s_{24})) = 0 \), and in the right-most diagram, the local metric near node 4 becomes negative. For higher order elements which contains mid-side or centroid nodes, the non-corner nodes must be placed sufficiently close to their reference element counterparts to avoid extreme distortion of the local metric.

The performance of the method is related to the aspect ratio of the element, the ratio of longest edge to shortest height of an element. For example, given a rectangular element with height \( h_y \) and width \( h_x \), then the rectangular aspect ratio is \( \beta = \max \left\{ \frac{h_x}{h_y}, \frac{h_y}{h_x} \right\} \). We request that the aspect ratio of all elements is \( \mathcal{O}(1) \) (say between 0.1 and 10 [206, p. 50]). In triangular elements, this is related to the minimum angle condition [68, p. 44].

With \( \det(\hat{J}(s_1, s_2)) > 0 \), \( \forall s_1, s_2 \in \hat{\Omega}_e \), the integration over an arbitrary element \( \Omega_e \) can be performed over the reference element \( \hat{\Omega}_e \), taking (2.3.39) as an example, we have

\[
\left( e \right)_{ij} = \int_{\Omega_e} \left( \frac{\partial \psi_j}{\partial x_1} \frac{\partial \psi_i}{\partial x_1} + \frac{\partial \psi_j}{\partial x_2} \frac{\partial \psi_i}{\partial x_2} \right) \, dx_1 dx_2 \\
= \int_{-1}^{1} \int_{-1}^{1} \left( \frac{\partial \psi_j^{(e)}}{\partial x_1} \frac{\partial \psi_i^{(e)}}{\partial x_1} + \frac{\partial \psi_j^{(e)}}{\partial x_2} \frac{\partial \psi_i^{(e)}}{\partial x_2} \right) \det(\hat{J}) \, ds_1 ds_2, \tag{2.4.12}
\]

for \( i, j = 1, 2, \ldots N_e \). For notational convenience, denote the entries in (2.4.7) by

\[ b_1 = J_{22}, \quad b_2 = -J_{12}, \quad c_1 = -J_{21}, \quad \text{and} \quad c_2 = J_{11}, \]

\footnote{Figures have been adapted with permission from [80].}
then the derivative transformation defined in (2.4.9) can be rewritten as

\[
\begin{bmatrix}
\frac{\partial}{\partial s_1} \\
\frac{\partial}{\partial s_2}
\end{bmatrix} = \frac{1}{\det(J)} \begin{bmatrix} b_1 & b_2 \\ c_1 & c_2 \end{bmatrix} \begin{bmatrix}
\frac{\partial}{\partial x_1} \\
\frac{\partial}{\partial x_2}
\end{bmatrix}.
\] (2.4.13)

Using the derivative transformation (2.4.13) we can write (2.4.12) in terms of the derivatives of the local basis functions:

\[
a_{ij}^{(e)} = \int_{-1}^{1} \int_{-1}^{1} \left( b_1 \frac{\partial \psi_j^{(e)}}{\partial s_1} + b_2 \frac{\partial \psi_j^{(e)}}{\partial s_2} \right) \left( b_1 \frac{\partial \psi_i^{(e)}}{\partial s_1} + b_2 \frac{\partial \psi_i^{(e)}}{\partial s_2} \right) \frac{1}{\det(J)} ds_1 ds_2
\]

\[+ \int_{-1}^{1} \int_{-1}^{1} \left( c_1 \frac{\partial \psi_j^{(e)}}{\partial s_1} + c_2 \frac{\partial \psi_j^{(e)}}{\partial s_2} \right) \left( c_1 \frac{\partial \psi_i^{(e)}}{\partial s_1} + c_2 \frac{\partial \psi_i^{(e)}}{\partial s_2} \right) \frac{1}{\det(J)} ds_1 ds_2.
\]

This would result in a \( N_e \times N_e \) elemental Jacobian which can be assembled into the global Jacobian (2.3.38) in an element-by-element fashion.

If any node \( j \) in element \( \Omega_e \) lie on \( \Omega_D \), then it is ‘pinned’ with the appropriate nodal value \( U_{ij}^{(e)} (i = 1 [2]) \), and does not make a contribution to the elemental Jacobian. In this construction, the discrete weighted residuals can be assembled in an element-by-element fashion

\[
\int_{\Omega} (...) d\Omega = \sum_{e=1}^{K} \int_{\hat{\Omega}_e} (...)^{(e)} d\hat{\Omega},
\]

where \( K \) is the total number of elements. The main computational issue in assembling the Jacobian from their elemental components is careful book keeping between global and local indices (indexing information) such as equation numbers and node numbers, and which nodes are pinned (see [121], Algorithm 10 for the assembly details in oomph-lib).

Practical evaluation of integrals that arise in the calculation of Jacobian entries and discrete residual vector in (2.3.34) and (2.3.35) is usually based on numerical integration techniques. The standard is to use Gauss quadrature rules, because in one dimension these have the best order of accuracy per computational cost, and generalise straightforwardly in higher dimensions by taking tensor products of the corresponding lower dimension Gauss rules. As mentioned previously, we only use quadrilateral/brick discretisation, as such we discuss numerical integration for integrals over quadrilaterals/bricks only. Gauss integration rules for triangular elements can be found in [136, pp. 164–174]. In one dimension, the classical Gauss rules for the integral
over the range $s \in [-1, 1]$ are defined by

$$\int_{-1}^{1} F(s) \, ds \approx \sum_{i=1}^{N_{\text{int}}} W_i F(S_i).$$

Here $N_{\text{int}}$ is the number of Gauss integration points, $S_i \in [-1, 1]$ ($i = 1, 2, \ldots, N_{\text{int}}$) are the sample points and the integration weights $W_i$ ($i = 1, 2, \ldots, N_{\text{int}}$) are given by [1, p. 887]

$$W_i = \frac{2}{(1 - S_i^2)[P'_{N_{\text{int}}}(S_i)]^2},$$

where $P_{N_{\text{int}}}(s)$ are Legendre polynomials, with the $N_{\text{int}}$th polynomial normalised as $P_{N_{\text{int}}}(1) = 1$. The $i$th sample point $S_i$ is the $i$th root of $P_{N_{\text{int}}}$. The first three one-dimensional Gauss rules are tabulated in Table 2.1 for weights up to $N_{\text{int}} = 6$ see [271, p. 642]. The one-dimensional $N_{\text{int}}$-point Gauss rule integrates exactly polynomials of order up to $2N_{\text{int}} - 1$. This is optimal in the sense that $2N_{\text{int}}$ order of accuracy is achieved using $2N_{\text{int}}$ free variables (the position of the Gauss points and the associated weights). In two dimensions we compute the integrals of the form:

$$\int_{-1}^{1} \int_{-1}^{1} F(s_1, s_2) \, ds_1 \, ds_2 \approx \sum_{i=1}^{N_{1\text{int}}} \sum_{j=1}^{N_{2\text{int}}} W_{ij} F(S_{i1}, S_{j2}),$$

where $N_{1\text{int}}$ and $N_{2\text{int}}$ are the number of Gauss points in the $s_1$ and $s_2$ direction, respectively. The sample points $(S_{i1}, S_{j2})$ and weights $W_{ij} = W_i \times W_j$ are calculated by taking the tensor products of the corresponding one-dimensional rules. For quadrilaterals presented in this thesis, the shape functions are the same in both the $s_1$ and $s_2$ directions, so taking $N_{1\text{int}} = N_{2\text{int}} = N_{\text{int}}$, the $N_{\text{int}} \times N_{\text{int}}$ product rule integrates exactly all $Q_{2N_{\text{int}}-1}$ functions. For bilinear approximations we typically use the $2 \times 2$ rule, and the $3 \times 3$ rule for biquadratic approximations. Numerical integration in

<table>
<thead>
<tr>
<th>Number of points, $N_{\text{int}}$</th>
<th>Points, $S_i$</th>
<th>Weights, $W_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$\pm \sqrt{\frac{1}{3}}$</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>$\frac{8}{9}$</td>
</tr>
<tr>
<td></td>
<td>$\pm \sqrt{\frac{1}{3}}$</td>
<td>$\frac{8}{9}$</td>
</tr>
</tbody>
</table>

*Table 2.1: One-dimensional Gauss quadrature rules for $N_{\text{int}} = 1, 2$ and 3.*
three dimensions is defined analogously. Taking the same number of Gauss points \( N_{\text{int}} \) in all three spatial dimensions, the approximate volume integral over the range \( s_1, s_2, s_3 \in [-1, 1] \) is approximated by

\[
\int_{-1}^{1} \int_{-1}^{1} \int_{-1}^{1} F(s_1, s_2, s_3) \, ds_1 \, ds_2 \, ds_3 \approx \sum_{i=1}^{N_{\text{int}}} \sum_{j=1}^{N_{\text{int}}} \sum_{k=1}^{N_{\text{int}}} W_{ijk} F(S_i, S_j, S_k),
\]

where the weights \( W_{ijk} \) is constructed by taking the tensor product of the corresponding one-dimensional \( N_{\text{int}} \)-point rules in \( s_1, s_2 \) and \( s_3 \) directions, respectively.

### 2.4.3 Stability of the Mixed Approximation

Discrete weak formulation is obtained by restricting (2.3.5) and (2.3.6) to finite-dimensional subspaces \( X_h^0 \subset H_0^1 \), \( X_h^E \subset H_E^1 \) and \( M_h \subset L^2(\Omega) \). The spaces \( X_h^0 \subset H_0^1 \) and \( M_h \subset L^2(\Omega) \) are approximated independently and simultaneously, giving rise to mixed approximation. Choosing the discrete spaces for the velocity and pressure is non-trivial, for example, choosing bilinear approximations for both velocity components and pressure, which is the simplest possible globally continuous approximation, is unstable [68, p. 229]. A necessary and sufficient condition for a discrete saddle point problems (2.3.38) arising from discretisation via Galerkin methods to be well posed is that the FE approximation spaces satisfy the discrete Ladyzhenskaya-Babuška-Brezzi (LBB) condition,

\[
\min_{\varphi_h \neq \text{const}} \max_{\psi_h \neq 0} \frac{|(\varphi_h, \nabla \cdot \psi_h)|}{\|\psi_h\|_{1,\Omega} \|\varphi_h\|_{0,\Omega}} \geq \gamma > 0 \tag{2.4.14}
\]

where \( \gamma \) is a positive constant independent of the grid size \( h \), \( \psi_h = [\psi_h]^d \) is the discrete velocity test function, and \( \varphi_h \) is the discrete pressure test function given by (2.3.33). This is a discrete form of the continuous inf-sup condition (2.3.25). See [68, pp. 228–229] for more detail.

The choice of FE subspaces for the NSE have been studied in detail. Gresho and Sani provide a very useful and comparative table of possible combinations in [99, pp. 462–465]. Here we present a few schemes discussed in [68, pp. 224–249], for a more rigorous treatment of this topic, refer to [99, pp. 457–622] and [68, pp. 224–268, 330–337].

FE spaces defined on quadrilateral elements which satisfy the LBB condition include \( Q_2 - Q_1 \), \( Q_2 - P_{-1} \) and \( Q_2 - P_0 \) pairs. The notation \( P_{-1} \) is used to emphasise the fact that the approximation is linear, but discontinuous across the inter-element
boundaries i.e. the nodes are inside the element, rather than at its vertices. The approximation $Q_2 - Q_1$ is known in literature as the Taylor-Hood method [99, p. 466, 68, p. 229], it is the simplest higher-order $C^0$-pressure quadrilateral. The elements $Q_1 - P_0$ and $Q_1 - Q_1$ are not stable, however, they can be stabilised as shown in [68, pp. 235–245]. These stabilisation methods introduce a stabilisation block $C$ in place of the zero block in (2.3.38).

FE subspaces defined on triangular elements that satisfy the LBB condition include $P_2 - P_1$ and $P_2 - P_{-1}$. The latter approximation is known as the Crouzeix-Raviart method [99, p. 463, 68, pp. 247–248] and is obtained by adding a cubic bubble function to the velocity space $P_2$ in $P_2 - P_{-1}$. This means that the velocity space is augmented with an additional local degree of freedom situated at the triangle centroid.

Quadrilateral elements in three dimensions are hexahedral (brick) elements. The three-dimensional counterparts of $Q_2 - Q_1$, $Q_2 - P_{-1}$ and $Q_2 - P_0$ satisfy the LBB condition and $Q_1 - P_0$ and $Q_1 - Q_1$ can be stabilised. Stable mixed approximation methods for triangles does not automatically generalise to tetrahedrals [68, p. 249]. Throughout this project, we use Taylor-Hood ($Q_2 - Q_1$) isoparametric LBB-stable FEs [68, p. 229]. In 2D(3D) these have piecewise bi(tri)-quadratic basis functions for all components of the velocity and piecewise bi(tri)-linear basis functions for the pressure. The two-dimensional Taylor-Hood element with velocity and pressure degrees of freedom is shown in Figure 2.11.

![Figure 2.11: Quadrilateral Taylor-Hood ($Q_2 - Q_1$) element: • represents the velocity degrees of freedom, and © represents the pressure degrees of freedom.](image)

2.5 Parallel Implementation of a Finite Element System Assembly

In this section we describe the parallel aspects of the finite element system assembly within oomph-lib. It follows from Section 2.4.1 that a FE system can be assembled from its elemental contributions, which can be computed independently (without the need for communication). Thus, the idea is to assign roughly equal number of elements...
2.5. PARALLEL IMPLEMENTATION OF AN FE SYSTEM ASSEMBLY

to each process and compute their contributions in parallel. Then these contributions are added to the (distributed) global matrix. Thus, it is natural to exploit data level parallelism, the domain decomposition process is described in the sequel.

Parallelism in oomph-lib is achieved with MPI [179], where a number of processes are created and each process holds a small part of the problem (a partial mesh) and is responsible for the computation and storage of data related to it. Communication between processes is required for global operations on the problem. Communication across machines typically uses interconnects, this can severely degrade the performance of the algorithm, thus when designing parallel software it is essential that we minimise the communication overhead. Typically, each process is executed by a single processor (core).

When oomph-lib is compiled with MPI support and executed on multiple processors, most phases of a typical computation (such as assembly of the Jacobian matrix and the residual vector, and the solution of a linear system within a Newton iteration) are automatically performed in parallel with little or no user intervention.

In oomph-lib the Problem object holds the data structures required to solve a problem, including a global Mesh (which may be comprised of multiple sub-Meshes). A Mesh contain Elements which describe the underlying PDE we wish to solve. Elements contain Nodes. Initially, each processor constructs the same Problem object with an initial coarse global Mesh. The Problem is distributed by domain decomposition, where the domain (Mesh) is subdivided into a number of smaller sub-domains. The main task of distributing a problem is to distribute the Problem's global Mesh among the processors. By default the domain decomposition is performed using the graph partitioning software METIS [146] (ParMETIS [147] for the parallel implementation). The distributed memory representation of the domain decomposition process is depicted in Figure 2.12 where we assume that the nodes are labelled in lexicographically from left to right and bottom to top (see Figure 2.2 for example). The domain is decomposed into sub-domains of roughly equal size (i.e. containing roughly the same number of elements) and each processor holds elements associated with one sub-domain. Nodes which lie on processor boundaries are associated with the processor with the highest ID (rank) and shared between the processors at the boundary. For example, in the rank = 0 column in Figure 2.12 where all objects coloured red are

\[4\]In oomph-lib, most data structures are referred to by pointers; i.e. a Problem object stores a pointer to a Mesh object. For an overview of the data structures in the implementation see, the oomph-lib documentation on data structures [119].
associated with processor 0, none of the nodes on processor boundaries are coloured red, since processor 0 has the lowest rank, however, it does retain access to the node (and elements) on processor boundaries (as they are not pruned) via halo elements. Each processor retains, besides the elements in the sub-domain, all the elements in the neighbouring sub-domains that are adjacent to its sub-domain boundary (see the bottom row in Figure 2.12). These are not associated with the processor, we call these halo Elements.

**Elements** which have a halo counterpart are called *haloed elements*. These halo[ed] Elements reduce the potential communication in some cases, due to the locality of the FE basis set, communication is usually required only when a node is on a processor boundary. After the problem has been distributed, additional mesh refinement or adaptation can be performed by each processor to its own partial mesh. For fur-

![Figure 2.12: Decomposition of a domain consisting of four node quadrilaterals. Each column illustrates the memory representation of the problem for a single process. The processes are assigned ranks one to four, distinguished by colour. Top row: METIS is used to associate each element and node with a process. Bottom row: Elements which are not associated with the process and is non-halo are pruned.](image)

other details on the problem distribution among processors, see oomph-lib’s parallel processing tutorial [124]. In this framework the assembly of the Jacobian matrix and residual vector is naturally parallelised. Each processor assembles its elemental Jacobian contributions and elemental residuals using locally stored elements. In this construction each processor also holds the unknowns associated with each equation in the linear system. Once the Newton corrections to the unknowns has been calculated, each processor can update the unknowns associated with its element’s nodes. The update to unknowns stored at a processor’s halo nodes is communicated to the
2.5. PARALLEL IMPLEMENTATION OF AN FE SYSTEM ASSEMBLY

appropriate processor.

Once the Problem (and the Mesh) have been distributed, each process has direct access (without communication) to its own associated Elements. References to a particular Element in a Mesh, say the first Element, will point to a different Element for different processes. A straightforward modification to an existing serial code is required if it contain references to a specific Element, in this case it will point to an element with the smallest ID in the sub-domain associated with that process. An example is the enclosed flow problem where a pressure degree of freedom has to be pinned, one approach would be to pin the pressure DOF of the first Node in the first Element of the process with rank 0. For further implementation details of this example, see the oomph-lib tutorial ‘Parallel solution of the adaptive driven cavity problem’ [125].

By default, the global Mesh is distributed such that approximately equal number of elements are stored on each processor and to minimise the anticipated inter-processor communication required to synchronise the equations associated with unknowns on the processor boundaries. Thus, this naturally leads to a Jacobian distributed (roughly equally) in a row-wise fashion. However, further non-uniform mesh adaptation may cause a significant increase in the number of elements on some processors. In this case we may recover a uniformly distributed matrix (with approximately equal number of rows per processor) with load balancing techniques such as re-distributing the problem, further details can be found in [112]. If we consider the extraction of a sub-block matrix from the Jacobian resulting from the distributed mesh shown in Figure 2.12, this sub-block matrix may not be uniformly distributed. For example, consider the sub-block matrix associated with the nodes on one of the four straight boundaries of the domain in Figure 2.12. Here, data is stored on only two of the four available processors, leading to a sub-block matrix which is stored on two processors instead of being distributed among all four processors. In this project, we work with row-based partitioned matrices where each processor holds approximately the same number of rows, we refer to this as a uniform row distribution. We achieve this via inter-processor communication to re-distribute the rows among the processors. This is handled by oomph-lib’s parallel block preconditioning framework, see Chapter 5 for more detail. Parallel issues associated with third party parallel linear solvers utilised in oomph-lib are discussed in Chapter 3.
2.6 Summary

To summarise this chapter, we presented the NSE in dimensional form, introduced their non-dimensionalisation and the discretisation by the FE method. To achieve this, we transform the classical formulation of the problem into the weak (integral) form using Galerkin’s method. Adopting FE basis sets, we approximate the continuous weak form, obtaining a discrete nonlinear problem which is solved by Newton’s method. Application of Newton’s method requires repeated solution of large, sparse linear systems. The methods for completing this task are the topic of Chapter 3. We briefly discussed the parallel implementation of the FEM assembly process as implemented in oomph-lib.
Chapter 3

Solution of Sparse Linear Systems

In this chapter we discuss efficient solvers for linear systems obtained from sparse discretisation methods\(^1\) of the Navier-Stokes equations (NSE), such as the finite element (FE), finite volume (FV), and finite difference (FD) methods. We have shown in Section 2.3.1 that the algebraic system arising from the Navier-Stokes equations have a certain block structure and sparsity pattern. In this thesis, a matrix is considered sparse if the number of nonzero (NNZ) coefficients per row is of order \(O(1)\). This implies that the NNZ in the coefficient matrix grows linearly with respect to the size of the discrete linear system \(N\), and the most optimal methods for the solution of such linear systems should have asymptotic complexity \(O(N)\). Iterative methods generally have storage requirements and computational cost per iteration that scale linearly with the NNZ in the coefficient matrix, making them a logical choice of solvers for sparse linear systems. To this effect, an overview of general iterative methods is presented in Section 3.3 which includes relaxation methods, multigrid methods, and Krylov subspace methods.

However, iterative methods may converge very slowly or even fail to converge. Typically, preconditioning techniques are employed which transform the linear system to an equivalent one for which iterative solvers will converge more rapidly. The concept of preconditioning is introduced in Section 3.4 with a brief discussion of common general preconditioners. In the case of multiphysics problems, or problems with vector-valued solutions, an appealing preconditioning methodology is block preconditioning. In this context, we present block preconditioners for the standard Navier-Stokes systems in Section 3.4.3. The solvers discussed in this section will form a basis for the

\(^1\)Methods where basis sets have local support resulting in linear systems with a sparse coefficient matrix.
new efficient preconditioning algorithms for a class of the NSE with weakly imposed boundary conditions described in Section 3.5. Parallelisation aspects of these solvers are presented in Section 3.5. Firstly we present an overview of sparse direct methods and discuss their lack of optimality when applied to solving sparse linear systems.

3.1 Direct Solvers for Dense Linear Systems

Let $A \in \mathbb{R}^{N \times N}$ and $\bar{x}, \bar{b} \in \mathbb{R}^{N}$, then the problem of solving a linear system is stated as: find $\bar{x}$ such that

$$A \bar{x} = \bar{b},$$

(3.1.1)

where $\bar{b}$ is a given (known) vector, $\bar{x}$ is a vector of unknowns and $A$ is a matrix with entries representing the coefficients of unknowns in each equation. Furthermore, we assume that the matrix $A$ is nonsingular. The coefficient matrix, $A$, can either be dense or sparse. In this thesis, we define a dense matrix as one where the NNZ grows asymptotically faster than the size of the matrix. The sparsity of the coefficient matrices obtained from different variants of the Galerkin approximation method depends on the locality of the basis set support it utilises. If this support is global, as in the spectral method, the coefficient matrix is small and dense. If this support is local (as in FE methods), then the coefficient matrix is large and sparse.

Most direct methods are based on matrix decomposition techniques with some sort of pivoting to preserve numerical stability of the process. For example, the $LU$-factorisation \[94, pp. 97–102\] performs the decomposition:

$$A = LU,$$

(3.1.2)

where $L, U \in \mathbb{R}^{N \times N}$ are lower and upper triangular matrices, respectively. The system (3.1.2) can then be solved by forward/back substitution via the two triangular systems $L\bar{y} = \bar{b}$ and then $U\bar{x} = \bar{y}$, respectively. The $QR$ factorisation \[94, pp. 223–236\] decomposes $A$ as: $A = QR$, where $Q \in \mathbb{R}^{N \times N}$ is orthogonal ($Q^TQ = QQ^T = I \in \mathbb{R}^{N \times N}$ is the identity matrix) and $R \in \mathbb{R}^{N \times N}$ is upper triangular. Then the solution $\bar{x}$ can be computed as $R\bar{x} = Q^T\bar{b}$. Computing the $QR$-factorisation is approximately twice as expensive as the $LU$-factorisation, thus QR-factorisation is only used in dense eigenvalue computations, with columns of $Q$ representing the eigenvectors. Algorithms which do not take into account the sparsity of the coefficient matrix are referred to as
3.2 SPARSE DIRECT SOLVERS

dense direct solvers. They are deterministic in a sense that the algorithm terminates predictably, in other words, the solution $\bar{x}$ for the linear system (3.1.1) is computed in a fixed and finite number of steps. In particular Gaussian Elimination (GE) requires $2N^3/3$ floating point operations (flops) [94, p. 100]. Encounters with small pivot elements in GE type algorithms will produce large values during the elimination process, which may cause numerical instability. To remedy this situation, pivoting strategies are introduced with the aim to reduce error propagation. The simplest strategy is to select the $k$th pivot (i.e. the diagonal entry $A(k,k)$ used in the elimination of the $k$th column) to be the element with the largest absolute value in the sub-column $A(k:N,k)$. The pivot element is permuted into the pivot position $A(k,k)$ with row interchange operations. This is called partial pivoting and there are $O(N^2)$ comparisons associated with the search for the pivots [94, p. 112]. In complete pivoting, the $k$th pivot is chosen as the element with the largest absolute value in the sub-matrix $A(k:N,k:N)$, then permuted into the position $A(k,k)$ using both row and column permutations. In this case there are $O(N^3)$ comparisons associated with the pivot search, which is a significant overhead when compared to the number of flops involved in the whole GE algorithm. However, GE with complete pivoting is more stable than with partial pivoting. In practice however, numerical instability of GE with partial pivoting is highly unlikely.

Dense direct solvers are expensive with respect to both their flops which is of order $O(N^3)$, and the storage requirement which is of order $O(N^2)$. This becomes infeasible even for moderate values of $N$, for example, solving a system of $N = 100,000$ linear equations requires at least 80 GB of memory.

3.2 Sparse Direct Solvers

Sparse direct solvers are designed to exploit the sparsity pattern of $A$ and use efficient data structures to store the matrix and its factors, such as compressed row storage (CRS) format, where only the nonzero entries are stored, along with two auxiliary vectors for their column indices and the new row starting indices, see [215, p. 93]. Thus, the storage requirement of a sparse matrix in CRS format is $O(N)$. Sparse direct factorisation methods have been well studied, for a detailed overview of the techniques, see the books by Duff, Erisman, and Reid [66], Davis [57] and Saad [215].

---

2Assuming that each double value require 8 bytes of storage.
CHAPTER 3. SOLUTION OF SPARSE LINEAR SYSTEMS

pp. 73–97], and the review papers by Duff [211] and Heath et al. [128]. In the sequel we discuss a few commonly used strategies.

3.2.1 Matrix Re-ordering Strategies

Prior to the factorisation, the coefficient matrix is permuted so that the factorisation process should not encounter zero pivots and with the aim of minimising fill-in (nonzero elements in the factors that do not exist in the original coefficient matrix). The permutation is usually generated by symbolic analysis on $A$ (where no numerical values are processed), these methods are called ordering methods. Ordering methods work with the graph representation of the matrix $A$, an example is the Cuthill-McKee [53] ordering which is based on the breadth first search traversal algorithm [148] used in graph theory. In general, this is not a trivial task, for example, it is shown in [264] that minimising fill-in for Cholesky factorisation is a NP-complete problem. Fortunately, there exist heuristics such as the minimum degree [240] and nested dissection [89] methods which reduce fill-in significantly. The ordering should also be numerically stable. The numerical stability depends on the growth of the entries during the elimination process, thus methods such as threshold pivoting, described in [66, p. 98], are employed to control the rate of growth of the entries in the matrix. There may be a trade-off between maintaining sparsity and numerical stability, this issue is discussed in [66, Chapter 7].

3.2.2 Supernodal LU Solvers

In the supernodal [161] approach, the matrix columns with similar sparsity pattern are grouped together to form ‘supernodes’. Then the factorisation operations on such supernodes are performed simultaneously, using dense level 3 BLAS kernels, which allow fast execution. SuperLU (see [60] for the sequential version and [163] for the distributed memory version) is an implementation of such solver and is oomph-lib’s default sparse direct solver.

3.2.3 Frontal Solvers

Frontal methods [142] were designed in 1970 by B. Irons to solve specifically linear systems that arise from FE discretisations. The idea is to overlap the assembly of the FE system and its solution. As soon as a row of the global Galerkin matrix is fully
3.2. SPARSE DIRECT SOLVERS

assembled, it can be used in the process of \( LU \)-factorisation. The \( LU \) or Cholesky decomposition of the coefficient matrix is applied concurrently with the assembly of element matrices by transforming the equations in (3.1.1) that corresponds to a subset of elements at a time. This subset is known as the front and it depicts the transition region between the elements which have been factorised and those which have not. The advantage is that the process automatically avoids operating on the large number zero terms in the coefficient matrix, in fact, the coefficient matrix is never explicitly formed.

Multifrontal methods introduced by Duff and Reid [65] are an improvement of the frontal method where many independent fronts are processed concurrently. This is an inherently parallel process as different fronts can be operated on by different processors. For an implementation of multifrontal methods see MUMPS [182] (a MUltifrontal Massively Parallel sparse direct Solver).

Remark 3.2.1. Sparse direct solvers are competitive to iterative solvers if the problems are reasonably small, i.e. up to a few thousand equations. For large problems, especially in three-dimensions, direct solvers often have memory requirements and computational cost that are too large for practical use. The large stencil sizes in 3D may cause excessive amount of fill-in within the matrix factors, which might require 100 or more times the storage than that for the original matrix. Iterative methods can be far more effective in such cases, both with respect to CPU time and memory requirements, provided that they can be made to converge in a small number of iterations, see the comparative studies in [31].

The complexity estimates for sparse direct solvers are dependent on the property of the coefficient matrix. If \( A \) is an SPD matrix derived from the discretisation of a PDE on a regular 2D FE mesh, followed by the generalised nested dissection ordering [168], then it is shown in [90] that the Cholesky factorisation of \( A \), can be computed using \( O(N^{3/2}) \) flops and with memory requirement of \( O(N \log N) \). The same complexity bounds have been demonstrated for when \( A \) is nonsymmetric and indefinite when GE with partial pivoting is applied [88]. For band matrix solvers, where the re-ordering minimises the bandwidth of \( A \), W. Bangerth reports in [172] that the number of flops is \( O(N^2) \) in 2D and \( O(N^{7/3}) \) in 3D, and the memory requirements are \( O(N^{3/2}) \) in 2D and \( O(N^{5/3}) \) in 3D.

By contrast, for sparse matrices, an optimal iterative method would have both time and space complexity of \( O(N) \), that is, both the number of flops and memory requirements during the execution of the algorithm should scale linearly with the size
of the discrete linear system. To achieve this bound, in terms of storage we either need in-place algorithms, or algorithms with the storage overhead of the same asymptotic order as the coefficient matrix itself. In terms of the computational effort, in the case for sparse matrices, iterative methods have optimal cost per iteration, thus to make them optimal solvers we need to ensure that the number of iterations required for convergence does not depend on $N$. In this thesis, we refer to solvers with time and space complexity of $O(N)$ as optimal solvers. However iterative methods can be less robust and may lose the generality of direct methods. This is especially the case when they are applied to find the solution of FE discretisations of PDEs.

3.3 Iterative Solvers

This section begins with a review of the relaxation methods: Jacobi, Gauss-Seidel (GS), successive over-relaxation (SOR) and symmetric successive over-relaxation (SSOR). Then we introduce the concept of projection methods, Krylov subspace methods, and in particular the Generalized Minimal Residual method (GMRES), which is used in this project.

In subsequent sections, we assume that the coefficient matrix of the system of linear equations (3.1.1) is derived from the discretisation of a PDE and is sparse (e.g. the FE method was used). Let $\vec{x}^* \in \mathbb{R}^N$ be the exact solution to the linear system (3.1.1). Iterative methods solve (3.1.1) by successively computing a series of approximations $\vec{x}^{[0]}, \vec{x}^{[1]}, \ldots, \vec{x}^{[k+1]} \in \mathbb{R}^N$ (known as iterates) that should converge in a suitable measure (e.g. a vector norm) to the exact solution $\vec{x}^*$. The method continues computing a series of iterates until convergence is achieved, which is usually defined by a user given tolerance $\varepsilon$ and some sort of error norm, for example, the iterate $\vec{x}^{[k]}$ is close enough to $\vec{x}^*$ if $\|A\vec{x}^{[k]} - \vec{b}\|_2 < \varepsilon$.

The convergence property of iterative solvers is dependent on the PDEs governing the linear system, thus general iterative solvers are likely to perform poorly. The basic iterative methods introduced in the next section may not converge for linear systems such as (2.3.38) or may take an impractically large number of iterations. They are seldom used on their own to solve realistic problems, but they do form a key components for more advanced solvers introduced later in this chapter.
3.3. ITERATIVE SOLVERS

3.3.1 Relaxation Methods

Relaxation methods such as Jacobi or Gauss-Seidel are based on the relaxation of coordinates and are the simplest examples of an iterative solver. At each iteration, one or more of the components of the approximate solution are modified simultaneously until convergence is reached, this is called a relaxation step.

We decompose the coefficient matrix $A$ of (3.1.1) to

$$A = D - L - U,$$  \hfill (3.3.1)

where $D \in \mathbb{R}^{N\times N}$ is the diagonal of $A$, $L \in \mathbb{R}^{N\times N}$ is the strictly lower triangular matrix of $A$, and $U \in \mathbb{R}^{N\times N}$ is the strictly upper triangular part of $A$. On the $k$th iteration, the next approximation $x^{[k+1]}$ is chosen to annihilate the $i$th component of the residual vector $\tilde{r} = \tilde{b} - A\tilde{x}$, i.e.

$$\tilde{r}_i^{[k+1]} = (\tilde{b} - A\tilde{x}^{[k+1]})_i = 0.$$ \hfill (3.3.2)

The Jacobi iteration is derived from the following relaxation scheme

$$b_i - \sum_{j=1}^{N} a_{ij} \tilde{x}_j^{[k]} - a_{ii} \tilde{x}_i^{[k+1]} = 0, \quad i = 1, \ldots, N.$$ \hfill (3.3.3)

Isolating the $\tilde{x}_i^{[k+1]}$ term, the Jacobi iteration in matrix form is:

$$\tilde{x}_i^{[k+1]} = D^{-1} (L + U) \tilde{x}_i^{[k]} + D^{-1} \tilde{b}.$$ \hfill (3.3.4)

It is clear from (3.3.3) that the relaxation steps for all components of the solution vector for the Jacobi iteration can be performed in parallel since each new update $\tilde{x}_i^{[k+1]}$ involves only the solution components from the previous iteration. The GS iteration is derived from the following relaxation scheme,

$$b_i - \sum_{j=1}^{i-1} a_{ij} \tilde{x}_j^{[k+1]} - a_{ii} \tilde{x}_i^{[k+1]} - \sum_{j=i+1}^{n} a_{ij} \tilde{x}_j^{[k]} = 0, \quad i = 1, \ldots, N,$$ \hfill (3.3.5)

in matrix form we have:

$$\tilde{x}_i^{[k+1]} = (D - L)^{-1} U \tilde{x}_i^{[k]} + (D - L)^{-1} \tilde{b}.$$ \hfill (3.3.6)
Equation (3.3.6) describes a forward Gauss-Seidel (FGS) iteration. If we traverse the equations in (3.3.5) backwards, i.e. $i = n, n - 1, \ldots, 1$, then an equivalent formulation is

$$\bar{x}^{[k+1]} = (D - U)^{-1} L \bar{x}^{[k]} + (D - U)^{-1} b.$$  \hspace{1cm} (3.3.7)

Equation (3.3.7) is referred to as the backward Gauss-Seidel (BGS) iteration. Performing a FGS iteration followed by a BGS iteration is referred to as a symmetric Gauss-Seidel iteration (SGS).

In contrast to the Jacobi iteration, from (3.3.5) we note that to relax the $i$th component at the $(k+1)$th iteration requires previous relaxation of all the components $\bar{x}^{[k+1]}_j$, $j = 1, \ldots, i - 1$, meaning that GS is sequential in its nature. Parallel versions of the GS method exist, which usually involve a permutation of the coefficient matrix. For example, if the PDE is discretised by a structured five-stencil square mesh, then we can update a whole diagonal in parallel, working from one corner to the opposite corner. This is referred to as wavefront ordering [268, pp. 33–34]. In this case, the workload increases as the wavefront moves towards the diagonal of the mesh, and decrease as it moves away from it. If this algorithm is implemented naïvely for common distributed matrix data structures (such as uniform row distribution, where each processor holds approximately the same number of rows of the coefficient matrix), then efficient load balancing may be difficult to achieve.

Another approach is the red-black ordering method [268, pp. 34–37]. This ordering comes from the observation that if the nodes of the mesh were coloured either red or black, such that no two nodes of the same colour are connected by an edge, then we can update all the nodes of the same colour in parallel. For the five point stencil, this would give yield a colouring scheme similar to a chess board. This ordering method has better parallel properties than the wavefront ordering since load balancing is easier to achieve. Another technique of parallelising the GS method is the so-called block GS method or hybrid GS [12], where the coefficient matrix is subdivided into a number of roughly equal blocks, and then a block Jacobi relaxation is applied, where each block is inverted by a sequential GS method. This method is parallel at the block level, but has inferior numerical properties to the standard GS method as the nonzeros in off-diagonal blocks are neglected. We use hybrid GS as the parallel smoother for AMG in Section 7.4.

The convergence of these methods can be accelerated with techniques such as block relaxation [215, p. 108–112], where more than one component of the residual
\( r \) are annihilated per relaxation step, as opposed to the point relaxation schemes described above. Another technique is over-relaxation or under-relaxation, where one modifies the approximations \( \tilde{x}_i^{[k+1]} \) by a factor that is smaller or larger than 1, than indicated by (3.3.4) and (3.3.6). Over-relaxation has been shown to improve the rate of convergence [259], but the choice of the optimal relaxation parameter is problem dependent. In the sequel, let \( \omega \in \mathbb{R} \) be the relaxation factor, we present the successive over-relaxation for the Jacobi method and GS methods. The over-relaxation for the Jacobi iteration (3.3.4) is often referred to as the damped or weighted Jacobi iteration, and \( \omega \) is referred to as the damping factor or weight parameter. The recurrence is

\[
\tilde{x}_i^{[k+1]} = \omega \tilde{x}_i^{[J]} + (1 - \omega) \tilde{x}_i^{[k]}, \tag{3.3.8}
\]

where \( \tilde{x}_i^{[J]} \in \mathbb{R}^N \) is the \((k+1)\)th Jacobi iterate given by (3.3.4). The over-relaxation for the GS iteration, referred to as the successive over relaxation method (SOR), is given by the recurrence relation

\[
\tilde{x}_i^{[k+1]} = \omega \tilde{x}_i^{[FGS]} + (1 - \omega) \tilde{x}_i^{[k]}, \tag{3.3.9}
\]

where \( \tilde{x}_i^{[FGS]} \in \mathbb{R}^N \) is the \((k+1)\)th FGS iterate given by (3.3.6). This is called the forward SOR (FSOR) iteration. The backward SOR (BSOR) iteration can be defined analogously as

\[
\tilde{x}_i^{[k+1]} = \omega \tilde{x}_i^{[BGS]} + (1 - \omega) \tilde{x}_i^{[k]}, \tag{3.3.10}
\]

where \( \tilde{x}_i^{[BGS]} \) is the \((k+1)\)th BGS iterate given by (3.3.7). The symmetric SOR (SSOR) method corresponds to taking a FSOR sweep followed by a BSOR sweep, i.e. it is an over-relaxation of the SGS method:

\[
\tilde{x}_i^{[k+1/2]} = \omega \tilde{x}_i^{[FGS]} + (1 - \omega) \tilde{x}_i^{[k]}, \quad \tilde{x}_i^{[k+1]} = \omega \tilde{x}_i^{[BGS]} + (1 - \omega) \tilde{x}_i^{[k+1/2]}. \tag{3.3.11}
\]

### 3.3.1.1 Rate of Convergence of the Relaxation Methods

The iterative methods described in the previous section lack robustness for many classes of PDEs. Sufficient (but not necessary) convergence criterion for the Jacobi method is a diagonal dominance of the coefficient matrix, see Saad [215, pp. 114 – 127]. Even if relaxation methods do converge, the rate of convergence can be very slow [35, p. 22]. The cause for this slow convergence is shown below. For simplicity, the following
demonstration considers only the damped Jacobi iteration applied to a discretisation of a second order ordinary differential equation (ODE). The observation also holds for SOR type methods, see [215, pp. 424–435]. This illustration will demonstrate a ‘smoothing property’ of relaxation methods, which is one of the two key concepts used in multigrid solvers, see Section 3.3.2.

Consider an FD discretisation of the one-dimensional model problem:

\[-\frac{\partial^2 u}{\partial x^2} = f(x) \quad \text{in } \Omega = [0, 1], \quad \text{subject to } u(0) = u(1) = 0. \tag{3.3.12}\]

The continuous domain \( \Omega \) is uniformly discretised into \( N + 2 \) points such that \( \Omega_h = \{x_i\} \) for \( i = 0, \ldots, N + 1 \), where \( x_i = i \times h \) and \( h = 1/(N + 1) \). Centered FD approximation [159, p. 7] of the second order derivative in (3.3.12) leads to a system of linear equations

\[ A\tilde{x} = \bar{b}, \tag{3.3.13} \]

where \( A = \text{tridiag}[-1, 2, -1] \in \mathbb{R}^{N \times N} \) is a tridiagonal coefficient matrix, \( \tilde{x} \in \mathbb{R}^N \) is the vector of unknowns and \( \bar{b} = h^2[f(x_i)] \in \mathbb{R}^N \) is the source function evaluated at each discrete point \( x_i \) in (0, 1).

Recall that \( \lambda_l(A) \in \mathbb{R} \) is an eigenvalue of \( A \) with the corresponding eigenvector \( \tilde{u}_l(A) \in \mathbb{R}^N \) if \( A\tilde{u}_l = \lambda_l\tilde{u}_l \), where \( l = 1 \ldots N \). The eigenvalues and eigenvectors of the matrix \( A \) from (3.3.13) can be obtained explicitly, see [35, p. 18]: The eigenvalues are

\[ \lambda_l(A) = 4 \sin^2 \left(\frac{l\pi}{N + 1}\right), \quad \text{for } l = 1, \ldots, N, \tag{3.3.14}\]

and the components \( u_{i,j} \) of the associated eigenvectors of the matrix \( A \) are given by

\[ u_{i,j} = \sin \left(\frac{j\pi}{N + 1}\right) \quad \text{for } j, l = 1, \ldots, N. \tag{3.3.15}\]

With the splitting \( A = \tilde{M} - \tilde{N} \), the iterates can be written in the form

\[ \tilde{x}^{[k+1]} = \tilde{M}^{-1}\tilde{N}\tilde{x}^{[k]} + \tilde{M}^{-1}\bar{b} = G\tilde{x}^{[k]} + \bar{f}, \tag{3.3.16}\]

where \( G = I - \tilde{M}^{-1}A \) is the iteration matrix. If we denote the iteration matrix for the damped Jacobi iteration (3.3.8) as \( G_{J_\omega} = (I - \omega D^{-1}A) \) with \( \bar{f}_{J_\omega} = \omega D^{-1} \), then
the approximate solution $\bar{x}^{[k+1]}$ is given by:

$$\bar{x}^{[k+1]} = G_{J_\omega} \bar{x}^{[k]} + \bar{f}_{J_\omega}. \quad (3.3.17)$$

Since $D = 2I$, then $G_{J_\omega} = (I - \frac{\omega}{2} A)$ and the eigenvalues of the iteration matrix $G_{J_\omega}$ are

$$\lambda_l(G_{J_\omega}) = 1 - 2\omega \sin^2 \left( \frac{l\pi}{2(N+1)} \right). \quad (3.3.18)$$

The eigenvectors of $A$ and $G_{J_\omega}$ are the same. If $\bar{x}^*$ is the exact solution for (3.1.1) then it is a fixed point of (3.3.17), i.e.

$$\bar{x}^* = G_{J_\omega} \bar{x}^* + \bar{f}_{J_\omega}. \quad (3.3.19)$$

Define the error at the $k$th iteration step to be $\bar{d}^{[k]} = \bar{x}^{[k]} - \bar{x}^*$. Subtracting (3.3.19) from (3.3.17) and expanding the resulting recurrence in $k$ gives

$$\bar{x}^{[k+1]} - \bar{x}^* = G_{J_\omega}(\bar{x}^{[k]} - \bar{x}^*) = \ldots = G_{J_\omega}^{k+1}(\bar{x}^{[0]} - \bar{x}^*). \quad (3.3.20)$$

Thus, the error at the $k$th iteration step satisfies

$$\bar{d}^{[k]} = G_{J_\omega}^{k} \bar{d}^{[0]}. \quad (3.3.21)$$

We expand the error vector $\bar{d}^{[0]}$ in the eigenbasis of $G_{J_\omega}$ as $\bar{d}^{[0]} = \sum_{l=1}^{N} \alpha_l \bar{v}_l$. Then from (3.3.17), (3.3.20) and (3.3.21) we have

$$\bar{d}^{[k]} = \sum_{l=1}^{N} (G_{J_\omega})^k \alpha_l \bar{v}_l = \sum_{l=1}^{N} (\lambda_l(G_{J_\omega}))^k \alpha_l \bar{v}_l. \quad (3.3.22)$$

The expansion (3.3.22) shows that after the $k$th iteration step, each error component is reduced by a factor of $\left[1 - 2\omega \sin^2 \left( \frac{l\pi}{2(N+1)} \right) \right]^k$. Thus, for the Jacobi iteration, the error depends on the eigenvalues of the iteration matrix $G_{J_\omega}$ and the damping factor $\omega$ – the error is reduced the fastest for small values of $|\lambda(G_{J_\omega})|$. The parameter $l$ is often termed the wave number. Note that for small values of $l$ ($l < N/2$), the eigenvectors (3.3.15) correspond to long and smooth sinusoidal oscillations on the grid, whilst large values of $l$ ($l > N/2$) correspond to high frequency oscillations. Applying this description to (3.3.22) low frequency error components correspond to
wave numbers $l \in [1, N/2)$, and high frequency error components correspond to wave numbers $l \in (N/2, 1]$. From Figure 3.1 we observe that $\omega = 1/3$ eliminates only highly oscillatory error components fast (where $|\lambda_l(G_{J_\omega})| \leq 0.5$), $\omega = 2/3$ eliminates over half of the error components fast, whilst $\omega = 1$ only eliminates the error components close to $l = N/2$. We can also infer that low frequency error components are the main cause of slow convergence, since $|\lambda_l(G_{J_\omega})| < 0.5$ will never hold for $l \leq N/4$ with any value of $\omega \in [0, 1]$. Thus, we conclude the damped Jacobi iteration generally eliminates high frequency errors faster. This is a common property of all relaxation methods, since they use the information from only the nearest neighbours when annihilating a particular error component. Elimination of low frequency error components require information from all the grid points.

![Plot of eigenvalues $\lambda_l$ for the damped Jacobi iteration matrix](image)

**Figure 3.1:** The eigenvalues $\lambda_l(G_{J_\omega})$ for the damped Jacobi iteration matrix as a function of the wave number $l$ for damping factors $\omega = \frac{1}{3}, \frac{2}{3}$ and 1. The horizontal lines at $\pm 0.5$ indicates the wave numbers for which $|\lambda_l(G_{J_\omega})| \leq 0.5$.

### 3.3.2 Multigrid Methods

In this section we review multigrid methods for the solution of sparse linear systems that arise from the discretisation of the second-order self-adjoint problems. Multigrid is at the heart of many optimal iterative solvers. A well tuned multigrid iteration will scale linearly with respect to wall-clock time and storage for the discrete Poisson problem. Such matrices and their singular perturbations (the convection-diffusion problem) arise in the Newton linearisation of the Navier-Stokes equations as seen
In the previous section we have demonstrated that relaxation methods annihilate high frequency error components faster than low frequency ones. Their slow rate of convergence is due to the relatively large eigenvalues of the iteration matrix \(|\lambda(G_J)| \approx 1\) associated with low frequency errors. It can be shown that some of the low frequency error components on a particular grid become progressively high frequency on a coarser grid \([215, p. 414]\). Then it is convenient to apply only a fixed number of iterations of a relaxation method to solve a system of linear equations on a fine grid, thus annihilating the high frequency errors on that grid, then transfer the system onto a coarse grid and continue with high frequency error reduction by a relaxation method on that grid (as seen in \((3.3.22)\)), improving the rate of convergence of the overall solver.

We formalise this procedure in the context of finite element discretisations. First we set up two nested grids, let the finite-dimensional space \(X^h_0\) be the set of FE basis functions associated the uniform fine grid \(\Omega_h\) with the mesh parameter \(h\), and the finite-dimensional space \(X^H_0\) be the set of FE basis functions associated with the uniform coarse grid \(\Omega_H\) with the mesh parameter \(H\), such that \(X^H_0 \subset X^h_0\). For simplicity, assume \(H = 2h\) so that if \(X^{2h}_0\) has dimension \(N_{2h}\), then the dimension of \(X^h_0\) is approximately \(N_h \approx 2^d N_{2h}\), where \(d\) is the spatial dimension. To facilitate the transfer of data between the two grids, we define two inter-grid operators. The prolongation (interpolation) operator, \(I^{2h}_h : X^{2h}_0 \to X^h_0\), map functions from \(X^{2h}_0\) to \(X^h_0\). The reverse process is defined by the restriction operator, \(I^h_{2h} : X^h_0 \to X^{2h}_0\). The prolongation operator must satisfy \(I^{2h}_hv^{2h}_j = v^h_j \in X^h_0\) for all functions \(v^{2h}_j \in X^{2h}_0\). The inclusion \(X^{2h}_0 \subset X^h_0\), implies that \(\psi_{2h,j} = \sum_{i=1}^{N_h} p_{ij} \psi_{h,i}\) for some \(P = [p_{ij}], i = 1, \ldots, N_h, j = 1, \ldots, N_{2h}\). The matrix \(P\) is referred to as the prolongation matrix associated with \(I^{h}_{2h}\) (see \([224, p. 82]\)). That is, if \(v^{2h}_j \in X^{2h}_0\) is associated with the coefficient vector \(\vec{v}^{2h}_j \in \mathbb{R}^{N_{2h}}\), then \(I^{h}_{2h}v^{2h}_j = v^h_j \in X^h_0\) is associated with the coefficient vector \(\vec{v}^h_j = P\vec{v}^{2h}_j\). The coefficients of \(P\) are dependent on the degree of the FE basis function, examples are given in \([106, pp. 58–79]\) and \([68, pp. 89–91]\). The restriction operator is defined as the transpose of the prolongation matrix \([224, pp. 108–109]\) and \([68, p. 91]\), that is, the restriction matrix \(R = P^T\) defines the restriction operator \(I^h_{2h}\) from \(X^h_0\) to \(X^{2h}_0\). There are other choices for restriction, the simplest is injection where the values from the fine grid is directly passed to the coarse grid. Thus, we can use the finite element basis functions introduced in Section 2.4.1 to interpolate between the two grids. Let the discrete operators \(A_h\) and \(A_{2h}\) be associated with the fine grid \(\Omega_h\) and coarse grid
\( \Omega_{2h} \), respectively. We now introduce the two-grid method, described in Algorithm 3.1.

**Algorithm 3.1: Two-grid method** [35, p. 37]

1. Choose initial iterate \( \bar{x}^{[0]} \);
2. Relax \( \nu_1 \) times on \( A_h \bar{x}_h = \bar{b}_h \) on \( \Omega_h \) (pre-smoothing);
3. Compute the fine grid residual \( \bar{r}_h = \bar{b}_h - A_h \bar{x}_h \);
4. Restrict the residual to the coarse grid by \( \bar{r}_{2h} = I_{2h}^{1/2} \bar{r}_h \);
5. Solve exactly \( A_{2h} \bar{e}_{2h} = \bar{r}_{2h} \) on \( \Omega_{2h} \) (solve for coarse grid correction);
6. Prolong and update \( \bar{x}_h \leftarrow \bar{x}_h + I_{2h}^{1/2} \bar{e}_{2h} \);
7. Relax \( \nu_2 \) times \( A_h \bar{x}_h = \bar{b}_h \) on \( \Omega_h \) with the improved initial guess (post-smoothing);

The smoother used in Algorithm 3.1 may be any relaxation method introduced in Section 3.3.1; however, it must rapidly reduce smooth error components for the PDE, for example, when an SPD system is solved by the conjugate gradient method and AMG preconditioner, one should use the SGS rather than the GS smoother. In Line 2 of Algorithm 3.1 the error that remains in the solution after the application of the smoother is referred to as the *smooth error*, i.e., \( \bar{e}_h = \bar{x}^* - \bar{x}_h \). The matrix \( A_{2h} \) in Line 3 is a projection of \( A_h \) onto the coarse grid \( \Omega_{2h} \), referred to as the *coarse grid operator*. For geometric multigrid \( A_{2h} \) can be obtained by direct discretisation on \( \Omega_{2h} \). In the case of algebraic multigrid, it can be obtained by the Galerkin projection \( A_{2h} = R A_h P = P^T A_h P \). Thus, the quality of the approximation of the two-grid method depends on the error reduction in the iterative process in the subspace \( X_0^h \): \( \bar{r}_h^{[\nu_1]} = A_h \bar{e}_h^{[\nu_1]} = A_h (I_h - \tilde{M}_h^{-1} A_h)^{\nu_1} \bar{e}_h^{[0]} \); and the (approximate) inverse operator for the subspace \( X_0^{2h} \), \( P^T A_h P \bar{e}_{2h} = \bar{r}_{2h} \), and on the quality of the interpolation operator (how accurately is smooth error represented on the coarse grid). The solution error after the \( i \)th application of the two-grid method is

\[
\bar{e}_h^{[i+1]} = (I - \tilde{M}_h^T A_h)^{\nu_2} (A_h^{-1} - P A_{2h}^{-1} P^T) A_h (I_h - \tilde{M}_h^{-1} A_h)^{\nu_1} \bar{e}_h^{[i]} = G_{TG}(\nu_1, \nu_2) \bar{e}_h^{[i]},
\]

where \( G_{TG}(\nu_1, \nu_2) \) defines the two-grid iteration matrix for Algorithm 3.1 (cf. \( 3.3.21 \)). To establish the convergence of the two-grid scheme, it is sufficient to verify two conditions. The smoothing property is defined by \[106, pp. 116–117\], \[68, p. 95\]:

\[
\left\| A_h (I_h - \tilde{M}_h^{-1} A_h)^{\nu_1} \bar{y}_h \right\| \leq \eta(\nu_1) h^\alpha \left\| \bar{y}_h \right\|_A \quad \eta(\nu_1) \to 0, \text{ as } \nu_1 \to \infty,
\]

for all \( \bar{y}_h \in \mathbb{R}^N \), where \( \eta(\nu_1) \) and \( \alpha \) depends on PDE and the smoother, see \[106\].
The approximation property is given by [106, pp. 135–137], [68, p. 96]:

\[
\| (A_h^{-1} - PA_h^{-1} P^T) \bar{y} \|_A \leq Ch^\beta \| \bar{y} \|,
\]

(3.3.26)

for all \( \bar{y}_h \in \mathbb{R}^N \), where \( \beta \) depends on the spatial dimension. Both \( \eta \) and \( C \) are constants independent of the discretisation parameter \( h \).

One can deduced that, as it stands, the two-grid correction scheme is computationally sub-optimal. The direct solver in Line 5 of Algorithm 3.1 will dominate the execution time. However, recursive application application of this process can lead to a computationally optimal solver, where at each coarse mesh we apply a fixed number of iterations of a relaxation method to annihilate high frequency errors revealed at that level of coarseness, then move on to a coarser mesh. This process is known as a V-cycle. The final solution is obtained by adding the corrections of each level of the mesh.

To perform a V-cycle, one must construct a series of coarse grid operators. For uniform or regular grid discretisation the coarse grid operator can be constructed from the mesh itself in a geometrically natural way since hierarchy of meshes are easy to obtain. This is the geometric multigrid approach. To generalise the multigrid approach, the construction of the coarse grid should not depend on geometric information, only on some properties of the matrices. That is, we must restrict the fine space \( X_h^0 \) to a ‘coarse’ space \( X_H^0 \) with an interpolation operator \( I_h^H : X_h^0 \rightarrow X_H^0 \). This process is referred to as the coarsening algorithm.

Let \( \Omega^{[1]}, \Omega^{[2]}, \ldots, \Omega^{[m]} \), be a series of grids with \( \Omega^{[1]} \) being the finest and \( \Omega^{[m]} \) be the coarsest. Coarsening algorithms will select a subset of points from \( \Omega^{[k]} \) to form \( \Omega^{[k+1]} \), denote these points C-points (coarse points). The points not selected to be in \( \Omega^{[k+1]} \) are denoted F-points (fine points). When selecting the coarse grid points, we seek the unknowns \( x_i \) which can be used to represent values of nearby unknowns \( x_j \). If the value of the unknown \( x_j \) is important in determining the value of \( x_i \), the we say \( i \) depends on \( j \), or equally, we can say \( j \) influences \( i \). We summarise the Ruge-St"uben coarsening procedure which is used for all serial computations in Chapter 6.

Ruge-St"uben coarsening. The Ruge-St"uben coarsening algorithm [131, p. 4] determines the set of points \( S_i \) which the unknowns \( x_i \) depends on by:

\[
S_i \equiv \left\{ j \neq i \mid a_{ij} \geq \theta \max_{k \neq i} (-a_{ik}) \right\},
\]

(3.3.27)
and set of points \(j\) that are influenced by \(i\) is defined as \(S_i^T \equiv \{ j \mid i \in S_j \}\). The selection of coarse points is based on two heuristics:

1. For each \(F\)-point \(i\), each \(j \in S_i\) is either a \(C\)-point or \(S_j \cap C_i \neq \emptyset\), where \(C_i\) is the set of coarse points associated with the \(F\)-point \(i\).

2. The set of coarse-points \(C\) should be the maximal set such that no points in \(C\) depends on another point.

If both criteria are satisfied simultaneously, we have two-pass Ruge-Stüben coarsening. If only the first criteria is satisfied, then we have one-pass coarsening. In Section 3.5.5 we introduce parallel coarsening strategies.

### 3.3.3 Projection Methods

In this section we follow the discussion by Saad in [215, pp. 129–131, 151–167]. The objective of projection methods is to extract an approximate solution of a linear system \( (3.1.1) \) from a subspace of \( \mathbb{R}^N \). The candidate approximant subspace of dimension \( M \leq N \), \( K \subset \mathbb{R}^M \) is the subspace of potential solutions. In general, there must be \( M \) constraints imposed to extract such an approximation. Thus, we impose the orthogonality condition \( \bar{r} = \bar{b} - A\bar{x} \perp L \), where \( L \) of dimension \( M \) is referred to as the subspace of constraints. This is known as the Petrov-Galerkin approach.

Let \( \bar{x}^{[k]} \) be the approximate solution to \( (3.1.1) \) and \( K \) and \( L \) be two \( M \)-dimensional subspaces of \( \mathbb{R}^N \) as described previously, then the projection method seeks:

\[
\bar{x}^{[k]} \in K \text{ such that } \bar{b} - A\bar{x}^{[k]} \perp L. \tag{3.3.28}
\]

We exploit the knowledge of an initial guess \( \bar{x}^{[0]} \) by seeking an approximation from the affine space \( \bar{x}^{[0]} + K \), and \( (3.3.28) \) can be re-written as follows: Find \( \bar{x}^{[k]} \in \bar{x}^{[0]} + K \) such that \( \bar{b} - A\bar{x}^{[k]} \perp L \). If we define the initial residual as \( \bar{r}^{[0]} = \bar{b} - A\bar{x}^{[0]} \), and write \( \bar{x}^{[k]} = \bar{x}^{[0]} + \delta \), then \( \bar{b} - A\bar{x}^{[k]} \perp L \) becomes \( \bar{r}^{[0]} - A\delta \perp L \). We can then reformulate the approximate solution as: \( \bar{x}^{[k]} = \bar{x}^{[0]} + \delta \) and \( \langle \bar{r}^{[0]} - A\delta, \bar{w} \rangle = 0 \), for \( \delta \in K \) and \( \forall \bar{w} \in L \). This is a general form of a basic projection step. In most projection methods, a sequence of subspaces \( K \) and \( L \) are generated for each new projection step. The two broad classes of projection methods are orthogonal projection methods (where \( K = L \)), or oblique (where \( K \neq L \)).
3.3.4 Krylov Methods

Krylov subspace methods are considered currently to be among the most important iterative techniques available for solving large sparse linear systems. Krylov methods are projection methods where the search subspace $\mathcal{K}_M$ is the Krylov subspace defined by

$$\mathcal{K}_M(A, \bar{r}[0]) = \text{span}\{\bar{r}[0], A\bar{r}[0], A^2\bar{r}[0], \ldots, A^{M-1}\bar{r}[0]\},$$  

where $\bar{r}[0] = \bar{b} - A\bar{x}[0]$ [251, p. 27]. For simplicity, the Krylov subspace $\mathcal{K}_M(A, \bar{r}[0])$ will be denoted by $\mathcal{K}_M$. The approximations $\bar{x}[M]$ generated by the Krylov subspace methods are of the form

$$A^{-1}\bar{b} \approx \bar{x}[M] = \bar{x}[0] + q_{M-1}(A)\bar{r}[0],$$

where $q_{M-1}(A)$ is $(M - 1)$th degree matrix polynomial which approximates the matrix $A^{-1}$. In the simplest case $\bar{x}[0] = \bar{0}$ and we have

$$A^{-1}\bar{b} \approx q_{M-1}(A)\bar{b}. \quad (3.3.30)$$

We observe from the relation (3.3.30) that all Krylov methods provide the same type of polynomial approximation. Different choices of the subspace of constraints $\mathcal{L}$ and the ways the linear system is preconditioned lead to different versions of the Krylov methods. The common choices are $\mathcal{L}_M = \mathcal{K}_M$, $\mathcal{L}_M = A\mathcal{K}_M$ and $\mathcal{L}_M = \mathcal{K}_M(A^T, \bar{r}[0])$. The dimension of the Krylov subspace $\mathcal{K}_M$ and the space of constraints $\mathcal{L}_M$ increase by one at each step of the approximation process.

3.3.5 Generalized Minimal Residual Method (GMRES)

The Generalized Minimal Residual method (GMRES), proposed by Saad and Schultz in [213], is a Krylov method for the solution of sparse linear systems with general nonsymmetric matrices. Such matrices often arise from the discretisation of non-self-adjoint problems, the simplest example in fluid mechanics is the convection-diffusion problem [68, pp. 113 - 161]. It is also applicable to both symmetric and nonsymmetric indefinite systems, such as the one obtained from the application of mixed FEM to the Stokes and the Navier-Stokes systems, see Section 2.4. The method is based on the choice $\mathcal{K} = \mathcal{K}_M$ and $\mathcal{L} = A\mathcal{K}_M$, where $\mathcal{K}_M$ is the $M$th Krylov subspace. The $M$th approximation vector $\bar{x}[M]$ in $\bar{x}[0] + \mathcal{K}_M$ can be written as $\bar{x}[M] = \bar{x}[0] + V_M\bar{y},$ where
\( y \in \mathbb{R}^M \) and the columns of matrix \( V_M \in \mathbb{R}^{N \times M} \) represent the orthogonal basis for the matrix \( K_M \), i.e. \( K_M = \text{span} \{ \vec{v}^{[0]}, \vec{v}^{[1]}, \ldots, \vec{v}^{[M-1]} \} \). The orthonormal basis \( \{ \vec{v}^{[j]} \}_{j=0}^{M-1} \) for \( K_M \) is generated using Arnoldi’s method \([96, p. 38]\), presented between Line 4 to Line 10 of Algorithm 3.2. Thus, we have \( V_M = [\vec{v}^{[0]}, \vec{v}^{[1]}, \ldots, \vec{v}^{[M-1]}] \in \mathbb{R}^{N \times M} \). Denote by \( \bar{H}_M \) the \((M+1) \times M\) Hessenberg matrix whose nonzero entries \( h_{ij} \) are defined by Arnoldi’s algorithm and the matrix \( H_M \) is obtained from \( \bar{H}_M \) by deleting its last row. Then the following relations hold:

\[
\begin{align*}
AV_M &= V_M H_M + \bar{w}_M \bar{e}_M^T = V_{M+1} \bar{H}_M, \\
V_M^T A V_M &= H_M,
\end{align*}
\]  

where \( \bar{e}_M \) is the \( M \)th column of an \((M+1) \times (M+1)\) identity matrix and \( \bar{w}_M \) is a vector orthonormalised against all previous \( \bar{v}^{[j]} \) \((i < M)\) by the standard Gram-Schmidt procedure \([215, p. 161]\). Using the relation (3.3.31) the \( M \)th residual \( \bar{r} \) becomes

\[
\bar{r}^{[M]} = \bar{b} - A \bar{x}^{[M]} = \bar{r}^{[0]} - AV_M \bar{y} = V_{M+1} (\beta \bar{e}_0 - \bar{H}_M \bar{y}),
\]

where we set \( \beta = \|\bar{r}^{[0]}\|_2 \). The GMRES algorithm seeks at each iteration a unique vector in \( \bar{x}^{[0]} + K_M \) that minimizes the norm of the residual

\[
J(\bar{y}) = \| \bar{b} - A \bar{x} \|_2 = \| \bar{b} - A(\bar{x}^{[0]} + V_M \bar{y}) \|_2 = \| \beta \bar{e}_0 - \bar{H}_M \bar{y} \|_2.
\]  

(3.3.33)

In other words, \( \bar{y} \) is the minimiser of the \((M+1) \times M\) least squares problem

\[
\min_{\bar{y}} \| \beta \bar{e}_0 - \bar{H}_M \bar{y} \|_2,
\]  

(3.3.34)

and the approximation vector is obtained as \( \bar{x}^{[M]} = \bar{x}^{[0]} + V_M \bar{y}_M \). The least square problem (3.3.34) can be efficiently solved by using Givens rotations to decompose the Hessenberg matrix \( \bar{H}_M \) into an \((M+1) \times M\) upper triangular matrix \( R \) and an \((M+1) \times (M+1)\) unitary matrix \( Q \), then solve \( R \bar{y} = \beta Q \bar{e}_0 \) for \( \bar{y} \), see \([215, pp. 174–178]\) for an example. This is represented by Line 12 in Algorithm 3.2.

If we assume the coefficient matrix \( A \) to be nonsingular, then we observe that Algorithm 3.2 will only break down at the \( j \)th step of the Arnoldi’s method if \( \bar{w}_j = \bar{0} \), which results in division by zero in Line 10 if the \texttt{go to} statement was not present. In this case, the next Arnoldi vector cannot be generated. However, this implies that the residual is zero, thus the solution at this step is exact, we note that this is a
very unlikely scenario. The GMRES method requires storage for the orthonormal basis $V_M$ (which dimension increases by 1 at each iteration). This affects the optimal cost property of GMRES per iteration. Both the computational cost and the storage requirement of the GMRES method is $O(j \cdot N)$, where $N$ is the size of the linear system [215, p. 167]. This can become a limiting factor in the applicability of GMRES when the iteration counts are large, this is one of the reasons why a good preconditioner is essential. Other (computationally cheaper) variants of GMRES exist, they are typically based on modifying the method that generates the basis set. The approach presented in Algorithm 3.2 is often referred to as the Full Orthogonalisation Method (FOM). To reduce computational and storage cost of the standard GMRES algorithm, one can restart the creation of the Krylov basis set periodically (every $m$ iterations), this is referred to as restarted GMRES (GMRES($m$)) [251, p. 66]. In exchange for a computationally cheaper algorithm, the restarted version always has poorer convergence properties than the original. In some cases it may not converge at all whilst the full GMRES algorithm is will always converge in at most $N$ steps (where $N$ is the size of the linear system). In the quasi-GMRES (QGMRES) method, the Arnoldi process is replaced with an incomplete orthogonalisation method (IOM), where the number of basis vectors which $	ilde{v}_j$ is to be orthogonalised against is reduced, see [215, p. 180].

---

3 The memory requirement for storing $V_j$ at the 1000th iteration with problem size $N = 1 \times 10^6$ is 8GB.
3.3.6 Other Krylov Methods

In this section, we review and contrast a few well known Krylov subspace methods. We follow the description of the methods presented in [215, 251], where they are discussed in more detail.

If the coefficient matrix is symmetric (but possibly indefinite), then the Hessenberg \( \bar{H}_M \) (defined by Line 11 of Algorithm 3.2) is tridiagonal, and usually denoted as \( T_M \) in the literature. This is immediate from (3.3.31) if \( A \) is symmetric, then \( V^T AV = \bar{H}_M \) is also symmetric, since \( \bar{H}_M \) is upper Hessenberg by construction, then it must be tridiagonal. This means that the Krylov basis vectors can be constructed via a short-term recurrence relation. Exploiting the tridiagonal structure of \( \bar{H} \) leads to the Minimal Residual (MINRES) algorithm [251, pp. 84–86]. In contrast to GMRES, where the Krylov basis set increases in dimension at each iteration, the MINRES algorithm only needs to save three vectors. This also makes MINRES more vulnerable to rounding errors for ill conditioned coefficient matrices [228]. However, MINRES is suitable for symmetric indefinite systems, such as the Stokes system [68, p. 286].

If the coefficient matrix \( A \) is symmetric positive definite (SPD), and we construct the \( M \)th iterate \( \bar{x}^{[M]} \) so that the \( M \)th residual \( \bar{r}^{[M]} = \bar{b} - A\bar{x}^{[M]} \) is orthogonal to all vectors in \( \mathcal{K}_M \), this would lead us to the Conjugate Gradient (CG) method. Details of the CG algorithm can be found in [215, pp. 196–200, 251, pp. 39–38]. In the CG algorithm, the new iterate is computed as \( \bar{x}^{[M+1]} = \bar{x}^{[M]} + \alpha^{[M]} \bar{p}^{[M]} \), where \( \bar{p}^{[M]} \) is a vector from the set of orthogonal A-conjugate vectors ((\( \bar{p}^{[i]} \))^T A\( \bar{p}^{[j]} \) = 0, for \( i \neq j \)) that represent search directions and \( \alpha^{[M]} \) is the parameter. The search directions \( \{\bar{p}^{[i]}\}_{i=0}^{M-1} \) form a Krylov basis set for \( \mathcal{K}_M \).

The CG algorithm requires storage for the coefficient matrix \( A \) and four vectors. Thus, it has \( O(N) \) storage requirements. One iteration of the algorithm requires one sparse matrix-vector product (SpMV), two inner products (IP) and three vector updates. The number of operations in each iteration of CG is dominated by the SpMV, which scales as \( O(N) \) for sparse FEM matrices. The CG method requires that \( A \) is symmetric so that the residual vectors can be made orthogonal with short recurrences, this is not possible if \( A \) is nonsymmetric [77]. Thus, if the CG method is

---

4Here, short-term implies that the update of a vector \( \bar{w}^{[j]} \) depends on only two or three previous vectors.

5Vector updates refer operations of the form \( \bar{y} := \bar{y} + a\bar{x} \), where \( \bar{y} \) and \( \bar{x} \) are vectors and \( a \) is a scalar.
preconditioned, then the preconditioned matrix needs to be SPD for CG to be applicable, see [68, p. 79]. The GMRES method does not require $A$ to be symmetric and retains orthogonality of the residuals with long recurrences, which leads to increased computational cost and storage requirement per iterations.

The Biconjugate Gradient (BiCG) method [215, pp. 234–236, 251, pp. 95–97] does not require $A$ to be symmetric and is based on two mutually orthogonal sequences of Krylov subspaces. The BiCG method seeks the approximation $\tilde{x}^{[M]}$ in the usual Krylov subspace $\mathcal{K}_M(A, \tilde{v}^{[0]})$ which is orthogonal to the dual Krylov subspace $\mathcal{K}^*_M(A, \tilde{w}^{[0]}) = \mathcal{K}_M(A^T, \tilde{w}^{[0]})$. This entails two sequences of search directions $\{\tilde{p}^{[i]}\}_{i=0}^{M-1}$ and $\{\tilde{p}^*[i]\}_{i=0}^{M-1}$ which form a basis set for $\mathcal{K}_M$ and $\mathcal{K}^*_M$, respectively. This implies that the BiCG algorithm produces a sequence of biorthogonal residual vectors, $\tilde{r}^{[j]} \cdot \tilde{r}^*[i] = \delta_{ij}$, where $\delta_{ij}$ is the Kronecker delta. Each iteration of the BiCG method requires two SpMV with $A$ and $A^T$, two IPs and five vector updates. However, we do not solve the dual system with the coefficient matrix $A^T$. Thus, the storage requirement and computational cost are twice that of the CG method. If the coefficient matrix is SPD (therefore $A = A^T$), then the CG method produces the same sequence of approximations $\tilde{x}^{[j]}$ as BiCG but at half the computational cost [251, p. 96].

To remedy this, transpose-free variations of BiCG exists such as the Conjugate Gradient Squared method (CGS) [230]. The CGS method is based on the observation that both sets of the BiCG residual vectors at the $j$th iteration can be written as a product of the initial residual and a $j$th degree polynomial $\phi_j$ with $\phi_j(0) = 1$.

This implies that $\tilde{r}^{[j]} = \phi_j(A)\tilde{r}^{[0]}$ and we can interpret $\phi_j(A)$ as an operator which transforms the initial residual to the $j$th residual vector. This suggests an algorithm which produces a sequence of iterates whose residuals $\tilde{r}^{[j]}$ satisfy $\tilde{r}^{[j]} = \phi_j^2(A)\tilde{r}^{[0]}$, i.e. the operator $\phi_j(A)$ is applied twice in succession to the initial residual. The derivation of the CGS algorithm based on this approach can be found in [215, pp. 241–244, 251, pp. 102–104]. Each iteration of the CGS algorithm requires two SpMV with $A$, two IPs, and six vector updates. Thus the computational cost is roughly the same as BiCG [18, p. 23] but with roughly half the storage requirement since the computations does not involve $A^T$. Due to the squaring of the polynomials $\phi_j(A)$, the CGS method generally converges faster than BiCG, requiring only half the number of iterations in some cases [251, p. 105]. The squaring process also means that the CGS method is more susceptible to rounding errors which may lead to irregular convergence (where the norms of successive residual vectors differ by orders of magnitude, and may even
be bigger than the norm of the residual vector), see [251, Chapter 8].

The Biconjugate Gradient Stabilized (BiCGSTAB) method introduced by van der Vorst [250] is a variant of CGS developed to remedy this situation. The BiCGSTAB produces a sequence of residuals of the form $\bar{r}^{[j]} = \psi_j(A)\phi_j(A)\bar{r}^{[0]}$, where $\phi_j(A)$ is the polynomial associated with the BiCG residual and $\psi_j(A)$ is a $j$th degree polynomial constructed to stabilise the convergence behaviour of the BiCG algorithm. The derivation can be found in [215, pp. 244–247, 251, pp. 133–136]. Each iteration of the BiCGSTAB method requires two SpMV with $A$, four IPs and six vector updates. The storage requirement is roughly the same as for the CGS method. The BiCGSTAB method may stagnate, for example, when $A$ has eigenvalues that have large imaginary parts [18, p. 25, 251, p. 139]. This is remedied with the BiCGSTAB2 method introduced in [104], this algorithm works with two search directions. This approach is generalised by the BiCGSTAB($\ell$) algorithm discussed in [227].

In summary, the CG, BiCG, CGS and BiCGSTAB methods all utilise short recurrence relations to create an orthonormal Krylov basis, thus in addition to the coefficient matrix $A$, they require the storage of a small fixed number of vectors of size $N$ (and storage for $A^T$ in the case of BiCG). The number of operations per iteration is dominated by SpMVs, giving the computational cost per iteration of $O(N)$. By contrast, the storage requirement and number of operations per iteration of the GMRES method grow with the number of iterations.

When the coefficient matrix is SPD, the CG method is preferable as it has the lowest storage requirements and computational cost of all the methods discussed in this section. For nonsymmetric but positive definite matrices, then BiCGSTAB can be used. However, it is less robust than the GMRES algorithm [215, p. 274]. If memory is not an issue and the problem converges in a modest number of iterations, then the GMRES method is preferred. If the problem requires a large number of iterations to converge then the BiCGSTAB algorithm may be faster in terms of the wall-clock time.

There exist many other Krylov subspace methods and many variations of the methods discussed in this section. A comprehensive overview of such methods can be found in [215] and references therein. For any Krylov solver, it is possible to construct a coefficient matrix for which that method performs poorly. Nachtigal, Reddy and Trefethen [184] demonstrated that for any class of Krylov subspace methods, there are problems for which a given method performs optimally, and there are problems for which they do not. Thus, finding a good preconditioner for a class of problems is
equally important as the choice of an actual Krylov method in achieving the ‘optimal’ linear solver, this is the topic of Section 3.4. In this project we only use GMRES, the relatively modest number of iteration counts reported in Chapter 6 justifies this choice.

3.4 Preconditioners

Krylov subspace methods have a sound theoretical foundation, but suffer from a well recognised weakness - the lack of robustness. If Krylov subspace methods are applied to the solution of a linear system such as \( (2.3.38) \), they tend to converge very slowly or may not converge at all. The residual at each iteration is connected with the value of the polynomial \( q_{M-1} \) from \((3.3.30)\) calculated at the eigenvalues of the coefficient matrix \( A \). On the one hand, coefficient matrices obtained from FE discretisation are known to be ill-conditioned, with eigenvalues spread across large parts of the complex plane. To make \( q_{M-1}(A) \) small simultaneously at all \( \lambda_i(A) \) for \( i = 1, \ldots, N \), we need a polynomial \( q_{M-1}(A) \) of a relatively high degree (to be a good fit), resulting in a relatively high number of iterations needed for the convergence. This situation can be improved if the eigenvalues of the coefficient matrix are tightly clustered. This can be achieved by preconditioning which can improve both the rate of convergence and robustness of the iterative techniques. The aim of preconditioning is to generate a linear system equivalent to the one we wish to solve, in the sense that they have the same solution, but for which the Krylov solvers converge more rapidly. If we can design iterative solvers that are robust and capable of converging to a solution at an optimal computational cost, then it is always preferable to use iterative solvers to sparse direct solvers (except perhaps for very small problems of up to few thousand unknowns). The use of iterative solvers is essential for three-dimensional problems.

Recalling the linear system \( A\bar{x} = \bar{b} \) in \((3.1.1)\) the general idea of preconditioning is to construct a matrix (or, more generally, an operator), \( P \), called a preconditioner or a preconditioning matrix such that it is a good approximation to the coefficient matrix \( A \) in the spectral sense. The linear system

\[
P^{-1}A\bar{x} = P^{-1}\bar{b}
\]  

(3.4.1)

should be much easier to solve iteratively than the original system \((3.1.1)\). To enable this, in light of the previous discussion regarding the convergence of Krylov methods,
we require the spectrum of the preconditioned matrix \( \mathcal{P}^{-1}A \) to have either:

- a small number of distinct eigenvalues with large multiplicity, or
- to consist of a small number of tightly bounded clusters.

In either case the coefficient matrix should not be, or close to, singular. If it is possible to precondition the coefficient matrix to have its spectrum bounded independently of the discretisation parameter (\( \ell h \) in the FEM case) or any other problem parameters, the iterative solver will converge to a desired accuracy in an asymptotically constant number of iterations as \( N \to \infty \). To preserve the optimal scaling of the computational cost per iteration and storage requirements of the non-preconditioned Krylov methods, we require in addition that the cost of assembling and applying the preconditioner scales linearly with respect to the problem size. To achieve this, the additional storage requirements needed for the implementation of the preconditioner should also be \( \mathcal{O}(N) \) and should not be significantly larger than those for the coefficient matrix. The number of operations involved in the assembly of \( \mathcal{P} \) should be comparable to the cost of assembling the matrix \( A \) (or at least not prohibitive) and the cost of applying the action of the inverse of \( \mathcal{P} \) to a vector, that is, to compute \( \mathcal{P}^{-1} \vec{z} \) for a given \( \vec{z} \) at each Krylov iteration should also scale linearly with the problem size.

There are three different ways of applying the preconditioner \( \mathcal{P} \) to the system (3.1.1). The transformed system (3.4.1) is known as left preconditioning and is the default implementation in \texttt{oomph-lib}. An alternative implementation is the so-called right preconditioning, in which the modified system to be solved is

\[
(A\mathcal{P}^{-1})[\mathcal{P}\vec{x}] = \vec{b}.
\]

In this case the right hand side vector is not affected by the preconditioning. It is possible to combine left and right preconditioning, for example if it is possible to factorise the preconditioner, \( \mathcal{P} = \mathcal{P}_1\mathcal{P}_2 \), then to obtain symmetric preconditioning, the modified system to be solved is

\[
\mathcal{P}^{-1}_1A\mathcal{P}^{-1}_2\mathcal{P}_2\vec{x} = \mathcal{P}^{-1}_1\vec{b}.
\]

All three different approaches of the preconditioner application give the same eigenvalue distribution for the preconditioned matrix \( \mathcal{P}^{-1}A \), i.e. the eigenvalue distribution for \( \mathcal{P}^{-1}A \) and \( A\mathcal{P}^{-1} \) is the same, but the eigenfunctions are different. This may have
an impact on the convergence of an iterative solver, since the solution error is decomposed along the eigenfunctions of the coefficient matrix, and at each Krylov iteration some of the error components are annihilated and thus the overall error is reduced. The situation is generally worse for non-normal matrices (e.g. the convection-diffusion problem [68, p. 113]) than for problems with complete and orthogonal eigenvector sets (e.g. the self-adjoint problems).

If the stopping criteria of Krylov solvers is based on errors, care must be taken since for left preconditioning, $\|P\bar{x} - P\bar{x}^{|k}\|_2$ can be much smaller than $\|\bar{x} - \bar{x}^{|k}\|_2$. In this project, we use the right preconditioning with GMRES so that the norm (3.3.33) to be minimized do not depend on the preconditioner. The GMRES algorithm with right-preconditioning is based on solving

$$AP^{-1}\bar{u} = \bar{b}, \quad \bar{u} = P\bar{x}. \quad (3.4.4)$$

The new variable $\bar{u}$ is never invoked explicitly [215, pp. 269-270], and the Arnoldi’s loop builds an orthogonal basis of the right-preconditioned Krylov subspace

$$\text{span}\left\{r^{[0]}, AP^{-1}r^{[0]}, \ldots, (AP^{-1})^{M-1}r^{[0]}\right\}.$$  

### 3.4.1 General Preconditioners

General-type preconditioners are designed from the algebraic properties of the coefficient matrix with no information about the specific underlying physical problem. These can be applied to a wide class of problems, but with varying rates of success. Typically they do not lead to optimal solvers, but are simple and their implementations can be found in many software libraries such as HYPRE [131, 137], PETSc [13], HSL [135] and NAG [238]. The main idea behind general preconditioners is to generate a sparse approximation of the coefficient matrix inverse. In this section we discuss some of the best known algorithms which belong to this class.

The iteration (3.3.16) is constructed to solve the linear system

$$(I - G)\bar{x} = \bar{f}, \quad (3.4.5)$$

then substituting the expression for $G$ into (3.4.5) yields $\bar{M}^{-1}A\bar{x} = \bar{M}^{-1}\bar{b}$, i.e. the preconditioned system with the preconditioner $\bar{M}$. Thus, we can view $\bar{M}$ associated with the splitting $A = \bar{M} - \bar{N}$ as a preconditioner for the system $A\bar{x} = \bar{b}$. For
the simplest case, the Jacobi preconditioner corresponds to \( \tilde{M}_J = D \), which is just the diagonal of the coefficient matrix. The Gauss-Seidel preconditioner is defined by \( \tilde{M}_{GS} = D - L \) and the SSOR preconditioner by \( \tilde{M}_{SSOR} = (D - \omega L)D^{-1}(D - \omega U) \), where \( \omega \) corresponds to the damping factor introduced in Section 3.3.1. All these preconditioners are optimal in terms of storage and computational cost of their assembly and the application to a vector.

Other well known general-type preconditioners are Incomplete LU factorization Preconditioners (ILU) \[215\], pp. 301–307], where the coefficient matrix is approximated by \( A \approx \tilde{L}\tilde{U} \) where \( \tilde{L} \) and \( \tilde{U} \) are lower and upper triangular matrices obtained by omitting certain fill-in. There are two main strategies for handling fill-in, numerical and position based. The numerical approach utilise a numerical drop tolerance strategy where the fill-in is dropped if it is too small, for example, a fill-in is dropped if its magnitude is smaller than the 2-norm of the row multiplied by a drop tolerance \( \tau \). The second approach is usually based on the level of fill. If the sparsity pattern of \( \tilde{L} \) and \( \tilde{U} \) is the same as the sparsity pattern of \( A \), this corresponds to zero fill-in, denoted as ILU(0) \[215\], pp. 307–310].

Standard ILU factorisation typically employ a Gaussian elimination type algorithm to compute the \( LU \) factors. This can be problematic for indefinite matrices due to zero pivots. To remedy this, one may compute the sparse approximation of the inverse explicitly, without solving a system of linear equations. These are known as sparse approximate inverses (SPAI). For a comparative study of well known algorithms for SPAI computation, see \[28\]. One of the well known approximate inverse method computes the approximation \( M \approx A^{-1} \) such that \( M \) minimises the Frobenius norm of \( I - AM \):

\[
\min_{M \in S} \| I - AM \|_F,
\]

where \( S \) is a suitably constrained set of sparse matrices. The constrained optimisation problem \( (3.4.6) \) can be decomposed into \( N \) independent least-squares problems, one for every column since \( \| I - AM \|_2^2 = \sum_{j=1}^{N} \| \tilde{e}_j - A\tilde{m}_j \|_2^2 \). This makes SPAI preconditioners inherently parallel and preferred to sequential ILU methods in a parallel computing environment. The nonzero entries in \( M \) must be constrained to some sparsity pattern defined by the set \( S \), otherwise the approximate inverse may be computationally expensive to apply and thus would not make a good preconditioner. An approach to impose the sparsity pattern of the approximate inverse automatically is introduced
3.4. PRECONDITIONERS

There are many other general preconditioning techniques; refer to [251, pp. 173-203, 215, pp. 283–405] and [46] for a comprehensive overview. By contrast, one can also develop specialised preconditioners, which work effectively only for a specific problem or a class of problems. Optimal preconditioners are inevitably problem-specific, where the optimal eigenvalue distribution of the preconditioned matrix is achieved through the knowledge of the spectral properties of a specific class of coefficient matrices. A particularly efficient framework for designing efficient preconditioners for FE discretisations of systems of PDEs arising in the multi-physics setting is block preconditioning.

3.4.2 Block Preconditioners

Block preconditioning is based on the assumption that the appropriate grouping of the discrete unknowns in the system (where all the degrees of freedom corresponding to a particular unknown type are grouped together) leads to a natural blocking of the coefficient matrix, where the individual blocks are the discrete representations of the continuous scalar operators in the PDE system. The block preconditioner usually takes the form of a block diagonal or block lower/upper triangular matrix. The action of such a preconditioner is achieved by forward/backward block substitution, where the inversion of the principal diagonal blocks (that corresponds to the scalar operators or the associated Schur complements) is performed inexactly using optimal methods such as algebraic multigrid.

For example, the two-dimensional Navier-Stokes Jacobian [2.3.38] can be subdivided into velocity and pressure unknowns,

\[
\mathcal{F} = \begin{bmatrix} F & B^T \\ B & O \end{bmatrix},
\]

and the velocity unknowns can be further subdivided into their directional components:

\[
F = \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix}.
\]
More generally, for the following $2 \times 2$ block system of linear equations:

$$
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
\bar{x}_1 \\
\bar{x}_2
\end{bmatrix}
=
\begin{bmatrix}
\bar{b}_1 \\
\bar{b}_2
\end{bmatrix},
$$

(3.4.8)

we can have three general types of block preconditioners: block diagonal ($\mathcal{P}_{BD}$), block upper triangular ($\mathcal{P}_{BUT}$), and block lower triangular ($\mathcal{P}_{BLT}$):

$$
\mathcal{P}_{BD} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},
\mathcal{P}_{BUT} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},
\text{ and }
\mathcal{P}_{BLT} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}.
$$

(3.4.9)

The block preconditioners in (3.4.9) are general-type preconditioners, no assumptions about the underlying problem are taken into account. In the next section we discuss problem specific block preconditioners for the Navier-Stokes equations.

### 3.4.3 Preconditioners for the Navier-Stokes System

In this section discuss three block preconditioners developed for Navier-Stokes equations. We derive the least-squares commutator (LSC) preconditioner \cite{72,68,pp.353–354}, which is used in this project, and review the pressure convection-diffusion (PCD) preconditioner \cite{226,68,pp.347–354} and the augmented Lagrangian (AL) preconditioner \cite{29,30}.

#### 3.4.3.1 Pressure Schur Complement Preconditioners

Consider the discrete Navier-Stokes linear system $\mathcal{F}\bar{x} = \bar{b}$ of the form

$$
\begin{bmatrix} F & B^T \\ B & O \end{bmatrix}
\begin{bmatrix}
\bar{u} \\
\bar{p}
\end{bmatrix}
=
\begin{bmatrix}
\bar{f} \\
\bar{g}
\end{bmatrix}.
$$

(3.4.10)

The coefficient matrix in (3.4.10) can be factorised using the block $LU$-decomposition as

$$
\begin{bmatrix} F & B^T \\ B & O \end{bmatrix}
= \begin{bmatrix} I & O \\ BF^{-1} & I \end{bmatrix}
\begin{bmatrix} F & B^T \\ O & -S \end{bmatrix},
$$

(3.4.11)

where $S = BF^{-1}B^T$ is the (pressure) Schur complement operator associated with the block $F$. Where there is no ambiguity, we refer to this simply as the Schur complement.
3.4. PRECONDITIONERS

Taking the block upper triangular matrix $U$ as the preconditioner for $F$, we obtain a preconditioned matrix $L = FU^{-1}$ whose eigenvalues are all identically equal to one, i.e. $U$ is an ideal preconditioner for $F$. This implies that the minimal polynomial of the preconditioned matrix has degree 2, and the Krylov basis subspace is of dimension 3. Thus, applying GMRES to the preconditioned system requires precisely two steps to converge \[2\]. We therefore consider a preconditioning matrix of the form

$$P_{NS} = \begin{bmatrix} F & B^T \\ O & -\tilde{S} \end{bmatrix}. \tag{3.4.12}$$

In the ideal case, $\tilde{S}$ is equal to $S = BF^{-1}B^T$, but in practice the Schur complement is a dense matrix whose assembly, let alone an exact inversion, is completely impractical, and would not lead to an optimal solver. Instead, we seek a spectrally equivalent optimal approximation $\tilde{S} \approx S$. Then the action of an inverse of a preconditioner $P_{NS}$ to a vector is achieved by a block back substitution. We now derive a method for efficient approximation of the Schur complement referred to as the least-squares commutator (LSC) preconditioner. We follow the derivation presented in [68, pp. 353–354]. For this purpose, we consider the dimensional, simplified form of the Navier-Stokes equations (2.1.5) and (2.1.18) with zero body force ($f = 0$), expressed using operator notation for simplicity,

$$\rho \left( \frac{\partial u}{\partial t} + u \cdot \nabla u \right) = -\nabla p + \mu \nabla^2 u \tag{3.4.13}$$

$$\nabla \cdot u = 0. \tag{3.4.14}$$

Define the linear convection-diffusion operator on the velocity space as

$$\mathcal{L}_u = -\mu \nabla^2 + w_h \cdot \nabla, \tag{3.4.15}$$

where $w_h$ is the approximation to the discrete velocity computed at the most recent nonlinear iteration. Suppose that we can define an analogous operator on the pressure space

$$\mathcal{L}_p = (-\mu \nabla^2 + w_h \cdot \nabla)_p, \tag{3.4.16}$$
and assume that the commutator of the convection-diffusion operators (3.4.15) and (3.4.16) with the gradient operator $\nabla$ defined as

$$E = (-\mu \nabla^2 + w_h \cdot \nabla) \nabla - \nabla (-\mu \nabla^2 + w_h \cdot \nabla)_p \nabla$$

(3.4.17)

is small in some sense. The discrete representation of (3.4.17) in terms of FE matrices is given by [68, p. 347]

$$E_h = (M_u^{-1}F)(M_u^{-1}B^T) - (M_u^{-1}B^T)(M_p^{-1}F_p),$$

(3.4.18)

where

$$M_u = [m_{u,ij}], \quad m_{u,ij} = \int_{\Omega} \psi_i \cdot \psi_j \, d\Omega, \quad i, j = 1, \ldots, N_u,$$

(3.4.19)

is the velocity mass matrix,

$$M_p = [m_{p,ij}], \quad m_{p,ij} = \int_{\Omega} \varphi_j \varphi_i \, d\Omega, \quad i, j = 1, \ldots, N_p,$$

(3.4.20)

is the pressure mass matrix, and

$$F_p = [f_{p,ij}], \quad f_{p,ij} = \mu \int_{\Omega} \nabla \varphi_j \cdot \nabla \varphi_i \, d\Omega + \int_{\Omega} (w_h \cdot \nabla \varphi_j) \varphi_i \, d\Omega, \quad i, j = 1, \ldots, N_p,$$

(3.4.21)

is the discrete convection-diffusion operator defined on the pressure space. The assumption that the commutator is small allows us to identify an approximation to the Schur complement matrix by pre-multiplying (3.4.18) by $BF^{-1}M_u$ and post multiplying by $F_p^{-1}M_u$, which yields

$$BF^{-1}B^T \approx BM_u^{-1}B^T F_p^{-1}M_u.$$  

(3.4.22)

Now we derive an approximation to $F_p$ by defining the $j$th column of $F_p$ to solve the weighted least squares problem

$$\min \left\| [M_u^{-1}FM_u^{-1}B^T]_j - M_u^{-1}B^TM_p^{-1}[F_p]_j \right\|_{M_u}.$$
By doing so, we are minimising the individual vector norms of the columns of the discrete commutator separately. The associated normal equations are

\[ M_p^{-1} B M_u^{-1} B^T M_p^{-1} [F_p] = [M_p^{-1} B M_u^{-1} F M_u^{-1} B^T]_{j_1}, \]

which leads to the approximation

\[ F_p = M_p (B M_u^{-1} B^T)^{-1} (B M_u^{-1} F M_u^{-1} B^T). \] (3.4.23)

Then an approximation to the Schur complement matrix is given by substituting (3.4.23) into (3.4.22):

\[ B F^{-1} B^T \approx (B M_u^{-1} B^T)(B M_u^{-1} F M_u^{-1} B^T)^{-1}(B M_u^{-1} B^T). \] (3.4.24)

The matrix \( B M_u^{-1} B^T \) in (3.4.24) is commonly referred to as the consistent pressure-Poisson matrix [99, pp. 550–552] and has the characteristics of a matrix arising from the discretisation of the Poisson problem on the pressure space [68, p. 273], that is, it is spectrally equivalent to the (sparse) pressure Laplacian. The matrix \( M_u^{-1} \) is a dense matrix [68, p. 353, 99, p. 551] and thus impractical to work with. It follows from [255, 68, p. 296] that a mass matrix can be approximated by its diagonal coefficients. Therefore, replacing \( M_u \) with \( \hat{M}_u = \text{diag}(M_u) \) leads to a sparse approximation

\[ S_{LSC} = (B \hat{M}_u^{-1} B^T)(B \hat{M}_u^{-1} F \hat{M}_u^{-1} B^T)^{-1}(B \hat{M}_u^{-1} B^T), \] (3.4.25)

where the matrix \( B \hat{M}_u^{-1} B^T \) is referred to as the sparse pressure Poisson operator. Application of the LSC preconditioner matrix to a vector requires the action of the Schur complement approximation inverse

\[ S_{LSC}^{-1} = (B \hat{M}_u^{-1} B^T)^{-1}(B \hat{M}_u^{-1} F \hat{M}_u^{-1} B^T)(B \hat{M}_u^{-1} B^T)^{-1} \] (3.4.26)

to a vector at each Krylov iteration. The construction of \( S_{LSC}^{-1} \) involves the assembly of the diagonal entries of the velocity mass matrix \( \hat{M}_u \) (which may already be available if a transient problem is solved), sparse matrix-matrix multiplications (SpGEMM) to construct the sparse pressure Poisson operators \( B \hat{M}_u^{-1} B^T \), and the setup of a solver for the sparse pressure Poisson operator (e.g. AMG coarsening). The application of \( S_{LSC}^{-1} \) involves two discrete Poisson solves, SpMV multiplication with the matrices.
Another strategy for approximating the Schur complement follows from the continuous version of the commutator \((3.4.17)\) rather than the discrete version \((3.4.18)\). This leads to the pressure convection-diffusion (PCD) preconditioner. The PCD preconditioner employs the approximation

\[
S_{PCD} = A_p F_p^{-1} M_p,
\]

where

\[
A_p = [a_{p,ij}], \quad a_{p,kl} = \int_{\Omega} \nabla \varphi_j \cdot \nabla \varphi_i \, d\Omega, \quad i, j = 1, \ldots, N_p, \quad (3.4.27)
\]

is the pressure Laplacian matrix, \(F_p\) is the convection-diffusion operator on the pressure space defined by \((3.4.21)\) and \(M_p\) is the pressure mass matrix defined by \((3.4.20)\). Derivation of the PCD preconditioner can be found in [68, pp. 347 – 353]. The setup of this preconditioning strategy requires the assembly of specially constructed auxiliary operators \(F_p, A_p\) and \(M_p\). The construction of \(F_p\) as described in [68] does not include the boundary conditions in the convection-diffusion operator i.e. natural (Neumann) boundary conditions are assumed on the entire boundary (where the solution is unique only up to a constant), therefore \(A_p\) and \(F_p\) as defined by \((3.4.27)\) and \((3.4.21)\) are a natural choice for an enclosed flow problem (such as the driven cavity problem [68, p. 220]). However the PCD strategy, does not work effectively for problems with inflow and outflow boundaries (such as the backwards facing step problem [68, p. 219]). For such problems Elman and Tuminaro modified \(F_p\) to enforce a Robin boundary condition at the inflow boundary, a similar modification for the LSC preconditioner was made for problems with Dirichlet inflow boundary conditions, details can be found in [70].

This reveals the main advantage of the LSC preconditioning strategy: it does not require the construction of additional operators, possibly with complex boundary conditions. Once the discrete linear system is assembled, all the ingredients required to construct \(S_{LSC}\), which is defined in terms of the matrices \(F\) and \(B\), can be obtained from the Jacobian matrix defined by \((2.3.38)\). In this sense, the LSC preconditioner fully automated and requires no additional user intervention. In this thesis, we refer to this feature as a ‘black-box’ feature. The assembly of the PCD preconditioner requires additional information which cannot be obtained from the discrete linear system.

The application of the PCD preconditioner requires the action of \(S_{PCD}^{-1} = M_p^{-1} F_p A_p^{-1}\).
which involves the action of a pressure Poisson solve, a mass matrix solve and a SpMV multiplication with a specially constructed matrix $F_p$. The LSC preconditioner requires two Poisson solves, thus the PCD preconditioner requires fewer computations per application. However, the LSC preconditioner requires approximately half the number of iteration counts to converge [68, p. 356], but for triangular meshes, dependence on the discretisation parameter, $h$, is observed [180, p. 85].

Strategies for automating the construction of $F_p$ discussed in [72] led to a new interpretation of BFBt methods originally proposed in [69]. Further improvements led to a scaled BFBt method and methods based on sparse approximate commutators (SPAC). The scaled BFBt strategy employs the same approximation for the Schur complement $S$ as the LSC preconditioner and the SPAC based methods employ SPAI to eliminate one of the two Poisson solves involved in the application of the BFBt approximation (at a cost of an additional SPAI setup). It was found that the scaled BFBt preconditioner generally outperforms the PCD preconditioner in terms of iteration counts at the cost of an additional Poisson solve at each Newton iteration. The iteration counts of the SPAC-M preconditioner (the SPAC preconditioner based on the scaled BFBt approximation) are similar to the PCD preconditioner with nearly identical execution time per iteration.

Application of preconditioners often require the computation of inverses. In the LSC method, we require the inverse of the sparse pressure Poisson operators $B\tilde{M}_u^{-1}B^T$ and the momentum block $F$. When these inverses are computed exactly by a (sparse) direct solver, such as SuperLU, we refer to these as the exact versions of the preconditioner. When these inverses are computed approximately (for example, by AMG), we refer the resulting preconditioning method as inexact versions.

### 3.4.3.2 Augmented Lagrangian Preconditioners

In this section we describe the augmented Lagrangian (AL) preconditioning strategy, we follow the discussion in [29, 30]. Instead of solving the discrete linear system (3.4.10), we consider the equivalent AL formulation $\tilde{F}\tilde{x} = \tilde{b}^*$ given by

$$
\begin{bmatrix}
F + \gamma B^T W^{-1} B & B \\
B & O
\end{bmatrix}
\begin{bmatrix}
\bar{u} \\
\bar{p}
\end{bmatrix} = 
\begin{bmatrix}
\bar{f}^* \\
\bar{g}
\end{bmatrix},
$$

(3.4.28)

where $\bar{f}^* := \tilde{f} + \gamma B^T W^{-1} \bar{g}$, $W$ is SPD, and $\gamma > 0$ is a parameter. The matrix $W$ is taken as the pressure mass matrix $M_p$, however, to maintain the sparsity pattern
in \( F + \gamma B^T W^{-1} B \), the diagonal approximation \( \hat{M}_p = \text{diag}(M_p) \) is used instead. By setting \( F_\gamma := F + \gamma B^T W^{-1} B \), then the (ideal) AL preconditioner is given by

\[
P_{AL} = \begin{bmatrix} F_\gamma & B^T \\ B & -\frac{1}{\gamma} W \end{bmatrix}.
\]  

(3.4.29)

For the ideal AL preconditioning strategy (3.4.29), the choice \( \gamma = 1 \) yields a nearly-optimal preconditioner in the sense that the iteration counts are low and independent of the problem parameters such as the discretisation parameter \( h \) and Reynolds number Re. It is demonstrated numerically to be more robust than the PCD and LSC preconditioners, particularly for high Reynolds numbers (Re > 200). However, the implementation described in [29] requires a specially developed geometric multigrid method based on [222] for the (1,1) block, whilst the PCD and LSC preconditioners can use various off-the-shelf solvers such as AMG.

The modified AL preconditioner proposed in [30] is obtained by using the block upper triangular approximation to \( \tilde{F}_\gamma = \tilde{F} + \gamma B^T W^{-1} B \), where \( \tilde{F} \) is the block diagonal approximation to \( F \). The main advantage of the modified AL preconditioner over the original AL preconditioner is that it can be implemented using readily available AMG solvers. The disadvantage is that the parameter \( \gamma \) becomes problem-dependent, in particular, in 2D it is sensitive to the discretisation parameter \( h \). The iteration counts were also significantly higher than in the ideal case [29] with a mild dependence on the Reynolds number.

### 3.5 Parallel Sparse Solvers

Due to the rapidly growing capabilities of parallel computing infrastructures, developing efficient and robust software becomes an indispensable phase in modern scientific computing. In this section we discuss some aspects of parallelising linear solvers. We start with parallelisation issues for sparse direct solvers. Then we look at parallelisation of iterative linear solvers, this would include AMG and parallel preconditioners.
3.5. **PARALLEL SPARSE SOLVERS**

3.5.1 Parallel Sparse Direct Solvers

Sparse direct solvers implement a version of Gaussian elimination algorithm to the linear system $A\vec{x} = \vec{b}$. Specifically, $A$ is decomposed into lower and upper triangular factors $L$ and $U$, respectively. This process is also referred to as sparse matrix factorisation. Sparse direct factorisation typically consist of four distinct phases, an ordering phase to either reduce fill-in or to transform $A$ into a specific structure (such as minimising the bandwidth), an analyse phase where a symbolic factorisation performed to determine the sparsity pattern of the matrix factors and set up of various auxiliary data structures such as elimination trees [57, p. 38], a factorisation phase which involves numerical computation of the factors $L$ and $U$, and a numerical solve phase where backwards and forwards substitution are performed. The actual subdivision into phases depend on the implementation, for example, SuperLU performs symbolic and numerical factorisation as a single phase [11, p. 329]. Typically, the largest proportion of the execution time is spent in the numerical factorisation phase.

Each of these phases have to be parallelised. Furthermore, the implementation may dependent on the properties of the coefficient matrix (for example, if $A$ is SPD or indefinite) and the target parallel architecture environment (shared or distributed memory). Here we provide a few examples of parallel implementations.

The two most frequently used algorithms for matrix re-ordering are the minimum degree and the nested dissection. The nested dissection lends itself better to parallelisation due to its divide and conquer approach, however, in practice the most efficient implementation is a hybrid of these methods [130]. Most sparse parallel direct solvers either implement their own variant of the nested dissection or provide interfaces to standard libraries such as PT-SCOTCH [48] or ParMETIS [147]. In this phase, the order of the elimination process is determined.

Parallelism in the analyse phase can be exploited when generating the elimination trees or other graphs depending on the implementation. These range from complex techniques utilising forests (a disjoint union of trees) or simple divide and conquer algorithms, see [169] and references therein.

Numerical stability is ensured with a pivoting strategy, which should also preserve the sparsity of the matrix factors (reduce fill-in). For symmetric matrices, the choice of pivots can be decoupled from the numerical factorisation phase, this is known as static pivoting. Static pivoting is harder to achieve for nonsymmetric matrices, therefore the pivoting strategy and numerical factorisation are usually coupled (referred...
to as dynamic or numerical pivoting), see for example SuperLU_MTS\textsuperscript{39}. In the supernodal approach, this technique is not suitable for distributed memory systems as it involves fine grained communication and memory access to manage the work order. SuperLU\textsuperscript{DIST}\textsuperscript{163} is a distributed memory implementation of a sparse direct solver for nonsymmetric matrices based on the supernodal approach. In this case, static pivoting is employed followed by iterative refinement, details of the algorithm can be found in\textsuperscript{162}. The algorithm groups the coefficient matrix columns together to form supernodes which are mapped to processors in a 2D block cyclic fashion. The numerical factorisation phase proceeds with parallel right-looking LU-factorisation using dense level 3 BLAS kernels.

The multifrontal algorithm utilises an assembly tree (a variation of the elimination tree) to identify independent tasks (referred to as tree parallelism). The algorithm proceeds by assembling $A$ into small dense blocks (referred to as frontal matrices), interleaved with the partial numerical factorisation of these matrices with parallel dense factorisation kernels (referred to as node parallelism). MUMPS\textsuperscript{182,4} is an implementation of the multifrontal method for distributed memory systems which can be applied to both symmetric and general nonsymmetric matrices. It employs threshold pivoting during the numerical factorisation phase facilitated with dynamic distributed task scheduling.

The main difficulty in parallelising the solve phase lies in the sequential nature of the forward/backwards substitution, due to the data interdependencies. However, the sparsity of the factors means that each unknown depends on only a subset of other variables. This interdependency can be modelled by a directed graph. The number of independent paths in this graph defines the subset of equations that can be operated on concurrently. This is known as level-scheduling\textsuperscript{215, pp. 387–390,5}, or more generally, as topological sorting\textsuperscript{105, pp. 612–615}.

### 3.5.2 Parallel Iterative Solvers

In this section, we describe parallelisation issues and strategies for iterative solvers. In contrast to sparse direct solvers, iterative solvers are relatively simple to implement. We focus on Krylov methods. Krylov methods presented in Section 3.3.6 require the parallelisation of three operations: vector updates, IPs and SpMV multiplication. The GMRES algorithm presented in Algorithm 3.2 requires the orthogonalisation process to be parallelised as well. We discuss each of these operations in turn. In the case of a
preconditioned iterative solver, in addition to the previously mentioned operations, we must also parallelise the preconditioning operator, this is discussed in Section 3.5.3. This is especially relevant because application of sophisticated preconditioners frequently take more than eighty percent of the execution time of an iterative solver.

In the sequel, we assume a distributed memory environment with MPI-based parallelisation. The matrices are stored in compressed row storage (CRS) format and distributed by rows, where each processing element holds a certain number of matrix rows. The vectors are stored in a dense format and are distributed coherently to match the distribution of the matrices. These are the settings employed in oomph-lib and used in this project.

Vector updates are trivial to parallelise. Each processor updates the values already resident on the processor and no communication is required. The parallel inner product of two distributed vectors \( \bar{x} \) and \( \bar{y} \) (assumed to have the same distribution) proceeds by having each processor calculate the local inner product (LIP) of the sub-vectors resident on the processor, followed by communication of the LIPs to all other processors where the summation of the LIPs is performed. This can be implemented efficiently with the MPI_Allreduce function \[212, pp. 76–78\]. We note that this process requires global communication which typically scales poorly.

For the SpMV multiplication \( \bar{y} = A \bar{x} \), let \( N_p \) be the number of local rows of \( A \) held by processor \( p \). Denote by \( A_p = [a_{p,ij}] \) and \( \bar{y}_p = [y_{p,i}] \), \( i = 0, \ldots, N_p, j = 0, \ldots, N \), the rectangular sub-matrix of \( A \) and the sub-vector of \( \bar{y} \) local to processor \( p \). Then processor \( p \) can calculate the component \( y_{p,i} \) as the inner product of the \( i \)th row of \( A_p \) and some of the entries in the global vector \( \bar{x} \). Depending on the bandwidth of \( A \), the computation of \( y_{p,i} \) may depend on some elements of \( \bar{x} \) which do not reside on \( p \). We denote these elements \( \bar{x}_{p,ext} \) (referred to as the external variables), likewise, \( \bar{x}_{p,loc} \) denotes the local elements of \( \bar{x} \) required in the SpMV computation (referred to as local variables). Those processors which \( p \) must communicate with to obtain \( \bar{x}_{p,ext} \) are termed as neighbouring processors. Denote by \( A_{p,loc} \) the restriction of \( A_p \) to \( \bar{x}_{p,loc} \) and \( A_{p,ext} \) the restriction of \( A_p \) to \( \bar{x}_{p,ext} \). Then processor \( p \) can calculate the local matrix vector product by computing \( \bar{y}_p := A_{p,loc} \bar{x}_{p,loc} \), then obtain \( \bar{x}_{p,ext} \) from its neighbouring processors, followed by \( \bar{y}_p := \bar{y}_p + A_{p,ext} \bar{x}_{p,ext} \). The data in \( A_{p,loc} \) and \( A_{p,ext} \) are held by the same processor. The decomposition of \( A_p \) into \( A_{p,loc} \) and \( A_{p,ext} \) identifies those operations which require communication and those which do not. This enables parallel
efficiency to be gained by overlapping computation with communication \(^6\). To gain parallel efficiency in most computationally intensive part of the Krylov iteration, the SpMV multiplication kernels typically pre-process the matrix data to generate the computational graphs and determine communication patterns. Communication takes place only between neighbouring processors, thus an efficient implementation of SpMV multiplication should have better parallel scaling than inner products (which requires global communication).

In addition to parallelising particular operations, another approach to increase parallelism is to re-arrange the iterative algorithms to decrease the number of synchronisation points or/and overlap communication with useful computation. For example, a variant of the CG algorithm presented in [18, p. 68] overlaps all inner product operations with computation.

In the GMRES Algorithm 3.2 method, the orthogonalisation of the vector \( \bar{w}_j = A\bar{v}_j \) against all previous \( \bar{v} \) is achieved with a modified Gram-Schmidt algorithm [215, p. 12]. This consists of a sequence of inner products and SAXPY operations \(^7\), which are easily parallelised as discussed above. More parallelism can be exploited with block Krylov methods as discussed in [18, p. 71], where the operator \( A \) (the coefficient matrix) acts on blocks of vectors instead of a single vector. Walker [253] proposed an alternative orthogonalisation process for the GMRES algorithm based on the Householder process [215, p. 173] which can be easily parallelised. The s-step Lanczos algorithm proposed by Kim and Chronopoulos [154] improves data locality by increasing the number of inner products per iteration of the orthogonalisation process, thus reducing the total number of global communication required to generate the Krylov basis. The idea of communication-reducing algorithm is further developed in recent works by Demmel et al. [61], where an implementation for general sparse matrices is provided. A more detailed discussion of the parallelisation techniques presented in this section can be found in [214, 44] and references therein.

3.5.3 Parallel Preconditioners

In this section, we discuss the parallelisation aspect of preconditioners. The application of a preconditioner \( P \) requires the (exact or approximate) solution of a linear system \( P\tilde{z} = \tilde{y} \) at each Krylov iteration. The preconditioning operations involved

\(^6\)This is achieved with asynchronous non-blocking communication in MPI, which allow processes to continue with computation while communication with other processes are pending [212, pp. 296-300].

\(^7\)Operations of the form \( \tilde{y} := a\tilde{x} + \tilde{y} \).
3.5. PARALLEL SPARSE SOLVERS

in this process can be grouped into two phases, the preconditioner setup phase and the preconditioner solve phase. It is essential that the preconditioner solve phase is implemented efficiently and have good parallel scaling since it accounts for a large proportion of the execution time (> 80%) of each Krylov iteration.

Parallel Block Preconditioners

First we discuss the parallelisation aspects of $\mathcal{P}_{BLT}$ defined by (3.4.9) followed by the parallelisation of the (problem-specific) LSC preconditioner discussed in Section 3.4.3 and introduce the block Jacobi preconditioner.

The application of $\mathcal{P}_{BLT}$ requires the solution of the system

$$
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
\tilde{z}_1 \\
\tilde{z}_2
\end{bmatrix}
= 
\begin{bmatrix}
\tilde{y}_1 \\
\tilde{y}_2
\end{bmatrix}.
$$

This can be performed in two steps, solve the first block equation for $\tilde{z}_1$ via $\tilde{z}_1 = A_{11}^{-1}\tilde{y}_1$, then solve the second block equation for $\tilde{z}_2$ via $\tilde{z}_2 = A_{22}^{-1}(\tilde{y}_2 - A_{21}\tilde{z}_1)$. For now, assume that all the sub-block matrices and vectors required for preconditioning are available.

We assume that the block matrices are uniformly distributed by rows among the processors with matching distribution for the sub-vectors.

The preconditioner setup phase consists of setting up a parallel solver for the diagonal blocks (for example, SuperLU_DIST or AMG) and SpMV multiplication setup for $A_{21}$. The preconditioner solve phase consists of the solver application, vector updates and application of the SpMV multiplication. For an efficient implementation of $\mathcal{P}_{BLT}$, we should choose an efficient parallel solver for the diagonal blocks. Parallelisation of other operations have been discussed in previous sections.

If we now consider the LSC preconditioner, the only additional operation compared to $\mathcal{P}_{BLT}$ is a SpGEMM performed during the construction of the sparse pressure Poisson operators. By contrast to previously discussed routines, it is difficult to implement an efficient and scalable parallel SpGEMM procedure. Classification of matrix-matrix multiplication algorithms are based on the distribution of the matrix data. For example, algorithms based on row or column partitioning of matrix data are referred to as one-dimensional algorithms, whereas two-dimensional algorithms are based on the decomposition of both columns and rows, where each processor holds a block of the matrix data. Buluç demonstrated that 1D algorithms for parallel SpGEMM routines typically have poor parallel scaling [36], a parallel 2D SpGEMM algorithm is presented
in [37] which offers significantly better parallel scaling. For both serial and parallel SpGEMM, we use the Trilinos EpetraExt library [244] (version 11.8.1), since this is oomph-lib’s default parallel SpGEMM kernel. This uses a 1D decomposition for sparse matrices. To compute $C = AB$, the algorithm proceeds by first generating a graph representation of $A$ and $B$, referred to as ‘views’. Any external (off-processor) data required for the multiplication is imported during this phase. Then the sparsity pattern of $C$ is determined via symbolic multiplication of $A$ and $B$. Finally, multiplication takes place by iterating through the rows of $A$ and the columns of $B$.

Thus, if we have efficient and scalable implementations of parallel solvers, SpMV multiplication and SpGEMM, all that is required for efficient parallel implementation of the LSC preconditioner is the blocking and extraction of the matrices and vectors. This is discussed in Section 5.1.

The block (overlapping) Jacobi preconditioner [215, p. 394] is derived from the set-decomposition of the set of matrix variables $S = \{1, 2, \ldots, N\}$ for the coefficient matrix $A \in \mathbb{R}^{N \times N}$. Suppose $S_i \subseteq S$ with $S = \cup_{i=1}^{p} S_i$, where $p$ is the number of blocks. Then the block Jacobi preconditioner $\tilde{M} = [\tilde{m}_{ij}]$ has entries:

$$
\tilde{m}_{ij} = \begin{cases} 
\tilde{a}_{ij} & \text{if } i \text{ and } j \text{ are in the same index set} \\
0 & \text{otherwise}.
\end{cases}
$$

Corrections are added to the overlapping sections during the iterative process, see [215, p. 395]. This construction is attractive to parallel computing environments for large enough blocks since they can be solved concurrently with a small ratio of communication (for the overlapping sections) to computational work. The set-partition of the variables can be obtained from the distribution of the coefficient matrix, in the case of vector problems, we can group together the variables which represent physical quantities (such as velocity and pressure in the case of the Navier-Stokes equations), or in the case of a PDE discretised on a physical domain, each set can corresponds to a partitioning of the domain. The last choice leads to domain decomposition methods.

### 3.5.4 Domain Decomposition Preconditioners

In domain decomposition (DD) methods, we re-formulate the boundary value problem (BVP) defined on the (discretised) domain $\Omega_h$ (with mesh size $h$) onto a set of smaller (coarse) sub-domains $\mathcal{Q}_H = \{\Omega_i\}, \ i = 1, \ldots, p$, where $p$ is the number of coarse
domains, and $H$ is the coarse mesh size. Each coarse sub-domain is typically mapped to a distinct processor. Partitioning of $\Omega_h$ into the set $Q_H$ can be element-based (where no elements split between two sub-domains), edge-based (where no edges are split between two sub-domains), or vertex-based (where both elements and edges are allowed to be split between two sub-domains). The coarse sub-domains can be overlapping or disjoint. The sub-problems are then solved in a divide and conquer like fashion with the values at the interfaces handled appropriately depending on the technique.

For overlapping methods, we discuss the Alternating Schwarz Procedures because they some the most well known techniques in overlapping DD methods. In this approach, each sub-domain $\Omega_i$ is extended by a distance $\delta$ to form $\Omega'_i$, where $\delta$ is the amount of overlap. In the Multiplicative Schwarz procedure, the most recently updated values of the approximate solution is used during a sweep through solving the BVP on each coarse sub-domain $\Omega'_i$. This is analogous to the block Gauss-Seidel iteration. The Additive Schwarz Procedure is analogous to the overlapping block Jacobi iteration, where only the solution computed at previous sweeps are used during the computation on a sub-domain. These methods can be interpreted as multi-level preconditioners. In this context, the associated preconditioner is defined by its application to a vector that is generated by the solves for each coarse sub-domain. Let $A_i \in \mathbb{R}^{N_i \times N_i}$ be the operators associated with each extended sub-domain $\Omega'_i$ and $A \in \mathbb{R}^{N \times N}$ the operator associated with $\Omega_h$. Denote by $R_i \in \mathbb{R}^{N_i \times N}$ the restriction operator which transforms a vector from $\Omega_h$ to $\Omega_i$, the transpose $R_i^T \in \mathbb{R}^{N \times N_i}$ is the prolongation operator. Then the restriction of $A$ to $\Omega_i$ is defined by $A_i = R_i A R_i^T$. For the Additive Schwarz procedure, the new approximation is

$$\bar{x}_{new} = \bar{x} + \sum_{i=1}^{p} R_i^T A_i^{-1} R_i (\bar{b} - A \bar{x}) = \left( I - \sum_{i=1}^{p} R_i^T A_i^{-1} R_i A \right) \bar{x} + \sum_{i=1}^{p} R_i^T A_i^{-1} R_i \bar{b} \quad (3.5.1)$$

Equation (3.5.1) is of the form $\bar{x}_{new} = G \bar{x} + \bar{f}$, where $G = I - \sum_{i=1}^{p} R_i^T A_i^{-1} R_i A$. Utilising the relation $G = I - \tilde{M}^{-1} A$, we have the inverse of the preconditioner matrix $\tilde{M}^{-1} = \sum_{i=1}^{p} R_i^T A_i^{-1} R_i A^{-1}$. On application to a vector, $\tilde{z} = \tilde{M}^{-1} \bar{y}$, we compute $\tilde{z}_i = R_i^T A_i^{-1} R_i A^{-1} \bar{y}$, $(i = 1, \ldots, p)$, followed by summation of the vectors $\tilde{z}_i$ in each domain to obtain a global vector $\tilde{z}$. Each $\tilde{z}_i$ can be computed in parallel. For the Multiplicative Schwarz preconditioner, see [213, p. 491].
3.5.5 Parallel Coarsening Algorithms for AMG

In this section we review two strategies for the parallel selection of coarse grid points. The discussion presented follows from Section 3.3.2.

**Cleary–Luby–Jones–Plassman (CLJP):** The CLJP coarsening strategy [131, p. 7] is based on parallel graph partitioning algorithms which makes it inherently parallel. The auxiliary influence matrix $S$ is defined by

$$
S_{ij} = \begin{cases} 
1 & \text{if } j \in S_i \\
0 & \text{if } j \not\in S_i 
\end{cases}
$$

(3.5.2)

where $S_i$ is the set of point on which $i$ depends. This implies that the $i$th row of $S$ corresponds to the set of dependence $S_i$, and the $i$th column of $S$ gives the set of influences $S^T_i$. Now we assign a measure $w(i) = |S^T_i| + \sigma$ to each point in $S$, where $|S^T_i|$ is the number of points $i$ influences and $\sigma \in (0, 1)$ is a random number (introduced to break ties for points with the same number of influences). Next a directed graph $D$ is constructed which consists of the points $i$ satisfying $w(i) > w(j) \forall j \in S_i \cap S^T_j$. Edges are removed from $D$ with the aim of preserving the quality of the coarse grid whilst controlling its size. The removal is based on two heuristics: points that influences $C-$points because they are less valuable than potential $C-$ points themselves; on a given $C-$point, if both $k$ and $j$ depends on $i$, and $j$ influences $k$, then $j$ is less valuable as a $C-$point since it $k$ can be interpolated from $i$.

**Falgout coarsening:** Falgout is a parallel coarsening strategy [131, pp. 164–165]. In parallel computations, the matrix (assumed to be row partitioned) will be split among many processors. Then Ruge-Stüben coarsening applies on the interior of each sub domain. CLJP is applied along the boundaries. We use Falgout coarsening for all our parallel computations.

3.6 Summary

In this section we reviewed existing linear solvers for the solution of a sparse linear system. The concept of preconditioners was introduced with the aim to accelerate linear solvers. Parallelisation aspects of all solvers presented was also discussed.
Chapter 4

Preconditioning of the Constrained Navier-Stokes Equations

In this chapter we present a generalised version of the no-slip and no-penetration boundary conditions. We show that imposing such conditions with Lagrange multipliers leads to the original Navier-Stokes system [2.3.38] augmented by the Lagrange multiplier constraints. We propose a preconditioner for the augmented system and give eigenvalue bounds of the preconditioned system.

4.1 No-slip and No-penetration Boundary Conditions

Full generality of the finite element method (FEM) makes it applicable to real-life problems posed over arbitrary shaped domains. This requires that the boundary conditions (BCs) are posed correctly on the boundaries of such domains. If the boundaries are aligned with the Cartesian directions, then imposing either parallel, axially-traction free outflow or no-penetration conditions is straightforward, see Section 2.2.2.

Consider the Navier-Stokes equations in two dimensions where a part of the boundary $\partial \Omega_P \subset \partial \Omega$ is aligned with the Cartesian axes, as depicted in Figure 4.1 (left). We want to enforce ‘parallel outflow’ through the boundary $\partial \Omega_P$. The alignment of $\partial \Omega_P$ with $x_2$ implies that parallel outflow through the boundary $\partial \Omega_P$ is equivalent to the $x_2$-component of the velocity enforced to be equal to 0 on $\partial \Omega_P$, while $u_1$ remains unknown (apply the do-nothing condition for $u_1$). Recall from Section 2.2.2, this implies that the applied traction force $T_1$ in the normal direction (in this case) is zero.
This construction satisfies the incompressibility constraint (2.1.5) at the outlet. For the no-penetration condition, as depicted in Figure 4.1 (right), there is no velocity normal to the boundary \( \partial \Omega_T \). The alignment of the boundary \( \partial \Omega_T \) with \( x_2 \) implies that we can set \( u_1 = 0 \) and apply the do-nothing condition for \( u_2 \).

We want to impose BCs depicted in Figure 4.1 on arbitrarily shaped boundaries, which are not aligned with coordinate axes, or even curvilinear boundaries (see for example Figures 4.2 and 4.3), in these situations contribution from both velocity components are required to correctly prescribe the velocity profile at the boundary. Simply setting one of the velocity components to zero as described above no longer works. In such situations it may be tempting to have \( T_i = 0 \) in all directions (i.e. do-nothing for all velocity components). However, this construction may result in non-physical behaviour in the computed flow at the outlet. For example, by examining the variational form of this construction, it can be shown that the sign and magnitude of the velocity at the outflow boundary cannot be controlled or bounded, which leads to inwards bending velocity and ill-posed formulations, see [86, pp. 292–293] and [34].

We can impose the parallel outflow condition by setting no tangential velocity component along \( \partial \Omega_P \), that is \( \mathbf{u} \cdot \hat{\mathbf{t}} = 0 \), where \( \hat{\mathbf{t}} \) is the unit tangent vector along \( \partial \Omega_P \). Similarly for the no-penetration condition, where the velocity along \( \partial \Omega_T \) has no normal component to \( \partial \Omega_T \), we can impose \( \mathbf{u} \cdot \hat{\mathbf{n}} = 0 \), where \( \hat{\mathbf{n}} \) is the outwards unit normal to the boundary \( \partial \Omega_T \). Imposing such flow profiles at the boundaries can be viewed as ‘penalising’ the flow along the boundary under consideration. This can be done by introducing additional constraints to the momentum equation. The new BCs
4.1. NO-SLIP AND NO-PENETRATION BOUNDARY CONDITIONS

Figure 4.2: The domains from Figure 4.1 are rotated by 30 degrees. The blue arrows indicate the Cartesian direction of the components of velocity $u_1$ and $u_2$. Setting $u_2 = 0$ for parallel outflow (left) and $u_1 = 0$ for no-penetration (right) no longer produces the desired flow profile at the boundaries.

Figure 4.3: Imposing parallel outflow through a curved boundary $\partial \Omega_P$. The blue arrows indicate the Cartesian direction of the components of velocity $u_1$ and $u_2$. At each point of the boundary the velocity $\mathbf{u}$ needs to be orthogonal to the tangential vector (red arrows).

can be stated as

\begin{align}
\text{‘parallel outflow’:} & \quad \mathbf{u} \cdot \hat{\mathbf{t}} = 0, \quad \text{on } \partial \Omega_P, \quad (4.1.1) \\
\text{‘no-penetration’:} & \quad \mathbf{u} \cdot \hat{\mathbf{n}} = 0, \quad \text{on } \partial \Omega_T, \quad (4.1.2)
\end{align}

where $\hat{\mathbf{t}}_s (s = 1, 2)$ in two- or three-dimensional domain) are the unit tangent vector(s) on $\partial \Omega_P$ and $\hat{\mathbf{n}}$ is the outwards unit normal vector to $\partial \Omega_T$. Figure 4.4 illustrates the effects of these constraints. The constraint $\mathbf{u} \cdot \hat{\mathbf{t}}_s = 0$ on $\partial \Omega_P$ enforces a flow which is orthogonal to $\hat{\mathbf{t}}$ through the boundary $\partial \Omega_P$, this is equivalent to the ‘parallel, axially-traction-free’ outflow BC as depicted in Figure 4.4 (left). Figure 4.4 (right) shows that $\mathbf{u} \cdot \hat{\mathbf{n}} = 0$ enforces ‘no-penetration’ along $\partial \Omega_T$ since the fluid flows orthogonally to $\hat{\mathbf{n}}$ i.e. parallel to the boundary $\partial \Omega_T$. These constraints are imposed weakly with Lagrange multipliers. Weakly imposing of Dirichlet BCs was first studied by Babuška.
CHAPTER 4. CONSTRAINED NAVIER-STOKES EQUATIONS

Figure 4.4: Penalising the flow along $\partial\Omega_P$ (left), enforcing ‘parallel outflow’; and $\partial\Omega_T$, enforcing ‘no-penetration’ (right). The red arrows indicate the direction of the unit tangent vector $\hat{t}$ or the outwards unit normal vector $\hat{n}$ to the constrained boundary, the blue arrow indicates the direction of the constrained flow $u$, the green colour indicates the dot product of the vectors in red and blue, and the angle $\alpha$ represents the angle formed between the constraint vectors ($\hat{t}$ or $\hat{n}$) and the Cartesian axes.

in [9]. Stability issues are treated in [167, 166, 157], in particular it is proven that weakly imposing Dirichlet BCs via Lagrange multipliers (with appropriate subspaces which we define in the next section) leads to the mixed formulation that satisfy the LBB condition. Imposing Dirichlet BCs weakly is sometimes preferable, for example, spurious oscillations may occur when the outflow BCs are strongly enforced (see Çağlar [38], and Bazilevs and Hughes [22]).

Remark 4.1.1. In viscous fluids, the no-slip condition,

$$u = 0 \quad \text{on} \; \partial\Omega,$$

as described in Section 2.2.2 accurately describes the motion of the fluid at the boundaries. It states that there is no relative motion between the boundary and the layer of molecules immediately next to the boundary (often referred to as the boundary layer), see [58] for a comprehensive discussion on the justifications for the no-slip condition. The no-slip condition (4.1.3) can be decomposed into two conditions:

$$u \cdot \hat{n} = 0 \quad \text{on} \; \partial\Omega \quad \text{and}$$

$$u \cdot \hat{t} = 0 \quad \text{on} \; \partial\Omega.$$

In literature, condition (4.1.4) requiring no normal motion relative to the boundary $\partial\Omega$ is referred to as the no-penetration condition (also called the impermeability condition). The condition (4.1.5), requiring no tangential motion relative to the boundary $\partial\Omega$ is also referred to as the no-slip condition. The no-slip condition (4.1.3) does not
always apply, for example, in free boundary problems \cite{25} and problems involving biological surfaces and interfaces \cite{23}. In \cite{185}, Navier proposed that the fluid velocity at a boundary should be proportional to the tangential component of stress, this can be expressed as:

\begin{align}
\mathbf{u} \cdot \hat{n} &= 0 \quad \text{on } \partial \Omega \quad \text{and} \\
\hat{n}^T \mathbf{\sigma} \hat{t} &= 0 \quad \text{on } \partial \Omega,
\end{align}

where \( \mathbf{\sigma} \) is the stress tensor defined by \cite{2.3.3}. The constraints \eqref{4.1.6} and \eqref{4.1.7} are referred to as the slip condition, see \cite{246, 247} for more details. Condition \eqref{4.1.7} naturally follows from the weak formulation of the momentum equations since the boundary term \eqref{2.3.8} implies \eqref{4.1.7}. In this work we refer to \( \mathbf{u} \cdot \hat{t} = 0 \) as the parallel outflow condition instead of ‘no-slip’, since this should not be confused with the no-slip condition \eqref{4.1.3}. We refer to \( \mathbf{u} \cdot \hat{n} = 0 \) as the no-penetration condition.

## 4.2 Weak Formulation

From the previous discussion we impose the flow at a boundary \( \partial \Omega_{\lambda} \) by insisting that

\begin{equation}
\mathbf{u} \cdot \hat{c} = 0,
\end{equation}

where \( \hat{c} \) is either the unit tangent vector \( \hat{t}_s, s = 1, 2 \) or the outwards unit normal vector \( \hat{n} \) at each point of the boundary \( \partial \Omega_{\lambda} \) for parallel outflow or no-penetration, respectively. We weakly enforce the flow constraint by augmenting the Navier-Stokes momentum residual equation \cite{2.3.5} with a Lagrange multiplier term

\begin{equation}
\rho_i^{(a)} = \int_{\Omega} \left[ \text{Re} \left( \frac{\partial \mathbf{u}_i}{\partial t} + u_j \frac{\partial \mathbf{u}_i}{\partial x_j} \right) \psi_i + \mathbf{\sigma}_{ij} \frac{\partial \psi_i}{\partial x_j} \right] d\Omega \\
- \int_{\partial \Omega} \sigma_{ij} \hat{n}_j \psi_i dS + \delta \Pi_{\lambda} = 0,
\end{equation}

where

\begin{equation}
\Pi_{\lambda} = \int_{\partial \Omega_{\lambda}} \lambda u_i \hat{c}_i dS = 0,
\end{equation}

and \( \lambda \) is the Lagrange multiplier. The first variation of \eqref{4.2.3} is

\begin{equation}
\delta \int_{\partial \Omega_{\lambda}} \lambda u_i \hat{c}_i dS = \int_{\partial \Omega_{\lambda}} \hat{c}_i \lambda \delta u_i + \hat{c}_i u_i \delta \lambda dS = 0, \quad \forall \delta u_i, \delta \lambda.
\end{equation}
Since $\delta u_i$ is a variation of $u_i$, we can replace it by the test function $\psi_i$ (as defined in Chapter 2) in (4.2.4) and noting that (due to (4.2.1), $u_i \hat{c}_i = 0$), we can rewrite (4.2.4) as

$$\int_{\partial \Omega_\lambda} \lambda \psi_i \hat{c}_i \, dS = 0.$$  \hspace{1cm} (4.2.5)

Substituting (4.2.5) into (4.2.2) yields the residual form of the constrained momentum equations

$$r_i^{(u)} = \int_{\Omega} \left[ \text{Re} \left( \text{St} \frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} \right) \psi_i + \sigma_{ij} \frac{\partial \psi_i}{\partial x_j} \right] \, d\Omega - \int_{\partial \Omega_N} \sigma_{ij} \hat{n}_j \psi_i \, dS + \int_{\partial \Omega_\lambda} \lambda \psi_i \hat{c}_i \, dS = 0. \hspace{1cm} (4.2.6)$$

Equation (4.2.6) reveals the physical meaning of the Lagrange multipliers. When $\hat{c} = \hat{t}$, then $\lambda$ is the negative of the tangential component of the stress tensor,

$$\lambda = -\hat{n}^T \sigma \hat{t}, \hspace{1cm} (4.2.7)$$

and when $\hat{c} = \hat{n}$, then $\lambda$ is the negative of the normal component of the stress tensor,

$$\lambda = -\hat{n}^T \sigma \hat{n}. \hspace{1cm} (4.2.8)$$

This is because we can recover the boundary term $-\int_{\partial \Omega_N} \hat{n}^T \sigma \psi \, dS$ in (4.2.6) by substituting either (4.2.7) or (4.2.8) into (4.2.5).

The weak formulation of the BC (4.2.1) in residual form is simply

$$r^{(c)} = \int_{\partial \Omega_\lambda} u_i \hat{c}_i \psi^c \, dS = 0, \hspace{1cm} (4.2.9)$$

where $\psi^c$ is a suitable test function which we shall formally define later. Since no derivatives of $\lambda$ and $\psi^c$ appear in (4.2.6) and (4.2.9), they just need to be square integrable, however, they must be restricted to the boundary $\partial \Omega_\lambda$. Let $Z = H^{-1/2}(\partial \Omega_\lambda)$ consist of the traces of all functions in $L^2(\Omega)$ on $\partial \Omega_\lambda$. In this setting, the whole boundary $\partial \Omega$ is now subdivided into three non-overlapping parts, $\partial \Omega_D$, $\partial \Omega_N$ and $\partial \Omega_\lambda$, which may be different for different velocity components: $\partial \Omega = \partial \Omega_D \cup \partial \Omega_N \cup \partial \Omega_\lambda$ and $\partial \Omega_D \cap \partial \Omega_N \cap \partial \Omega_\lambda = \emptyset$. The weak formulation of the problem now can be stated as:
Find \( \mathbf{u} = \[u_i\]_{i=1}^d \in \mathbf{H}_E^1 \), \( p \in L^2 \) and \( \lambda \in Z \) such that

\[
r^{(u)}_i = \int_\Omega \left[ \text{Re} \left( \frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} \right) \psi_i + \sigma_{ij} \frac{\partial \psi_i}{\partial x_j} \right] d\Omega + \int_{\partial \Omega} \lambda \phi_i \psi_i dS = 0, \quad \text{for } i = 1, 2[3],
\]

(4.2.10)

\[
r^{(p)} = - \int_\Omega \frac{\partial u_j}{\partial x_j} \varphi d\Omega = 0,
\]

(4.2.11)

and

\[
r^{(c)} = \int_{\partial \Omega} u_i \phi_i \psi_c dS = 0, \quad \text{for } i = 1, 2[3],
\]

(4.2.12)

satisfying BCs

\[
u_i = w_i, \quad \text{on } \partial \Omega_D, \quad \text{for } i = 1, 2[3],
\]

(4.2.13)

\[
T_i = -p \hat{n}_i + \left( \frac{\partial u_i}{\partial x_i} + \frac{\partial u_j}{\partial x_i} \right) \hat{n}_j = 0, \quad \text{on } \partial \Omega_N, \quad \text{for } i = 1, 2[3],
\]

(4.2.14)

with initial conditions

\[
\frac{\partial u_i}{\partial x_i} = 0, \quad \text{in } \Omega, \quad \text{for } i = 1, 2[3],
\]

(4.2.15)

and

\[
\hat{n}_i \cdot u_i^0 = \hat{n}_i \cdot w_i^0 \quad \text{on } \partial \Omega_D, \quad \text{for } i = 1, 2[3],
\]

(4.2.16)

\( \forall \psi = \[\psi_i\]_{i=1}^d \in \mathbf{H}_0^1 \), \( \forall \varphi \in L^2 \) and \( \forall \psi_c \in Z \).

### 4.3 The FEM Discretisation

Discretisation of the momentum equation is discussed in Section 2.4 here we consider only the Lagrange multiplier term in (4.2.10) and the weakly imposed BC (4.2.12). Restrict \( Z \) to the finite-dimensional subspace \( Z^h \subset Z \). Let \( \lambda_h, \psi_h^c \in Z^h \) be the discrete approximation for \( \lambda \) and \( \psi^c \), respectively. Then the discrete residuals of the momentum equations (4.2.10) and the BC (4.2.12) are

\[
r^{(u)}_h = \int_{\partial \Omega} \lambda_h \psi_h \psi_i dS = 0, \quad \text{for } i = 1, 2[3], \quad \text{and}
\]

(4.3.1)

\[
r^{(c)} = \int_{\partial \Omega} u_h \psi_h c dS = 0, \quad \text{for } i = 1, 2[3],
\]

(4.3.2)

\footnotetext{1}{For notational simplicity, we have omitted all terms from (4.2.10) except for the Lagrange multiplier boundary term. The discrete weak formulation for the omitted terms can be found in Chapter 2}
where \( \psi_{h,i} \in X_h^0 \) and \( u_{h,i} \in X_h^h \), as described in Section 2.3.1. We introduce a set of global basis functions \( Z_h = \text{span}\{\varphi_k\}_{k=1}^{N_t} \), where \( N_t \) is the size of the discrete basis set of \( Z_h \). Define the discrete approximations as:

\[
\lambda_h = \sum_{k=1}^{N_t} \Lambda^\lambda_k \varphi_k, \quad \quad \quad u_{h,i} = \sum_{k=1}^{N_t} U_{ik} \psi_{ik}, \quad (4.3.3)
\]

\[
\psi_{h,i} = \sum_{k=1}^{N_t} \Phi_{ik} \psi_{ik} \quad \text{and} \quad \psi^c_h = \sum_{k=1}^{N_t} \Lambda^c_k \varphi_k. \quad (4.3.4)
\]

where \( N_{t,i} \) is the number of velocity unknowns in the \( i \)th direction affected by the Lagrange multipliers. Substituting (4.3.3) and (4.3.4) into (4.3.1) and (4.3.2) yields

\[
r_{il}^{(u_h)} = \int_{\partial \Omega} \hat{c}_i \sum_{k=1}^{N_t} \Lambda^\lambda_k \varphi_k \psi_{il} \, dS = 0 \quad \text{and} \quad (4.3.5)
\]

\[
r_{il}^{(c_h)} = \int_{\partial \Omega} \hat{c}_i \sum_{k=1}^{N_{t,i}} U_{ik} \psi_{ik} \varphi_k \, dS = 0,
\]

respectively. Then the Jacobian entries are

\[
\frac{\partial r_{il}^{(u_h)}}{\partial \Lambda^\lambda_k} = \int_{\partial \Omega} \hat{c}_i \varphi_k \psi_{il} \, dS \quad \text{and} \quad (4.3.6)
\]

\[
\frac{\partial r_{il}^{(c_h)}}{\partial U_{ik}} = \int_{\partial \Omega} \hat{c}_i \psi_{ik} \varphi_k \, dS. \quad (4.3.7)
\]

Due to [246, 247, 167], the above discrete weak formulation will have a unique solution if there is a constant \( \beta > 0 \) independent of the discretisation parameter \( h \), such that

\[
\inf_{p_h \neq 0 \in M_h} \sup_{\lambda_h \neq 0 \in Z_h} \frac{(p_h, \nabla \cdot \psi_h) - (\lambda_h, \psi_h \cdot \hat{c})_{\partial \Omega}}{\|\psi\|_{1,\Omega} \left\{ \|p_h\|_{0,\Omega}^2 + \|\lambda_h\|_{Z_h}^2 \right\}^{1/2}} \geq \beta. \quad (4.3.8)
\]

Now we define a discrete basis set for \( Z_h \). We discretise the Lagrange multiplier terms on a mesh on the boundary \( \partial \Omega_\lambda \) which is matching the bulk mesh. A matching mesh is one where adjacent elements share sides, nodes and degrees of freedom, see Figure 4.5. The Lagrange multipliers unknowns \( \Lambda^\lambda_k \) are assigned to each vertex of the boundary \( \partial \Omega_\lambda \), using equal order approximation as for the velocity space with one dimension lower. For example, let a domain \( \Omega \in \mathbb{R}^2 \) be discretised by \( Q_2 \) elements and associate the elements to the boundary \( \partial \Omega_\lambda \) with BCs imposed via Lagrange multipliers as described above. Then the mesh used for the Lagrange multiplier terms consist
of one-dimensional $Q_2$ elements, as shown in Figure 4.6. The setup in Figure 4.6 assumes the bulk mesh with the standard elements (in our case two-dimensional $Q_2$ elements) used to discretise the interior of $\Omega$, while the boundary mesh consists of one-dimensional $Q_2$ elements, referred to as face elements. The face element framework is implemented using omph-lib’s FaceElements \cite{115, 111}, with one face element attached to each bulk element adjacent to the boundary $\partial \Omega_\lambda$. This gives the Lagrange multipliers access to the unknowns within the adjacent bulk element, enabling the constraint in (4.3.1) to be imposed to the momentum equation. In this construction, the discrete Lagrange multiplier subspace $Z^h$ is approximated by the same quadratic basis functions we use for the velocity approximation adjacent to the boundary $\partial \Omega_\lambda$ (the red nodes of the bulk elements). In three dimensions the constraints would be imposed by two-dimensional FaceElements and the Lagrange multiplier subspace is approximated by bi-quadratic basis functions. It is shown in \cite{24} that this construction satisfies the inf-sup condition (4.3.8) and thus is a stable approximation.

By the above construction, (4.3.6) and (4.3.7) are velocity mass matrices on the boundary $\partial \Omega_\lambda$ multiplied by a component of the unit tangent vector or the outwards unit normal vector. The solution of the resulting nonlinear algebraic equations by
Newton’s method leads to a sequence of linear systems

$$\mathcal{F}^{[k]} \delta \vec{X}^{[k]} = -\vec{r}^{[k]}, \quad (4.3.9)$$

where $\mathcal{F}^{[k]}$ is the Jacobian arising from the $k$th Newton step which has the saddle point structure

$$\mathcal{F}^{[k]} = \begin{bmatrix} \mathcal{F}^{[k]}_{ns} & L^T \\ L & \end{bmatrix}, \quad (4.3.10)$$

where $\mathcal{F}^{[k]}_{ns}$ corresponds to the Navier-Stokes block in (2.3.38) and the matrix $L$ is the discrete Lagrange multiplier constraint. The first block row in equation (4.3.10) correspond to equations (4.2.10) and (4.2.11) and the second block row corresponds to equation (4.2.12). The degrees of freedom (DOF) in this problem corresponds to nodal values of the Cartesian velocity components and pressure, and the Lagrange multiplier field $\lambda$. For simplicity, we limit ourselves to two spatial dimensions. We group the velocity DOFs into $x_1$ and $x_2$ components and further subdivide the each of the velocity components into DOFs which lie on the boundary subject to Lagrange multiplier constraints (i.e. $\partial \Omega^\lambda$) and those that are not. Then equation (4.3.9) can be written in the following block form

$$\begin{bmatrix} F_{11} & F_{11} & F_{12} & F_{12} \\ F_{11} & F_{11} & F_{12} & F_{12} \\ F_{21} & F_{21} & F_{22} & F_{22} \\ F_{21} & F_{21} & F_{22} & F_{22} \end{bmatrix} \begin{bmatrix} (B_1)^T \\ (B_1)^T \\ (B_2)^T \\ (B_2)^T \end{bmatrix} \begin{bmatrix} \delta \vec{U}_1 \\ \delta \vec{U}_1 \\ \delta \vec{U}_2 \\ \delta \vec{U}_2 \end{bmatrix} = \begin{bmatrix} \vec{r}_1 \\ \vec{r}_1 \\ \vec{r}_2 \\ \vec{r}_2 \end{bmatrix} \quad \uparrow N_{u_1} \quad \uparrow N_{u_1}, \quad (4.3.11)$$

$$\begin{bmatrix} B_1 & B_1 & B_2 & B_2 \\ M_1 & M_2 \end{bmatrix} \begin{bmatrix} \delta \vec{P} \\ \delta \vec{\Lambda} \end{bmatrix} = \begin{bmatrix} \vec{r}_p \\ \vec{r}_\ell \end{bmatrix} \quad \uparrow N_p \quad \uparrow N_\ell,$$

where the Newton corrections $[\delta U_i, \delta U_i]^T \ (i = 1, 2)$ correspond to the unknown velocity components in the $i$th Cartesian direction. The hat on the subscript represents the nodes affected by the Lagrange multiplier constraint. The notation on the right border represent the dimension of sub-blocks. The block vector $\delta \vec{\Lambda}$ represent the Newton corrections for the Lagrange multiplier constraint. The blocks $M_i \ (i = 1, 2)$ corresponds to the Jacobian entries defined by (4.3.7) and the coefficients are calculated by (for
example $M_1$:

\[ m_{1,ij} = \int_{\partial \Omega_\lambda^T} \hat{c}_i \psi_i \psi_j dS \quad i, j = 1, \ldots, N_\ell. \]  

(4.3.12)

Consider the case of a straight boundary rotated by an angle $\alpha$. From Figure 4.4 (left) we see that the $x_1$ and $x_2$ components of the unit tangent vector are $-\sin(\alpha)$ and $\cos(\alpha)$, respectively, and from Figure 4.4 (right) we see that the $x_1$ and $x_2$ components of the outwards unit normal vector are $\cos(\alpha)$ and $\sin(\alpha)$, respectively. For parallel outflow we have $\hat{c} = \hat{t}$, and from (4.3.7) we see that $L$ has the following block structure

\[ L = \begin{bmatrix} O & -\sin(\alpha)M & O & \cos(\alpha)M & O \end{bmatrix}, \]  

(4.3.13)

where $M$ is the mass matrix defined by (4.3.7). Likewise, for no-penetration we have $\hat{c} = \hat{n}$ and according to (4.3.7) $L$ as the following block structure

\[ L = \begin{bmatrix} O & \cos(\alpha)M & O & \sin(\alpha)M & O \end{bmatrix}. \]  

(4.3.14)

For no-penetration along a non-straight boundary as depicted in Figure 4.7 (left), the angle of the unit normal vector with the $x_1$-axis at each Lagrange multiplier vertex is not constant along the entire boundary $\partial \Omega_T$, as seen in Figure 4.7 (right). We can no longer factor out the normal vector components since the elements of the matrices $M_1$ and $M_2$ are obtained by multiplying the $i$th row of a mass matrix $M \in \mathbb{R}^{N_\ell \times N_\ell}$ by $\cos(\alpha_j)$ and $\sin(\alpha_j)$, respectively, where $\alpha_j$ is the angle between $\hat{n}_j$ (the outward unit normal at the point $j$ on $\partial \Omega_T$) and the $x_1$-axis.

Consider the case where there are more than one constraints, for example, enforcing
parallel outflow in a three-dimensional domain, as shown in Figure 4.8. Here we have two tangent vectors which results in two Lagrange multipliers and the matrix $L$ will have the following block structure

$$L = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix} = \begin{bmatrix} O & M_{1,1} & O & M_{1,2} & O & M_{1,3} & O \\ O & M_{2,1} & O & M_{2,2} & O & M_{2,3} & O \end{bmatrix}.$$  \hspace{1cm} (4.3.15)

This can be generalised to any number of Lagrange multiplier constraints. Another example would be to impose parallel outflow on the boundary $\partial \Omega_P$ and no-penetration on the boundary $\partial \Omega_T$ as depicted in Figure 4.9. We have considered such configuration in \cite{258}. In the next section we develop a novel preconditioning algorithm for the linear system (4.3.11).
4.4 Augmentation Block Preconditioner

Consider the linear system $A\overline{x} = \overline{b}$ with following saddle point structure:

$$
\begin{bmatrix}
F & B^T \\
B & O
\end{bmatrix}
\begin{bmatrix}
\overline{x}_1 \\
\overline{x}_2
\end{bmatrix}
= 
\begin{bmatrix}
\overline{b}_1 \\
\overline{b}_2
\end{bmatrix},
$$

(4.4.1)

where $F \in \mathbb{R}^{N_1 \times N_1}$, $B \in \mathbb{R}^{N_2 \times N_1}$, with $N_2 \leq N_1$, and $\overline{x} = [\overline{x}_1, \overline{x}_2]^T$, $\overline{b} = [\overline{b}_1, \overline{b}_2]^T \in \mathbb{R}^{N_1+N_2}$ are the unknown solution and the (known) right hand side, respectively. There is a wealth of information on solving such systems, for example, see [27] and reference therein. The saddle point system (4.4.1) can be preconditio ned by so-called augmentation preconditioners [97, 98, 202] of the form

$$
P_{aug} = 
\begin{bmatrix}
\tilde{F} & O \\
O & W
\end{bmatrix}
= 
\begin{bmatrix}
F + B^T W^{-1} B & O \\
O & W
\end{bmatrix},
$$

(4.4.2)

where $W \in \mathbb{R}^{N_2 \times N_2}$ is a symmetric positive definite weight matrix. This preconditioning technique was already studied in the context of micromagnetics problems [97]. The main idea behind choosing this approach for preconditio ning the system (4.3.11) lies in the special structure of the nonzero constraint blocks in $L$, namely $M_1$ and $M_2$. These closely resemble the mass matrices which have predictable spectral properties and can be effectively approximated by its diagonal. This specific structure of the constraint blocks in this case leads to a specific choice of the matrix $W$. Successful application of the augmentation preconditioning to elasticity problems subjected to constraint imposed by Lagrange multipliers described in [181] is the main motivation for this work. The block matrix $W$ should be chosen such that:

- The sparsity of $\tilde{F} = F + B^T W^{-1} B$ is similar to $F$ (or at least, should not be significantly more dense than $F$), so the storage requirement is still of order $O(N_1)$.

- The preconditioner $P_{aug}$ is an efficient preconditioner in the sense that the eigenvalues of the linear system $P_{aug}^{-1} A\overline{x} = P_{aug}^{-1} \overline{b}$ exhibit a tight clustering.

- $P_{aug}$ is computationally inexpensive to form and apply.
Choosing $W$ is a non-trivial task. First we need to know some information about the linear system to be solved. Re-writing the Jacobian in (4.3.11) as

$$
F = \begin{bmatrix}
F_{ns} & L^T \\
L & 0
\end{bmatrix}
$$

where $F_{ns} \in \mathbb{R}^{N_{ns} \times N_{ns}}$ corresponds to the original Navier-Stokes system described by (2.3.38) and $L \in \mathbb{R}^{N_\ell \times N_{ns}}$ is the Lagrange multiplier constraint block as described in (4.3.15). We note the following:

1. The linear system obtained from the discretisation of weakly imposed BCs by Lagrange multipliers give rise to an indefinite system with a saddle point structure.

2. Since the method of Lagrange multipliers were used to weakly impose the BCs, the nonzero blocks $M_i (i = 1, 2, 3)$ of $L$ are mass-like matrices defined by (4.3.7). Each $M_i$ is full rank by the construction of our Lagrange multiplier space, thus $L$ has full row rank.

3. Construction of $L^T W^{-1} L$ requires the inverse $W^{-1}$. This needs to be sparse to ensure the sparsity of $F_{ns} + L^T W^{-1} L$. For this we may be able to exploit the properties of the mass-like matrices in the constraint block $L$.

Greif and Shoutau [97], presented general analysis for preconditioners for saddle point problems (4.4.1) where the (1, 1)-block is highly singular. It is shown that $F$ as constructed in (4.4.2) gives rise to a preconditioned matrix $P_{nm}^{-1} A$ that has eigenvalues $\lambda = 1$ of algebraic multiplicity at least $N_1 - N_2$. Muddle in [181] used the construction $W = \frac{1}{\sigma} M^2$, where $\sigma \neq 0 \in \mathbb{R}$ and $M$ is a mass matrix, for the preconditioning of saddle point problems with symmetric positive definite (1, 1)-block arising from pseudo-elastic problems. An optimal preconditioner was developed. We extend this work to the case where the (1, 1)-block is nonsymmetric and indefinite with a saddle point structure by introducing a general method for the construction of $W$. 
4.5 Preconditioning for the Augmented Navier-Stokes System

For the saddle point problem (4.3.11) we seek an augmentation preconditioner of the form

\[ P = \begin{bmatrix} F_{ns} + L^T W^{-1} L & O \\ O & W \end{bmatrix}. \]  

The matrix \( W \in \mathbb{R}^{N \times N} \) is an arbitrary matrix and should be constructed to preserve the sparsity of \( F_{ns} \) and lead to \( P \) being an effective preconditioner for \( F \). We generalise the approach of Muddle from [181, pp. 120–125] by choosing

\[ W = \frac{1}{\sigma} LL^T, \]  

where \( \sigma > 0 \in \mathbb{R} \) is an arbitrary positive parameter. We determine the optimal value for \( \sigma \) in Section 4.5 with an aim of good clustering of the eigenvalues of a preconditioned matrix. For simplicity, we restrict ourselves to two spatial dimensions, generalisation to three dimensions is straightforward. First we examine the structure of the matrix \( L^T W^{-1} L = L^T (\frac{1}{\sigma} LL^T)^{-1} L \) for the case of a single Lagrange multiplier constraint, then we look at a generalised version of the preconditioner, with more than one Lagrange multiplier constraint, for example Figures 4.8 and 4.9.

Recall from (4.3.11) that \( L = \begin{bmatrix} O & M_1 & O & M_2 & O \end{bmatrix} \), which yields

\[ W = \frac{1}{\sigma} LL^T = \frac{1}{\sigma} ((M_1)^2 + (M_2)^2), \]  

and

\[ L^T W^{-1} L = \begin{bmatrix} O & O & O & O & O \\ O & \sigma M_1((M_1)^2 + (M_2)^2)^{-1} M_1 & \sigma M_1((M_1)^2 + (M_2)^2)^{-1} M_2 & O & O \\ O & O & O & O & O \\ O & \sigma M_2((M_1)^2 + (M_2)^2)^{-1} M_1 & \sigma M_2((M_1)^2 + (M_2)^2)^{-1} M_2 & O & O \\ O & O & O & O & O \end{bmatrix}. \]

The exact computation of \( W^{-1} \) from (4.5.3) would result a dense matrix. However, from [255], it follows that the mass matrix \( M \in \mathbb{R}^{N \times N} \) is spectrally equivalent to its diagonal approximation \( \tilde{M} = \text{diag}(M) \in \mathbb{R}^{N \times N} \) in the sense that the eigenvalues \( \lambda_M \)
of the generalised eigenvalue problem [68, p. 82]

\[ M\bar{x} = \lambda_M \hat{M}\bar{x} \]

satisfy the inequalities

\[ k \leq \frac{\bar{x}^T M\bar{x}}{\bar{x}^T \hat{M}\bar{x}} \leq K, \quad \forall \bar{x} \neq \bar{0} \in \mathbb{R}^N, \quad (4.5.4) \]

where \( k \) and \( K \) are constants independent of the grid size. For any Hermitian matrix \( A \in \mathbb{R}^{N \times N} \) we have \( \lambda_i(A^2) = \lambda_i^2(A) \) for \( i = 1, \ldots, N \). This enables us to bound the eigenvalues of the problem

\[ M^2\bar{x} = \lambda \hat{M}^2\bar{x}. \quad (4.5.5) \]

These satisfy

\[ \lambda((\hat{M}^2)^{-1}M^2) = \lambda^2(\hat{M}^{-1}M), \]

and since \((4.5.4)\) holds for \( \lambda(\hat{M}^{-1}M) \), we have

\[ k^2 \leq \frac{\bar{x}^T M^2\bar{x}}{\bar{x}^T \hat{M}^2\bar{x}} \leq K^2, \quad \forall \bar{x} \neq \bar{0} \in \mathbb{R}^N. \quad (4.5.6) \]

If \( \hat{W} = \text{diag}(W) \), then from \((4.5.6)\) it is sufficient to use \( \hat{W} = \frac{1}{\sigma} \text{diag}(M_1^2 + M_2^2) \) and inverting \( \hat{W} \) is straightforward.

In special cases of straight constrained boundaries, where the angle between the normal / tangential vector and the Cartesian directions remain constant and \( L \) defined by \((4.3.13)\) and \((4.3.14)\), \( W \) can be further simplified as follows

\[ W = \frac{1}{\sigma^2} LL^T = \frac{1}{\sigma}(M_1^2 + M_2^2) = \frac{1}{\sigma}((\sin^2(\alpha) + \cos^2(\alpha))M^2) = \frac{1}{\sigma}M^2. \quad (4.5.7) \]

Therefore, for the case of parallel outflow \((4.3.13)\)

\[
L^T W^{-1} L = \begin{bmatrix}
0 & O & O & O & O & O \\
O & \sigma \sin^2(\alpha)I & O & -\sigma \sin(\alpha) \cos(\alpha)I & O & O \\
O & O & O & O & O & O \\
O & -\sigma \cos(\alpha) \sin(\alpha)I & O & \sigma \cos^2(\alpha)I & O & O \\
O & O & O & O & O & O \\
\end{bmatrix}, \quad (4.5.8)
\]
and for the no-penetration \((4.3.14)\) we have

\[
L^TW^{-1}L = \begin{bmatrix}
O & O & O & O & O \\
O & \sigma \cos^2(\alpha)I & O & \sigma \cos(\alpha)\sin(\alpha)I & O \\
O & O & O & O & O \\
O & \sigma \sin(\alpha)\cos(\alpha)I & O & \sigma \sin^2(\alpha)I & O \\
O & O & O & O & O
\end{bmatrix}.
\] (4.5.9)

Thus the formation of the modified blocks \(\hat{F}_{ij}\) \((\hat{i}, \hat{j} = 1, 2)\) requires the addition of a constant to the diagonal entries of the corresponding sub-blocks \(F_{ij}\). This clearly demonstrates the fact that the sparsity of the matrix \(F_{ns} + L^TW^{-1}L\) is preserved. The assembly of this matrix can be done with optimal computational cost. Furthermore, as only a few nonzero elements of the original coefficient matrix \(F_{ns}\) are modified, the conjecture is that the existing preconditioners for the Navier-Stokes problem can be applied effectively in a black-box manner (or with minimal modifications) to precondition the subsystem with the coefficient matrix \(\tilde{F}_{ns} = F_{ns} + L^TW^{-1}L\). In the case of constrained elasticity problem \([176]\), the preconditioner used for the standard problem lead to optimal performance for the augmentation preconditioner \([181\text{, pp. 124–127}]\).

The previous discussion can be easily extended to the case of \(k\) constraints imposed by \(k\) simultaneous Lagrange multipliers, giving the following block row partitioning of the constraint block \(L = [L_1 \, L_2 \ldots \, L_k]^T\), then we have

\[
\hat{W} = \frac{1}{\sigma} \begin{bmatrix}
\text{diag}(L_1L_1^T) \\
\text{diag}(L_2L_2^T) \\
\cdots \\
\text{diag}(L_kL_k^T)
\end{bmatrix} = \begin{bmatrix}
\hat{W}_1 \\
\hat{W}_2 \\
\cdots \\
\hat{W}_k
\end{bmatrix},
\]

and

\[
L^T\hat{W}^{-1}L = L_1^T\hat{W}_1^{-1}L_1 + L_2^T\hat{W}_2^{-1}L_2 + \ldots + L_k^T\hat{W}_k^{-1}L_k.
\] (4.5.10)

We observe from \((4.5.10)\) that when there are more than one imposed Lagrange multiplier constraints, the perturbation matrix is simply a sum of the component (individual) perturbation matrices.
4.6 Spectral Properties of the Preconditioned Operator

In this section we provide an analysis that demonstrates the exact version of the augmentation preconditioner $P$ given by (4.5.1) is spectrally close to a simplified Jacobian matrix. As discussed in Chapter 3, this would guarantee rapid convergence of a Krylov solver, with the iteration counts remaining asymptotically constant under mesh refinement. Specifically, we aim to characterise the eigenvalues $\nu$ of the generalised eigenvalue problem

$$\mathcal{J} \bar{v} = \nu P \bar{v}. \quad (4.6.1)$$

We perform the analysis for the case of a simplified form of the Navier-Stokes equations, arising from Oseen’s problem [68, p. 329]. This assumes a linearised version of the convection term in the momentum equation, achieved either through Picard’s linearisation [68, p. 326], or through a Simo–Armero type scheme in the case of a time dependent problem [226]. In addition, we consider only the simple form of the viscous term. This configuration leads to a block diagonal form of the momentum block, where each of the diagonal blocks are the discrete scalar convection-diffusion operator (the blocks $G_{ij}$ and $W_{ij}$ in (2.3.38) due to the stress divergence form and Newton’s linearisation, respectively, are equal to zero). Thus, the generalised eigenvalue problem (4.6.1) can be written in the block form as:

$$
\begin{bmatrix}
F_{ns} & L^T \\
L & O
\end{bmatrix}
\begin{bmatrix}
\bar{v}_{ns} \\
\bar{v}_t
\end{bmatrix} = \nu
\begin{bmatrix}
\bar{F}_{ns} & O \\
O & W
\end{bmatrix}
\begin{bmatrix}
\bar{v}_{ns} \\
\bar{v}_t
\end{bmatrix}, \quad (4.6.2)
$$

where $F_{ns} \in \mathbb{R}^{(N_u+N_p) \times (N_u+N_p)}$ is the linearised Navier-Stokes block

$$F_{ns} = \begin{bmatrix}
F & B^T \\
B & O
\end{bmatrix}, \quad (4.6.3)$$

with

$$F = \begin{bmatrix}
F_{11} & O \\
O & F_{22}
\end{bmatrix}. \quad (4.6.4)$$

In two spatial dimensions ($F_{ii} = A_{ii} + N_{ii}$, $i = 1, 2$). The matrix $\bar{F}_{ns} = F_{ns} + L^T W^{-1} L$ is the augmented Navier-Stokes matrix defined in (4.5.1). We first rewrite the augmented
4.6. SPECTRAL PROPERTIES OF THE PRECONDITIONED OPERATOR

Navier-Stokes matrix from (4.6.2) as

$$
\mathcal{F} = \begin{bmatrix} F & B^T & \hat{L}^T \\ B & O & O \\ \hat{L} & O & O \end{bmatrix} \sim \begin{bmatrix} F & \hat{L}^T & B^T \\ \hat{L} & O & O \\ B & O & O \end{bmatrix} = \begin{bmatrix} \tilde{F} & \tilde{B}^T \\ \tilde{B} & O \end{bmatrix} \uparrow N_u + N_\ell. \quad (4.6.5)
$$

Following this $2 \times 2$ blocking the preconditioner matrix $\mathcal{P}$ can be written as

$$
\mathcal{P} = \begin{bmatrix} F + \hat{L}^T W^{-1} \hat{L} & O & B^T \\ O & W & O \\ B & O & O \end{bmatrix} = \begin{bmatrix} \tilde{F}_a & \tilde{B}^T \\ \tilde{B} & O \end{bmatrix}. \quad (4.6.6)
$$

Based on matrix blocking introduced in (4.6.5)–(4.6.6) the analysis of the eigenvalues of (4.6.2) can be performed in two stages. Firstly, we establish spectral equivalence between $(1, 1)$ blocks of $\mathcal{F}$ and $\mathcal{P}$. In this context we extend the general spectral analysis for augmentation preconditioners introduced in [97] and applied to the case of symmetric saddle point systems arising from the linear elasticity problem with weakly imposed boundary conditions in [181]. The extension includes a non-symmetric $(1, 1)$ block, leading to complex eigenvalues in general. The analysis also bears some resemblance to that used for the augmented Navier-Stokes preconditioner from [30]. Then, in the second part we apply general theory for constraint preconditioners introduced in [10, 43, 63, 150] to bound the eigenvalues in a special case of spectrally equivalent $(1, 1)$ blocks.

4.6.1 Spectral Analysis of the Constrained Momentum Block Sub-problem

Based on the notation from (4.6.5)–(4.6.6) we consider the eigenvalue problem

$$
\tilde{F} \tilde{v} = \mu \tilde{F}_a \tilde{v}, \quad (4.6.7)
$$

or, in an extended $2 \times 2$ block form:

$$
\begin{bmatrix} F & \hat{L}^T \\ \hat{L} & O \end{bmatrix} \begin{bmatrix} \tilde{v}_u \\ \tilde{v}_\ell \end{bmatrix} = \mu \begin{bmatrix} F + \hat{L}^T W^{-1} \hat{L} & O \\ O & W \end{bmatrix} \begin{bmatrix} \tilde{v}_u \\ \tilde{v}_\ell \end{bmatrix}. \quad (4.6.8)
$$
CHAPTER 4. CONSTRAINED NAVIER-STOKES EQUATIONS

We can rewrite (4.6.8) as a system of block equations:

\[
F \bar{v}_u + \hat{L}^T \bar{v}_u = \mu (F + \hat{L}^T W^{-1} \hat{L}) \bar{v}_u \\
\hat{L} \bar{v}_u = \mu W \bar{v}_\ell
\]  

(4.6.9)  

(4.6.10)

Expressing \( \bar{v}_\ell \) from (4.6.10) as
\[
\bar{v}_\ell = \frac{1}{\mu} W^{-1} \hat{L} \bar{v}_u,
\]
(4.6.11)

and substituting into (4.6.9) yields
\[
F \bar{v}_u + \hat{L}^T \left( \frac{1}{\mu} W^{-1} \hat{L} \right) \bar{v}_u = \mu (F + \hat{L}^T W^{-1} \hat{L}) \bar{v}_u,
\]
(4.6.12)

i.e.
\[
(\mu^2 - \mu) F \bar{v}_u + (\mu^2 - 1) \hat{L}^T W^{-1} \hat{L} \bar{v}_u = 0.
\]
(4.6.13)

From (4.6.13) it follows that \( \mu = 1 \) is the eigenvalue of (4.6.8) of multiplicity \( N_u \) and (4.6.13) is satisfied for any \( \bar{v}_u \neq 0 \). Thus, \( \left[ \bar{v}_u, W^{-1} \hat{L} \bar{v}_u \right]^T \) is the eigenvector associated with the unit eigenvalue.

Now consider the case \( \mu \neq 1 \). Then dividing (4.6.13) by \( \mu^2 - 1 \) gives
\[
-\frac{\mu}{\mu + 1} F \bar{v}_u = \hat{L}^T W^{-1} \hat{L} \bar{v}_u.
\]
(4.6.14)

Denote \( \lambda = -\frac{\mu}{\mu + 1} \), i.e. \( \mu = -\frac{\lambda}{\lambda + 1} \). Then (4.6.14) becomes
\[
\lambda F \bar{v}_u = \hat{L}^T W^{-1} \hat{L} \bar{v}_u.
\]
(4.6.15)

The matrix \( \hat{L}^T W^{-1} \hat{L} \in \mathbb{R}^{N_u \times N_u} \) is symmetric and indefinite with rank \( N_\ell < N_u \). Thus the eigenvalue problem (4.6.15) has \( N_\ell \) nonzero eigenvalues which are the non-unit eigenvalues of (4.6.8).

The choice of the matrix \( W \) in the augmentation preconditioner is arbitrary, however it should lead to good clustering of the non-unit eigenvalues. We choose \( W = \frac{1}{\sigma} \hat{L} \hat{L}^T \). For the case of straight constrained boundaries this choice is the same as in [181], i.e. \( W = \frac{1}{\sigma} M^2 \), where \( M \) is the mass matrix defined on the discrete constraint space (see (4.3.12)). In our subsequent analysis we assume that \( \sigma > 0 \). The eigenvalue
where \( \tilde{\lambda} = \frac{\sigma}{\lambda} \) is, in general, a complex eigenvalue. Equation (4.6.17) can be rewritten as a 2 \times 2 eigenvalue problem

\[
\begin{bmatrix}
F_{11} & O & F_{1\tilde{i}} & O \\
O & F_{22} & O & F_{2\tilde{i}} \\
F_{1\tilde{i}} & O & F_{1\tilde{i}} & O \\
O & F_{2\tilde{i}} & O & F_{2\tilde{i}}
\end{bmatrix}
\begin{bmatrix}
\tilde{v}_1 \\
\tilde{v}_2 \\
\tilde{v}_1 \\
\tilde{v}_2
\end{bmatrix}
= \hat{\lambda}
\begin{bmatrix}
O & O & O & O \\
O & O & O & O \\
O & O & \alpha_{11}I & \alpha_{12}I \\
O & O & \alpha_{21}I & \alpha_{22}I
\end{bmatrix}
\begin{bmatrix}
\tilde{v}_1 \\
\tilde{v}_2 \\
\tilde{v}_1 \\
\tilde{v}_2
\end{bmatrix},
\] (4.6.17)

The \( N_{\ell} \) non-unit eigenvalues \( \mu \) of the problem (4.6.15) are connected to the \( N_{\ell} \) eigenvalues \( \hat{\lambda} \) of (4.6.18) through the relation

\[
\hat{\lambda} = \frac{\sigma}{\lambda} = -\frac{\mu}{\mu + 1} \implies \mu = -\frac{\sigma}{\sigma + \lambda}.
\] (4.6.19)

Notice that the same relation holds in the case of an SPD constrained problem (cf. equation (27) in [181]). Expressing \( \tilde{v}_b \) from the first block equation in (4.6.18) and substituting into the second one gives

\[
S_{\ell\ell} \tilde{v}_\ell = \hat{\lambda} \mathbf{I} \tilde{v}_\ell,
\] (4.6.20)

where \( S_{\ell\ell} = F_{\ell\ell} - F_{\ell b} F_{b b}^{-1} F_{b \ell} \) is the Schur complement. Equation (4.6.20) can be written.
in a $2 \times 2$ block form as:

$$S_{i1} \tilde{v}_1 = \tilde{\lambda} \alpha_{11} I \tilde{v}_1 + \alpha_{12} I \tilde{v}_2,$$

(4.6.21)

$$S_{22} \tilde{v}_2 = \tilde{\lambda} \alpha_{21} I \tilde{v}_1 + \alpha_{22} I \tilde{v}_2.$$

(4.6.22)

Recall that in the case of parallel outflow boundary tilted by an angle $\alpha$ with respect to the $x_1$-axis we have $\alpha_{11} = \cos^2(\alpha)$, $\alpha_{22} = \sin^2(\alpha)$, and $\alpha_{12} = \alpha_{21} = \sin(\alpha) \cos(\alpha)$, cf. (4.5.8). The block LU decomposition of the matrix $I$ from (4.6.18) gives

$$I = \begin{bmatrix} \alpha_{11} I & \alpha_{12} I \\ \alpha_{21} I & \alpha_{22} I \end{bmatrix},$$

(4.6.23)

with $S_I = (\alpha_{22} - \frac{\alpha_{21}}{\alpha_{11}}) I = O$ reveals that further $N_{\ell}$ eigenvalues $\tilde{\lambda}$ of (4.6.17) are equal to 0. The remaining $N_{\ell}$ eigenvalues can be characterised by expressing $\tilde{v}_2$ from (4.6.21) and substituting into (4.6.22) to give

$$S_{22} S_{i1} \tilde{v}_1 = \tilde{\lambda} \alpha_{11} S_{22} + \alpha_{22} S_{i1} \tilde{v}_1.$$

(4.6.24)

There are two special cases of (4.6.24): for $\alpha = k\pi$, and for $\alpha = \left(k + \frac{1}{2}\right)\pi$, $k \in \mathbb{Z}$. In the former case (4.6.24) reduces to

$$S_{i1} \tilde{v}_1 = \tilde{\lambda} \tilde{v}_1,$$

(4.6.25)

and in the latter

$$S_{22} \tilde{v}_1 = \tilde{\lambda} \tilde{v}_1.$$

(4.6.26)

The form of the eigenvalue problems (4.6.25) and (4.6.26) is the same as in the case of the augmentation preconditioner for the constrained elasticity problem (see equation (26) in [181]).

Next we demonstrate that the eigenvalues $\tilde{\lambda}$ of (4.6.24) (and consequently of (4.6.25) and (4.6.26)) have positive real parts. This property is required to derive the bounds for the eigenvalues $\mu$ of (4.6.8). The matrix $F$ in (4.6.16) has a block diagonal structure with the diagonal blocks being the discrete scalar convection-diffusion operators. Thus, the eigenvalues of $F$ are, in general, complex, but following from Bendixson’s theorem [134, p. 69] they have positive real parts. Namely, if we denote the eigenvalues of $F$ by $\lambda_F = \lambda_F^r \pm i \lambda_F^i$, then an SPD matrix $G = \frac{1}{2}(F + F^T) = \frac{1}{2} A$ (where $A$ is
4.6. SPECTRAL PROPERTIES OF THE PRECONDITIONED OPERATOR

the discrete Poisson operator) has all real and positive eigenvalues \( \lambda_G \), and according to Bendixson theorem \( \lambda_F \) satisfy the relation \( \lambda_G^{\min} \leq \lambda_F^r \leq \lambda_G^{\max} \), from which it follows that \( \lambda_F^r > 0 \) for all the eigenvalues of \( F \). From the block factorisation of \( F \)

\[
F = \begin{bmatrix}
F_{11} & F_{12} \\
F_{21} & F_{22}
\end{bmatrix} = \begin{bmatrix} I & O \\
F_{11}^{-1}F_{11}^{-1} & I
\end{bmatrix} \begin{bmatrix} F_{11} & F_{12} \\
F_{21} & F_{22}
\end{bmatrix} = \begin{bmatrix} I & O \\
F_{22}F_{22}^{-1} & I
\end{bmatrix} \begin{bmatrix} F_{11} & F_{12} \\
F_{21} & F_{22}
\end{bmatrix},
\]

it follows that \( \lambda_F \) are the eigenvalues of the diagonal blocks of the factor \( U \). Thus, the real parts of the eigenvalues of the Schur complements \( S_{11} \) and \( S_{22} \) are positive. This makes the real parts of \( \tilde{\lambda} \) positive in special cases (4.6.21) and (4.6.22). We can demonstrate this property for the general case (4.6.24) analytically only for \( \text{Re} = 0 \). In this case the matrix \( F \) is a discrete Laplacian (i.e. an SPD matrix), with all positive and real eigenvalues, and the same property extends to the Schur complements \( S_{11} \) and \( S_{22} \). Then the positiveness of the eigenvalues of (4.6.24) follows from the generalised Courant-Fischer theorem [94, p. 394]. For more general cases of \( \alpha \) and \( \text{Re} > 0 \) we verified numerically on a number of examples that the condition \( \text{Re}(\tilde{\lambda}) > 0 \) holds.

We now proceed to bound the non-unit eigenvalues \( \mu = \mu^r \pm \mu^i \) of (4.6.7)

\[
(4.6.27)
\]

From (4.6.19) we have

\[
\mu = -\frac{\sigma}{\sigma + \lambda^r} \pm i \lambda^i = -\frac{\sigma(\sigma + \tilde{\lambda}^r)}{(\sigma + \lambda^r)^2 + (\tilde{\lambda}^i)^2} \pm i \frac{\sigma \lambda^i}{(\sigma + \lambda^r)^2 + (\tilde{\lambda}^i)^2} = \mu^r \pm \mu^i. \tag{4.6.28}
\]

To bound the imaginary part \( \mu^i \), we find the derivative with respect to \( \sigma \):

\[
\frac{d\mu^i(\sigma)}{d\sigma} = \frac{\tilde{\lambda}^i \left[ (\sigma + \tilde{\lambda}^r)^2 + (\tilde{\lambda}^i)^2 \right] - 2\sigma \tilde{\lambda}^i (\sigma + \tilde{\lambda}^r)}{\left[ (\sigma + \lambda^r)^2 + (\tilde{\lambda}^i)^2 \right]^2} = \frac{\tilde{\lambda}^i \left[ -\sigma^2 + (\tilde{\lambda}^r)^2 + (\tilde{\lambda}^i)^2 \right]}{\left[ (\sigma + \lambda^r)^2 + (\tilde{\lambda}^i)^2 \right]^2}. \tag{4.6.29}
\]

The extremal values of \( \mu^i \) are obtained from the condition \( \frac{d\mu^i(\sigma)}{d\sigma} = 0 \) which holds

\[\text{2There is no equivalent to the Courant-Fischer theorem for non-symmetric matrices, thus there is no supporting theory to prove analytically that } \tilde{\lambda}^r > 0 \text{ in the general case.}\]
when:

\[
\tilde{\lambda}^i = 0 \quad \text{or} \quad \sigma^2 = (\tilde{\lambda}^r)^2 + (\tilde{\lambda}^i)^2 = |\tilde{\lambda}|^2. \tag{4.6.30}
\]

For \(\tilde{\lambda}^i = 0\) we have the minimum value \(\mu^i = 0\), while the case \(\sigma^2 = |\tilde{\lambda}|^2\) gives the maximum. With the assumption \(\tilde{\lambda}^r > 0\) we substitute \(\sigma^2 = |\tilde{\lambda}|^2\) into the expression for \(\mu^i\) in (4.6.28) obtaining

\[
\mu^i = \pm \frac{|\tilde{\lambda}| \tilde{\lambda}^i}{(|\tilde{\lambda}| + \tilde{\lambda}^r)^2 + (\tilde{\lambda}^r)^2} = \pm \frac{|\tilde{\lambda}| \tilde{\lambda}^i}{2|\tilde{\lambda}|^2 + 2|\tilde{\lambda}| \tilde{\lambda}^r} = \pm \frac{\tilde{\lambda}^i}{2 (|\tilde{\lambda}| + \tilde{\lambda}^r)}
\]

\[
= \pm \frac{1}{2} \frac{\tilde{\lambda}^i}{1 + \frac{\tilde{\lambda}^r}{|\tilde{\lambda}|}} \subset \left[ -\frac{1}{2}, \frac{1}{2} \right]. \quad \tag{4.6.31}
\]

as \(\tilde{\lambda}^i \leq |\tilde{\lambda}|\). Thus, the imaginary parts \(\mu^i\) of the eigenvalues \(\mu\) are bounded in the interval \([-\frac{1}{2}, \frac{1}{2}]\). The maximum is achieved when \(\sigma = |\tilde{\lambda}|\).

Now we turn to finding the bounds for \(\mu^r\). In this context we consider two cases: \(\tilde{\lambda}^i = 0\) (i.e. \(\text{Re} = 0\)), and \(\tilde{\lambda}^i \neq 0\) (i.e. \(\text{Re} > 0\)). In the former case we have:

\[
\mu^r = -\frac{\sigma (\sigma + \tilde{\lambda}^r)}{(\sigma + \tilde{\lambda}^r)^2} = -\frac{\sigma}{\sigma + \tilde{\lambda}^r} = -\frac{1}{1 + \frac{\tilde{\lambda}^r}{\sigma}}. \tag{4.6.32}
\]

This, together with an assumption \(\sigma > 0\), implies \(\mu^r \geq -1\). In order to bound \(\mu^r\) from above we need to select a specific value of \(\sigma\), similarly as in [181]. The choice \(\sigma \geq \tilde{\lambda}^r\) for all the eigenvalues \(\tilde{\lambda}^r\) of (4.6.24) implies \(\mu^r \in \left[-1, -\frac{1}{2}\right]\). In cases when \(\tilde{\lambda}^i \neq 0\) we have

\[
\frac{1}{\mu^r} = -\frac{(\sigma + \tilde{\lambda}^r)^2 + (\tilde{\lambda}^i)^2}{\sigma (\sigma + \tilde{\lambda}^r)} = -\frac{\sigma + \tilde{\lambda}^r}{\sigma} - \frac{(\tilde{\lambda}^i)^2}{\sigma (\sigma + \tilde{\lambda}^r)}
\]

\[
= -1 - \frac{\tilde{\lambda}^r}{\sigma} - \frac{(\tilde{\lambda}^i)^2}{1 + \frac{\tilde{\lambda}^r}{\sigma}} \geq -1 - \frac{\tilde{\lambda}^r}{\sigma} - \left(\frac{\tilde{\lambda}^i}{\sigma}\right)^2. \tag{4.6.33}
\]

The choice \(\sigma \geq |\tilde{\lambda}|\) implies \(\sigma \geq \tilde{\lambda}^r\) and \(\sigma \geq \tilde{\lambda}^i\). Thus

\[
\frac{1}{\mu^r} \geq -3 \quad \implies \quad \mu^r \leq -\frac{1}{3}. \tag{4.6.34}
\]

This bounds the real part \(\mu^r\) of the eigenvalues \(\mu\) when \(\text{Re} > 0\) to the interval \([-1, -\frac{1}{3}]\). Figure 4.10 summarises the eigenvalue distribution for the problem (4.6.8).

To select an optimal value of the parameter \(\sigma\), we need an estimate to the largest
4.6. SPECTRAL PROPERTIES OF THE PRECONDITIONED OPERATOR

Figure 4.10: The bounds of the eigenvalues \( \mu \) in (4.6.8) for: (left) \( \text{Re} = 0 \), (right) \( \text{Re} = 100 \).

For the case \( \text{Re} = 0 \) we can use the approach similar to that of [181], but for \( \text{Re} > 0 \) and an arbitrary tilting angle \( \alpha \) this is a non-trivial task. In our computations, reported in Chapter 6, we use the same choice as in [181], i.e. \( \sigma = \|F\|_\infty \), which proved to work well. An alternative approach would be to use Gershgorin’s theorem [94, p. 320] applied to the momentum block \( F \).

4.6.2 Spectral Analysis of the Exact Augmentation Preconditioner

Based on the results from the previous section, we proceed to characterise the spectrum of the preconditioned Jacobian (4.6.1). Starting from a \( 2 \times 2 \) blocking (4.6.5) we seek to characterise the eigenvalues of

\[
\begin{bmatrix}
\hat{F} & \hat{B}^T \\
\hat{B} & O
\end{bmatrix}
\begin{bmatrix}
\bar{v}_1 \\
\bar{v}_2
\end{bmatrix} = \nu
\begin{bmatrix}
\hat{F}_a & \hat{B}^T \\
\hat{B} & O
\end{bmatrix}
\begin{bmatrix}
\bar{v}_1 \\
\bar{v}_2
\end{bmatrix}
\Downarrow N_u + N_\ell \\
\Downarrow N_p
\] (4.6.35)

In this context we deploy a general theory for constraint preconditioning in symmetric and non-symmetric cases (see [150], [43], [10] and [63]). From (4.6.35) we have

\[
\mathcal{J}\mathcal{P}^{-1} = 
\begin{bmatrix}
\hat{F} & \hat{B}^T \\
\hat{B} & O
\end{bmatrix}
\begin{bmatrix}
\hat{F}_a^{-1} + \hat{F}_a^{-1}\hat{B}^T\hat{S}_a^{-1}\hat{B}\hat{F}_a^{-1} & -\hat{F}_a^{-1}\hat{B}^T\hat{S}_a^{-1} \\
-\hat{S}_a^{-1}\hat{B}\hat{F}_a^{-1} & \hat{S}_a^{-1}
\end{bmatrix}
\begin{bmatrix}
\hat{F}\hat{F}_a^{-1} + (\hat{F}\hat{F}_a^{-1} - I)\hat{B}^T\hat{S}_a^{-1}\hat{B}\hat{F}_a^{-1} & (I - \hat{F}\hat{F}_a^{-1})\hat{B}^T\hat{S}_a^{-1} \\
O & I
\end{bmatrix}
\] (4.6.36)

where \( \hat{S}_a = -\hat{B}\hat{F}_a^{-1}\hat{B}^T \) is the Schur complement. From (4.6.36) it follows that the preconditioned matrix \( \mathcal{J}\mathcal{P}^{-1} \) has \( N_p \) unit eigenvalues.
If we reverse the notation from (4.6.5) and (4.6.6) to a $3 \times 3$ blocking, we have the following block equations

\begin{align*}
F \bar{v}_u + \tilde{L}^T \bar{v}_\ell + B^T \bar{v}_p &= \nu F_a \bar{v}_u + \nu B^T \bar{v}_p \tag{4.6.37} \\
\tilde{L} \bar{v}_u &= \nu W \bar{v}_\ell \tag{4.6.38} \\
B \bar{v}_u &= \nu B \bar{v}_u \tag{4.6.39}
\end{align*}

Expressing $\bar{v}_\ell$ from (4.6.38) and substituting to (4.6.37) gives (similarly to (4.6.13)):

\begin{equation}
(\nu^2 - \nu) F \bar{v}_u + (\nu^2 - 1) \tilde{L}^T W^{-1} \tilde{L} \bar{v}_u + (\nu^2 - 1) B^T \bar{v}_p = \bar{0}. \tag{4.6.40}
\end{equation}

From (4.6.40) it follows that $\nu = 1$ is the eigenvalue of (4.6.35) with the same multiplicity as $\mu$ is for (4.6.8), i.e. we have further $N_u$ unit eigenvalues. The remaining $N_\ell$ non-unit eigenvalues are characterised by the generalised eigenvalue problem

\begin{equation}
\tilde{Z}^T \tilde{F} \tilde{Z} \bar{v} = \nu \tilde{Z}^T \tilde{F}_a \tilde{Z} \bar{v}, \tag{4.6.41}
\end{equation}

where $\tilde{Z} \in \mathbb{R}^{(N_u + N_\ell) \times (N_u + N_\ell - N_p)}$ is the basis of the null space of $\tilde{B} = [B \ O]$. The exact bounds of the eigenvalues of (4.6.41) are not available, although some estimates, based on Cauchy’s interlace theorem, are possible in the cases when the $(1,1)$ block is an SPD matrix. Some further insight into the eigenvalues of (4.6.41) might be possible in the symmetric case ($\text{Re} = 0$) by extending the analysis from [95]. We emphasise that in our case of constraint preconditioning, the spectral equivalence of the $(1,1)$ block in the original matrix and the preconditioner is the ideal case. The conjecture is that the tight clustering of the eigenvalues $\mu$ of (4.6.7) demonstrated in the previous subsection would lead to favourable eigenvalue distribution for $N_\ell$ non-unit eigenvalues of (4.6.35). This conjecture is verified numerically by calculating the eigenvalues for the symmetric case in Chapter 6. In addition, convergence results from Chapter 6 with the exact version of the preconditioner $P$ demonstrate low and bounded iteration counts in all cases, suggesting that the non-unit eigenvalues of (4.6.35) are tightly clustered.
4.7 Summary

In this chapter we have introduced the concept of weakly imposed Dirichlet BCs via Lagrange multipliers. This approach leads to the augmented structure of the standard Navier-Stokes problem in the discrete case. The structure of the resulting discrete problem and the spectral properties of the constraint blocks allow efficient preconditioning of the linear system by the Augmented Lagrangian type methodology.

We presented a novel preconditioning technique for which we characterised the spectrum of the preconditioned operator in both symmetric and non-symmetric case. The analysis reveals that the preconditioned operator has a majority of unit eigenvalues with a small number (equal to the size of the constraint space) of eigenvalues that we conjecture are tightly clustered. In Chapter 6 we verify our analytical results numerically.
Chapter 5

The Development of oomph-lib’s Parallel Block Preconditioning Framework

In this chapter we discuss oomph-lib’s parallel Block Preconditioning Framework (BPF) which aided the implementation and testing of the augmentation preconditioner introduced in Section 4.5. For a detailed discussion on the usage of the BPF (including new functionalities) see [126].

The BPF is a software middleware, developed to facilitate the rapid and efficient development of parallel block preconditioners within oomph-lib. As discussed in Section 2.5, a design principle of parallel computing within oomph-lib is that parallelisation of different phases in finite element simulation, such as mesh generation, the assembly of the Jacobian and the solution of the linear systems, is done using MPI and involve little or no intervention from the user. This philosophy extends to the BPF, in the sense that the development of a block preconditioner should not involve the complexities associated with parallel implementation, i.e. the code written for a serial implementation should parallelise automatically. Another design principle of the BPF is the re-use of existing block preconditioners. The developer should be able to use the existing block preconditioners as building blocks for more complex preconditioners without modifying the existing code or re-implementing a new version of it specific to their problem. This allows preconditioners for fairly complex multi-physics problems to be developed rapidly and the existing efficient implementation of the component (single-physics) preconditioners should lead to an efficient overall implementation.

An initial version of the parallel BPF was implemented by Muddle in [180]. This
5.1. **OVERVIEW OF THE INITIAL VERSION OF THE BPF**

was used to implement generic block preconditioners (i.e. block diagonal and block triangular preconditioners discussed in [Section 3.4.2](#)), as well as problem specific preconditioners such as the LSC preconditioner for the Naiver-Stokes problem [120] and a block preconditioner for the fluid structure interaction (FSI) problem [118]. The FSI preconditioner involves the re-use of the LSC preconditioner as its component. However, the initial version was not general enough to facilitate the implementation of block preconditioners with a more sophisticated structure, such as our augmentation preconditioner. Thus, a major task of this project was to extended the functionality of the parallel BPF. In addition, further performance optimisations were introduced, leading to an improved overall efficiency.

5.1 **Overview of the Initial Version of the BPF**

In this section we review the functionality of the BPF that was available at the start of the project. This version was developed by R. Muddle and described in [180]. The key concepts of the BPF are explained on the case of implementing the LSC preconditioner.

When the Jacobian matrix is assembled, the global numbers of the degrees of freedom (DOF) follow the natural ordering of the elements and nodes. This ordering assumes consecutive enumeration of all DOFs associated with a node, with the nodes sequentially numbered with their global number. Block preconditioning requires special enumeration schemes for the unknowns in the problem (equivalent to re-ordering the linear systems, i.e. a permutation of the rows and columns) where the unknowns corresponding to each type of DOF are grouped together and enumerated consecutively. This leads to a natural block structure of the linear systems. For instance, Navier-Stokes problems discussed in [Chapter 2](#) involve $d$ velocity DOF types and a pressure DOF type, where $d = 2$ [3] is the spatial dimension. In two dimensions we re-order the linear system to group together the three DOF types corresponding to the unknowns $u_1$, $u_2$ and $p$, which yields the $3 \times 3$ DOF type structure:

\[
\begin{bmatrix}
F_{11} & F_{12} & B_{1}^T \\
F_{21} & F_{22} & B_{2}^T \\
B_{1} & B_{2} & O
\end{bmatrix}
\begin{bmatrix}
\delta \bar{u}_1 \\
\delta \bar{u}_2 \\
\delta \bar{p}
\end{bmatrix}
= - \begin{bmatrix}
\bar{r}_1 \\
\bar{r}_2 \\
\bar{r}_p
\end{bmatrix}.
\] (5.1.1)

Application of the LSC preconditioner requires the (exact or approximate) solution of
the linear system

\[
\begin{bmatrix}
F & B^T \\
O & -S_{LSC}
\end{bmatrix}
\begin{bmatrix}
\bar{z}_u \\
\bar{z}_p
\end{bmatrix} =
\begin{bmatrix}
\bar{y}_u \\
\bar{y}_p
\end{bmatrix}
\] (5.1.2)

for \(\bar{z} = [\bar{z}_u, \bar{z}_p]^T\) at each Krylov iteration. Here, \(S_{LSC}\) is the LSC approximation to the pressure Schur complement \(S = BF^{-1}B^T\):

\[
S_{LSC} = (B\hat{M}_u^{-1}B^T)(B\hat{M}_u^{-1}F\hat{M}_u^{-1}B^T)^{-1}(B\hat{M}_u^{-1}B^T),
\] (5.1.3)

where \(\hat{M}_u\) is the diagonal of the velocity mass matrix. Then we can solve the system (5.1.2) by block back-substitution:

\[
\bar{z}_p = -(B\hat{M}_u^{-1}B^T)^{-1}(B\hat{M}_u^{-1}F\hat{M}_u^{-1}B^T)(B\hat{M}_u^{-1}B^T)^{-1}\bar{y}_p
\] (5.1.4)

\[
\bar{z}_u = F^{-1}(\bar{y}_u - B^T\bar{z}_p)
\] (5.1.5)

If we were to invert the entire \(F\) block, then we must group together the velocity DOF types from the sub-block matrices presented in (5.1.1):

\[
\begin{bmatrix}
F & B^T \\
B & O
\end{bmatrix}
\begin{bmatrix}
\delta \bar{u} \\
\delta \bar{p}
\end{bmatrix} =
\begin{bmatrix}
\bar{r}_u \\
\bar{r}_p
\end{bmatrix}.
\] (5.1.6)

Evaluation of the expression (5.1.4) requires two linear solves involving the sparse pressure Poisson operator

\[
P = (B\hat{M}_u^{-1}B^T),
\] (5.1.7)

and a sequence of sparse matrix-vector products to apply \((B\hat{M}_u^{-1}F\hat{M}_u^{-1}B^T)\). For the purpose of preconditioning, the system (5.1.2) does not have to be solved exactly. Therefore we can replace the solution of linear subsystems involving \(F\) and \(P\) by existing preconditioners (interpreted as inexact solvers). In oomph-lib, these are known as subsidiary preconditioners.

The example above shows that the application of block preconditioners require several generic steps: classification of the DOFs in (5.1.1), determining the block structure of the preconditioner from the DOF types and the application of subsidiary preconditioning operators including either a solution of a linear system (e.g. for (5.1.5) and (5.1.4)) or the evaluation of matrix-vector products using some of the blocks that are extracted from the original linear system or computed during the assembly of the preconditioner. In the sequel we describe how these tasks are performed within
oomph-lib’s BPF.

All block preconditioners defined in oomph-lib inherit from the base class BlockPreconditioner which provides generic functionality required for an implementation of a block preconditioner, such as generating lookup schemes for classifying DOFs, DOF types, and block types, and the functionality to extract matrices and vectors. All inter-processor communication is implemented within the BlockPreconditioner class, relieving the programmer of dealing with complicated parallel programming details when implementing a new preconditioner. This implies that a serial implementation of a preconditioner is automatically parallelised when oomph-lib is compiled with MPI support. All BlockPreconditioners must implement two key functions, setup(…), which is automatically called at each Newton iteration to assemble the preconditioner and setup the solution procedures for all subsidiary preconditioners (for example, either LU factorisation or AMG coarsening), and preconditioner_solve(…) which is called automatically to apply the inverse of the preconditioner to a vector at each Krylov iteration. In this discussion we are primarily concerned with the setup(…) function.

We first describe how to classify the DOFs which yields the DOF type ordering. In the sequel, the indices of mathematical objects follow the natural number ordering, starting from 1. When referring to a code snippet, we use the C++ zero-based indexing. To facilitate the classification of DOFs, the class GeneralisedElement contains two pure virtual functions that must be implemented by the user for each concrete problem to label the DOFs (within an element) with their type. The functions are:

- GeneralisedElement::ndof_types() must return the number of DOF types associated with an element.
- GeneralisedElement::get_dof_numbers_for_unknowns(...) must return a list of pairs comprising a map from global DOF number to DOF type for all unknowns in the element.

We refer to this classification of DOFs as the elemental DOF type classification. When these two functions are implemented, the element is termed as block preconditionable. Many block preconditionable elements are already implemented in oomph-lib. For the two-dimensional Navier-Stokes, we can use the block preconditionable Navier-Stokes elements QTaylorHoodElement<2>, for which ndof_types() returns 3 (two velocity DOF types and one pressure DOF type), and enumerate the DOFs 0, 1, 2 for $u_1$, $u_2$ and $p$, respectively. This will produce the DOF type ordering as in (5.1.1).
Some elements may classify, in addition to their own DOFs, the DOFs of adjacent elements. For example, the \texttt{ImposeParallelOutflowElement} used to impose the parallel outflow constraint $\mathbf{u} \cdot \mathbf{t} = 0$ (see Section 4.3), classify its own DOFs and the velocity DOFs of the adjacent bulk element. For a two-dimensional problem, this element has three DOF types, two for the velocity components in the bulk element, $u_1^c$ and $u_2^c$, and one for the Lagrange multiplier $\Lambda$. We have used superscript $c$ to denote the velocity DOFs classified by the \texttt{ImposeParallelOutflowElement}. The DOF types are enumerated as 0, 1 and 2 for the unknowns $u_1^c$, $u_2^c$ and $\Lambda$, respectively. In this construction, the velocity DOFs on the boundary with \texttt{ImposeParallelOutflowElement} are classified twice, both by \texttt{QTaylorHoodElement<2>} and \texttt{ImposeParallelOutflowElement}. If certain DOFs are classified by multiple elements, then the most recent classification of DOF type overwrites all previous assignments.

The information on the elemental classification of DOFs is then passed to the BPF. This can be done using the meshes. In the context of the BPF, finite element meshes are interpreted as containers for elements and are used to distinguish between different types of elements. Therefore when using the BPF, a mesh must contain only one element type. To give the BPF access to the meshes, two functions have to be called, usually within the \texttt{setup(...)} function:

- \texttt{BlockPreconditioner::set_nmesh(...)} takes an \texttt{unsigned}, the number of meshes in the problem.
- \texttt{BlockPreconditioner::set_mesh(...)} takes two arguments, a pointer to a mesh and the unique ID of the mesh. This must be called once for each mesh in the problem.

Given a list of meshes used to construct the \texttt{Problem}, the BPF visits each mesh following the ID order and assigns a DOF type to the associated global DOF, followed by the DOFs in the next mesh, and so on. For example, assume that a two-dimensional Navier-Stokes problem is discretised using Taylor-Hood elements and the parallel outflow condition is imposed by Lagrange multipliers implemented by face elements. Then the problem has two meshes (thus \texttt{set_nmesh(2)} must be called). If \texttt{Bulk_mesh_pt} is a pointer to a mesh of \texttt{QTaylorHoodElement<2>} elements and \texttt{Boundary_mesh_pt} a pointer to a mesh of \texttt{ImposeParallelOutflowElement} elements, then the instructions:
\begin{verbatim}
set_mesh(0, Bulk_mesh_pt);
set_mesh(1, Boundary_mesh_pt);
\end{verbatim}
assign ID 0 to the bulk mesh and 1 to the boundary mesh, and the relationship between meshes, elemental DOF types and DOF types is summarised in the following table:

<table>
<thead>
<tr>
<th>Mesh position</th>
<th>Bulk</th>
<th>Boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Associated unknowns</td>
<td>( u_1 )</td>
<td>( u_2 )</td>
</tr>
<tr>
<td>Elemental DOF ordering</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>DOF type ordering</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Swapping the mesh IDs corresponds to a different block permutations of the coefficient matrix (i.e. \( u_1' \) would have DOF type number 0 rather than 3). Thus, the order in which the meshes are enumerated does matter.

After `set_nmesh(...)` and `set_mesh(...)` have been called, the BPF has access to the necessary information to determine the number of DOF types and their order. The order of the DOF types is of particular importance, as problem specific block preconditioners assume some sort of DOF type ordering. For example, a two-dimensional problem consisting of a single mesh of `QTaylorHoodElement<2>` elements, assumes the DOF type order

\[
\begin{bmatrix}
  u_1 \\
  u_2 \\
  p 
\end{bmatrix}
\]

Any subsequent implementation of a Navier-Stokes preconditioner, such as the LSC preconditioner implemented in the class `NavierStokesSchurComplementPreconditioner`, must follow this convention.

The BPF operates on the sub-blocks of the Jacobian. To facilitate extraction of the sub-blocks, we must associate a block type with each DOF type. A block type consists of one or more DOF type(s), thus, the number of block types is at most equal to the number of DOF types. For example, in (5.1.1) we have three DOF types, therefore it is impossible to have more than three block types. The case of the finest granularity of block types (corresponding to the 1-1 mapping between the block and the DOF types) gives rise to the DOF blocks. When a block type consist of more than one DOF types, the associated block matrices are the compound blocks (e.g. \( F \) in (5.1.6)). The ability to have different granularity between the DOF and the block types is required to facilitate the idea of hierarchical design of preconditioners for complex (multi-physics) problems. In this context we need to know the information about different unknowns that will be involved in creating preconditioners at all levels of the problem hierarchy. The DOF types define the finest granularity of the unknowns required in this context. The requirement to combine multiple DOF types to create preconditioners at different
levels of the problem hierarchy is catered for by the introduction of the block types. The next step is to determine the block types from the DOF types.

The block structure of the system is handled by the \texttt{BlockPreconditioner::block\_setup(...)} function. This must be called within the block preconditioner’s \texttt{setup(...)} function. It takes a \texttt{Vector<unsigned>}, usually named \texttt{dof\_to\_block\_map} as an input. This vector defines the mapping between DOF types and block types. The size of this vector is equal to the total number of DOF types at a current level of the preconditioner hierarchy. It has the following construction:

\[
dof\_to\_block\_map[\text{DOF type}] = \text{block type}
\]

For example, the $2 \times 2$ block structure of the Navier-Stokes problem is obtained by calling \texttt{block\_setup(...)} with the input argument $[0,0,1]$, which merges together the two velocity and one pressure $3 \times 3$ DOF types of (5.1.1) as follows:

\[
J = \begin{bmatrix}
\begin{array}{cc|c}
  u_1 & u_2 & p \\
  F_{11} & F_{12} & (B_1)^T \\
  F_{21} & F_{22} & (B_2)^T \\
\end{array}
\end{bmatrix}. \tag{5.1.9}
\]

By contrast, \texttt{block\_setup([0,1,2])} would produce the DOF block structure:

\[
J = \begin{bmatrix}
\begin{array}{cc|c}
  u_1 & u_2 & p \\
  F_{11} & F_{12} & (B_1)^T \\
  F_{21} & F_{22} & (B_2)^T \\
  B_1 & B_2 & O \\
\end{array}
\end{bmatrix}. \tag{5.1.10}
\]

Once \texttt{block\_setup(...)} has been invoked, the following functions can be called to extract the matrix blocks from the Jacobian which are required to assemble the preconditioner:

- \texttt{BlockPreconditioner::ndof\_types()} returns the number of DOF types. For both (5.1.9) and (5.1.10), this function returns 3.
- \texttt{BlockPreconditioner::nblock\_types()} returns the number of block types. This function returns 2 for (5.1.9) and 3 for (5.1.10).
- \texttt{BlockPreconditioner::get\_block(...)} is used to extract a block from the

\footnote{In \texttt{oomph-lib}, the \texttt{Vector} class is a slight extension to the standard template \texttt{vector} class so that ‘graceful’ array range checks can be included.}
naturally ordered Jacobian. The block is in the compressed row storage format, implemented in the class \texttt{CRDoubleMatrix}. For example, within the \texttt{NavierStokesSchurComplementPreconditioner::setup(\ldots)} function, where the block structure is set up as given by \eqref{eq:5.1.9} to extract the $F$ block from \eqref{eq:5.1.6}, we call:

\begin{verbatim}
CRDoubleMatrix F_block = get_block(0,0);
\end{verbatim}

An important point to note is that the $F$ block is extracted from a naturally ordered Jacobian and hence it will not have a component substructure as in \eqref{eq:5.1.9}. In the case of the block setup \eqref{eq:5.1.10}, the above function call will return $F = \begin{bmatrix} F_{11} \end{bmatrix}$. Once we have the required block, other preconditioner operations can be performed with it. Where possible, we use efficient third party libraries to perform most of the preconditioning operations, such as \texttt{SuperLU} \cite{60} for LU factorisation Trilinos’ \texttt{Epetra} \cite{243} and \texttt{EpetraExt} \cite{244} libraries for sparse matrix-vector and sparse matrix-matrix products, respectively.

Once the preconditioner has been assembled in the setup phase, its action is implemented in the \texttt{preconditioner_solve(\ldots)} function. A typical implementation involve the following operations:

- Calls to \texttt{get\_block\_vector(\ldots)}, which extracts a block vector from a naturally ordered global vector. Both the block vector and the global vector are defined in \texttt{oomph-lib} as \texttt{DoubleVector} objects (with the elements stored in a C-style array of \texttt{doubles}). For example, if the block structure is set up as in \eqref{eq:5.1.9} to extract $\hat{y}_u$ defined by \eqref{eq:5.1.2} from the global vector $y$, we call:

\begin{verbatim}
DoubleVector y_u;
this->get_block_vector(0,y,y_u);
\end{verbatim}

- The application of the preconditioning constructed (prepared) during the setup phase, such as forward and backwards substitution.

- Calls to \texttt{return\_block\_vector(\ldots)}, which transforms (permutes) a block vector to a naturally ordered global vector. For example, after we have solved the system \eqref{eq:5.1.5} to feed back $\hat{z}_u$ defined by \eqref{eq:5.1.2} to a global vector we call:

\begin{verbatim}
this->return_block_vector(0,z_u,z);
\end{verbatim}

All BPF functions are automatically parallelised. For example, the \texttt{get\_block(\ldots)} function will return a uniformly row distributed matrix among the processors (with
roughly equal number of rows per processor) when executed with MPI. The function `get_block_vector(...)` will also return vector with rows uniformly distributed among the processors in a coherent manner with the preconditioner matrix. Interprocessor communication is used to ensure that the (sub-block) matrices and (sub-block) vectors are uniformly distributed.

Now we introduce how the preconditioning hierarchy is formed. In the context of the BPF, the terms master and subsidiary describe a connection between to preconditioners in a preconditioning hierarchy. The preconditioner for the original (top-level) problem is referred to as a master preconditioner. A preconditioner used within another preconditioner (at any lower level of the hierarchy) is referred to as a subsidiary preconditioner. A preconditioner which has a subsidiary preconditioner is referred to as a master preconditioner. The BPF was envisaged and designed as a generic tool that will aid the development and implementation of arbitrary block preconditioners in oomph-lib. The library was envisaged as a framework for hierarchical FEM modelling of multi-physics problems, using a bottom up approach of combining in a monolithic fashion already developed simpler models as constituent parts. This motto was extended to the BPF, where the preconditioners for complex multi-physics problems are created in a top-down hierarchical manner, re-using already developed preconditioners for the constituent sub-problems. This is summarised in [Figure 5.1](#). Consider for example the following version of the LSC preconditioner:

\[
\hat{P}_{LSC} = \begin{bmatrix}
F_{BUT} & B^T \\
- & -S_{LSC}
\end{bmatrix}
= \begin{bmatrix}
F_{11} & F_{12} & B_{11}^T \\
F_{21} & F_{22} & B_{21}^T \\
& & -S_{LSC}
\end{bmatrix}
\tag{5.1.11}
\]

where \(F_{BUT}\) is the block upper triangular approximation of the momentum block \(F\) (as suggested in [67]) and \(S_{LSC}\) is defined by [Equation 5.1.3](#). Then \(\hat{P}_{LSC}\) is a master preconditioner, which is a problem specific preconditioner, consisting of two subsidiary preconditioners:

- For the subsystem defined [Equation 5.1.5](#), we use the preconditioner \(F_{BUT}\), which is a general, block upper triangular preconditioner for the \(F\) block (following the blue arrow in [Figure 5.1](#)). This preconditioner has two diagonal blocks, which are scalar problems and can be inverted either exactly by SuperLU or approximately by AMG (following the red arrow in [Figure 5.1](#)). Note that \(F_{BUT}\) is a master preconditioner for the two (trivial) sub-preconditioners \(F_{11}\) and \(F_{22}\).
The preconditioner $S_{LSC}$ which is a method for the exact or approximate inversion of the Schur complement. In general Schur complements may themselves be considered as preconditioners, but in this case it is a scalar problem consisting exact or approximate inversion of two scalar problems defined by $[5.1.7]$ plus a number of sparse updates.

The re-use of existing preconditioners is facilitated by turning them into subsidiary preconditioners. This is achieved by the member function `turn_into_subsidiary_block_preconditioner(...)`. This function takes a pointer to the (current) master preconditioner and a `Vector` specifying the DOF types of the (current) master preconditioner to be used by the subsidiary preconditioners. We demonstrate this functionality by turning an existing implementation of a general upper triangular preconditioner into a subsidiary preconditioner in the context of $[5.1.11]$, when approximating $F$ by $F_{BUT}$. In `oomph-lib`, a block upper triangular preconditioner is implemented in the class `BlockTriangularPreconditioner`. It uses the identity map for `dof_to_block_map`. The `NavierStokesSchurComplementPreconditioner` provides an access function `set_f_preconditioner(...)` which takes a pointer to an existing preconditioner and sets the class variable `F_preconditioner_pt` (this is called from within...
CHAPTER 5. PARALLEL BLOCK PRECONDITIONING FRAMEWORK

If F_preconditioner_pt points to BlockTriangularPreconditioner, then it can be turned into a subsidiary preconditioner within the NavierStokesSchurComplementPreconditioner::setup(...) function by the following instructions:

Vector<unsigned> dof_map(nvelocity_dof_types);
for (unsigned i = 0; i < nvelocity_dof_types; i++)
{dof_map[i] = i;}
F_block_preconditioner_pt->
    turn_into_subsidiary_block_preconditioner(this, dof_map);
F_block_preconditioner_pt->setup(matrix_pt());

The variable nvelocity_dof_types is the number of velocity DOF types, matrix_pt() is an access function to the naturally ordered Jacobian. The vector dof_map contains the mapping between the subsidiary and master DOF types, for example, dof_map[i] = j means that the subsidiary DOF type i is associated with the master DOF type j. Let \( N^S_{dof} \) and \( N^M_{dof} \) be the number of DOF types which the subsidiary and master preconditioner operate on, respectively. Then the dof_map vector needs to be of the size \( N^S_{dof} \) (set by the master preconditioner) and to contain an injective mapping from the set of subsidiary to the set of master DOF types (in our example, dof_map = [0,1]).

At the BlockTriangularPreconditioner level, the DOF type order and block structure are

\[
\begin{bmatrix}
    u_1 & u_2 \\
    0 & 1
\end{bmatrix}
\quad \text{and} \quad
u_1 \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} u_2
\]

respectively. This can be used to construct a block upper triangular preconditioner using existing BPF functions described in [180, pp. 204–210]. The vector dof_map plays a vital role in determining the DOF type order of the subsidiary block preconditioners. For example, if the master Navier-Stokes preconditioner passes the dof_map = [1,0] to the subsidiary block triangular preconditioner, this defines the mapping \( u^M_1 \rightarrow u^S_2 \) and \( u^M_2 \rightarrow u^S_1 \) where the superscripts \( M \) and \( S \) denotes the DOF types for the master and subsidiary preconditioners, respectively. The BlockTriangularPreconditioner still passes the identity dof_to_block_map to it’s block_setup(...) function, and the resulting block structure is

\[
\begin{bmatrix}
    u_2 & u_1 \\
    F_{22} & F_{21} \\
    F_{12} & F_{11}
\end{bmatrix}
\]
### Remark 5.1.1

The block structure of a master preconditioner does not play a role in determining how the subsidiary preconditioners work. Therefore only the DOF type information is passed from master to subsidiary preconditioners. For example, the `NavierStokesSchurComplementPreconditioner` may pass the `dof_to_block_map` with entries `[0,0,1]` or `[0,1,2]` to its `block_setup(...)` function, which give rise to the block structure (5.1.9) or (5.1.10) respectively. In both cases, the subsidiary preconditioner for the $F$ block must still receive a vector `dof_map` associating the subsidiary DOF types with the master DOF types. A fundamental design philosophy in the BPF is that the user does not need to know the block structure of a block preconditioner to use it. They just need to know the required DOF type ordering (i.e. the number and the order of the DOF types). This order is assumed by an implementation of a preconditioner for a particular problem.

Each preconditioner sets up its own block structure, for example, general preconditioners always use the most fine grained block structure and the LSC preconditioner will always work with a $2 \times 2$ block structure (which corresponds to the velocity and pressure blocks). But we can still use the BUT preconditioner to approximate the $F$ block in the LSC preconditioner, even though the LSC operates on the entire $F$ block. This is summarised in **Figure 5.2**.

**Figure 5.2**: A generic relationship between the DOF types $\{D^\alpha_i\}$ and block types $\{B^\alpha_i\}$ for a hierarchy of preconditioners $P_\alpha$ ($\alpha = 1, 2, 3$). Each preconditioner sets up its block structure with the `dof_to_block_map` vector. Between hierarchy levels, only DOF type information is passed, via the `dof_map` vector.

In this section we have described the aim and design philosophy of the BPF and introduced the existing functionalities in the initial version. In the next section we identify the limitations of the BPF and motivate the need for additional functionalities.
5.2 Limitations of the initial version of the BPF

In this section we discuss the lack of flexibility and functionality with the initial version of the BPF. These issues prevented the rapid development of the augmentation preconditioner since we were not able to re-use existing implementation of the LSC preconditioner. The issue with flexibility is discussed in Section 5.2.1, where the limitations associated with the mapping between subsidiary and master DOF types are explored. Then we discuss functionality in Section 5.2.2 where the issues with the initial implementation of the get_block(...) function are revealed.

5.2.1 Issues with the Discrepancy of DOF Type Granularity

The rapid development of block preconditioners in oomph-lib is facilitated by the BPF’s ability to re-use existing preconditioners. In the initial version of the BPF, there must be an injective map from the set of subsidiary DOF types to the master DOF types. This map is defined in dof_map and is passed from the master preconditioner to a subsidiary preconditioner via the function turn_into_subsidiary_block_preconditioner(...). Recall from the previous section that the DOF type ordering in a preconditioner is prescribed. With the knowledge of this ordering, we can correctly construct a dof_map which associates every subsidiary DOF type with a master DOF type. However, if there is a mismatch in the granularity between the DOF types of a master preconditioner and an existing implementation of a subsidiary preconditioner, this implementation cannot be used. A new implementation with matching DOF type granularity is required.

We demonstrate this limitation with a master augmentation preconditioner (see Section 4.5) and a subsidiary LSC preconditioner. Assume that we have a bulk mesh of QTaylorHoodElement<2> elements with mesh ID 0 and a boundary mesh of Impose-ParallelOutflowElement with mesh ID 1. Then the resulting DOF type order is:

\[
\begin{array}{cccccc}
\text{Associated unknown} & u_1 & u_2 & p & u_1^c & u_2^c & A \\
\text{DOF type} & 0 & 1 & 2 & 3 & 4 & 5
\end{array}
\]  

(5.2.1)

The next step is to construct the blocks from the DOF types with a call to block_setup(...). The augmentation preconditioner (which is a problem specific preconditioner) works with the DOF blocks. If we pass the identity map to block_setup(...), then the block structure is given by (5.2.1). Instead we construct a block structure
5.2. LIMITATIONS OF THE INITIAL VERSION OF THE BPF

where the velocity unknowns for each Cartesian direction come first, then the pressure unknown, followed by the Lagrange multiplier unknown. This is achieved with the \texttt{dof_to_block_map} = [0,2,4,1,3,5], resulting in a block structure:

\[
J = \begin{bmatrix}
  u_1 & u_1^* & u_2 & u_2^* & p & \Lambda \\
  F_{11} & F_{11} & F_{12} & F_{12} & (B_1)^T & O \\
  F_{11} & F_{11} & F_{12} & F_{12} & (B_1)^T & M_1 \\
  F_{21} & F_{21} & F_{22} & F_{22} & (B_2)^T & O \\
  F_{21} & F_{21} & F_{22} & F_{22} & (B_2)^T & M_2 \\
  B_1 & B_1 & B_2 & B_2 & O & O \\
  O & M_1 & O & M_2 & O & O
\end{bmatrix}
\]

(5.2.2)

Our augmentation preconditioner \texttt{[4.5.1]} is the 6 \times 6 block matrix:

\[
P = \begin{bmatrix}
  F_{11} & F_{11} & F_{12} & F_{12} & (B_1)^T & O \\
  F_{11} & \tilde{F}_{11} & F_{12} & \tilde{F}_{12} & (B_1)^T & O \\
  F_{21} & F_{21} & F_{22} & F_{22} & (B_2)^T & O \\
  F_{21} & \tilde{F}_{21} & F_{22} & \tilde{F}_{22} & (B_2)^T & O \\
  B_1 & B_1 & B_2 & B_2 & O & O \\
  O & O & O & O & O & \tilde{W}
\end{bmatrix}
\]

(5.2.3)

where \(\tilde{W}\) is a matrix consisting of the diagonals of \(\frac{1}{\sigma}(M_1^2 + M_2^2)\), and \(\tilde{F}_{ij} = F_{ij} + M_i\tilde{W}^{-1}M_j\) are the augmented blocks \((i,j = 1,2)\). The block structure in \texttt{[5.2.3]} is essentially a 2 \times 2 block diagonal matrix (indicated by the thicker partition lines), where the (1,1) block which corresponds to the Navier-Stokes problem is decomposed further into a 5 \times 5 block matrix. The 5 \times 5 block subdivision of the Navier-Stokes sub-matrix (rather than the usual 3 \times 3 in two dimensions) is required in this case as we need to identify a subset of velocity DOFs affected by the Lagrange multipliers in order to construct the augmented blocks \(\tilde{F}_{ij}\). Suppose we want to approximate the Navier-Stokes block in \texttt{[5.2.3]} by the LSC preconditioner. Within the implementation of the augmentation preconditioner, there exists a class variable named \texttt{Navier_stokes_block_preconditioner_pt}. Setting this pointer allows the user to approximate the inverse of the Navier-Stokes block with an existing implementation of a preconditioner. Suppose that in the \texttt{setup(...)} function of the augmentation preconditioner we declared an existing implementation of the LSC preconditioner as
the subsidiary preconditioner with the following code:

```cpp
Navier_stokes_block_preconditioner_pt
  ->turn_into_subsidary_block_preconditioner(this, dof_map);
Navier_stokes_block_preconditioner_pt
  ->setup(matrix_pt());
```

The master augmentation preconditioner requires the LSC subsidiary preconditioner to operate on five of its DOF types (four velocity and one pressure). However, for a two-dimensional problem, the existing implementation of the LSC preconditioner is designed to work with only three DOF types in the DOF type ordering presented by (5.1.8). This is a consequence of the elemental DOF type ordering imposed by the QTaylorHoodElement<2> elements. Recall that the DOF types are used to facilitate two procedures: generating the block structure (via the `block_setup(...)` function), and generating the preconditioning hierarchy by associating the DOF types of the subsidiary preconditioner to its master preconditioner (via the `turn_into_subsidary_block_preconditioner(...)` function). The mismatched granularity between the master preconditioner and the subsidiary preconditioner affects both of these two procedures. We explore this below.

Assume that the master augmentation preconditioner has turned the subsidiary LSC preconditioner with a `dof_map` vector consisting of the velocity and pressure DOF types. One approach is `dof_map = [0,3,1,4,2]`, then the DOF type association between the subsidiary and the master is:

<table>
<thead>
<tr>
<th>Subsidiary DOF type</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master DOF type</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Associated unknowns</td>
<td>$u_1$</td>
<td>$u_1^c$</td>
<td>$u_2$</td>
<td>$u_2^c$</td>
<td>$p$</td>
</tr>
</tbody>
</table>

Thus the DOF type ordering given to the subsidiary LSC preconditioner is

```
Associated unknowns                  $u_1^c$ $u_1$ $u_2$ $u_2^c$ $p$
Subsidiary DOF type                  $0$ $1$ $2$ $3$ $4$
```

(5.2.4)

The existing implementation of the LSC preconditioner assumes the DOF type ordering (5.1.8), thus it constructs the vector `dof_to_block_map = [0, 0, 1]`. However, the master augmentation preconditioner passes five DOF types to it. In this case, the block structure (5.1.9) corresponds to `dof_to_block_map = [0, 0, 0, 0, 1]`. This issue can be easily fixed with a small modification to the existing LSC preconditioner implementation: Given `n_dof_types` (this comes from the size of the `dof_map` vector from the master preconditioner) construct the `dof_to_block_map` vector by initialising a zero vector of size `n_dof_types` and set the last entry to 1. This is a special
5.2. LIMITATIONS OF THE INITIAL VERSION OF THE BPF

In this case since the NavierStokesSchurComplementPreconditioner works with two block types, where the first block type contains DOF types 0 to ndof_types – 2 and the second block type contains the single remaining DOF type. However, this approach would not be suitable if the master preconditioner had passed to the LSC preconditioner more than one pressure DOF type. That is, for every possible DOF type ordering of the Navier-Stokes Jacobian, the LSC preconditioner have to create an appropriate dof_to_block_map vector mapping DOF types of this fine grained DOF ordering to the standard 2 × 2 blocking.

Continuing with the previous example, where the LSC preconditioner has been modified to work with the DOF type ordering (5.2.4), assume that we want to use the existing implementation of BlockTriangularPreconditioner to form a BUT approximation for the F block. Then the master LSC preconditioner indicate to the subsidiary BUT preconditioner which DOF types to operate on with the following code:

```cpp
// More general approach to get the velocity DOF types
// nvelocity_dof_types is 4 in this case
nvelocity_dof_types = ndof_types() - 1;
Vector<unsigned> dof_map(nvelocity_dof_types);
for (unsigned i = 0; i < nvelocity_dof_types; i++)
    dof_map[i] = i;
F_block_preconditioner_pt->
    turn_into_subsidiary_block_preconditioner(this, dof_map);
F_block_preconditioner_pt->setup(matrix_pt());
```

Here, ndof_types() returns 5 therefore dof_map = [0,1,2,3]. This means that the DOF types of the subsidiary preconditioner for the F block are:

\[
\begin{array}{cccc}
\text{Associated unknown} & u_1 & u_2^c & u_2 \\
\text{Subsidiary DOF type} & 0 & 1 & 2 & 3
\end{array}
\]  

(5.2.5)

Based on (5.1.11) we would expect a subsidiary BUT preconditioner of the form:

\[
\begin{pmatrix}
F_{11} & F_{12} \\
F_{12} & F_{22}
\end{pmatrix}
= \begin{pmatrix}
\tilde{F}_{11} & \tilde{F}_{12} \\
\tilde{F}_{12} & \tilde{F}_{22}
\end{pmatrix}.
\]

(5.2.6)

However, the BlockTriangularPreconditioner is a general purpose preconditioner
which works with the DOF blocks, this leads to the block structure:

\[
\begin{bmatrix}
  F_{11} & F_{12} & F_{13} & F_{14} \\
  F_{21} & F_{22} & F_{23} & F_{24} \\
  F_{31} & F_{32} & F_{33} & F_{34} \\
  F_{41} & F_{42} & F_{43} & F_{44}
\end{bmatrix}
\]  

In the initial version of the BPF, to achieve the block structure (5.2.6), we would have to write another version of an upper block triangular preconditioner which deals with specifically the DOF types assumed by the augmentation preconditioner.

To conclude, the initial version of the BPF required a match in the granularity of the DOF types between the master preconditioner and an existing implementation of a preconditioner we wish to use as a subsidiary preconditioner. If this is not the case, a re-implementation of an existing preconditioner with a matching DOF granularity would be required. This would lead to an immense overhead in the amount of code and would be against the main idea of the BPF, to aid development with seamless code re-use.

### 5.2.2 Issues with the Block Extraction Process

In the initial version of the BPF, a call to the \texttt{get\_block(...)} function extracts the required block from the naturally ordered Jacobian. However, there are cases, such as the preconditioner introduced in this thesis (see (5.2.3)), which require certain blocks to be different from those in the Jacobian matrix (such as the augmented blocks \( \hat{F}_{ij} \)). However, the use of the \texttt{NavierStokesSchurComplementPreconditioner} will assume the original Navier-Stokes Jacobian (the \( 5 \times 5 \) block in (5.2.2)), rather than the augmented Navier-Stokes block (the \( 5 \times 5 \) block in (5.2.3)).

Thus, the initial version of the BPF would not allow the re-use of the existing version of the LSC preconditioner involving block modification. The same would be the case for any subsidiary preconditioner used within the LSC preconditioner (such as the BUT preconditioner (5.2.6)).

Thus, we would need to re-implement all subsidiary preconditioners used within the augmentation preconditioner. This will lead to repeated and possibly inefficient code, increase the potential for errors and will ultimately slow down the development of preconditioners within the library.
These drawbacks of the BPF extend to other general and problem specific preconditioners. Any mismatch in the DOF type granularity (between master and subsidiary DOF types), or the modification of certain preconditioner blocks will require the re-implementation or modification of all existing preconditioners in a particular hierarchy. This issue had motivated a significant upgrade of the initial version of the BPF which was performed as a part of this project.

5.3 Newly Implemented Functionalities

In this section we describe the two newly implemented functionalities designed to address the previously discussed deficiencies in the BPF. In Section 5.3.1 we provide a solution to the block extraction issue and discuss the overhead associated with this new functionality. In Section 5.3.3 we introduce the solution to the DOF type mismatch problem between the DOF types of master and subsidiary preconditioners.

5.3.1 DOF Block Replacement Functionality

A potential solution to the issue described in Section 5.2.2 is that when a preconditioner constructs a modified block matrix (e.g. the blocks $\hat{F}_{ij}$ in (5.2.3)), it stores this block as a replacement block. This block is used in all subsequent calls to get_block(i,j) in the preconditioning hierarchy, i.e. when a subsidiary preconditioner calls get_block(i,j), a replacement block is used instead of extracting from the original Jacobian. For reasons which will become clear later, only DOF blocks (blocks consisting of a single DOF type) can be replaced by their modification. We outline this procedure in Algorithm 5.1.

In the newly implemented get_block(...) function, we traverse up the preconditioner hierarchy tree until the first replacement DOF block is found. If a replacement DOF block is not found and the root (top level preconditioner) is reached, a block is extracted from the Jacobian. This implies that only the preconditioner at the root of the hierarchy performs the block extraction. The additional functionality required here is the concatenation of sparse distributed block matrices with no inter-processor communication, this is discussed Section 5.3.2. By contrast, in the initial version of the BPF, when a preconditioner at a certain hierarchy level calls get_block(i,j), an entire block $(i,j)$ was extracted from the original Jacobian matrix with no reference to other levels in the hierarchy. The drawback of this approach is that any modification
Algorithm 5.1: The get_block(...) function for the DOF block replacement functionality. The set $B$ consists the ordered pairs $(k,l)$ of DOF types associated with the block $(i,j)$. This is defined by the Cartesian product $D_i \times D_j = \{(k,l) \mid k \in D_i \text{ and } l \in D_j\}$.

\begin{verbatim}
input : Two integers $i$ and $j$
output: Block $(i, j)$
1 Get the set of DOF types $D_i$ associated block type $i$;
2 Get the set of DOF types $D_j$ associated block type $j$;
3 Compute $B = D_i \times D_j$;
4 foreach $(k, l) \in B$ do
5 if Replacement DOF block $(k, l)$ exists then
6 Use this block for concatenation.
7 else if master preconditioner exists then
8 Get DOF block $(k, l)$ from master preconditioner
9 else
10 Extract DOF block $(k, l)$ from the Jacobian
11 Concatenate all DOF blocks and return;
\end{verbatim}

to block matrices introduced at a higher hierarchy level is lost.

A replacement block is stored at the current hierarchy level by invoking the Block-Preconditioner::set_replacement_dof_block(...) function, which takes as input arguments two unsigned indicating the position of the DOF block (in the DOF type ordering) and a pointer to the replacement DOF block. A preconditioner which sets a replacement block also performs the garbage collection when the block is no longer required. In the sequel, we demonstrate this new functionality of the BPF using a generic block system where the Jacobian $J$ has six DOF types which is associated with the unknowns $a_i$, $i = 1, 2, \ldots, 6$:

\[
\begin{array}{cccccc}
\text{DOF type} & 0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

and the blocked Jacobian has the following DOF block structure:

\[
J = \begin{pmatrix}
J_{11} & J_{12} & J_{13} & J_{14} & J_{15} & J_{16} \\
J_{21} & J_{22} & J_{23} & J_{24} & J_{25} & J_{26} \\
J_{31} & J_{32} & J_{33} & J_{34} & J_{35} & J_{36} \\
J_{41} & J_{42} & J_{43} & J_{44} & J_{45} & J_{46} \\
J_{51} & J_{52} & J_{53} & J_{54} & J_{55} & J_{56} \\
J_{61} & J_{62} & J_{63} & J_{64} & J_{65} & J_{66}
\end{pmatrix}
\]

(5.3.2)

Example 5.3.1. Replacement of DOF blocks. In this example we demonstrate how to
5.3. NEWLY IMPLEMENTED FUNCTIONALITIES

replace DOF blocks and reveal the problem with replacement of the compound blocks. Consider a $5 \times 5$ block preconditioner for (5.3.2) of the form:

$$P_1 = \begin{bmatrix}
J_{11} & & & & \\
K_{22} & K_{23} & & & \\
K_{32} & K_{33} & & & \\
& & K_{44} & J_{45} & J_{46} \\
& & J_{54} & K_{55} & J_{56} \\
& & J_{64} & J_{65} & J_{66}
\end{bmatrix}, \tag{5.3.3}$$

where $K_{44}$ and $K_{55}$ are replacement DOF blocks for $J_{44}$ and $J_{55}$, respectively. Denote

$$\tilde{K}_{22} = \begin{bmatrix}
K_{22} & K_{23} \\
K_{32} & K_{33}
\end{bmatrix}, \tag{5.3.4}$$

then $\tilde{K}_{22}$ is a replacement compound block for

$$\tilde{J}_{22} = \begin{bmatrix}
J_{22} & J_{23} \\
J_{32} & J_{33}
\end{bmatrix}. \tag{5.3.5}$$

To construct $P_1$, we must first set up the block structure in the setup(...) function we pass the `dof_to_block_map = [0,1,1,2,3,4]` to the function block_setup(...). At this stage, no replacement blocks have been set. Therefore, calls to `get_block(2,2)` and `get_block(3,3)` will extract the blocks $J_{44}$ and $J_{55}$ from the Jacobian matrix. The call to `get_block(1,1)` will extract the DOF blocks $J_{22}$, $J_{23}$, $J_{32}$ and $J_{33}$ from the Jacobian matrix, followed by concatenation of the blocks into $\tilde{J}_{22}$. We create the matrices $K_{44}$ and $K_{55}$, and set them as replacement DOF blocks with the following instructions:

```cpp
this->set_replacement_dof_block(3,3,K44_pt); // DOF type 3
this->set_replacement_dof_block(4,4,K55_pt); // DOF type 4
```

where $K44_{pt}$ and $K55_{pt}$ are pointers to the matrices $K_{44}$ and $K_{55}$, respectively. There is no functionality to set replacement compound blocks. Replacement matrices set by `set_replacement_dof_block(...)` are assumed to be DOF blocks. To demonstrate why this is the case, for now, assume that $\tilde{K}_{22}$ can be set as a replacement block and when $P_1$ calls `get_block(1,1)`, instead of extracting blocks from the Jacobian matrix,

\footnote{For simplicity, we omit the calls `set_nmesh(...)` and `set_mesh(...), and assume that they are called appropriately.}
If we want to use a block diagonal sub-preconditioner \( P_2 \) to approximate \( P_1 \), then we turn \( P_2 \) into a subsidiary preconditioner with the following function calls:

\[
P_2.p\text{t}\to\text{turn\_into\_subsidiary\_block\_preconditioner}(\text{this}, \text{dof\_map});
P_2.p\text{t}\to\text{setup}(\text{matrix\_pt}());
\]

Here, the vector \( \text{dof\_map} = [0,1,2,3,4,5] \). Assume further that \( P_2 \) replaces its DOF block \((3,3)\) with \( L_{44} \):

\[
\text{this} \to \text{set\_replacement\_dof\_block}(3,3,L_{44}.\text{pt});
\]

then we would expect \( P_2 = \text{blockdiag}(J_{11}, K_{22}, K_{33}, L_{44}, K_{55}, J_{66}) \).

The function call \text{get}\_block\((i,j)\) will return any associated replaced blocks (for example, \( L_{44} \)). If \( P_2 \) cannot find a replaced block, then it will request this block from its master preconditioner, \( P_1 \). The call \text{get}\_block\((4,4)\) from \( P_2 \) will cause \( P_1 \) to make a deep copy of \( K_{55} \) and return it to \( P_2 \). If \( P_1 \) cannot find a replaced block, then it will extracted it from the Jacobian matrix, i.e. \( J_{55} \) (if \( K_{55} \) does not exist).

The call \text{get}\_block\((1,1)\) in \( P_2 \) should either return the block \( J_{22} \) or its most recent replacement in the preconditioning hierarchy. This would be the block \( K_{22} \) set by \( P_1 \). However, \( P_1 \) has replaced the entire compound block \((\hat{J}_{22} \text{ by } \hat{K}_{22})\) and cannot extract \( K_{22} \) from it. This problem occurs when a subsidiary preconditioner requests a block of finer granularity than a replaced block higher up in the hierarchy. To alleviate this problem of extracting replacement blocks from compound blocks, only blocks of the finest granularity can be replaced, i.e. the DOF blocks.

To use the replacement block functionality for compound blocks (rather than just DOF blocks) we would require additional functionality for the BPF to either decompose a compound block into its DOF blocks and then use the newly implemented replacement DOF block functionality, or return the entries from the modified (compound) block to a ‘naturally ordered’ matrix which is used at the current and all lower levels of the preconditioner hierarchy. The second option requires a method to detect if an entry in a block matrix should be extracted from the original Jacobian or the new naturally ordered matrix. This is not a trivial task to perform and may degrade the performance of the BPF. There is currently no preconditioner in the library that requires the replacement of compound blocks.

If \texttt{oomph-lib} is compiled with the debug flag \texttt{paranoia}, then the BPF will ensure that the dimensions of the replacement block matches that of the associated DOF block. Therefore, all replacement blocks are assumed to be of DOF block granularity or a runtime error will occur. The example presented here will never arise in an
implementation (i.e. the BPF will not allow the replacement of the compound block $J_{22}$) but it highlights a restriction of the new functionality.

Thus, at each level of the preconditioning hierarchy the function call `get_block(i, j)` returns either the most recent update of block $(i, j)$ associated with that level, or looks further up the preconditioning hierarchy for any updates of that block at higher levels of the preconditioning hierarchy. Ultimately, if the topmost level preconditioner is reached and no replacement block is found, then the block $J_{ij}$ from the original Jacobian matrix is extracted and returned to the sub-preconditioner.

### 5.3.2 Block Concatenation and Splitting without Communication

In the initial version of the BPF, the `get_block(...)` function extracts an entire block from the original naturally ordered Jacobian. Asynchronous and non-blocking communication is employed to uniformly distribute the matrix rows among the processors. This process is facilitated by lookup schemes generated in `block_setup(...)`. One way to ensure backward compatibility with existing implementations is to utilise the initial block extraction routines. The challenge now is to implement an algorithm that scales well in parallel with small associated overhead. In this context, we focus on communication avoiding algorithms and implementations. The concatenation of the DOF blocks into ‘compound’ blocks is implemented without communication (i.e. the data on a particular processor stay local). This naturally leads to a block row permutation in the resulting matrix. To ensure the correctness of operations such as a scalar addition to the diagonal entries, the block columns are also permuted, following the same permutation pattern as the rows. Figure 5.3 illustrates a matrix consisting of $5 \times 5$ DOF blocks, the partition lines denote the $2 \times 2$ block partitioning. For a comparison of the concatenation with and without communication see Figure 5.4. The column permutation for non-square matrices is explained in Figure 5.5.

The block column permutation implies the matching block vector permutation. Recall from Section 5.1 that application of a preconditioner is implemented in the function `preconditioner_solve(...)`. Within this function we use `get_block_vector(...)` and `return_block_vector(...)` to extract and return a block vector from and to a global vector. The `get_block_vector(...)` function of the initial BPF implementation extracts an entire block vector from a global vector. Similarly, the `return_block_vector(...)` permutes the entries of a block vector back to a global
CHAPTER 5. PARALLEL BLOCK PRECONDITIONING FRAMEWORK

**Figure 5.3:** Illustration of $5 \times 5$ DOF blocks distributed on 3 processors. The partition lines represents a $2 \times 2$ block type partitioning. The block rows and columns are labelled in order to determine the data location after concatenation depicted in Figures 5.4 and 5.5.

**Figure 5.4:** Concatenation of the upper left $3 \times 3$ DOF blocks from Figure 5.3 without communication (left) and with communication (right). Without communication, the concatenation process naturally leads to a block row permutation of the matrix, and the columns are permuted according to the row permutation.

**Figure 5.5:** Concatenation of the DOF blocks from Figure 5.3 without communication for the block $(0,1)$ (left) and the block $(1,0)$ (right).
vector, these two procedures require inter-processor communication. The \texttt{get\_block\_vector(...)} function has been modified by first extracting all the DOF vectors, followed by a newly implemented concatenation procedure as depicted in Figure 5.6 (left to right). The \texttt{return\_block\_vector(...)} function follows the reverse process as shown in Figure 5.6 (right to left). This ensures that block vectors follow the same column permutation as the block matrices from \texttt{get\_block(...)}.

![Figure 5.6](image)

\textbf{Figure 5.6:} Left to right: Procedure for \texttt{get\_block\_vector(...)}. Firstly the DOF vectors are extracted from the global vector, this process requires inter-processor communication, then the DOF vectors are concatenated without communication to form block vector 1. Right to left: Procedure for \texttt{return\_block\_vector(...)}. The block vector is split into its corresponding DOF vectors without communication. Then the DOF vectors are returned to the global vector using inter-processor communication.

\section{5.3.3 DOF Type Coarsening Functionality}

The information about the DOF types is passed from the master to the subsidiary preconditioner via the \texttt{dof\_map} vector. This is an injective map between the subsidiary and master DOF types. Problem specific preconditioners are implemented with an assumed DOF type ordering. In Section 5.2.1 we have shown that if there is a discrepancy between the granularity of the DOF types assumed by the subsidiary preconditioner and \texttt{dof\_map}, then the existing implementations needs to be modified to map the additional DOF types in \texttt{dof\_map} to the block structure of the subsidiary preconditioner. In this section we describe a newly introduced functionality which allows
existing implementations of preconditioners to operate on different DOF granularities.

Consider again the Jacobian \((5.3.2)\) with six DOF types and the system is preconditioned with \(P_1\) as in \((5.3.3)\). Suppose that we want to approximate the bottom right \(3 \times 3\) subsidiary system with a block diagonal preconditioner \(P_3\) which assumes only two DOF types:

\[
P_3 = \begin{bmatrix} C_{11} & \cdot & \cdot \\ \cdot & C_{22} & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}.
\]

That is, \(P_3\) calls \texttt{block\_setup(...)} with a vector of size two, \((\text{dof\_to\_block\_map} = [0,1])\), mapping each DOF type to a block type, therefore each of the blocks \(C_{ii}\) \((i = 1, 2)\) are DOF blocks. However, \(P_1\) requires that \(P_3\) operate on three of its DOF types. The new version of the BPF allow master preconditioners to merge (coarsen) the DOF types passed to subsidiary preconditioners. If \(P_3\) is turned into a subsidiary preconditioner with the \text{dof\_map}=[3,4,5], \(P_1\) needs to pass additional information detailing which DOF types should be treated as single DOF type.

Note that only the DOF types passed to \(P_3\) are merged. To facilitate the merger of DOF types in \(P_3\) defined by \((5.3.6)\), we need to define the vector \texttt{doctype\_coarsen\_map} of size 2 with elements being vectors that define which additional DOF types in \(P_3\) are merged. For example, \text{doctype\_coarsen\_map}=[[[0,1],[2]] leads to the following DOF structure in \(P_3\):

\[
P_3 = \begin{bmatrix} K_{44} & J_{45} & \cdot \\ J_{45} & K_{55} & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}.\]

Internally, \(P_3\) still operates on 3 DOF types. The \texttt{doctype\_coarsen\_map} provides the look up scheme between \(P_3\)’s \textit{local} DOF types and the \textit{internal} DOF types. The local DOF types mimic the assumed DOF type structure the preconditioner was originally implemented with. \(P_3\) has two local DOF types and three internal DOF types. In \((5.3.7)\) the input argument to \(P_3\)’s \texttt{block\_setup(...) function is still \text{dof\_to\_block\_map} = [0,1]. There is a discrepancy between the granularity of \text{dof\_to\_block\_map} vector in \(P_3\) and the master preconditioner’s DOF types (the vector \text{dof\_map}). This was not allowed in the initial version of the BPF. The additional vector \texttt{doctype\_coarsen\_map} enables the BPF to internally create a vector that matches both the granularity of the master DOF types in \text{dof\_map} and the block structure of the subsidiary preconditioner. This new vector, referred to as \texttt{internal\_dof\_to\_block\_map},
is generated in the `block_setup(...)` function of \( P_3 \). It consists of the block types defined by `dof_to_block_map` (provided by the existing implementation of \( P_3 \)) and uses `doftype_coarsen_map` (provided by \( P_1 \)) to fill in the additional DOF types to match the granularity of `dof_map`. For example, in \([5.3.7]\) the `internal_dof_to_block_map` = \([0,0,1]\).

Thus, the *master preconditioner* have coarsened three subsidiary DOF types into two subsidiary DOF types to ensure that the DOF types required by the subsidiary preconditioner are satisfied. The three internal DOF types of \( P_3 \) are hidden from the user, in this construction, the user sees only the two local DOF types. Further modifications were made throughout the BPF to ensure that, for all intent and practical purposes, the local DOF types are treated by the subsidiary preconditioner as the DOF types as defined in Section 5.1 and the internal DOF types are known only internally and not visible to the user. For example, \( P_3 \) is restricted to have at most two block types, and calls `ndof_types()` will return the value 2. Thus, block matrices \( C_{ii} \) in \([5.3.6]\) are DOF blocks for \( P_3 \), rather than compound blocks. The new DOF type coarsening functionality requires further modification to the `get_block(...)` function discussed in Section 5.3.1 see Algorithm 5.2.

Algorithm 5.2: The `get_block(...)` function for both DOF block replacement and DOF type coarsening functionalities. The Cartesian products of two sets is used to generate the association between blocks and DOF blocks, and between DOF blocks and internal DOF blocks.

```
input : Two integers \( i \) and \( j \)
output: Block \((i,j)\)
1 Get local DOF types \( D_i \) associated block type \( B_i \);
2 Get local DOF types \( D_j \) associated block type \( B_j \);
3 Compute \( D_{ij} = D_i \times D_j \);
4 foreach \((k,l)\) \( \in D_{ij} \) do
5   if Replacement DOF block \((k,l)\) exists then
6     Use this block for concatenation.
7 else if master preconditioner exists then
8    Get internal DOF types \( I_i \) associated with DOF type \( k \);
9    Get internal DOF types \( I_j \) associated with DOF type \( l \);
10   Compute \( I_{ij} = I_i \times I_j \);
11   foreach \((m,n)\) \( \in I_{ij} \) do
12     Get master preconditioner’s (local) DOF block \((m,n)\);
13 else
14   Extract DOF block \((k,l)\) from the Jacobian
15 Concatenate all local and internal DOF blocks;
```

The main difference from the `get_block(...)` procedure described in Algorithm 5.1 is the lookup between local and internal DOF types in lines 8 – 12 of Algorithm 5.2.
This is because we no longer have an injective map between the subsidiary and master DOF types. In Example 5.3.1 we have shown that compound blocks cannot be replaced because a subsidiary preconditioner may need access to a block of finer granularity than the replaced block. If a preconditioner replaces a local DOF block, this block may not be the most fine grained block (due to the new DOF type coarsening functionality). However, we note that a subsidiary internal DOF type have the same granularity as its master’s local DOF types. Due to this relation, when constructing a block at a certain level in the preconditioner hierarchy, the local DOF blocks have sufficient granularity to construct all conceivable blocks at the current level and all lower levels of the hierarchy. Therefore, the local DOF blocks can be replaced. The relationship of the granularity of the internal and local DOF types at different levels of the preconditioner hierarchy is summarised in Figure 5.7.

**Figure 5.7:** A generic relationship between internal DOF types \( \{I^\alpha_i\} \), (local) DOF types \( \{D^\alpha_i\} \) and block types \( \{B^\alpha_i\} \) for preconditioners \( P_\alpha \) (\( \alpha = 1, 2 \)). Block types are formed from local DOF types (defined by `dof_to_block_map`), local DOF types consist of one or more internal DOF types (defined by `doftype_coarsen_map`).

Finally, we demonstrate that by using both the newly integrated functionalities of the DOF block replacement and the DOF type coarsening, we can efficiently and rapidly develop the augmentation preconditioner described in (5.2.3) using the existing LSC preconditioner for the Navier-Stokes subsystem and using a BUT approximation for \( F \) block, as defined in (5.1.11). Here, we focus only on the Navier-Stokes subsystem. We have seen that with the initial version of the BPF, using a general BUT preconditioner as a subsidiary preconditioner for the \( F \) block leads to the form (5.2.7) rather than (5.2.6) since general preconditioners assume DOF blocks.

The block structure of the augmentation preconditioner is defined by (5.2.2). Suppose we have created the modified (augmented) blocks \( \hat{F}_{11} \), \( \hat{F}_{12} \), \( \hat{F}_{21} \), and \( \hat{F}_{22} \). Using the DOF block replacement functionality, we set them as replacement block as shown.
The new BPF ensures that at the current level and all lower levels of the hierarchy, these replacement blocks are used to construct block matrices. Next, we coarsen the 5 DOF types shown in (5.2.4) to the 3 DOF types assumed by the implementation of the LSC preconditioner with the instructions:

```c
Navier_stokes_block_preconditioner_pt
->turn_into_subsidiary_block_preconditioner(
  this, dof_map, doftype_coarsen_map);
```

Where `dof_map = [0,1,2,3,4]`, which corresponds to the associated unknowns $u_1$, $u_2$, $p$, $u_1'$, and $u_2'$ as shown in (5.2.1). In the `doftype_coarsen_map` we coarsen the additional DOF types to those assumed by the LSC preconditioner $(u_1, u_2$ and $p$), thus `doftype_coarsen_map = [[0,3],[1,4],[2]]`. We instruct the LSC preconditioner to use a BUT approximation for the $F$ block with the following function calls:

```c
Navier_stokes_block_preconditioner_pt
->set_f preconditioner (Block_triangular_preconditioner_pt);
```

Where the pointer `Block_triangular_preconditioner_pt` holds an instance of `BlockTriangularPreconditioner`. Finally, we call the LSC preconditioner’s `setup(...)` function:

```c
Navier_stokes_block_preconditioner_pt->setup(matrix_pt());
```

The augmentation preconditioner has coarsened the DOF types passed to the LSC preconditioner. This feeds down to the BUT preconditioner, thus the BUT has two (local) DOF types. The DOF block replacement functionality guarantees that the modified blocks $\hat{F}_{ij}$ from the augmentation preconditioner are used, thus the BUT approximation to the $F$ block is

\[
\begin{pmatrix}
\tilde{F}_{11} & \tilde{F}_{12} \\
\tilde{F}_{12} & \tilde{F}_{22}
\end{pmatrix}
\begin{pmatrix}
\hat{F}_{11} & \hat{F}_{12} \\
\hat{F}_{12} & \hat{F}_{22}
\end{pmatrix}
= \begin{pmatrix}
F_{11} & F_{12} & F_{11} & F_{12} \\
F_{11} & F_{11} & F_{12} & F_{12} \\
F_{12} & F_{12} & F_{22} & F_{22} \\
F_{12} & F_{12} & F_{22} & F_{22}
\end{pmatrix},
\]

which is the desired approximation shown in (5.2.6).

## 5.4 Summary and Discussion

In this chapter we presented the functionality of the BPF. We first reviewed the initial version of the software as developed in [180]. Then we discussed additional functionality developed during the course of this project, which was necessary for the efficient
implementation of the augmentation preconditioner for the constrained Navier-Stokes problem.

In particular, to enable re-use of existing implementation of block preconditioners without modification, two additional functionalities were required: replacement of some DOF blocks in a manner where subsidiary preconditioners use the most recent update for a particular block; and coarsening of DOF types so that subsidiary preconditioners can operate on more DOF types than anticipated when the subsidiary preconditioner was implemented.

Both functionalities have been carefully integrated within the initial version of the BPF such that existing implementations of block preconditioners (both general and problem specific) do not have to be modified.

All parallel programming (for example, MPI calls and dealing with distributed matrices and vectors) is handled by the BlockPreconditioner base class, therefore, the same serial implementation of a block preconditioner is automatically parallelised. Thus, the main challenge in implementing the new features is to ensure the efficient implementation of the matrix and vector concatenation, and splitting of vector routines with good parallel scalability. This is achieved with communication avoiding algorithms. The routines are then carefully integrated with the existing block/vector extraction and vector return functionalities of the initial BPF implementation. The parallel scaling of the BPF is investigated in [Chapter 7](#).

Careful bookkeeping of the internal and local DOF types is also required. In the initial version of the BPF, there is only one set of DOF types, which is passed from the master to the subsidiary preconditioner via an injective map and used throughout the BPF, for example, to construct block types. Thus modification of the whole framework was required so that the two sets of DOF types (internal and local) are coherent and to ensure that the software is backwards compatible.

Throughout this project, many small optimisations have been made to the initial version of the BPF which has improved its overall efficiency. The examples include utilising `std::copy` and `std::sort` from the C++ standard library, instead of own implementations of these procedures. Many instances where containers where initialised via loops were also changed to use proper initialisation techniques.
5.4. SUMMARY AND DISCUSSION

5.4.1 Current Limitations of the BPF

We discuss some limitations of the current implementation of the BPF, giving some future directions for the development. In the current implementation, a block preconditioner can only replace a DOF block. The suggested method of replacing a compound block is to decompose it into corresponding DOF blocks, then replacing these using the newly implemented functionality. This is not implemented since we do not currently require this feature.

At each level of the block preconditioning hierarchy, if a DOF block replacement has been set, it can be deleted only by the preconditioner which sets it. The replacement block must remain in memory until the subsidiary preconditioners which requires the block finishes its \texttt{setup(...)} function. If these replacement DOF blocks are deleted beforehand, then the DOF blocks are extracted from the original Jacobian matrix. In the most extreme case, where each preconditioner in the hierarchy replaces all its DOF blocks, the memory usage would become an issue. In practice, only a small number of DOF blocks are ever replaced. In the case of the augmentation preconditioner these replaced blocks are typically much smaller than other blocks in the system.
Chapter 6

Numerical Evaluation

In this chapter we present numerical evaluation of the performance (in terms of execution times) and efficiency (the convergence rate, in terms of the average number of iterations required for a linear solver to converge) of the augmentation preconditioner when applied to the Navier-Stokes (NS) system with weakly imposed boundary conditions \([4.2.1]\) described in Section 4.2. All numerical simulations are implemented in \texttt{oomph-lib} \[122\] using the newly implemented functionalities (DOF block replacement and DOF type coarsening) of the Block Preconditioning Framework described in Section 5.3.

This chapter contains three main sections. Firstly, we discuss three different versions of the augmentation preconditioner designed with the aim of getting an optimal solver, i.e. a solver with the execution time that scales linearly as a function of the discrete problem size. Then we evaluate their performance on a range of steady-state two-dimensional problems in Section 6.2 and unsteady three-dimensional problems in Section 6.3. Both simple \([2.2.4]\) and stress divergence \([2.2.3]\) forms of the viscous terms are considered. Spatial discretisation of the problem is performed by quadrilateral/brick uniform grids using Taylor-Hood \((Q_2 - Q_1)\) elements, as described in Section 2.4.1.

The Lagrange multipliers terms are discretised using matching boundary grids on \(\partial \Omega_\lambda\) with equal order approximation as for the velocity space. The face element framework is implemented using \texttt{oomph-lib’s} \texttt{FaceElements} \[115, 111\], this process is described in Section 4.3.

For steady-state simulations the resulting system of nonlinear equations is solved by Newton’s method with the absolute tolerance of \(\varepsilon_N = 10^{-6}\) and with an initial solution guess of zero. For unsteady problems we perform an impulsive start from
rest [99, pp. 794–795]. The initial solution guess at initial time is set to zero, and at subsequent time steps we use the solution computed at previous time steps as the initial guess. The adaptive BDF-2 time stepping method [114, 245, p. 67, 99, p. 715] is used to solve the resulting system of differential algebraic equations (DAE) with time tolerance constant of \( \varepsilon_T = 10^{-4} \). The systems of linear equations are then solved by the right-preconditioned GMRES [215, p. 270] with the newly introduced augmentation preconditioner. We employ Trilinos [132] AztecOO [242] implementation of GMRES with a relative stopping tolerance of \( \varepsilon_K = 10^{-6} \). We consider three versions of the augmentation preconditioner: Exact (\( P_E \)), exact LSC (\( P_{LSC} \)), and inexact LSC (\( \hat{P}_{LSC} \)), which are described in the next section.

All computations in this chapter are performed using the Computational Shared Facility (also known as Danzek or CSF), a High Performance Computing (HPC) cluster located at the University of Manchester [235]. We run the simulations on a single Westmere node with two Intel® Xeon® Processors X5650 (12M Cache, 2.66 GHz, 6.40 GT/s Intel® QPI) [141]. These have six cores per processor with 8GB of memory per core, for more details on the system configuration of the CSF see [237].

6.1 Efficient Implementation of the Augmentation Preconditioner

According to the analysis presented in Section 4.6, the ideal augmentation preconditioner \( P \) from (4.5.1) is spectrally close to the Jacobian in (4.3.11). To achieve targeted optimal implementation of the GMRES solver, we require an optimal implementation of the augmentation preconditioner, i.e. an implementation with the computational cost that scales linearly with the problem size. In this context, we look at the steps taken to apply the preconditioner. On application, the inverse of (4.5.1) is taken,

\[
P^{-1} = \begin{bmatrix} \tilde{F}_{ns} & \end{bmatrix}^{-1} = \begin{bmatrix} \tilde{F}_{ns}^{-1} \\ W^{-1} \end{bmatrix},
\]

(6.1.1)

with \( W = \frac{1}{\sigma} LL^T \in \mathbb{R}^{N_t \times N_t} \) and \( \tilde{F}_{ns} = F_{ns} + L^T W^{-1} L \in \mathbb{R}^{N_{ns} \times N_{ns}} \). Here, \( N_{ns} \) and \( N_t \) are the sizes of original Navier-Stokes system described by (2.3.38) and the size of the Lagrange multiplier constraint block as described by (4.3.13) respectively. Thus, the main challenge in obtaining an optimal version of (6.1.1) is to provide...
approximate, but spectrally equivalent, inverses to the diagonal blocks $\tilde{F}_{ns}$ and $W$, therefore achieving bounded and low iteration counts of the resulting iterative solver. A suitable approximation to $W$ can substantially lower the computational cost of inverting it, and make the perturbation block $L^T W^{-1} L$ sparse, thus computationally cheap to compute.

In Section 4.3 we have shown that $L$ consists of mass matrix-like constraint blocks. Based on Wathen [255] and the discussion in Section 4.5, we have shown that the square of a mass matrix, $M^2$, is spectrally equivalent to its diagonal coefficients. Then we have

$$W = \frac{1}{\sigma} L L^T = \frac{1}{\sigma} \sum_{i=1}^{d} M_i^2 \approx \frac{1}{\sigma} \sum_{i=1}^{d} \text{diag}(M_i^2) = \hat{W},$$

(6.1.2)

where the symbol $\approx$ denotes spectral equivalence, $d = 2$ or 3 is the spatial dimension. Since $\hat{W}$ is a diagonal matrix, its inverse $\hat{W}^{-1}$ can be computed in $O(N)$ time and space complexity. Then the perturbation block $L^T \hat{W}^{-1} L$ is much cheaper to form and apply than $L^T W^{-1} L$, since $W^{-1}$ is a dense matrix. Thus, we replace $W$ by $\hat{W}$, and (6.1.1) becomes

$$\hat{P}^{-1} = \begin{bmatrix} \hat{F}_{ns}^{-1} \\ \hat{W}^{-1} \end{bmatrix},$$

(6.1.3)

with $\hat{F}_{ns} = F_{ns} + L^T \hat{W}^{-1} L$. If the inversion of both diagonal blocks in (6.1.3) is computed exactly, then we have the exact preconditioner $P_E$, as described in Preconditioner 6.1.

---

**Preconditioner 6.1: $P_E$. The exact Augmentation Preconditioner**

**Preconditioner:**  Block diagonal preconditioner

**Navier-Stokes Preconditioner:**  Exact inversion of $\hat{F}_{ns}$

**W-block Preconditioner:**  Exact inversion of $\hat{W}$

---

We use SuperLU [60] for the inversion of the diagonal blocks in (6.1.3). The SuperLU solver is implemented as a preconditioner in the class SuperLUPreconditioner (i.e. it inherits from the Preconditioner base class and implements the functions Preconditioner::setup(...) and Preconditioner::preconditioner_solve(...)). The setup phase of the SuperLUPreconditioner performs an LU decomposition with partial pivoting, which can subsequently be re-used, while preconditioner.solve(...) performs forward and backward substitution.

To use SuperLUPreconditioner within the augmentation preconditioner, we simply call two functions: within the augmentation preconditioner’s setup(...) function
we call

\[ \text{W\_preconditioner\_pt->setup(w\_pt);} \]

where the variable \( \text{W\_preconditioner\_pt} \) points to an instance of the \text{SuperLUPreconditioner} and \( \text{w\_pt} \) points to the \( \hat{W} \) block. In the augmentation preconditioner's \text{preconditioner\_solve(...)} function, to apply the preconditioner operator to the vector \( r \) and return the vector \( z \), we call

\[ \text{W\_preconditioner\_pt->preconditioner\_solve(r,z);} \]

Due to the fact that \( \hat{W} \) is a diagonal matrix, its inversion is trivial. However, we still opt to use the \text{SuperLUPreconditioner} rather than invert the diagonal entries using our own code due to parallelisation considerations. When \text{oomph-lib} is compiled with MPI support and ran on multiple processors, both the above function calls are automatically parallelised, \text{SuperLU\_DIST} [163] is used (the version of \text{SuperLU} for distributed memory machines), and the matrices and vectors involved are also automatically distributed among the participating processors. The inversion of the \((2,2)\) block with own code (i.e. by looping through the diagonal entries of the \( \hat{W} \) block), would require parallel considerations of this procedure rather than using the existing parallel capability. In our experience, the computational overhead incurred when using \text{SuperLUPreconditioner} to invert the \((2,2)\) block is negligible, for example, the wall-clock time required to set up the LU decomposition of the \((2,2)\) block using \text{SuperLU} for a three-dimensional problem with \( N = 1,130,511 \) unknowns takes approximately 0.005 seconds.

Now we turn our attention to finding an efficient method of inverting the augmented NS block \( \hat{F}_{ns} \). We note that only a small proportion of the nonzero entries of \( F_{ns} \) are modified by adding the perturbation term \( L^T\hat{W}^{-1}L \), thus the sparsity of \( F_{ns} \) is maintained. To see this, notice that the sparsity pattern of \( L^T\hat{W}^{-1}L \) is the same as that of \( L^T L \), and the individual perturbation block matrices are \( M_i^T\hat{W}^{-1}M_j \) for \( i,j = 1,2[3], \) which in turn have the same sparsity pattern as \( M_iM_j \). Due to the asymptotic relations \( N_\ell = \mathcal{O}(N_u^{1/2}) \) in 2D and \( N_\ell = \mathcal{O}(N_u^{2/3}) \) in 3D (where \( N_u \) is the number of velocity degrees of freedom), it follows that \( N_\ell \ll N_u \) and a relatively small proportion of matrix rows in the momentum block \( F \) are affected by the perturbation. Furthermore, with the choice \( \sigma = \|F\|_\infty \) (see Section 4.6), we have that \( \sigma \) is constant in 2D and \( \sigma \to 0 \) in 3D. These facts provide a heuristic argument that only a

\footnote{This may cause problems for large systems. We may have to set a lower bound on \( \sigma \) in 3D, but all our three-dimensional simulations have not been affected by this.}
small proportion of entries in the NS Jacobian $F_{ns}$ are affected by a relatively small perturbation. We note that if more than one constraint is enforced (as is the case with parallel outflow in three dimensions), the addition of multiple blocks of the form $M_i^T \tilde{W}^{-1} M_j$ to the NS coefficient matrix does not significantly affect its sparsity either.

Based on the previous discussion, we will attempt to use the efficient preconditioners for the NS problem to approximate in a black-box manner the augmented NS block $\hat{\mathcal{F}}_{ns}$. This approach proved to be successful in the case of constrained pseudo-elastic sub-problems, considered as a part of the FSI preconditioner [180, p. 163], where coefficient matrix is symmetric positive definite. Based on this result, we conjecture that the existing (nearly)-optimal preconditioners for the Navier-Stokes equations (NSE) will retain their effectiveness when applied in a black-box manner to approximate the inverse of $\hat{\mathcal{F}}_{ns}$.

As stated in [68, p. 359], the advantage of the LSC preconditioner is that once the linear system is specified, all the ingredients needed for setting up the preconditioner are available. By contrast, the PCD preconditioner requires the assembly of the pressure convection-diffusion operator $F_p$ and the pressure Laplacian operator $A_p$, which may not be a trivial thing to do. In addition, in our case, it is not clear or straightforward how to incorporate the perturbation $L^T W^{-1} L$ into the matrix $F_p$. The AL preconditioner do not have these problems as it is (1,1) block Schur complement based, but it requires the assembly of the matrix $B^T M_p^{-1} B$, where $M_p$ is the pressure mass matrix. Furthermore, it requires a geometric multigrid solver specially constructed for the (1,1) block, see [29]. The modified AL preconditioner can be implemented with readily available AMG solvers, but has a free parameter $\gamma$, which (in 2D) is dependent on the discretisation parameter $h$ and is mildly dependent on the Reynolds number in 3D, see [30] for more detail.

Of the three NS preconditioners, the LSC preconditioner requires the least additional information to implement. Thus, it has the best ‘black-box’ characteristics, and this is why it is the method of choice for LBB-stable discretisations. In spirit of this, we use the LSC preconditioner to approximate the inverse of the block $\hat{\mathcal{F}}_{ns}$. The downside is that the augmented momentum block $\hat{\mathcal{F}} = F + \hat{\mathcal{L}}^T \tilde{W}^{-1} \hat{\mathcal{L}}$, where $\hat{\mathcal{L}}$ is a sub-matrix consisting of the blocks of $L$ which corresponds to the momentum block, participates in the Schur complement $B \hat{\mathcal{F}}^{-1} B^T$, and this may affect the effectiveness of its LSC approximation. This gives the second version of the augmentation preconditioner: the exact LSC ($\mathcal{P}_{LSC}$), described by Preconditioner 6.2.
Preconditioner 6.2: $\mathcal{P}_{LSC}$, The exact LSC Augmentation Preconditioner

- **Preconditioner**: Block diagonal
- **Navier-Stokes Preconditioner**: LSC
- **Momentum Preconditioner**: Exact inversion of $\hat{F}$
- **Pressure Poisson Operators**: Exact inversion of $(BM_u^{-1}B^T)$
- **W-block Preconditioner**: Exact inversion of $\hat{W}$

The main difference to $\mathcal{P}_E$ is that the $(1,1)$ block, $\hat{F}_{ns}$, is approximated by the exact LSC preconditioner. This means that the inverse of the LSC preconditioner is obtained by inverting exactly the augmentation momentum block $\hat{F}$ and the pressure Poisson matrices $BM_u^{-1}B^T$, where $M_u$ is the diagonal of the velocity mass matrix and the Schur complement is replaced by its LSC approximation \((3.4.25)\).

The inexact LSC version of the augmentation preconditioner, $\hat{\mathcal{P}}_{LSC}$, is obtained by replacing the exact inverses in $\mathcal{P}_{LSC}$ by multigrid approximations. For this, we use HYPRE BoomerAMG, a state-of-the-art algebraic multigrid solver \([131, 137]\). The exact specification of the multigrid parameters (the type of coarsening, the strength of dependence parameter, the number of V-cycles, and the number of pre/post smoothing sweeps) is obtained experimentally. The preconditioner ($\hat{\mathcal{P}}_{LSC}$) is described by Preconditioner 6.3. We note that since the $(2,2)$ block is a diagonal matrix, exact inversion remains optimal.

Preconditioner 6.3: $\hat{\mathcal{P}}_{LSC}$, The inexact LSC Augmentation Preconditioner

- **Preconditioner**: Block diagonal
- **Navier-Stokes Preconditioner**: LSC
- **Momentum Preconditioner**: Inexact inversion of $\hat{F}$ by AMG
- **Pressure Poisson Operators**: Inexact inversion of $BM_u^{-1}B^T$ by AMG
- **W-block Preconditioner**: Exact inversion of $\hat{W}$

6.2 Two-dimensional Case Studies

In this section we consider two steady-state problems in two spatial dimensions: the Poiseuille flow through a square domain rotated by an arbitrary angle, and the uniform radial flow through a quarter annulus.
6.2.1 Poiseuille Flow through a Unit Square Rotated by an Arbitrary Angle

We study the Poiseuille flow through a unit square domain rotated by an arbitrary angle $\alpha$ (see Figure 6.1). The domain $\Omega^{[\alpha]}$ is obtained by rotating the discrete points $(x_1, x_2)$ in the unit square $\Omega = [0, 1]^2$ by the following transformation:

$$R(\alpha) = \begin{bmatrix} \cos(\alpha) - \sin(\alpha) \\ \sin(\alpha) + \cos(\alpha) \end{bmatrix},$$  \hspace{1cm} (6.2.1)

where $\alpha$ is the angle of rotation.

**Example 6.2.1.** Poiseuille flow through a rotated unit square domain.

**Boundary conditions:** for convenience, we present the boundary conditions for the non-rotated unit square $\alpha = 0^\circ$. In order to obtain the boundary conditions for $\alpha \neq 0^\circ$, we only have to apply the rotation [6.2.1]. The flow is driven by imposing a parabolic velocity profile along the inflow boundary $\partial \Omega_I$. Along the characteristic boundary, $\partial \Omega_C$, the no-slip condition $u_i = 0, i = 1, 2$ is prescribed. We enforce ‘parallel outflow’ along the outlet $\partial \Omega_O \equiv \partial \Omega_{\lambda}$ using Lagrange multipliers. Thus, boundary conditions on each of the four segments of the boundary are:

- $u_1 = u_2 = 0$, at $x_1 \in [0, 1], x_2 = 0$ (characteristic (bottom) boundary $\partial \Omega_C$),
- $u_1 = u_2 = 0$, at $x_1 \in [0, 1], x_2 = 1$ (characteristic (top) boundary $\partial \Omega_C$),
- $u_1 = x_2(1-x_2), u_2 = 0$ at $x_1 = 0, x_2 \in [0, 1]$ (parabolic inflow boundary $\partial \Omega_I$),
- $u \cdot \hat{t} = 0$, at $x_1 = 1, x_2 \in [0, 1]$ ('parallel outflow' boundary $\partial \Omega_O$).

6.2.1.1 Numerical Verification of Eigenvalue Bounds

First we validate numerically the analytical bounds of eigenvalues $\mu$ of (4.6.8) and demonstrate numerically a tight clustering of eigenvalues $\nu$ for (4.6.2). We use the NSE with the simple form of the viscous term and Re = 0 for several discrete problem sizes.

From Table 6.1 we observe a tight clustering of $N_\ell$ real non-unit eigenvalues of (4.6.8) in the interval $[-1, -\frac{1}{2}]$ independently of the problem size as predicted by the analysis. The numerical bounds are actually considerably better than the prediction. The numerical eigenvalue bounds of (4.6.2) observed in Table 6.2 are not
6.2. TWO-DIMENSIONAL CASE STUDIES

Figure 6.1: Plot of the solution (the pressure is in colour and the flow stream lines) for the steady Poiseuille flow with Re = 100 in a unit square domain rotated by α = 30°.

as tightly clustered as for (4.6.8) but is still considerably better than the analytical bounds presented in Section 4.6.1.

<table>
<thead>
<tr>
<th>$N_u + N_f$</th>
<th>119</th>
<th>495</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_f$</td>
<td>7</td>
<td>15</td>
<td>31</td>
</tr>
<tr>
<td>$(\mu = 1)$</td>
<td>112</td>
<td>489</td>
<td>1984</td>
</tr>
<tr>
<td>$(\mu \neq 1)$</td>
<td>7</td>
<td>15</td>
<td>31</td>
</tr>
<tr>
<td>max($\mu \neq 1$)</td>
<td>-0.8588</td>
<td>-0.8586</td>
<td>-0.8586</td>
</tr>
<tr>
<td>min($\mu \neq 1$)</td>
<td>-0.9674</td>
<td>-0.9832</td>
<td>-0.9914</td>
</tr>
</tbody>
</table>

Table 6.1: Numerical eigenvalue bounds of (4.6.8) for Example 6.2.1 with exact preconditioning and Re = 0. For the non-unit eigenvalues we present the numerical bounds in the last two rows.

<table>
<thead>
<tr>
<th>$N_u + N_p + N_f$</th>
<th>144</th>
<th>576</th>
<th>2304</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_f$</td>
<td>7</td>
<td>15</td>
<td>31</td>
</tr>
<tr>
<td>$(\nu = 1)$</td>
<td>137</td>
<td>561</td>
<td>2273</td>
</tr>
<tr>
<td>$(\nu \neq 1)$</td>
<td>7</td>
<td>15</td>
<td>31</td>
</tr>
<tr>
<td>max($\nu \neq 1$)</td>
<td>-0.7873</td>
<td>-0.7928</td>
<td>-0.7938</td>
</tr>
<tr>
<td>min($\nu \neq 1$)</td>
<td>-0.9459</td>
<td>-0.9714</td>
<td>-0.9853</td>
</tr>
</tbody>
</table>

Table 6.2: Numerical eigenvalue bounds of (4.6.2) for Example 6.2.1 with exact preconditioning and Re = 0. For the non-unit eigenvalues we present the numerical bounds in the last two rows.

AMG Calibration

In order to obtain an efficient inexact implementation of the augmentation preconditioner $\hat{P}_{LSC}$, we need to calibrate the AMG solver. Application of the LSC preconditioner to approximate the inverse of the augmented NS matrix $\hat{F}_{ns}$ involves two solves
with the sparse pressure Poisson operator
\[ P = B\hat{M}_u^{-1}B^T, \]  
(6.2.2)

where \( \hat{M}_u \) is the diagonal of the velocity mass matrix, and one solve involving the augmented momentum block
\[ \hat{F} = F + \hat{L}^T\hat{W}^{-1}\hat{L}. \]  
(6.2.3)

The matrix \( P \) is spectrally equivalent to the (sparse) pressure Laplacian [68, p. 273]. The HYPRE BoomerAMG parameters for the Poisson problem have been studied by Muddle in [180, pp. 57–61] on various oomph-lib test problems. It was observed that the Ruge-Stüber coarsening scheme with a strength of dependence of \( \theta = 0.25 \) with \( 2\times V(1,1) \) cycles using Jacobi smoother with a damping factor of \( \omega = 2/3 \) gives optimal solver performance in 2D. The same AMG parameters were utilised to approximate the inverse of the pressure Poisson operator \( P \) in an inexact implementation of the LSC preconditioner in [180, pp. 90–93], where an efficient implementation of the LSC preconditioner was studied. A slight increase in iteration counts was observed in 2D leading to sub-optimal scaling in the execution times. For our two-dimensional test problems, we use the same AMG setting when with HYPRE BoomerAMG. We formalise these parameters in Preconditioner 6.4.

| Preconditioner 6.4: A classical AMG preconditioner for 2D Poisson problems. |
|------------------|------------------|
| **Preconditioner:** | BoomerAMG |
| **Coarsening:** | Ruge-Stüber, \( \theta = 0.25 \) |
| **Smoothing:** | Damped Jacobi, \( \omega = 2/3 \) |
| **Cycles:** | \( 2 \times V(1,1) \) |

Thus, we primarily focus on determining AMG parameters for the approximate inverse of the augmented momentum matrix \( \hat{F} \). The augmented momentum matrix is a \( d \times d \) block matrix, where \( d \) is the spatial dimension, with all \( d^2 \) blocks being nonzero for both forms of the viscous terms. We tried four different preconditioning strategies for the \( \hat{F} \) block: black-box application of AMG to the entire block, a block diagonal preconditioner, with each of the \( d \) diagonal blocks inverted approximately by AMG [68, p. 361], and both an upper block triangular and lower block triangular preconditioner with the same treatment of diagonal blocks as before. For this experiment, we consider the NSE with the simple form of the viscous term with \( \alpha = 30^\circ \), Re = 100 and \( \hat{P}_{LSC} \)
described by Preconditioner [6.3]. In the sequel we use SVT-NSE to abbreviate the NSE with Simple Viscous Term (SVT) and SDVT-NSE for the NSE with Stress Divergence Viscous Term (SDVT). All references to timing results are for the wall-clock time. The AMG parameters used for the augmented momentum block $\hat{F}$ are the same as those used in [180, p. 91]:

- **Momentum preconditioner:** Inexact inversion with BoomerAMG
  - **Coarsening:** Ruge-Stüben (strength of dependence $\theta = 0.25$)
  - **Smoothing:** Damped Jacobi ($\omega = 1/2$)
  - **Cycle:** $2 \times V(1,1)$

The average GMRES iteration counts (with an initial guess of zero and relative convergence tolerance of $\varepsilon_K = 10^{-6}$) are presented in Table 6.3 and the corresponding execution times are tabulated in Table 6.4.

<table>
<thead>
<tr>
<th>Prec</th>
<th>576</th>
<th>2304</th>
<th>9216</th>
<th>36864</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>BD</td>
<td>50.0</td>
<td>71.3</td>
<td>97.0</td>
<td>136.3</td>
<td>176.5</td>
</tr>
<tr>
<td>BUT</td>
<td>41.0</td>
<td>53.3</td>
<td>66.3</td>
<td>95.3</td>
<td>128.5</td>
</tr>
<tr>
<td>BLT</td>
<td>43.0</td>
<td>52.7</td>
<td>67.7</td>
<td>95.3</td>
<td>125.5</td>
</tr>
<tr>
<td>Entire</td>
<td>28.7</td>
<td>32.0</td>
<td>35.0</td>
<td>42.3</td>
<td>50.0</td>
</tr>
</tbody>
</table>

*Table 6.3: The average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver using different block approximations of $\hat{F}$ in the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and $Re = 100$. BD = block diagonal, BUT/BLT = block upper/lower triangular, Entire = the entire $\hat{F}$ block.*

<table>
<thead>
<tr>
<th>Prec</th>
<th>576</th>
<th>2304</th>
<th>9216</th>
<th>36864</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>BD</td>
<td>0.07</td>
<td>0.18</td>
<td>0.87</td>
<td>6.22</td>
<td>33.50</td>
</tr>
<tr>
<td>BUT</td>
<td>0.03</td>
<td>0.13</td>
<td>0.64</td>
<td>3.42</td>
<td>22.94</td>
</tr>
<tr>
<td>BLT</td>
<td>0.04</td>
<td>0.13</td>
<td>0.62</td>
<td>3.51</td>
<td>19.98</td>
</tr>
<tr>
<td>Entire</td>
<td>0.03</td>
<td>0.10</td>
<td>0.40</td>
<td>2.15</td>
<td>8.63</td>
</tr>
</tbody>
</table>

*Table 6.4: The execution times (s) taken by the GMRES+$\hat{P}_{LSC}$ solver using different block approximations of $\hat{F}$ in the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and $Re = 100$. BD = block diagonal, BUT/BLT = block upper/lower triangular, Entire = the entire $\hat{F}$ block.*

The execution times shown in Table 6.4 are the sum of the average time taken to set up the preconditioner (which includes operations such as extraction of blocks matrices, setting up the LU decomposition and setting up matrix vector products) and linear solver time in one Newton iteration. We refer to these times as the execution times. We see that the lowest iteration counts and shortest execution times are obtained when AMG is applied to the entire augmented momentum block. The number of Newton iterations required for convergence is the same in all four cases (between

---

2This does not include the time to assemble the Jacobian.
3 and 2 depending on the discretisation parameter). We note that for the standard NS problem (e.g., Oseen’s problem [68, p. 329]), the diagonal approximation of $F$ with AMG subsidiary solves yields an efficient iterative solver as observed in [180, pp. 91–92]. This is due to the natural block diagonal structure of the momentum block (with the individual blocks being the convection-diffusion matrices). The block diagonal and block upper triangular approximations to the standard NS momentum block with Newton linearisation was studied by Elman, Loghin and Wathen in [67]. They advocate the upper block triangular version of the preconditioner, but this strategy is clearly inferior to the full AMG treatment of the momentum block in our case. In our example, the considerably higher iteration counts for the block diagonal and upper/lower block triangular versions of the preconditioner are due to the presence of the off-diagonal perturbation blocks. Omitting these blocks clearly does not lead to a good approximation of the augmented momentum matrix $\hat{F}$. Thus for two-dimensional case studies we apply AMG to the entire augmented momentum block $\hat{F}$.

We next present a systematic parameter study to determine the optimal AMG settings for the approximate inverse of $\text{entire}$ augmented momentum block (6.2.3) using BoomerAMG for two-dimensional cases.

**Coarsening strategy:** For serial experiments, it is well known that the ‘classical’ Ruge-Stüber coarsening method is one of the most robust approaches which gives nearly optimal convergence rates for convection-dominated problems. Thus, we use it for all our two-dimensional test cases.

**Strength of dependence, $\theta$:** We vary the strength of dependence parameter between 0.25 and 0.75. Increasing $\theta$ will generally lead to a preconditioner with fewer coarse levels and the coarse level matrices will be progressively sparser. This means that an AMG preconditioner will lose its quality and robustness with increasing $\theta$, but will also be computationally cheaper to assemble and apply. We approximate the augmented momentum block $\hat{F}$ by $1 \times V(2, 2)$ cycles with Gauss-Seidel smoothing. Other choices of smoothers are the damped Jacobi method and Euclid ILU(0). We choose Gauss-Seidel for this experiment because the damped Jacobi method requires the tuning of the damping factor $\omega$ and Euclid ILU(0) is expensive to apply. Thus, the AMG parameters for the augmented momentum block $\hat{F}$ are as follows:

- **Momentum preconditioner:** Inexact inversion with BoomerAMG
  - Coarsening: Ruge-Stüber (variable strength of dependence parameter $\theta$)
  - Smoothing: Gauss-Seidel
  - Cycle: $1 \times V(2, 2)$
For the pressure Poisson operators we use the AMG settings defined in Preconditioner 6.4.

Here we show only the results for $Re = 100$, the same observations apply for $Re = 200$, see Appendix A.1.1. The average iteration counts taken for the GMRES solver to converge to a relative tolerance of $\varepsilon_K = 10^{-6}$ with an initial guess of zero for the SVT-NSE and SDVT-NSE are presented in Tables 6.5 and 6.6, respectively. The wall-clock times taken to set up the augmented preconditioner are given in tables Tables 6.7 and 6.8 and finally the wall-clock solution times are presented in Tables 6.9 and 6.10. With all the other elements of the solver kept constant, the difference in the preconditioner setup times from Tables 6.7 and 6.8 is due the differences in the coarsening of the momentum block.

<table>
<thead>
<tr>
<th>$\theta$</th>
<th>0.25</th>
<th>0.50</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>19.3</td>
<td>20.3</td>
<td>21.0</td>
</tr>
<tr>
<td></td>
<td>21.3</td>
<td>25.3</td>
<td>27.7</td>
</tr>
<tr>
<td></td>
<td>25.7</td>
<td>27.3</td>
<td>32.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>35.7</td>
<td>47.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>44.5</td>
<td>48.0</td>
</tr>
</tbody>
</table>

Table 6.5: The average number of iterations taken by the GMRES$+\hat{P}_{LSC}$ solver for varying strength of dependence parameter $\theta$ in the AMG approximation of the $\hat{F}$ block within the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and $Re = 100$.

<table>
<thead>
<tr>
<th>$\theta$</th>
<th>0.25</th>
<th>0.50</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>23.7</td>
<td>21.3</td>
<td>20.7</td>
</tr>
<tr>
<td></td>
<td>37.0</td>
<td>30.7</td>
<td>26.0</td>
</tr>
<tr>
<td></td>
<td>75.3</td>
<td>62.3</td>
<td>30.7</td>
</tr>
<tr>
<td></td>
<td>149</td>
<td>121.5</td>
<td>33.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31.5</td>
<td>46.0</td>
</tr>
</tbody>
</table>

Table 6.6: The average number of iterations taken by the GMRES$+\hat{P}_{LSC}$ solver for varying strength of dependence parameter $\theta$ in the AMG approximation of the $\hat{F}$ block within the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SDVT-NSE with $\alpha = 30^\circ$ and $Re = 100$.

<table>
<thead>
<tr>
<th>$\theta$</th>
<th>0.25</th>
<th>0.50</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.04</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>0.05</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>0.53</td>
<td>0.59</td>
<td>0.56</td>
</tr>
<tr>
<td></td>
<td>2.06</td>
<td>2.23</td>
<td>2.22</td>
</tr>
</tbody>
</table>

Table 6.7: The average preconditioner setup times (s) taken by $\hat{P}_{LSC}$ for varying strength of dependence parameter $\theta$ in the AMG approximation of the $\hat{F}$ block within the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and $Re = 100$.

From Table 6.5 we observe that the best convergence rates for the SVT-NSE are achieved with $\theta = 0.25$. The preconditioner set up times from Table 6.7 show that the
difference in coarsening times for increasing $\theta$ is negligible. However, it is observed from Table 6.9 that $\theta = 0.25$ yields the shortest linear solve times.

The experimental study for the SDVT-NSE (see Table 6.6) suggests that the coarsening with smaller values of the parameter $\theta$ does not lead to a convergent preconditioner. For $\theta \leq 0.5$ the GMRES solver does not converge within 300 iteration for problem size $N = 147456$. We observe that the preconditioner setup times, shown in Table 6.8 are highest for $\theta = 0.25$ and the lowest for $\theta = 0.75$. This is not surprising, as the increase in $\theta$ leads to a sparser AMG preconditioner. However, a lower strength of dependence will lead to better convergence rates in general, but at the expense of a higher setup time [131, p. 168]. This was not observed for the SVT-NSE case, where lower values of $\theta$ does not lead to an increase in setup times over higher $\theta$ values. The FEM discretisation of the SDVT-NSE presented in Section 2.3 gives rise to the block matrices $G_{ij}$ ($i, j = 1, 2, 3$) defined by (2.3.43) which are not
present in the discretisation of the SVT-NSE. Careful study of the coarsening procedure and the smoother convergence rates for the SVT and SDVT momentum blocks is required to reveal the reasons behind a poor AMG behaviour for $\theta \leq 0.5$ in the SDVT case. From Table 6.10 we see that the shortest solution times are obtained for $\theta = [0.67, 0.75]$. For this problem, we ran additional tests for $\theta = [0.66, 0.8]$ in increments of 0.001, the iteration counts and execution times (preconditioner setup time plus linear solution time) are depicted in Figure 6.2 and Figure 6.3 respectively. The sudden decrease in iteration counts occur between $\theta = 0.666$ and 0.667 for both $\theta = 0.66$ and 0.667 for both $\theta = 100$ and 200. The iteration counts decrease from 168 to 43, resulting in the reduction in execution times from 44 seconds to 10 seconds. Increasing $\theta$ from 0.667 towards 0.8 generally leads to slightly higher iteration counts and execution times. In conclusion, we adopt the strength of dependence parameter $\theta = 0.25$ for the SVT-NSE and $\theta = 0.668$ for the SDVT-NSE in 2D experiments.

**Smoothing method:** We compare the performance of three standard smoothers. Jacobi with no damping,\(^3\) Gauss-Seidel, and Euclid ILU(0). We tested these smoothers

\(^3\)Numerical evidence presented in Appendix A.1.2 show that $\omega = 1$ yields both the lowest number of iteration counts and shortest execution times.
for one and two $V(2,2)$ cycles while keeping all other parameters fixed (as given in Preconditioner 6.4). In this experiment the AMG parameters for the approximation of the augmented momentum block $\hat{F}$ are as follows:

- **Momentum preconditioner:** Inexact inversion by BoomerAMG
  - **Coarsening:** Ruge-Stüben ($\theta = 0.25/0.668$ for SVT-NSE and SDVT-NSE, respectively)
  - **Smoothing:** Jacobi, Gauss-Seidel, ILU(0)
  - **Cycle:** $1 \times V(2,2), 2 \times V(2,2)$

We consider both forms of the viscous term with Re = 100 and an angle of rotation $\alpha = 30^\circ$. The average iteration counts required for the GMRES solver to converge to a relative tolerance of $\varepsilon_K = 10^{-6}$ from an initial solution guess of zero is given in Tables 6.11 and 6.12 for the SVT-NSE and SDVT-NSE, respectively. The corresponding execution times are presented in Tables 6.13 and 6.14.

The log-log graphs corresponding to Tables 6.11 to 6.14 are given in Figures 6.4 to 6.7. For the Euclid ILU(0) smoother, we only tested one $V(2,2)$ cycle since ILU(0) is computationally expensive to perform. However, it is observed from Tables 6.11 and 6.12 that even with one $V(2,2)$ cycle, the ILU(0) smoother achieves convergence within fewer iterations when compared to both Jacobi and Gauss-Seidel with $2 \times V(2,2)$
cycles. Predictably, performing $2 \times V(2, 2)$ cycles yield lower iteration counts than their single V-cycle counterparts, however the $2 \times V(2, 2)$ cycles have led to higher solution times.

From Figures 6.4 and 6.5 we see that all the smoothers exhibit similar asymptotic convergence rates. In Figure 6.4 we plot a line with a gradient of 0.1972, obtained from the line of best fit based the last five data points for Gauss-Seidel smoothing with $1 \times V(2, 2)$ cycles. This suggests that the iteration counts for the considered AMG setting grows as $O(N^{0.2})$, where $N$ is the size of the linear system in the range of the problem size we consider. However, looking over a different interval for $N$ it does look like the iteration counts may grow faster than $O(N^{0.2})$, if we consider only the last two data points in Figure 6.4 for all smoothing methods and cycles, they seem to be drifting towards and above the line $y = 0.1972x + 1.3976$. We were unable to consider larger problem sizes due to memory restrictions. In Figure 6.5, the line with a gradient of 0.2618 comes from the best fit based on the last five data points for the Gauss-Seidel $1 \times V(2, 2)$ cycles.

In Figure 6.6 we have plotted a line with a gradient of 1.0769, this comes from the line of best fit based on all the data points obtained from Gauss-Seidel smoothing with $1 \times V(2, 2)$ cycles. We have also plotted a line with a gradient of 1 for comparison. We have done the same for Figure 6.7 for the SDVT case. We see that for the SVT-NSE, all smoothers exhibit nearly optimal execution times (the execution times scale as $O(N^{1.07})$). From Figure 6.7 we see that asymptotic behaviour of the execution times for the SDVT case are only slightly worse than what was observed SVT ($O(N^{1.12})$).

For both forms of the viscous term, the approximation of the $\hat{F}$ block with $1 \times V(2, 2)$ cycles with Gauss-Seidel smoothing yields the shortest execution times. We formalise our findings in Preconditioner 6.5.

<table>
<thead>
<tr>
<th>Smoother</th>
<th>Cycle</th>
<th>144</th>
<th>576</th>
<th>2304</th>
<th>9216</th>
<th>36846</th>
<th>147456</th>
<th>589824</th>
<th>2359296</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>$1 \times V(2, 2)$</td>
<td>17.3</td>
<td>21</td>
<td>24.7</td>
<td>27.7</td>
<td>34.7</td>
<td>43.5</td>
<td>57</td>
<td>83</td>
</tr>
<tr>
<td>Jacobi</td>
<td>$2 \times V(2, 2)$</td>
<td>14.7</td>
<td>18.7</td>
<td>20.3</td>
<td>23.3</td>
<td>30</td>
<td>36</td>
<td>47.5</td>
<td>71</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>$1 \times V(2, 2)$</td>
<td>14.7</td>
<td>18.9</td>
<td>21.3</td>
<td>25.7</td>
<td>32</td>
<td>39.5</td>
<td>53.5</td>
<td>78</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>$2 \times V(2, 2)$</td>
<td>14.7</td>
<td>18</td>
<td>19</td>
<td>21.3</td>
<td>25.3</td>
<td>33</td>
<td>45</td>
<td>66</td>
</tr>
<tr>
<td>Euclid ILU(0)</td>
<td>$1 \times V(2, 2)$</td>
<td>14.7</td>
<td>17.3</td>
<td>18.3</td>
<td>20.7</td>
<td>26</td>
<td>29.5</td>
<td>41.5</td>
<td>59.5</td>
</tr>
</tbody>
</table>

Table 6.11: The average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver with different smoothers and different V-cycles used for the AMG approximation of the $\hat{F}$ block, applied to Example 6.2.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and Re = 100.

\footnote{We used the MATLAB function \texttt{polyfit} to generate a first order polynomial based on the data points.}
Table 6.12: The average number of iterations taken by the GMRES+$\hat{P}_{\text{LSC}}$ solver with different smoothers and different $V-$cycles used for the AMG approximation of the $\hat{F}$ block, applied to Example 6.2.1. The results are presented for the SDVT-NSE with $\alpha = 30^\circ$ and $Re = 100$.

<table>
<thead>
<tr>
<th>Smoother</th>
<th>Cycle</th>
<th>144</th>
<th>576</th>
<th>2304</th>
<th>9216</th>
<th>36846</th>
<th>147456</th>
<th>589824</th>
<th>2359296</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi, $\omega = 1$</td>
<td>$1 \times V(2, 2)$</td>
<td>17</td>
<td>22.3</td>
<td>26.3</td>
<td>29.3</td>
<td>32.5</td>
<td>45</td>
<td>63.5</td>
<td>95.5</td>
</tr>
<tr>
<td>Jacobi, $\omega = 2$</td>
<td>$2 \times V(2, 2)$</td>
<td>14.3</td>
<td>16.7</td>
<td>20</td>
<td>23</td>
<td>25.5</td>
<td>33.5</td>
<td>50</td>
<td>75.5</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>$1 \times V(2, 2)$</td>
<td>15.7</td>
<td>20.3</td>
<td>24.7</td>
<td>28.3</td>
<td>31.5</td>
<td>43</td>
<td>63</td>
<td>93</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>$2 \times V(2, 2)$</td>
<td>14.7</td>
<td>16.7</td>
<td>19.7</td>
<td>23</td>
<td>25</td>
<td>33</td>
<td>47</td>
<td>73</td>
</tr>
<tr>
<td>Euclid ILU(0)</td>
<td>$1 \times V(2, 2)$</td>
<td>13</td>
<td>16</td>
<td>18.3</td>
<td>20.7</td>
<td>23</td>
<td>31</td>
<td>44</td>
<td>64.5</td>
</tr>
</tbody>
</table>

Table 6.13: The execution times (s) taken by the GMRES+$\hat{P}_{\text{LSC}}$ solver with different smoothers and different $V-$cycles used for the AMG approximation of the $\hat{F}$ block, applied to Example 6.2.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and $Re = 100$.

<table>
<thead>
<tr>
<th>Smoother</th>
<th>Cycle</th>
<th>144</th>
<th>576</th>
<th>2304</th>
<th>9216</th>
<th>36846</th>
<th>147456</th>
<th>589824</th>
<th>2359296</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi, $\omega = 1$</td>
<td>$1 \times V(2, 2)$</td>
<td>0.01</td>
<td>0.02</td>
<td>0.08</td>
<td>0.38</td>
<td>1.55</td>
<td>9.18</td>
<td>38.00</td>
<td>253.44</td>
</tr>
<tr>
<td>Jacobi, $\omega = 2$</td>
<td>$2 \times V(2, 2)$</td>
<td>0.01</td>
<td>0.03</td>
<td>0.09</td>
<td>0.48</td>
<td>1.79</td>
<td>11.71</td>
<td>47.26</td>
<td>329.20</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>$1 \times V(2, 2)$</td>
<td>0.01</td>
<td>0.02</td>
<td>0.07</td>
<td>0.32</td>
<td>1.46</td>
<td>6.83</td>
<td>35.46</td>
<td>253.80</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>$2 \times V(2, 2)$</td>
<td>0.01</td>
<td>0.03</td>
<td>0.09</td>
<td>0.35</td>
<td>1.71</td>
<td>7.60</td>
<td>53.23</td>
<td>255.79</td>
</tr>
<tr>
<td>Euclid ILU(0)</td>
<td>$1 \times V(2, 2)$</td>
<td>0.01</td>
<td>0.04</td>
<td>0.13</td>
<td>0.57</td>
<td>2.62</td>
<td>10.81</td>
<td>52.55</td>
<td>335.00</td>
</tr>
</tbody>
</table>

Table 6.14: The execution times (s) taken by the GMRES+$\hat{P}_{\text{LSC}}$ solver with different smoothers and different $V-$cycles used for the AMG approximation of the $\hat{F}$ block, applied to Example 6.2.1. The results are presented for the SDVT-NSE with $\alpha = 30^\circ$ and $Re = 100$.

<table>
<thead>
<tr>
<th>Smoother</th>
<th>Cycle</th>
<th>144</th>
<th>576</th>
<th>2304</th>
<th>9216</th>
<th>36846</th>
<th>147456</th>
<th>589824</th>
<th>2359296</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi, $\omega = 1$</td>
<td>$1 \times V(2, 2)$</td>
<td>0.01</td>
<td>0.02</td>
<td>0.09</td>
<td>0.39</td>
<td>1.77</td>
<td>9.37</td>
<td>54.79</td>
<td>400.32</td>
</tr>
<tr>
<td>Jacobi, $\omega = 2$</td>
<td>$2 \times V(2, 2)$</td>
<td>0.01</td>
<td>0.02</td>
<td>0.10</td>
<td>0.44</td>
<td>1.98</td>
<td>11.35</td>
<td>68.71</td>
<td>433.85</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>$1 \times V(2, 2)$</td>
<td>0.01</td>
<td>0.02</td>
<td>0.09</td>
<td>0.38</td>
<td>1.77</td>
<td>8.93</td>
<td>63.45</td>
<td>390.84</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>$2 \times V(2, 2)$</td>
<td>0.01</td>
<td>0.02</td>
<td>0.10</td>
<td>0.43</td>
<td>1.89</td>
<td>9.99</td>
<td>60.37</td>
<td>418.58</td>
</tr>
<tr>
<td>Euclid ILU(0)</td>
<td>$1 \times V(2, 2)$</td>
<td>0.01</td>
<td>0.04</td>
<td>0.18</td>
<td>0.74</td>
<td>4.02</td>
<td>16.66</td>
<td>97.26</td>
<td>512.48</td>
</tr>
</tbody>
</table>

Preconditioner 6.5: $\hat{P}_{\text{LSC}}$, An inexact LSC Augmentation Preconditioner for two-dimensional problems.

Preconditioner: Block diagonal
Navier-Stokes Preconditioner: LSC
Momentum Preconditioner: Inexact inversion by BoomerAMG
Coarsening: Ruge-Stüber, $\theta = 0.25/0.668$ for the SVT-NSE and SDVT-NSE, respectively
Smoothing: Gauss-Seidel
Cycle: $1 \times V(2, 2)$
Pressure Poisson Operators: Inexact inversion by BoomerAMG
Coarsening: Ruge-Stüber, $\theta = 0.25$
Smoothing: Damped Jacobi, $\omega = \frac{3}{2}$
Cycle: $2 \times V(1, 1)$
W-block Preconditioner: Exact inversion by SuperLU
6.2. TWO-DIMENSIONAL CASE STUDIES

![Comparison of smoothing methods for the SVT of the NSE](image1)

Figure 6.4: Log-log plot of the average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver with different smoothers and different $V-$cycles used for the AMG approximation of the $\hat{F}$ block, applied to Example 6.2.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and $Re = 100$.

![Comparison of smoothing methods for the SDVT of the NSE](image2)

Figure 6.5: Log-log plot of the average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver with different smoothers and different $V-$cycles used for the AMG approximation of the $\hat{F}$ block, applied to Example 6.2.1. The results are presented for the SDVT-NSE with $\alpha = 30^\circ$ and $Re = 100$. 
Comparison of smoothing methods for the SVT of the NSE

\[ y = 1.0769x - 10.7188 \]
\[ y = x - 10.7188 \]

Jacobi 1xV(2,2)
Jacobi 2xV(2,2)
Gauss Seidel 1xV(2,2)
Gauss Seidel 2xV(2,2)
Euclid 1xV(2,2)

Figure 6.6: Log-log plot of the execution times (s) taken by the GMRES+\( \hat{P}_{LSC} \) solver with different smoothers and different V−cycles used for the AMG approximation of the \( \hat{F} \) block, applied to Example 6.2.1. The results are presented for the SVT-NSE with \( \alpha = 30^\circ \) and \( \text{Re} = 100 \).

Comparison of smoothing methods for the SDVT of the NSE

\[ y = 1.1247x - 10.9665 \]
\[ y = x - 10.9665 \]

Jacobi 1xV(2,2)
Jacobi 2xV(2,2)
Gauss Seidel 1xV(2,2)
Gauss Seidel 2xV(2,2)
Euclid 1xV(2,2)

Figure 6.7: Log-log plot of the execution times (s) taken by the GMRES+\( \hat{P}_{LSC} \) solver with different smoothers and different V−cycles used for the AMG approximation of the \( \hat{F} \) block, applied to Example 6.2.1. The results are presented for the SDVT-NSE with \( \alpha = 30^\circ \) and \( \text{Re} = 100 \).
6.2. TWO-DIMENSIONAL CASE STUDIES

Convergence Evaluation

In this section we numerically evaluate the performance of the different versions of the augmented preconditioner: $P_E$ (Preconditioner 6.1), $P_{LSC}$ (Preconditioner 6.2), and $\hat{P}_{LSC}$ (Preconditioner 6.5). We perform experiments for Example 6.2.1 with the following configurations:

- Both SVT and SDVT forms.
- Angle of rotation $\alpha = 0^\circ, 30^\circ$ and $67^\circ$.
- Reynolds number $Re = 0, 100$ and $200$.
- Uniform mesh refinements. The domain is discretised using uniform quadrilateral grids with discrete problem sizes between $N = 2304$ (corresponding to mesh size $h = 1/16$) and $N = 147456$ (mesh size $h = 1/128$) for $P_E$ and $P_{LSC}$ versions for the preconditioner and up to $N = 2359296$ ($h = 1/512$) for $\hat{P}_{LSC}$. These are the maximum problem sizes we were able to run with 8GB of RAM without using the swap memory.
- Preconditioned GMRES solver with a relative convergence tolerance of $\varepsilon_K = 10^{-6}$ starting from an initial guess of zero.

First we consider the performance of $P_E$. Here we are able to verify the results of the eigenvalue analysis reported in Section 4.6. The iteration counts and execution times are presented in Tables 6.15 and 6.16, respectively. We observe that there is no dependence on the angle of rotation $\alpha$. The iteration counts for the SVT-NSE show a slight increase with the Reynolds number, whilst for the SDVT-NSE we see that the iteration counts decrease with an increase in Reynolds number. The numbers of iterations in the SVT case are lower than for the SDVT counter-pairs. Overall, the iteration counts are low and bounded under mesh refinement. These results are in agreement with the analysis presented in Section 4.6 and the numerical eigenvalue results presented earlier.

A log-log plot of the execution times for both SVT and SDVT, with $\alpha = 67^\circ$ and $Re = 200$ is given in Figure 6.8 with a line of gradient 1 for comparison. A gradient of 1 would indicate optimal execution time (the execution time is linearly proportional to the size of the linear system). We see that the execution times are sub-optimal. The gradient of the line of best fit for all problem parameters considered in Figure 6.8 is approximately 1.7. Thus, we conjecture that the asymptotic behaviour of execution time of the iterative solver GMRES+$P_E$ is $O(N^{1.7})$. 
CHAPTER 6. NUMERICAL EVALUATION

Table 6.15: The average number of iterations taken by the GMRES+\( PE \) solver, applied to Example 6.2.1, as a function of the viscous term, ‘vis’, angle of rotation ‘\( \alpha \)’, Reynolds number, ‘\( Re \)’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>( \alpha )</th>
<th>( Re )</th>
<th>2304</th>
<th>9216</th>
<th>36846</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>0°</td>
<td>0</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>2.3</td>
<td>2.3</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30°</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2.3</td>
<td>2.3</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>2.3</td>
<td>2.3</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>67°</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2.3</td>
<td>2.3</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>2.3</td>
<td>2.3</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SDVT</td>
<td>0°</td>
<td>0</td>
<td>6.5</td>
<td>6.5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>4.7</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>4.3</td>
<td>4.7</td>
<td>4.7</td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30°</td>
<td>0</td>
<td>6.5</td>
<td>6.5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>4.7</td>
<td>4.7</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>4.3</td>
<td>4.7</td>
<td>4.7</td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>67°</td>
<td>0</td>
<td>6.5</td>
<td>6.5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>4.7</td>
<td>4.7</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>4.3</td>
<td>4.7</td>
<td>4.7</td>
<td>3.5</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.16: The execution times (s) taken by the GMRES+\( PE \) solver, applied to Example 6.2.1, as a function of the viscous term, ‘vis’, angle of rotation ‘\( \alpha \)’, Reynolds number, ‘\( Re \)’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>( \alpha )</th>
<th>( Re )</th>
<th>2304</th>
<th>9216</th>
<th>36846</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>0°</td>
<td>0</td>
<td>0.06</td>
<td>0.62</td>
<td>6.42</td>
<td>69.14</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.07</td>
<td>0.60</td>
<td>7.43</td>
<td>75.71</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.59</td>
<td>7.53</td>
<td>76.13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30°</td>
<td>0</td>
<td>0.06</td>
<td>0.58</td>
<td>5.93</td>
<td>67.53</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.07</td>
<td>0.59</td>
<td>7.79</td>
<td>75.96</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.59</td>
<td>7.71</td>
<td>75.26</td>
<td></td>
</tr>
<tr>
<td></td>
<td>67°</td>
<td>0</td>
<td>0.06</td>
<td>0.57</td>
<td>5.95</td>
<td>67.53</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.07</td>
<td>0.60</td>
<td>7.63</td>
<td>75.44</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.59</td>
<td>7.89</td>
<td>76.38</td>
<td></td>
</tr>
<tr>
<td>SDVT</td>
<td>0°</td>
<td>0</td>
<td>0.07</td>
<td>0.64</td>
<td>8.58</td>
<td>90.57</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.07</td>
<td>0.67</td>
<td>8.38</td>
<td>87.94</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.62</td>
<td>8.74</td>
<td>87.96</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30°</td>
<td>0</td>
<td>0.08</td>
<td>0.65</td>
<td>8.50</td>
<td>95.76</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.07</td>
<td>0.63</td>
<td>8.34</td>
<td>89.64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.63</td>
<td>8.33</td>
<td>87.76</td>
<td></td>
</tr>
<tr>
<td></td>
<td>67°</td>
<td>0</td>
<td>0.07</td>
<td>0.65</td>
<td>8.40</td>
<td>88.49</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.07</td>
<td>0.63</td>
<td>8.31</td>
<td>86.58</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.63</td>
<td>8.55</td>
<td>87.13</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.16: The execution times (s) taken by the GMRES+\( PE \) solver, applied to Example 6.2.1, as a function of the viscous term, ‘vis’, angle of rotation ‘\( \alpha \)’, Reynolds number, ‘\( Re \)’, and discrete problem size.

Next we present the set of results for GMRES preconditioned with the \( P_{LSC} \). For comparison purposes, in the case \( \alpha = 0 \), we are able to generate an equivalent problem to Example 6.2.1 but instead of imposing parallel outflow with Lagrange multipliers at the boundary \( \partial \Omega_O \), we imposed parallel and axially traction-free outflow by setting \( u_2 = 0 \) and leaving \( u_1 \) free (see Section 2.2.2). Recall that this implies that the traction
Execution times for GMRES+$P_E$ with different problem sizes

Figure 6.8: Log-log plot of the execution times (s) taken by the GMRES+$P_E$ solver, applied to Example 6.2.1. The results are presented for angle of rotation $\alpha = 67^\circ$ and Reynolds number $Re = 200$. The best fit line has gradient $1.7$, the line of gradient $1$ is for comparison.

(force) components, $T_i$, applied to the flow at $\partial\Omega_O$ are equal to zero:

$$T_i = -p\hat{n}_i + \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \hat{n}_j = 0, \quad \text{on } \partial\Omega_O. \quad (6.2.4)$$

Given that $u_2$ is set to zero and $\hat{n} = [1, 0]^T$ at $\partial\Omega_O = \{x_1 = 1, x_2 \in [0, 1]\}$ (6.2.4) reduces to

$$-p + 2\frac{\partial u_1}{\partial x_1} = 0, \quad \text{on } \partial\Omega_O. \quad (6.2.5)$$

We summarise this in Example 6.2.2.

Example 6.2.2. Poiseuille flow through a unit square with $\alpha = 0^\circ$.

The domain: a unit square $\Omega = [0, 1]^2$.

Boundary conditions: The flow is driven by imposing a parabolic velocity profile along $\partial\Omega_1$. Along the characteristic boundary, $\partial\Omega_C$, the no-slip condition $u_i = 0$, $i = 1, 2$ is prescribed. At the outflow boundary, $\partial\Omega_O$, we impose parallel and axially traction-free outflow by setting $u_2 = 0$ and ‘do nothing’ for $u_1$. 
Thus, boundary conditions on each of the four segments of the boundary are:

\[
\begin{align*}
  u_1 = u_2 &= 0, \quad \text{at } x_1 \in [0, 1], x_2 = 0 \quad \text{(characteristic (bottom) boundary } \partial \Omega_C), \\
  u_1 = u_2 &= 0, \quad \text{at } x_1 \in [0, 1], x_2 = 1 \quad \text{(characteristic (top) boundary } \partial \Omega_C), \\
  u_1 = x_2(1 - x_2), u_2 &= 0, \quad \text{at } x_1 = 0, x_2 \in [0, 1] \quad \text{(parabolic inflow boundary } \partial \Omega_I), \\
  -p + 2 \frac{\partial u_1}{\partial x_1} &= 0, u_2 = 0 \quad \text{at } x_1 = 1, x_2 \in [0, 1] \quad \text{('parallel outflow' boundary } \partial \Omega_O).
\end{align*}
\]

The FEM discretisation of Example 6.2.2 leads to the standard form of the NS problem, described in Section 2.3.1, with the linear system obtained from Newton linearisation \((2.3.38)\). The LSC preconditioner is known to work well in this case \([68, pp. 354–359]\). This gives us the opportunity to compare the performance of using the LSC preconditioner when applied to the augmented NS block \(\tilde{F}_{ns}\) and to the standard NS Jacobian \(F_{ns}\). When using the LSC preconditioner for Example 6.2.2, we use SuperLU for the solves involving the momentum block \(F\) and the sparse pressure Poisson operator \([6.2.2]\). We denote this by \(P_{LSC}^{NS}\). The iteration counts and execution times are presented in Tables 6.17 and 6.18, respectively. The iteration counts and execution times for \(\alpha \neq 0\) when \(P_{LSC}\) is used as a preconditioner for Example 6.2.1 are presented in Tables 6.19 and 6.20, respectively.

![Table 6.17: The average number of iterations taken by the GMRES+P_LSC solver applied to Example 6.2.1 with \(\alpha = 0^\circ\) (left), and the GMRES+P_{LSC}^{NS} solver applied to Example 6.2.2 (right), as a function of the viscous term, ‘vis’, Reynolds number, ‘Re’, and discrete problem size.](image)

In general, the execution times observed in Table 6.18 are comparable for both Examples 6.2.1 and 6.2.2, and the execution times for the SVT-NSE are shorter despite showing slightly higher iteration counts than their SDVT counter-pairs. This can be explained by the number of nonzeros and sparsity pattern of the Jacobian matrices. In the cases of exact versions of a preconditioner, the majority of the computational work happens in the preconditioner setup phase for the LU decomposition for the momentum and pressure Poisson matrices, not in the solve phase (unless we have a very large
### Table 6.18: The execution times (s) taken by the GMRES+$P_{LSC}$ solver applied to Example 6.2.1 with $\alpha = 0^\circ$ (left), and the GMRES+$P_{NSLSC}$ solver applied to Example 6.2.2 (right), as a function of the viscous term, ‘vis’, Reynolds number, ‘Re’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>$\alpha$</th>
<th>Re</th>
<th>$P_{LSC}$</th>
<th>$P_{NSLSC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>0</td>
<td>0.05</td>
<td>0.36</td>
<td>2.98</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.06</td>
<td>0.44</td>
<td>3.85</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.43</td>
<td>3.93</td>
</tr>
<tr>
<td>SDVT</td>
<td>0</td>
<td>0.08</td>
<td>0.52</td>
<td>4.41</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.07</td>
<td>0.48</td>
<td>4.10</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.48</td>
<td>4.23</td>
</tr>
</tbody>
</table>

### Table 6.19: The average number of iterations taken by the GMRES+$P_{LSC}$ solver applied to Example 6.2.1, as a function of the viscous term, ‘vis’, angle of rotation ‘$\alpha$’, Reynolds number, ‘Re’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>$\alpha$</th>
<th>Re</th>
<th>$P_{LSC}$</th>
<th>$P_{NSLSC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>30$^\circ$</td>
<td>0</td>
<td>0.05</td>
<td>20.5</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.06</td>
<td>0.45</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.45</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>67$^\circ$</td>
<td>0</td>
<td>0.05</td>
<td>20.5</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.06</td>
<td>0.45</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.45</td>
<td>24</td>
</tr>
<tr>
<td>SDVT</td>
<td>30$^\circ$</td>
<td>0</td>
<td>0.07</td>
<td>16.3</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.06</td>
<td>0.45</td>
<td>18.5</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.45</td>
<td>21.7</td>
</tr>
<tr>
<td></td>
<td>67$^\circ$</td>
<td>0</td>
<td>0.08</td>
<td>16.3</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.07</td>
<td>0.49</td>
<td>18.5</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.48</td>
<td>21.7</td>
</tr>
</tbody>
</table>

### Table 6.20: The execution times (s) taken by the GMRES+$P_{LSC}$ solver applied to Example 6.2.1, as a function of the viscous term, ‘vis’, angle of rotation ‘$\alpha$’, Reynolds number, ‘Re’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>$\alpha$</th>
<th>Re</th>
<th>$P_{LSC}$</th>
<th>$P_{NSLSC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>30$^\circ$</td>
<td>0</td>
<td>0.05</td>
<td>3.13</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.06</td>
<td>0.45</td>
<td>4.12</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.45</td>
<td>4.03</td>
</tr>
<tr>
<td></td>
<td>67$^\circ$</td>
<td>0</td>
<td>0.05</td>
<td>3.12</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.07</td>
<td>0.45</td>
<td>4.04</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.45</td>
<td>4.01</td>
</tr>
<tr>
<td>SDVT</td>
<td>30$^\circ$</td>
<td>0</td>
<td>0.07</td>
<td>4.53</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.07</td>
<td>0.49</td>
<td>4.53</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.48</td>
<td>4.33</td>
</tr>
<tr>
<td></td>
<td>67$^\circ$</td>
<td>0</td>
<td>0.08</td>
<td>4.51</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.07</td>
<td>0.49</td>
<td>4.53</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.07</td>
<td>0.48</td>
<td>4.33</td>
</tr>
</tbody>
</table>
number of iteration counts, say a few hundreds). Thus, the execution times are primarily dependent on the sparsity pattern of the Jacobian matrix. A nonzero Reynolds number will introduce the block matrices $W_{ij}$ ($i, j = 1, 2, 3$) defined by (2.3.42) and if we are using the SDVT-NSE, we have the block matrices $G_{ij}$ ($i, j = 1, 2, 3$) as defined by (2.3.43). The block matrices $W_{ij}$ have sparsity similar to that of a mass matrix, whilst the matrices $G_{ij}$ will make the off-diagonal momentum blocks as dense as the diagonal momentum blocks. Because the momentum and pressure Poisson matrices used within the LSC approximation are similar in terms of size and sparsity pattern for both Examples 6.2.1 and 6.2.2, the solvers exhibit comparable execution times. This is also the reason why the execution times for the SDVT-NSE are generally longer than for SVT despite having lower iteration counts. Note that the largest iteration counts and the shortest execution times are observed for the SVT-NSE with $Re = 0$. Thus, unless the iteration counts for Examples 6.2.1 and 6.2.2 differ significantly, we expect the execution times for the solvers with $P_{LSC}$ and $P_{NS LSC}$ to remain comparable.

The iteration counts for $\alpha \neq 0$ are not affected by the angle of rotation. The largest increase in iteration counts under mesh refinement is observed for $Re = 0$. We note that the LSC preconditioner is not designed for the case $Re = 0$, we should use the block diagonal preconditioner described in [256] instead. The most important observation from Table 6.17 is that the asymptotic rate of increase in iteration counts for $P_{LSC}$ in Example 6.2.1 matches that of $P_{NS LSC}$ in Example 6.2.2. This suggests that the LSC preconditioner exhibit similar efficiency when applied in a black-box fashion to the augmented NS Jacobian $\tilde{F}_{ns}$ as in the standard case $F_{ns}$. This observation holds for both forms of the viscous term. To demonstrate this, we plot in Figure 6.9, the iteration counts as a function of $\log(N)$ for the SVT-NSE with Reynolds numbers $Re = 0$ and 200. We have joined the data points with dashed lines to emphasise that the iteration counts with $P_{LSC}$ follow the same pattern as $P_{NS LSC}$. We also note that if we consider only the three data points corresponding to the largest problem sizes in Figure 6.9, the straight line exhibited by the iteration counts for $Re = 200$ suggests that the iteration counts grow as $O(\log N)$ in both Example 6.2.1 and Example 6.2.2.

The log-log plot for the execution times as a function of the problem size for parameters $\alpha = 67^\circ$ and $Re = 0$ and 200 is depicted in Figure 6.10 along with a line with a gradient of 1 for comparison. We observe that the execution times still do not increase linearly with the discrete problem size. The gradient of the line of best
fits through the data points is approximately 1.55, thus this suggests that the $\mathcal{P}_{LSC}$ version of the preconditioner leads to an iterative solver which execution time scales as $O(N^{1.55})$.

Lastly, we present the results for $\hat{\mathcal{P}}_{LSC}$. Again, for $\alpha = 0^\circ$ we are able to compare the iteration counts and solution times for $\hat{\mathcal{P}}_{LSC}$ with that of the LSC preconditioner $\mathcal{P}_{LSC}^N$, applied to an equivalent problem presented in Example 6.2.2. The AMG parameters used for $\mathcal{P}_{LSC}^N$ are given in Preconditioner 6.5. We denote this preconditioner $\hat{\mathcal{P}}_{LSC}^N$. The iteration counts are shown in Table 6.21 and execution times are given in Table 6.22. For $\alpha \neq 0$ the corresponding results for iteration counts and execution times are shown in Tables 6.23 and 6.24, respectively.

![Figure 6.9: Semi-log plot of the average number of iterations taken by the GMRES+$\mathcal{P}_{LSC}$ and GMRES+$\mathcal{P}_{LSC}^N$ solver in Examples 6.2.1 and 6.2.2, respectively, for the SVT-NSE.](image)

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>9216</th>
<th>36846</th>
<th>147456</th>
<th>589824</th>
<th>2359296</th>
<th>9090</th>
<th>36610</th>
<th>146946</th>
<th>588802</th>
<th>2357250</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>0</td>
<td>28</td>
<td>34.5</td>
<td>42.5</td>
<td>53</td>
<td>70.5</td>
<td>14</td>
<td>17.5</td>
<td>23.5</td>
<td>31</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>24.3</td>
<td>29.3</td>
<td>33.5</td>
<td>40.5</td>
<td>52</td>
<td>12.7</td>
<td>16</td>
<td>20</td>
<td>25</td>
<td>31.5</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>23.7</td>
<td>29.7</td>
<td>37.3</td>
<td>40.5</td>
<td>52.5</td>
<td>12</td>
<td>15.7</td>
<td>21</td>
<td>25</td>
<td>32</td>
</tr>
<tr>
<td>SDVT</td>
<td>0</td>
<td>27</td>
<td>33.5</td>
<td>44.5</td>
<td>61</td>
<td>96</td>
<td>11.5</td>
<td>15.5</td>
<td>19.5</td>
<td>25.5</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>24.7</td>
<td>26.5</td>
<td>34</td>
<td>45.5</td>
<td>78.5</td>
<td>11</td>
<td>14</td>
<td>16.5</td>
<td>20.5</td>
<td>26.5</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>28.7</td>
<td>29.3</td>
<td>41.5</td>
<td>43.5</td>
<td>70</td>
<td>12.3</td>
<td>14</td>
<td>18</td>
<td>20</td>
<td>26.5</td>
</tr>
</tbody>
</table>

Table 6.21: The average number of iterations taken by the GMRES+$\hat{\mathcal{P}}_{LSC}$ solver applied to Example 6.2.1 with $\alpha = 0^\circ$ (left), and the GMRES+$\hat{\mathcal{P}}_{LSC}^N$ solver applied to Example 6.2.2 (right), as a function of the viscous term, ‘vis’, Reynolds number, ‘Re’, and discrete problem size.
Figure 6.10: Log-log plot of the execution times (s) taken by the GMRES\(\hat{+}\)PLSC solver applied to Example 6.2.1. The results are presented for \(\alpha = 67^\circ\). The best fit line has gradient 1.55 and the line of slope 1 is for comparison.

Table 6.22: The execution times (s) taken by the GMRES\(\hat{+}\)PLSC solver applied to Example 6.2.1 with \(\alpha = 0^\circ\) (left), and the GMRES\(\hat{+}\)PLSC solver applied to Example 6.2.2 (right), as a function of the viscous term, ‘vis’, Reynolds number, ‘Re’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>9216</th>
<th>36846</th>
<th>147456</th>
<th>589824</th>
<th>2359296</th>
<th>9090</th>
<th>36610</th>
<th>146946</th>
<th>588802</th>
<th>2357250</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>0</td>
<td>0.26</td>
<td>1.20</td>
<td>6.14</td>
<td>34.22</td>
<td>196.68</td>
<td>0.20</td>
<td>0.95</td>
<td>4.63</td>
<td>24.16</td>
<td>137.15</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.31</td>
<td>1.38</td>
<td>6.66</td>
<td>29.01</td>
<td>188.23</td>
<td>0.24</td>
<td>1.09</td>
<td>5.25</td>
<td>27.44</td>
<td>108.71</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.31</td>
<td>1.48</td>
<td>6.99</td>
<td>34.98</td>
<td>164.33</td>
<td>0.24</td>
<td>1.18</td>
<td>5.66</td>
<td>28.56</td>
<td>128.70</td>
</tr>
<tr>
<td>SDVT</td>
<td>0</td>
<td>0.38</td>
<td>1.85</td>
<td>10.25</td>
<td>67.40</td>
<td>366.01</td>
<td>0.25</td>
<td>1.26</td>
<td>6.36</td>
<td>36.99</td>
<td>160.83</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.37</td>
<td>1.66</td>
<td>8.11</td>
<td>44.97</td>
<td>305.25</td>
<td>0.27</td>
<td>1.42</td>
<td>6.60</td>
<td>32.73</td>
<td>131.21</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.40</td>
<td>2.26</td>
<td>8.64</td>
<td>50.80</td>
<td>283.70</td>
<td>0.28</td>
<td>1.39</td>
<td>7.39</td>
<td>31.49</td>
<td>136.34</td>
</tr>
</tbody>
</table>

From Tables 6.21 and 6.23 we observe that the iteration counts increase under mesh refinement for all problem parameters considered. The largest iteration counts are observed for \(\text{Re} = 0\). The increase in \(\text{Re}\) generally leads to the decrease in iteration counts (for a fixed problem size). We observe that the iteration counts for Example 6.2.1 are roughly double than of Example 6.2.2. In order to establish the correlation between the asymptotic growth in iteration counts of the GMRES solver with \(\hat{\text{PLSC}}\) for Example 6.2.1 and \(\hat{\text{PLSC}}\) for Example 6.2.2 we plot a log-log graph in Figure 6.11 of the results from Table 6.21 for \(\text{Re} = 0\) and 200. We observe from Figure 6.11 that the asymptotic behaviour of the iteration counts for Example 6.2.1 roughly follow that observed in Example 6.2.2. The computational work is now split more evenly between
the preconditioner setup phase and preconditioner solve phase. Thus, the increase in iteration counts generally leads to longer execution times, as observed in Table 6.22.

Now we establish the correlation between the asymptotic growth in execution times of the GMRES solver with $\hat{P}_{LSC}$ for Example 6.2.1 and $\hat{P}_{NS,LSC}$ for Example 6.2.2 by plotting the log-log graph of the results from Table 6.21 for $Re = 0$ and 200. In Figure 6.12 a strong correlation in execution times is observed.

For $\alpha = 67^\circ$, we plot a log-log graph for both the SVT and SDVT-NSE with $Re = 0$ and 200 along with a line of gradient 1 for comparison in Figure 6.13. We do not observe optimal scaling. The lines of best fits for the problem parameters considered have gradients of approximately 1.2, thus we conjecture the asymptotic growth in execution times of $O(N^{1.2})$.

An important point to note is that the iteration counts for GMRES+$\hat{P}_{LSC}$ are no longer independent of the angle of rotation $\alpha$, (see Table 6.23, cf. Tables 6.16

<table>
<thead>
<tr>
<th>vis</th>
<th>$\alpha$</th>
<th>Re</th>
<th>9216</th>
<th>36846</th>
<th>147456</th>
<th>589824</th>
<th>2359296</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>$30^\circ$</td>
<td>0</td>
<td>30</td>
<td>39.5</td>
<td>54</td>
<td>74.5</td>
<td>102.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>25.7</td>
<td>32</td>
<td>39.5</td>
<td>53.5</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
<td>24.7</td>
<td>32</td>
<td>41.7</td>
<td>53</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>$67^\circ$</td>
<td>0</td>
<td>33.5</td>
<td>44</td>
<td>61</td>
<td>84</td>
<td>109.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>26.7</td>
<td>34.3</td>
<td>45</td>
<td>60</td>
<td>91.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
<td>26.3</td>
<td>34</td>
<td>46.3</td>
<td>60</td>
<td>88.5</td>
</tr>
<tr>
<td>SDVT</td>
<td>$30^\circ$</td>
<td>0</td>
<td>30.5</td>
<td>40</td>
<td>56</td>
<td>81.5</td>
<td>107.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>28.3</td>
<td>31.5</td>
<td>43</td>
<td>63</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
<td>28.3</td>
<td>37</td>
<td>42</td>
<td>59.5</td>
<td>86.5</td>
</tr>
<tr>
<td></td>
<td>$67^\circ$</td>
<td>0</td>
<td>30</td>
<td>40</td>
<td>56</td>
<td>82.5</td>
<td>107.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>27</td>
<td>30.5</td>
<td>42</td>
<td>62.5</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
<td>33.7</td>
<td>35</td>
<td>42.5</td>
<td>59</td>
<td>89</td>
</tr>
</tbody>
</table>

Table 6.23: The average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver applied to Example 6.2.1, as a function of the viscous term, ‘vis’, angle of rotation ‘$\alpha$’, Reynolds number, ‘Re’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>$\alpha$</th>
<th>Re</th>
<th>9216</th>
<th>36846</th>
<th>147456</th>
<th>589824</th>
<th>2359296</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>$30^\circ$</td>
<td>0</td>
<td>0.27</td>
<td>1.35</td>
<td>8.45</td>
<td>54.53</td>
<td>281.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>0.36</td>
<td>1.44</td>
<td>8.69</td>
<td>36.64</td>
<td>262.22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
<td>0.31</td>
<td>1.64</td>
<td>7.43</td>
<td>36.11</td>
<td>243.24</td>
</tr>
<tr>
<td></td>
<td>$67^\circ$</td>
<td>0</td>
<td>0.29</td>
<td>1.49</td>
<td>8.43</td>
<td>56.53</td>
<td>299.38</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>0.32</td>
<td>2.26</td>
<td>7.67</td>
<td>41.03</td>
<td>316.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
<td>0.32</td>
<td>1.63</td>
<td>8.04</td>
<td>39.14</td>
<td>256.06</td>
</tr>
<tr>
<td>SDVT</td>
<td>$30^\circ$</td>
<td>0</td>
<td>0.39</td>
<td>2.00</td>
<td>11.62</td>
<td>73.54</td>
<td>415.41</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>0.38</td>
<td>1.66</td>
<td>9.36</td>
<td>54.07</td>
<td>345.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
<td>0.40</td>
<td>1.92</td>
<td>9.40</td>
<td>55.54</td>
<td>310.87</td>
</tr>
<tr>
<td></td>
<td>$67^\circ$</td>
<td>0</td>
<td>0.38</td>
<td>2.12</td>
<td>11.15</td>
<td>67.75</td>
<td>407.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>0.37</td>
<td>1.91</td>
<td>8.92</td>
<td>51.26</td>
<td>330.02</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
<td>0.40</td>
<td>1.87</td>
<td>9.06</td>
<td>50.31</td>
<td>350.29</td>
</tr>
</tbody>
</table>

Table 6.24: The execution times (s) taken by the GMRES+$\hat{P}_{LSC}$ solver applied to Example 6.2.1, as a function of the viscous term, ‘vis’, angle of rotation ‘$\alpha$’, Reynolds number, ‘Re’, and discrete problem size.
Iteration counts for GMRES+$\hat{P}_{LSC}$ and GMRES+$\hat{P}_{NS}^{LSC}$ using the SVT of the NSE

$$\begin{array}{c}
\begin{array}{c}
\text{Log of iteration counts} \\
\text{Log of problem size}
\end{array}
\end{array}$$

Figure 6.11: Log-log plot of the average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver and the GMRES+$\hat{P}_{NS}^{LSC}$ solver in Examples 6.2.1 and 6.2.2, respectively.

Execution times for GMRES+$\hat{P}_{LSC}$ and GMRES+$\hat{P}_{NS}^{LSC}$ for the SVT of the NSE

$$\begin{array}{c}
\begin{array}{c}
\text{Log of execution time} \\
\text{Log of problem size}
\end{array}
\end{array}$$

Figure 6.12: Log-log plot of the execution times (s) taken by the GMRES+$\hat{P}_{LSC}$ solver and the GMRES+$\hat{P}_{NS}^{LSC}$ solver, using the SVT-NSE.
6.2. TWO-DIMENSIONAL CASE STUDIES

Figure 6.13: Log-log plot of the execution times (s) taken by the GMRES+\(\hat{P}_{LSC}\) solver applied to Example 6.2.1. The results are presented for \(\alpha = 67^\circ\). The best fit line has gradient 1.2 and the line of slope 1 is for comparison.

and [6.19]. This dependence is only observed in the AMG version of the augmentation preconditioner \(\hat{P}_{LSC}\). To investigate this effect further we vary the angle of rotation for \(\alpha \in [0^\circ, 360^\circ]\) with increments of 1\(^{\circ}\) for the SVT-NSE with Re = 100 and a discrete problem size of \(N = 589824\) (corresponding to mesh size \(h = 1/256\)). The iteration counts are illustrated in Figure 6.14.

We observe a repeated pattern in the peaks and troughs of iteration counts, this suggests the influence of the perturbation term \(L^TW^{-1}L\) in the NS momentum block. For the case of parallel outflow through a straight boundary tilted by the angle \(\alpha\), the perturbation block matrix \(L^TW^{-1}L\) is defined by (4.5.8). Due to the analysis in Section 4.6 we have \(\sigma = \|F\|_\infty \geq 0\). By taking \(\sigma = 1\) for simplicity, the values added to the diagonal entries of the diagonal blocks (2, 2) and (4, 4) are \(\sin^2(\alpha)\) and \(\cos^2(\alpha)\), respectively. The value added to the diagonal entries of the off-diagonal blocks (2, 4) and (4, 2) is \(-\cos(\alpha)\sin(\alpha)\). In Figure 6.15 we superimpose the iteration counts depicted in Figure 6.14 and the scaled perturbation values (\(\sigma = 1\)) added to the diagonals of the augmented momentum blocks. We can observe a striking correlation between the iteration counts and the magnitude and sign of the perturbation terms added to the off-diagonal blocks of \(\hat{F}\). If we consider the interval \(\alpha \in [0^\circ, 180^\circ]\), we observe that the iteration counts are lower for \(\alpha \in [0^\circ, 90^\circ]\), where \(-\cos(\alpha)\sin(\alpha)\)
Figure 6.14: The average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver applied to Example 6.2.1, as a function of the angle of rotation $\alpha^\circ = [0^\circ, 360^\circ]$ in increments of $1^\circ$. The results are presented for the SVT-NSE with $Re = 100$, problem size $N = 589824$, and $\sigma = \|F\|_\infty$.

Figure 6.15: Plot of the scaled perturbation values ($\sigma = 1$) together with the average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver (with $\sigma = \|F\|_\infty$) applied to Example 6.2.1 as a function of the angle of rotation $\alpha^\circ \in [0^\circ, 360^\circ]$ in increments of $1^\circ$. The results are presented for the SVT-NSE with $Re = 100$ and $N = 589824$. 
is negative. This implies that we are adding positive numbers to the diagonal of the
blocks (2, 2) and (4, 4) and negative numbers to the diagonals of the off-diagonal blocks
(2, 4) and (4, 2). Both cases have beneficial effects to the performance of AMG. The
former increases diagonal dominance of the Jacobian, making smoothers more effective,
while the latter makes the influence of the unknowns at $\partial \Omega$ to its neighbours larger,
making them good candidates for coarse grid points. The lowest iteration counts are
observed for $\alpha = 0^\circ$ and $90^\circ$ when the off-diagonal perturbation blocks are zero, and
$\alpha = 45^\circ$ when the off diagonal perturbation terms are the most negative.

The iteration counts are higher in the interval $\alpha \in (90^\circ, 180^\circ)$ where all the per-
turbation terms are positive. In this case the augmented momentum block $\hat{F}$ becomes
less diagonally dominant, which decreases the effectiveness of the AMG smoother and
the positive off-diagonal perturbation terms could lead to positive off-diagonal entries
in $\hat{F}$ which decreases the effectiveness of the AMG interpolation and coarsening pro-
cedures. With this in mind, we investigate if $\hat{P}_{LSC}$ will always yield a lower execution
time than $P_{LSC}$ for Example 6.2.1. In this context the iteration counts, preconditioner
setup times and linear solve times for $N = 147456$ are presented in Table 6.25. We see
that even for $\alpha = 135^\circ$, where the iteration counts for $\hat{P}_{LSC}$ is more than double of
$P_{LSC}$, the execution time is still higher for the $P_{LSC}$ since setting up the LU decom-
position of $\hat{F}$ and $BM^{-1}B^T$ takes longer than the time for extra iterations required
for the GMRES solver to converge with $\hat{P}_{LSC}$.

<table>
<thead>
<tr>
<th>Prec.</th>
<th>0°</th>
<th>45°</th>
<th>90°</th>
<th>135°</th>
<th>180°</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{LSC}$</td>
<td>26.3</td>
<td>26.3</td>
<td>26.3</td>
<td>26.3</td>
<td>26.3</td>
</tr>
<tr>
<td>$\hat{P}_{LSC}$</td>
<td>37.3</td>
<td>37</td>
<td>37.3</td>
<td>68.7</td>
<td>37.3</td>
</tr>
</tbody>
</table>

Table 6.25: Comparison of the performance of the GMRES+$P_{LSC}$ solver and the
GMRES+$\hat{P}_{LSC}$ solver, applied to Example 6.2.1, in terms of the average iteration counts,
preconditioner setup times (s) and linear solve times (s). The results are presented for the
SVT-NSE with Reynolds number Re = 200 and problem size $N = 147456$.

We conclude this section with a comparison of the execution times of direct solve
(using SuperLU) and GMRES (using Trilinos AztecOO) preconditioned by $P_E$, $P_{LSC}$
and $\hat{P}_{LSC}$ in Figure 6.16. We have not used a log scale in this graph to compare
the absolute growth rates of these times. For the solvers using SuperLU, the maximum
problem size achievable on the target architecture is $N = 147456$. For these
problem sizes, we see that direct solve and GMRES+$\mathcal{P}_E$ have similar execution times, GMRES+$\mathcal{P}_{LSC}$ takes half as long, while GMRES+$\hat{\mathcal{P}}_{LSC}$ is considerably faster.

![Comparison of solvers](Figure 6.16: Execution times (s) for direct solve, and the GMRES solver preconditioned with $\mathcal{P}_E$, $\mathcal{P}_{LSC}$ and $\hat{\mathcal{P}}_{LSC}$ applied to [Example 6.2.1] using the SVT-NSE with $\alpha = 67^\circ$ and $Re = 200$.)

### 6.2.2 Uniform Radial Flow through a Quarter Annulus

Next we study an example of a domain with a curvilinear outflow boundary $\partial \Omega_O$. In this case it is not possible to enforce parallel and axially traction-free flow on $\partial \Omega_O$ using the standard approach by setting the velocity component orthogonal to the fluid flow to zero, and we need to use Lagrange multipliers. We consider uniform radial flow through a quarter annulus (see Figure 6.17). The domain is the area in the first quadrant enclosed by the Cartesian coordinate axes and two circles given by $x_1^2 + x_2^2 = 1$ and $x_1^2 + x_2^2 = 9$. Then the four boundary segments are:

- $\partial \Omega_I = [x_1, x_2 > 0, x_1^2 + x_2^2 = 1]$, (Inflow boundary),
- $\partial \Omega_O = [x_1, x_2 > 0, x_1^2 + x_2^2 = 9]$, (Outflow boundary),
- $\partial \Omega_{s1} = [x_1 \in [1,3], x_2 = 0]$, (Characteristic boundary),
- $\partial \Omega_{s2} = [x_1 = 0, x_2 \in [1,3]]$, (Characteristic boundary).
This domain is obtained by transforming the unit square domain from Example 6.2.1 as follows: Given \((x_1^*, x_2^*) \in [0, 1]^2\), the new coordinates are calculated by
\[x_1 = r \cos(\alpha^*) \quad \text{and} \quad x_2 = r \sin(\alpha^*),\]
where \(r = 1 + 2x_1^*, \alpha^* = 90x_2^*\). The formal specification of the problem is as follows:

**Example 6.2.3. Uniform radial flow through a quarter annulus.**

The domain: a quarter annulus, \(\Omega = \{1 \leq x_1^2 + x_2^2 \leq 9\}\)

The boundary \(\partial \Omega\): inner radius 1 and outer radius 3 (see Figure 6.17). Uniform radial flow is imposed at \(\partial \Omega_I\). For the two straight boundaries: on \(\partial \Omega_{s1}\) we set \(u_2 = 0\) and ‘do nothing’ for \(u_1\), while on \(\partial \Omega_{s2}\) we set \(u_1 = 0\) and ‘do nothing’ for \(u_2\). Parallel outflow is enforced with Lagrange multipliers along \(\partial \Omega_O\). Let \(\alpha\) be the angle formed between the horizontal axis and the outwards unit normal at each discrete point on \(\partial \Omega_I\). Then the boundary conditions are:

\[u_1 = \cos(\alpha), u_2 = \sin(\alpha), \quad x_1, x_2 > 0, x_1^2 + x_2^2 = 1 \quad (\text{Uniform radial inflow } \partial \Omega_I),\]
\[\mathbf{u} \cdot \hat{\mathbf{t}} = 0, \quad x_1, x_2 > 0, x_1^2 + x_2^2 = 9 \quad (\text{Parallel outflow’ boundary } \partial \Omega_O),\]
\[u_1 = \text{‘free’}, u_2 = 0, \quad x_1 \in [1, 3], x_2 = 0 \quad (\text{Bottom boundary } \partial \Omega_{s1}),\]
\[u_1 = 0, u_2 = \text{‘free’}, \quad x_1 = 0, x_2 \in [1, 3] \quad (\text{Left boundary } \partial \Omega_{s2}).\]

![Figure 6.17: Plot of the solution for the steady-state uniform radial flow through a quarter annulus with Re = 100. The coloured contours and streamlines represent the pressure gradient and velocity flow field, respectively.](image)

**Convergence Evaluation**

In this section we numerically evaluate the performance of the different versions of the augmented preconditioner \(\mathcal{P}_E, \mathcal{P}_{LSC}, \text{ and } \hat{\mathcal{P}}_{LSC}\). For \(\hat{\mathcal{P}}_{LSC}\) we use the same settings...
as Preconditioner 6.5 except for that we use $2 \times V(2,2)$ cycles with Gauss-Seidel smoothing for SVT and $1 \times V(2,2)$ cycles with ILU(0) smoothing for the SDVT-NSE. The increased robustness of the smoothers reflects the difficulty of the problem under consideration. We perform experiments on Example 6.2.3 with the following configurations:

- Both the SVT-NSE and SDVT-NSE.
- Reynolds number $Re = 0, 100$ and 200.
- Several uniform mesh refinements. The domain is discretised using uniform quadrilateral grids with discrete problem sizes of $N = 2368$ (corresponding to mesh size $h = 1/16$) up to $N = 147968$ (mesh size $h = 1/128$) for $P_E$ and $P_{LSC}$ and up to $N = 2361344$ ($h = 1/512$) for $\hat{P}_{LSC}$.
- Preconditioned GMRES solver with a relative convergence tolerance of $\varepsilon_K = 10^{-6}$ starting with an initial solution guess of zero.

First we assess the performance of $P_E$. The iteration counts and execution times are presented in Tables 6.26 and 6.27, respectively.

### Table 6.26: The average number of iterations taken by the GMRES+$P_E$ solver applied to Example 6.2.3 as a function of the viscous term, ‘vis’, Reynolds number, ‘Re’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>2368</th>
<th>9344</th>
<th>37120</th>
<th>147968</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2.8</td>
<td>2</td>
<td>1.7</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>2.5</td>
<td>2.5</td>
<td>1.7</td>
<td>1.3</td>
</tr>
<tr>
<td>SDVT</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>2.8</td>
<td>2</td>
<td>1.7</td>
<td>1.7</td>
</tr>
</tbody>
</table>

### Table 6.27: The execution times (s) taken by the GMRES+$P_E$ solver applied to Example 6.2.3 as a function of the viscous term, ‘vis’, Reynolds number, ‘Re’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>2368</th>
<th>9344</th>
<th>37120</th>
<th>147968</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>0</td>
<td>0.07</td>
<td>0.60</td>
<td>6.05</td>
<td>66.73</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.08</td>
<td>0.70</td>
<td>7.68</td>
<td>73.46</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.08</td>
<td>0.71</td>
<td>7.89</td>
<td>73.50</td>
</tr>
<tr>
<td>SDVT</td>
<td>0</td>
<td>0.08</td>
<td>0.79</td>
<td>8.47</td>
<td>79.05</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.07</td>
<td>0.76</td>
<td>8.46</td>
<td>77.36</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>0.08</td>
<td>0.79</td>
<td>8.28</td>
<td>79.28</td>
</tr>
</tbody>
</table>

From Table 6.26 we see that the iteration counts for $P_E$ remain low and bounded under mesh refinement for all problem parameters. We observe a slight decrease in iteration counts for $Re \neq 0$. The log-log plot of the execution times from Table 6.27 for
Re = 0 and 200 is presented in Figure 6.18 with a line of gradient 1 for comparison. As expected $P_E$, leads to sub-optimal execution times. The gradients of the lines of best fit through the data points are approximately 1.67, suggesting the asymptotic behaviour of $O(N^{5/3})$, which is similar to what is observed in Example 6.2.1 (cf. Figure 6.8).

The iteration counts and execution times when $P_{LSC}$ is used are shown in Tables 6.28 and 6.29 respectively. Under mesh refinement the iteration counts decrease for $Re \neq 0$, thus we ran an additional test for $N = 590848$ to investigate further. We have not included the execution time for this test because swap memory was used. We observed that the iteration counts do start to increase very slightly with mesh refinement. The SDVT-NSE yields slightly lower iteration counts than the SVT-NSE but have higher execution times. This is expected, due to the additional nonzeros entries present in Jacobian derived from the SDVT-NSE. This is consistent with the results for Example 6.2.1 (cf. Table 6.19). Overall the iteration counts for $P_{LSC}$ remain low for the problem sizes considered.

The execution times for Re = 0 and 200 are presented in Figure 6.19 with a line of gradient 1 for comparison. As expected, the execution time is sub-optimal and scales approximately as $O(N^{1.5})$. This is similar to the results observed in Figure 6.10.

Now we evaluate the performance of $\hat{P}_{LSC}$ on Example 6.2.3. The iteration counts are given in Table 6.30. For the SVT-NSE the GMRES solver did not converge after
Table 6.28: The average number of iterations taken by the GMRES+PLSC solver applied to Example 6.2.3 as a function of the viscous term, ‘vis’, Reynolds number, ‘Re’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>SVT</th>
<th>SDVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2368</td>
<td>9344</td>
<td>37120</td>
<td>147968</td>
</tr>
<tr>
<td>590848</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>SVT</th>
<th>SDVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11.5</td>
<td>12.5</td>
<td>14</td>
</tr>
<tr>
<td>100</td>
<td>19.5</td>
<td>14.7</td>
<td>12.7</td>
</tr>
<tr>
<td>200</td>
<td>26</td>
<td>21.5</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>SVT</th>
<th>SDVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>100</td>
<td>13.7</td>
<td>11.7</td>
<td>11</td>
</tr>
<tr>
<td>200</td>
<td>19.2</td>
<td>13.7</td>
<td>11.7</td>
</tr>
</tbody>
</table>

Table 6.29: The execution times (s) taken by the GMRES+PLSC solver applied to Example 6.2.3 as a function of the viscous term, ‘vis’, Reynolds number, ‘Re’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>SVT</th>
<th>SDVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2368</td>
<td>9344</td>
<td>37120</td>
<td>147968</td>
</tr>
<tr>
<td>590848</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>SVT</th>
<th>SDVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.05</td>
<td>0.33</td>
<td>2.80</td>
</tr>
<tr>
<td>100</td>
<td>0.08</td>
<td>0.46</td>
<td>3.89</td>
</tr>
<tr>
<td>200</td>
<td>0.09</td>
<td>0.51</td>
<td>3.82</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>SVT</th>
<th>SDVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.07</td>
<td>0.48</td>
<td>4.03</td>
</tr>
<tr>
<td>100</td>
<td>0.07</td>
<td>0.47</td>
<td>4.00</td>
</tr>
<tr>
<td>200</td>
<td>0.08</td>
<td>0.48</td>
<td>4.00</td>
</tr>
</tbody>
</table>

Figure 6.19: Log-log plot of the execution times (s) taken by the GMRES+PLSC solver applied to Example 6.2.3 as a function of the viscous term, ‘vis’, Reynolds number, ‘Re’, and discrete problem size. The best fit line has gradient 1.5 and the line of slope 1 is for comparison.
200 iterations for $N = 9344$ and $Re = 200$ because the grid resolution was not fine enough to resolve the solution in this case. This phenomena is also observed in [180, p. 93, 68, p. 362] where for large $Re$, the mesh refinement leads to a reduction in iteration counts to a certain point. To better understand the convergence behaviour of the iteration counts we plot a log-log graph of the iteration counts as a function of the problem size in Figure 6.20. A straight line is observed for the data points in Figure 6.20 for larger problem sizes, for example if we exclude the first two data points. The three lines with gradient 0.25 are based on the last three data points. The data points for the problem parameters considered lie almost parallel to these lines, this suggests that the iteration counts grow as $O(N^{0.25})$.

The execution times for Reynolds numbers $Re = 0$ and 200 are depicted in Figure 6.21 with a line of gradient 1 for comparison. We do not observe linear scaling with respect to discrete problem size. We calculated the asymptotic behaviour of the execution times to be approximately $O(N^{1.23})$, which is only marginally worse than Example 6.2.1 (cf. Figure 6.13).

We conclude this section with a comparison of the execution times of direct solve (using SuperLU) and GMRES (using Trilinos AztecOO) preconditioned with $P_E$, $P_{LSC}$ and $\tilde{P}_{LSC}$ in Figure 6.22 where a log scale is not used so we can compare the execution times in absolute terms. For the solvers using SuperLU, the maximum problem size attainable is $N = 147968$. For comparable problem sizes, we see that direct solve and GMRES+$P_E$ has similar execution times, GMRES+$P_{LSC}$ takes half as long while GMRES+$\tilde{P}_{LSC}$ is much faster. These results are consistent with the execution times depicted in Figure 6.16.

---

Table 6.30: The average number of iterations taken by the GMRES+$\tilde{P}_{LSC}$ solver applied to Example 6.2.3 as a function of the viscous term, ‘vis’, Reynolds number, ‘Re’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>9344</th>
<th>37120</th>
<th>147968</th>
<th>590848</th>
<th>2361344</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>0</td>
<td>37</td>
<td>45.5</td>
<td>58.5</td>
<td>80</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>26.7</td>
<td>26.7</td>
<td>33</td>
<td>48</td>
<td>74.7</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>-</td>
<td>31</td>
<td>30.7</td>
<td>41.3</td>
<td>63</td>
</tr>
<tr>
<td>SDVT</td>
<td>0</td>
<td>24</td>
<td>29.5</td>
<td>42.5</td>
<td>64.5</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>17.3</td>
<td>20.7</td>
<td>26.3</td>
<td>37.3</td>
<td>58.3</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>18.7</td>
<td>18.3</td>
<td>24.3</td>
<td>33.3</td>
<td>48</td>
</tr>
</tbody>
</table>

---

\[5\text{ We have also plot a semi-log graph but a linear dependence of iteration counts on } N \text{ was not observed, which suggests an asymptotic behaviour greater than } O(\log N).\]
2.5
3
3.5
4
4.5
Log of iteration counts

2.5
3
3.5
4
4.5
Log of problem size

Figure 6.20: Log-log plot of the average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver applied to Example 6.2.3. For larger problem sizes, the iteration counts loosely follow the lines with a slope of 0.25.

2.5
3
3.5
4
4.5
Log of execution time

2.5
3
3.5
4
4.5
Log of problem size

Figure 6.21: Log-log plot of the execution times (s) taken by the GMRES+$\hat{P}_{LSC}$ solver applied to Example 6.2.3. The best fit line has gradient 1.23 and the line of slope 1 is for comparison.
6.3. THREE-DIMENSIONAL CASE STUDIES

In this section we consider two unsteady problems in three spatial dimensions: a flow through a cubic domain rotated by an arbitrary angle, and uniform radial flow through a quarter annulus. For both unsteady problems considered we set St = 1 and...
perform an impulsive start from rest \cite{99}, pp. 794–795]. An alternate approach would be to apply a gradual transition of the inflow BC from the state of rest into the final configuration \cite{229}. This avoids spurious oscillations generated by a time integration method due to sudden changes in the system configuration. We have not observed any instabilities in the solutions for both our unsteady test cases, thus an impulsive start is acceptable. The initial condition for the velocity field is $u(t = 0) = 0$. At subsequent time steps we use the solution computed at previous time steps as the initial guess for the Newton’s method. The adaptive BDF-2 time stepping method is used with the local truncation error tolerance of $\varepsilon_T = 10^{-4}$. This value is recommended in \cite{99}, p. 276]. In the time step size selection we use the relative root mean square (RMS) of the velocity local truncation error, $e_j (j = 1, \ldots N_u)$, where $N_u$ is the number of velocity nodes. Thus, the next time step size $\Delta t_{k+1}$ is estimated using the following formula

$$\Delta t_{k+1} = \Delta t_k \left( \frac{\varepsilon_T}{\sqrt{\frac{1}{N_u} \sum_{j=1}^{N_u} e_j^2}} \right)^{1/3}.$$  \hspace{1cm} (6.3.1)

When using the AMG approximation of the pressure Poisson operators in 3D, we increase the strength of dependence parameter to $\theta = 0.7$. This comes from the \texttt{oomph-lib} default AMG parameters for 3D Poisson problem, which was determined experimentally by Muddle \cite{180}. We summarise this in Preconditioner 6.6.

**Preconditioner 6.6:** A classical AMG preconditioner for 3D Poisson problems.

- **Preconditioner:** BoomerAMG
- **Coarsening:** Ruge-St"uben, $\theta = 0.7$
- **Smoothing:** Damped Jacobi, $\omega = 2/3$
- **Cycles:** $2 \times V(1,1)$

### 6.3.1 Flow through a Unit Cube Rotated by an Arbitrary Angle

We study the flow through a unit cube domain rotated by an arbitrary angle $\alpha$ (see Figure 6.24). The domain $\Omega^{(a)}$ is obtained by rotating the discrete points in the unit cube $\Omega = [0, 1]^3$ using the following transformation:

$$R_{xyz}(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix} \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix} \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix},$$  \hspace{1cm} (6.3.2)
where $\alpha$ is the angle of rotation. We define the boundary conditions for $\alpha = 0$. The inflow boundary is located at $\partial \Omega_I = [x_1 = 0, 0.5 < x_2, x_3 < 1]$, and the outflow boundary is located on $\partial \Omega_O = [x_1 = 1, 0 < x_2, x_3 < 0.5]$. The no-slip condition $u_i(x_1, x_2, x_3) = 0$ $(i = 1, 2, 3)$ is applied to all other points on the boundary, i.e. $(x_1, x_2, x_3) \in \partial \Omega \setminus \{\partial \Omega_I \cup \partial \Omega_O\}$. This is summarised in Figure 6.23.

![Figure 6.23: The domain, the inflow (\(\partial \Omega_I\)) and the outflow (\(\partial \Omega_O\)) boundaries for Example 6.3.1.](image)

Denote by $t_k$ the current time and define $C_k = -\cos(\pi t_k)/2 + 0.5$. Then, the inflow BC is defined by $u_1 = C_k(x_2 - 0.5)(1 - x_2)(x_3 - 0.5)(1 - x_3)$, $u_2 = u_3 = 0$ on $\partial \Omega_I$. One key difference between imposing parallel outflow with Lagrange multipliers in 2D and 3D is that in 3D we require two tangent vectors to define a surface. Thus, we impose the parallel outflow condition by defining two arbitrary but orthogonal tangent vectors $\hat{t}_1$ and $\hat{t}_2$ ($\hat{t}_1 \cdot \hat{t}_2 = 0$) at each discrete point on the boundary $\partial \Omega_O$, and require that $\mathbf{u} \cdot \hat{t}_1 = 0$ and $\mathbf{u} \cdot \hat{t}_2 = 0$ simultaneously. We adopt the initial time step size $\Delta t_0 = 0.1$.

The formal definition of the case study is summarised as follows:

**Example 6.3.1. Flow through a rotated unit cube.**

The domain: unit cube $\Omega \in [0, 1]^3$.

The time interval: $T = [0, 1]$.

Boundary conditions: for convenience, we present the boundary conditions for the non-rotated unit cube, $\alpha = 0$. In order to obtain the boundary conditions for $\alpha \neq 0$, we need to apply the rotation. Let $\partial \Omega$ be the boundary $\Omega$. The inflow and outflow

---

6It is unrealistic for Example 6.3.1 to have no tangential component of velocity at the outflow, however, it is considered as a test case and a proof of concept.
part of the boundary are located at
\[
\partial \Omega_I = \{x_1 = 0 \text{ and } 0.5 < x_2, x_3 < 1\}, \text{ and }
\partial \Omega_O = \{x_1 = 1 \text{ and } 0 < x_2, x_3 < 0.5\},
\]
respectively. Denote two orthogonal tangent vectors at each discrete point on \( \partial \Omega_O \) by \( \hat{t}_1 \) and \( \hat{t}_2 \), the current time by \( t_k \), and \( C_k = -\cos(\pi t_k)/2 + 0.5 \). Then the initial conditions and boundary conditions are:

\[
u_i = 0, \quad (x_1, x_2, x_3) \in \Omega, t = 0,
\]

\[
u_i = 0, \quad (x_1, x_2, x_3) \in \partial \Omega \setminus \{\partial \Omega_I \cup \partial \Omega_O\},
\]

\[
u_1 = C_k(x_2 - 0.5)(1 - x_2)(x_3 - 0.5)(1 - x_3), \quad u_2 = 0, \quad u_3 = 0, \quad (x_1, x_2, x_3) \in \partial \Omega_I, t_k \in T,
\]

\[
u \cdot \hat{t}_1 = 0, \quad \nu \cdot \hat{t}_2 = 0, \quad (x_1, x_2, x_3) \in \partial \Omega_O.
\]

for \( i = 1, 2, 3 \) and \( t_k \in [0, 1] \). An impulsive start from rest is performed with an initial time step of \( \Delta t_0 = 0.1 \). On average, the four time steps were taken, thus \( \Delta t_k \approx 0.25 \).

Figure 6.24: Plot of the solution for the unsteady flow with \( \text{Re} = 100 \) through a unit cube rotated by \( \alpha = 30^\circ \) at \( t = 0.35 \). The colour contours and streamlines represent the pressure gradient and velocity flow field, respectively.
AMG Calibration

First we study the application of AMG to the approximate inversion of the augmented momentum block $\hat{F}$ in $\hat{P}_{LSC}$. For these tests we start from an initial guess of zero and adopt the relative tolerance $\varepsilon_K = 10^{-6}$ as a stopping criterion for GMRES. The mesh sizes are $h = 1/\ell$ for $\ell = 4, 6, \ldots, 20$, which correspond to the discrete problem sizes between $N = 1199$ and $N = 189023$. We use an angle of rotation $\alpha = 30^\circ$.

We test the same four preconditioning strategies as before (cf. Section 6.2), AMG inverse of the entire block $\hat{F}$, block diagonal and upper/lower block triangular approximation. We perform these tests for the SVT-NSE with $Re = 100$. We use a strength of dependence of $\theta = 0.5$ as it is recommended for 3D problems in [131, p. 168]. For this test, the AMG parameters used are as follows:

- **Momentum preconditioner**: Inexact inversion by BoomerAMG
  - **Coarsening**: Ruge-Stüben (strength of dependence $\theta = 0.5$)
  - **Smoothing**: Gauss-Seidel
  - **Cycle**: $1 \times V(2, 2)$

The pressure Poisson operators are approximated with the AMG settings defined by Preconditioner 6.6 and SuperLU is used for the $W$ block. The iteration counts and execution times are depicted in Figures 6.25 and 6.26 respectively. We see

![Figure 6.25: Semi-log plot of the average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver using different block approximations of $\hat{F}$ in the LSC preconditioner, applied to Example 6.3.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and $Re = 100$. BD = block diagonal, BUT/BLT = block upper/lower triangular, Entire = the entire $\hat{F}$ block.](image-url)
Figure 6.26: Log-log plot of the execution times (s) taken by the GMRES+$\hat{P}_{LSC}$ solver using different block approximations of $\hat{F}$ in the LSC preconditioner, applied to Example 6.3.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and Re = 100. BD = block diagonal, BUT/BLT = block upper/lower triangular, Entire = the entire $\hat{F}$ block.

from Figure 6.25 that the iteration counts for the different block approximations exhibit different asymptotic behaviour. The iteration counts for the block diagonal approximation grow faster than $O(\log N)$, while upper/lower block triangular approximations grow as $O(\log N^{15.4})$ (based on the best fit line). When using the entire block we observe significantly lower iteration counts, and we estimate the asymptotic behaviour to be $O(\log N^{5.7})$. Despite a significant difference in iteration counts, all four approximations of the inverse of $\hat{F}$ have similar asymptotic behaviour in execution times, see Figure 6.26. We have tabulated the execution times for the largest five problem sizes in Table 6.31 from which we observe that application of AMG on the entire $\hat{F}$ block produces the lowest execution times. Thus, for the remainder for the AMG calibration tests we apply AMG to the entire augmented momentum block. This is consistent with our findings for 2D problems (cf. Tables 6.3 and 6.4).

**Strength of dependence, $\theta$:** Next we experiment with the strength of dependence. We consider both the SVT-NSE and SDVT-NSE with Re = 100 and $\theta = 0.5, 0.668$ and 0.75. The iteration counts for the largest feasible five problem sizes are presented in Tables 6.32 and 6.33 for the SVT-NSE and SDVT-NSE, respectively. The corresponding execution times are given in Tables 6.34 and 6.35.
6.3. THREE-DIMENSIONAL CASE STUDIES

Table 6.31: The execution times (s) taken by the GMRES+\(\hat{P}_{LSC}\) solver using different block approximations of \(\hat{F}\) in the LSC preconditioner, applied to Example 6.3.1. The results are presented for the SVT-NSE with \(\alpha = 30^\circ\) and \(Re = 100\). BD = block diagonal, BUT/BLT = block upper/lower triangular, Entire = the entire \(\hat{F}\) block.

<table>
<thead>
<tr>
<th>Prec</th>
<th>BD</th>
<th>BUT</th>
<th>BLT</th>
<th>Entire</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22.27</td>
<td>21.13</td>
<td>19.56</td>
<td>17.08</td>
</tr>
<tr>
<td></td>
<td>49.13</td>
<td>38.68</td>
<td>40.65</td>
<td>31.62</td>
</tr>
<tr>
<td></td>
<td>70.78</td>
<td>64.65</td>
<td>55.87</td>
<td>55.98</td>
</tr>
<tr>
<td></td>
<td>105.07</td>
<td>91.64</td>
<td>88.95</td>
<td>77.07</td>
</tr>
<tr>
<td></td>
<td>181.48</td>
<td>138.90</td>
<td>148.35</td>
<td>118.52</td>
</tr>
</tbody>
</table>

Table 6.32: The average number of iterations taken by the GMRES+\(\hat{P}_{LSC}\) solver for varying strength of dependence parameter \(\theta\) in the AMG approximation of the \(\hat{F}\) block within the LSC preconditioner, applied to Example 6.3.1. The results are presented for the SVT-NSE with \(\alpha = 30^\circ\) and \(Re = 100\).

<table>
<thead>
<tr>
<th>(\theta)</th>
<th>0.5</th>
<th>0.67</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td>39303</td>
<td>48.5</td>
<td>49.8</td>
<td>49.2</td>
</tr>
<tr>
<td>63269</td>
<td>51.2</td>
<td>53</td>
<td>53.5</td>
</tr>
<tr>
<td>95411</td>
<td>56.5</td>
<td>58.2</td>
<td>58.2</td>
</tr>
<tr>
<td>136929</td>
<td>58.8</td>
<td>62.5</td>
<td>62.8</td>
</tr>
<tr>
<td>189023</td>
<td>62.8</td>
<td>66.5</td>
<td>66.8</td>
</tr>
</tbody>
</table>

Table 6.33: The average number of iterations taken by the GMRES+\(\hat{P}_{LSC}\) solver for varying strength of dependence parameter \(\theta\) in the AMG approximation of the \(\hat{F}\) block within the LSC preconditioner, applied to Example 6.3.1. The results are presented for the SDVT-NSE with \(\alpha = 30^\circ\) and \(Re = 100\).

<table>
<thead>
<tr>
<th>(\theta)</th>
<th>0.5</th>
<th>0.67</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td>39303</td>
<td>53</td>
<td>56.8</td>
<td>55.8</td>
</tr>
<tr>
<td>63269</td>
<td>56.5</td>
<td>59.8</td>
<td>60.2</td>
</tr>
<tr>
<td>95411</td>
<td>59</td>
<td>65</td>
<td>66.5</td>
</tr>
<tr>
<td>136929</td>
<td>62.8</td>
<td>66.5</td>
<td>68</td>
</tr>
<tr>
<td>189023</td>
<td>67.8</td>
<td>70.2</td>
<td>73</td>
</tr>
</tbody>
</table>

Table 6.34: The execution times (s) taken by the GMRES+\(\hat{P}_{LSC}\) solver for varying strength of dependence parameter \(\theta\) in the AMG approximation of the \(\hat{F}\) block within the LSC preconditioner, applied to Example 6.3.1. The results are presented for the SVT-NSE with \(\alpha = 30^\circ\) and \(Re = 100\).

<table>
<thead>
<tr>
<th>(\theta)</th>
<th>0.5</th>
<th>0.67</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td>39303</td>
<td>13.58</td>
<td>14.08</td>
<td>15.51</td>
</tr>
<tr>
<td>63269</td>
<td>28.39</td>
<td>22.45</td>
<td>28.31</td>
</tr>
<tr>
<td>95411</td>
<td>44.39</td>
<td>37.45</td>
<td>41.67</td>
</tr>
<tr>
<td>136929</td>
<td>65.89</td>
<td>63.18</td>
<td>76.79</td>
</tr>
<tr>
<td>189023</td>
<td>98.77</td>
<td>96.11</td>
<td>100.99</td>
</tr>
</tbody>
</table>

Table 6.35: The execution times (s) taken by the GMRES+\(\hat{P}_{LSC}\) solver for varying strength of dependence parameter \(\theta\) in the AMG approximation of the \(\hat{F}\) block within the LSC preconditioner, applied to Example 6.3.1. The results are presented for the SDVT-NSE with \(\alpha = 30^\circ\) and \(Re = 100\).

<table>
<thead>
<tr>
<th>(\theta)</th>
<th>0.5</th>
<th>0.67</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td>39303</td>
<td>57.17</td>
<td>28.88</td>
<td>21.03</td>
</tr>
<tr>
<td>63269</td>
<td>76.20</td>
<td>43.19</td>
<td>37.81</td>
</tr>
<tr>
<td>95411</td>
<td>135.36</td>
<td>81.10</td>
<td>58.47</td>
</tr>
<tr>
<td>136929</td>
<td>165.49</td>
<td>110.76</td>
<td>88.47</td>
</tr>
<tr>
<td>189023</td>
<td>241.87</td>
<td>151.37</td>
<td>161.70</td>
</tr>
</tbody>
</table>
For both SVT-NSE and SDVT-NSE we observe that $\theta = 0.5$ yields the lowest iteration counts, this comes at the expense of higher execution times as seen in Tables 6.34 and 6.35. The iteration counts for the SDVT-NSE exhibit continuous increase when $\theta$ is increased, and there is no sudden drop observed at $\theta = 2/3$ as in the 2D case (cf. Figures 6.2 and 6.3). For $\theta = 0.67$ and 0.75 we plot a semi-log graph for the iteration counts and log-log graph for the execution times in Figures 6.27 and 6.28, respectively. In Figure 6.27 we also plot the lines of gradient 9.1 and 9.9 for SVT and SDVT, respectively. The asymptotic behaviour of the iteration is slightly faster for the SDVT then for SVT, with similar trends holding for $\theta = 0.668$ and 0.75. We estimated the asymptotic growth in the execution times to be approximately $O(N^{1.27})$. Thus, we adopt $\theta = 0.668$ for both the SVT-NSE and SDVT-NSE.

Smoothing method: We compare the performance when using three standard smoothers. Jacobi with a damping factor of $\omega = 0.5$ (this value was determined experimentally to give the lowest execution times), Gauss-Seidel, and Euclid ILU(0). We tested one and two V(2,2) cycles. As with the previous experiments, the sparse pressure Poisson operators are approximated by AMG with the parameters defined by Preconditioner 6.6. The AMG parameters for the approximation of the augmented
momentum block $\hat{F}$ are as follows:

- **Momentum preconditioner**: Inexact inversion by BoomerAMG
  - Coarsening: Ruge-Stüben, $\theta = 0.668$
  - Smoothing: Jacobi ($\omega = 0.5$), Gauss-Seidel, Euclid ILU(0)
  - Cycle: $1 \times V(2, 2)$, $2 \times V(2, 2)$

First we consider the SVT-NSE with Reynolds number $Re = 100$ and an angle of rotation $\alpha = 30^\circ$. The average iteration counts required for the GMRES solver to converge to a relative tolerance of $\varepsilon_K = 10^{-6}$ from an initial guess of zero are depicted in Figure 6.29 and the execution times are illustrated in Figure 6.30. From Figure 6.29 we observe that only $1 \times V(2, 2)$ cycles of Euclid ILU(0) exhibit $O(\log N)$ growth. Although the $1 \times V(2, 2)$ cycles of Euclid ILU(0) smoother yields the lowest iteration counts, from Figure 6.30 we observe considerably higher execution times. In this case, this is expected, as the stencil sizes both at the finest and all coarse levels in the AMG hierarchy are much larger in 3D, than in 2D when Ruge-Stüben coarsening is used [33] (for coarse levels this particularly applies for smaller values of $\theta$). This is documented in Table 6.36 where we observe that using $1 \times V(2, 2)$ cycles of Euclid ILU(0) smoother yield execution times more than nine times longer than when $1 \times V(2, 2)$ cycles of Gauss-Seidel smoothing is used. We choose $1 \times V(2, 2)$ cycles with
Comparison of smoothing methods for the SVT of the NSE

![Graph showing iteration counts vs. log of problem size](image)

Figure 6.29: Semi-log plot of the average number of iterations taken by the GMRES+PLSC solver with different smoothers and different V-cycles used for the AMG approximation of the $F$ block, applied to Example 6.3.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and Re = 100.

Comparison of smoothing methods for the SVT of the NSE

![Graph showing execution times vs. log of problem size](image)

Figure 6.30: Log-log plot of the execution times (s) taken by the GMRES+PLSC solver with different smoothers and different V-cycles used for the AMG approximation of the $F$ block, applied to Example 6.3.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and Re = 100.
6.3. THREE-DIMENSIONAL CASE STUDIES

Table 6.36: The execution times (s) taken by the GMRES+\(\hat{P}_{LSC}\) solver with different smoothers and different \(V\)–cycles used for the AMG approximation of the \(\hat{F}\) block, applied to Example 6.3.1. The results are presented for the SVT-NSE with \(\alpha = 30^\circ\) and \(Re = 100\).

<table>
<thead>
<tr>
<th>Smoother</th>
<th>Cycle</th>
<th>39303</th>
<th>63269</th>
<th>95411</th>
<th>136929</th>
<th>189023</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi, (\omega = 0.5)</td>
<td>1 (\times V(2, 2))</td>
<td>23.06</td>
<td>37.28</td>
<td>70.39</td>
<td>88.51</td>
<td>132.25</td>
</tr>
<tr>
<td>Jacobi, (\omega = 0.5)</td>
<td>2 (\times V(2, 2))</td>
<td>23.19</td>
<td>46.05</td>
<td>66.43</td>
<td>108.22</td>
<td>179.57</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>1 (\times V(2, 2))</td>
<td>13.91</td>
<td>26.98</td>
<td>38.32</td>
<td>64.50</td>
<td>98.70</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>2 (\times V(2, 2))</td>
<td>16.21</td>
<td>28.53</td>
<td>49.29</td>
<td>93.19</td>
<td>108.24</td>
</tr>
<tr>
<td>Euclid ILU(0)</td>
<td>1 (\times V(2, 2))</td>
<td>160.50</td>
<td>255.82</td>
<td>423.42</td>
<td>698.08</td>
<td>914.11</td>
</tr>
</tbody>
</table>

Gauss-Seidel smoothing for the SVT-NSE.

Now we consider the SDVT-NSE. The iteration counts for different smoothers and different \(V\)–cycles are depicted in the semi-log plot Figure 6.31, whilst the corresponding execution times are shown in the log-log graph Figure 6.32. We observe that the iteration counts for 1 \(\times V(2, 2)\) cycles with Jacobi smoothing are considerably worse than for the other smoothers considered. The near constant slopes observed for 2 \(\times V(2, 2)\) Jacobi smoothing and Gauss-Seidel (for both smoothing cycles considered) suggests that the iteration counts grow as \(O(\log N)\). The growth rate in iteration counts when the ILU(0) smoother is used is slower than for the other four cases, but at the expense of by far the longest execution times, see Table 6.37. Similar asymptotic behaviour is observed for the other smoothers with both variants of Gauss-Seidel.

Figure 6.31: Semi-log plot of the average number of iterations taken by the GMRES+\(\hat{P}_{LSC}\) solver with different smoothers and different \(V\)–cycles used for the AMG approximation of the \(\hat{F}\) block, applied to Example 6.3.1. The results are presented for the SDVT-NSE with \(\alpha = 30^\circ\) and \(Re = 100\).
smoothed being superior to their Jacobi counterparts. For the problem sizes considered, $1 \times V(2, 2)$ cycles with Gauss-Seidel smoothing yield the lowest execution times. Thus we adopt $1 \times V(2, 2)$ cycles of Gauss-Seidel smoothing for the SDVT-NSE. We formalise these findings in Preconditioner 6.7.

![Comparison of smoothing methods for the SDVT of the NSE](image)

**Figure 6.32:** Log-log plot of the execution times (s) taken by the GMRES+$\hat{P}_{LSC}$ solver with different smoothers and different $V-$cycles used for the AMG approximation of the $\hat{F}$ block, applied to Example 6.3.1. The results are presented for the SDVT-NSE with $\alpha = 30^\circ$ and $Re = 100$.

<table>
<thead>
<tr>
<th>Smoother</th>
<th>Cycle</th>
<th>39303</th>
<th>63269</th>
<th>95411</th>
<th>136929</th>
<th>189023</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi, $\omega = 0.5$</td>
<td>$1 \times V(2, 2)$</td>
<td>40.38</td>
<td>62.19</td>
<td>87.39</td>
<td>154.75</td>
<td>241.36</td>
</tr>
<tr>
<td>Jacobi, $\omega = 0.5$</td>
<td>$2 \times V(2, 2)$</td>
<td>46.35</td>
<td>79.95</td>
<td>100.91</td>
<td>157.33</td>
<td>249.35</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>$1 \times V(2, 2)$</td>
<td>31.20</td>
<td>44.63</td>
<td>54.63</td>
<td>113.83</td>
<td>166.16</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>$2 \times V(2, 2)$</td>
<td>37.08</td>
<td>52.05</td>
<td>71.23</td>
<td>130.74</td>
<td>174.16</td>
</tr>
<tr>
<td>Euclid ILU(0)</td>
<td>$1 \times V(2, 2)$</td>
<td>368.39</td>
<td>560.41</td>
<td>690.59</td>
<td>1089.83</td>
<td>1563.39</td>
</tr>
</tbody>
</table>

**Table 6.37:** The execution times (s) taken by the GMRES+$\hat{P}_{LSC}$ solver with different smoothers and different $V-$cycles used for the AMG approximation of the $\hat{F}$ block, applied to Example 6.3.1. The results are presented for the SDVT-NSE with $\alpha = 30^\circ$ and $Re = 100$. 
Preconditioner 6.7: $\hat{P}_{LSC}$, An inexact LSC Augmentation Preconditioner for 3D problems.

**Preconditioner:** Block diagonal  
**Navier-Stokes Preconditioner:**  LSC  
**Momentum Preconditioner:** Inexact inversion by BoomerAMG  
**Coarsening:** Ruge-Stüben, $\theta = 0.668$  
**Cycle/Smoothing:** $1 \times V(2, 2)$ with Gauss-Seidel smoothing.  
**Pressure Poisson Operators:** Inexact inversion by BoomerAMG  
**Coarsening:** Ruge-Stüben, $\theta = 0.7$  
**Smoothing:** Damped Jacobi, $\omega = \frac{2}{3}$  
**Cycle:** $2 \times V(1, 1)$  
**W-block Preconditioner:** Diagonal preconditioner  
**Subsidiary Preconditioner:** Exact inversion by SuperLU

Convergence Evaluation

In this section we numerically evaluate the performance of the different versions of the augmented preconditioner $P_E$, $P_{LSC}$, and $\hat{P}_{LSC}$. We perform experiments on Example 6.3.1 with the following configurations:

- Both the SVT-NSE and SDVT-NSE.
- Angle of rotation $\alpha = 0^\circ$ and $67^\circ$ degrees.
- Reynolds number $Re = 100, 200$ and $500$.
- Several uniform mesh refinements. The domain is discretised using uniform quadrilateral grids with discrete problem sizes between $N = 1199$ (mesh size $h = 1/4$) and $N = 63269$ (mesh size $h = 1/14$) unknowns for $P_E$ and $P_{LSC}$ versions for the preconditioner and $N = 950609$ ($h = 1/34$) for $\hat{P}_{LSC}$ version of the preconditioner. This is the maximum problem size we could run with 8GB of RAM without using the swap memory.
- Preconditioned GMRES solver with a relative convergence tolerance of $\varepsilon_K = 10^{-6}$ starting from an initial guess of zero.
- Adaptive time stepping using BDF-2 method with a local truncation error tolerance $\varepsilon_T = 10^{-4}$. Starting with from an initial solution of zero, at each new time step we use the solution computed at the previous time step as the initial guess for Newton’s method.

First we consider the performance of $P_E$, the iteration counts are shown in Table 6.38 and we present a log-log plot of the execution times for both the SVT-NSE and SDVT-NSE and $Re = 100$ and $500$ in Figure 6.33 with a line of gradient 1 for comparison. As observed before, under mesh refinement, $P_E$ yields very low and
bounded iteration counts. The increase in Reynolds number leads to a negligible increase in iteration counts (fewer than one iteration on average). From Figure 6.33 we computed the asymptotic behaviour of the execution times to be $O(N^{2.1})$, that is, the iterative solver exhibits approximately quadratic asymptotic behaviour.

Next we examine the performance of $\mathcal{P}_{LSC}$, the iteration counts are tabulated in Table 6.39. We observe negligible dependence in iteration counts with respect to the angle of rotation. The observations here are similar to the 2D test cases: an increase in Reynolds number leads to a decrease in iteration counts, and SDVT-NSE

<table>
<thead>
<tr>
<th>vis</th>
<th>$\alpha$</th>
<th>Re</th>
<th>1199</th>
<th>4461</th>
<th>11099</th>
<th>22313</th>
<th>39303</th>
<th>63269</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0°</td>
<td>200</td>
<td>5</td>
<td>6</td>
<td>6.6</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>SVT</td>
<td>0°</td>
<td>100</td>
<td>5.2</td>
<td>6.3</td>
<td>7.2</td>
<td>7.2</td>
<td>7.2</td>
<td>7.2</td>
</tr>
<tr>
<td></td>
<td>67°</td>
<td>200</td>
<td>5</td>
<td>6</td>
<td>6.6</td>
<td>7</td>
<td>7</td>
<td>7.2</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>5.2</td>
<td>6.3</td>
<td>7.2</td>
<td>7.2</td>
<td>7.2</td>
<td>7.2</td>
<td>7.2</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>4.6</td>
<td>5.4</td>
<td>6.3</td>
<td>6.6</td>
<td>6.7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>SDVT</td>
<td>0°</td>
<td>100</td>
<td>5.3</td>
<td>6.2</td>
<td>6.4</td>
<td>6.4</td>
<td>6.5</td>
<td>6.8</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>5.1</td>
<td>5.7</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>6.5</td>
<td>6.5</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>4.9</td>
<td>5.3</td>
<td>5.6</td>
<td>6.4</td>
<td>6.6</td>
<td>6.7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>5.3</td>
<td>6</td>
<td>6</td>
<td>6.2</td>
<td>6.2</td>
<td>6.2</td>
<td>6.2</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>4.9</td>
<td>5.3</td>
<td>5.6</td>
<td>6.1</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
</tr>
</tbody>
</table>

*Table 6.38: The average number of iterations taken by the GMRES+$\mathcal{P}_E$ solver applied to Example 6.3.1, as a function of the viscous term, ‘vis’, angle of rotation ‘$\alpha$’, Reynolds number, ‘Re’, and discrete problem size.*

*Figure 6.33: Log-log plot of the execution times (s) taken by the GMRES+$\mathcal{P}_E$ solver applied to Example 6.3.1. The results are presented for $\alpha = 67^\circ$. The best fit line has gradient 2.1 and the line of slope 1 is for comparison.*
generally yields lower iteration counts than their SVT-NSE counter-pairs. In order to study the asymptotic behaviour of $P_{LSC}$ we present a semi-log plot of the iteration counts as a function of the log of the problem size $N$ in Figure 6.34 for problem parameters $\alpha = 67^\circ$ and $Re = 100, 200$. The line with gradient 5.2 is superimposed, which represents the best fit of the first four data points from the SVT-NSE and $Re = 100$. This demonstrates that the slope of the curves are becoming less steep for larger problem sizes. Thus, the asymptotic growth in iteration counts is likely to be sub-logarithmic.

![Figure 6.34: Semi-log plot of the average number of iterations taken by the GMRES+$P_{LSC}$ solver applied to Example 6.3.1 as a function of the log of the problem size. The results are presented for $\alpha = 67^\circ$. The line of slope 5.2 is for comparison.](image)

<table>
<thead>
<tr>
<th>vis</th>
<th>$\alpha$</th>
<th>Re</th>
<th>1199</th>
<th>4461</th>
<th>11099</th>
<th>22313</th>
<th>39303</th>
<th>63269</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>0$^\circ$</td>
<td>100</td>
<td>12.6</td>
<td>19.5</td>
<td>24.2</td>
<td>27.6</td>
<td>29.2</td>
<td>30.8</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>12.6</td>
<td>18.6</td>
<td>22.2</td>
<td>25.2</td>
<td>27.2</td>
<td>28.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>12.3</td>
<td>17.7</td>
<td>20.9</td>
<td>23.1</td>
<td>24.6</td>
<td>26.3</td>
<td></td>
</tr>
<tr>
<td>SDVT</td>
<td>0$^\circ$</td>
<td>100</td>
<td>12.6</td>
<td>19.5</td>
<td>24.2</td>
<td>27.6</td>
<td>29.2</td>
<td>30.8</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>12.6</td>
<td>18.5</td>
<td>22.2</td>
<td>25.2</td>
<td>27.2</td>
<td>28.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>12.3</td>
<td>17.7</td>
<td>20.9</td>
<td>23.1</td>
<td>24.8</td>
<td>26.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>67$^\circ$</td>
<td>100</td>
<td>12.8</td>
<td>19.5</td>
<td>23.4</td>
<td>25.4</td>
<td>26.5</td>
<td>28.7</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>12.4</td>
<td>18.3</td>
<td>21.6</td>
<td>24</td>
<td>25.4</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>12.4</td>
<td>17.6</td>
<td>20.3</td>
<td>23.1</td>
<td>23.9</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>67$^\circ$</td>
<td>100</td>
<td>12.7</td>
<td>19.2</td>
<td>22.8</td>
<td>25.6</td>
<td>27.2</td>
<td>27.5</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>12.3</td>
<td>18.2</td>
<td>22.2</td>
<td>24.6</td>
<td>26</td>
<td>26.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>12.3</td>
<td>17.6</td>
<td>21.1</td>
<td>23.4</td>
<td>24.7</td>
<td>25.7</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.39: The average number of iterations taken by the GMRES+$P_{LSC}$ solver applied to Example 6.3.1 as a function of the viscous term, ‘vis’, angle of rotation ‘$\alpha$’, Reynolds number, ‘$Re$’, and discrete problem size.
The execution times for problem parameters $\alpha = 67^\circ$ and $Re = 100, 200$, with a line of gradient 1 are shown in the log-log plot Figure 6.35. As expected, we do not see linear growth in execution times with respect to the discrete problem size. Using the gradients of the best fit lines we compute the asymptotic growth of execution time to be $O(N^{1.94})$, again approximately quadratic.

Lastly, we consider the convergence properties of $\hat{P}_{LSC}$ as specified by Preconditioner 6.7. In this case we consider mesh sizes between $h = \frac{1}{4}$ and $h = \frac{1}{31}$, giving discrete problem sizes between $N = 4461$ and $N = 950609$. The iteration counts for $\hat{P}_{LSC}$ are presented in Table 6.40. General observations are similar to the $P_{LSC}$ case, the SDVT-NSE yields lower iteration counts than the SVT-NSE for the same problem parameters $N$ and $Re$. As the Reynolds number increases, a decrease in iteration count is observed. We do observe a significant dependence on the angle of rotation, as in the 2D case (cf. Table 6.23). This effect is only observed when using the AMG approximation for the $\hat{F}$ block. We examine this in more detail. Suppose that the two unit tangent vectors $\hat{t}_1$ and $\hat{t}_2 \in \mathbb{R}^3$ at each discrete point on the outflow boundary
Table 6.40: The average number of iterations taken by the GMRES+\(\tilde{P}_{LSC}\) solver applied to Example 6.3.1, as a function of the viscous term, ‘vis’, angle of rotation ‘\(\alpha\)’, Reynolds number, ‘\(\text{Re}\)’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>(\alpha)</th>
<th>\text{Re}</th>
<th>11099</th>
<th>39303</th>
<th>189023</th>
<th>950609</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>0°</td>
<td>100</td>
<td>25.2</td>
<td>29.8</td>
<td>35.8</td>
<td>42.5</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>23.6</td>
<td>27.4</td>
<td>32.6</td>
<td>40</td>
<td>45.5</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>22.1</td>
<td>26.1</td>
<td>29.4</td>
<td>36.2</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>24.7</td>
<td>33</td>
<td>42.8</td>
<td>59.3</td>
<td>77.5</td>
</tr>
<tr>
<td></td>
<td>67°</td>
<td>24.2</td>
<td>29.4</td>
<td>37</td>
<td>52.8</td>
<td>71.2</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>24.6</td>
<td>28.4</td>
<td>33.3</td>
<td>43.6</td>
<td>60.5</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>24.6</td>
<td>28.4</td>
<td>33.3</td>
<td>43.6</td>
<td>60.5</td>
</tr>
<tr>
<td>SDVT</td>
<td>0°</td>
<td>100</td>
<td>25.2</td>
<td>29.2</td>
<td>32.5</td>
<td>38.5</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>23.9</td>
<td>27.2</td>
<td>31.2</td>
<td>37.5</td>
<td>42.5</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>22.6</td>
<td>25.7</td>
<td>29.1</td>
<td>34.6</td>
<td>39.2</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>22.6</td>
<td>25.7</td>
<td>29.1</td>
<td>34.6</td>
<td>39.2</td>
</tr>
<tr>
<td></td>
<td>67°</td>
<td>28.8</td>
<td>37.2</td>
<td>47.2</td>
<td>60.2</td>
<td>81.2</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>27.5</td>
<td>34.4</td>
<td>44</td>
<td>56.8</td>
<td>73.2</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>26</td>
<td>31.4</td>
<td>39.5</td>
<td>50.8</td>
<td>63.8</td>
</tr>
</tbody>
</table>

Then according to (4.3.6), (4.3.7) and (4.3.15), a constrained two-dimensional boundary leads to the matrix \(L\) with the following block structure

\[
L(\alpha) = \begin{bmatrix}
O & t_{11}(\alpha)M & O & t_{12}(\alpha)M & O & t_{13}(\alpha)M & O \\
O & t_{21}(\alpha)M & O & t_{22}(\alpha)M & O & t_{23}(\alpha)M & O
\end{bmatrix},
\]

(6.3.4)

where \(M\) is the mass matrix defined on the constrained velocity space. Taking \(\sigma = 1\) for simplicity, the matrix \(W\) has block structure:

\[
W = \begin{bmatrix}
(t_{11}^2 + t_{12}^2 + t_{13}^2)M^2 & (t_{11}t_{21} + t_{12}t_{22} + t_{13}t_{23})M^2 \\
(t_{11}t_{21} + t_{12}t_{22} + t_{13}t_{23})M^2 & (t_{21}^2 + t_{22}^2 + t_{23}^2)M^2
\end{bmatrix},
\]

(6.3.5)

where we have omitted the dependence on \(\alpha\) for notational simplicity. To determine the block structure of the perturbation matrix \(L^T\hat{W}^{-1}L\) employed in our implementation, we take the block diagonal approximation of (6.3.5):

\[
\hat{W}_{BD} = \begin{bmatrix}
(t_{11}^2 + t_{12}^2 + t_{13}^2)M^2 & O \\
O & (t_{21}^2 + t_{22}^2 + t_{23}^2)M^2
\end{bmatrix}.
\]

(6.3.6)
Then the perturbation matrix $L^T \hat{W}^{-1} L$ will have the same block structure with values of similar magnitude and same sign as

$$L^T \hat{W}^{-1}_{BD} L = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & d_1 I & b_3 I & 0 & b_2 I & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & b_3 I & d_2 I & 0 & b_1 I & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & b_2 I & 0 & b_1 I & 0 & d_3 I \\
0 & 0 & 0 & 0 & 0 & 0 
\end{bmatrix}, \quad (6.3.7)$$

where the values at the $d_j$ diagonals ($j = 1, 2, 3$) are:

$$d_1 = \frac{2t_1^2 t_2^2 + a_6 + a_4 + a_5 + a_3}{a_7}, \quad (6.3.8)$$

$$d_2 = \frac{a_6 + a_5 + 2t_2^2 t_2^2 + a_2 + a_1}{a_7} \quad \text{and} \quad (6.3.9)$$

$$d_3 = \frac{a_4 + a_2 + a_3 + a_1 + 2t_3^2 t_2^2}{a_7}, \quad (6.3.10)$$

while values at the $b_j$ diagonals ($j = 1, 2, 3$) are:

$$b_1 = \frac{t_1^2 t_2^2 t_3 + t_2^2 t_2^2 t_3 + t_1^2 t_3^2 t_2^2 + t_2^2 t_1^2 t_3^2 + t_2^2 t_1^2 t_3^2 + t_3^2 t_2^2 t_1^2}{a_7}, \quad (6.3.11)$$

$$b_2 = \frac{t_1^2 t_1^2 t_2^3 + t_1^2 t_3^2 t_2^3 + t_1^2 t_1^2 t_2^3 + t_2^2 t_2^2 t_1^3 + t_2^2 t_2^2 t_2^3 + t_2^2 t_3^2 t_2^3}{a_7} \quad \text{and} \quad (6.3.12)$$

$$b_3 = \frac{t_1^2 t_1^2 t_2^3 + t_1^2 t_1^2 t_3^2 + t_1^2 t_1^2 t_2^3 + t_2^2 t_2^2 t_1^3 + t_2^2 t_3^2 t_2^3 + t_2^2 t_3^2 t_3^2}{a_7}, \quad (6.3.13)$$

with $a_1 = t_1^2 t_2^2$, $a_2 = t_1^2 t_2^2$, $a_3 = t_1^2 t_2^2$, $a_4 = t_1^2 t_2^2$, $a_5 = t_1^2 t_2^2$, $a_6 = t_1^2 t_2^2$, and $a_7 = (t_1^2 + t_2^2 + t_3^2)(t_2^2 + t_2^2 + t_3^2)$. The components of the tangent vectors $t_{sj}$, $(s = 1, 2, j = 1, 2, 3)$ are more difficult to determine analytically in 3D, because the two tangent vectors are rotated with respect to all three Cartesian axes. Instead, we output the values $t_{sj}$ computed by the code. In Figure 6.36 we superimpose the iteration counts for the SVT-NSE with $Re = 100$ (using $\sigma = ||F||_\infty$), and the scaled perturbation values $d_i$, $b_j$ ($\sigma = 1$) $i, j = 1, 2, 3$. Firstly, we note that the values $d_j > 0$, $(j = 1, 2, 3)$. Therefore, adding them to the diagonal of the momentum block will make the matrix more diagonally dominant, and hence have a beneficial effect to the performance of the smoothers in AMG. The peaks in iteration counts are located at
6.3. THREE-DIMENSIONAL CASE STUDIES

Figure 6.36: Plot of the scaled perturbation values ($\sigma = 1$) together with the average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver (with $\sigma = \|F\|_\infty$) applied to Example 6.3.1 as a function of the angle of rotation $\alpha \in [0^\circ, 180^\circ]$ in increments of $5^\circ$. The results are presented for the SVT-NSE with $Re = 100$ and $N = 589824$. The values added to the diagonal blocks are in solid lines and the values added to the off-diagonal blocks in dashed lines.

$\alpha \approx 40^\circ$ and $\alpha \approx 145^\circ$. In the case $\alpha \approx 40^\circ$ we have $b_1, b_2 > 0$, $b_3 < 0$. This implies that negative numbers are added to the off-diagonal blocks $(4, 2)$ and $(2, 4)$, whilst positive numbers are added to the off-diagonal blocks $(2, 6)$, $(6, 2)$, $(4, 6)$ and $(6, 4)$. The increased proportion of positive numbers added to the off-diagonals appear to cause a decrease in the effectiveness of the AMG interpolation and coarsening procedures. The same observation can be made for the case $\alpha \approx 145^\circ$, where $b_1 < 0$ and $b_2, b_3 > 0$. The minima in iteration counts are located at $\alpha \approx 0^\circ$, $85^\circ$ and $180^\circ$. Here the perturbation values $b_j \approx 0$ ($b_j = 1, 2, 3$), whilst two of the perturbation values $d_i$ are at their maximum (for example, at $\alpha = 0$, $d_2 = d_3 = 1$). This increases the diagonal dominance of the Jacobian matrix, making smoothers more effective. These findings are consistent with the 2D case (cf. Figure 6.15).

To analyse the asymptotic behaviour of the iteration counts with respect to $N$, we plot a semi-log graph for problem parameters $\alpha = 67^\circ$, $Re = 100$ and 500 in Figure 6.37. Whilst the iteration counts do not appear bounded in Figure 6.37, we do observe asymptotic behaviour similar to $O(\log N)$ for the larger problem sizes (as demonstrated by the line with gradient 13.1). The situation is less clear for $Re = 500$, however the asymptotic behaviour seems generally better than for $Re = 100$. 
Figure 6.37: Semi-log plot of the average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver applied to Example 6.3.1. The results are presented for $\alpha = 67^\circ$. The line of slope 13.1 is for comparison.

The execution times for $\alpha = 67^\circ$ and Re = 100, 500 are depicted in the log-log plot Figure 6.38 with a line of gradient 1 for comparison. We observe near-linear

Figure 6.38: Log-log plot of the execution times (s) taken by the GMRES+$\hat{P}_{LSC}$ solver applied to Example 6.3.1. The results are presented for $\alpha = 67^\circ$. The best fit line has gradient 1.2 and the line of slope 1 is for comparison.
scaling of the execution times under mesh refinement for the SVT-NSE, whilst slightly higher asymptotic behaviour is observed for SDVT-NSE. The line of gradient 1.2 comes from the line of best fit of all problem parameters shown in Figure 6.38 which suggests that the asymptotic behaviour of the execution times for \( \hat{P}_{LSC} \) grows as \( O(N^{1.2}) \).

We conclude this section with a comparison of the execution times of direct solve (using SuperLU) and GMRES (using Trilinos AztecOO) preconditioned with \( P_E \), \( P_{LSC} \) and \( \hat{P}_{LSC} \) in Figure 6.39. We have not used a log scale in this experiment to compare the execution times in absolute terms. For the solvers that use SuperLU, the maximum problem size that fits into the computer’s memory is \( N = 63269 \). The observations are similar to the 2D case studies, for comparable problem sizes, we see that the direct solve and GMRES+\( P_E \) have similar execution times, while GMRES+\( P_{LSC} \) takes less than half the time for comparable problem sizes, and GMRES+\( \hat{P}_{LSC} \) takes a fraction of the time.

![Comparison of solvers](image)

**Figure 6.39:** Execution times (s) for direct solve, and the GMRES solver preconditioned with \( P_E \), \( P_{LSC} \) and \( \hat{P}_{LSC} \) for Example 6.3.1 with \( \alpha = 67^\circ \) using the SVT-NSE and Re = 500.

### 6.3.2 Radial Flow through a Quarter Annulus

We study the convergence properties of the new augmented preconditioner on a problem posed over a three-dimensional quarter annulus domain. This is an extension of the 2D case studied in Section 6.2.2, where the domain is extended along the \( x_3 \) axis, see Figure 6.40. The domain \( \Omega \) is the volume bounded by four planes: \( (x_2 = 0) \)
and \(x_1 = 0\), which corresponds to the \(\partial \Omega_{s1}\) and \(\partial \Omega_{s2}\) boundaries in the 2D problem, cf. Figure 6.17 \((x_3 = 0)\) and \((x_3 = 2)\), as a consequence of extending along the \(x_3\) axis, and the two cylinders, the inner cylinder is the inflow boundary \(\partial \Omega_I\) and the outer cylinder is the outflow boundary \(\partial \Omega_O\). Let \(t_k\) denote the current time, \(\alpha = \arctan(x_2/x_1)\) and define \(C_k = - \cos(\pi t_k)/2 + 0.5\). Then we drive the inflow along the boundary \(\partial \Omega_I\) by multiplying the imposed inflow in Example 6.2.3 by \(C_k\) and set \(u_3 = 0\). Thus the inflow boundary condition is

\[
 u_1 = C_k \cos(\alpha), \quad u_2 = C_k \sin(\alpha) \quad \text{and} \quad u_3 = 0 \quad \text{on} \quad \partial \Omega_I.
\]

Formal specification of the case study is as follows:

**Example 6.3.2.** Uniform radial flow through a quarter annulus wedge in 3D.

**The domain:** \(\Omega = \{1 \leq x^2 + y^2 \leq 9\} \times [0, 2]\).

**The time interval:** \(T = [0, 1]\).

**The boundaries:**

\[
\begin{align*}
\partial \Omega_I & = [x_1, x_2 > 0, x_1^2 + x_2^2 = 1], x_3 \in [0, 2], \\
\partial \Omega_O & = [x_1, x_2 > 0, x_1^2 + x_2^2 = 9], x_3 \in [0, 2], \\
\partial \Omega_{s1} & = \{(x_1, x_2, x_3) \mid x_1 \in [1, 3], x_2 = 0, x_3 \in [0, 2]\}, \\
\partial \Omega_{s2} & = \{(x_1, x_2, x_3) \mid x_1 = 0, x_2 \in [1, 3], x_3 \in [0, 2]\}, \\
\partial \Omega_{s3} & = \{(x_1, x_2, x_3) \mid 1 \leq x_1^2 + x_2^2 \leq 9, x_3 = 2\} \quad \text{and} \\
\partial \Omega_{s4} & = \{(x_1, x_2, x_3) \mid 1 \leq x_1^2 + x_2^2 \leq 9, x_3 = 0\}.
\end{align*}
\]

Denote two arbitrary orthogonal tangent vectors at each discrete point on \(\partial \Omega_O\) by \(\hat{\mathbf{t}}_1\) and \(\hat{\mathbf{t}}_2\), the current time by \(t_k\), and \(C_k = - \cos(\pi t_k)/2 + 0.5\). Then the initial and boundary conditions are:

\[
\begin{align*}
u_i & = 0, \quad (x_1, x_2, x_3) \in \Omega, \quad t = 0, \\
u_1 & = C_k \cos(\alpha), \quad u_2 = C_k \sin(\alpha), \quad u_3 = 0, \quad (x_1, x_2, x_3) \in \partial \Omega_I, \quad t_k \in T, \\
u \cdot \hat{\mathbf{t}}_1 & = 0 \quad \text{and} \quad u \cdot \hat{\mathbf{t}}_2 = 0, \quad (x_1, x_2, x_3) \in \partial \Omega_O, \\
u_1 & = \text{‘do nothing’}, \quad u_2 = u_3 = 0, \quad (x_1, x_2, x_3) \in \partial \Omega_{s1}, \\
u_1 & = u_3 = 0, \quad u_2 = \text{‘do nothing’}, \quad (x_1, x_2, x_3) \in \partial \Omega_{s2}, \\
u_1 & = \text{‘do nothing’}, \quad u_2 = \text{‘do nothing’}, \quad u_3 = 0, \quad (x_1, x_2, x_3) \in \partial \Omega_{s3} \cup \partial \Omega_{s4}.
\end{align*}
\]
for \( i = 1, 2, 3 \) and \( t_k \in [0, 1] \). An impulsive start from rest is performed with an initial time step of \( \Delta t_0 = 0.01 \). On average, 15 steps were taken, therefore \( \Delta t_k \approx 0.067 \).

Figure 6.40: Plot of the solution for the unsteady uniform radial flow through an annulus wedge in Example 6.3.2 with \( \text{Re} = 100 \) at time \( t_k = 0.3 \). The colour contours and streamlines represent the pressure gradient and velocity flow field, respectively.

The AMG settings given in Preconditioner 6.7 will be used in this case.

Convergence Evaluation

We numerically evaluate the performance of the different versions of the augmented preconditioner \( P_E, P_{LSC} \), and \( \hat{P}_{LSC} \) applied to Example 6.3.2 with the following configurations:

- Both the SVT-NSE and SDVT-NSE.
- Reynolds number \( \text{Re} = 100, 200 \) and 500.
- Several uniform mesh refinements. The domain is discretised using uniform quadrilateral grids with discrete problem sizes between \( N = 1767 \) (mesh size \( h = 1/4 \)) and \( N = 70717 \) (\( h = 1/14 \)) unknowns for \( P_E \) and \( P_{LSC} \) versions of the preconditioner, and \( N = 995217 \) (\( h = 1/34 \)) for the \( \hat{P}_{LSC} \) version of the preconditioner.
- Preconditioned GMRES solver with a relative tolerance of \( \varepsilon_K = 10^{-6} \) starting from an initial solution of zero.
Adaptive time stepping using BDF-2 method with a local truncation error tolerance \( \varepsilon_T = 10^{-4} \). Starting with from an initial solution of zero, at each new time step we use the solution computed at the previous time step as the initial guess for the Newton’s method.

Firstly, we assess the performance of \( P_E \). The iteration counts are presented in Table 6.41, and execution times for \( Re = 100 \) and 500 are shown in Figure 6.41. We observe that the iteration counts for \( P_E \) are very small and bounded under mesh refinement. From Figure 6.41 we have calculated the asymptotic behaviour of the execution times to be \( O(N^{2.14}) \). Next we evaluate the exact LSC augmented preconditioner \( P_{LSC} \). The iteration counts are tabulated in Table 6.42. The iteration counts for \( P_{LSC} \) are on average four times higher than \( P_E \) but a very robust performance is observed.

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>1767</th>
<th>5781</th>
<th>13483</th>
<th>26073</th>
<th>44751</th>
<th>70717</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>100</td>
<td>4.2</td>
<td>4</td>
<td>3.9</td>
<td>3.5</td>
<td>3.4</td>
<td>3.4</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>4.4</td>
<td>4</td>
<td>4</td>
<td>3.6</td>
<td>3.6</td>
<td>3.4</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>4.8</td>
<td>4.5</td>
<td>4</td>
<td>3.9</td>
<td>3.6</td>
<td>3.6</td>
</tr>
<tr>
<td>SDVT</td>
<td>100</td>
<td>4.2</td>
<td>4</td>
<td>3.9</td>
<td>3.5</td>
<td>3.4</td>
<td>3.4</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>4.4</td>
<td>4</td>
<td>4</td>
<td>3.6</td>
<td>3.6</td>
<td>3.4</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>4.8</td>
<td>4.5</td>
<td>4</td>
<td>3.9</td>
<td>3.6</td>
<td>3.6</td>
</tr>
</tbody>
</table>

Table 6.41: The average number of iterations taken by the GMRES+\( P_E \) solver applied to Example 6.3.2 as a function of the viscous term, ‘vis’, Reynolds number, ‘Re’, and discrete problem size.

Figure 6.41: Log-log plot of the execution times (s) taken by the GMRES+\( P_E \) solver applied to Example 6.3.2. The best fit line has gradient 2.14 and the line of slope 1 is for comparison.
and the iteration counts are bounded under mesh refinement. A slight increase in iteration counts is evident as the Reynolds number increases. The SVT-NSE requires slightly higher iteration counts than their SDVT counter-pairs. The execution times for Re = 100 and 500, with a line of gradient 1 are shown in Figure 6.42. As expected, the execution times are sub-optimal. Using the line of best fit through the data points we calculate the asymptotic behaviour for the execution times to be $O(N^{1.92})$, i.e. approximately quadratic scaling.

Lastly, we evaluate the performance of $\hat{P}_{LSC}$. The iteration counts for mesh sizes $h = 1/\ell$, for $\ell = 6, 8, 12, 20$ and 34 are shown in Table 6.43. The log-log plot of execution times with Re = 100 and 500 is given in Figure 6.43.

### Table 6.42: The average number of iterations taken by the GMRES+$P_{LSC}$ solver applied to Example 6.3.2 as a function of the viscous term, ‘vis’, Reynolds number, ‘Re’, and discrete problem size.

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>1767</th>
<th>5781</th>
<th>13483</th>
<th>26073</th>
<th>44751</th>
<th>70717</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>100</td>
<td>14.7</td>
<td>15.7</td>
<td>14.8</td>
<td>13.8</td>
<td>13.4</td>
<td>13.1</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>14.6</td>
<td>15.8</td>
<td>15.8</td>
<td>14.5</td>
<td>14.1</td>
<td>13.9</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>14.6</td>
<td>16</td>
<td>15.6</td>
<td>15.7</td>
<td>15.2</td>
<td>14.8</td>
</tr>
<tr>
<td>SDVT</td>
<td>100</td>
<td>14.8</td>
<td>15.5</td>
<td>14.3</td>
<td>13.4</td>
<td>12.8</td>
<td>12.6</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>14.5</td>
<td>15.7</td>
<td>15.5</td>
<td>14.5</td>
<td>13.7</td>
<td>13.1</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>14.6</td>
<td>15.8</td>
<td>15.5</td>
<td>15.4</td>
<td>14.8</td>
<td>14.7</td>
</tr>
</tbody>
</table>

Figure 6.42: Log-log plot of the execution times (s) taken by the GMRES+$P_{LSC}$ solver applied to Example 6.3.2. The best fit line has gradient 1.92 and the line of slope 1 is for comparison.
CHAPTER 6. NUMERICAL EVALUATION

<table>
<thead>
<tr>
<th>vis</th>
<th>Re</th>
<th>5781</th>
<th>13483</th>
<th>44751</th>
<th>204343</th>
<th>995217</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVT</td>
<td>100</td>
<td>27.2</td>
<td>28.1</td>
<td>26.3</td>
<td>26.1</td>
<td>32.1</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>25.7</td>
<td>29.7</td>
<td>29</td>
<td>27.5</td>
<td>27.6</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>31.6</td>
<td>33.2</td>
<td>31</td>
<td>29.6</td>
<td>27.1</td>
</tr>
<tr>
<td>SDVT</td>
<td>100</td>
<td>26.3</td>
<td>27.2</td>
<td>26.2</td>
<td>28.5</td>
<td>33.2</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>27.8</td>
<td>29.1</td>
<td>27.8</td>
<td>27.2</td>
<td>30.3</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>31.2</td>
<td>32.1</td>
<td>30</td>
<td>28.7</td>
<td>28.1</td>
</tr>
</tbody>
</table>

Table 6.43: The average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver applied to Example 6.3.2 as a function of the viscous term, ‘vis’, Reynolds number, ‘Re’, and discrete problem size.

Figure 6.43: Log-log plot of the execution times (s) taken by the GMRES+$\hat{P}_{LSC}$ solver applied to Example 6.3.2. The best fit line has gradient 1.17 and the line of slope 1 is for comparison.

From Table 6.43 we observe moderate growth in iteration counts for Re = 100 under mesh refinement, whilst for Re = 500 the iterations decreases as the mesh is refined. The difference in iteration counts between the largest and smallest problem size shown is very small, as the problem size increases by approximately 172 times, the change in iteration counts is less than 5. A slight dependence on Re is observed which seems to diminish as the mesh is refined. In Figure 6.43 near-linear execution times are observed under mesh refinement. The line of best fit for the problem parameters shown has a gradient of 1.17 which suggests an asymptotic behaviour of $O(N^{1.17})$.

We conclude this section with a comparison of the execution times of direct solve (using SuperLU) and GMRES (using Trilinos AztecOO) preconditioned with $P_E$, $P_{LSC}$ and $\hat{P}_{LSC}$ in Figure 6.44. We have not used a log scale so we can compare the
execution times in absolute terms. For the solvers using SuperLU, the maximum problem size is $N = 70717$. These results are consistent with those shown in Figure 6.39.

![Comparison of solvers](image)

Figure 6.44: Execution times (s) for direct solve, and the GMRES solver preconditioned with $P_E$, $P_{LSC}$ and $\hat{P}_{LSC}$ for Example 6.3.2 using the SVT-NSE and Re = 500.

### 6.4 Summary

In this chapter we presented numerical evaluation of the performance of three different versions of the new augmented preconditioner for a range of NS problems with weakly imposed natural outflow boundary condition (steady-state two-dimensional problems and unsteady problems in three spatial dimensions).

We start with the exact version of the preconditioner $P_E$ for which we have an analytical proof that it is spectrally equivalent to the simplified Jacobian, independently of the discretisation parameter, Reynolds number, or the orientation of the boundary $\partial \Omega$. We observed robust performance of the GMRES solver with $P_E$, with low and bounded iteration counts under mesh refinement for all test cases and all problem parameters. This is in agreement with the analytical results presented in Section 4.6. However, this version of the preconditioner led to an iterative solver with the worst asymptotic scaling of the wall clock times, ($O(N^{1.7})$ in 2D and $O(N^{2.1})$ in 3D).
When using the $P_{LSC}$ version of the preconditioner, we observed that the asymptotic behaviour of the GMRES solver follows the pattern of the GMRES solver with the LSC preconditioner applied to the standard NS problem. The iteration counts under mesh refinement were either bounded in the cases of Example 6.2.3 and Example 6.3.2 (for $Re \neq 0$) or a logarithmic asymptotic growth was observed in Example 6.2.1 and Example 6.3.1. Such robust iteration counts come at a price of high sub-optimal scaling of the execution times. For 2D problems the asymptotic behaviour in wall clock times was estimated to be $O(N^{1.55})$. We observed even worse asymptotic behaviour of $O(N^{1.9})$ for 3D problems.

This issue was addressed to a large extent by the introduction of a fully inexact version of the preconditioner, $\hat{P}_{LSC}$, where BoomerAMG is used to approximate inverse of the $\hat{F}$ block and the pressure Poisson operators in the LSC preconditioner. In this case a moderate growth in iteration counts were observed for Examples 6.2.1, 6.2.3 and 6.3.1 as $O(N^{0.17})$, $O(N^{0.25})$ and $O(\log N)$, respectively. The iteration counts appear to be bounded for Example 6.3.2. The asymptotic behaviour of the execution times were greatly reduced to approximately $O(N^{1.2})$ for Examples 6.2.1, 6.2.3 and 6.3.1 and $O(N^{1.17})$ for Example 6.3.2 - a nearly-optimal behaviour. The superiority of the GMRES solver with $\hat{P}_{LSC}$, both in terms of memory requirements and asymptotic execution times, when compared to a sparse direct solver and two other versions of the augmented preconditioner that require exact block inversions was clearly demonstrated in Figures 6.16, 6.22, 6.39 and 6.44.

The optimal AMG parameters for the approximate inverse of the augmented momentum block $\hat{F}$ were experimentally determined for both 2D and 3D cases. For 2D problems, the AMG settings were different for the two forms of the viscous term. The behaviour of the AMG approximation of the augmented momentum block $\hat{F}$ was unexpected for the SDVT-NSE in 2D. It is unclear why smaller values of the strength of dependence parameter led to significantly higher iteration counts (including the recommended value $\theta = 0.25$ for the Poisson problem), since it implies a higher quality preconditioner. The dramatic drop in iteration counts observed at $\theta \approx 2/3$ was also unexplained. This requires further research. For 3D problems we found a set of AMG parameters which works well for both forms of the viscous term and all problem parameters considered.
6.5 Discussion

In this section we discuss some of the assumptions and algorithmic choices, giving the rationale behind our decisions and putting them in a wider context of the existing literature.

In all the simulations shown in the previous sections, we have started from the simplest choice of an initial iterate of zero for the Newton’s method. It is well known that Newton’s method has local quadratic convergence, that is, if the initial iterate $\bar{x}^{[0]}$ is sufficiently close to the exact solution $\bar{x}^*$, and let $\bar{e}^{[k]} = \bar{x}^* - \bar{x}^{[k]}$ denote the error at the $k$th iteration, then there exists a constant $C_1$ such that

$$\|\bar{e}^{[k+1]}\| \leq C_1 \|\bar{e}^{[k]}\|^2.$$  \hspace{1cm} (6.5.1)

For proof see [151, p. 71]. For the incompressible NSE, the radius of Newton’s method ball of convergence is typically proportional to the kinematic viscosity $\nu = \mu/\rho$. Recall from Section 2.2.1 that Reynolds number is $\text{Re} = \frac{UL}{\mu} = \frac{U\mathcal{L}\rho}{\mu}$. Thus as the Reynolds number increases, it becomes a challenge to determine sufficiently accurate initial solution guesses to ensure convergence. Picard’s iteration is widely accepted as being globally convergent [151, p. 67, 68, p. 326]. The main drawback is that Picard’s iteration has asymptotically linear convergence. Thus, when Newton’s method does not converge, it is always possible to improve the initial guess with a few iterations of Picard’s method before switching to Newton’s method to take advantage of its quadratic rate of convergence. This technique is used in [68, p. 374] for enclosed flow problems with $\text{Re} = 1000$. Another alternative is to use a continuation method with $\text{Re}$ as a continuation parameter [196, pp. 127–128]. In this case we would use the solution computed at a lower Reynolds number as the initial iterate for Newton’s method. Newton’s method converged with a zero initial guess for all the steady-state experiments presented in this chapter.

For unsteady problems we take the initial iterate of the Newton’s method at each time step to be the solution computed at the previous time step. This is the default setting in oomph-lib and we found that Newton’s method converged for all the unsteady problems presented. An alternative would be to take an initial guess to be the predictor value at the current time step. This option is not implemented in oomph-lib.

We used adaptive BDF-2 as the time integrator for all our unsteady problems with a local truncation error tolerance of $\varepsilon_T = 10^{-4}$. Reducing $\varepsilon_T$ would increase time
accuracy at the expense of having more smaller time steps taken by the integration. The default value $\varepsilon_T = 10^{-4}$ is both recommended [99, p. 276] and determined experimentally using the strategy described in [114]. First, we run the simulations for (fixed) decreasing time steps $\Delta t$ until the solutions agree within four significant figures and monitor the value of the root mean square (RMS) $\left( \| \cdot \|_2 / \sqrt{N_u} \right)$ (where $N_u$ is the number of velocity nodes), of velocity local truncation error, and check that further decreasing $\Delta t$ does not change the solution within four significant figures. Then we set this value for $\varepsilon_T$. In the adaptive time step size selection procedure, we use the RMS of the velocity local truncation error, as recommended in [114, 99, p. 276].

Since we are using iterative methods to solve the linear system arising from the Newton linearisation of the discretised NS system, the system at the $k$th iteration

$$
\begin{bmatrix}
F^{[k]} & B^T \\
B & O
\end{bmatrix}
\begin{bmatrix}
\delta \bar{U}^{[k]} \\
\delta \bar{P}^{[k]}
\end{bmatrix}
= -
\begin{bmatrix}
\bar{r}_u^{[k]} \\
\bar{r}_p^{[k]}
\end{bmatrix}
$$

(6.5.2)

may be over-solved, i.e. it may not be necessary to solve the linearised system to high accuracy if the nonlinear iterate is far from the solution. This observation has lead to the development of inexact Newton methods. In the following summary of inexact Newton method we follow [68, pp. 371–373, 151, pp. 95–97]. Consider a generic system of nonlinear algebraic equations $F(\bar{x}) = \bar{0}$, and assume that $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ and $\bar{x} \in \mathbb{R}^n$ such that $F(\bar{x}^*) = \bar{0}$, the Jacobian $F' (\bar{x})$ is nonsingular at $\bar{x}^*$, and that it is Lipschitz continuous near $\bar{x}^*$. The inexact Newton method constructs a sequence of iterates $\bar{x}^{[k+1]} := \bar{x}^{[k]} + \bar{s}^{[k]}$ where $\bar{s}^{[k]}$ is computed to satisfy the inexact Newton condition

$$
\left\| F' (\bar{x}^{[k]}) \bar{s}^{[k]} + F(\bar{x}^{[k]}) \right\| \leq \eta^{[k]} \left\| F(\bar{x}^{[k]}) \right\|,
$$

(6.5.3)

where $\eta^{[k]}$ is the forcing term for the $k$th step. Note that when $\eta^{[k]} = 0$, (6.5.3) reduces to the exact Newton’s method. Let $\bar{e}^{[k]} = \bar{x}^* - \bar{x}^{[k]}$ denote the error at the $k$th step of the inexact Newton iteration. It is shown in [151, pp. 95–96], Theorem 6.1.2, that if $\bar{x}^{[0]}$ is sufficiently close to $\bar{x}^*$, $\bar{s}^{[k]}$ is computed to satisfy (6.5.3) and $\bar{x}^{[k+1]} := \bar{x}^{[k]} + \bar{s}^{[k]}$, then there exist two constants $C_1$ and $C_2$ such that

$$
\left\| \bar{e}^{[k+1]} \right\| \leq C_1 \left\| \bar{e}^{[k]} \right\|^2 + C_2 \eta^{[k]} \left\| \bar{e}^{[k]} \right\|.
$$

(6.5.4)

The first term on the right hand side of (6.5.4) corresponds to the error associated with the exact Newton iteration (6.5.1). The second term is associated with the error
from the approximate solution of the linearised system. If $\eta^{[k]}$ is small, more linear iterations are required to satisfy the stopping criterion of the linear solver, but if $\eta^{[k]}$ is significantly smaller than the norm of the current $\tilde{e}^{[k]}$, then the bound in (6.5.4) is dominated by the first term and the extra linear iterations associated with small $\eta^{[k]}$ will not improve the accuracy of the nonlinear iterate. In this sense a good choice for $\eta^{[k]}$ is one which balances the magnitude of the two terms on the right of (6.5.4). This would lead to a reduction in the total number of linear iterations. We note that when $\bar{x}^{[k]}$ is close to $\bar{x}^*$, then $\mathcal{F}'(\bar{x})$ should not change very much and $\mathcal{F}(\bar{x}^{[k]}) \approx \mathcal{F}'(\bar{x}^*) \tilde{e}^{[k]}$, which implies that $\|\tilde{e}^{[k]}\|$ is proportional to $\|\mathcal{F}(\bar{x}^{[k]})\|$. Thus the choice [68, p. 372]

$$\eta^{[k]} = \tau \|\mathcal{F}(\bar{x}^{[k]})\| \quad (6.5.5)$$

implies that the quadratic rate of convergence of Newton’s method is preserved and the two terms in (6.5.4) are balanced. Elman, Silvester and Wathen in [68, pp. 372–374] suggest the choice

$$\eta^{[k]} = 10^{-2}\|\tilde{r}^{[k]}\| \quad (6.5.6)$$
In this chapter we investigate the scalability of the BPF described in Chapter 5 in terms of (wall-clock) execution time and memory usage on a parallel architecture with up to 16 cores. The computations reported in this chapter are performed using the Computational Shared Facility (also known as Danzek or CSF), a High Performance Computing (HPC) cluster located at the University of Manchester. We run the simulations on a reserved Haswell node with two Intel® Xeon® Processors E5-2690 v3 (30M Cache, 2.60 GHz). These have twelve cores per processor with 128GB of shared RAM. For more details on the system configuration of the CSF see [237]. We reiterate that the serial computations presented in Chapter 6 were ran on Westmere nodes, because are designated as high memory nodes on the CSF. However, these nodes only have 12 cores. In this chapter we use the Haswell nodes, that offer 24 cores per node. Since the Haswell node is a multicore-multiprocessor system, to avoid ambiguity, in the sequel we refer to an execution unit (a core) as a processing element (PE).

In the context of HPC, scalability refers to the ability of a program to maintain efficiency when executed in an parallel setting with an increasing number of PEs. Parallel efficiency can be derived from either strong scalability or weak scalability. Let \( p \) denote the number of PEs. Then a program is strongly scalable if the execution time decreases linearly with the increase of \( p \) when the problem size is fixed. Likewise, a program is weakly scalable if the execution time is constant as \( p \) is increased for a fixed problem size per PE. In subsequent sections we present both the weak and strong scalability of the BPF.

\[^{1}\text{We reserved a node to obtain reliable timing results. The influence of other users may negatively impact the execution time of our code if, for example, their code is using the swap space.}\]
The most time consuming procedures of the BPF are the calls to the functions `block_setup(...)`, `get_block(...)`, `get_block_vector(...)` and `return_block_vector(...)`. In Section 5.3 we have presented the necessary modification of these functions with the newly implemented routines for the concatenation of matrices and vectors and the splitting of vectors. Here we demonstrate that the modifications do not significantly increase the execution time of the BPF and that these functions exhibit good parallel scaling, that is, the parallel efficiency is maintained as $p$ is increased.

The parallel performance of `get_block(...)` is investigated in Section 7.2, and the scalability of the `get_block_vector(...)` and `return_block_vector(...)` functions are explored in Section 7.3. In Section 7.4 we demonstrate the scalability of the BPF when applied to facilitate parallel iterative solution of linear systems that arise from the discretisation of the 3D quarter annulus problem discussed in Section 6.3.2.

## 7.1 Parallel Performance Review of the Initial BPF

The weak parallel scalability of the BPF was studied (up to $p = 64$) by Muddle in [180]. The computations were performed on the (now retired) University of Manchester Horace architecture [203], where each node had eight cores. We compare weak parallel scalability for the functions `block_setup(...)` and `get_block(...)` with those presented in [180, p. 117, Table 4.23]. Our results are obtained for Example 6.3.2, while [180] are based on a 3D lid driven cavity problem [68, pp. 316–317]. The Jacobian matrices for both problems have a similar sparsity pattern and matrix properties. The performance of the functions `get_block_vector(...)` and `return_block_vector(...)` is compared with [180, pp. 110–117, Tables 4.8 and 4.14] for the 2D and 3D linear elasticity problem, since no equivalent results for a 3D Navier-Stokes problem were presented in [180].

It was shown in [180] that the `block_setup(...)` function exhibit poor weak parallel scaling, this is due to the all-to-all communication between PEs used to generate additional lookup schemes for parallel execution. We refer to these as parallel lookup schemes, since they are constructed only in the parallel setting. For example, the map between the global indices of the Jacobian matrix, the PE number (rank) and the indices of the block matrices. This establishes the data locations which are used to
facilitate subsequent send/receive calls between PEs. The construction of these parallel lookup schemes is expensive in terms of the execution time but they ensure short execution times for the remaining BPF functions. The newly introduced functionality requires new lookup schemes to be generated in the `block_setup(...)` function, such as the mapping between the internal DOF types, DOF types, and block types. The new lookup schemes are small (the size of the most fine grained internal DOF types), they are constructed with few integer additions and do not require inter-processor communication. Therefore, the overhead associated with the construction and storage of the new lookup schemes is negligible which implies that the modified `block_setup(...)` function should have the parallel scaling as those observed in [180]. The `get_block(...)` function from the initial BPF implementation demonstrated good weak parallel scaling, and the same was observed in our computations. Very good weak parallel scaling (nearly constant execution times) was presented for the initial version of the `get_block_vector(...)` and `return_block_vector(...)` functions, which was not observed in our experiments. Details will follow in the subsequent sections.

### 7.2 Performance Analysis of the `get_block(...)` Function

The newly implemented `get_block(...)` function (described in Section 5.3) performs the extraction of block matrices (using existing functionality) followed by concatenation of the extracted blocks (using newly implemented functionality). As discussed in Section 5.3.2, the concatenation routine must also permute the columns to ensure the correctness of certain matrix operations, such as the scalar addition to the diagonal entries. In a typical implementation of a `BlockPreconditioner`, the `get_block(...)` function is only called from the preconditioner’s `setup(...)` function, which in turn is called once per nonlinear solve.

We demonstrate the scalability of the `get_block(...)` function on the Jacobian of the 3D quarter annulus problem Example 6.3.2 which has a $9 \times 9$ block structure. For this experiment we use the SVT-NSE. In this test firstly we extract all the DOF blocks then concatenate them back into a single matrix. Initial testing show that the execution time for the concatenation routines (particularly for vectors) is so short that
we require the problem size per PE to be much larger than that of a typical FEM application to be able to notice the associated overhead. Therefore, the results presented in this section are for the first execution of the preconditioner setup(...) function only. The block extraction functionality utilises inter-processor communication to ensure that the rows of the (extracted) block matrices are uniformly distributed among the PEs. Due to the pre-processing performed in the block_setup(...) function, block extraction involves only MPI send and receive operations and vector updates with offsets. The implementation of this functionality has not been modified during this project. The concatenated matrix will be (roughly) uniformly distributed by rows, since each PE copies data from the resident block matrices. The most expensive operations in the concatenation routine are copying the values from the sub-block matrices into the result matrix and calculating new column positions for each nonzero entry to ensure a coherent row and column permutation.

7.2.1 Weak Scaling for the get_block(...) Function

We set the problem size per PE to \( N_1 \approx 1 \times 10^6 \) (mesh size \( h = \frac{1}{34} \)). For \( p \) PEs, the ideal problem size would be \( pN_1 \). However, the construction of a 3D bulk mesh and 2D boundary mesh makes it impossible to achieve the exact size \( pN_1 \). In Table 7.1 we have tabulated the mesh size \( (h) \), the associated problem size \( (N) \) and the ideal problem size \( (pN_1) \) for comparison.

\[
\begin{array}{cccccc}
\hline
p & 1 & 2 & 4 & 8 & 16 \\
\hline
\text{Mesh size } h & 1/34 & 1/43 & 1/54 & 1/68 & 1/85 \\
N (\times10^6) & 0.995 & 2.008 & 3.969 & 7.911 & 15.432 \\
pN_1 (\times10^6) & 0.995 & 1.990 & 3.980 & 7.962 & 15.923 \\
\hline
\end{array}
\]

Table 7.1: Comparison of problem sizes \( N \) and the ideal problem size \( pN_1 \) for different number of processing elements \( p \).

Let \( T(N, p) \) denote the (wall-clock) execution time to perform \( N \) unit of work on \( p \) PEs. Then we define the weak parallel efficiency \( [84] \), expressed in (%), as

\[
E_w(N_1, p) = \frac{T(N_1, 1)}{T(pN_1, p)} \times 100\%.
\] (7.2.1)

Optimal weak scaling would imply constant execution time – as \( p \) increases, \( E_w(N_1, p) \) becomes constant. In this context, if a mild increase in execution time is observed as \( p \) is increased, we describe this as good weak parallel scaling. The execution time, weak
parallel efficiency and the memory usage for extraction and concatenation of the DOF blocks for Example 6.3.2 are shown in Table 7.2. We observe that the majority of the execution time is spent in the block extraction process. The concatenation procedure takes less than one second for the problem sizes considered and is negligible when compared to the total execution time. Block extraction scales poorly from 1 to 2 PEs. This trend is caused by the cost of parallelisation, which includes both changes in the algorithm and the communication overhead. There exists a serial and a parallel version of the block extraction routine. The parallel code is more complex and utilises the parallel lookup schemes (pre-computed in block_setup(...))\(^3\). Therefore we generally expect parallel versions of the code to have slightly higher execution times. But we observe more than doubling in the execution time. This can be explained by the nature of the ‘useful’ work: the serial algorithm copies data from the Jacobian matrix to the block matrix, and almost no numerical computation is involved, i.e. the task is memory bound, rather than computation bound. In the parallel version, data must be pre-processed before sending: each PE copies the matrix data required by every other processing element into arrays with various offsets for post-processing. Post-processing involves the construction of the data structures required to build a CRS matrix. The pre- and post-processing operations associated with the overhead of parallelisation are the same as those used to do the work in the serial case. Thus, a doubling of execution times from \(p = 1\) to \(p = 2\) is not unreasonable. From 2 to 16 PEs, good parallel scaling is observed. The results presented here are consistent with those from [180, p. 117, Table 4.23].

The concatenation routine exhibit good scaling from 1 to 4 PEs, but the weak

\(^2\)We note that both these functions have short execution times when compared to an actual FEM simulation. For example the assembly of the Jacobian matrix for the largest problem size took approximately 3 hours.

\(^3\)This is a common approach for many underlying routines in oomph-lib and ensures fast execution of the code in serial setting. We re-iterate that the user is not aware of this, and the code automatically selects the appropriate routine.

### Table 7.2: Execution time, weak efficiency and memory usage for the extraction of DOF blocks and matrix concatenation.

<table>
<thead>
<tr>
<th>(p)</th>
<th>(N \times 10^6)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOF block extraction (s)</td>
<td>10.187</td>
<td>26.818</td>
<td>28.651</td>
<td>32.152</td>
<td>33.472</td>
<td></td>
</tr>
<tr>
<td>Weak Efficiency (%)</td>
<td>100</td>
<td>37.987</td>
<td>35.557</td>
<td>31.685</td>
<td>30.435</td>
<td></td>
</tr>
<tr>
<td>Concatenation (s)</td>
<td>0.461</td>
<td>0.476</td>
<td>0.482</td>
<td>0.552</td>
<td>0.658</td>
<td></td>
</tr>
<tr>
<td>Weak Efficiency (%)</td>
<td>100</td>
<td>96.771</td>
<td>95.648</td>
<td>83.518</td>
<td>70.193</td>
<td></td>
</tr>
<tr>
<td>Peak mem. per PE (GB)</td>
<td>3.579</td>
<td>6.081</td>
<td>6.135</td>
<td>6.600</td>
<td>6.702</td>
<td></td>
</tr>
</tbody>
</table>
efficiency drops to 84% and 70% for 8 and 16 PEs, respectively. Unlike the block extraction routine, the same code is used in both serial and parallel setting and there is no communication between the PEs. Therefore the poor scaling is not due to parallelisation overhead. An explanation for the poor scaling is that the code becomes memory bound for the problem sizes considered, where the limiting factor for the execution speed is the amount of available memory or the memory access speed. The matrix concatenation algorithm is very read/write intensive with very little data processing required to compute the new column indices. At each access to nonzero entries, the PEs are reading and writing to different locations in memory. There are two potential issues here. Firstly, the memory bandwidth between the CPU and RAM may become saturated. It is shown in [107] that the main memory bandwidth usage depends on the number of active cores and their clock frequencies. For Haswell processors, it was demonstrated that the main memory bandwidth starts to saturate at $p = 8$ at all frequencies. This may be the cause for the mild increase in execution times observed for $p = 8$. The second issue is the contention for memory access. The Haswell node is a symmetric (shared-memory) multiprocessor (SMP) machine, with the RAM shared across all 24 cores. The main difficulty in application scaling on SMP machines is the contention for shared resources and hardware related delays such as the processor memory bandwidth, bus speed and the Column Access Strobe (CAS) latency of the RAM [193, p. 98]. Typically, two different locations in RAM cannot be accessed concurrently. As the number of PEs increases on a single SMP machine, hardware related delays start to significantly impact the efficiency of a memory bound function. As noted in [193, pp. 345–346], multiprocessor machines typically have fewer than 32 PEs for this reason. In the case of matrix concatenation, $p$ PEs need to read/write concurrently from/to different memory locations with a few integer additions to permute the columns. This significantly affects the performance of the concatenation routine. Since the matrix concatenation algorithm only operates on local data, and poor weak parallel scaling on a single node is due to memory bandwidth saturation and RAM access contention, the non-uniform memory access (NUMA) architecture may offer a better parallel scaling performance. On multiple nodes, assuming that all nodes are identical and have SMP architectures, then the weak efficiency should be bounded from below by the lowest efficiency of a single node.

---

4The implementation copies a whole row of a sub-matrix with the C++ standard library function `memcpy` since it is considered one of the most efficient methods for copying blocks of memory.

5We used ‘cores’ instead of processing elements to be consistent with the paper [107].
The memory usage values reported in Table 7.2 are calculated by taking the \texttt{maxvmem} value divided by the number of PEs running the job. This memory usage accounts for the finite element meshes, the original Jacobian matrix, the extracted DOF blocks and the concatenated result matrix. Under weak scaling, constant peak memory usage per PE is optimal. By contrast, moving from \( p = 1 \) to \( p = 2 \), approximately doubles the memory usage. This is due to parallelisation overhead: the storage for auxiliary lookup schemes (generated in \texttt{block setup(...)}) and for the pre-processing of matrix data. The newly introduced functionalities do not introduce additional lookup schemes that are only utilised in the parallel regime and should not contribute to the increase in memory usage. Thus, the increase in peak memory usage per PE comes from the initial BPF functionalities and the rest of the library. For \( p \geq 2 \) we observe good scaling of peak memory usage.

In conclusion, good weak parallel scaling is observed for both the block extraction and matrix concatenation procedures. The overhead in terms of the execution times of the newly introduced functionality is negligible. For \( p = 1 \) the matrix concatenation functionality accounts for 4\% of the \texttt{get block(...)} execution time, which decreases to less than 2\% for \( p \geq 2 \). This is due to the parallelisation overhead in the block extraction functionality which does not exist in the concatenation procedure.

### 7.2.2 Strong Scaling for the \texttt{get block(...) Function}

We investigate the strong scaling of the \texttt{get block(...) function}. In this experiment we fix the problem size at \( N = 15.432 \times 10^6 \) and execute the program on an increasing number of PEs (up to \( p = 16 \)). We define the strong parallel scaling efficiency (in \%) as:

\[
E_s(N,p) = \frac{T(N,1)}{p \cdot T(N,p)} \times 100%,
\]

where \( T(N,1) \) is the execution time on one PE with problem size \( N \) and \( T(N,p) \) is the execution time to complete the same amount of work on \( p \) PEs. Optimal strong scaling would imply that the execution time is halved as we double the number of PEs. In practice there is a strong scalability limit, where adding more resources beyond a certain point will result in a decrease in performance.

The execution time, strong efficiency and the memory usage are shown in Table 7.3. The results are similar to those observed in weak efficiency. The efficiency of the
7.3. PARALLEL PERFORMANCE OF VECTOR FUNCTIONS

<table>
<thead>
<tr>
<th>p</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOF block extraction (s)</td>
<td>157.122</td>
<td>206.309</td>
<td>107.006</td>
<td>61.887</td>
<td>33.197</td>
</tr>
<tr>
<td>Strong Efficiency (%)</td>
<td>100</td>
<td>38.079</td>
<td>36.709</td>
<td>31.736</td>
<td>29.581</td>
</tr>
<tr>
<td>Concatenation (s)</td>
<td>5.294</td>
<td>2.651</td>
<td>1.391</td>
<td>0.779</td>
<td>0.475</td>
</tr>
<tr>
<td>Strong Efficiency (%)</td>
<td>100</td>
<td>99.826</td>
<td>95.164</td>
<td>84.931</td>
<td>69.703</td>
</tr>
<tr>
<td>Peak mem. per PE (GB)</td>
<td>32.188</td>
<td>45.532</td>
<td>23.296</td>
<td>11.666</td>
<td>6.674</td>
</tr>
</tbody>
</table>

Table 7.3: Execution time, strong efficiency (in %) and memory usage for the extraction of DOF blocks and matrix concatenation for a fixed problem size of $N = 15.432 \times 10^6$.

concatenation procedure drops to 85% and 70% on 8 and 16 PEs, respectively. We believe this is due to shared resource contention as discussed previously. Again, we note that the overhead due to concatenation is very small compared to the block extraction times. For $p = 1$ the matrix concatenation procedure accounts for approximately 3% of the `get_block(...)` execution time and less than 2% for $p \geq 2$. There were no strong parallel scaling results presented in [180] for this function.

Under strong scaling, the peak memory usage per PE should roughly half as we double the number of PEs. This is not observed from $p = 1$ to $p = 2$ and is due to parallelisation overhead as discussed in Section 7.2.1. This is consistent with the memory usage results from Table 7.2 (where we observed almost doubling when moving from 1 to 2 PEs). For $p \geq 2$, we observe good parallel scaling in memory usage.

### 7.3 Performance Analysis of the Functions `get_block_vector(...) and return_block_vector(...)`

As discussed in Section 5.3, the `get_block_vector(...)` and `return_block_vector(...)` functions were modified so that the block vector permutation is consistent with the column permutation of the block matrices returned via `get_block(...)`. The function `get_block_vector(...)` performs extraction of DOF vectors followed by a concatenation procedure. The `return_block_vector(...)` function follows the reverse process by splitting the block vector and return them to a global vector. This process is described in Section 5.3.2. Extracting and returning sub-vectors is using existing functionality from the initial BPF implementation. This process requires inter-processor communication to ensure that the vectors are uniformly distributed among the PEs. The concatenation and splitting routines are newly introduced to the BPF. Here we examine their parallel scaling. For this experiment we use the problem
Chapter 7. Parallel Performance Evaluation

7.3.1 Weak Scaling for the Functions get_block_vector(...) and return_block_vector(...) 

The execution times and weak scaling efficiency are presented in Table 7.4. The results shown in this section are for first execution of preconditioner_solve(...) only. We omit the memory usage since this is dominated by the finite element mesh and the Jacobian matrix. The DOF vector extraction and return procedures exhibit very poor weak parallel scaling, but the absolute execution times are so short that these operations should not significantly affect the overall efficiency of an actual parallel linear system solve. The sudden drop in parallel efficiency between $p = 1$ and $p = 2$ is expected due to parallelisation overhead as discussed in Section 7.2.1. The reason for the poor parallel efficiency for $p \geq 2$ is the unfavourable computational work to communication ratio that is inherent to these routines. As with the block extraction routine, the ‘work’ here is simply to load and store data. The data locations for the send/receive calls are precomputed and the routines just fetch the data from memory, send it by MPI to the destination, and store it back to memory. In the case of block extraction, there is a lot more data to be pre-processed, this leads to an increase in parallelism and better parallel scaling, but increases the overall execution time. Here the execution times are very short but with poor parallel scaling due to contention for memory access on SMP architectures. This explains the result observed in Table 7.4.

These results are not consistent with those presented in [180, Tables 4.8 and 4.14], where it was observed that the execution time for get_block_vector(...) and return_block_vector(...) approximately doubles from $p = 1$ to $p = 2$, with nearly

<table>
<thead>
<tr>
<th>$p$</th>
<th>DOF vector extraction (s)</th>
<th>Weak efficiency (%)</th>
<th>Concatenation (s)</th>
<th>Weak efficiency (%)</th>
<th>Split (s)</th>
<th>Weak efficiency (%)</th>
<th>DOF vector return (s)</th>
<th>Weak efficiency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.002</td>
<td>100</td>
<td>0.001</td>
<td>100</td>
<td>0.001</td>
<td>100</td>
<td>0.002</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>0.004</td>
<td>54.714</td>
<td>0.001</td>
<td>66.786</td>
<td>0.001</td>
<td>62.396</td>
<td>0.005</td>
<td>64.888</td>
</tr>
<tr>
<td>4</td>
<td>0.014</td>
<td>17.452</td>
<td>0.001</td>
<td>68.652</td>
<td>0.001</td>
<td>65.140</td>
<td>0.013</td>
<td>75.585</td>
</tr>
<tr>
<td>8</td>
<td>0.020</td>
<td>12.041</td>
<td>0.002</td>
<td>38.763</td>
<td>0.002</td>
<td>63.893</td>
<td>0.019</td>
<td>11.944</td>
</tr>
<tr>
<td>16</td>
<td>0.057</td>
<td>4.328</td>
<td>0.003</td>
<td>24.008</td>
<td>0.003</td>
<td>20.721</td>
<td>0.055</td>
<td>4.166</td>
</tr>
</tbody>
</table>

Table 7.4: Execution time and weak scaling efficiency for the extraction of DOF vectors, concatenation of DOF vectors, splitting block vectors into DOF vectors and returning DOF vectors.
7.3. PARALLEL PERFORMANCE OF VECTOR FUNCTIONS

constant execution times for \( p \geq 2 \). However, the parallel computations in [180] were performed on multiple nodes (with eight core per node). Such architecture may offer better parallel scaling for data transfer intensive jobs. Furthermore, the problem sizes for the vector extraction and return functions studied in [180] were much smaller (by two orders of magnitude) than those considered here. Therefore, it is possible that an effective use of cache at a hardware level is involved. As previously stated, the problem sizes considered here are much larger than a typical FEM simulation to obtain reliable scaling results for the vector concatenation and splitting routines.

Both the concatenation and splitting of vectors exhibit similar weak parallel scaling. There is an initial decrease in performance from \( p = 1 \) to \( p = 2 \), then the efficiency remains approximately the same for \( p = 2 \) and \( p = 4 \), followed by a significant decrease in performance for \( p = 8 \) and \( p = 16 \). This pattern of poor scaling was also observed for the matrix concatenation routine. In this case we believe that the poor scaling is due to a combination of the memory bound nature of the algorithm, memory bandwidth saturation and RAM access contention.

7.3.2 Strong scaling for the functions \texttt{get\_block\_vector(...) and return\_block\_vector(...)}

For the strong scaling results we fix the problem size at \( N = 15.432 \times 10^6 \). The execution times with strong parallel scaling efficiency are presented in Table 7.5. The

<table>
<thead>
<tr>
<th>p</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOF vector extraction (s)</td>
<td>0.059</td>
<td>0.047</td>
<td>0.039</td>
<td>0.049</td>
<td>0.055</td>
</tr>
<tr>
<td>Strong efficiency (%)</td>
<td>100</td>
<td>61.780</td>
<td>37.337</td>
<td>15.020</td>
<td>6.676</td>
</tr>
<tr>
<td>Concatenation (s)</td>
<td>0.019</td>
<td>0.010</td>
<td>0.005</td>
<td>0.004</td>
<td>0.003</td>
</tr>
<tr>
<td>Strong efficiency (%)</td>
<td>100</td>
<td>93.435</td>
<td>91.755</td>
<td>62.286</td>
<td>35.042</td>
</tr>
<tr>
<td>Split (s)</td>
<td>0.019</td>
<td>0.010</td>
<td>0.005</td>
<td>0.004</td>
<td>0.003</td>
</tr>
<tr>
<td>Strong efficiency (%)</td>
<td>100</td>
<td>96.297</td>
<td>94.986</td>
<td>66.187</td>
<td>37.729</td>
</tr>
<tr>
<td>DOF vector return (s)</td>
<td>0.055</td>
<td>0.046</td>
<td>0.046</td>
<td>0.047</td>
<td>0.055</td>
</tr>
<tr>
<td>Strong efficiency (%)</td>
<td>100</td>
<td>60.133</td>
<td>29.586</td>
<td>14.617</td>
<td>6.168</td>
</tr>
</tbody>
</table>

Table 7.5: Execution time and strong efficiency for the extraction of DOF vectors, concatenation of DOF vectors, splitting block vectors into DOF vectors and returning DOF vectors for a fixed problem size of \( N = 15.432 \times 10^6 \).

DOF vector extraction and return procedures exhibit significantly poor scaling. We observe that the execution time starts to increase for \( p \geq 8 \). Although there is double the amount of data to be moved per PE for \( p = 8 \) than for \( p = 16 \), this operation may be optimised at both hardware and software level since arrays in C++ are stored in contiguous memory locations. For the concatenation and split procedures again
we observe good scaling from $p = 1$ to $p = 4$, there is a significant degradation in performance for $p \geq 8$, which is also explained by contention for memory access. There are no strong parallel scaling results presented in [180] for these functions.

In conclusion, the functionalities from the initial BPF version used to extract and return vectors scale very poorly for the type of parallel architecture used in our experiments. However, this is the most efficient implementation since the main computational work is done once per Newton solve in the `block_setup(...)` function, followed by multiple extractions and returns for each DOF vector during the Krylov iterations. The newly implemented vector concatenation and splitting routines are also impacted by memory access contention, but contribute only to a very small increase in the total execution time for the `get_block_vector(...)` and `return_block_vector(...)` functions.

Next, we investigate the parallel scaling performance of the BPF when applied to a 3D problem and demonstrate that the overhead incurred when using the BPF is low when compared the total execution time of the iterative solver.

### 7.4 Parallel Experiments: Flow through a Quarter Annulus in Three Spatial Dimensions

In this section we investigate the parallel scaling of the BPF in terms of execution time and memory usage when using $\hat{P}_{LSC}$ in parallel setting for Example 6.3.2. Here we consider only the SVT-NSE with Reynolds number $Re = 200$. We employ the preconditioned GMRES solver with a relative tolerance of $\varepsilon_K = 10^{-6}$ starting from the zero initial solution guess. The BDF-2 adaptive time stepping is used with a tolerance of $\varepsilon_T = 10^{-4}$ and an initial solution of zero, at each new time step we use the solution computed at the previous time step as the initial guess for Newton’s method.

For the weak scaling experiments, the problem size for one PE is fixed to $N_1 = 149313$ equations (corresponding to mesh size $h = \frac{1}{18}$). As noted previously, it is impossible to construct a problem that has an ideal problem size of $pN_1$ for $p$ PEs due to the features of the used meshes. In Table 7.6 we have presented the mesh size and the associated problem size along with $pN_1$ for comparison. For the strong scaling test we fix the problem size at $N = 1085248$ (corresponding to $h = \frac{1}{32}$), as this is the maximum problem size that can fit on a single PE without memory allocation failures within HYPRE.
7.4. PARALLEL EXPERIMENTS

<table>
<thead>
<tr>
<th>$p$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesh size $h$</td>
<td>$1/18$</td>
<td>$1/23$</td>
<td>$1/29$</td>
<td>$1/36$</td>
<td>$1/45$</td>
</tr>
<tr>
<td>$N$</td>
<td>149313</td>
<td>309928</td>
<td>618892</td>
<td>1180551</td>
<td>2300268</td>
</tr>
<tr>
<td>$pN_1$</td>
<td>149313</td>
<td>298626</td>
<td>597252</td>
<td>1194504</td>
<td>2389008</td>
</tr>
</tbody>
</table>

Table 7.6: Comparison of problem sizes $N$ and the ideal problem size $pN_1$.

We only consider the $\hat{P}_{LSC}$ version of the preconditioner. The main difference between the serial implementations of $\hat{P}_{LSC}$ presented in Chapter 6 and the parallel version is the selection of the coarsening and smoothing strategies. For the AMG approximation of the inverse of the augmented momentum block $\hat{F}$ we use Falgout coarsening [131, pp. 164–165], because it produces coarse grids that are independent of parallelisation. Parallel versions of Ruge-Stüben coarsening (see [131, pp. 162–164]) may generate coarse grids that are dependent on the number of sub-domains and partitioning of the graph which may affect the performance of an iterative solver.

For the AMG cycles we choose $1 \times V(2, 2)$ cycles with parallel hybrid Gauss-Seidel (hybrid GS) smoothing [263, 262, 127]. Hybrid GS is a block Jacobi smoother, where each of the diagonal block is approximately inverted using GS method. It is numerically inferior to the standard GS method but has superior parallelisation properties. For the pressure Poisson operators we use the settings recommended in [180, p. 106]. As noted in Section 6.1 since we are using a SuperLUPreconditioner to invert the $W$-block, SuperLU_DIST is automatically used and no modification of the preconditioner code is required. The only difference here is we must set the appropriate coarsening and smoothing types in the HyprePreconditioners via the functions HYPRE_BoomerAMGSetCoarsenType(...) and HYPRE_BoomerAMGSetRelaxType(...), respectively.

7.4.1 Weak Scaling Evaluation

We evaluate the parallel weak scaling of the iterative solver for the solution of Example 6.3.2. In Table 7.7 we present the average execution times for the preconditioner setup (performed at each Newton iteration), preconditioner application (performed at each GMRES iteration) and the total GMRES solution time. We also include the iteration counts and memory usage. Recall that for weak scaling, optimal performance implies a near constant execution time as $p$ is increased. The significant sub-optimal performance in execution times from 1 to 2 PEs is expected, and is due to the overheads in the parallel versions of the code, see the discussion in Section 7.2.1. For $p \geq 2$, the

---

\[ \text{We note that Falgout coarsening and hybrid GS smoothing are the default options in BoomerAMG.} \]
preconditioner setup times exhibit sub-optimal performance. The application of the preconditioner and the GMRES solver exhibit significant sub-optimal performance. The iteration counts appear to be mildly increasing. The peak memory usage per PE has good weak parallel scaling. To investigate further the cause of the sub-optimal performance, we analyse the scaling of the methods employed in the implementation of \( \hat{P}_{LSC} \). These are tabulated in Table 7.8. We discuss each operation in turn.

<table>
<thead>
<tr>
<th>p</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>149313</td>
<td>309928</td>
<td>618892</td>
<td>1180551</td>
<td>2300268</td>
</tr>
<tr>
<td>Preconditioner Setup (s)</td>
<td>12.316</td>
<td>26.082</td>
<td>35.591</td>
<td>45.020</td>
<td>74.071</td>
</tr>
<tr>
<td>Preconditioner Application (s)</td>
<td>0.970</td>
<td>1.313</td>
<td>1.390</td>
<td>1.648</td>
<td>2.407</td>
</tr>
<tr>
<td>GMRES Solve (s)</td>
<td>29.141</td>
<td>38.738</td>
<td>41.859</td>
<td>52.237</td>
<td>85.150</td>
</tr>
<tr>
<td>Iterations Counts</td>
<td>27.867</td>
<td>27.533</td>
<td>28.138</td>
<td>29.724</td>
<td>33.320</td>
</tr>
<tr>
<td>Peak mem. per PE (GB)</td>
<td>3.556</td>
<td>4.254</td>
<td>4.426</td>
<td>4.465</td>
<td>4.854</td>
</tr>
</tbody>
</table>

Table 7.7: Execution times for preconditioner setup, preconditioner application and GMRES solution time; iteration counts and memory usage (GB) for Example 6.3.2.

<table>
<thead>
<tr>
<th>p</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>149313</td>
<td>309928</td>
<td>618892</td>
<td>1180551</td>
<td>2300268</td>
</tr>
<tr>
<td>block_setup(...) (s)</td>
<td>0.023</td>
<td>0.153</td>
<td>0.218</td>
<td>0.504</td>
<td>1.071</td>
</tr>
<tr>
<td>get_block(...) (s)</td>
<td>4.121</td>
<td>9.063</td>
<td>10.010</td>
<td>10.321</td>
<td>11.663</td>
</tr>
<tr>
<td>( W, \sigma M_j^T ) ( W_j^{-1} M_j, \hat{F} ) (s)</td>
<td>0.201</td>
<td>0.237</td>
<td>0.305</td>
<td>0.659</td>
<td>2.683</td>
</tr>
<tr>
<td>AMG setup for ( \hat{F} ) and ( P ) (s)</td>
<td>6.643</td>
<td>14.930</td>
<td>22.307</td>
<td>28.123</td>
<td>47.337</td>
</tr>
<tr>
<td>SpGEMM (s)</td>
<td>0.377</td>
<td>0.511</td>
<td>0.780</td>
<td>1.126</td>
<td>3.126</td>
</tr>
<tr>
<td>SpMV setup (s)</td>
<td>0.947</td>
<td>1.157</td>
<td>1.887</td>
<td>4.165</td>
<td>7.955</td>
</tr>
<tr>
<td>SuperLU_DIST setup (s)</td>
<td>0.003</td>
<td>0.032</td>
<td>0.122</td>
<td>0.236</td>
<td>0.236</td>
</tr>
</tbody>
</table>

Table 7.8: Execution times for various operations within \( \text{setup(...)} \).

The function \( \text{block\_setup(...)} \) generates the necessary lookup schemes for extracting block matrices and vectors as discussed in Section 5.3. In Section 7.1 we stated that additional lookup schemes are generated if the code is executed in the parallel setting. Therefore an increase in execution time from \( p = 1 \) to \( 2 \) is to be expected. The sub-optimal scaling for \( p \geq 2 \) is also expected and is due to the all-to-all communication required when computing the additional lookup schemes. A similar parallel scaling is observed in the initial BPF implementation, see [180, pp. 114–117]. These additional lookup schemes are constructed only once per Newton solve and only by the top-level master preconditioner. Given that the execution times are short when compared to the overall solution time, this function does not significantly impact the overall efficiency.

The function \( \text{get\_block(...)} \) returns a requested block matrix as detailed in Section 5.3. This function is used within the augmentation preconditioner to extract
blocks corresponding to the Navier-Stokes subsystem and within the LSC preconditioner to extract the blocks $\hat{F}$, $B$ and $B^T$. The extraction of the mass matrices to create the $W$ block and perturbation blocks are treated separately (see below). We observe a significant increase in execution time from $p = 1$ to 2. This is expected and is due to parallelisation overhead, see Section 7.2.1. For $p \geq 2$, nearly optimal scaling is observed. The scaling of this method is comparable with that presented in [180, p. 117, Table 4.23] and is consistent with the performance observed in Table 7.2.

The construction of the $W$ block, perturbation blocks $\sigma M^T \hat{W}^{-1} M_j$ and the augmented momentum block $\hat{F} = F + \sigma \hat{\mathcal{L}}^T \hat{W}^{-1} \hat{\mathcal{L}}$ is done by several nested for loops. This involves the extraction of the mass matrices $M_i$ (using the get_block(...) function), SpGEMM (using Trilinos’ EpetraExt package), and the component-wise addition of two sparse matrices (using an inbuilt oomph-lib function). The results presented in Table 7.8 are for the total execution time.

While these operations have short execution times, poor parallel scaling is observed. This suggests that the code should be further profiled to investigate the source of the sub-optimality. The most time consuming tasks when constructing each perturbation block are the calls to get_block(...) to extract the mass matrices, and two SpGEMM. In general, SpGEMM kernels are known to have poor parallel scaling (see Section 3.5.3), thus this routine may contribute significantly to the sub-optimal execution time. We suggest profiling this routine first for the future work.

The AMG setup phase, performs the construction of the AMG coarse grid operators for the augmented momentum block $\hat{F}$ and the pressure Poisson operator $P = B \hat{M}_u^{-1} B^T$, see [138, p. 135]. This is by far the most time consuming procedure in the preconditioner setup phase and significant sub-optimal performance is observed. This is a third party solver and it is difficult to profile a complex code and pinpoint the sources of performance degradation. Similar parallel performance is observed in [180, p. 117, Table 4.22] where the AMG solver made a significant contribution to the sub-optimal solution time of the iterative solver.

The implementation of the LSC preconditioner requires two SpGEMM for the construction of the pressure Poisson operators. First $\hat{M}_u^{-1} B^T$ is computed, followed by $B(\hat{M}_u^{-1} B^T)$. For this we use the Trilinos’ EpetraExt package \footnote{8} . The execution times shown in Table 7.8 account for both SpGEMM procedures, this includes the time taken to setup and multiply the Epetra_CrsMatrix objects required by EpetraExt.

\footnote{8} Implemented as a wrapper function in the oomph-lib CRDoubleMatrix class. This was not a part of this project.
Sub-optimal performance is observed, similar to that in [180, p. 115, Table 4.17]. 

For SpMV, three sparse matrix operators are required (for $\hat{F}$, $\hat{M}_u^{-1}B^T$ and $B$). We use the Epetra_CrsMatrix class from Trilinos’ Epetra package. Poor parallel scaling is observed. We also observe that the times taken for a SpMV setup are much longer than those observed for SpGEMM. In both SpMV and SpGEMM routines (wrappers to Trilinos routines) the Trilinos’ Epetra_CrsMatrix objects are constructed from oomph-lib’s CRDoubleMatrix. For SpGEMM, this is immediately followed by multiplication, whereas for SpMV the multiplication is required during the preconditioner solve phase. If we consider the setup (the construction of Epetra_CrsMatrix) and application phases separately for both SpGEMM and SpMV (shown in Table 7.9), we observe that the longer setup times for SpMV observed in Table 7.8 are due to constructing the Epetra_CrsMatrix for a much larger matrix. The application time for SpMV is much shorter than that for SpGEMM.

<table>
<thead>
<tr>
<th>p</th>
<th>$B$ to Epetra_CrsMatrix (s)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\hat{M}_u^{-1}B^T$ to Epetra_CrsMatrix (s)</td>
<td>0.068</td>
<td>0.134</td>
<td>0.200</td>
<td>0.282</td>
<td>0.571</td>
</tr>
<tr>
<td></td>
<td>SpGEMM application (s)</td>
<td>0.059</td>
<td>0.071</td>
<td>0.097</td>
<td>0.164</td>
<td>0.250</td>
</tr>
<tr>
<td></td>
<td>$\hat{F}$ to Epetra_CrsMatrix (s)</td>
<td>0.120</td>
<td>0.146</td>
<td>0.231</td>
<td>0.263</td>
<td>0.563</td>
</tr>
<tr>
<td></td>
<td>SpMV application (s)</td>
<td>0.026</td>
<td>0.029</td>
<td>0.030</td>
<td>0.033</td>
<td>0.046</td>
</tr>
</tbody>
</table>

Table 7.9: Execution times under weak scaling for the SpGEMM of $B$ and $(\hat{M}_u^{-1}B^T)$, and the SpMV of $\hat{F}$ to a vector. Also included are the execution times for construction of Epetra_CrsMatrix objects from oomph-lib’s CRDoubleMatrix.

SuperLU_DIST setup performs sparse parallel LU decomposition of the $W$ block. This is a third party code. Poor scaling is observed, however the execution times are very short. This justifies using SuperLUPreconditioner as a subsidiary preconditioner for the $W$ block.

The procedures presented in Tables 7.7 and 7.8 are either performed by the third party libraries or routines implemented as a part of the BPF. The third party libraries are used for the AMG solvers, SpGEMMs, SpMVs, and SuperLU_DIST for the inversion of the $W$ block. The performance of the third party libraries generally exhibit poor weak parallel scaling for our test case and is similar to the performance presented in [180].

The BPF routines include block_setup(...) and get_block(...), both have been modified during this project, adding new functionality as discussed in Section 5.3.

\footnote{Implemented in the oomph-lib wrapper class MatrixVectorProduct. This was not a part of this project.}
We have demonstrated in Section 7.2 that the overhead due to the matrix concatenation is negligible and the execution time is dominated by the existing block extraction functionality. We have stated in Section 7.1 that block_setup(...) has been modified to generate new lookup schemes which do not require communication to construct and are small. Thus, the execution time of the block_setup(...) function is not significantly affected by the modifications, and the parallel scaling of this method is dominated by the existing functionality. Overall, the BPF routines have either short execution times or have good parallel scaling. The execution time for both block_setup(...) and get_block(...) are comparable with the execution times presented in [180, p. 117, Table 4.23].

In Figure 7.1 we present the contributions (in %) of different parts of the preconditioner setup phase to the overall execution time. The execution time associated the overhead of using the BPF is denoted ‘BPF’, this includes the calls to block_setup(...), get_block(...), and the construction of the $W, \sigma M_i \hat{W}^{-1} M_j$ and $\hat{F}$. We observe that the overhead incurred by using the BPF accounts for about 26% and 21% of the total execution time for the preconditioner setup phase for $p = 8$ and 16, respectively. The relative contribution of the BPF to the total execution time is decreasing as $p$ is increased. We observed in Table 7.8 that the main contribution to BPF overhead times is the get_block(...) function, which scales nearly optimally. The main source of sub-optimal performance in the total execution times come from setting up the AMG coarse grid operators, which is third party software.

![Figure 7.1: Execution times of different contributions (phases) (in %) of the preconditioner setup phase for Example 6.3.2 under weak scaling.](image-url)
Next, we analyse the operations within `preconditioner_solve(...).` The execution times are presented in Table 7.10.

<table>
<thead>
<tr>
<th>p</th>
<th>N</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>149313</td>
<td>309928</td>
<td>618892</td>
<td>1180551</td>
<td>2300268</td>
</tr>
<tr>
<td><code>get_block_vector(...)</code> (s)</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
<td>0.003</td>
<td>0.004</td>
<td></td>
</tr>
<tr>
<td><code>return_block_vector(...)</code> (s)</td>
<td>0.001</td>
<td>0.001</td>
<td>0.002</td>
<td>0.002</td>
<td>0.003</td>
<td></td>
</tr>
<tr>
<td><code>SpMV application</code> (s)</td>
<td>0.035</td>
<td>0.040</td>
<td>0.042</td>
<td>0.048</td>
<td>0.076</td>
<td></td>
</tr>
<tr>
<td><code>AMG application</code> (s)</td>
<td>0.934</td>
<td>1.270</td>
<td>1.344</td>
<td>1.590</td>
<td>2.315</td>
<td></td>
</tr>
<tr>
<td><code>SuperLU_DIST application</code> (s)</td>
<td>0.000</td>
<td>0.001</td>
<td>0.002</td>
<td>0.005</td>
<td>0.009</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.10: Execution times for the operations within the `preconditioner_solve(...)` function.

Similar execution times are observed for the functions `get_block_vector(...)` and `return_block_vector(...).` We observe good parallel scaling for $p = 1$ to $p = 4$, sub-optimal performance is observed for $p = 8$ and $p = 16$. This is due to contention for memory access as discussed in Section 7.3.1. The execution times for this function are only a small proportion of the overall preconditioner application time.

Four applications of SpMV are performed for $\hat{F}$, $\hat{M}_u^{-1}B^T$, $B\hat{M}_u^{-1}$ and $B^T$. Good parallel scaling is observed for $p = 1$ to $p = 8$, followed by a significant degradation in scaling performance for $p = 16$.

The AMG application phase (see [138, p. 135]) accounts for the majority of the execution time in the `preconditioner_solve(...)` function, the scaling is similar to the SpMV application – while the scaling is good up to $p = 8$, there is a significant increase in execution times for $p = 16$.

The execution time for application of `SuperLU_DIST` show good scaling up to $p = 4$, the performance decreases significantly for $p = 8$ and $p = 16$.

The overhead incurred by using the BPF in the solve phase is from extracting and returning the block vectors via the functions `get_block_vector(...)` and `return_block_vector(...)`. All other operations utilise the third party software. Figure 7.2 depicts the contributions (in %) of different phases in the preconditioner application phase. We observe from Figure 7.2 that the BPF overhead accounts for less than 0.5% of the total execution times during the preconditioner application phase.

Most of the execution times presented in Table 7.8 and Table 7.10 exhibit a mild growth for $p = 1$ to $p = 4$, followed by a significant increase for $p = 8$ and $p = 16$. This behaviour was also observed for the weak parallel scaling of the `get_block(...), `get_block_vector(...)` and `return_block_vector(...)` functions presented in Sections 7.2.1 and 7.3.1. For these routines we believe that the cause of this poor scaling is due to the contention for shared resources, in particular, memory bandwidth and
the RAM access. These routines are memory bound since the workload consists of (unavoidable) read/write operations from/to different locations in memory. When executed on an SMP architecture, the RAM access contention becomes a major bottleneck.

### 7.4.2 Strong Scaling Evaluation

For strong scaling evaluation, we fix the problem size at $N = 1085248$. The execution times (in seconds) for the preconditioner setup, preconditioner application, the GMRES solver, iteration counts and memory usage (in GB) are presented in Table 7.11. The execution times presented in Table 7.11 exhibit significant sub-optimal scaling. The peak memory usage per PE should roughly half as the number of PE is doubled, this was observed.

<table>
<thead>
<tr>
<th>$p$</th>
<th>Preconditioner Setup (s)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>94.062</td>
<td>81.065</td>
<td>58.334</td>
<td>40.561</td>
<td>29.791</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6.681</td>
<td>4.552</td>
<td>2.448</td>
<td>1.511</td>
<td>1.091</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>229.788</td>
<td>146.800</td>
<td>80.532</td>
<td>50.157</td>
<td>33.927</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>22.596</td>
<td>13.417</td>
<td>7.245</td>
<td>3.609</td>
<td>1.913</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.11: Execution times for preconditioner setup, preconditioner application and Trilinos AztecOO GMRES solve, followed by iteration counts and memory usage for Example 6.3.2 under strong scaling, $N = 1085248$.

To investigate the sub-optimal performance, the preconditioner setup operations
are presented in Table 7.12. For block_setup(...) we observe that the execution time increases from \( p = 1 \) to \( p = 2 \) due to the cost of parallelisation. For \( p = 2 \) to \( p = 16 \), very poor scaling is observed. For get_block(...), the execution time is roughly the same for \( p = 1 \) and \( p = 2 \), again, due to the overhead incurred by parallelisation. For \( p = 4 \) to \( p = 16 \), near optimal scaling is observed. This is consistent with the results presented in Section 7.2.2. The execution times for the construction of the \( W \) block, the perturbation blocks \( \sigma M_i^T \tilde{W}^{-1} M_j \) and the augmented momentum block \( \tilde{F} \) exhibit very poor strong scaling. A mild increase in execution time is observed for \( p = 8 \) followed by its tripling for \( p = 16 \). This can be justified by the fact that the mass matrices are very small, with only 4761 rows. For such small problem sizes, scaling to even a moderate number of PEs will lead to a decrease in performance. We note that the execution times also increase for SuperLU_DIST setup, where the \( W \) is the size of the mass matrix.

<table>
<thead>
<tr>
<th>( p )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>block_setup(...) (s)</td>
<td>0.109</td>
<td>0.580</td>
<td>0.407</td>
<td>0.424</td>
<td>0.273</td>
</tr>
<tr>
<td>get_block(...) (s)</td>
<td>30.038</td>
<td>32.001</td>
<td>17.696</td>
<td>9.218</td>
<td>4.043</td>
</tr>
<tr>
<td>( W, \sigma M_i^T \tilde{W}^{-1} M_j, \tilde{F} ) (s)</td>
<td>0.986</td>
<td>0.598</td>
<td>0.347</td>
<td>0.600</td>
<td>1.892</td>
</tr>
<tr>
<td>AMG setup for ( \tilde{F} ) and ( P ) (s)</td>
<td>43.257</td>
<td>40.826</td>
<td>33.167</td>
<td>25.628</td>
<td>20.442</td>
</tr>
<tr>
<td>SpGEMM (s)</td>
<td>2.824</td>
<td>1.893</td>
<td>1.552</td>
<td>1.049</td>
<td>0.874</td>
</tr>
<tr>
<td>SpMV setup (s)</td>
<td>6.742</td>
<td>5.052</td>
<td>4.835</td>
<td>3.505</td>
<td>2.101</td>
</tr>
<tr>
<td>SuperLU_DIST setup (s)</td>
<td>0.045</td>
<td>0.115</td>
<td>0.130</td>
<td>0.138</td>
<td>0.167</td>
</tr>
</tbody>
</table>

Table 7.12: Execution times for various operations within setup(...) under strong scaling, \( N = 1085248 \).

We analyse further the operations within the preconditioner_solve(...) function, with the execution times presented in Table 7.13. The functions get_block_vector(...) and return_block_vector(...) generally have very poor scaling, as they are memory bound. But they account for only a small proportion of the solve time. We do not expect these results to have good parallel scaling on a single SMP node. These results are consistent with Table 7.5. Poor scaling is also observed for operations utilising the third party software.

<table>
<thead>
<tr>
<th>( p )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_block_vector(...) (s)</td>
<td>0.004</td>
<td>0.003</td>
<td>0.002</td>
<td>0.003</td>
<td>0.003</td>
</tr>
<tr>
<td>return_block_vector(...) (s)</td>
<td>0.006</td>
<td>0.004</td>
<td>0.002</td>
<td>0.002</td>
<td>0.003</td>
</tr>
<tr>
<td>SpMV application (s)</td>
<td>0.249</td>
<td>0.139</td>
<td>0.073</td>
<td>0.044</td>
<td>0.037</td>
</tr>
<tr>
<td>AMG application (s)</td>
<td>6.422</td>
<td>4.406</td>
<td>2.367</td>
<td>1.439</td>
<td>1.042</td>
</tr>
<tr>
<td>SuperLU_DIST application (s)</td>
<td>0.000</td>
<td>0.002</td>
<td>0.003</td>
<td>0.004</td>
<td>0.005</td>
</tr>
</tbody>
</table>

Table 7.13: Execution times for various operations within the preconditioner_solve(...) function under strong scaling, \( N = 1085248 \).

We conclude this section with the percentage charts for the operations utilised
7.4. PARALLEL EXPERIMENTS

in the \texttt{setup(...)} and \texttt{preconditioner\_solve(...)} phases. In Figure 7.3 the overhead from using the BPF is denoted by ‘BPF’, and is the sum of the execution time taken for \texttt{block\_setup(...)}, \texttt{get\_block(...)}, the construction of the $W$ block, the perturbation matrices $\sigma M_i W^{-1} M_j$, and the augmented momentum block $\widehat{F}$.

This overhead is reduced as $p$ increases. This implies that the AMG setup, SpGEMM and SpMV setup have poorer strong scaling than the BPF related procedures.

![Figure 7.3: Execution times of different contributions (phases) (in %) of the preconditioner setup phase for Example 6.3.2 under strong scaling, with $N = 1085248$.](image)

In Figure 7.4 ‘BPF’ denotes the sum of the execution times taken for the calls to \texttt{get\_block\_vector(...)} and \texttt{return\_block\_vector(...)}. The overhead incurred from using the BPF is less than 0.5% of the preconditioner application time.
Figure 7.4: Execution times of different contributions (phases) (in %) of the preconditioner solve phase for Example 6.3.2 under strong scaling, with $N = 1085248$.

7.5 Summary

In this section we have evaluated the parallel performance of the BPF under weak and strong scaling. In Sections 7.2 and 7.3 we have shown that the overheads due to the newly introduced functionalities from Section 5.3 are negligible when compared to the execution time of the existing functionalities that is still utilised in the current version of the BPF. The newly introduced routines for the concatenation of matrices and vectors and the splitting of vectors show good parallel scaling up to $p = 4$. For $p > 4$, there is a significant deterioration in parallel performance, we believe this to be caused by the shared resource contention as the newly introduced routines are strongly memory bound. Such cases are known to scale poorly on SMP architectures where access to main memory is shared among the active of PEs. We believe that better parallel scaling for these routines would be observed as we scale to multiple nodes or run the code on a distributed memory machine.

In Section 7.4 we have shown that the overhead incurred from using the BPF is either negligible (for \texttt{get\_block\_vector(...)} and \texttt{return\_block\_vector(...)} or have good parallel scaling (such as \texttt{get\_block(...)}). The \texttt{block\_setup(...)} function exhibits poor parallel scaling in both weak and strong scaling experiments. This is due to the deployed all-to-all communication (see [180, p. 110]). Overall, the BPF routines are computationally cheap when compared to the total execution time of the iterative solver, where the main source of sub-optimal performance comes from the
poor parallel scaling of the third party software such as HYPRE BoomerAMG, Trilinos Epetra EpetraExt and the Trilinos AztecOO GMRES solver.

Due to the complexity of the third party software utilised in the implementation of the augmentation block preconditioner, it is difficult to predict the weak scaling as \( p \) increases and multiple nodes are employed. For example, because intra-node communication is generally faster than inter-node communication, we would expect that the weak parallel scaling efficiency to decrease as we scale from one node to multiple nodes. This was not observed for HYPRE BoomerAMG in [180, p. 107], where the greatest increase in execution time was (unexpectedly) observed for \( p \leq 8 \) on a single node, and near-constant execution time was observed for \( p = 8 \) to \( p = 64 \) using multiple nodes. Thus, further work is required here to draw a definite conclusion.
Chapter 8

Conclusions and Future Work

This dissertation describes a novel preconditioning strategy for the finite element (FE) discretisation of the incompressible Navier-Stokes equations (NSE) with weakly imposed boundary conditions. Analysis on the eigenvalue bounds of the preconditioned operator was performed and verified numerically. The robustness of the preconditioner was investigated through numerical experiments on a representative range of test problems in two- and three-spatial dimensions. Parallel scaling of the methodology was also investigated. In this chapter we summarise the contributions of this work and propose a number of possible avenues for its extension.

8.1 Project Contributions

We have successfully developed and analysed an augmentation type preconditioner for the finite element discretisation of the incompressible Navier-Stokes equations with weakly imposed boundary conditions by Lagrange multipliers. This work was motivated by Muddle [180], where an optimal preconditioner of this type was developed for the case of pseudo-elastic problem with weakly imposed boundary conditions. We generalised the problem-specific choice for the $W$ block from [180] and extended the preconditioning methodology from a symmetric indefinite case (with a symmetric positive definite $(1, 1)$ block) to a general case where the $(1, 1)$ block is non-symmetric and indefinite with a saddle point structure (the Navier-Stokes problem).

The analysis of the preconditioned operator presented in Section 4.6 demonstrates that it has a large proportion of unit eigenvalues, together with a small number (equal to the number of nodes at the constrained boundary) of well clustered eigenvalues which implies good convergence of the resulting iterative solver. This is numerically
verified in Section 6.2.1.1, where even tighter numerical bounds for the eigenvalues were observed.

In Chapter 6 we investigate the robustness and efficiency of the augmentation preconditioner through a range of numerical experiments. The exact version of the augmentation preconditioner exhibits very robust performance which is in line with the analytical results, with low and bounded iteration counts for all problem parameters considered. However the wall-clock execution times scale sub-optimally. In order to obtain an optimal solver, we relaxed the exact inverses of certain blocks in the augmentation preconditioner. We presented the inexact least squares commutator (LSC) version of the preconditioner, where the inverses of the scalar problems were approximated by algebraic multigrid (AMG). We numerically determined and documented two sets of AMG parameters which work well for problems in two- and three-spatial dimensions. This was particularly relevant in the case of the momentum block, where we have demonstrated that the application of AMG in a black-box manner to the entire modified momentum block yields significantly superior overall performance of the preconditioner to other possible block approximations previously discussed in the literature. As a result of this, the inexact preconditioning strategy led to a significant reduction in execution time with near-optimal scaling with respect to the problem size. The analytical characterisation of the eigenvalues for the exact preconditioner implies that the iteration counts are independent of the angle of rotation in the problems with straight outflow boundaries. This was not observed for the inexact version of the preconditioner. We have shown numerically the effect that the rotation angle has on the AMG approximation of the modified momentum block.

The preconditioners were developed and tested in oomph-lib’s parallel block preconditioning framework (BPF), which was significantly upgraded and partially re-implemented during the course of this project. The aim of the BPF is to facilitate the rapid development of block preconditioners within oomph-lib. The main advantage of the BPF is the ability to re-use existing block preconditioners as subsidiary preconditioners in a hierarchical fashion. In Chapter 5 we documented the issues with the initial version of the BPF and the new functionalities implemented during this project. This involved a re-implementation of all public functions to include new numbering schemes due to the introduction of additional data types that facilitate the compatibility of preconditioners with different degrees of freedom type granularities. Subsequently we were able to consolidate multiple implementations of the same block preconditioners,
such as \texttt{oomph-lib}'s FSI block preconditioner \cite{118}. As a result, \texttt{oomph-lib} currently has one of the most sophisticated and functional block preconditioning frameworks compared to other FEM modelling software.

In Section 7 we have demonstrated the weak and strong scalability of the block preconditioning framework in terms of the execution time and memory usage. In particular, we have shown that the overhead due to the new functionalities implemented in this project is negligible.

8.2 Future Work

We have identified three main avenues of possible extension for this research: the preconditioner analysis and implementation, additional numerical tests and software modifications.

8.2.1 Extensions to the Preconditioner Analysis and Implementation

The analysis of the exact preconditioner could be extended to include eigenvalue bounds for non-unit eigenvalues, at least in the symmetric case (with zero Reynolds number) by generalising the results of Gould and Simoncini \cite{95}, obtained for an indefinite matrix with an indefinite (1,1) block, to the preconditioned case with an augmentation preconditioner.

For the two-dimensional case studies shown in Section 6.2 we observed unexpected results for the inexact version of the preconditioner using the stress divergence viscous term (SDVT) of the Navier-Stokes equations. When using AMG to approximate the inverse of the augmented momentum block, we observed a significant reduction in iteration counts for the strength of dependence parameter $\theta \gtrsim 2/3$. This was not observed for the simple viscous term (SVT) case, nor in 3D. This effect should be thoroughly investigated. A starting point would be to examine both the setup and solve diagnostic information from \texttt{HYPRE BoomerAMG} via the \texttt{oomph-lib} wrapper function \texttt{HyprePreconditioner::amg\_print\_level()}, for both the SDVT and SVT of the NSE for $\theta = 0.5$ and $\theta = 2/3$. Tests should also be carried out for the standard Navier-Stokes problem to determine if the unexpected behaviour is due to the augmentation of the momentum block. These tests may help locate the cause of such behaviour and provide more insight.
It would be worth to consider an alternative subsidiary preconditioner for the augmented Navier-Stokes block due to the following reasons: our experiments in [258] show a significant increase in iteration counts for the exact LSC version of the augmentation preconditioner when the Navier-Stokes problem with the no-penetration condition ($u \cdot \hat{n} = 0$) is solved. In addition, the iteration counts for the LSC preconditioner are dependent on the discretisation parameter $h$ when the domain is discretised by triangular and tetrahedral elements [180, p. 85]. In such cases it is difficult to bound the eigenvalues of the exact LSC preconditioner. An alternative to the LSC preconditioner is the modified augmented Lagrangian preconditioner due to Benzi, Olshanskii and Wang [30]. We conjecture that some of these problems may not exist for this choice, as the AL preconditioner is a $(1,1)$ Schur complement-based. Then it may be possible to extend the analysis to an exact ‘double-augmented’ preconditioner by combining the analysis from this project and the analysis of the modified AL preconditioner [30]. The AL preconditioner is also more robust than the LSC preconditioner with respect to the Reynolds number. The performance of the double-augmented preconditioner should be compared with the results presented in this thesis.

### 8.2.2 Extensions to Numerical Simulations

In Chapter 6 we presented the numerical results for problems posed on structured meshes. This work can be extended by testing the augmentation preconditioner on more complex domains and meshes. One possibility would be to consider problems relevant to biomechanical engineering such as an iliac bifurcation, thus simulating an arterial junction as shown in the oomph-lib tutorial [123]. The bifurcation means that no matter how we rotate the domain, the outflow boundary $\partial \Omega_O$ will not align with the Cartesian axes unless the branches are at a $90^\circ$ angle. Thus, for most realistic problems the axially aligned parallel outflow boundary condition is not applicable and imposing boundary conditions via Lagrange multipliers is essential in this context. The complexity of the flow profile in this case would naturally require discretisations with unstructured and adaptively refined grids.

Our generalised choice for the $W$ block makes no assumption on the underlying weakly imposed boundary condition. Thus it would be possible to extend the preconditioning methodology to other weakly imposed boundary conditions and to other partial differential equations. This may require a replacement of the subsidiary preconditioners for the $(1,1)$ block. We have already considered the weakly imposed
no-penetration condition \cite{258}, which suggested that the LSC preconditioner may not be the best choice in this case.

The parallel performance of the augmentation preconditioner was investigated only on distributed memory architectures with up to 16 cores on a single node (thus, essentially an SMP architecture), with a limited set of test problems. More parallel tests are needed, here we outline some potential additional parallel experiments:

- Tests should be performed over multiple nodes with an increasing number of cores.

- Parallel performance tests should be conducted on a wider range of problems, for two- and three-dimensional domains and different discretisations (especially with adaptively refined grids).

- Although we have good evidence that the overhead incurred from the new functionalities of the BPF is negligible in terms of execution times, additional performance analysis for the BPF should be considered, using problems detailed in \cite{180}, and have the weak scaling efficiency compared to these existing results.

### 8.2.3 Extensions for the Software Implementation

Finally, we list potential further optimisations and improvements for the software utilised in oomph-lib’s iterative linear solver:

- Linear operators such the constraint blocks $L$ in the augmentation preconditioner, the discrete gradient operator $B^T$, and the discrete divergence operator $B$ in the LSC preconditioner are extracted at every Newton iteration to form the preconditioner. Associated preconditioning operators such as matrix-vector products are also assembled per Newton iteration.

  Assembling the linear operators only when the problem changes will likely lead to a reduction in execution time but at the cost of an increase in peak memory usage. This should be beneficial in cases where the storage of the linear blocks in the preconditioner is small compared to the overall memory usage.

- Sparse matrix-matrix (SpGEMM) multiplication, required for the assembly of the pressure Poisson operator $B\tilde{M}_u^{-1}B^T$ is a known performance bottleneck in the parallel setting \cite{37}. One potential way of improvement is to write a customised
8.2. FUTURE WORK

SpGEMM product function, for example by following the ideas from \([36, 37]\). An alternative approach as suggested in \([180, pp. 185–186]\) would be to explicitly construct the Poisson operator on the pressure space. In this instance, further investigation is required to show how this impacts the performance of the LSC preconditioner.

- Matrix-vector product setup and application scale sub-optimally. The Optimized Sparse Kernel Interface (OSKI) package \([252, 189]\) is shown to produce significant speed up over Trilinos Epetra sparse matrix vector product in \([145]\). The scaling of OSKI used within oomph-lib should be investigated.

- We observed in Chapter 7 that Trilinos’s AztecOO GMRES solver scaled sub-optimally, especially in the weak scaling experiments for \(p = 16\) (see Table 7.7). The Trilinos Belos \([21]\) package supersedes the AztecOO package with many improvement for modern HPC architectures and is likely to be more efficient than AztecOO. We recommend investigating the scaling of Trilinos Belos package.
Appendix A

Supporting Numerical Results

A.1 Numerical Results for Example 6.2.1

In this section we provide additional numerical results for the two-dimensional steady state Poiseuille flow through a rotated domain considered in Example 6.2.1. 

A.1.1 AMG Strength of Dependence Parameter

In Section 6.2.1 we experimentally determined the AMG strength of dependence parameter $\theta$ for the augmented momentum matrix $\hat{F} = F + \hat{L}^T \hat{W} \hat{L}$, based on Reynolds number $Re = 100$. Here we present the same study with Reynolds number $Re = 200$. The average iteration counts taken for the GMRES preconditioned with the inexact LSC preconditioner ($GMRES + \hat{P}_{LSC}$) to converge to a relative tolerance of $\varepsilon_K = 10^{-6}$ with an initial guess of zero for the simple viscous term (SVT) and stress divergence viscous term (SDVT) of the Navier-Stokes equations (NSE) are presented in Tables A.1 and A.2, respectively. The wall-clock times (in seconds) taken to set up the augmented preconditioner are given in Tables A.3 and A.4, and finally the solution times are presented in Tables A.5 and A.6. The same conclusions hold for $Re = 200$ as for $Re = 100$, namely, for the SVT-NSE, $\theta = 0.25$ yield the lowest iteration counts (shown in Table A.1), and the shortest coarsening and solution times (shown in Table A.3 and Table A.5, respectively). For the SDVT-NSE, there is a sudden decrease in iteration counts after $\theta = 0.5$. Further tests with the SDVT-NSE were performed with $\theta = [0.66, 0.8]$ in increments of 0.001, the iteration counts and total execution times are depicted in Figure 6.2 and Figure 6.3, respectively, the results are discussed in Section 6.2.1. For the SDVT-NSE, we choose $\theta = 0.668$. 

276
A.1. NUMERICAL RESULTS FOR Example 6.2.1

Table A.1: The average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver for varying strength of dependence parameter $\theta$ in the AMG approximation of the $\hat{F}$ block within the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and $Re = 200$.

<table>
<thead>
<tr>
<th>$\theta$</th>
<th>576</th>
<th>2304</th>
<th>9216</th>
<th>36864</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>22.3</td>
<td>22.7</td>
<td>24.7</td>
<td>32</td>
<td>41.7</td>
</tr>
<tr>
<td>0.5</td>
<td>26.7</td>
<td>24.7</td>
<td>29</td>
<td>36</td>
<td>48.7</td>
</tr>
<tr>
<td>0.75</td>
<td>25.7</td>
<td>24.7</td>
<td>33.3</td>
<td>43.7</td>
<td>63.7</td>
</tr>
</tbody>
</table>

Table A.2: The average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver for varying strength of dependence parameter $\theta$ in the AMG approximation of the $\hat{F}$ block within the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SDVT-NSE with $\alpha = 30^\circ$ and $Re = 200$.

<table>
<thead>
<tr>
<th>$\theta$</th>
<th>576</th>
<th>2304</th>
<th>9216</th>
<th>36864</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>21.3</td>
<td>35.7</td>
<td>67.7</td>
<td>154</td>
<td>$&gt;300$</td>
</tr>
<tr>
<td>0.5</td>
<td>22</td>
<td>28</td>
<td>47.7</td>
<td>126.7</td>
<td>$&gt;300$</td>
</tr>
<tr>
<td>0.67</td>
<td>21</td>
<td>24.3</td>
<td>28.3</td>
<td>37</td>
<td>42</td>
</tr>
<tr>
<td>0.75</td>
<td>21.3</td>
<td>25.7</td>
<td>31.7</td>
<td>41.3</td>
<td>46</td>
</tr>
</tbody>
</table>

Table A.3: The average preconditioner setup times (s) taken by $\hat{P}_{LSC}$ for varying strength of dependence parameter $\theta$ in the AMG approximation of the $\hat{F}$ block within the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and $Re = 200$.

<table>
<thead>
<tr>
<th>$\theta$</th>
<th>576</th>
<th>2304</th>
<th>9216</th>
<th>36864</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>0.03</td>
<td>0.04</td>
<td>0.14</td>
<td>0.56</td>
<td>2.16</td>
</tr>
<tr>
<td>0.5</td>
<td>0.03</td>
<td>0.04</td>
<td>0.14</td>
<td>0.58</td>
<td>2.42</td>
</tr>
<tr>
<td>0.75</td>
<td>0.01</td>
<td>0.03</td>
<td>0.14</td>
<td>0.59</td>
<td>2.35</td>
</tr>
</tbody>
</table>

Table A.4: The average preconditioner setup times (s) taken by $\hat{P}_{LSC}$ for varying strength of dependence parameter $\theta$ in the AMG approximation of the $\hat{F}$ block within the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SDVT-NSE with $\alpha = 30^\circ$ and $Re = 200$.

<table>
<thead>
<tr>
<th>$\theta$</th>
<th>576</th>
<th>2304</th>
<th>9216</th>
<th>36864</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>0.01</td>
<td>0.05</td>
<td>0.21</td>
<td>0.91</td>
<td>3.75</td>
</tr>
<tr>
<td>0.5</td>
<td>0.01</td>
<td>0.04</td>
<td>0.17</td>
<td>0.66</td>
<td>2.63</td>
</tr>
<tr>
<td>0.67</td>
<td>0.01</td>
<td>0.04</td>
<td>0.15</td>
<td>0.59</td>
<td>2.59</td>
</tr>
<tr>
<td>0.75</td>
<td>0.01</td>
<td>0.04</td>
<td>0.15</td>
<td>0.67</td>
<td>2.57</td>
</tr>
</tbody>
</table>

Table A.5: The average solution times (s) taken by the GMRES+$\hat{P}_{LSC}$ solver for varying strength of dependence parameter $\theta$ in the AMG approximation of the $\hat{F}$ block within the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and $Re = 200$.

<table>
<thead>
<tr>
<th>$\theta$</th>
<th>576</th>
<th>2304</th>
<th>9216</th>
<th>36864</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\theta = 0.25$</td>
<td>0.02</td>
<td>0.06</td>
<td>0.18</td>
<td>0.99</td>
<td>5.35</td>
</tr>
<tr>
<td>$\theta = 0.5$</td>
<td>0.03</td>
<td>0.06</td>
<td>0.24</td>
<td>1.28</td>
<td>9.06</td>
</tr>
<tr>
<td>$\theta = 0.75$</td>
<td>0.02</td>
<td>0.05</td>
<td>0.32</td>
<td>1.95</td>
<td>13.25</td>
</tr>
</tbody>
</table>
A.1.2 Damping Parameter for the Jacobi Smoother

In Section 6.2.1 we experimentally determine the smoothing method with the best performance (shortest execution time) for the AMG approximation for the augmented momentum matrix \( \hat{F} = F + \hat{L}^T \hat{W} \hat{L} \). In this context, we experimentally determine the optimal damping factor \( \omega \) for Jacobi’s method. We use Ruge-Stüber coarsening with strength of dependence \( \theta = 0.25 \) for the SVT-NSE and \( \theta = 0.668 \) for the SDVT-NSE with \( 1 \times V(2,2) \) pre/post smoothing cycles. For the pressure Poisson operators we use the parameters given in Preconditioner 6.4. The iteration counts are shown in Table A.7 and Table A.8 for the SVT-NSE and the SDVT-NSE, respectively. The execution times are given in Table A.9 and Table A.10 for the SVT-NSE and the SDVT-NSE, respectively. We observe that \( \omega = 1 \) leads to both the lowest iteration counts and shortest execution times. This justifies our choice on pages 193–195.

\[
\begin{array}{c|cccc}
\theta & 0.25 & 0.5 & 0.67 & 0.75 \\
\hline
576 & 0.02 & 0.01 & 0.01 & 0.01 \\
2304 & 0.11 & 0.07 & 0.06 & 0.07 \\
9216 & 0.92 & 0.56 & 0.30 & 0.29 \\
36864 & 13.17 & 7.39 & 1.33 & 2.54 \\
147456 & - & - & 8.61 & 9.26 \\
\end{array}
\]

Table A.6: The average solution times (s) taken by the GMRES+\( \hat{P}_{LSC} \) solver for varying strength of dependence parameter \( \theta \) in the AMG approximation of the \( \hat{F} \) block within the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SDVT-NSE with \( \alpha = 30^\circ \) and \( \text{Re} = 200 \).

\[
\begin{array}{c|ccc|ccc}
\omega & 0.2 & 0.4 & 0.6 & 0.8 & 1 & \\
\hline
9216 & 64.7 & 47.3 & 43.7 & 36.7 & 27.7 \\
36864 & 68.7 & 47.3 & 43.7 & 36.7 & 27.7 \\
147456 & 76.5 & 54.5 & 48.5 & 44.5 & 34.7 \\
\hline
9216 & 69.7 & 48.3 & 43.7 & 28.7 & 27.7 \\
36864 & 75 & 57.3 & 40 & 36.7 & 35.7 \\
147456 & 81.7 & 50.3 & 46.7 & 45 &
\end{array}
\]

Table A.7: The average number of iterations taken by the GMRES+\( \hat{P}_{LSC} \) solver for varying Jacobi damping parameter \( \omega \) for the AMG approximation of the \( \hat{F} \) block within the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SVT-NSE with \( \alpha = 30^\circ \) and \( \text{Re} = 100 \) (left), \( \text{Re} = 200 \) (right).
A.1. NUMERICAL RESULTS FOR Example 6.2.1

<table>
<thead>
<tr>
<th>$\omega$</th>
<th>9216</th>
<th>36864</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>63</td>
<td>64.5</td>
<td>82</td>
</tr>
<tr>
<td>0.4</td>
<td>40.7</td>
<td>43</td>
<td>57.5</td>
</tr>
<tr>
<td>0.6</td>
<td>34</td>
<td>36.5</td>
<td>50</td>
</tr>
<tr>
<td>0.8</td>
<td>30.7</td>
<td>33.5</td>
<td>46.5</td>
</tr>
<tr>
<td>1</td>
<td>29.3</td>
<td>32.5</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\omega$</th>
<th>9216</th>
<th>36864</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>73.3</td>
<td>77.3</td>
<td>87</td>
</tr>
<tr>
<td>0.4</td>
<td>46.3</td>
<td>50.7</td>
<td>61</td>
</tr>
<tr>
<td>0.6</td>
<td>37.3</td>
<td>42.7</td>
<td>50.5</td>
</tr>
<tr>
<td>0.8</td>
<td>32.3</td>
<td>39.7</td>
<td>46</td>
</tr>
<tr>
<td>1</td>
<td>30.3</td>
<td>38.3</td>
<td>44</td>
</tr>
</tbody>
</table>

Table A.8: The average number of iterations taken by the GMRES+$\hat{P}_{LSC}$ solver for varying Jacobi damping parameter $\omega$ for the AMG approximation of the $\hat{F}$ block within the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SDVT-NSE with $\alpha = 30^\circ$ and $Re = 100$ (left), $Re = 200$ (right).

<table>
<thead>
<tr>
<th>$\omega$</th>
<th>9216</th>
<th>36864</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>0.63</td>
<td>2.93</td>
<td>12.46</td>
</tr>
<tr>
<td>0.4</td>
<td>0.43</td>
<td>2.02</td>
<td>8.95</td>
</tr>
<tr>
<td>0.6</td>
<td>0.37</td>
<td>1.76</td>
<td>9.03</td>
</tr>
<tr>
<td>0.8</td>
<td>0.37</td>
<td>1.92</td>
<td>7.58</td>
</tr>
<tr>
<td>1</td>
<td>0.33</td>
<td>1.77</td>
<td>7.84</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\omega$</th>
<th>9216</th>
<th>36864</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>0.76</td>
<td>3.04</td>
<td>16.05</td>
</tr>
<tr>
<td>0.4</td>
<td>0.50</td>
<td>2.18</td>
<td>11.90</td>
</tr>
<tr>
<td>0.6</td>
<td>0.47</td>
<td>1.91</td>
<td>10.21</td>
</tr>
<tr>
<td>0.8</td>
<td>0.41</td>
<td>1.76</td>
<td>10.49</td>
</tr>
<tr>
<td>1</td>
<td>0.40</td>
<td>1.73</td>
<td>9.59</td>
</tr>
</tbody>
</table>

Table A.9: The execution times (s) taken by the GMRES+$\hat{P}_{LSC}$ solver for varying Jacobi damping parameter $\omega$ for the AMG approximation of the $\hat{F}$ block within the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SVT-NSE with $\alpha = 30^\circ$ and $Re = 100$ (left), $Re = 200$ (right).

<table>
<thead>
<tr>
<th>$\omega$</th>
<th>9216</th>
<th>36864</th>
<th>147456</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>0.76</td>
<td>3.04</td>
<td>16.05</td>
</tr>
<tr>
<td>0.4</td>
<td>0.50</td>
<td>2.18</td>
<td>11.90</td>
</tr>
<tr>
<td>0.6</td>
<td>0.47</td>
<td>1.91</td>
<td>10.21</td>
</tr>
<tr>
<td>0.8</td>
<td>0.41</td>
<td>1.76</td>
<td>10.49</td>
</tr>
<tr>
<td>1</td>
<td>0.40</td>
<td>1.73</td>
<td>9.59</td>
</tr>
</tbody>
</table>

Table A.10: The execution times (s) taken by the GMRES+$\hat{P}_{LSC}$ solver for varying Jacobi damping parameter $\omega$ for the AMG approximation of the $\hat{F}$ block within the LSC preconditioner, applied to Example 6.2.1. The results are presented for the SDVT-NSE with $\alpha = 30^\circ$ and $Re = 100$ (left), $Re = 200$ (right).
Bibliography


