Impact of Nested Preconditioning in hp −Adaptive Continuous Galerkin Solutions of the Poisson Problem

Saleh Saharkhiz

Thesis submitted to the University of Ottawa
in partial Fulfillment of the requirements for the

Master of Applied Science

Department of Mechanical Engineering
Faculty of Engineering
University of Ottawa

Abstract

This thesis investigates the performance of several preconditioning strategies for the Conjugate Gradient (CG) method applied to finite element discretization of the Poisson equation in one and two dimensions. Both uniform and adaptive refinement approaches are considered, including p −refinement, h −refinement, and the combined hp −adaptivity strategy. The study evaluates nine different preconditioners, including classical methods such as Jacobi and SSOR, low order finite element-based (order 1) preconditioners (FEM1, F3T, F4R, F4CR), p −multigrid techniques, and multi-preconditioned conjugate gradient (MPCG) schemes.

The analysis is carried out systematically by examining three aspects: iteration counts, execution times, and condition numbers, with particular attention given to eigenvalue distributions. Results show that condition number alone is not a sufficient predictor of solver performance. Instead, the clustering and spread of eigenvalues play a decisive role in determining the convergence rate of CG. Preconditioners such as FEM1 in one dimension, and F3T or F4R in two dimensions, demonstrate the ability to cluster eigenvalues into compact groups, which significantly reduce iteration counts compared to Jacobi and SSOR. In contrast, preconditioners like F4CR exhibit very small condition numbers but still require many more iterations due to poor eigenvalue clustering.

The uniform refinement results highlight the reliability of the p −multigrid method, whose iteration counts remain nearly independent of system size and polynomial degree. However, its runtime efficiency only becomes evident at large matrix sizes, due to the overhead of inter-level transfers at smaller scales. The adaptive refinement studies reveal distinct behaviors: p −adaptivity consistently produced moderate system sizes with stable iteration counts, h −adaptivity led to prohibitively expensive runtimes despite moderate number of iterations, and hp −adaptivity achieved the best balance, requiring roughly the same number of iterations as p −adaptivity while generating much larger systems.

Overall, the results emphasize that the effectiveness of preconditioners must be judged not only by condition number reduction, but also by their impact on eigenvalue distribution and clustering, which ultimately governs CG performance.

Acknowledgements

First, I want to thank my supervisor, Professor Catherine Mavriplis. She is not only a great scientist with international recognition, but also someone I will always remember as a mother. Her kindness is not only because she cares, but because of her big heart. From her I learned that in science we should not rush with excitement but move forward with calm steps.

I still remember one of her sentences that stayed with me. After my Gas Dynamics exam, she asked me: “How was the exam?” I told her: “It was challenging, but I liked it because of the challenge.” She answered: “Yes, we are here (at the university) for challenges.” From that moment, I could continue my path even when difficulties came, and this sentence always gave me strength.

I would also like to thank my research team members, in alphabetical order: Aktham, Amit, Bharathwaj, and Ming. We shared many laughs and memories during this journey. Without you, this trip would not have been so enjoyable. I hope our friendship will stay strong, and I wish to see each of you in great positions in the future. One funny memory: in the first weeks, I registered for a Machine Learning course. During class, I realized it looked very similar to a Linear Control course from my bachelor studies that I really hated. So, in the middle of class while the professor of this course was teaching, I dropped the course! Later Amit asked me what I had done, and I explained the whole story. We laughed so much that day we almost exploded from laughing.

I also thank my office-mates, in alphabetical order: Blake, Kazem, Payam, and Shahin. Because of you, I never felt alone while working in the office.

A special thanks goes to Payam Parvizi. In one August afternoon, he spent three hours teaching me programming logic. From that day, I moved forward in my research with much more confidence and energy. His hard work and determination will always be a role model for me.

I want to thank my family for their endless support. Thank you, Mom (Mojgan), for being with me even on the hardest days. Thank you, Dad (Farhad), for being like a mountain in my life and

for being my role model. I could not become a world wrestling champion, but I tried to make you proud in the scientific path I chose.

My twin brother Saber deserves a very special thank you. When I first came to Canada, I was alone, but he guided me and helped me to adapt. Thank you, Saber, for everything. I also thank my younger brother, Salar, who helped me take the first steps on this path. At a time when I was exhausted from trying, he was the one who encouraged me to begin this journey. I still remember the day university entrance exam results were about to be announced: Salar, you were stuck in the bathroom and insisted that nobody reveal the results until you came out, calm and prepared. In that exact moment, I played the famous Snoop Dogg song and shouted through the door that you had been accepted into electrical engineering at Shahid Beheshti University. You were frustrated, and I could not stop laughing, but it remains one of my favorite memories with you.

I would also like to sincerely thank my aunt, Zohreh, who traveled from Germany to Canada in the summer of 2024. During her stay, she cared for us with warmth and attention, and her presence brought a sense of ease to our home. Before she arrived, I jokingly told her, “Just make sure your suitcase doesn’t disappear like mine once did.” On her very first trip to Canada, that exact thing happened: her suitcase, filled with gifts, was lost forever (LOL). When we realized what had happened, I couldn’t stop laughing at the irony while she was understandably upset. Even through that experience, her patience, kindness, and sense of responsibility never changed, and I am truly grateful for everything she did for us during her visit.

I also want to thank my bachelor professors, Dr. Mohammad Hamed Hekmat and Dr. Mohammad Ali Mehrpouya. They were the ones who introduced me to the beauty of scientific thinking and taught me how to approach problems critically. Without their influence, I might have continued on my athletic path and, quite possibly, been preparing for Olympic wrestling or triathlon competitions today. Instead, the elegance and depth of the scientific world captivated me, and I remain grateful that they guided me toward a journey I am still proud to follow.

Finally, I want to mention a very special member of our family, Mr. Gholi. He is an African Grey parrot. He is always talking, making jokes nonstop, and sometimes fighting with my father! For me, he has a very special place. Many times, when I was struggling with dark and difficult days, he was Gholi who reminded me to keep going, to smile, and to enjoy life. Maybe it is better to say that he came to my life to show me what living really means. Sometimes I think I wish I was a Gholi.

Contents

Abstract ii
Acknowledgements iii
List of Figures ix
List of Tables xvi
Nomenclature xviii
List of Abbreviations xxii
Chapter 1 : Introduction 1
1.1 The Need for Numerical Methods 2
1.2 A Brief Overview of Numerical Methods for PDEs 3
1.2.1 Finite Difference Methods (FDM) 3
1.2.2 Finite Volume Methods (FVM) 4
1.2.3 Finite Element Method (FEM) 5
1.3 Advantages of the Galerkin Method for Elliptic Problem 6
1.4 Characteristics of the Discontinuous Galerkin Method 8
1.5 Adaptive Mesh Refinement (AMR) 8
1.6 Error Estimation in Adaptive Finite Element Methods 10
1.6.1 A Priori Error Estimates 10
1.6.2 A Posteriori Error Estimates. 10
1.7 Iterative Methods 11
1.7.1 Classical Iterative Methods (CIMs) 12
1.7.2 Krylov Subspace Methods (KSMs) 13
1.8 Preconditioning 15
1.8.1 Single Preconditioning 16
1.8.2 Multiple Preconditioned Conjugate Gradient (MPCG) 17
1.8.3 Multigrid Preconditioning 18
Chapter 2 : Literature Review 21
Chapter 3 : Methodology 26
3.1 Variational Formulation 26
3.2 Finite Element Discretization 26
3.3 Weighted Residual Approximation 27
3.3.1 Galerkin Method as a Special Case 28

3.4 Shape Functions and Interpolation 29
3.4.1 One-Dimensional Lagrange Shape Functions 30
3.4.2 Two-Dimensional Lagrange Shape Functions (Quadrilateral Elements) 32
3.4.3 Barycentric Interpolation and Weight Computation 33
3.5 Numerical Integration 33
3.5.1 Gaussian Quadrature in $1D$ 34
3.5.2 Gaussian Quadrature in $2D$ (Quadrilateral Elements) 34
3.5.3 Application to Stiffness Matrix and Load Vector 35
3.6 Assembly of Global Matrices 36
3.6.1 Local-to-Global Mapping and Implementation 36
3.7 Error Estimation 37
3.7.1 Spectral Expansion 38
3.8 Adaptivity implementation 45
3.8.1 hp $-$Aadaptivity in $1D$: 45
3.8.2 p-Adaptivity in $1D$: 47
3.8.3 h $-$Adaptivity in $1D$: 48
3.8.4 p $-$Adaptivity in $2D$: 49
3.9 Solver 56
3.9.1 CG Convergence Behavior 57
3.9.2 Sensitivity to Eigenvalue Distribution in CG Convergence 58
3.9.3 Preconditioned Conjugate Gradient (PCG) 59
3.9.4 Multi-Preconditioned Conjugate Gradient (MPCG) Method 62
3.9.5 SSOR Iterative Method 64
3.10 Preconditioners Used 66
3.10.1 Jacobi (Diagonal) Preconditioner 66
3.10.2 Symmetric Successive Over-Relaxation (SSOR) preconditioner 67
3.10.3 Finite Element Order-1 (FEM1) Preconditioner 68
3.10.4 p-Multigrid Preconditioner 72
Chapter 4 : Results and Discussion 76
4.1 Problem Setup for 1D Cases 76
4.2 1D Uniform Mesh Refinement 76
4.2.1 *p*- Uniform Refinement 77
4.2.2 *h* - Uniform Refinement 85
4.3 1D Adaptive Mesh Refinement 92
4.3.1 *p* -, *h* -, *hp* - Refinement 93

4.4 Problem Setup for 2D Cases 107
4.5 2D Uniform Refinement Mesh 107
4.5.1 *P* - Uniform Mesh Refinement 109
4.5.2 *h* - Uniform Mesh Refinement 119
4.5.3 2D Adaptive Mesh Refinement 129
Chapter 5 : Conclusion 136
5.1 Summary 136
5.2 : Future Work 137
Appendix A: Additional Material 139
Bibliography 140

List of Figures

Figure 1.1: Liquid-fraction contours depicting the temperature-driven melting field in the helical PCM storage unit. The melt-fraction distribution reflects the underlying steady-state conduction solution of the Poisson equation that defines the temperature field within the system [2]. 1
Figure 1.2: (a) Flow streamlines and cross-sectional vortex structures showing how the two inlet streams (indicated by the incoming-flow arrows) enter the junction and interact to form a three-dimensional mixing region. The counter-rotating vortices visible in the cross-sections illustrate the swirl generated as the flows merge. (b) The corresponding normalized temperature field, demonstrating how the mixing pattern redistributes thermal energy within the junction. Both the incompressible pressure field and the temperature distribution are obtained from Poisson-type equations, making the Poisson equation fundamental to resolving the coupled fluid–thermal behavior captured in this figure [3]. 2
Figure 1.3: Illustration of one- and two-dimensional Cartesian grids with uniform spacing (Δx, Δy), n the finite difference method. Solid markers indicate boundary nodes, while hollow markers represent interior computational nodes, with the point (i, j) denoting a generic interior location [10]. 4
Figure 1.4: Two-dimensional Cartesian grid in the finite volume method, illustrating the control volume around node $\psi i, j$. Filled symbols represent nodal points, open symbols denote face-centered values, and Δx and Δy indicate the grid spacing. The arrows Fw, Fe, Fs, and Fn represent the fluxes through the west, east, south, and north faces of the control volume, respectively. Conservation is enforced by balancing these fluxes across the control volume boundaries, so that the flux leaving one cell enters its neighbor. 5
Figure 1.5: 2D hp-adaptive illustration. In (A) h-adaptive refinement, a single element is subdivided into four smaller child elements, reducing the element size. In (B) p-adaptive refinement, the polynomial order of the basis functions is increased, enriching the approximation space without changing the mesh. In (C) hp-adaptive refinement, both strategies are combined: the element is subdivided, and the polynomial order is raised simultaneously. 7
Figure 1.6: Shock reflection on an oblique wedge using an Adaptive Mesh Refinement technique [25]. 9
Figure 1.7: Steps of an AMG V-cycle: smoothing → restriction → coarse-grid solve → interpolation [60]. 20
Figure 1.8: p −Multigrid coarsening step: Reducing a second-order quadrilateral element to first-order. 20

Figure 3.1: Reference domain (A) 2D: 0,1 × 0,1, $fi\ i = 1, \dots, 4$ are defined on edges; (B) 1D: 0,1, with $fi\ i = 1,2$ specified at the endpoints. In both cases ni denotes the outward unit normal vector associated with the corresponding boundary segment or point. 27
Figure 3.2: One-dimensional Lagrange basis functions constructed using equidistant nodes over the interval [0, 1]. Plot Ashows the basis functions for polynomial degree $p = 1$,while plot (B) illustrates the basis for $p = 5$. 31
Figure 3.3: Two-dimensional Lagrange basis functions defined over a reference quadrilateral element with uniformly spaced nodes in the domain $[0,1] \times [0,1]$. Plot (A) displays the basis functions for polynomial degree $p = 1$,, while Plot (B) corresponds to degree $p = 2$. 31
Figure 3.4: Steps of the assembly process: local stiffness matrices and local vectors are computed for each element. 37
Figure 3.5: (A) Mapping between local and global node numbering in a structured 2D quadrilateral mesh. (B) Schematic illustration of the global stiffness matrix assembly, where local element contributions are inserted and summed at positions corresponding to shared global nodes. 37
Figure 3.6: Spectrum decay trend of modal coefficients: (A) $\sigma > 1$ indicates a good-quality element, while (B) $\sigma \leq 1$ indicates a poor-quality element with faster decay. 45
Figure 3.7: Illustrating the decision-making process in an hp −adaptivity framework. The algorithm identifies poor and good elements based on error estimation, then selectively refines the mesh (h −adaptivity) or increases the polynomial order (p −adaptivity) to efficiently satisfy the desired tolerance. 46
Figure 3.8: Comparison of solutions before and after *hp*−adaptivity to the one-dimensional Poisson problem $-u" = x50$. (A) shows the numerical solution on a uniform mesh of three elements, each with $p = 2$ ($L2\ error = 2.3540 \times 10 - 5$). (B) shows the adaptive case, where the third element is split into two sub-elements with $p = 2$, while the first and second elements are enriched to $p = 3$ resulting in improved accuracy ($L2\ error = 1.0353 \times 10 - 5$). 47
Figure 3.9: Comparison of solutions before and after p −adaptivity for the one-dimensional Poisson problem $-u" = x50$. (A) shows the numerical solution on a uniform mesh of 3 elements, each with polynomial order $p = 2$ ($L2\ error = 2.3540 \times 10 - 5$). (B) shows the result after three successive p −adaptive steps, where the third element (identified by $\sigma > 1$) has been progressively enriched up to polynomial order $p = 5$, while the first and second elements remain at $p = 2$ ($L2\ error = 1.0429 \times 10 - 5$). 48
Figure 3.10: Comparison of solutions before and after h −Adaptivity for the one-dimensional Poisson problem $-u" = x50$. (A) shows the initial solution on a uniform mesh of three elements, each with polynomial degree of $p = 2$ ($L2\ error = 2.3540 \times 10 - 5$). (B) shows the result

after one successive h −adaptive step, where the third element is subdivided into two elements, both with polynomial order $p = 2$, while the first and second elements remain unchanged ($L2\ error\ =\ 1.1445\ \times\ 10-5$). 49
Figure 3.11: Numerical solution of the 2D Poisson equation $-\nabla 2ux, y = fx, y$ in the domain $\Omega = [0, 1]2$, subject to Dirichlet boundary conditions and fx, y is provided in Appendix A. (A) Initial mesh of 6 × 6 elements with $p = 3$ ($L2\ error\ =\ 1.1\ \times\ 10-3$). (B) After the first adaptation part 2 refined to $p = 4$, part 1 remains at $p = 3$ ($L2\ error\ =\ 5.43\ \times\ 10-4$). (C) After the second adaptation, three regions with $p = 3,4,5$ ($L2\ error\ =\ 5.72\ \times\ 10-5$). (D) After four adaptations, four regions with $p = 3,4,5,6$ ($L2\ error\ =\ 1.92\ \times\ 10-5$)..... 51
Figure 3.12: A 2D mesh with four elements, highlighting neighboring pairs that share an interface. Element 1 and Element 2 share a vertical edge, while Element 2 (Ωm) and Element 4 (Ωs) share a horizontal edge denoted by Γ. The polynomial orders satisfy $ps > pm$. 52
Figure 3.13: Global node numbering for the structured 2D mesh used in the test case. All elements are linear ($p\ =\ 1$), and shown with red nodes, except Element 2, which is upgraded to quadratic order ($p\ =\ 2$) and displayed using blue nodes. The blue nodes represent the additional mid-edge degrees of freedom introduced by the higher-order element. At interfaces where Element 2 connects to lower-order neighbors, overlapping red and blue nodes indicate shared locations, ensuring continuity across elements with different polynomial degrees..... 54
Figure 3.14: Flowchart illustrating the implementation of p −adaptivity in two dimensions. The process involves detecting non-conforming interfaces, identifying master and slave elements, evaluating shape functions at hanging nodes, and constructing the global interpolation matrix ℝGlobal 56
Figure 3.15: Illustration of the nested preconditioning approach. The outer solver handles the main system, while each preconditioning step uses an inner iterative solver. 61
Figure 3.16: Sparsity patterns of (A) the stiffness matrix A (nz=28464) from the adaptive mesh in Figure 3.11(C), and (B) its Jacobi preconditioner $MJacobi$ (nz=704). In contrast to A, the Jacobi preconditioner retains only diagonal entries, producing an extremely sparse matrix with far fewer nonzero elements. 67
Figure 3.17: Sparsity patterns for (A) the stiffness matrix A (nz = 28,464) corresponding to the mesh in Figure 3.11(C), and (B) the SSOR preconditioning matrix (nz = 57,636). The SSOR preconditioner increases the number of nonzero entries compared to A, reflecting the additional fill-in introduced by the forward and backward sweeps. 68
Figure 3.18: Construction of the F4R variant for the FEM1 preconditioner. (A) node numbering of a single third-order quadrilateral element. (B) re-interpretation of the same element as a set of first-order four-node square elements for preconditioner assembly. 69

Figure 3.19: Example of the F3T (first-order, three-node triangle) variant used in the FEM1 preconditioner. (A) shows the node numbering for a third-order quadrilateral element, while (B) illustrates its subdivision into multiple first-order triangular elements for use in the preconditioner assembly..... 70
Figure 3.20: Example of the F4CR (first-order, four-node coarse rectangle) variant of the FEM1 preconditioner. (A) Node numbering of a third-order quadrilateral element. (B) The four corner vertices (1, 4, 13, 16) are selected to form the coarse rectangular element used in the stiffness calculation, while the internal nodes are connected through an identity mapping to ensure continuity in the preconditioner assembly. 71
Figure 3.21: Sparsity patterns for (A) the global stiffness matrix A (nz = 28,464) corresponding to the mesh shown in Figure 3.11(c), and the associated preconditioner matrices: (B) F4R variant (nz = 3,766), (C) F4CR variant (nz = 868), and (D) F3T variant (nz = 3,456). Each preconditioner significantly reduces the number of nonzero entries compared to A, with the degree of sparsity reflecting the underlying coarsening strategy F4R and F4CR retain only the connectivity of selected coarse nodes, while F3T preserves additional couplings due to its triangular-element construction..... 72
Figure 3.22: Sparsity patterns of the coarse-level matrices obtained after four successive p −adaptive refinements. Panels (A–D) correspond to the coarse-level counterparts of the solutions in Figure 3.14(A–D), respectively. These patterns represent the coarse operators used in the p −multigrid cycle rather than the original stiffness matrices. 75
Figure 4.1: Example of p-uniform refinement on a uniform mesh with three elements. (A) corresponds to polynomial degree 2, while (B) corresponds to polynomial degree 3. The blue points indicate the additional nodes introduced by the higher-order basis functions. 77
Figure 4.2: Example of h-uniform refinement. In (A), the mesh consists of three elements with polynomial degree 2. In (B), the mesh is refined into 13 elements while maintaining the same polynomial degree 2..... 77
Figure 4.3: Iteration counts for different preconditioners under p-uniform refinement for 1D..... 79
Figure 4.4: Runtime comparison of Jacobi, FEM1, SSOR, and FEM1+Jacobi preconditioners with increasing matrix size in p-uniform refinement for 1D. 80
Figure 4.5: Variation of condition numbers with polynomial degree for preconditioned and Stiffness Matrix CG solvers under p −uniform refinement with three elements for 1D..... 82
Figure 4.6: Eigenvalue distributions under different preconditioners for p-refinement with matrix size 17×17, obtained after four refinement steps. (A) Unpreconditioned system. (B) Jacobi preconditioner. (C) SSOR preconditioner. (D) FEM1 preconditioner. (E) MPCG preconditioner for 1D..... 83

Figure 4.7: Eigenvalue distributions of the FEM1 preconditioned system under p −uniform refinement. (A) corresponds to the initial case with polynomial degree $p = 3$. (B) shows the refined case after uniformly increasing the polynomial degree of all elements to $p = 7$ for 1D. 84
Figure 4.8: Finite element solutions under *p*-uniform refinement with three elements. (A) Solution with polynomial degree $P = 3$ (error = 2.21×10^{-5}). (B) Solution with polynomial degree $P = 7$ (error = 1.65×10^{-6}). In both cases, refinement is performed by uniformly increasing the polynomial order of all elements for 1D. 85
Figure 4.9: Iteration counts for different preconditioners under h −uniform refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 87
Figure 4.10: Runtime comparison of Jacobi, FEM1, SSOR, and FEM1+Jacobi preconditioners with increasing matrix size in h −uniform refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 88
Figure 4.11: Variation of condition numbers with polynomial degree for preconditioned and Stiffness Matrix CG solvers under h −uniform refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 90
Figure 4.12: Eigenvalue distributions under different preconditioners for h −uniform refinement with matrix size 85×85, obtained after four refinement steps (43 elements). (A) Stiffness Matrix. (B) Jacobi preconditioner. (C) SSOR preconditioner. (D) FEM1 preconditioner. (E) MPCG preconditioner for 1D. 91
Figure 4.13: Finite element solutions under h −refinement in 1D. (A) Initial solution with 3 elements, each of polynomial degree $p = 2$ (error = 2.35×10^{-5}). (B) Solution after four refinement steps, resulting in 43 elements, each of polynomial degree $p = 2$ (error = 1.09×10^{-8}), highlighting the substantial improvement in accuracy in 1D. 92
Figure 4.14: Iteration counts for different preconditioners under adaptivity: (A) p −adaptivity, (B) h −adaptivity, and (C) hp −adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 96
Figure 4.15: Runtime comparison of Jacobi, FEM1, SSOR, and MPCG (FEM1+Jacobi) preconditioners with increasing matrix size under adaptivity: (A) p −adaptivity, (B) h −adaptivity, and (C) hp −adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 100
Figure 4.16: Variation of condition numbers with polynomial degree for preconditioned and unpreconditioned CG solvers under adaptivity, starting from 3 elements of polynomial degree 2: (A) p −adaptivity, (B) h −adaptivity, and (C) hp −adaptivity for 1D. 104
Figure 4.17: Sparsity patterns and corresponding solutions for adaptive refinements with a 9×9 stiffness matrix. (A) p −adaptivity after the 4th refinement, with its corresponding solution

shown in (D). (B) h −adaptivity after the 2nd refinement, with its corresponding solution shown in (E). (C) hp −adaptivity after the 1st refinement, with its corresponding solution shown in (F). The subfigures (D–F) display the numerical solutions compared against the exact (analytical) solution for the respective adaptive cases. All refinements start from an initial mesh of 3 elements with polynomial degree $p = 2$. 105
Figure 4.18: Eigenvalue distributions of the preconditioned and unpreconditioned system (Stiffness Matrix) under different types of adaptivity in size of matrix 9×9. Columns correspond to (A) p −adaptivity, (B) h −adaptivity, and (C) hp −adaptivity, while rows show the spectra for the unpreconditioned system (Stiffness Matrix) and for the Jacobi, SSOR, FEM1, and MPCG preconditioners in 1D. 106
Figure 4.19: Example of p −uniform refinement on a two-dimensional uniform mesh together with the corresponding solutions of the model problem. (A) shows the mesh with polynomial degree 3, and (B) shows the mesh with polynomial degree 4, obtained by uniformly increasing the order of all elements by one. (C) and (D) display the numerical solutions for polynomial degrees 3 and 4, respectively. 108
Figure 4.20: Example of h-uniform refinement on a two-dimensional mesh together with the corresponding solutions of the model problem. (A) shows the mesh with 6 elements in both the $x-$ and y −directions, each of polynomial degree 3. (B) shows the uniformly refined mesh with 7 elements in both directions, again with polynomial degree 3. (C) and (D) display the numerical solutions corresponding to the meshes in (A) and (B), respectively. 109
Figure 4.21: Iteration counts for different preconditioners under p −uniform refinement for 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 111
Figure 4.22: Runtime comparison of nine preconditioners with increasing matrix size in p −uniform refinement for 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 114
Figure 4.23: Variation of condition numbers with polynomial degree for preconditioned and unpreconditioned CG solvers under adaptivity, starting from 36 elements of polynomial degree 3 under p −uniform for 2D. 117
Figure 4.24: Eigenvalue distributions of the preconditioned and unpreconditioned system (Stiffness Matrix) under p −uniform of size 1849×1849 corresponding to fourth adaptation fin 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. (A) (A) shows the unpreconditioned system A. (B) Jacobi preconditioned system $\mathrm{MJacobi} - 1A$. (C) SSOR preconditioned system $\mathrm{MSSOR} - 1A$. (D) F4R preconditioned system $\mathrm{MF4R} - 1A$. (E) F4CR preconditioned system $\mathrm{MF4CR} - 1A$. (F) F3T preconditioned system $\mathrm{MF3T} - 1A$. (G) Multi-

PCG1 preconditioned system MMPCG1 − 1A. (H) Multi-PCG2 preconditioned system MMPCG2 − 1A. (I) Multi-PCG3 preconditioned system MMPCG3 − 1A. 118
Figure 4.25: Iteration counts for different preconditioners under h −uniform refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 121
Figure 4.26: Runtime comparison of nine preconditioners with increasing matrix size in h −uniform refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 123
Figure 4.27: Variation of condition numbers with polynomial degree for preconditioned and unpreconditioned CG solvers under adaptivity, starting from 36 elements of polynomial degree 3 under h −uniform for 2D. 126
Figure 4.28: Eigenvalue distributions of the nine preconditioned and unpreconditioned system (Stiffness Matrix) under h −uniform corresponding to the 7th refinement for 2D. (A) shows the unpreconditioned system A. (B) Jacobi preconditioned system $MJacobi - 1A$. (C) SSOR preconditioned system $MSSOR - 1A$. (D) F4R preconditioned system $MF4R - 1A$. (E) F4CR preconditioned system $MF4CR - 1A$. (F) F3T preconditioned system $MF3T - 1A$. (G) Multi-PCG1 preconditioned system $MMPCG1 - 1A$. (H) Multi-PCG2 preconditioned system $MMPCG2 - 1A$. (I) Multi-PCG3 preconditioned system $MMPCG3 - 1A$. 128
Figure 4.29: Iteration counts for nine preconditioners under p −adaptive refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 130
Figure 4.30: Runtime comparison of nine preconditioners with increasing matrix size under adaptivity p −adaptivity in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 132
Figure 4.31: Variation of condition numbers with polynomial degree for preconditioned and unpreconditioned CG solvers under adaptivity, starting from 36 elements of polynomial degree 3 under p −adaptivity for 2D. 134
Figure 4.32: Eigenvalue distributions of the preconditioned and unpreconditioned system (Stiffness Matrix) under p −adaptivity of size 568×568 corresponding to first adaptation. (A) shows the unpreconditioned system A. (B) Jacobi preconditioned system $MJacobi - 1A$. (C) SSOR preconditioned system $MSSOR - 1A$. (D) F4R preconditioned system $MF4R - 1A$. (E) F4CR preconditioned system $MF4CR - 1A$. (F) F3T preconditioned system $MF3T - 1A$. (G) Multi-PCG1 preconditioned system $MMPCG1 - 1A$. (H) Multi-PCG2 preconditioned system $MMPCG2 - 1A$. (I) Multi-PCG3 preconditioned system $MMPCG3 - 1A$. 135

List of Tables

Table 4.1: Iterative performance of preconditioned CG solvers for increasing matrix sizes under p −uniform refinement, where the polynomial degree of all three elements is raised uniformly from 2 to 12 for 1D. 78
Table 4.2: Runtimes of different preconditioners with increasing matrix size in p −uniform refinement for 1D. 79
Table 4.3: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and the unpreconditioned system under p-uniform with 3 elements for 1D. 81
Table 4.4: Iterative performance of preconditioned CG solvers for growing matrix sizes in h −uniform refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 86
Table 4.5: Runtimes of different preconditioners with increasing matrix size in h −uniform refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 87
Table 4.6: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and the Stiffness Matrix under h −uniform refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 89
Table 4.7: Iterative performance of preconditioned CG solvers for growing matrix sizes in p −adaptive mesh refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 94
Table 4.8: Iterative performance of preconditioned CG solvers for growing matrix sizes in h −adaptive mesh refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 94
Table 4.9: Iterative performance of preconditioned CG solvers for growing matrix sizes in hp −adaptive mesh refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 95
Table 4.10: Runtimes of different preconditioners with increasing matrix size in p −adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 97
Table 4.11: Runtimes of different preconditioners with increasing matrix size in h −adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 98
Table 4.12: Runtimes of different preconditioners with increasing matrix size in hp −adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 99

Table 4.13: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and unpreconditioned system (Stiffness matrix) under p −adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 101
Table 4.14: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and unpreconditioned system (Stiffness matrix) under h −adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 102
Table 4.15: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and the unpreconditioned system (Stiffness matrix) under hp −adaptivity, starting from an initial mesh of 3 elements with polynomial degree $p = 2$. 103
Table 4.16: Iterative performance of preconditioned CG solvers for growing matrix sizes in p −uniform refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 110
Table 4.17: Runtimes of different preconditioners with increasing matrix size in p −uniform refinement for 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 113
Table 4.18: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and the unpreconditioned system (Stiffness Matrix) under p −uniform refinement for 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 116
Table 4.19: Iterative performance of preconditioned CG solvers for growing matrix sizes in h −uniform refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 120
Table 4.20: Runtimes of different preconditioners with increasing matrix size in h −uniform refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 122
Table 4.21: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and the unpreconditioned system (Stiffness Matrix) under h −uniform in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 125
Table 4.22: Iterative performance of preconditioned CG solvers for growing matrix sizes in p −adaptive refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 129
Table 4.23: Runtimes of nine preconditioners with increasing matrix size in p −adaptivity in 2D starting from an initial mesh of 36 elements with polynomial degree $p = 3$. 131
Table 4.24: Comparison of condition numbers for preconditioned CG solvers using nine preconditioners and the unpreconditioned system (Stiffness Matrix) under p −adaptivity starting from 36 elements of polynomial degree 3 for 2D. 133

Nomenclature

Roman Characters

$\boldsymbol{a_n}$	Modal coefficients in spectral (Legendre) expansion.
$\boldsymbol{A}$	Global stiffness matrix of the finite element system.
$\boldsymbol{A^{-1}}$	Inverse of A.
$\boldsymbol{C}$	Constant in exponential modal-decay model.
$\boldsymbol{F}$	Global load vector; also used for body force function.
$\boldsymbol{f}$	Source term in the Poisson equation.
$\boldsymbol{F_{w,e,s,n}}$	Face fluxes (west, east, south, north) in finite volume methods.
$\boldsymbol{h}$	Element size.
$\boldsymbol{hp}$	Combined refinement strategy using both h and p refinement.
$\boldsymbol{i, j}$	Node indices; summation or iteration counters.
$\boldsymbol{k}$	Iteration counter in iterative methods.
$\boldsymbol{K}$	Local stiffness matrix (element stiffness).
$\boldsymbol{L^2(\Omega)}$	Standard L^2 (Euclidean) norm over the computational domain.
$\boldsymbol{M}$	Preconditioning matrix (Jacobi, SSOR, FEM1, etc.).

$\boldsymbol{n}$	Polynomial mode index; number of nodes along an element edge; or dimension size.
$\boldsymbol{N}$	Number of degrees of freedom (DOFs) or number of modal points.
$\boldsymbol{p}$	Polynomial degree of finite element basis functions.
$\mathbf{r}$	Residual vector.
$\boldsymbol{u}$	Exact solution of the Partial Differential Equations.
$\boldsymbol{u}_{\mathrm{h}}$	Finite element approximation of the solution.
$\bar{\boldsymbol{u}}_{\mathrm{i}}$	Coefficients of basis-function expansion in the finite element methods weak form.
$\boldsymbol{U}$	Vector of unknown solution coefficients.
$\boldsymbol{x}, \boldsymbol{y}$	Spatial coordinates.
$\boldsymbol{z}$	Preconditioned residual vector in Preconditioning Conjugate gradient method.

Greek Symbols

$\boldsymbol{\alpha}$	Intercept in the linearized modal–decay model.
$\boldsymbol{\beta}$	Slope in the linearized modal–decay model.
$\boldsymbol{\Delta x}$	Mesh spacing in the x −direction.
$\boldsymbol{\Delta y}$	Mesh spacing in the y −direction.
$\boldsymbol{\Gamma}$	Interface between elements (non-conforming hp interfaces).
$\boldsymbol{\varepsilon}$	Numerical error or tolerance (e.g., truncation or quadrature error).
$\boldsymbol{\eta}$	Local coordinate in the y −direction.
$\boldsymbol{\lambda_{max}, \lambda_{min}}$	Maximum and minimum eigenvalues of the (preconditioned) stiffness matrix.
$\boldsymbol{\xi}$	Local coordinate in the x −direction.
$\boldsymbol{\sigma}$	Modal coefficient decay rate in hp spectral estimators.
$\boldsymbol{\tau}$	Truncation error.
$\boldsymbol{\phi_i, \phi_j}$	Shape (basis) functions in the finite element methods weak formulation.
$\boldsymbol{\varphi_i, \varphi_j}$	Test/weight functions in weak-form equations.
$\boldsymbol{\psi_{i,j}}$	Scalar field variable in finite volume methods representation.
$\boldsymbol{\omega}$	Relaxation factor in iterative over-relaxation methods methods.
$\boldsymbol{\Omega}$	Computational domain.

Other Symbols

$\boldsymbol{\delta_{ij}}$	Kronecker delta.
$\boldsymbol{\nabla}$	Laplacian operator.
$\boldsymbol{\partial\Omega}$	Boundary of the domain.

List of Abbreviations

1D	One-dimensional.
2D	Two-dimensional.
AMG	Algebraic Multigrid.
AMR	Adaptive Mesh Refinement.
CG	Conjugate Gradient.
CIMs	Classical Iterative Methods.
DOF	Degree of Freedom.
EHD	Electrohydrodynamic.
FDM	Finite Difference Method.
FEM	Finite Element Method.
FEM1	Finite Element Order-1 Preconditioner.
FVM	Finite Volume Method.
GS	Gauss–Seidel.
Jacobi	Jacobi Diagonal Preconditioner.
KSMs	Krylov Subspace Methods.
MPCG	Multi-Preconditioned Conjugate Gradient.

PCG	Preconditioned Conjugate Gradient.
SOR	Successive Over-Relaxation.
PDEs	Partial Differential Equations.
SSOR	Symmetric Successive Over-Relaxation.
UP	Unpreconditioned System.
V-cycle	Multigrid V-shaped cycle.

Chapter 1: Introduction

Partial Differential Equations (PDEs) form the mathematical foundation for modeling a wide range of physical, engineering, and natural phenomena. They describe the behavior of systems with respect to both spatial and temporal variations. Applications of PDEs are extensive including heat conduction, fluid flow, electromagnetic, solid mechanics, and chemical diffusion. Due to their importance in modelling many physical systems, efficient and accurate solution methods for PDEs have been a major topic of study in computer science and engineering.

Among all types of PDEs, elliptic equations are very distinct. They often occur in steady-state problems like thermal equilibrium, electrostatics, and structural analysis in the static state. The most common and basic elliptic PDE is the Poisson equation, which appears in numerous fields either as a direct model or as the steady-state form of a diffusion process. Examples of such applications are shown in Figures 1.1 and 1.2, where the Poisson equation governs temperature fields in heat-transfer problems and pressure in incompressible fluid dynamics respectively. In addition to these areas, Poisson's equation also appears in electrohydrodynamic (EHD) spray modeling, where it is used to compute the electric potential field that governs droplet motion and Coulomb-driven breakup processes [1].

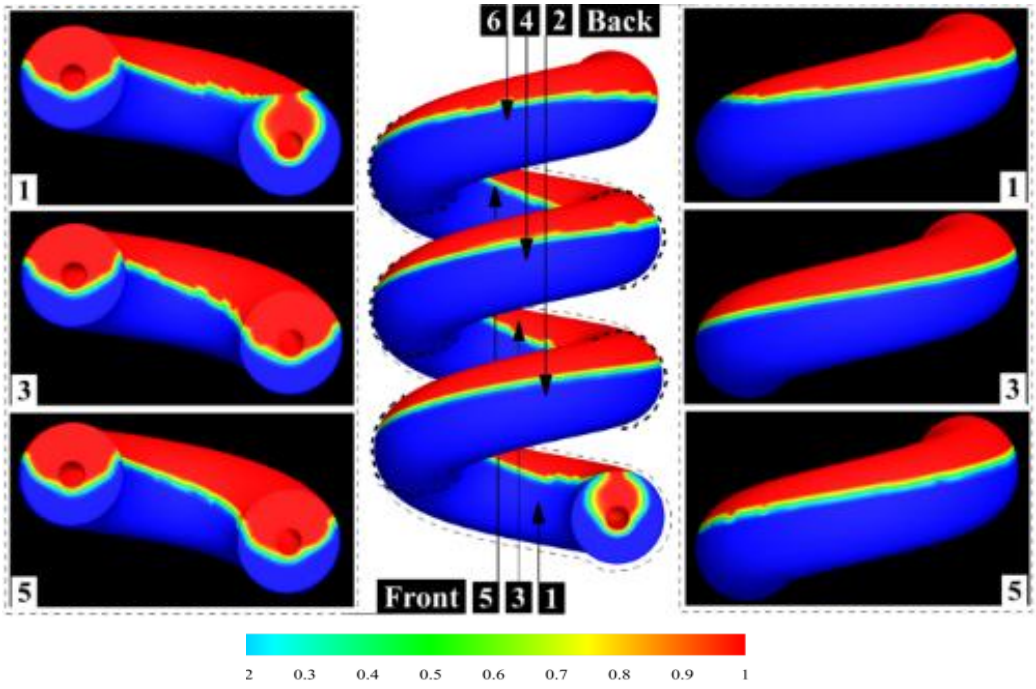


Figure 1.1: Liquid-fraction contours depicting the temperature-driven melting field in the helical PCM storage unit. The melt-fraction distribution reflects the underlying steady-state conduction solution of the Poisson equation that defines the temperature field within the system [2].

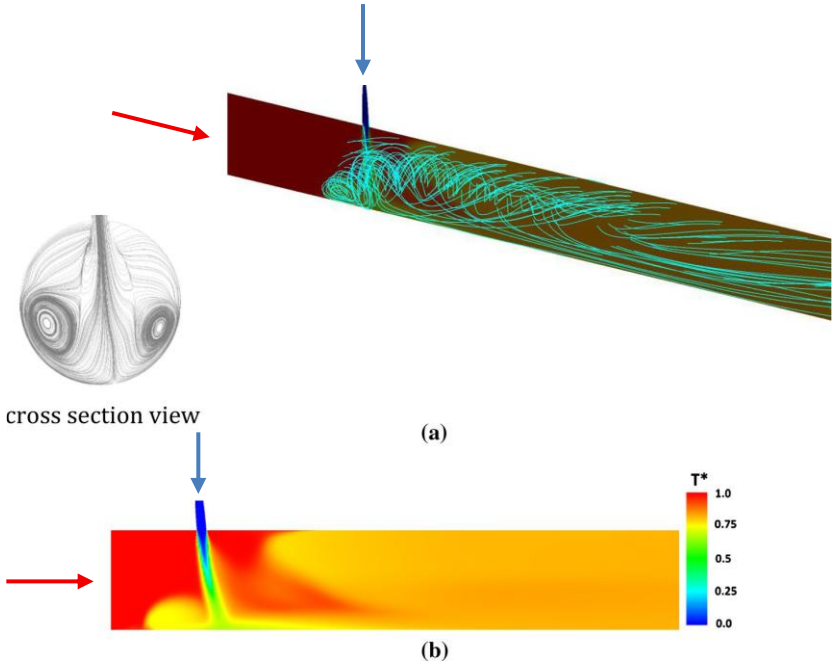


Figure 1.2: (a) Flow streamlines and cross-sectional vortex structures showing how the two inlet streams (indicated by the incoming-flow arrows) enter the junction and interact to form a three-dimensional mixing region. The counter-rotating vortices visible in the cross-sections illustrate the swirl generated as the flows merge. (b) The corresponding normalized temperature field, demonstrating how the mixing pattern redistributes thermal energy within the junction. Both the incompressible pressure field and the temperature distribution are obtained from Poisson-type equations, making the Poisson equation fundamental to resolving the coupled fluid–thermal behavior captured in this figure [3].

Accurate and efficient solutions to the Poisson equation are therefore essential in many engineering simulations. However, solving large elliptic systems remains computationally challenging, especially when high accuracy or highly refined meshes are required. This thesis addresses these challenges within the framework of hp −adaptive finite element methods (FEM) and highlights the critical role of preconditioning techniques in reducing computational complexity and improving solver performance.

1.1 The Need for Numerical Methods

Analytical solutions to partial differential equations are usually restricted to idealized cases with simple geometries and boundary conditions. Most problems in the real world are characterized by complexities in geometry, material behavior, and boundary conditions that render analytical solutions inapplicable or impracticable. Therefore, in such problems, numerical methods are increasingly becoming indispensable in approximating the solution.

Numerical methods are flexible in representing complicated domains, support different types of boundary conditions, and facilitate adaptive methods of reaching the desired accuracy. Different types of numerical methods may be employed based on the problem's type of PDE and the needs of the computations. Advanced methods, particularly with

adaptivity and optimization of the solvers, allow for realistic and efficient methods of handling large-scale problems.

1.2 A Brief Overview of Numerical Methods for PDEs

Several numerical methods have been developed for solving PDEs. The most prominent among them are:

1.2.1 Finite Difference Methods (FDM)

The Finite Difference Method (FDM) approximates derivatives by using differences between points on structured grids. For instance, the second derivative in one dimension can be approximated as

$$u_{xx}(x_i) \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} + O((\Delta x^2)) \tag{1.1}$$

where $u_i = u(x_i)$ is the function value at node i, Δx is the grid spacing, and $O((\Delta x^2))$ denotes the truncation error of order (Δx^2), showing that the scheme is second-order accurate. This central difference formula illustrates how derivatives at a given point are computed directly from their neighbors, and the same principle extends naturally to two dimensions, leading to the standard five-point stencil for the Laplacian. This technique is simple and easy to apply to box-shaped domains. However, its accuracy relies on the grid's resolution and the approximation's order, since truncation errors play a big role. FDM work well with regular grids but struggle on uneven domains or when handling complex materials and shapes [4][5][6]. They provide efficient performance and can achieve accurate results when applied to smooth problems on regular gridss [7][8]. Doing so often means using fine grids or larger stencils, that can raise computational costs and make things more complicated in areas that need mesh refinement [9].

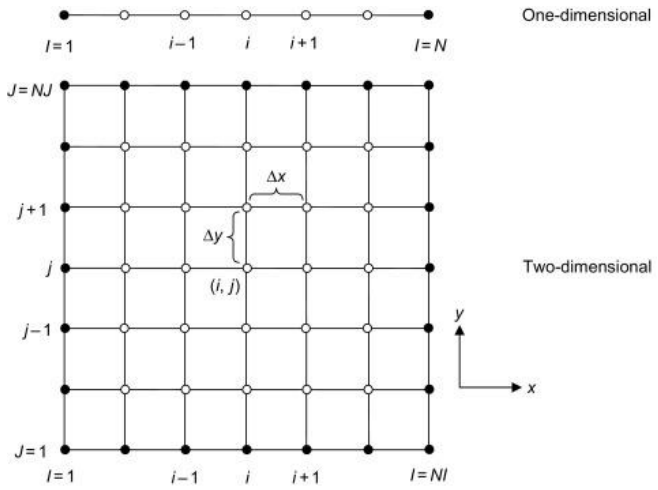


Figure 1.3: Illustration of one- and two-dimensional Cartesian grids with uniform spacing (Δx, Δy), n the finite difference method. Solid markers indicate boundary nodes, while hollow markers represent interior computational nodes, with the point (i, j) denoting a generic interior location [10].

1.2.2 Finite Volume Methods (FVM)

The finite volume methods (FVM) is regarded as a reliable numerical technique due to two key advantages: it ensures conservation of physical quantities such as mass and total energy at the integral level by balancing fluxes across control-volume faces, and it is particularly effective when dealing with complex boundary conditions [11]. One issue, however, is that when we try to increase the order of accuracy where higher order is needed to make simulations more efficient, we run into problems near curved boundaries [12]. The main issue is that the mesh does not perfectly match the actual shape of the domain. Since polygonal cells cannot follow curved edges exactly, this mismatch can seriously affect accuracy. Without carefully handling the boundaries, even a high-order method can end up performing like a simple second order one [12]. Despite various attempts to tackle this problem, it continues to pose significant challenges. In [13][14], the authors suggested using a simpler polygonal version instead of the physical domain and made changes to the normal vector to match the wall boundary condition.

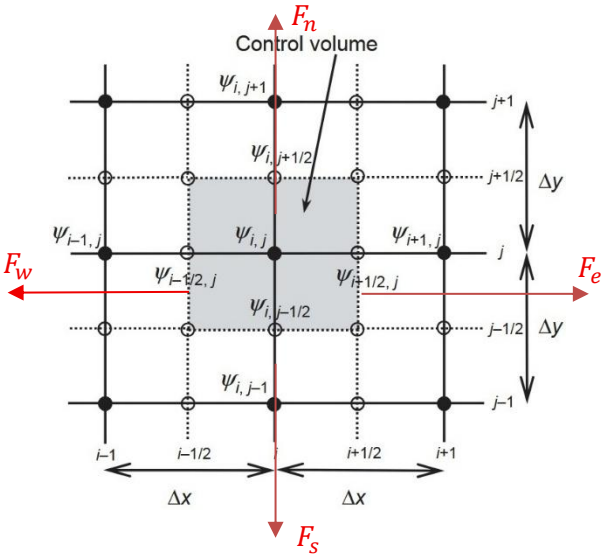


Figure 1.4: Two-dimensional Cartesian grid in the finite volume method, illustrating the control volume around node $\psi_{i,j}$. Filled symbols represent nodal points, open symbols denote face-centered values, and Δx and Δy indicate the grid spacing. The arrows F_w, F_e, F_s, and F_n represent the fluxes through the west, east, south, and north faces of the control volume, respectively. Conservation is enforced by balancing these fluxes across the control volume boundaries, so that the flux leaving one cell enters its neighbor.

1.2.3 Finite Element Method (FEM)

The finite element method (FEM) partitions the computational domain into many elements that are small in size and approximate the solution by local basis functions on those elements. Among the many finite element formulations, the most common are the Continuous Galerkin Finite Element Method (CGFEM) and the Discontinuous Galerkin Finite Element Method (DG or DCGFEM).

The Galerkin method, was introduced by the Russian engineer and mathematician Boris Galerkin [15] in 1915, in his work on the bending of elastic plates. His method was to project the governing differential equations onto a finite dimensional subspace with the method of weighted residuals and select the trial and the test functions from the same function space. This concept later became one of the standard finite element methods and is still the basis in the numerical solution of partial differential equations today. As an example, consider the Poisson problem

$$-\nabla^2 u = f \quad \text{in } \Omega, \qquad u = 0 \quad \text{on } \partial\Omega, \tag{1.2}$$

where u is the unknown solution, f is a prescribed source term, Ω is the computational domain, and $\partial\Omega$ is its boundary. The Galerkin formulation requires finding an approximate solution u_h such that

$$\int_{\Omega} \nabla u_h \cdot \nabla v_h \, d\Omega = \int_{\Omega} f \; v_h \, d\Omega \qquad \forall v_h \in V_h, \tag{1.3}$$

where V_h is the finite-dimensional space of test functions. After discretization with chosen basis functions, this weak form leads to the standard algebraic system

$$KU = F, \tag{1.4}$$

with the stiffness matrix and load vector respectively defined by

$$K_{ij} = \int_{\Omega} \nabla \varphi_i \cdot \nabla \varphi_j \, d\Omega, \qquad F_i = \int_{\Omega} f \; \varphi_i \, d\Omega, \tag{1.5}$$

where φ_i are the basis functions, U is the vector of unknown coefficients, and F is the discrete source vector. This short example illustrates how the Galerkin idea is applied in FEM. The detailed variational formulation, finite element discretization, weighted residual approach, and the construction of the stiffness matrix are presented in Sections 3.1–3.4.

While the Continuous Galerkin (CG) method enforces continuity at element interfaces, the Discontinuous Galerkin (DG) method, first introduced by Reed and Hill [16] in 1973 for neutron transport problems, allows for discontinuities between elements. Major advancements in DG methods were made by Cockburn and Shu [17] in the 1990s, especially in the context of hyperbolic conservation laws.

1.3 Advantages of the Galerkin Method for Elliptic Problem

While both CGFEM and DGFEM provide flexibility, their differences make them suitable for different types of problems. CGFEM provides efficiency and simplicity for smooth solutions. In the continuous Galerkin formulation, continuity is enforced across element boundaries. This leads to symmetric, sparse, linear systems and draws on a well-established mathematical framework for stability, convergence, and error analysis, which is especially mature for elliptic problems.

Key advantages of Continuous Galerkin FEM include:

- A strong and well-understood theory for dealing with elliptic problems.
- An effective way to apply boundary conditions.
- Computational efficiency for problems on structured mesh.

CGFEM is well-suited for elliptic PDEs like the Poisson equation, offering smooth and continuous solutions across the domain. Its structure makes it an effective and reliable choice for such problems.

In finite element methods, *adaptivity* refers to systematically refining the discretization in order to achieve higher accuracy with fewer degrees of freedom. The two classical approaches are $h-$refinement, where elements are subdivided into smaller ones, and $p-$refinement, where the polynomial order of the basis functions within each element is increased. A combination of these, known as $hp-$refinement, uses both strategies

simultaneously, allowing very efficient error reduction by adjusting both mesh size and polynomial degree [18]. These three refinement strategies are illustrated in Figure 1.3, which shows how elements are subdivided or enriched to produce more accurate approximations of the solution.

When solving PDEs with the finite element method, the discretization process leads to a large system of linear equations $Au = b$, where A is the global stiffness matrix is assembled from the local stiffness matrix of all elements. The stiffness matrix is symmetric positive definite (SPD), meaning it is symmetric ($A = A^T$) and all its eigenvalues are strictly positive. As explained in detail in Section 3.2, such systems often become ill-conditioned, meaning that small errors in the data or intermediate computations can be greatly amplified, which slows down the convergence of iterative solvers [19]. Mathematically, the degree of conditioning is measured by the condition number of A, defined as $\kappa(A) = \frac{\lambda_{max}}{\lambda_{min}}$ where λ_{max} and λ_{min} are the largest and smallest eigenvalues of A respectively [20]. If this ratio becomes very large, the system is said to be ill-conditioned. To address this difficulty, preconditioning (PCD) is introduced, and the system is transformed into an equivalent form with more favorable numerical properties. In practice, a preconditioner acts as an approximate inverse of A, reducing the effective condition number where A is premultiplied by the preconditioner clustering the eigenvalues, and thereby accelerating the convergence of iterative solvers [19].

Efficient preconditioning methods play a crucial role in achieving good performance in CGFEM, especially with *hp*-adaptive refinement strategies, since the resulting systems are ill-conditioned by nature.

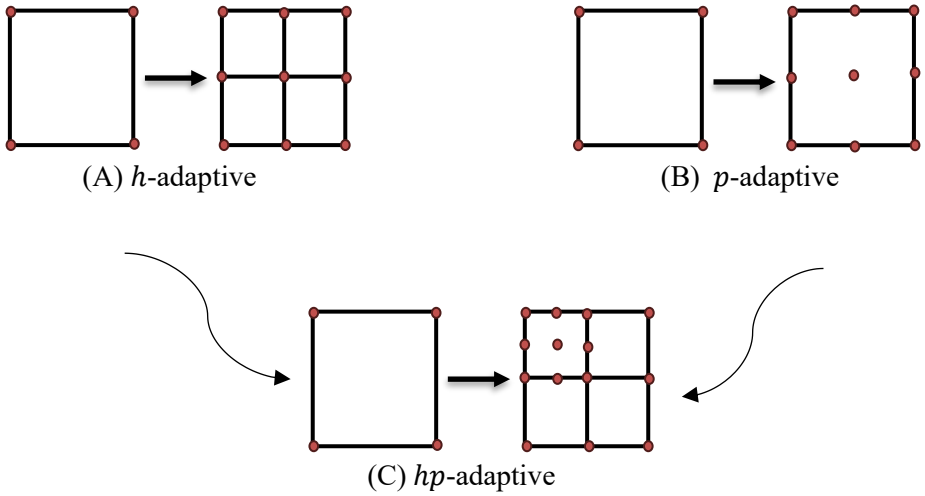


Figure 1.5: 2D *hp*-adaptive illustration. In (A) h-adaptive refinement, a single element is subdivided into four smaller child elements, reducing the element size. In (B) p-adaptive refinement, the polynomial order of the basis functions is increased, enriching the approximation space without changing the mesh. In (C) hp-adaptive refinement, both strategies are combined: the element is subdivided, and the polynomial order is raised simultaneously.

1.4 Characteristics of the Discontinuous Galerkin Method

The Discontinuous Galerkin Method differs from the continuous version by allowing solutions to be discontinuous between elements. It is similar to the finite volume (FV) method, where elements share information through fluxes computed at their boundaries [21]. First introduced by two scientists; Reed and Hill in 1973 [16] to solve the neutron transport equation.

This local element-by-element freedom renders DG highly flexible, especially in problems with discontinuities, steep gradients, or localized phenomena. Sample problems include shock waves, multi-phase flow tracking, and convection-dominated transport problems. Since each element is independent, the DG methods parallelize well and are ideal for adaptive refinement.

Key strengths of the Discontinuous Galerkin FEM include

- Local conservation: Each element acts as a self-contained control volume, making DG naturally conservative.
- Enhanced local adaptivity: Mesh refinement and polynomial enrichment can be applied element-wise without global remeshing.
- Superior handling of non-smooth solutions: Especially useful in hyperbolic and conservation dominated PDEs.
- Flexible treatment of complex geometries and non-conforming meshes.
- Suitability for high performance computing: Minimal communication between elements enables efficient parallelization.

DG methods are effective in addressing hyperbolic PDEs or problems with strong convection, where discontinuities are common. For elliptic problems with smooth solutions, like the Poisson equation, DG methods can introduce unnecessary computational overhead due to the extra degrees of freedom and the added complexity of flux formulation.

1.5 Adaptive Mesh Refinement (AMR)

In most real-world problems, the complexity of the solution is not evenly distributed across the domain, and as a result adaptive refinement is required. Areas near singularities or boundary layers often require a mesh with superior quality to achieve acceptable accuracy. Other regions of the domain are smooth enough that using a fine mesh there is computationally inefficient. Applying uniform refinement or a fixed polynomial degree throughout the entire domain often leads to high computational cost without a matching gain in accuracy. To address this challenge, adaptive finite element methods have been developed [22]. There are two main types of finite element methods: h-version and p-version. These methods improve solution quality in certain regions of the domain by increasing mesh refinement (h-refinement), raising the polynomial order (p-refinement), or implementing both strategies (hp-refinement) [23][18]. Among these approaches, the

hp-refinement FEM is especially effective [24], it has the flexibility to refine in local element size and polynomial order.

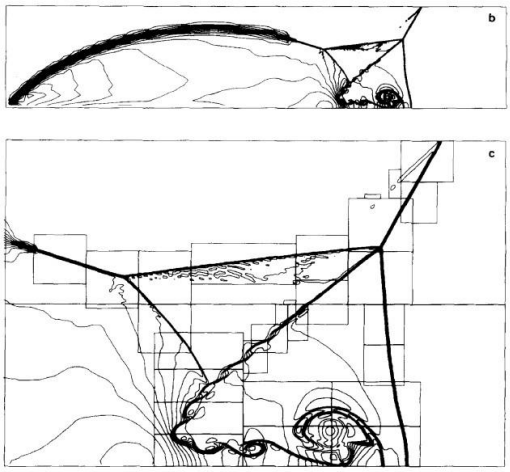


Figure 1.6: Shock reflection on an oblique wedge using an Adaptive Mesh Refinement technique [25].

One of the key issues of adaptivity techniques is how to choose where, when and how to refine. If the exact solution is known, one should calculate the actual error and control the refinement process. Yet in most examples or situations in the solution of PDEs the exact solution is not known or not tractable analytically. Therefore, there is no choice but to use the so-called error estimation technique.

Error estimation is a key component [26] of adaptive finite element methods; it guides mesh and polynomial refinement to efficiently improve solution accuracy. It helps decide which parts of the mesh need to be improved and whether h- or p-refinement is preferable. This choice is particularly significant in hp-adaptive strategies: in smooth areas that aren't accurate enough, we can increase the polynomial degree (p-refinement), and also in areas where the solution is not smooth, we can split the mesh into smaller parts (h-refinement) to better capture the details.

hp-adaptivity in FEM tries to make the solution more accurate without using too much computing power by improving the mesh only in the parts where the error is high [27]. These methods are important for solving hard problems where we do not know the exact answer. But making the solution more accurate can use a lot of computer power. When the mesh becomes more detailed and we use higher math (polynomials), the equations can become harder for the computer to solve correctly. This is due to several factors:

- The higher condition number of stiffness matrix for higher order basis functions
- The non-uniform element size, which introduces scale disparities.
- The complex coupling between elements of varying p-level and mesh densities.

1.6 Error Estimation in Adaptive Finite Element Methods

Error estimation in the finite element method (FEM) is categorized into two main classes: a prior error estimation and a posterior error estimation [28].

1.6.1 A Priori Error Estimates

A priori error estimates tell us how far the numerical answer might be from the exact answer, even before we solve the problem. These estimates are made using guesses about how smooth the exact answer is, the size of the mesh (h), the degree of the polynomials (p) and how well the finite element method can approximate the solution.

Typically, a priori error estimates take the form [29][30]:

$$|u - u_h|_{H^1(\Omega)} \leq Ch^p |u|_{H^{p+1}(\Omega)} \tag{1.6}$$

where u is the exact solution. The value p shows the degree of the polynomial used in the method and is related to how smooth the exact solution is. The constant C does not change with the mesh size h, but it can depend on the shape of the domain and the polynomial degree.

Some error estimates help us understand how the solution improves. They show how the error decreases when we refine the mesh (h) or use a higher-order polynomial (p); especially if the solution is smooth. However a priori estimates come with several limitations:

- They require knowledge (or assumptions) about the exact solution's regularity, which is often not available in practice.
- They provide global bounds rather than local error indicators and therefore cannot guide local refinement.
- The constants involved (e.g., C) are typically unknown and may vary significantly depending on the problem.

Because of these limits, a priori error estimates are mainly used to study the theoretical behavior of finite element methods.

1.6.2 A Posteriori Error Estimates.

A posteriori error estimates, unlike a priori ones, are calculated after solving the problem. According to the obtained results an a posteriori error estimate predicts how accurate the solution is [31]. Unlike a priori estimates, a posteriori estimate does not depend on knowing the exact solution. The main goal of a posteriori error estimation is to calculate, when possible, an estimate and ideally to bounds the solution error in a given norm or for a specific quantity of interest, based on the problem data and the obtained finite element solution [28][24]. Several common types of these estimators are widely used, such as residual-based estimators, hierarchical estimators, and dual weighed residual (DWR) estimators [26]. These estimators vary in complexity, robustness, and computational cost.

However, they all share the key property of being computable from the numerical solution and thus usable in adaptive strategies. There estimators are good choices in adaptive mesh refinement where local information needs efficient improvement in accuracy without refining the entire domain [28].

An effective error estimator should meet several key requirements [28]:

- It should provide an accurate estimate that is close to the true (unknown) error.
- The estimate must be asymptotically correct, meaning it decreases at the same rate as the real error when the mesh is refined.
- Ideally, it should give guaranteed and sharp upper and lower bounds for the actual error.
- The estimator should be computationally efficient, adding little extra cost compared to the total simulation time.
- It should be robust and work well across several types of problems, including nonlinear ones.
- Lastly, it should be suitable for guiding adaptive mesh refinement to improve the mesh based on the goal of the simulation.

In the context of hp-adaptivity finite element methods, posteriori error estimators play a central role [27][28]. Because hp-FEM deals with nonuniform meshes and high order basis functions, an estimator must not only respond to spatial variation but also spectral content. Although traditional estimators remain efficient, they might not be suited to the structure of high order approximation [32].

In this work, we adopt a coefficient-based a posteriori estimator, which leverages the decay of Legendre polynomial coefficients to measure local solution smoothness and accuracy [33]. This method works well for high-order hp-refinement because it takes full advantage of the polynomial structure of the approximation space. The estimator is specifically designed to work efficiently with high-order modes, fitting naturally with the hp-FEM refinement process. It ensures that computational resources are focused on under-resolved areas, leading to accurate and efficient results.

1.7 Iterative Methods

The finite element method typically results in large linear systems and solving them is often the most computationally intensive step. While direct solvers are accurate and robust, they become impractical for large-scale problems due to their high memory and computational costs. Consequently, iterative methods are generally preferred, especially for large simulations [22].

The first ideas for solving linear systems using iterative methods were introduced in the 1800s by Gauss, Jacobi, Seidel, and Nekrasov. Some important progress was made in the early 1900s. However, the real study of these methods for large systems started only after

digital computers were invented, around the end of World War II. So, everything before the late 1940s is considered the early history of this topic [34].

In FEM, discretization of partial differential equations typically leads to a system of linear equations:

$$Ax = b \tag{1.7}$$

where A is the system matrix arising from the discretization of the governing PDE, x is the vector of unknown degrees of freedom (e.g., nodal values of the solution), and b is the corresponding load or right-hand-side vector.

Solving this type of linear system quickly is a challenge, especially when using high-order finite element methods that adapt to the problem. As the mesh gets finer and we use more complex functions (in hp-adaptivity), the system becomes harder to solve because the matrix becomes ill-conditioned. Two well-known solvers that are affected by this are the Conjugate Gradient (CG) method and the GMRES method. In iterative methods for solving large linear systems, especially in CG the convergence properties of the CG algorithm depend highly on the condition number of A and can be influenced by reducing the condition number by proper measures [35]. We will study this problem in detail later, but it is important to say now that these issues directly affect how well and how fast solvers work.

For big or dynamically changing problems, direct solvers are not useful because they take too much memory and take a long time to run. They are more scalable and use fewer resources. To understand the methods in this work, we first look at two main types of solvers: classical (stationary) methods and Krylov subspace methods. The next parts explain how they work, what they do well, and where they have problems when solving FEM systems.

1.7.1 Classical Iterative Methods (CIMs)

Classical Iterative Methods (CIMs) are the earliest techniques for solving linear equations. They are usually stationary, meaning they use a fixed formula repeatedly to improve the solution from one step to the next.

$$x^{k+1} = Tx^k + c, k = 0,1, \ldots, \tag{1.8}$$

Here, T is a fixed matrix in every step, c is a fixed vector, and x^0 is the starting guess for the solution vector x. The goal is to keep reducing the error until the solution converges [19].

Some of the most practical, and also well-known classical methods include [22]:

- Jacobi method: All variables are updated simultaneously. Each new value is computed using only the values from the previous iteration, and all updates are applied together at the end of the iteration.

- Gauss-Seidel method: Variables are updated sequentially. As soon as a new value is computed, it is immediately reused in the subsequent calculations within the same iteration.
- Successive Over-Relaxation (SOR): This method builds on Gauss–Seidel by adding a relaxation factor that controls how large each update step is. By taking slightly larger steps toward the solution, SOR can reach convergence more quickly when the factor is chosen appropriately.
- Symmetric Successive Over-Relaxation (SSOR): A symmetric version of SOR often used for symmetric matrices.

These methods are simple to implement, require low memory, and are computationally cheap per iteration. However, they also have significant limitations:

- They often converge slowly, especially for large scale or even ill-conditioned systems.
- For sparse systems arising from finite element methods, the solver might take many steps to solve, or it might not work at all.
- While classical methods still play a role, particularly as preconditioners in more advanced algorithms, they are not suitable as stand-alone solvers for large, complex problems.
- The limitations of classical iterative methods motivated the development of more powerful techniques capable of handling large, sparse, and ill-conditioned systems more efficiently.

1.7.2 Krylov Subspace Methods (KSMs)

The history of Krylov subspace methods can be explained briefly as follows (for more details, see [36][37]). This method is used to solve linear systems when matrix A is symmetric and positive definite. In the past, Krylov space solvers had other names like semi-iterative methods or polynomial acceleration methods with flexible preconditioning. We can use one Krylov solver to make another Krylov solver work better [38].

Krylov subspace methods (KSMs), like Conjugate Gradient (CG) and GMRES, work differently. They do not use a fixed formula. Instead, they create a set of search directions by applying the matrix to the residuals [38]. Each step tries to make the error smaller within a space called the Krylov subspace.

The main advantages of Krylov methods include:

- Fast convergence for large sparse systems compared to classical methods.
- Better scalability with problem size.
- Less sensitivity to certain matrix properties, although preconditioning is often still needed for optimal performance.

- Methods like GMRES can manage non-symmetric and non-positive definite matrices, in other words we can have flexibility.

In today's finite element simulations, especially with hp-adaptive methods, we generate very large and complex matrices. Krylov subspace methods are a good and reliable way to solve these matrix equations. They constitute the main part of advanced solvers that help us solve large science and engineering problems more easily.

1.7.2.1 Conjugate Gradient (CG) Method and its Properties

In 1952, Lanczos [39] and also Hestenes and Stiefel [40] came up with the Conjugate Gradient (CG) method to solve large systems of equations. The CG method is a popular choice for solving large systems of linear equations, especially when the matrix is symmetric, positive definite, and has mostly zero entries (sparse) [41][42]. These conditions are common in FEM problems, particularly with elliptic equations like the Poisson equation. Because CG is specifically designed for such matrices, it is highly effective for solving linear systems that arise in FEM [35].

Key Features of the Conjugate Gradient Method [35]:

- Designed for symmetrical and positive definite matrices: The method works best when the system matrix is both symmetrical and positive definite.
- Efficient with sparse matrices: It fully takes advantage of sparsity, avoiding the need to store the entire matrix.
- Iterative method: It approaches the solution step by step, which is useful for large problems.
- Based on the operator principle: The method represents the system using operator notation, which simplifies storage and computation.
- Low memory usage: Since it does not need to store all entries of the matrix, it saves memory, which is an important advantage, especially in the earlier days of computing.
- No need to choose tuning parameters: Unlike other methods, CG does not require manual parameter selection to speed up convergence.
- Well-suited for large systems: Particularly effective for solving large systems from finite element discretization of elliptic and biharmonic problems.

A key factor in how well CG works is its connection to the eigenvalues of the system matrix. CG performs better when the eigenvalues are tightly grouped and the matrix has a low condition number, leading to faster convergence. But if the eigenvalues are spread out, convergence can be affected [41].

The performance of iterative methods strongly depends on the condition number of the linear system. A high condition number (ill-conditioning) slows convergence and may

prevent the method from reaching a solution within a reasonable time For example, in second-order elliptic PDEs solved by FEM, the condition number grows like $0(h^{-2})$, where h is the size of the element, and the required CG iterations scale as $0(h^{-1})$, making preconditioning crucial as the mesh is refined [19]. The Poisson equation, in particular, is known to be inherently ill-conditioned [43]. In high-order and adaptive FEM, using higher polynomial degrees makes the matrices denser and more ill-conditioned [44].

To fix this, we use preconditioners. Preconditioning changes the system to make it easier to solve, usually by improving the spread of eigenvalues and lowering the condition number. We will explain eigenvalues, and preconditioning in detail later, but for now, just know that these ideas are important for building fast and reliable solvers [19][45].

1.8 Preconditioning

The word "preconditioning" was first used in 1948 by Turing [46]. The first time "preconditioning" was used with iterative methods was in a 1968 paper by Evans, where he talked about using Chebyshev acceleration with the SSOR method [47]. Preconditioning is a technique that changes the system into a new form with better properties, making iterative methods faster and more stable [48], thus easier and faster to solve [49]. As far back as 1845 Jacobi used a trick called plane rotations to make equations easier to solve [50]. This made his method, now called Jacobi's method, work better and find the answer more easily. The idea of using preconditioning with the conjugate gradient method started in the 1960s [36]. Some early papers mentioned it, and in 1972, Axelsson [51] studied how SSOR preconditioning could help the CG method work faster for certain problems.

A good preconditioner p should follow these simple rules:

- The new system, after using the preconditioner, should be simple and quick to solve.
- The preconditioner should be quick and easy to make and use.
- The eigenvalues of the preconditioned matrix should be clustered and bounded away from zero, which improves the conditioning of the system and accelerates convergence.

This means a good preconditioner should help the method finish quickly (fast convergence), but it shouldn't be too costly to use in each step. These two goals can conflict, so we need to find a good balance. If the preconditioner is well-designed, the total time to solve the system should be much shorter than solving it without one [19][52].

In general, there are two approaches to building a preconditioner [19]:
A) **Application-Specific Preconditioning**: One way to build a preconditioner is to design it for a specific type of problem, especially when working with partial differential equations (PDEs). This method uses full information about the problem, like the main equations, boundary conditions, discretization method, geometry, and physical laws.

By using this full information, the preconditioner can be made highly efficient for that particular case.

- ✓ Main features:
  - Works best for one specific kind of problem.
  - Needs full understanding of the problem set up.
  - Aims to be nearly optimal in performance.

- ❖ Examples:
  - Using a simpler, related PDE that is easier to solve.
  - Lower order discretization.
  - Multigrid preconditioners based on similar equations.

B) **General-Purpose (Algebraic) Preconditioning**: This method makes preconditioners that can work for many kinds of problems and does not need full details about the original equation. It only relies on the system matrix (A).

- ✓ Main features:
  - Works for a wide range of problems.
  - Needs only the matrix, not full problem details.
  - Easier to develop and use.
  - Less sensitive to problem changes.
  - Usually not optimal but performs well in general.

- ❖ Examples:
  - Incomplete factorizations (e.g., ILU).
  - Sparse approximate inverse methods.
  - Algebraic MultiGrid (AMG)

1.8.1 Single Preconditioning

The ideal situation would be to transform the original system $Au = b$ into a system where the matrix is the identity matrix. In such a case, solving the system would be immediate. This can be achieved by applying the exact inverse A^{-1} to both sides of the equation. Calculating the inverse of a large, sparse matrix is usually impractical and too expensive, making it unsuitable for large problems. In this study, we focused on solving everything numerically using methods that are not direct but rather iterative. Thus, preconditioning focuses on finding an approximate inverse. A preconditioner M is introduced to modify the original system:

$$(M^{-1}A)u = M^{-1}b \tag{1.9}$$

where M is designed to approximate A in a way that improves the conditioning of the system. The better the preconditioner captures the key properties of A^{-1}, the more efficiently the iterative solver can perform.

If M were exactly equal to A, the system would reduce to an identity matrix after preconditioning, and convergence would be achieved in a single step. However, creating a perfect preconditioner is usually not possible. So, practical preconditioners try to balance being easy to use and effectively improving the matrix's spectral properties.

A wide range of preconditioning techniques exists. Common examples include:

- Jacobi (diagonal scaling): A simple and cheap preconditioner that scales the system by the inverse of diagonal entries.
- Successive Over Relaxation (SOR) and Symmetric SOR (SSOR): Based on Gauss-seidel iterations, often used for structured problems.
- Incomplete LU (ILU) factorization: An approximate factorization that retains the sparsity pattern and approximates the full LU decomposition of the matrix A.

Sometimes, using several simple preconditioners together can create a more powerful combined preconditioner. This approach better captures the system's key features and speeds up convergence.

Using just one preconditioner can boost solver performance, but in complex hp-adaptive finite element problems, one may not be enough. Usually, a single preconditioner only captures some aspects of the system matrix inverse A^{-1}, not everything. Different preconditioners focus on different things, some reduce high-frequency errors, some fix scaling issues, and others improve the matrix's sparsity. Combining them lets us take advantage of their strengths together.

1.8.2 Multiple Preconditioned Conjugate Gradient (MPCG)

In complex finite element simulations, especially with hp-adaptive methods, system matrices can behave very differently across the domain. A single preconditioner may not capture all the key features needed for fast convergence. This is why using multiple preconditioners at the same time led to the development of the Multi-Preconditioned Conjugate Gradient (MPCG) method [53].

The basic idea behind MPCG is to use several preconditioners together in order to better approximate the action of the true inverse matrix A^{-1}. Each preconditioner can focus on a different part of the problem, like handling certain regions of the mesh, targeting specific error frequencies, or fixing scaling issues. By combining these strengths, MPCG can achieve faster convergence than using just one preconditioner.

Combining multiple preconditioners is not easy. They must be chosen carefully to work well together and not interfere. If they overlap too much or miss key errors, they can slow

things down instead of speeding them up. That is why picking the right set of preconditioners is crucial for making MPCG work well [54][55].

Within the MPCG framework, two variants are distinguished:

- Full MPCG: In this approach, all preconditioners are applied simultaneously in each iteration, and the search directions are updated based on the computational and memory resources per iteration.
- Non-Full MPCG (Partial MPCG): In the non-full approach, only a subset of preconditioners is used at each iteration, or their contributions are selectively incorporated. This reduces the computational cost per iteration but may lead to slower convergence compared to the full version.

In MPCG, choosing between full and non-full versions is about balancing speed and cost. Full MPCG converges faster but is more expensive per iteration, while non-full MPCG aims to balance the time spent on each iteration with the total number of iterations. The key to building fast solvers for large, high-order adaptive FEM problems is designing good MPCG strategies and selecting the right preconditioners.

1.8.3 Multigrid Preconditioning

Algebraic Multigrid (AMG) is another popular method for large systems, known for its efficiency and scalability. Here, we also use AMG as a preconditioner to evaluate its effect on solver performance. AMG has been developed continuously since the 1980s and became popular in CFD, it is now used in many commercial CFD programs as a fast solver [56]. This method works by creating a multilevel structure of matrices derived from the original system. It solves the linear system across these levels, where coarse levels smooth out low-frequency errors and fine levels handle high-frequency ones. One of AMG's main strengths is that it performs well without needing any geometric information, making it a flexible black-box preconditioner for various iterative solvers [22].

Basic iterative methods such as Gauss–Seidel, Jacobi, and block Jacobi are inexpensive and effective at eliminating high-frequency components of the error but struggle with low-frequency errors. These methods are referred to as smoothers because they primarily reduce the oscillatory (high-energy) modes while leaving behind smooth, low-frequency components. A common strategy to address this limitation is to compute corrections on a coarser grid that captures the low-frequency content.

This coarse-level projection is analogous to the finite element method's approach of approximating continuous solutions in finite-dimensional subspaces. By combining smoothing with coarse-grid corrections, multigrid methods systematically reduce errors across the entire spectral range [57] Given this multilevel capability, multigrid methods are not only effective as independent solvers but are also highly suitable for use as preconditioners in Krylov subspace methods such as CG.

Multigrid as a preconditioner [23], improves the spectral properties of the system matrix, leading to faster convergence and better scalability. This makes it particularly useful for large-scale finite element problems, where combining multigrid preconditioning with CG offers both robustness and computational efficiency.

A typical Multigrid V-cycle follows a structured sequence of operations, as outlined below and illustrated schematically in Figure 1.3:

- **Pre-smoothing:** Apply a few iterations of a basic iterative method (e.g., Gauss–Seidel or Jacobi) to reduce high-frequency errors on the fine grid.

- **Residual computation:** Calculate the residual $r = b - Au_0$, where A is the system matrix and u_0 is the current approximation.

- **Restriction:** Transfer the residual to a coarser grid using a restriction operator (e.g., full weighting or injection).

- **Coarse grid correction:** Solve the error equation on the coarse grid, either exactly or approximately.

- **Prolongation:** Interpolate the coarse-grid correction back to the fine grid and update the solution.

- **Post-smoothing:** Apply a few more iterations of the smoother to eliminate any high-frequency errors introduced during interpolation.

There are two primary multigrid strategies [58]: h-multigrid, which relies on a hierarchy of meshes with varying element sizes, and p-multigrid, which keeps the mesh fixed but varies the polynomial degree across levels. Each targets error reduction across scales using different discretization refinements.

It is worth noting that the effectiveness of multigrid methods heavily depends on the choice of smoother [29]; therefore, selecting an appropriate smoothing strategy is crucial for achieving optimal performance. Commonly used smoothers include Jacobi, Gauss–Seidel, symmetric SOR (SSOR), and Chebyshev iterations. Figure 1.4 demonstrates how p-multigrid coarsening is applied to a regular quadrilateral element by reducing its polynomial degree from second to first order. Figure 1.4 illustration shows the coarsening step in p-multigrid for a quadrilateral element, transitioning from a second order to a first-order representation [59].

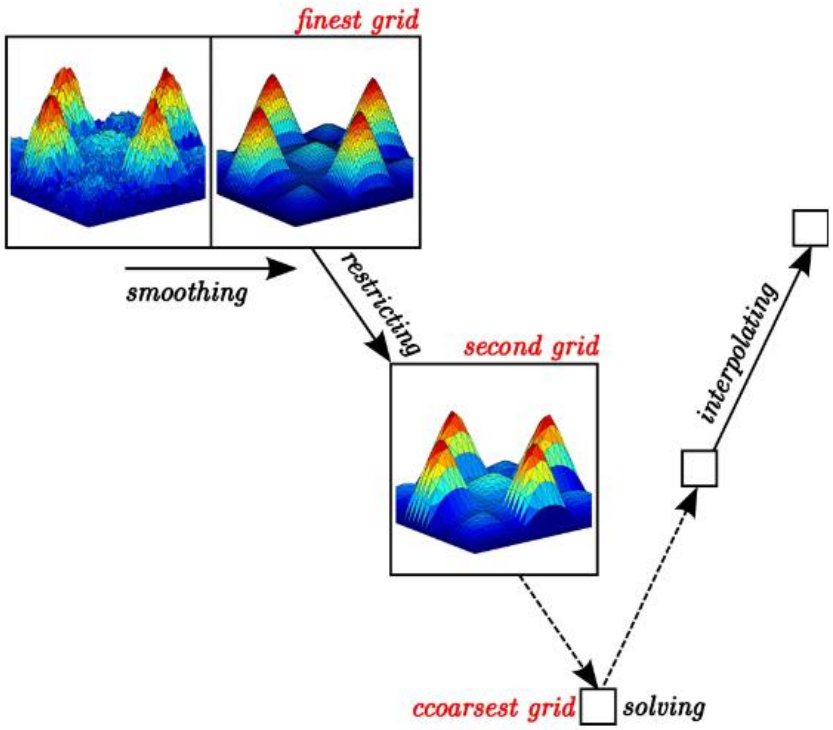


Figure 1.7: Steps of an AMG V-cycle: smoothing → restriction → coarse-grid solve → interpolation [60].

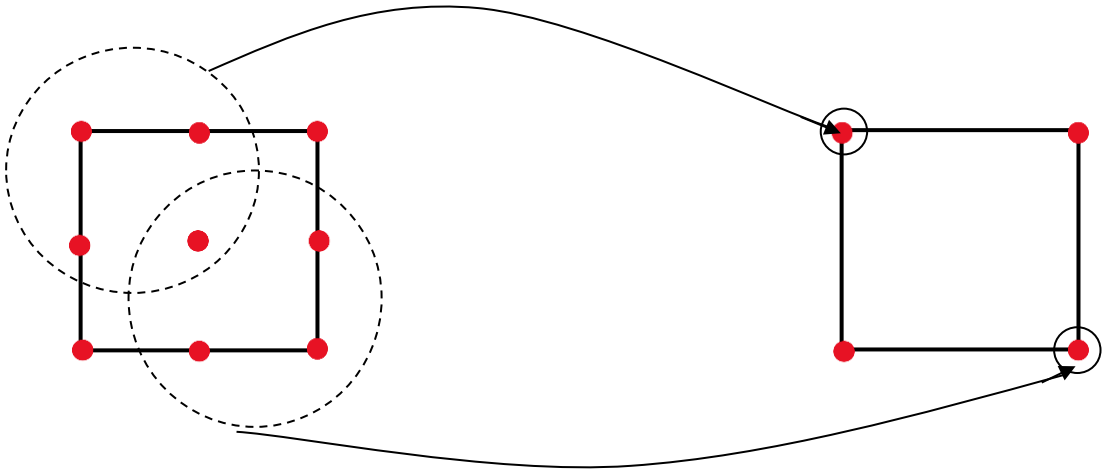


Figure 1.8: p −Multigrid coarsening step: Reducing a second-order quadrilateral element to first-order.

Chapter 2: Literature Review

The numerical solution of Poisson equations has been a central topic in numerical analysis for many decades. In particular, the past five decades have seen extensive research on the development of advanced methods for their efficient and accurate solution [61]. Yet, the Poisson equation remains inherently ill-conditioned, mainly due to the unbounded nature of the Laplace operator. The eigenvalues of this operator are real and countable, ranging from nonpositive values down to negative infinity. When the equation is discretized using numerical methods, the resulting matrix approximating the Laplacian retains similar spectral characteristics. Its eigenvalues range from values close to zero to large negative magnitudes, leading to a system with a high condition number. Moreover, this condition number grows even larger as the computational grid is refined, making the system increasingly difficult to solve efficiently [43].

A wide range of numerical techniques exist for solving elliptic partial differential equations (PDEs). A strong numerical method should ideally provide high accuracy, be simple to use, and work well with complicated shapes but in reality through most methods fulfill one or two of these goals [61], researcher often use approaches like finite element method, finite difference method, finite volume methods, boundary element methods, and interface methods. The choice of method depends on the specific application [55][62][63].

Spectral methods are favored for high accuracy problems as they achieve exponential convergence for smooth solutions [64]. However, they have some limitations, although they work well in some situations, they struggle with handling complex shapes mesh generation can get quite complicated in higher-dimensional problems [61][8].

The Boundary Element Method (BEM) provides an advantageous way of solving problems as it requires discretization of the boundary that satisfies far-field conditions [65]. For example, in complex Visco-thermal scenarios, BEM eliminates the need for the extensive meshing and artificial boundaries that (FEM) requires [66][67], which results in reducing the problem's dimensionality [65]. Although it is evident that FEM may demand significant computational effort, it provides access to internal field variables. In contrast, several adaptations of BEM have been developed to approximate boundary losses, either by simplifying or omitting boundary viscosity or by introducing boundary layer impedance [68][69][70]. They also remain limited in their ability to handle overlapping boundary layers or complex geometries [71][72].

In FEM, shape functions are used to build the approximate solution. Some of these functions are active only inside the element (bubble functions), while others act on the edges (edge functions) and this design ensures that the overall solution remains continuous across the elements [73]. One of the primary motivations for employing the finite element method (FEM), especially in high-order formulations, lies in its superior approximation capabilities and faster convergence. As highlighted in [73] high-order FEM significantly

reduces the size of the discrete problem compared to low-order methods while maintaining high accuracy, making it a preferred choice in computational mechanics. High-order FEMs offer faster convergence and can solve partial differential equations more efficiently in terms of accuracy versus computational cost [74]. For example, high-order FEMs have recently gained prominence in Lagrangian hydrodynamics on curved grids, where they enable accurate resolution of complex physical phenomena while also delivering efficient performance on modern computing architectures [75]. However, they can lead to ill-conditioned system matrices if not carefully designed and this issue often arises due to the nature of the shape functions used in high-order formulation [73]. Similarly, Adaptive Mesh Refinement (AMR) commonly used to enhance efficiency by concentrating resolution in critical regions can also produce ill-conditioned system matrices due to local mesh irregularities and stiffness imbalances. Therefore, recent research has turned to advanced preconditioning techniques to improve conditioning in high-order FEM.

Ainsworth et al. [76] introduces a local Additive Schwarz preconditioner for the p-version mass matrix on high-order triangular meshes achieving p- and h-independent conditioning with uniform PCG (Preconditioned Conjugate Gradient) iterations in time-stepping tests yet its runtime performance is unassessed. J.P. Whiteley. [77] presents a two-stage preconditioner for Newton-linearized incompressible large-deformation elasticity achieving nearly size-independent GMRES iterations via multigrid V-cycles, yet its actual runtime performance on large-scale problems is not reported. Phoon et al. [78] proposes a generalized diagonal preconditioner for the finite-element solution of Biot's consolidation equations and demonstrates that choosing an optimal scalar weight can reduce iteration counts by over 50%, but its practical applicability is constrained by its dependence on this weight and the limited scope of numerical tests.

Dutt et al. [79] show a separation-of-variables preconditioner for spectral-element discretization of elliptic and parabolic problems that yields uniformly bounded condition numbers by diagonally inverting tensor-product quadrilateral blocks, but it is restricted to (curvilinear) quadrilateral meshes and becomes computationally expensive at very high polynomial orders.Andrej et al. [80] note that Low-Order Refined (LOR) preconditioning has emerged as a powerful approach for addressing the ill-conditioning challenges in high-order finite element methods. Originally proposed by Orszag in 1980 [81] who explored how classic finite element scheme could act as preconditioner for spectral methods, this technique constructs an auxiliary low-order finite element discretization on a refined mesh that is spectrally equivalent to the high-order system. The fundamental principle behind LOR preconditioning is commonly referred to as the finite element method-spectral element method (FEM-SEM) equivalence which was later formalized by Pazner [82][58]. He proposed a matrix-free LOR preconditioner for high-order continuous and discontinuous Galerkin methods, achieving convergence rates independent of mesh size, polynomial degree, and DG penalty via a multigrid V-cycle with ILU (0) smoothing. However, this approach requires specialized smoothers for the highly anisotropic refined meshes and does not support nonconforming hp-refinement. Pazner & Kolev [83]

developed a subspace-correction LOR preconditioner that combines a conforming low-order-refined component with block-Jacobi-smoothed nonconforming edge spaces to yield conditioning independent of mesh size, polynomial degree, and irregularity, albeit with increased algorithmic complexity. Numerical results by Fischer [84] show that the condition number of the preconditioned system $A^{-1}{}_f A_s$, where A_s is the stiffness matrix is from the high-order discretization and A_f is derived from a low-order finite element mesh is bounded and this bound becomes particularly small when a lumped (approximate) mass matrix is used in constructing A_f.

Deville & Mund [85] introduced a finite-element preconditioner for Gauss–Lobatto pseudospectral discretization of elliptic problems that achieves fast convergence by tightly clustering its eigenvalues, but the long-range couplings produce very large, memory-intensive matrices which requires specialized transformation routines, which severely limiting its scalability. Kalchev &Vassilevski [86] propose a mortar-based preconditioner that condenses interior unknowns to interface variables for mesh-independent convergence in elliptic FEM problems, but it adds extra interface variables and makes assembly and solver setup more complex. Langer et al. [87] propose a two-level balancing domain decomposition by constraints (BDDC) preconditioner that maintains constant PCG iteration counts under adaptive mesh refinement and as subdomains increase demonstrating strong scalability but building its coarse space via augmented interface solves that incurs extra setup overhead. Pazner et al. [88] show that iteration counts are generally higher for unstructured meshes than for structured grids, with most preconditioners showing only mild growth in iterations as the polynomial degree increases. The Jacobi preconditioner with Gauss–Legendre basis performs best in terms of iteration counts. Kong et al. [89] introduce a low-order finite-element preconditioner for Gauss–Lobatto pseudospectral discretization of elliptic problems, proving that it tightly clusters eigenvalues and yields GMRES iteration counts independent of mesh size and polynomial degree. A particularly valuable finding is that the preconditioner maintains spectral bounds even for small penalty parameters. However, theoretical analysis assumes uniform affine meshes, and its effectiveness on irregular or curved grids is only supported by numerical experiments.

Zhou [90] addresses the dense linear system from non-adaptive Gaussian-process spatial regression and benchmarks including CG (Conjugate Gradient) preconditioners Cholesky variants, Jacobi, least-squares, and specialized sparse solvers finding that the simple diagonal preconditioner consistently speeds convergence by 10–20 % and remains effective even for systems with hundreds of thousands of unknowns. Kim et al. [91] propose a Legendre–Gauss–Lobatto finite difference preconditioner for CG-based, non-adaptive spectral-element discretizations of elliptic problems, showing it bounds the condition number independently of mesh size h and polynomial degree p. They further report that its effectiveness improves as the element count or polynomial degree grows, keeping iteration counts low even for very large problems.

Collier et al. [92] evaluate CG preconditioners for the homogeneous Poisson equation on fixed isogeometric meshes, finding that simple Jacobi and SSOR incur almost no setup cost but suffer rapidly rising iteration counts and thus solve time increases as the mesh is refined or approximation order increases. In contrast, the Incomplete Lower Upper (ILU) matrix decomposition preconditioner provides low iteration counts and fast runtimes, although it lacks a formal convergence guarantee. Salimi and Salkuyeh [93] introduce PSSOR, a two-parameter extension of the classical SSOR method that incorporates both a relaxation factor and a preconditioning parameter to solve complex symmetric linear systems. Their results show that PSSOR reduces iteration count and CPU time by up to 77% and 71%, respectively, compared to SSOR. Liu et al. [94] propose dynamic variants of SSOR, (accelerated over-relaxation) AOR, and (symmetric accelerated over-relaxation) SAOR and their dynamic optimal variants (DOSSOR, DOAOR, DOSAOR), where relaxation parameters are adaptively optimized at each iteration using a projection-based merit function an approach that enhances convergence but incurs additional computational costs due to parameter tuning. These methods are applied to linear systems arising from finite-difference discretization of 2D Poisson equations, both in matrix–vector and Lyapunov forms ($AX + XA^{\top} = Q$, where A and Q are known and X is an unknown matrix). DOSSOR achieves nearly twice the convergence speed of classical SSOR in terms of iteration count only, with significantly improved accuracy, owing to the dynamic parameter tuning guided by a merit function.

Traditional single-level approximate inverse methods often become inefficient for large-scale problems, as they require an increasing number of iterations to reach a desired level of accuracy. To address this limitation, multilevel preconditioners such as Geometric Multigrid (GMG) and Algebraic Multigrid (AMG) have been developed as more scalable alternatives. By combining local smoothing with coarse-grid corrections, multigrid methods maintain convergence rates that remain nearly independent of the problem size. In particular, AMG offers a significant advantage in large-scale applications by ensuring that the computational cost grows only linearly with the size of the grid [95] In [84], Heys et al, noted that using Jacobi as a smoother in multigrid methods for high-order finite element discretization results in a convergence factor that depends on the polynomial degree p, $q = 1 - \frac{c}{p}$, where c is a constant independent of p. Although employing Chebyshev acceleration improves the convergence to $q = 1 - \frac{c}{\sqrt{P}}$, the rate still remains dependent on p, indicating that neither approach achieves p-independent convergence. Additionally, the use of Gauss–Seidel as a smoother in higher-dimensional problems leads to even stronger p- dependence, further limiting its effectiveness for high-order discretization.

Wang and Chen [96] examine AMG as both a solver and a preconditioner for CG in elliptic PDEs on non-adaptive grids. Under p-refinement (increasing polynomial degree with fixed mesh), they show that PCG–AMG significantly lowers iteration counts, convergence rates, and total work. Its performance remains stable as system size grows,

demonstrating excellent scalability for large symmetric problems. Di Pietro et al. [97] evaluate multigrid strategies including: h-only (mesh refinement), $p-h$ (polynomial refinement over fixed mesh), and $hp-h$ (hp then h) for solving elliptic problems using the Hybrid High-Order (HHO) method. Their results show that $p-h$ (p then h) achieves the best balance of fast convergence and low setup cost for single right-hand side systems. Using these strategies as preconditioners also reduces computational cost by 25%.

Kim and Hwang [98] evaluated two AMG-preconditioned CG solvers (AMGPCG0 and AMGPCG1) for large unstructured 3D problems with non-adaptive, non-anisotropic meshes. Both solvers converged in fewer than 20 iterations, and AMGPCG1 achieved faster setup and reduced memory usage by employing single-precision variables. In one ill-conditioned test case, AMGPCG was over 20 times faster than SSOR as a preconditioned for CG, highlighting SSOR's poor efficiency under such conditions. MacLachlan and Olson [44] theoretically showed that the AMG method may not be optimal for high-order problems, because the system matrix becomes denser and more ill-conditioned by increasing the order of polynomial. However, in our work we still observed perfect performance with AMG. Moreover, Notay et al. [99] found that AMG V-cycles become less effective for high-order problems, suggesting W-cycles for better robustness. In contrast, our results show that V-cycles maintain strong performance even as the polynomial degree increases, highlighting their efficiency without extra computational cost. Additionally, Grauschopf et al. [100] showed that AMG offers robust and efficient convergence for second-order elliptic-problems, with relatively low setup costs and strong scalability. Although full theoretical guarantees on h-independence remain open, their results support AMG as a practical preconditioner for large-scale systems.

In light of this discussion, the present work focuses on clarifying how different preconditioners influence the performance of high-order finite element solvers for the Poisson equation, with emphasis on condition numbers, eigenvalue distributions, iteration counts, and runtime efficiency under uniform and adaptive refinement strategies.

Chapter 3: Methodology

3.1 Variational Formulation

We consider the Poisson equation in both one and two spatial dimensions with homogeneous Dirichlet boundary conditions. The strong form of the problem is defined as:

- 1D case (on a domain $\Omega = (0, L)$):

$$-\frac{d^2u}{dx^2} = f(x), \quad x \in (0, L), \quad u(0) = u(L) = 0 \tag{3.1}$$

- 2D case (on a domain $\Omega \subset R^{\mathbb{2}}$)

$$-\Delta u = f(x, y), \quad (x, y) \in \Omega, \quad u = 0 \text{ on } \partial\Omega \tag{3.2}$$

In both cases, the solution u is assumed to be sufficiently smooth, and f is a known source function.

To derive the weak form, the strong form is multiplied by a test function v belonging to the Sobolev space V: $H_0^1(\Omega)$, followed by integration by parts:

- 1D weak form:

$$\int_0^L \frac{du}{dx}\frac{dv}{dx}\, dx = \int_0^L f(x)v(x)\, dx \quad \forall v \in H_0^1(0, L) \tag{3.3}$$

- 2D weak form:

$$\int_\Omega \nabla u \cdot \nabla v\, d\Omega = \int_\Omega f(x, y)v(x, y)\, d\Omega \quad \forall v \in H_0^1(\Omega) \tag{3.4}$$

These weak forms (Equation 3.3 and 3.4) serve as the foundation for applying the finite element method (FEM), where the function space V is replaced by a finite-dimensional subspace V_h.

3.2 Finite Element Discretization

The domain Ω is discretized into a set of finite elements which is shown in Figure 3.1 (intervals in 1D, and typically triangles or quadrilaterals in 2D). Over this mesh, we define the finite dimensional space V_h consisting of continuous, piecewise polynomial basis functions of degree p, denoted as $\{\phi_j\}_{j=1}^n$. The approximate solution $u_h$is expressed as a linear combination of basis functions:

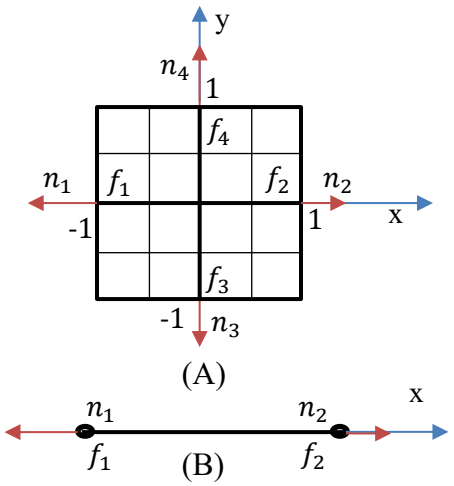


Figure 3.1: Reference domain (A) 2D: $[0,1] \times [0,1]$, f_i $(i = 1, \dots, 4)$ are defined on edges; (B) 1D: $[0,1]$, with f_i $(i = 1,2)$ specified at the endpoints. In both cases n_i denotes the outward unit normal vector associated with the corresponding boundary segment or point.

$$u_h(x) = \sum_{j=1}^{n} U_j \phi_j(x), \qquad (1D) \tag{3.5}$$

$$u_h(x,y) = \sum_{j=1}^{n} U_j \phi_j(x,y), \qquad (2D) \tag{3.6}$$

Substituting u_h into the weak form and selecting $v = \phi_i$, we obtain the discrete Galerkin system:

- 1D discrete form:

$$\sum_{j=1}^{n} U_j \int_0^L \frac{d\phi_j}{dx}\frac{d\phi_i}{dx}\, dx = \int_0^L f(x)\phi_i(x)\, dx \tag{3.7}$$

- 2D discrete form:

$$\sum_{j=1}^{n} U_j \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \, d\Omega = \int_\Omega f(x,y)\phi_i(x,y)\, d\Omega \tag{3.8}$$

In matrix form, the resulting linear system is:

$$K_{ij}U_i = F_i\,. \tag{3.9}$$

In Equation (3.9), $K_{ij} = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \, d\Omega$ is the stiffness matrix, $F_i = \int_\Omega f\, \phi_i \, d\Omega$ is the load (right hand side) vector and $U = [U_1, U_2, \dots, U_n]^T$ contains the unknown nodal values, and these parameters are locally calculated for each element.

3.3 Weighted Residual Approximation

In the finite element method, the solution u is approximated by a finite linear combination of basis functions from a finite-dimensional function space $X_h \subset X$. Here X denotes the infinite-dimensional function space where the exact solution u is defined (for

example, $H_0^1(\Omega)$ in the case of Poisson Problem), while X_h is its finite-dimensional subspace spans the chosen finite element basis functions. This approximate solution $u_h \in X_h$ can be expressed as:

$$u \approx u_h = \sum_{i=1}^{N} \overline{u_i}\ \phi_i(x)\,. \tag{3.10}$$

where $\phi_i(x)$ are the basis (trial) functions, $\overline{u_i}$ are the unknown coefficients (degrees of freedom) and N is the total number of basis functions. Similarly, we introduce a set of weight (test) functions $\psi_j \in X_m$, which may or may not be the same as the basis functions. The residual R is defined by substituting u_h into the Equation 3.1.

$$R(x) = \frac{d^2u_h}{dx^2} - f(x)\,. \tag{3.11}$$

In the method of weighted residuals, we enforce that the residual is orthogonal to each test function leading to the integral condition:

$$\int_\Omega R(x)\ \psi_j(x)\,dx = 0, \quad j = 1,2,\dots,t \tag{3.12}$$

This condition leads to a system of equations for the unknowns $\overline{u_i}$, where t is the number of test functions.

3.3.1 Galerkin Method as a Special Case

In the Galerkin method, the test functions are chosen to be the same as the basis functions:

$$\psi_j = \phi_j, \quad j = 1,\dots,N \tag{3.13}$$

This yields the Galerkin weak formulation:

$$\int_\Omega \left[\frac{d^2u_h}{dx^2} - f(x)\right]\phi_j(x)\ dx = 0, \quad j = 1,\dots,N \tag{3.14}$$

Substituting the approximation $u_h = \sum_{i=1}^{N} \overline{u_i}\ \phi_i$ the residual becomes:

$$R = -\sum_{i=1}^{N} \overline{u_i}\frac{d^2\phi_i}{dx^2} - f(x) \tag{3.15}$$

Applying the Galerkin condition:

$$\int_\Omega \left(-\sum_{i=1}^{N} \overline{u_i}\frac{d^2\phi_i}{dx^2} - f(x)\right)\phi_j(x)\ dx = 0 \tag{3.16}$$

To reduce the order of derivatives, we integrate by parts the Equation (3.16) and apply boundary conditions. This leads to the weak form; We start from the weighted residual form in both $1D$ and $2D$:

($1D$)

$$\int_{\Omega}\left(-\frac{d^2\varphi_i}{dx^2}-f(x)\right)\varphi_j\ dx=-\frac{d\varphi_i}{dx}\ \varphi_j\Big|_{\partial\Omega}+\int_{\Omega}\frac{d\varphi_i}{dx}\frac{d\varphi_j}{dx}\ dx \tag{3.17}$$

$(2D)$

$$\int_{\Omega}\left(-\sum_{i=1}^{N}\overline{u_i}\,\Delta\phi_i(x,y)-f(x,y)\right)\phi_j(x,y)\ dx\,dy=0 \tag{3.18}$$

Therefore, the full weak form becomes:

$(1D)$

$$\sum_{i=1}^{N}\overline{u_i}\left[-\frac{d\phi_i}{dx}\ \phi_j\Big|_{\partial\Omega}+\int_{\Omega}\frac{d\phi_i}{dx}\frac{d\phi_j}{dx}\ dx\right]=\int_{\Omega}f(x)\phi_j(x)\ dx \tag{3.19}$$

$(2D)$

$$\sum_{i=1}^{N}\overline{u_i}\left[-\int_{\partial\Omega}\frac{\partial\phi_i}{\partial n}\phi_j\ ds+\int_{\Omega}\nabla\phi_i\cdot\nabla\phi_j\ dx\,dy\right]=\int_{\Omega}f(x,y)\phi_j(x,y)\ dx\,dy \tag{3.20}$$

In the weak formulation, the boundary condition term arises from the integration by parts. In $1D$ and $2D$ the boundary terms are $-\frac{d\phi_i}{dx}\phi_j\Big|_{\partial\Omega}$, $-\int_{\partial\Omega}\frac{\partial\phi_i}{\partial n}\ \phi_j\ ds$ respectively; where $\frac{\partial\phi_i}{\partial n}$ is the normal derivative on the boundary $(2D)$, and $\partial\Omega$ denotes the boundary of the domain. These boundary terms are essential when applying Neumann boundary conditions, and they vanish if homogeneous Neumann boundary conditions are applied (i.e., zero flux across the boundary).

If the solution u satisfies homogeneous Dirichlet conditions in both Equations (3.19) and (3.20), then the basis and test functions vanish on the boundary:

$$\phi_j\big|_{\partial\Omega}=0\quad\Rightarrow\quad\frac{d\phi_i}{dx}\ \phi_j\Big|_{\partial\Omega}=0\,,\qquad-\int_{\partial\Omega}\frac{\partial\phi_i}{\partial n}\phi_j\ ds=0 \tag{3.21}$$

Thus, the weak form simplifies to:

$$\sum_{i=1}^{N}\overline{u_i}\int_{\Omega}\frac{d\phi_i}{dx}\frac{d\phi_j}{dx}\ dx=\int_{\Omega}f(x)\phi_j(x)\ dx \tag{3.22}$$

$$\sum_{i=1}^{N}\overline{u_i}\int_{\Omega}\nabla\phi_i\cdot\nabla\phi_j\ dx\,dy=\int_{\Omega}f(x,y)\phi_j(x,y)\ dx\,dy \tag{3.23}$$

This is the standard finite element formulation used to assemble the stiffness matrix and load vector.

3.4 Shape Functions and Interpolation

In the finite element method, we use shape functions on small parts called elements to find an approximate solution. These functions give the value of the unknown at any point

by using the values at certain points called nodes. In this work, we use Lagrange shape functions for both 1D and 2D cases. The nodes are placed at equal distances inside the reference domain. It is well documented that the use of equidistant nodal spacing for high-order Lagrange elements tends to cause Runge-type oscillations and significantly deteriorates the stiffness-matrix condition numbers [101]. For example, in $1D$, nodes are distributed evenly over the interval $[0,1]$. This uniform spacing simplifies the implementation and ensures consistent behavior of the basis functions across elements.

As shown in Figure 3.2 and 3.3, each Lagrange shape function has the value 1 at its own node and 0 at all other nodes. This property makes it possible to match the nodal values exactly. The nodes are placed at equal distances from each other, which simplifies the mapping and the Jacobian transformation. Gaussian quadrature points are not uniformly spaced, but they are defined as the reference element and mapped consistently to each physical element, so they can be used without difficulty.

3.4.1 One-Dimensional Lagrange Shape Functions

In the one-dimensional case, the reference element is defined as the interval $\Omega = [0,1]$. For a polynomial of degree p, we define $p + 1$ distinct interpolation nodes $\{\xi_0, \xi_1, \dots, \xi_p\}$ on this interval. The Lagrange shape function associated with node ξ_j is defined as:

$$l_j(\xi) = \prod_{m=0,\, m\neq j}^{p} \frac{\xi - \xi_m}{\xi_j - \xi_m}, \quad j = 0,1,\dots,p \tag{3.24}$$

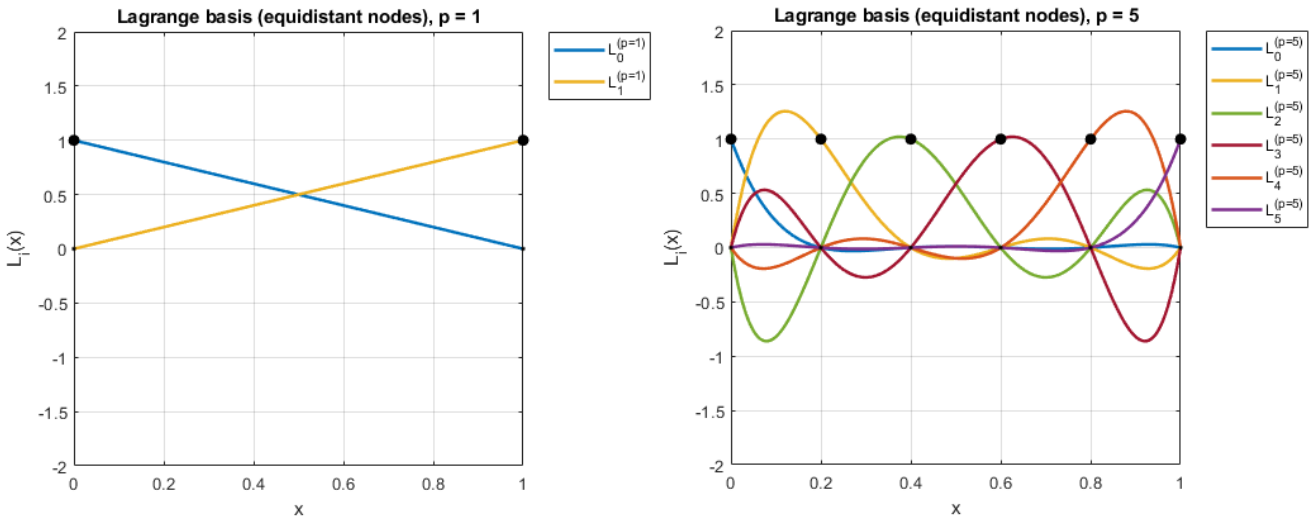
This function satisfies:

- The 1D Kronecker delta property:

$$l_j(\xi_i) = \delta_{ij}, \quad \text{for } i,j = 0,\dots,p \tag{3.25}$$

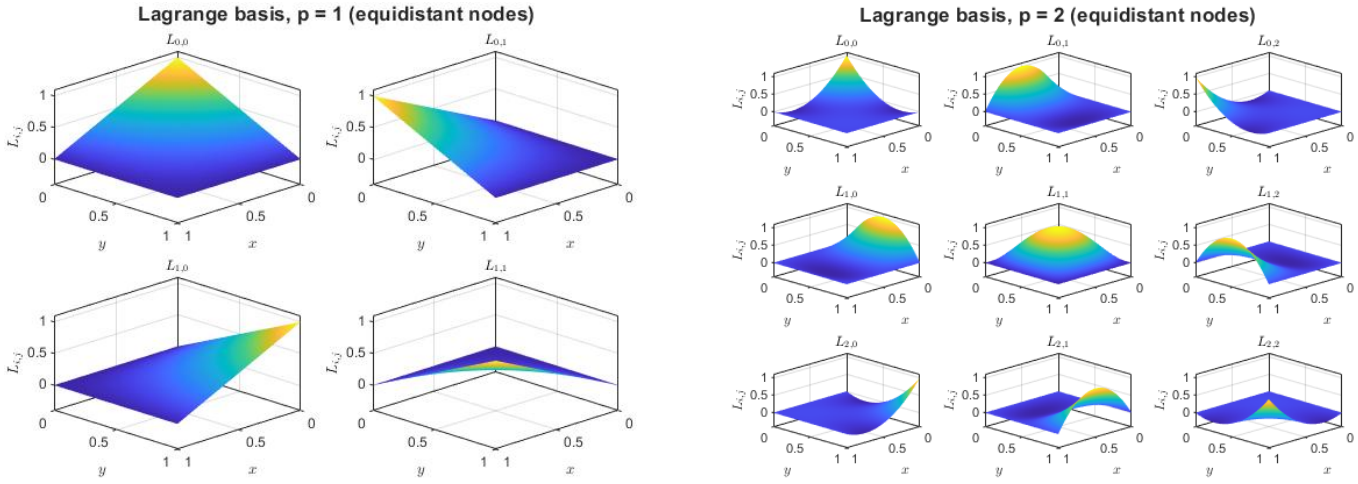
- The partition of unity:

$$\sum_{j=0}^{p} l_j(\xi) = 1, \quad \forall \xi \in [0,1] \tag{3.26}$$



A) B)

Figure 3.2: One-dimensional Lagrange basis functions constructed using equidistant nodes over the interval [0, 1]. Plot (A)shows the basis functions for polynomial degree $p = 1$,while plot (B) illustrates the basis for $p = 5$.



A) B)

Figure 3.3: Two-dimensional Lagrange basis functions defined over a reference quadrilateral element with uniformly spaced nodes in the domain $[0,1] \times [0,1]$. Plot (A) displays the basis functions for polynomial degree $p = 1$,, while Plot (B) corresponds to degree $p = 2$.

The approximation solution $u_h(\xi)$ in $1D$ element is expressed as a linear combination of Lagrange shape functions $l_j(\xi)$ and the nodal values U_j that are unknown solution values at the nodes ξ_i:

$$u_h(\xi) = \sum_{j=0}^{p} U_i l_j(\xi)\,. \tag{3.27}$$

The shape functions are constructed to satisfy the interpolation property:

$$l_j(\xi_i) = \delta_{ij} = \begin{cases} 1, & if\ i = j \\ 0, & if\ i \neq j \end{cases} \tag{3.28}$$

This ensures that $u_h(\xi) = U_i$ works for Equation (3.27) at the nodes ξ_i.

Additionally, the derivatives of the shape functions (Equation (3.24)) with respect to ξ are used in forming the stiffness matrix. The derivative of $l_j(\xi)$ is:

$$l_i'(\xi) = \sum_{m=0, m\neq j}^{p} \left[\prod_{n=0, n\neq j, n\neq m}^{p} \frac{\xi - \xi_n}{\xi_j - \xi_n} \cdot \frac{1}{\xi_j - \xi_m} \right] \tag{3.29}$$

3.4.2 Two-Dimensional Lagrange Shape Functions (Quadrilateral Elements)

For the two-dimensional case, the reference element is taken as the square $\Omega = [0,1] \times [0,1]$, and quadrilateral elements are used in the physical domain. To construct $2D$ shape functions, we use the tensor product of $1D$ Lagrange polynomials, equation (3.25). Let $\{\xi_0, \dots, \xi_p\}$ and $\{\eta_0, \dots, \eta_p\}$ be the interpolation points in the $\xi -$ and $\eta -$direction, respectively. Then, the shape function associated with node (ξ_i, η_j) is defined as:

$$\phi_{ij}(\xi_m, \eta_n) = l_i(\xi_m) \cdot l_j(\eta_n) \tag{3.30}$$

m, n refer to the coordinates of a particular node (ξ_m, η_n).

This function satisfies:

- $2D$ Kronecker delta property:

$$\phi_{ij}(\xi_m, \eta_n) = \delta_{im} \cdot \delta_{jn} \tag{3.31}$$

- Partition of unity:

$$\sum_{i=0}^{p} \sum_{j=0}^{p} \phi_{ij}(\xi_m, \eta_n) = 1, \quad \forall (\xi, \eta) \in [0,1]^2 \tag{3.32}$$

The approximate solution within a quadrilateral element is written as:

$$u_h(\xi_m, \eta_n) = \sum_{i=0}^{p} \sum_{j=0}^{p} U_{ij}\ \phi_{ij}(\xi_m, \eta_n) \tag{3.33}$$

where U_{ij} are the unknown solution values at the nodes (ξ_i, η_j), and $l_i(\xi)$ and $l_j(\eta)$ are $1D$ Lagrange shape function in the $\xi -$ and $\eta -$directions respectively. The $2D$ shape functions inherit the interpolation property in each coordinate direction:

$$\phi_{ij}(\xi, \eta) = l_i(\xi_m) \cdot l_j(\eta_n) = \begin{cases} 1, & if\ (i,j) = (m,n) \\ 0, & otherwise \end{cases} \tag{3.34}$$

As in one-dimensional this guarantees that $u_h(\xi_m, \eta_n) = U_{ij}$, ensuring exact interpolation at nodal locations.

The gradients of the shape functions (Equation (3.30)) are computed via partial derivatives of the 1D basis functions:

$$\frac{\partial \phi_{ij}}{\partial \xi} = l_i'(\xi_m) \cdot l_j(\eta_n), \quad \frac{\partial \phi_{ij}}{\partial \eta} = l_i(\xi_m) \cdot l_j'(\eta_n) \tag{3.35}$$

Finally, these derivatives in $1D$ and $2D$ are transformed into physical coordinates using the Jacobian, as described in the next section (Equation (3.47) (A)).

3.4.3 Barycentric Interpolation and Weight Computation

To improve stability and accuracy, particularly for high polynomial degrees the Lagrange shape functions are implemented using the barycentric form of interpolation.
The barycentric interpolant in $1D$ is given by:

$$u_h(\xi) = \frac{\sum_{j=0}^{p} \frac{w_j}{\xi - \xi_j} U_j}{\sum_{j=0}^{p} \frac{w_j}{\xi - \xi_j}}, \qquad \xi \neq \xi_j \tag{3.36}$$

At the interpolation nodes, the expression reduces to:

$$u_h(\xi_j) = U_j, \tag{3.37}$$

The barycentric weights w_j are computed as:

$$w_j = \frac{1}{\prod_{0 \leq m \leq p,\, m \neq j} (\xi_j - \xi_m)}, \tag{3.38}$$

These weights are calculated one time for the reference element and then used again for finding both shape functions and their derivatives. It is important to say that w_j are not the quadrature weight, which we will see later in this chapter.

In $2D$ (quadrilateral elements), the barycentric form is extended via tensor products. The weights in the $\xi -$ and $\eta -$ directions are used to compute:

$$\phi_{ij}(\xi, \eta) = \frac{\sum_{k=0}^{p} \frac{w_k^{\xi}}{\xi - \xi_k} \delta_{ik}}{\sum_{k=0}^{p} \frac{w_k^{\xi}}{\xi - \xi_k}} \cdot \frac{\sum_{l=0}^{p} \frac{w_l^{\eta}}{\eta - \eta_l} \delta_{jl}}{\sum_{l=0}^{p} \frac{w_l^{\eta}}{\eta - \eta_l}}. \tag{3.39}$$

Equations 3.38 and 3.39 allow fast and numerically stable evaluation of high-order shape functions in both $1D$ and $2D$.

3.5 Numerical Integration

Accurate numerical integration is very important in the finite element method, especially when using high-order shape functions. In this work, I use Gaussian quadrature to calculate integrals for both one-dimensional and two-dimensional elements. The

integration is first done on the reference element, and then the results are moved to the physical element by using the Jacobian determinant.

3.5.1 Gaussian Quadrature in $1D$

Consider The $1D$ integral over an interval $[a,b]$:

$$\int_a^b g(x)\ dx\,. \tag{3.40}$$

This is transformed into the reference interval $[0,1]$ using an affine mapping, and the integral is approximated by:

$$\int_0^1 g(\xi)\ d\xi \approx \sum_{i=1}^{N} w_i\ g(\xi_i), \tag{3.41}$$

where: $\{\xi_i\}_{i=1}^{N}$ are the Gauss quadrature points (not equally spaced but chosen as the roots of Legendre polynomials on the reference interval), and $\{w_i\}_{i=1}^{N}$ are the corresponding Gauss quadrature weights. Here, N is the number of quadrature points. The rule is exact for polynomials of degree up to $2N-1$.

For integrals over a physical element $e=[x_a,x_b]$, the transformation is:

$$x(\xi) = x_a + (x_b - x_a)\,\xi, \qquad \xi \in [0,1] \tag{3.42}$$

or, in the case of Gauss–Legendre quadrature where the reference domain is $[-1,1]$,

$$x(\xi) = \frac{1}{2}[(1-\xi)x_a + (1+\xi)x_b], \qquad \xi \in [-1,1] \tag{3.42}$$

and the integral becomes:

$$\int_{x_a}^{x_b} g(x)\ dx = (x_b - x_a)\int_0^1 g\big(x(\xi)\big)\ d\xi \approx (x_b - x_a)\sum_{i=1}^{N} w_i\ g\big(x(\xi_i)\big) \tag{3.43}$$

3.5.2 Gaussian Quadrature in $2D$ (Quadrilateral Elements)

In $2D$, integration is carried out over the reference square $\widehat{\Omega} = [0,1]\times[0,1]$ using tensor-product Gauss quadrature:

$$\iint_{\Omega} g(\xi,\eta)\ d\xi d\eta \approx \sum_{i=1}^{N}\sum_{j=1}^{N} w_i w_j\ g\big(\xi_i,\eta_j\big), \tag{3.44}$$

For each quadrilateral element in the physical domain, bilinear mapping is used from the reference coordinates (ξ,η) to physical coordinates (x,y), defined as:

$$x(\xi,\eta) = \sum_{k=1}^{n} x_k \phi_k(\xi,\eta)\,, \quad y(\xi,\eta) = \sum_{k=1}^{n} y_k \phi_k(\xi,\eta) \tag{3.45}$$

where $\{(x_k,y_k)\}_{k=1}^{n}$ are the coordinates of the element's nodes. And the integral over the physical element becomes:

$$\iint_{\Omega_e} g(x,y)\ dxdy = \iint_{\widehat{\Omega}} g\big(x(\xi,\eta), y(\xi,\eta)\big)\ |J(\xi,\eta)|\, d\xi d\eta \tag{3.46}$$

where $|J(\xi,\eta)|$ is the Jacobian determinant of the transformation:

$$J(\xi,\eta) = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \eta} & \frac{\partial y}{\partial \xi} \end{bmatrix}, \qquad (A) \tag{3.47}$$

$$|J(\xi,\eta)| = \left|\frac{\partial x}{\partial \xi}\frac{\partial y}{\partial \xi} - \frac{\partial x}{\partial \eta}\frac{\partial y}{\partial \eta}\right|, \qquad (B)$$

The numerical approximation is then:

$$\iint_{\Omega_e} g(x,y)\ dx\, dy \approx \sum_{i=1}^{N}\sum_{j=1}^{N} w_i w_j\ g\left(x(\xi_i,\eta_j), y(\xi_i,\eta_j)\right)\ |J(\xi_i,\eta_j)| \tag{3.48}$$

3.5.3 Application to Stiffness Matrix and Load Vector

With the numerical integration methods explained above, I calculate the entries of the local stiffness matrix and the load vector for each element.

- $1D$ element stiffness matrix:

$$K_{ij}^{(e)} = \int_{x_a}^{x_b} l_i'(x)\ l_j'(x)\, dx \approx \sum_{k=1}^{N} w_k\ l_i'(\xi_k)\, l_j'(\xi_k)\, J(\xi_k). \tag{3.49}$$

- $2D$ element stiffness matrix:

$$K_{ij}^{(e)} = \iint_{\Omega_e} \nabla\phi_i \cdot \nabla\phi_j\, dx\, dy \approx \sum_{k=1}^{N}\sum_{l=1}^{N} w_k w_l\ \big(\nabla\phi_i \cdot \nabla\phi_j\big)(\xi_k,\eta_l)\ J(\xi_k,\eta_l). \tag{3.50}$$

In Equation 3.49 $J(\xi_k)$: Jacobian $= \frac{\partial x}{\partial \xi}$ which maps from the reference interval [0,1] to the element e $[x_a, x_b]$.

- The right-hand side in $1D$ is given by:

$$F_i^{(e)} = \int_{\Omega_e} f(x)\phi_i(x)\ dx \approx \sum_{k} w_k\ f(x_k)\, \phi_i(x_k) \cdot J(\xi_k) \tag{3.51}$$

- Right hand in $2D$:

$$F_i^{(e)} = \iint_{\Omega_e} f(x,y)\ \phi_i(x,y)\, dx\, d \approx \sum_{k=1}^{N}\sum_{l=1}^{N} w_k w_l\ f(\xi_k,\eta_l)\, \phi_i(\xi_k,\eta_l) J(\xi_k,\eta_l) \tag{3.52}$$

For Equations (3.49) to (3.51), w_k and also w_l are the quadrature weights for element e.

In real problems, I do the integration for each element separately. After that, I put them together to make the global system, which will be solved in the next sections.

3.6 Assembly of Global Matrices

Once the local element stiffness matrices and load vectors are computed using numerical integration, they must be assembled into a global system of the form:

$$\mathrm{KU} = \mathrm{F}. \tag{3.53}$$

where K is the global stiffness matrix of size $n_{\text{dof}} \times n_{\text{dof}}$, F is the global load (right-hand side) vector and U contains the global unknown nodal values with size of n_{dof}. This process is called assembly and is fundamental to the finite element method.

3.6.1 Local-to-Global Mapping and Implementation

Each element contributes a local stiffness matrix $K^{(e)}$ and local vector $F^{(\mathrm{e})}$, computed using:

$$K_{ij}^{(e)} = \int_{\Omega_e} \nabla\phi_i \cdot \nabla\phi_j \, dx, \qquad (A) \tag{3.54}$$

$$F_i^{(e)} = \int_{\Omega_e} f\,\phi_i \, dx, \qquad (B)$$

To incorporate these into the global matrix and vector, we use a local-to-global mapping array, denoted as $\mathrm{elem\_dof}\,[e]$, which associates each local node donated by $i \; and \; j$, in element e to the corresponding global degree of freedom index I, J:

$$K_{IJ} \mathrel{+}= K_{ij}^{(e)}, \quad F_I \mathrel{+}= F_i^{(e)} \tag{3.55}$$

where I = $\mathrm{elem\_dof}[e][i]$, J = $\mathrm{elem\_dof}[e][j]$. This process is repeated for all elements in the mesh.

In $1D$, for a polynomial of degree p, each element contains $p + 1$ nodes. The global node numbering is typically sequential, and the local-to-global map is straightforward. In $2D$, for quadrilateral elements of degree p, each element contains $(p + 1)^{\,2}$ nodes.

The global nodes are numbered in a structured grid. I make the local-to-global mapping from a mesh connectivity table. In both cases, the global stiffness matrix is sparse and symmetrical. Figure 3.4 shows the assembly step for each element, where I calculate the local stiffness matrices and load vectors and then put them into the global system. Figure 3.5 shows an example of local-to-global mapping for quadrilateral elements in a 2D structured mesh and then gives a simple view of how the local stiffness matrices join together to make the global matrix.

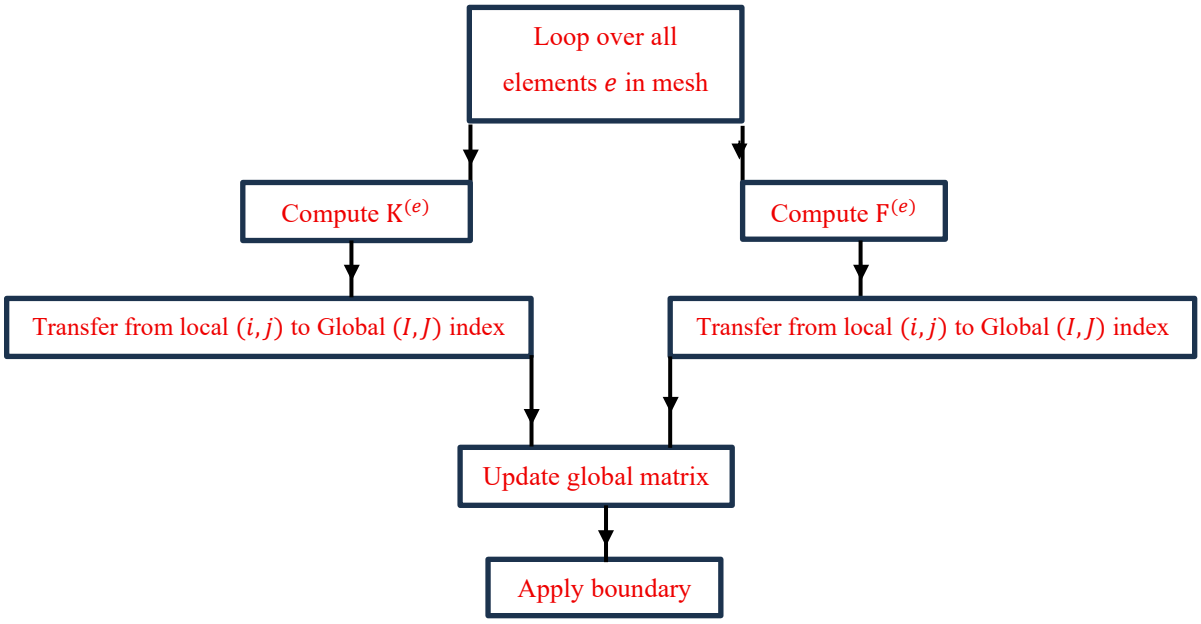
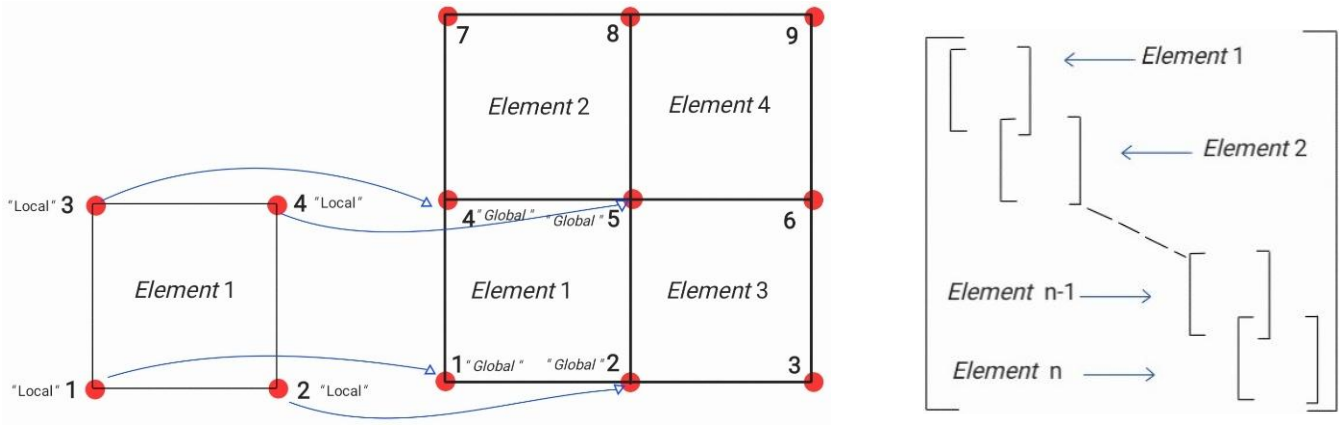


Figure 3.4: Steps of the assembly process: local stiffness matrices and local vectors are computed for each element.



A) B)

Figure 3.5: (A) Mapping between local and global node numbering in a structured 2D quadrilateral mesh. (B) Schematic illustration of the global stiffness matrix assembly, where local element contributions are inserted and summed at positions corresponding to shared global nodes.

3.7 Error Estimation

To check the numerical error, an a posteriori method is used based on the spectral decay of modal coefficients [33]. The finite element solution is built with Lagrange shape functions on equally spaced nodes, and then it is projected onto the orthogonal Legendre basis to find useful error indicators. Looking at the behavior of the high-order modes makes it possible to estimate the truncation error in the L^2-norm (see Equation (3.69)).

We define two types of L^2 errors:

1) Elementwise L^2 Error: Used locally to guide adaptive refinement.

2) Global L^2 Norm: Used to evaluate the overall accuracy of the computed solution across the domain.

3.7.1 Spectral Expansion

Let $u_h(x)$ denote the numerical solution. We express its spectral representation using an infinite expansion in Legendre polynomials:

$$u(x) = \sum_{n=0}^{\infty} a_n L_n(x) \tag{3.56}$$

where $L_n(x)$ denotes the Legendre polynomial of degree n and a_n are corresponding modal coefficients. In practice, we cannot retain infinitely many modes. Thus, we truncate expansion at a finite order N, giving:

$$u(x) \approx \sum_{n=0}^{N} a_n L_n(x) \tag{3.57}$$

with the remaining high-frequency content represented by the truncation residual:

$$\tau(x) = \sum_{n=N+1}^{\infty} a_n L_n(x). \tag{3.58}$$

This truncation error $\tau(x)$ is part of the solution that cannot be represented in the polynomial space formed by the chosen finite basis.

3.7.1.1 Orthogonality and Coefficient Calculation

To estimate the truncation error $\tau = \sum_{n=N+1}^{\infty} a_n L_n(x)$, we compute the modal coefficients a_n for $n > N$. These coefficients are not directly available from the finite element approximation but can be determined by exploiting the orthogonality of Legendre polynomials. The inner product of two Legendre polynomials on $[-1,1]$ satisfies:

$$\langle L_n, L_m \rangle = \int_{-1}^{1} L_n(x) L_m(x)\ dx = \frac{2}{2n+1} \delta_{nm} \tag{3.59}$$

where δ_{nm} is the Kronecker delta (equal to 1 if $n = m$, and 0 otherwise). To extract the coefficient a_n , we take the inner product of the function $u_h(x)$ with $L_n(x)$:

$$\int_{-1}^{1} u_h(x) L_n(x)\ dx = \int_{-1}^{1} \left(\sum_{k=0}^{\infty} a_k L_k(x) \right) L_n(x)\ dx \tag{3.60}$$

Interchanging the sum and the integral:

$$= \sum_{k=0}^{\infty} a_k \int_{-1}^{1} L_k(x) L_n(x)\ dx\ . \tag{3.61}$$

Due to orthogonality, all terms vanish except for $k = n$, yielding:

$$\int_{-1}^{1} u_h(x) L_n(x)\ dx = a_n \cdot \frac{2}{2n+1}. \tag{3.62}$$

Solving for a_n, we obtain the closed-form expression:

$$a_n = \frac{2n+1}{2} \int_{-1}^{1} u_h(x) L_n(x)\ dx. \tag{3.63}$$

This formula allows us to project the numerical solution onto the Legendre basis and extract the modal coefficients required for error estimation.

3.7.1.2 L^2 Error Evaluation

To quantitatively assess the accuracy of the numerical solution, we compute the L^2 norm of the error between the exact solution $u(x)$ and the numerical approximation $u_h(x)$.

3.7.1.2.1 Elementwise L^2 Error:

Supposed our domain Ω is partitioned into finite elements Ω_e, where $e = 1,2,\dots,N_e$. Within each element Ω_e, the local L^2 error norm is computed as:

$$\left|\left|u - u_h\right|\right|_{L^2(\Omega_e)} \approx \left(\sum_{i=1}^{n_q} w_i \big(u(x_i) - u_h(x_i)\big)^2 \right)^{1/2} \tag{3.64}$$

where $u(x_i)$ is the exact solution at $x_i \in \Omega_e$, and $u_h(x_i)$ is the corresponding numerical approximation at the same points. The quadrature points x_i and weight w_i are defined as in Equation (3.41), based on Gauss quadrature rules. Here n_q denotes the number of quadrature points used within element Ω_e. Equation (3.65) evaluates the elementwise L^2 error norm by integrating the squared difference between the exact and numerical solutions over each element domain. In this way, both the magnitude of the error and the geometric size of each element are properly considered, making it an effective criterion for adaptive refinement. In essence, Equation (3.65) is a discrete approximation of the continuous error norm:

$$\left|\left|u - u_h\right|\right|_{L^2(\Omega_e)} = \left(\int_{\Omega_e} \big(u(x) - u_h(x)\big)^2 dx \right)^{1/2} \tag{3.65}$$

But since we already have u and u_h are already evaluated at quadrature points, the continuous integral is computed numerically using a weighted sum, consistent with the quadrature rule.

Similarly, in 2D:

$$\left|\left|u-u_h\right|\right|_{L^2(\Omega_e)}=\left(\sum_{i=1}^{n}\sum_{j=1}^{n}w_iw_j\cdot\left(u(x_i,y_j)-u_h(x_i,y_j)\right)^2\right)^{1/2}. \tag{3.66}$$

3.7.1.2.2 Global L^2 Error:

The global L^2 norm provides a scalar measure of the total error over the entire domain Ω, to assess the overall accuracy across the full domain $\Omega=\bigcup_{e=1}^{N_e}\Omega_e$, the global error norm can be defined in two ways:

a) With local weights (elementwise quadrature, more accurate than uniform averaging)

The global error is the square root of the sum of squared local errors:

$$\left|\left|u-u_h\right|\right|_{L^2(\Omega)}=\left(\sum_{e=1}^{N_e}\sum_{i=1}^{n_q^e}w_i^e\left(u(x_i^e)-u_h(x_i^e)\right)^2\right)^{1/2} \tag{3.67}$$

where the entire computational domain is Ω and also N_e , n_q^e are known as total number of elements and number of quadrature points in element e respectively. The $i-th$ quadrature point in element e and quadrature weight at point are x_i^e and w_i^e in order. This formulation correctly accounts for non-uniform grid spacing and variable element sizes.

Similarly, in 2D:

$$\left|\left|u-u_h\right|\right|_{L^2(\Omega)}=\left(\sum_{e=1}^{N_e}\sum_{i=1}^{n}\sum_{j=1}^{n}w_iw_j\cdot\left(u\left(x_i^{(e)},y_j^{(e)}\right)-u_h\left(x_i^{(e)},y_j^{(e)}\right)\right)^2\right)^{1/2}. \tag{3.68}$$

b) With uniform averaging (simplified form, less accurate than local weights)

If the domain is discretized with uniform spacing, or for simplicity in evaluation, an alternative form normalizes the sum by the total number of sampling points n:

$$\left|\left|u-u_h\right|\right|_{L^2(\Omega)}=\left(\frac{1}{N_e}\sum_{e=1}^{N_e}\sum_{j=1}^{n}\left(u(x_i)-u_h(x_i)\right)^2\right)^{1/2} \tag{3.69}$$

3.7.1.3 Truncation Error

Here, the spectral approximation is limited to a polynomial degree N. As a result, the higher-frequency components of the solution, found in terms beyond this degree, are left out. This missing information is known as the truncation error $\tau(x)$. Since these higher-order coefficients a_n for $n>N$ are not computed directly in the solution, we cannot evaluate the truncation error exactly. Rather than computing it exactly, we estimate it by observing the decay pattern of the known coefficients and applying extrapolation to predict the missing ones. This allows us to approximate the L^2 Norm $\left|\left|\tau\right|\right|=\left|\left|u_{\text{trunc}}\right|\right|$ of the truncation error, which serves as a quantitative measure of the missing spectral content and is critical for guiding adaptive strategies.

To quantify the magnitude of this error, we compute the L^2 norm of the truncation error over the reference interval $[-1,1]$ for $1D$ and then extend it for $2D$, given by:

$$||\tau|| = ||u_{\text{trunc}}|| = \left\| \sum_{n=N+1}^{\infty} a_n L_n(x) \right\|_{L^2}. \tag{3.70}$$

Expanding this norm explicitly:

$$||u_{\text{trunc}}|| = \left(\int_{-1}^{1} \left(\sum_{n=N+1}^{\infty} a_n L_n(x) \right)^2 dx \right)^{1/2}. \tag{3.71}$$

Assuming the coefficients decay fast enough, we can approximate this using predicted coefficients $\widetilde{a_n}$

$$||u_{\text{trunc}}|| \approx \left(\sum_{n=N+1}^{\infty} \widetilde{a_n^2} \cdot \int_{-1}^{1} L_n^2(x)\ dx \right)^{1/2}. \tag{3.72}$$

Using the orthogonality again:

$$\int_{-1}^{1} L_n^2(x)\ dx = \frac{2}{2n+1} \tag{3.73}$$

we finally get:

$$||u_{\text{trunc}}|| \approx \left(\sum_{n=N+1}^{\infty} \frac{\widetilde{a_n^2} \cdot 2}{2n+1} \right)^{1/2}. \tag{3.74}$$

This formula gives a useful and reliable way to measure the part of the solution that is not resolved in the spectrum. It can help to choose where and how to refine the mesh by looking at where the error is and controlling it.

Similarly, in $2D$:

$$||u_{\text{trunc}}|| = \left(\sum_{n=N+1}^{\infty} \sum_{m=M+1}^{\infty} \frac{\widetilde{a_{nm}^2}}{\frac{(2n+1)(2m+1)}{2^2}} \right)^{1/2}. \tag{3.75}$$

3.7.1.4 Quadrature Errors

In spectral or high-order finite element methods, when integrals are calculated numerically (for example with Gaussian quadrature, or even with simpler rules using equally spaced nodes), some quadrature error is inevitable. This becomes more noticeable when the integrand contains polynomial terms of high degree. The discussion below provides a way of estimating quadrature-induced and other approximation errors beyond truncation error, rather than a strict definition of quadrature error from quadrature formulas.

Assume that the exact solution in $1D$ is expanded as:

$$u(x) = \sum_{n=0}^{N} b_n L_n(x), \tag{3.76}$$

and the numerical approximation based on quadrature-integrated stiffness and mass matrices gives:

$$u_h(x) = \sum_{n=0}^{N} a_n L_n(x), \tag{3.77}$$

where $L_n(x)$ are Legendre polynomials, and b_n, a_n are the corresponding spectral coefficients of the exact and numerical solutions, respectively. The quadrature error in the L^2 norm is defined as:

$$\left|\left|u_{\text{quad}}\right|\right|_{L^2} = \left(\int_{-1}^{1} \left[\sum_{n=0}^{N} (b_n - a_n) L_n(x) \right]^2 dx \right)^{1/2}. \tag{3.78}$$

Using the orthogonality of Legendre polynomials and assuming normalized basis functions, this simplifies to:

$$\left|\left|u_{\text{quad}}\right|\right|_{L^2} = \left(\sum_{n=0}^{N} \frac{(b_n - a_n)^2}{\frac{2n+1}{2}} \right)^{1/2}. \tag{3.79}$$

In practice, however, the exact coefficients b_n are not available, so we approximate the dominant quadrature error by its final term:

$$\left|\left|u_{\text{quad}}\right|\right|_{L^2} \approx \left(\frac{(a_N)^2}{\frac{2N+1}{2}} \right)^{1/2} \tag{3.80}$$

This gives an upper limit for the error caused by quadrature compared to the exact value of the integral.

Similarly, in two dimensions assuming a tensor-product basis with maximum polynomial orders N and M in x and y directions respectively, the quadrature error becomes:

$$\left|\left|u_{\text{quad}}\right|\right|_{L^2} \approx \left(\sum_{j=0}^{M} \frac{(a_{Nj})^2}{\frac{2N+1}{2}} + \sum_{i=0}^{N} \frac{(a_{Mi})^2}{\frac{2M+1}{2}} \right)^{1/2}. \tag{3.81}$$

In the Equation (3.81) a_{Nj} and a_{iM} are spectral coefficients at the edge of polynomial orders, i.e., highest-index terms along $x-$ and $y-$directions.

3.7.1.5 Spectral Decay-Based Error Indicator for Adaptivity

To guide adaptivity in our numerical method, we analyze the behavior of the modal coefficients a_n obtained by projecting the numerical solution $u_h(x)$ onto the Legendre polynomial basis. For sufficiently smooth solutions and high polynomial degrees, these coefficients are expected to decay exponentially:

$$a_n \approx Ce^{-\sigma n} \tag{3.82}$$

where C is a constant depending on the amplitude of the solution, σ is the decay rate and n is the modal index.

The exponential decay shown in Equation (3.82) is a sign that the solution is well resolved. When the decay is fast (large σ), it means that the missing high-frequency part is very small, so the current polynomial order is enough. If the decay is slow (small σ), it means that the present approximation is missing some important details of the solution, and it may be necessary to increase the polynomial order (p −adaptivity). To extract the decay rate σ from the modal coefficients a_n, we take the natural logarithm of both sides of Equation (3.82):

$$\ln(|a_n|) = \ln(C) - \sigma n \tag{3.83}$$

we assume that $y_n = \ln(|a_n|)$ and $x_n = n$ then the equation becomes a linear model:

$$y_n = \alpha + \beta x_n \tag{3.84}$$

where $\alpha = \ln(C)$ and $\beta = -\sigma$ This transforms our nonlinear decay model into a linear regression problem. Next step is fitting a straight line to at least 4 nonzero modes using linear least squares. Given data points $(x_i, y_i) = \left(n_i, \ln\left|a_{n_i}\right|\right)$, the least-squares solution for the best-fit line $y = \alpha + \beta x$ is given by:

$$\beta = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{N}(x_i - \bar{x})^2}, \quad where \qquad \bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i\,, \quad \bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i \tag{3.85}$$

$$\alpha = \bar{y} - \beta\bar{x}, \tag{3.86}$$

From the slope β, we retrieve the decay rate $\sigma = -\beta$ and from the intercept α, we reconstruct the constant $C = e^{\alpha}$.

3.7.1.6 Total Error Estimation

The total numerical error is estimated by adding the truncation error and the quadrature error. The total estimated error is given as:

$$\epsilon_{\text{est}} = \left|\left|u - u_h\right|\right|_{L^2(\Omega)} \approx \left(\epsilon_{\text{trunc}}^2 + \epsilon_{\text{quad}}^2\right)^{1/2} \tag{3.87}$$

when Gauss-Legendre–Lobatto (GLL) points are used both for interpolation and quadrature, both types of error must be considered. The total estimated error in $1D$ and $2D$ is given by:

$$\epsilon_{\text{est}} \approx \left(\frac{a_N^2}{\frac{2n+1}{2}} + \sum_{n=N+1}^{\infty} \frac{a_n^2}{\frac{2n+1}{2}}\right)^{1/2}. \qquad (1D) \tag{3.88}$$

$$\epsilon_{\text{est}} \approx \left(\sum_{j=0}^{M} \frac{a_{Nj}^2}{\frac{2N+1}{2}} + \sum_{i=0}^{N} \frac{a_{iM}^2}{\frac{2M+1}{2}} + \sum_{n=N+1}^{\infty}\sum_{m=M+1}^{\infty} \frac{a_{nm}^2}{\frac{(2n+1)(2m+1)}{2^2}}\right)^{1/2}. \qquad (2D) \tag{3.89}$$

In this thesis, Gauss-Legendre (GL) quadrature is used for numerical integration, which eliminates error for integrands up to degree $2N-1$. Thus, only truncation errors remain:

1D:

$$\epsilon_{\text{est}} \approx \left(\sum_{n=N+1}^{\infty} \frac{a_n^2}{\frac{2n+1}{2}} \right)^{1/2}. \tag{3.90}$$

2D:

$$\epsilon_{\text{est}} \approx \left(\sum_{n=N+1}^{\infty} \sum_{m=M+1}^{\infty} \frac{a_{nm}^2}{\frac{(2n+1)(2m+1)}{2^2}} \right)^{1/2}. \tag{3.91}$$

The infinite series can be approximated as integrals to obtain an upper bound:

$$\epsilon_{\text{est}} \leq \left(\int_{N+1}^{\infty} a_n^2 \, dn \right)^{1/2} \quad (1D) \tag{3.92}$$

$$\epsilon_{\text{est}} \leq \left(\int_{M+1}^{\infty} \int_{N+1}^{\infty} a_{nm}^2 \, dn \, dm \right)^{1/2} \quad (2D) \tag{3.93}$$

According to Equation (3.93) we are able to evaluate the truncation error, in $1D$ the evaluation becomes:

$$\epsilon_{\text{est}} \approx \left(\int_{N+1}^{\infty} (Ce^{-\sigma n})^2 \, dn \right)^{1/2} = \left(\frac{C^2}{2\sigma} \right)^{1/2} e^{-\sigma(N+1)}. \tag{3.94}$$

For the 2D case, the spectrum from Equation (3.94) does not show a monotonic decay. In other words, the decrease of the coefficients in two dimensions is not always separable or monotonic in both directions. This makes the expression more difficult to work with directly. To overcome this, we create an equivalent one-dimensional average spectrum, $\overline{a_N}$, by collapsing the 2D spectrum into a single sequence.

We assume equal polynomial order N in both $x-$and $y-$directions. To obtain the averaged spectrum we define:

$$\overline{a_N} = |a_{N,N}| + \sum_{i=0}^{N-1} \left(|a_{i,N}| + |a_{N,i}| \right) \tag{3.95}$$

Equation (3.95) gives one-dimensional sequences $\overline{a_N}$ that represent the spectral energy along the highest mode in both directions. Next, using the averaged $1D$ spectrum $\overline{a_p}$, Equation (3.93) now can be written as:

$$\epsilon_{\text{est}} = \left(\int_{M+1}^{\infty} \int_{N+1}^{\infty} a_{nm}^2 \, dn \, dm \right)^{1/2} \tag{3.96}$$

we simplify this expression (Equation (3.94)) using the collapsed 1D spectrum:

$$= \left(\int_{N+1}^{\infty} \overline{a_p^2} \, dp \right)^{1/2}, \tag{3.97}$$

Assuming that $\overline{a_p}$ decays exponentially as:

$$\overline{a_p} = Ce^{-\sigma p} \tag{3.98}$$

we substitute into the integral:

$$= \left(\int_{N+1}^{\infty} C^2 e^{-2\sigma p}\, dp \right)^{1/2}, \tag{3.99}$$

This gives:

$$= \left(\frac{C^2}{2\sigma} e^{-2\sigma(N+1)} \right)^{1/2} = \left(\frac{C}{\sqrt{2\sigma}} \right) e^{-\sigma(N+1)} = \left(\frac{C^{-2}}{2\sigma} \right)^{-1/2} e^{-\sigma(N+1)}. \tag{3.100}$$

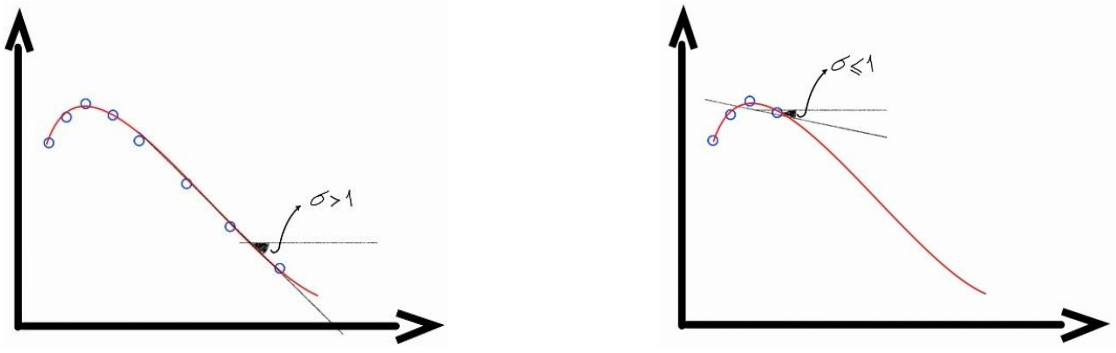
The decay rate σ is found by applying linear least-squares fit to the logarithm of the averaged spectrum a_n, as explained in Section 3.8.1.5. With this value, the truncation error can be estimated more accurately from how the modal coefficients decay.

3.8 Adaptivity implementation

In this section I began by discussing $h-$, p $-$, and hp $-$adaptivity for one-dimensional settings. Then I extend the discussion to two-dimensional, focusing specifically on p-adaptivity.

3.8.1 hp $-$Aadaptivity in $1D$:

The computed decay rate which has been written in section 3.7.1.5 σ plays as an error indicator and to decide between h- and p-refinement. If $\sigma > 1$, the coefficients decay rapidly [Figure 3.6 (a)] meaning the solution is smooth and well-resolved. In this case, increasing the polynomial degree (p-refinement) is effective. If $\sigma \leq 1$, the decay is slow [Figure 3.6 (b)], suggesting the presence of sharp features or unresolved behavior, refining the mesh (h-refinement) is more appropriate. This criterion helps choose the most efficient refinement strategy based on the smoothness of the local solution. Figure 3.8 illustrates the application of hp $-$adaptivity to the 1D Poisson problem $-u'' = x^{50}$ on the interval [0, 1], which exhibits a sharp gradient near the right boundary. This makes it an ideal benchmark for testing adaptive mesh refinement strategies.



A) B)

Figure 3.6: Spectrum decay trend of modal coefficients: (A) $\sigma > 1$ indicates a good-quality element, while (B) $\sigma \leq 1$ indicates a poor-quality element with faster decay.

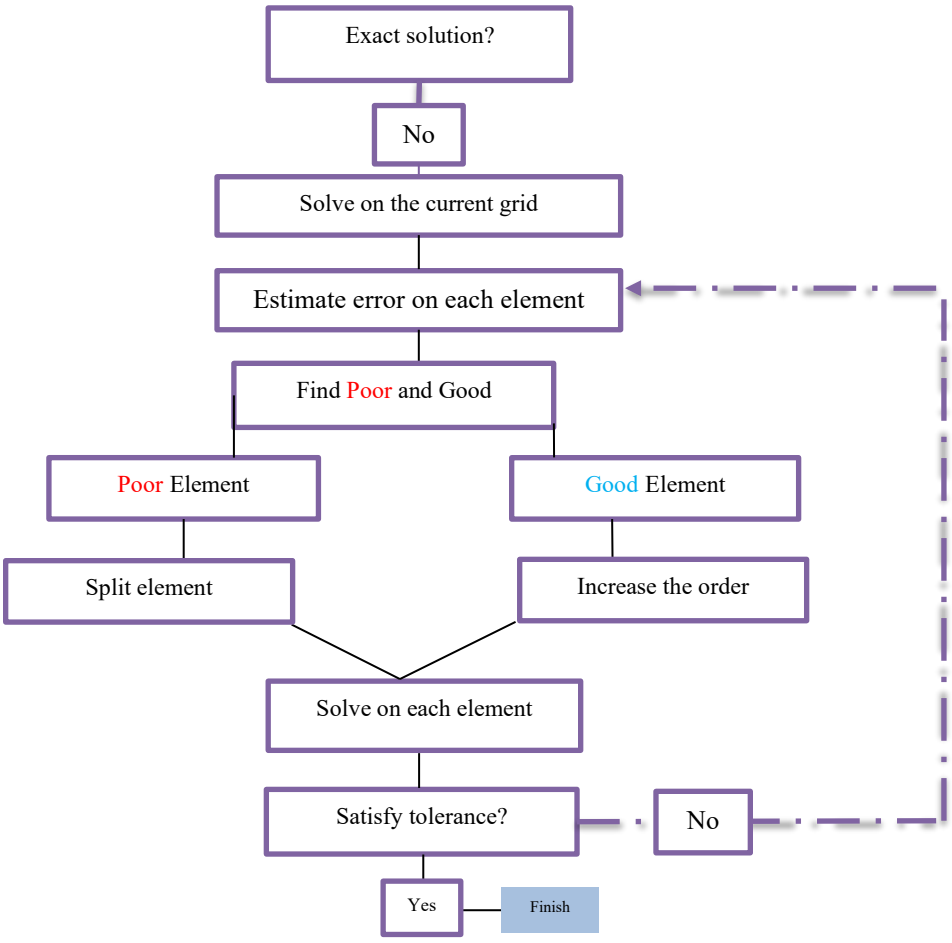


Figure 3.7: Illustrating the decision-making process in an hp −adaptivity framework. The algorithm identifies poor and good elements based on error estimation, then selectively refines the mesh (h −adaptivity) or increases the polynomial order (p −adaptivity) to efficiently satisfy the desired tolerance.

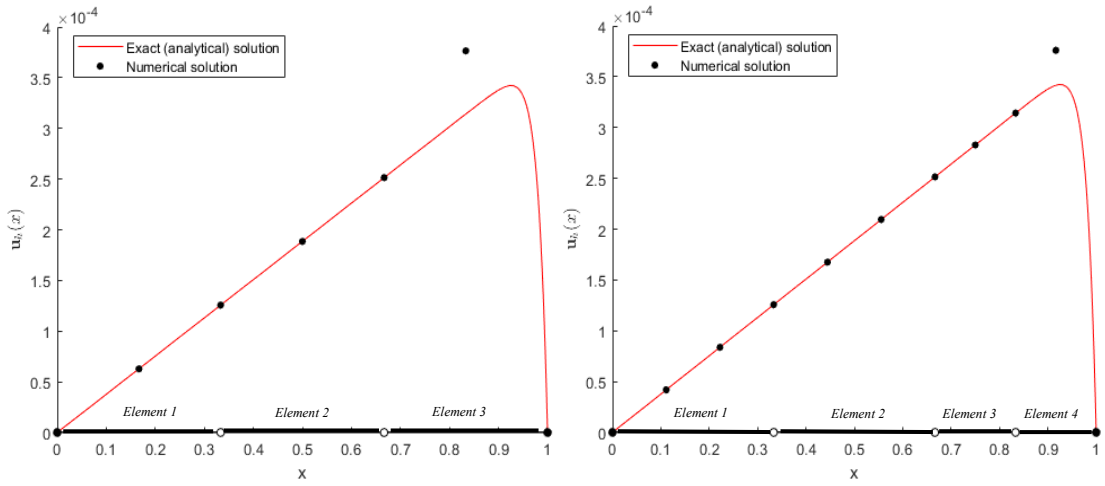


Figure 3.8: Comparison of solutions before and after $hp-$adaptivity to the one-dimensional Poisson problem $-u" = x^{50}$. (A) shows the numerical solution on a uniform mesh of three elements, each with $p = 2$ ($L_2\, error = 2.3540 \times 10^{-5}$). (B) shows the adaptive case, where the third element is split into two sub-elements with $p = 2$, while the first and second elements are enriched to $p = 3$ resulting in improved accuracy ($L_2\, error = 1.0353 \times 10^{-5}$).

3.8.2 p-Adaptivity in $1D$:

In the p-adaptive scheme if ($\sigma > 1$), the element is assumed to be in the exponential convergence regime and is therefore refined by increasing its polynomial degree. On the other hand, if ($\sigma \leq 1$), the element is retained with its current polynomial degree, as further increases in p would not significantly improve the approximation due to lack of smoothness. It should be noted that no element subdivision occurs in this strategy since $1D$ meshes are conforming by construction, and no interpolation or continuity correction is required between elements of differing polynomial degrees. Further, in $1D$ (vs $2D$) we do not incur a continuity problem by increasing the order of elements. This behavior is demonstrated for one-dimensional Poisson problem $-u" = x^{50}$ on the interval [0, 1]in the results shown in Figure 3.9.

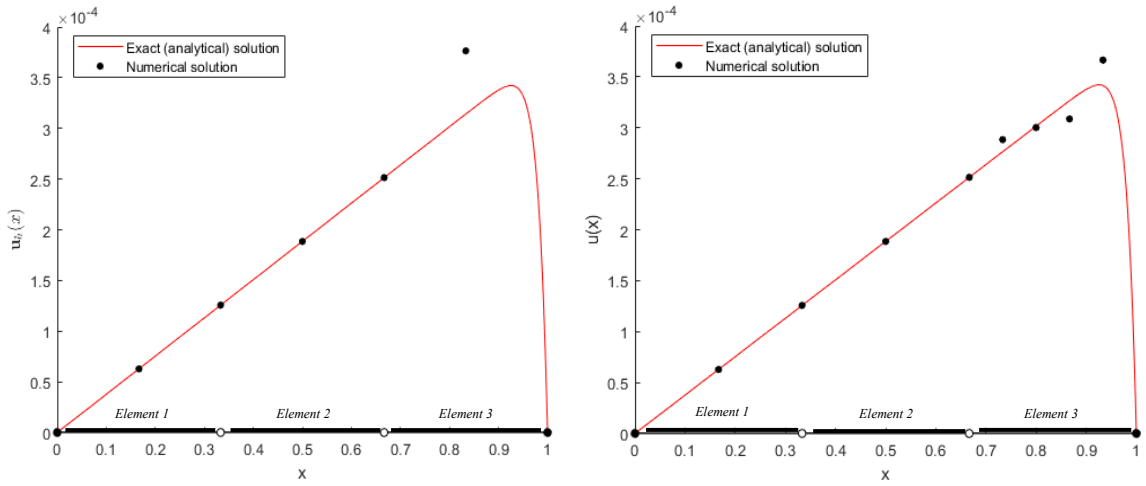


Figure 3.9: Comparison of solutions before and after p −adaptivity for the one-dimensional Poisson problem $-u" = x^{50}$. (A) shows the numerical solution on a uniform mesh of 3 elements, each with polynomial order $p = 2$ ($L_2\ error\ =\ 2.3540\ \times\ 10^{-5}$). (B) shows the result after three successive p −adaptive steps, where the third element (identified by $\sigma > 1$) has been progressively enriched up to polynomial order $p = 5$, while the first and second elements remain at $p = 2$ ($L_2\ error\ =\ 1.0429\ \times\ 10^{-5}$).

3.8.3 h −Adaptivity in $1D$:

The logic of h −adaptivity follows the same principle as p −adaptivity, elements with $\sigma \leq 1$ are interpreted as having non-smooth features or insufficient resolution. These elements are refined through bisection to better capture local behavior. In contrast, elements with $\sigma > 1$ are considered adequately resolved and are left unchanged in both size and polynomial degree. In this approach, the polynomial order remains constant throughout the mesh, and only the mesh topology is modified to improve accuracy by splitting the target element or elements into two elements. This strategy is particularly efficient in $1D$, where conforming meshes are preserved by construction and no special treatment is needed to maintain continuity between elements of differing sizes. For example, Figure 3.10 illustrates the application of this method to the one-dimensional Poisson problem, which presents a steep gradient near the right end of the domain. Through, h −adaptivity, elements identified as under-resolved based on the spectral decay indicator are selectively refined, improving the solution's accuracy in regions with sharp variations. This targeted refinement enhances resolution without increasing the polynomial degree, preserving computational efficiency.

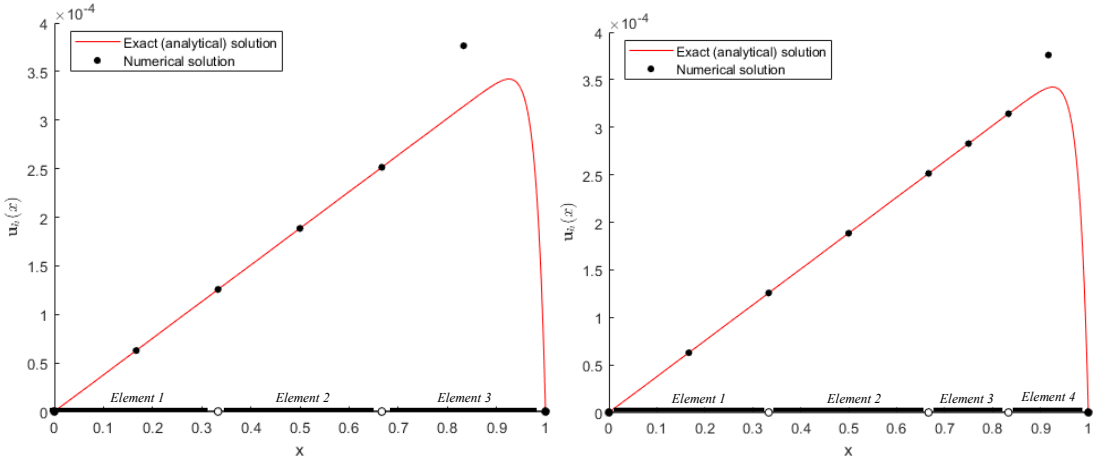


Figure 3.10: Comparison of solutions before and after h −Adaptivity for the one-dimensional Poisson problem $-u'' = x^{50}$. (A) shows the initial solution on a uniform mesh of three elements, each with polynomial degree of $p = 2$ ($L_2\ error\ =\ 2.3540\ \times\ 10^{-5}$). (B) shows the result after one successive h −adaptive step, where the third element is subdivided into two elements, both with polynomial order $p = 2$, while the first and second elements remain unchanged ($L_2\ error\ =\ 1.1445\ \times\ 10^{-5}$).

3.8.4 p −Adaptivity in $2D$:

In two-dimensional problems, p −adaptivity involves selectively increasing the polynomial degree of elements identified as under-resolved based on the behavior of the exact or reference solution. This targeted refinement improves accuracy while keeping the total number of elements unchanged. However, raising the polynomial order of isolated elements introduces a challenge: when an element of higher order shares an edge with a lower-order neighbor, the associated degrees of freedom along the common edge become inconsistent. This leads to a non-conforming mesh, where hanging nodes appear along the interfaces. These nodes present only in the higher-order element do not coincide with nodes from the lower-order side, leading to discontinuity in the solution across the shared interface.

To address this issue and ensure conformity across elements of differing polynomial degrees, an interpolation-based constraint mechanism is employed. This method offers a simple and efficient alternative to more complex techniques such as mortar methods [102][18] or Lagrange multipliers [102], making it particularly attractive for adaptive procedures that require frequent updates to the mesh or basis.

The key idea is to preserve the degrees of freedom associated with the lower-order (master) element and eliminate those introduced by the higher-order (slave) element along the interface [102]. This is accomplished by evaluating the values of the hanging nodes on the slave element using the shape functions of the adjacent master element. More precisely,

each hanging node on the shared edge is projected onto the function space of the lower-order neighbor by evaluating the master shape functions at the coordinates of the hanging node. To visualize this adaptive refinement process, Figure 3.11 illustrates a structured 2D mesh where elements of varying polynomial degrees are assigned based on local solution behavior. In regions with minimal variation (lower-right), as confirmed by the adaptive logic shown in Figure 3.13, the element order remains low and unchanged. However, moving toward the upper-left corner, where the solution exhibits higher variation, the polynomial degree of elements increases progressively. This spatially varying distribution of element orders effectively controls the approximation error while maintaining mesh conformity through interpolation constraints along shared edges between higher- and lower-order elements.

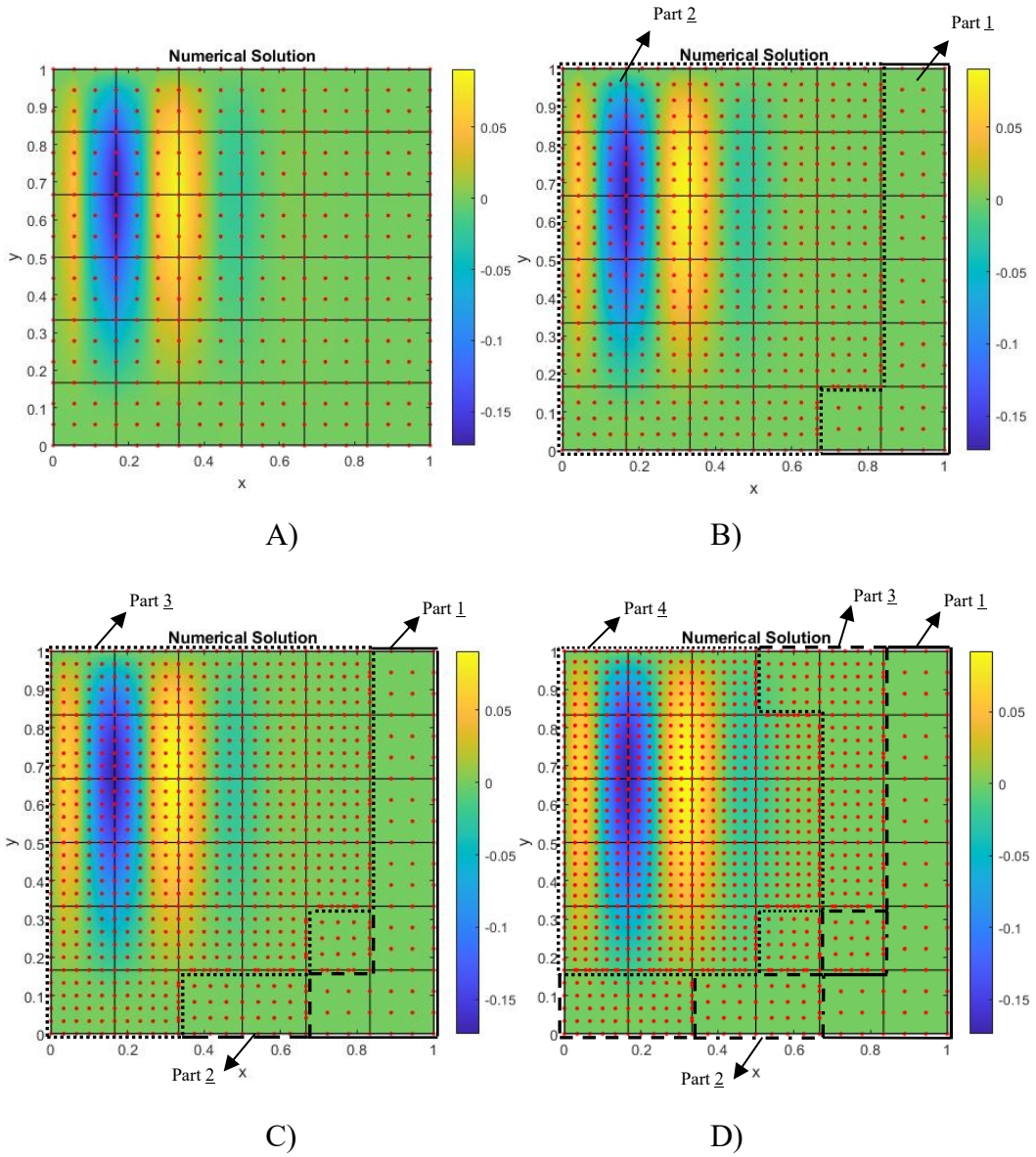


Figure 3.11: Numerical solution of the 2D Poisson equation $-\nabla^2 u(x,y) = f(x,y)$ in the domain $\Omega = [0,1]^2$, subject to Dirichlet boundary conditions and $f(x,y)$ is provided in Appendix A. (A) Initial mesh of 6×6 elements with $p = 3$ ($L_2\, error = 1.1 \times 10^{-3}$). (B) After the first adaptation part 2 refined to $p = 4$, part 1 remains at $p = 3$ ($L_2\, error = 5.43 \times 10^{-4}$). (C) After the second adaptation, three regions with $p = 3,4,5$ ($L_2\, error = 5.72 \times 10^{-5}$). (D) After four adaptations, four regions with $p = 3,4,5,6$ ($L_2\, error = 1.92 \times 10^{-5}$).

3.8.4.1 Construction of the Global Interpolation Matrix $\mathbb{R}^{(Global)}$:

Consider a pair of neighboring elements in a two-dimensional mesh. Let one element, denoted as Ω_s, have a polynomial degree p_s, and its neighbor Ω_m have a lower degree p_m, such that $p_s > p_m$. Let Γ be their shared edge (see Figure 3.12). The DOFs on Γ for Ω_s will include additional hanging nodes that are not present in Ω_m.

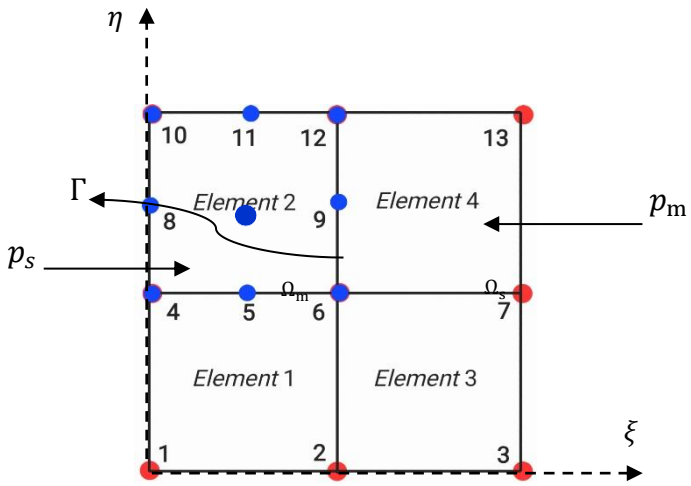


Figure 3.12: A 2D mesh with four elements, highlighting neighboring pairs that share an interface. Element 1 and Element 2 share a vertical edge, while Element 2 (Ω_m) and Element 4 (Ω_s) share a horizontal edge denoted by Γ. The polynomial orders satisfy $p_s > p_m$.

In order to create this interpolation matrix, the following steps must be followed:

1. Identify the common edge and local nodes:

   Determine the parametric coordinates $\{\xi_i\}$ of the nodes on the shared edge Γ belonging to the higher-order element Ω_s. These include both shared nodes (aligned with Ω_m) and hanging nodes (present only in Ω_s).

2. Evaluate shape functions from the master element:

   For each hanging node $\xi_i \in \Gamma$, evaluate the shape functions $\left\{\phi_j^{(m)}\right\}$ of the master element Ω_m at ξ_i. These shape functions are defined based on the polynomial degree p_m and are associated with the nodal basis on Ω_m. The interpolation condition is:

   $$u_h^{(s)}(\xi_i) = \sum_{j=1}^{N_m} \phi_j^{(m)}(\xi_i)\, u_j^{(m)} \tag{3.101}$$

   where $u_h^{(s)}(\xi_i)$ is the value at the hanging node in Ω_s, and $u_j^{(m)}$ are the DOFs of Ω_m.

3. Populate the local Interpolation matrix $\mathcal{R}$:

   Construct a matrix $\mathcal{R} \in \mathbb{R}^{N_s \times N_m}$, where N_s is the total number of DOFs (including hanging nodes) on the element edge, and N_m is the number of DOFs on the master edge.

   3.1 Each row of $\mathcal{R}$ corresponds to a DOF on the slave element.

   3.2 Each column of $\mathcal{R}$ corresponds to a DOF on the master element, serving as the retained interpolation basis on the common interface.

3.3 Rows corresponding to shared DOFs are assigned entries from the identity matrix (i.e., 1 at the diagonal position, 0 elsewhere), preserving the values of DOFs that are conforming.

3.4 Rows corresponding to hanging DOFs are filled with the evaluations of the master's shape functions at the coordinates of the hanging nodes. For a hanging node at coordinate ξ_i, the entries are:

$$R_{i,j} = \phi_j^{(m)}(\xi_i) \tag{3.102}$$

where $\phi_j^{(m)}$is the $j-$th shape function of the master element, and $j \in \{1, \dots, N_m\}$ indexes the retained DOFs.

3.5 For hanging nodes, the row entries are the evaluated shape functions $\phi_j^{(m)}(\xi_i)$.

4. Assemble the global interpolation matrix:

This local interpolation matrix $\mathcal{R}$ is assembled into the global interpolation matrix $\mathbb{R}^{(Global)}$, which spans. It maps the global reduced DOFs u_m(from all master sides) to the expanded u_s that include higher-order contributions:

$$u_s = \mathbb{R}^{(Global)} u_m \tag{3.103}$$

This transformation is then applied to the global stiffness matrix and load vector:

$$\mathrm{K}_{\text{modified}} = \mathbb{R}^{(Global)^{\mathsf{T}}} \mathrm{K}\, \mathbb{R}^{(Global)} \tag{3.104}$$

$$\mathrm{F}_{\text{modified}} = \mathbb{R}^{(Global)^{\mathsf{T}}} \mathrm{F} \tag{3.105}$$

Example:

Consider a shared edge between a linear element ($p_m = 1$) and a quadratic element ($p_s = 2$).

The linear element has two nodes located at $\xi \in [0,\ 0.5]$ and $\eta \in [0.5,\ 1]$, while the quadratic element includes an additional mid-edge (hanging) node at $(\xi, \eta) = (0.25, 0.5)$ and also $(\xi, \eta) = (0.5, 0.75)$. The nodes at $(\xi, \eta) = (0, 0.25)$, $(\xi, \eta) = (0.5, 0.5)$ and $(\xi, \eta) = (0.5, 1)$ are common to both elements. The node configuration and element arrangement are schematically illustrated in Figure 3.13.

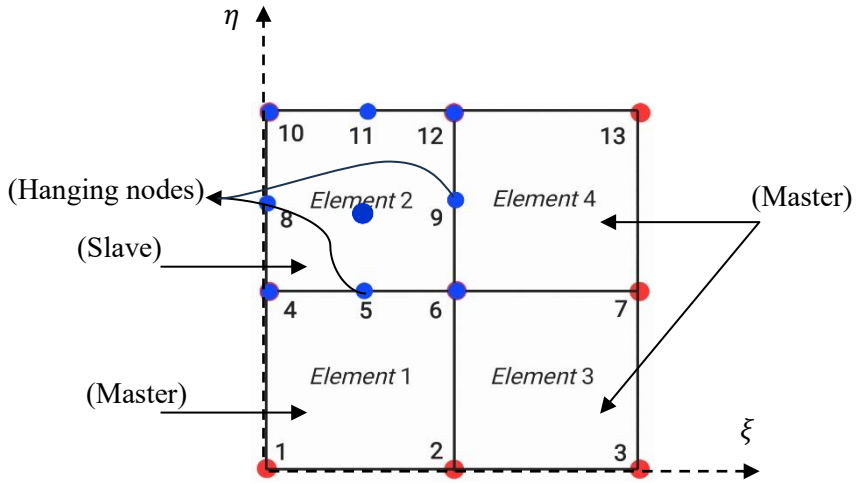


Figure 3.13: Global node numbering for the structured 2D mesh used in the test case. All elements are linear ($p = 1$), and shown with red nodes, except Element 2, which is upgraded to quadratic order ($p = 2$) and displayed using blue nodes. The blue nodes represent the additional mid-edge degrees of freedom introduced by the higher-order element. At interfaces where Element 2 connects to lower-order neighbors, overlapping red and blue nodes indicate shared locations, ensuring continuity across elements with different polynomial degrees.

To evaluate the value of this hanging node, we use the linear shape functions by using Equation (3.30):

$$\phi_4^{(e1)}(\xi,\eta) = -2\,\eta\,(2\xi - 1) \qquad (3.106)$$
$$\phi_6^{(e1)}(\xi,\eta) = 4\,\eta\,\xi$$

$$\phi_6^{(e4)}(\xi,\eta) = (2\xi - 2)(2\eta - 2) \qquad (3.107)$$
$$\phi_{12}^{(e4)}(\xi,\eta) = -(2\xi - 2)(2\eta - 1)$$

So, evaluating Node's number 5 which is located at $(\xi,\eta) = (0, 0.5)$ by using Equation 3.107:

$$\phi_4^{(e1)}(\xi,\eta) = 0.5, \quad \phi_6^{(e1)}(\xi,\eta) = 0.5$$

Similarity, evaluation Node's number 9 which is located at $(\xi,\eta) = (0.5, 0.75)$ by utilizing Equation 3.108:

$$\phi_5^{(e4)}(\xi,\eta) = 0.5, \quad \phi_{12}^{(e4)}(\xi,\eta) = 0.5$$

This gives the corresponding row in the local interpolation matrix $\mathcal{R}$:

$$\mathcal{R} = \begin{bmatrix} 1 & 0 \\ 0.5 & 0.5 \\ 0 & 1 \end{bmatrix}$$

This matrix which is of size 3×2 maps the two degrees of freedom (DOFs) of the master edge to the three DOFs of the slave edge, thereby handling the nonconforming between them. $\mathcal{R}$ operates on the DOFs associated with the shared edge of element 2. Specifically,

Row 1 corresponds to Node 4, Row 2 to Node 5, and Row 3 to Node 6 (see Figure 3.13). The columns of $\mathcal{R}$ are associated with the nodes whose shape functions are used to evaluate the values at these hanging nodes. This matrix structure can be extended similarly to other shared edges. To assemble the global system, the local interpolation matrix is extended into a global interpolation matrix $\mathbb{R}^{(\mathrm{Global})}$. $\mathbb{R}^{(\mathrm{Global})}$, which is of size$13\times 11$, has one row for each node (both hanging and non-hanging), while its columns correspond only to non-hanging nodes. The columns are ordered according to the global numbering of the retained (non-hanging) nodes. This gives the global interpolation matrix shown below:

$$\mathbb{R}^{(Global)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

The implementation of p −adaptivity in two dimensions requires systematic treatment of non-conforming interfaces that arise when neighboring elements have different polynomial orders. The procedure is summarized in the flowchart of Figure 3.14. After looping over all elements, non-conforming interfaces are identified and the master–slave relationship between adjacent elements is established. Hanging nodes on the slave side are then evaluated using the shape functions of the master element. This information is assembled into the global interpolation matrix $\mathbb{R}^{(\mathrm{Global})}$, which treats nonconformity.

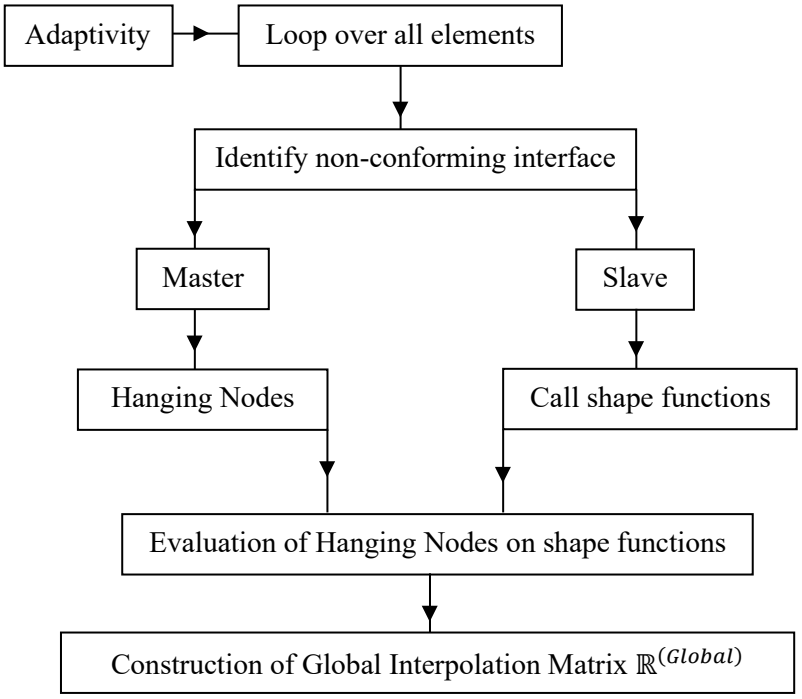


Figure 3.14: Flowchart illustrating the implementation of p −adaptivity in two dimensions. The process involves detecting non-conforming interfaces, identifying master and slave elements, evaluating shape functions at hanging nodes, and constructing the global interpolation matrix $\mathbb{R}^{(\mathrm{Global})}$.

3.9 Solver

The linear system arising from the finite element formulation is given by Equation (3.54), where K ∈ $R^{n\times n}$ is symmetric and positive-definite (SPD). Thus, we solve it using the Conjugate Gradient (CG) method. For notational simplicity, we express the system using the same form as Equation (1.3), corresponding to KU = F. The CG method interprets the problem as the minimization of a quadratic functional:

$$f(x) = \frac{1}{2}x^{\top}Ax - b^{\top}x + c \tag{3.108}$$

where the matrix $A \in R^{n\times n}$ is symmetric positive-definite and $x, b \in R^{n}$. The minimum of f occurs when the gradient vanishes.

$$\nabla f(x) = Ax - b = 0 \tag{3.109}$$

Hence, the minimum of $f(x)$ corresponds to the solution of Equation 3.109.

The CG algorithm iteratively updates the solution by combining the current residual direction with the previous search direction. The iterative scheme is defined as:

$$x^{(k+1)} = x^{(k)} + \alpha_k \mathrm{p}^{(k)} \tag{3.110}$$

where $x^{(k)}$ is the current approximation, $\mathrm{p}^{(k)}$ is the search direction, and α_k is the step size chosen to minimize $f(x)$ along $\mathrm{p}^{(k)}$, calculated as:

$$\alpha_{\mathrm{k}} = \frac{\mathrm{r}_{\mathrm{k}}^{\mathrm{T}}\mathrm{r}_{\mathrm{k}}}{\mathrm{p}_{\mathrm{k}}^{\mathrm{T}}\mathrm{A}\mathrm{p}_{\mathrm{k}}} \tag{3.111}$$

where $\mathrm{r}_{\mathrm{k}} = \mathrm{b} - \mathrm{A}\mathrm{x}_{\mathrm{k}}$ is the residual. After updating $\mathrm{x}_{\mathrm{k}+1}$, the new residual is computed:

$$\mathrm{r}_{\mathrm{k}+1} = \mathrm{r}_{\mathrm{k}} - \alpha_{\mathrm{k}}\mathrm{A}\mathrm{p}_{\mathrm{k}} \tag{3.112}$$

Then, the new direction is obtained by combining the current residual with the previous direction:

$$\beta_{\mathrm{k}} = \frac{\mathrm{r}_{\mathrm{k}+1}^{\mathrm{T}}\mathrm{r}_{\mathrm{k}+1}}{\mathrm{r}_{\mathrm{k}}^{\mathrm{T}}\mathrm{r}_{\mathrm{k}}} \tag{3.113}$$

$$\mathrm{p}_{\mathrm{k}+1} = \mathrm{r}_{\mathrm{k}+1} + \beta_{\mathrm{k}}\mathrm{p}_{\mathrm{k}} \tag{3.114}$$

The algorithm continues until the residual norm $\left|\left|\mathrm{r}^{(\mathrm{k}+1)}\right|\right|$ falls below a specified tolerance. For a more detailed derivation and theoretical background, the reader is referred to [41] and [103].

Algorithm 3.1: The Conjugate Gradient Method
1) Input: Matrix A, right hand side b. Initial guess $\mathrm{x}^{(0)}$ **2) Output:** Approximate solution x to $Ax = b$
3) Initialize the residual and search direction: $\mathrm{r}^{(0)} = \mathrm{b} - \mathrm{A}\mathrm{x}^{(0)}, \quad \mathrm{p}^{(0)} = \mathrm{r}^{(0)}.$
4) For $\mathrm{k} = 0, 1, 2, \ldots$ until convergence: $\alpha_k = \frac{\langle \mathrm{r}^{(\mathrm{k})}, \mathrm{r}^{(\mathrm{k})} \rangle}{\langle \mathrm{A}\mathrm{p}^{(\mathrm{k})}, \mathrm{p}^{(\mathrm{k})} \rangle},$ $\mathrm{x}^{(\mathrm{k}+1)} = \mathrm{x}^{(\mathrm{k})} + \alpha_{\mathrm{k}}\mathrm{p}^{(\mathrm{k})},$ $\mathrm{r}^{(\mathrm{k}+1)} = \mathrm{r}^{(\mathrm{k})} - \alpha_{\mathrm{k}}\mathrm{A}\mathrm{p}^{(\mathrm{k})},$ If $\left|\left|\mathrm{r}^{(\mathrm{k}+1)}\right|\right|$ is below a desired tolerance, terminate. $\beta_k = \frac{\langle r^{(k+1)}, r^{(k+1)} \rangle}{\langle r^{(k)}, r^{(k)} \rangle},$ $\mathrm{p}^{(k+1)} = r^{(k+1)} + \beta_k \mathrm{p}^{(k)}.$

3.9.1 CG Convergence Behavior

The convergence of the CG method for solving a linear system given in Equation (1.3) is closely tied to the spectral properties of A. In particular, a commonly used upper bound for the error in the A-norm at iteration K is given by [41]:

$$\left|\left|e_k\right|\right|_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^i \left|\left|e_0\right|\right|_A \tag{3.115}$$

where $\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$ is the condition number of A (see Section 1.3) and $e_k = x_k - x^*$ is the error after i iterations. This inequality shows that the number of iterations required to reduce the error by a given factor depends logarithmically on the condition number. For instance, reducing the relative error by one order of magnitude:

$$\frac{\left|\left|e_k\right|\right|_A}{\left|\left|e_0\right|\right|_A} \leq \frac{1}{10}$$

requires approximately:

$$i \geq \frac{\log(1/10)}{\log\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)}$$

For a matrix with $k = 1000$, this yields $i \approx 37$ iterations.

In practice, the Conjugate Gradient method often converges faster than expressed in Equations (3.115), primarily due to favorable eigenvalue distributions of the system matrix or well-chosen initial guesses.

3.9.2 Sensitivity to Eigenvalue Distribution in CG Convergence

To better understand the behavior of the Conjugate Gradient (CG) method, consider the error at iteration k, denoted by $e_k = x_k - x^*$, where x^* is the exact solution to Equation (1.3). Since A is (SPD), it has an orthonormal eigenbasis:

$$A = Q\Lambda Q^T \tag{3.116}$$

In Equation (3.116) $Q \in R^{n\times n}$ is an orthogonal matrix of eigenvectors ($Q^T Q = I$), and $\Lambda = diag(\lambda_1, \ldots, \lambda_n)$ contains the eigenvalues of A. In CG, the error after k iterations can be written as the action of a polynomial $p_k(\cdot)$ on the initial error:

$$e_k = p_k(A)e_0 \tag{3.117}$$

By substituting in Equation (3.117), we get:

$$e_k = p_k(A)e_0 = Q\, p_k(\Lambda)\, Q^T e_0 \tag{3.118}$$

which shows that the polynomial is applied directly to the eigenvalues of A. This follows from Equation (3.116) as spectral decomposition. Since A is orthogonal, applying a polynomial to A yields:

$$p_k(A) = Q\, p_k(\Lambda)\, Q^T \tag{3.119}$$

which makes it clear that the polynomial acts only on the eigenvalues contained in Λ. Thus, the convergence of CG depends on how small the polynomial $p_k(\lambda)$ can be made on the spectrum $\{\lambda_1, \ldots, \lambda_n\}$, subject to the constraint $p_k(0) = 1$. The best such polynomial is often expressed in terms of Chebyshev polynomials, which minimize the maximum error over a given interval. If the eigenvalues are clustered, it is easier to find a polynomial that

remains small over the whole spectrum, resulting in faster convergence. Conversely, if the eigenvalues are spread non-uniformly, or contain outliers, then the polynomial must oscillate significantly to remain small over all eigenvalues, leading to slower convergence.

Although two matrices may have the same condition number, their Conjugate Gradient (CG) convergence behavior can differ significantly depending on the distribution of their eigenvalues. For a more detailed theoretical discussion on this topic can be found in [41].

3.9.3 Preconditioned Conjugate Gradient (PCG)

As discussed in Section 1.8.1 preconditioning is employed to accelerate the convergence of iterative solvers by improving the conditioning of the linear system. Here, M denotes the preconditioner, a matrix chosen to approximate A in a way that is inexpensive to invert. In practice, it is M^{-1} that is applied to both sides of the system, leading to the preconditioned form:

$$M^{-1}Ax = M^{-1}b \tag{3.120}$$

which has the same solution x as the original system but typically a much more favorable eigenvalue distribution. Since M is assumed to be nonsingular, the solution x to the preconditioned system is also a solution to the original system. To derive a usable form for iterative implementation, we define the residual vector at iteration K as:

$$r^{(k)} = b - Ax^{(k)} \tag{3.121}$$

multiplying both sides of the residual equation by M^{-1}, we define:

$$z^{(k)} \coloneqq M^{-1}r^{(k)} \;\Rightarrow\; Mz^{(k)} = r^{(k)} \tag{3.122}$$

This vector $z^{(k)}$ is introduced to avoid direct use of M^{-1}, and instead rely on solving a linear system involving M, which is chosen to be easier to invert or apply than A. To solve Equation (3.122) the CG method is used to solve this system efficiently, offering faster convergence compared to other iterative methods.

The rest of the CG algorithm can be reformulated in terms of these modified vectors. The preconditioned version of the CG algorithm proceeds similarly, with the search directions $\mathrm{p}^{(\mathrm{k})}$ now updated using $z^{(k)}$instead of $r^{(k)}$, and the scalar coefficient α_k and β_k adjusted accordingly.

Algorithm 3.2: Preconditioned Conjugate Gradient (PCG) Method

1) Input: Matrix A, right hand side b. Initial guess $x^{(0)}$, preconditioner M, tolerances ε

2) Output: Approximate solution x to $Ax = b$

3) Compute:

$$r^{(0)} = b - Ax^{(0)}.$$

4) Solve iteratively (CG):

$$MZ^{(0)} = r^{(0)}.$$

5) Set:

$$p^{(0)} = r^{(0)}$$

6) For $k = 0, 1, 2, \ldots$ until convergence:

$$\alpha_k = \frac{\langle r^{(k)}, r^{(k)} \rangle}{\langle Ap^{(k)}, p^{(k)} \rangle},$$

$$x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)},$$

$$r^{(k+1)} = r^{(k)} - \alpha_k Ap^{(k)},$$

If $\left\| r^{(k+1)} \right\|$ is below a desired tolerance ε, terminate.

$$\beta_k = \frac{\langle r^{(k+1)}, r^{(k+1)} \rangle}{\langle r^{(k)}, r^{(k)} \rangle},$$

$$p^{(k+1)} = r^{(k+1)} + \beta_k p^{(k)}.$$

3.9.3.1 Nested Preconditioning in PCG

According to Algorithm 3.2 we need to solve $Mz = r$, since M is assumed to be symmetric and positive-definite (SPD), the inner system is well-suited for solutions via the CG method. However, the same challenges that affect the original system may also impact on the inner problem. Therefore, a second-level preconditioner M_1, typically simpler and cheaper (such as the Jacobi preconditioner built from the diagonal of M), is applied within the inner CG iterations (see Figure 3.14). This yields a nested preconditioning structure:

- The outer CG iteration solves $Ax = b$, with preconditioner M.
- The inner CG iteration approximately solves $Mz = r$, using M_1 as its preconditioner.

This nested strategy allows the use of sophisticated outer preconditioners without incurring excessive cost, as the inner solve can be accelerated using lightweight techniques. For example, if M is not diagonal (and thus not trivially invertible), but we apply Jacobi preconditioning to it $\left(\text{i.e., } M_1 = \text{diag}(M)\right)$, then the nested solver retains efficiency while improving convergence behavior. Importantly, this method preserves the symmetrical and

positive definiteness (SPD) required by the CG method, both at the outer and inner levels, ensuring the theoretical guarantees of convergence remain valid.

Algorithm 3.3: Preconditioned Conjugate Gradient with Nested Preconditioning (MPCG)

1) Input: Matrix A, right hand side b. Initial guess $x^{(0)}$, preconditioner M and M_1 and tolerances ε for outer loop, ε_{inner} for inner solve

Output: Approximate solution x to $Ax = b$

2) Compute:

$$r^{(0)} = b - Ax^{(0)}.$$

3) Solve $Mz^{(0)} = r^{(0)}$, approximately using CG (inner loop), with preconditioner M_1, Tolerance ε_{inner}

4) Set:

$$p^{(0)} = z^{(0)}$$

5) For $k = 0, 1, 2, \ldots$ until convergence $\left||r^{(k)}|\right| \leq \varepsilon||b||$:

$$\alpha_k = \frac{\langle z^{(k)}, r^{(k)} \rangle}{\langle Ap^{(k)}, p^{(k)} \rangle},$$

$$x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)},$$

$$r^{(k+1)} = r^{(k)} - \alpha_k Ap^{(k)},$$

Solve $Mz^{(k+1)} = r^{(k)}$, approximately using CG with preconditioner M_1.

If $\left||r^{(k+1)}|\right|$ is below a desired tolerance, terminate.

$$\beta_k = \frac{\langle z^{(k+1)}, r^{(k+1)} \rangle}{\langle z^{(k)}, r^{(k)} \rangle},$$

$$p^{(k+1)} = z^{(k+1)} + \beta_k p^{(k)}.$$

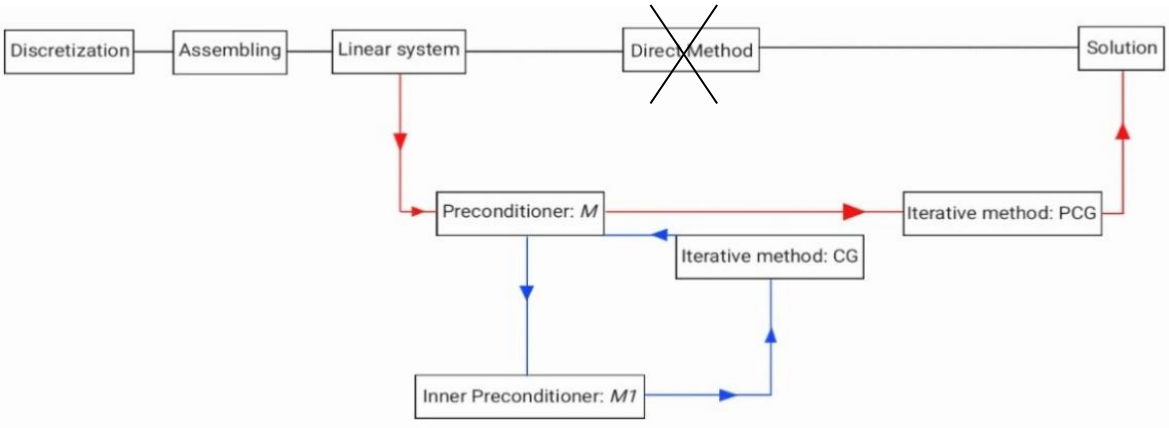


Figure 3.15: Illustration of the nested preconditioning approach. The outer solver handles the main system, while each preconditioning step uses an inner iterative solver.

3.9.4 Multi-Preconditioned Conjugate Gradient (MPCG) Method

As introduced in Section 1.8.2, the following provides the mathematical formulation of the method. The core of idea of MPCG is to extend the search space at each iteration by incorporating multiple preconditioned residuals. Starting with an initial residual $r_0 = b - Ax_0$, we compute the first set of search directions as:

$$p_1^j = M_j^{-1} r_0, \quad j = 1, \dots, k \tag{3.123}$$

By taking inspiration from Equation (3.111), we can rewrite Equation (3.112) as follows:

$$p_1^j \coloneqq M_j^{-1} r_0 \;\Rightarrow\; M_j p_1^j = r_0 \tag{3.124}$$

Equation (3.123) is combined to form the search direction matrix:

$$P_1 = [p_1^1, p_1^2, \dots, p_1^k] \tag{3.125}$$

To compute P_1, in Equation (3.125), we solve the system iteratively using the CG method. The first step then proceeds similarly to block the CG method:

$$\alpha_1 = (P_1^T A P_1)^{-1} (P_1^T r_0) \tag{3.126}$$
$$x_1 = x_0 + P_1 \alpha_1$$
$$r_1 = r_0 - A P_1 \alpha_1$$

Subsequent iterations extend the search space by computing new preconditioned residuals $z_{i+1}^j = M_j^{-1} r_i$ and forming:

$$Z_{i+1} = \left[z_{i+1}^1, z_{i+1}^2, \dots, z_{i+1}^k\right] \tag{3.127}$$

To maintain A-conjugacy, we orthogonalize Z_{i+1} against all previous directions using a projection:

$$P_{i+1} = Z_{i+1} - \sum_{j=1}^{i} P_j \left(P_j^T A P_j\right)^{-1} P_j^T A \, Z_{i+1} \tag{3.128}$$

With this new direction matrix P_{i+1}, the update step proceeds as:

$$\alpha_{i+1} = \left(P_{i+1}^T A P_{i+1}\right)^{-1} \left(P_{i+1}^T r_i\right) \tag{3.129}$$
$$x_{i+1} = x_i + P_{i+1} \alpha_{i+1}$$
$$r_{i+1} = r_i - A P_{i+1} \alpha_{i+1}$$

These steps are repeated until convergence.

A detailed derivation of the Multi-Preconditioned Conjugate Gradient (MPCG) method is beyond the scope of this thesis. For a comprehensive theoretical discussion and formulation, the reader is referred to [54]. However, the algorithmic structure of the MPCG method is presented below for completeness.

Algorithm 3.4: Multi-Preconditioned Conjugate Gradient (MPCG)

1) **Input:** Matrix A, right hand side b. Initial guess $x^{(0)}$, a set of symmetric positive-definite preconditioners $\{M_j\}_{j=1}^{k}$, tolerances $\varepsilon\ for$ outer loop and ε_{inner} for inner loop.
2) **Output:** Approximate solution x to $Ax = b$

3) Compute initial residual:

$$r^{(0)} = b - Ax^{(0)}.$$

4) For $j = 1, \dots, k$:

Solve

$$p_1^j = M_j^{-1} r^{(0)} \quad for\, j = 1, \dots, k$$

using CG method (inner loop) and tolerance ε_{inner}.

5) Initial search matrix:

$$P_1 = [\, p_1^1 \mid p_1^2 \mid \dots \mid p_1^k \,]$$

6) Set:

$$\alpha_1 = (P_1^T A P_1)^{-1}\left(P_1^T r^{(0)}\right), \qquad x^{(1)} = x^{(0)} + P_1\alpha_1, \qquad r^{(1)} = r^{(0)} - AP_1\alpha_1$$

7) For $i = 0, 1, 2, \dots$ until convergence $\left|\left|\mathbf{r}^{(i)}\right|\right| \leq \varepsilon||\mathrm{b}||$:

a) Apply preconditioners to new residual:

$$z_{i+1}^j = M_j^{-1} r^{(i)} \quad \text{for } j = 1, \dots, k$$

$$Z_{i+1} = \left[\, z_{i+1}^1 \mid z_{i+1}^2 \mid \dots \mid z_{i+1}^k \,\right]$$

b) Orthogonalize to maintain A-conjugacy:

$$P_{i+1} = Z_{i+1} - \sum_{j=1}^{i} P_j \left(P_j^T A P_j\right)^{-1} P_j^T A\, Z_{i+1}$$

c) Local Update:

$$\alpha_{i+1} = (P_{i+1}^T A P_{i+1})^{-1}\left(P_{i+1}^T r^{(i)}\right), \qquad x^{(i+1)} = x^{(i)} + P_{i+1}\alpha_{i+1}, \qquad r^{(i+1)} = r^{(i)} - AP_{i+1}\alpha_{i+1}$$

3.9.4.1 Nested Preconditioning in MPCG:

As discussed in Section 3.9.3.1, solving $Mz = r$ in each preconditioning step can itself be accelerated by applying a second-level preconditioner M_1within an inner CG solve. In the MPCG context, this idea is applied independently for each outer preconditioner $M_j = 1, \dots, k$. The procedure is:

- Outer loop: Solve $Ax = b$ using MPCG with preconditioners $\{M_j\}$.
- Inner loop: Whenever $M_j z = r$ is required, solve it approximately using PCG with preconditioner M_1 (often chosen as the Jacobi preconditioner based on the diagonal of M_j) and tolerance ε_{inner}.

Algorithm 3.5: Multi-Preconditioned Conjugate Gradient (MPCG) with Nested Preconditioning

1) **Input:** Matrix A, right hand side b. Initial guess $\mathrm{x}^{(0)}$, a set of symmetric positive-definite preconditioners $\{M_j\}_{j=1}^{k}$, tolerances $\varepsilon\ for$ outer loop and ε_{inner} for inner loop.
2) **Output:** Approximate solution x to $Ax = b$

3) Compute initial residual:

$$\mathrm{r}^{(0)} = \mathrm{b} - \mathrm{A}\mathrm{x}^{(0)}.$$

4) For $j = 1, \dots, k$:

Solve

$$p_1^j = M_j^{-1} r^{(0)} \quad for j = 1, \dots, k$$

using PCG (inner loop) with Jacobi preconditioner $M_1 = \mathrm{diag}(M_j)$ and tolerance ε_{inner}.

5) Form initial search matrix:

$$P_1 = [\, p_1^1 \mid p_1^2 \mid \dots \mid p_1^k \,]$$

6) Set:

$$\alpha_1 = (P_1^T A P_1)^{-1}(P_1^T r^{(0)}), \qquad x^{(1)} = x^{(0)} + P_1\alpha_1, \qquad r^{(1)} = r^{(0)} - AP_1\alpha_1$$

7) For $i = 0, 1, 2, \dots$ until convergence $\left\|\mathrm{r}^{(i)}\right\| \le \varepsilon \|\mathrm{b}\|$:

d) Apply preconditioners to new residual:

For $j = 1, \dots, k$:

$$z_{i+1}^j = M_j^{-1} r^{(i)}$$

using PCG with Jacobi preconditioner$M_1 = \mathrm{diag}(M_j)$ and tolerance ε_{inner}.

$$Z_{i+1} = \left[\, z_{i+1}^1 \mid z_{i+1}^2 \mid \dots \mid z_{i+1}^k \,\right]$$

e) Orthogonalize to maintain A-conjugacy:

$$P_{i+1} = Z_{i+1} - \sum_{j=1}^{i} P_j \left(P_j^T A P_j\right)^{-1} P_j^T A\, Z_{i+1}$$

f) Local Update:

$$\alpha_{i+1} = (P_{i+1}^T A P_{i+1})^{-1}(P_{i+1}^T r^{(i)}), \qquad x^{(i+1)} = x^{(i)} + P_{i+1}\alpha_{i+1}, \qquad r^{(i+1)} = r^{(i)} - AP_{i+1}\alpha_{i+1}$$

3.9.5 SSOR Iterative Method

The Symmetric Successive Over-Relaxation (SSOR) method is an iterative technique for solving symmetric positive-definite (SPD) linear systems of the form equation (1.3) and can be regarded as a symmetric extension of the classical Successive Over-Relaxation (SOR) method. The starting point is the decomposition of the coefficient matrix A into its diagonal, strictly lower, and strictly upper triangular parts:

$$A = D + L + U, \qquad p_1^j = M_j^{-1} r_0, \quad j = 1, \dots, k \tag{3.130}$$

where $D = \mathrm{diag}(a_{11}, a_{22}, \ldots, a_{nn})$, contains the diagonal entries of A, L is strictly lower triangular, and $U = L^{\mathrm{T}}$, is strictly upper triangular by symmetry. The SSOR scheme consists of two successive SOR sweeps a forward sweep followed immediately by a backward sweep applied within a single iteration.

In the forward sweep, each component of X is updated sequentially from the first equation to the last, using the most recently computed values for already-updated components and the old values for the rest. The update formula for the i −th component is:

$$x_i^{\left(k+\frac{1}{2}\right)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{\left(k+\frac{1}{2}\right)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right) \tag{3.131}$$

where $0 < \omega < 2$, is the relaxation parameter. This forward SOR step reduces high-frequency error components but leaves the error distribution asymmetric.

To restore symmetry, the backward sweep proceeds in reverse order from the last equation to the first again using the most recently updated values where possible:

$$x_i^{(k+1)} = (1-\omega)x_i^{\left(k+\frac{1}{2}\right)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{\left(k+\frac{1}{2}\right)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k+1)}\right) \tag{3.132}$$

These two sweeps can also be expressed in matrix form, making the symmetry of the method explicit. The forward sweep is written as:

$$x^{\left(k+\frac{1}{2}\right)} = (D + \omega L)^{-1}\left[\omega b - (\omega U + (\omega - 1)D)x^{(k)}\right] \tag{3.133}$$

and the backward sweep as:

$$x^{(k+1)} = (D + \omega U)^{-1}\left[\omega b - (\omega L + (\omega - 1)D)x^{\left(k+\frac{1}{2}\right)}\right] \tag{3.134}$$

Together, these steps constitute one SSOR iteration. The iteration continues until a convergence criterion is met, typically when the relative residual norm:

$$\frac{||b - Ax^{(k)}||}{||b - Ax^{(0)}||} < \varepsilon \tag{3.135}$$

falls below a prescribed tolerance ε.

Because SSOR applies both forward and backward sweeps in each iteration, it symmetrically damps oscillatory components of the error from both ends of the system, making it particularly effective in applications such as multigrid smoothing, where reduction of high-frequency error is essential. In this study, the SSOR iterative method will be employed within the smoothing stages of the p −multigrid preconditioner described in Section 3.11.4, ensuring efficient attenuation of high-frequency modes at the fine level.

3.10 Preconditioners Used

The stiffness matrix A, used in the following preconditioning tests, corresponds to the system matrix assembled for the finite element solution shown in Figure 3.11(C) of section 3.9, which is part of the 2D Poisson problem with adaptive mesh refinement. To ensure a fair and consistent comparison across all preconditioners, the complete assembled matrix was used in full form (including zero and non-zero entries), so each method operated on exactly the same system structure when solving the iterative relation $Mz = r$. In this thesis, four different preconditioners are implemented and tested within the CG, PCG, and MPCG frameworks:

3.10.1 Jacobi (Diagonal) Preconditioner

It is the simplest preconditioner, obtained by extracting only the diagonal entries of A, to form the preconditioning matrix.

$$M_{\text{Jacobi}} = \text{diag}(A) = \text{diag}(a_{11}, a_{22}, \dots, a_{nn}) \tag{3.136}$$

Its inverse is trivial to compute, as

$$M_{\text{Jacobi}}^{-1} = \text{diag}(A)^{-1} \tag{3.137}$$

which simply corresponds to taking the reciprocal of each diagonal entry $\left(\frac{1}{a_{11}}, \frac{1}{a_{22}}, \dots, \frac{1}{a_{nn}}\right)$.

However, in this thesis, as discussed in Section 3.10.3, all preconditioners are applied by iteratively solving $Mz = r$, ensuring a uniform implementation framework. Jacobi is computationally inexpensive, requires minimal storage, and is also used as the inner preconditioner in the nested PCG and MPCG configurations. To illustrate the structure of each preconditioner, we also show the sparsity patterns of the original matrix A, alongside those of the preconditioning matrix M.

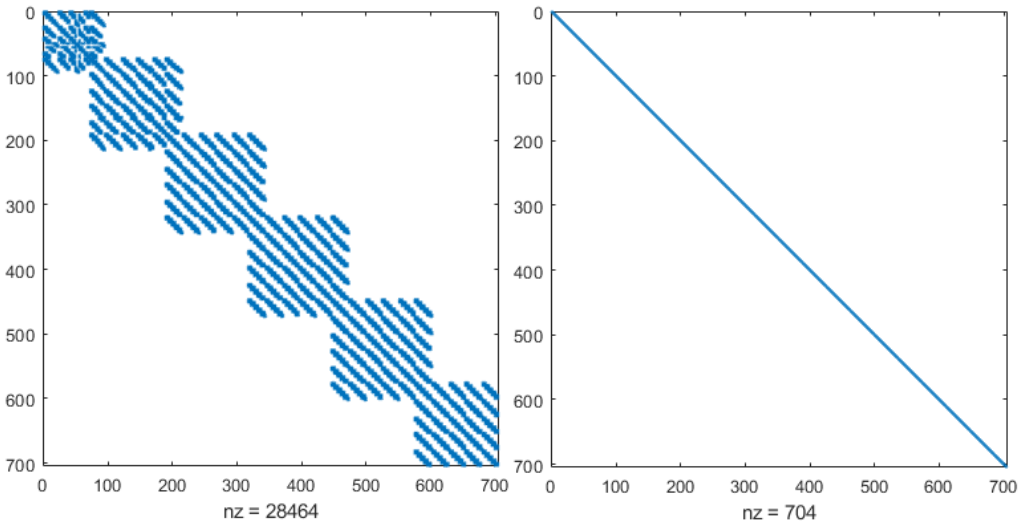


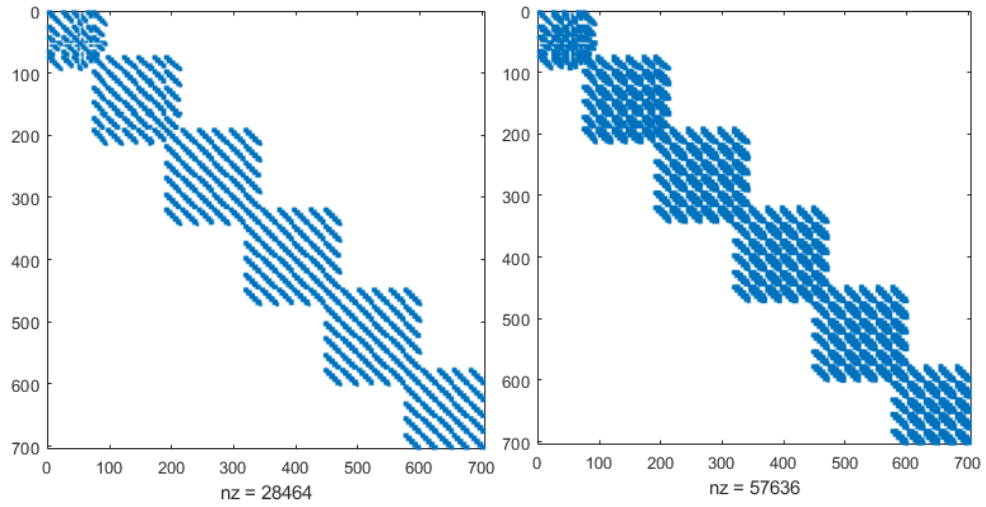
Figure 3.16: Sparsity patterns of (A) the stiffness matrix A (nz=28464) from the adaptive mesh in Figure 3.11(C), and (B) its Jacobi preconditioner M_{Jacobi} (nz=704). In contrast to A, the Jacobi preconditioner retains only diagonal entries, producing an extremely sparse matrix with far fewer nonzero elements.

3.10.2 Symmetric Successive Over-Relaxation (SSOR) preconditioner

The SSOR preconditioner is

$$M_{\text{SSOR}} = \frac{\omega}{2-\omega}(D+\omega L)D^{-1}(D+\omega U) \tag{3.138}$$

where the forward sweep $(D+\omega L)$ and backward sweep $(D+\omega U)$ are applied sequentially and the component of Equation (3.138) has been explained in section 3.9.5. In Equation (3.138) the scaling factor $\frac{\omega}{2-\omega}$ ensures symmetry. In this work, $\omega = 1$ is chosen, reducing SSOR to the Symmetric Gauss–Seidel preconditioner, which offers robust convergence without parameter tuning. A key advantage of SSOR is that it operates directly on the original matrix A without constructing an explicit preconditioner matrix, requiring only access to its diagonal and triangular parts. This preserves sparsity, avoids extra storage, and keeps setup costs low. To illustrate its structure, Figure 3.16 shows the sparsity pattern of the stiffness matrix A (taken from Figure 3.11(C)) alongside the effective sparsity pattern of the SSOR preconditioner.



A) B)

Figure 3.17: Sparsity patterns for (A) the stiffness matrix A (nz = 28,464) corresponding to the mesh in Figure 3.11(C), and (B) the SSOR preconditioning matrix (nz = 57,636). The SSOR preconditioner increases the number of nonzero entries compared to A, reflecting the additional fill-in introduced by the forward and backward sweeps.

3.10.3 Finite Element Order-1 (FEM1) Preconditioner

In this work, three variants of the Finite Element Order-1 (FEM1) preconditioner were implemented in 2D:

I. **F4R**−First-order formulation on four nearest neighboring nodes forming a rectangle (R), corresponding to a standard 4-node square element on the fine mesh.

II. **F3T**−First-order formulation on three nodes (T) forming a triangular element.

III. **F4CR**−First-order formulation on four nodes forming a rectangle (R) at a coarser scale (C), corresponding to the corners of each coarse element.

3.10.3.1 F4R

In the first variant of the FEM1 preconditioner (F4R), the complete set of degrees of freedom

$$\mathcal{N} = \{n_1, n_2, \dots, n_N\} \tag{3.139}$$

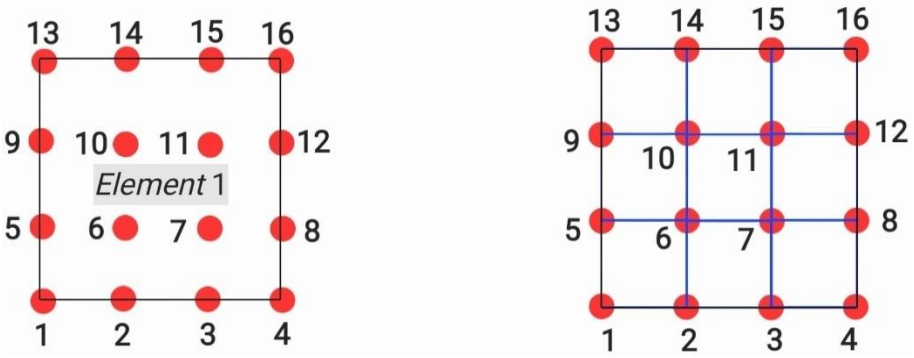
is partitioned into subsets of four consecutive nodes

$$\mathcal{E}_k = \{n_{i_1}, n_{i_2}, n_{i_3}, n_{i_4}\} \subset \mathcal{N} \tag{3.140}$$

Each subset represents the vertices of a first-order, four-node square element. For each $\mathcal{E}_k$, the standard bilinear shape functions are employed to derive the corresponding local stiffness matrix $K^{(e)}$ as discussed in Section 3.6.1. The global preconditioner matrix K_{FEM1}

is then obtained by assembling all such local matrices over the domain, following the procedure described in section 3.6.1 by utilizing Equation (3.55).

This construction yields a matrix with the structural characteristics of a first-order FEM stiffness matrix, fully compatible with the original problem size, and suitable for direct use as a preconditioner in the iterative solver. The generation of the F4R variant from a higher-order element is illustrated in Figure 3.17. Starting from a third-order element (A), appropriate groups of four nodes are identified to form first-order four-node square elements (B), which are then assembled into the global FEM1 preconditioner matrix (M_{F4R}) using the method outlined in Section 3.6.



A) B)

Figure 3.18: Construction of the F4R variant for the FEM1 preconditioner. (A) node numbering of a single third-order quadrilateral element. (B) re-interpretation of the same element as a set of first-order four-node square elements for preconditioner assembly.

3.10.3.2 F3T

In the second variant of the FEM1 preconditioner (F3T), the complete set of degrees of freedom noted in Equation (3.21) is partitioned into subsets of three consecutive nodes

$$\mathcal{E}_k = \{n_{i_1}, n_{i_2}, n_{i_3}\} \subset \mathcal{N} \tag{3.141}$$

Each forming a first-order triangular element with node coordinates (x_i, y_i), $i = 1,2,3$, the linear shape functions are:

$$\phi_i(x, y) = \frac{a_i + b_i x + c_i y}{2\tilde{A}}, \quad i = 1,2,3 \tag{3.142}$$

where the area $\tilde{A}$ and coefficients are determined purely by geometry:

$$\tilde{A} = \frac{1}{2} det \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} \tag{3.143}$$

$$b_i = y_j - y_k, \quad c_i = x_k - x_j, \qquad (i, j, k) \text{ cyclic in } \{1,2,3\}. \tag{3.144}$$

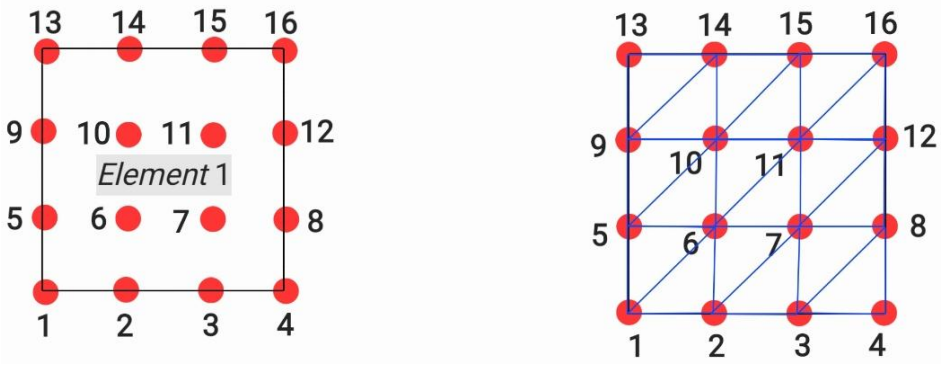
Because ϕ_i, are linear, their derivatives are constant on the element, and the gradients of Equation (3.142) are constant over the element:

$$\nabla\varphi_i = \frac{1}{2\tilde{A}_e}(b_i c_i), \quad i = 1,2,3 \tag{3.145}$$

The local stiffness matrix for the Poisson operator is:

$$K_{ij}^{(e)} = \frac{b_i b_j + c_i c_j}{4\tilde{A}_e}, \quad i,j = 1,2,3 \tag{3.146}$$

The global FEM1 preconditioner matrix (M_{F3R}) is obtained by assembling all such triangular local matrices across the domain using the procedure in Section 3.6 (Assembly of Global Matrices), resulting in a first-order triangular stiffness structure compatible with the original problem size and directly usable as a preconditioner in the iterative solver. An example of how the F3T variant is constructed is shown in Figure 3.18. In this example, a third-order element (A) is reinterpreted by selecting groups of three nodes, each forming a first-order triangular element.



A) B)

Figure 3.19: Example of the F3T (first-order, three-node triangle) variant used in the FEM1 preconditioner. (A) shows the node numbering for a third-order quadrilateral element, while (B) illustrates its subdivision into multiple first-order triangular elements for use in the preconditioner assembly.

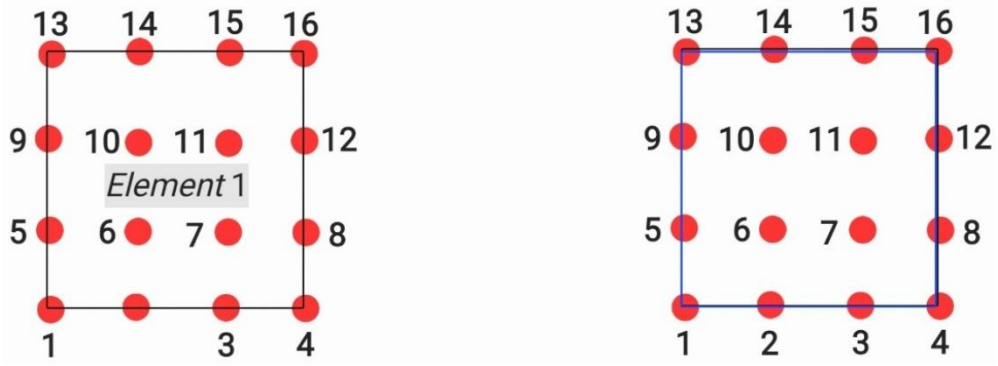
3.10.3.3 F4CR

The F4CR variant follows the same construction principles described for the F4R preconditioner (see Section 3.10.3.1, F4R Variant), with the key difference being the selection of nodes. Instead of choosing four adjacent nodes from the fine mesh, the F4CR selects the four vertices of each coarse element, those with the greatest separation forming a first-order, four-node rectangular element, as illustrated in Figure 3.19.

In F4CR, only the four coarse-vertex nodes of each element contribute to the local stiffness matrix. For any remaining node $n_m \in \mathcal{N}$, that lies along the edges between these vertices (and is therefore not included in the coarse element's DOF set), the mapping is defined by assigning an identity relation:

$$M_{mm} = 1, \quad M_{mj} = 0 \quad \text{for } j \neq m \tag{3.147}$$

which effectively leaves these nodes unchanged rather than coupling them with other DOFs. Here, M denotes the preconditioner matrix prior to full assembly. This operation keeps the original connectivity of these DOFs intact while integrating seamlessly with the coarse-level structure. After this adjustment, the coarse-vertex elements are treated identically to the F4R formulation for local stiffness computation. The global FEM1 preconditioner matrix M_{FFCR} is then obtained by assembling all local coarse-vertex element matrices across the domain using the procedure outlined in section 3.6.



A) B)

Figure 3.20: Example of the F4CR (first-order, four-node coarse rectangle) variant of the FEM1 preconditioner. (A) Node numbering of a third-order quadrilateral element. (B) The four corner vertices (1, 4, 13, 16) are selected to form the coarse rectangular element used in the stiffness calculation, while the internal nodes are connected through an identity mapping to ensure continuity in the preconditioner assembly.

After introducing the individual preconditioners in this section, it is useful to compare their sparsity patterns alongside that of the original stiffness matrix. Figure 3.20 shows a clear progression in how nonzero entries are distributed symmetrically around the main diagonal. In the original stiffness matrix (A), the fill is defined solely by the connectivity of the underlying finite element mesh, producing a moderately narrow band structure. Applying the F4R preconditioner (B) preserves this band width but spreads the nonzero more evenly away from the diagonal, as a result of couplings between more distant nodes across the 4-node elements. Switching to F4CR (C) reduces the overall number of nonzero and slightly narrows the band, reflecting the coarser mesh resolution used in its construction, while still maintaining a regular, symmetric distribution. In contrast, the triangular-element version F3T (D) has a nonzero count similar to F4R but concentrates on these entries much closer to the diagonal in a symmetric, compact core. Compared to the more dispersed pattern of F4R, this closer clustering means that most couplings occur between nearby degrees of freedom, which improves numerical conditioning and can reduce the number of CG iterations required for convergence.

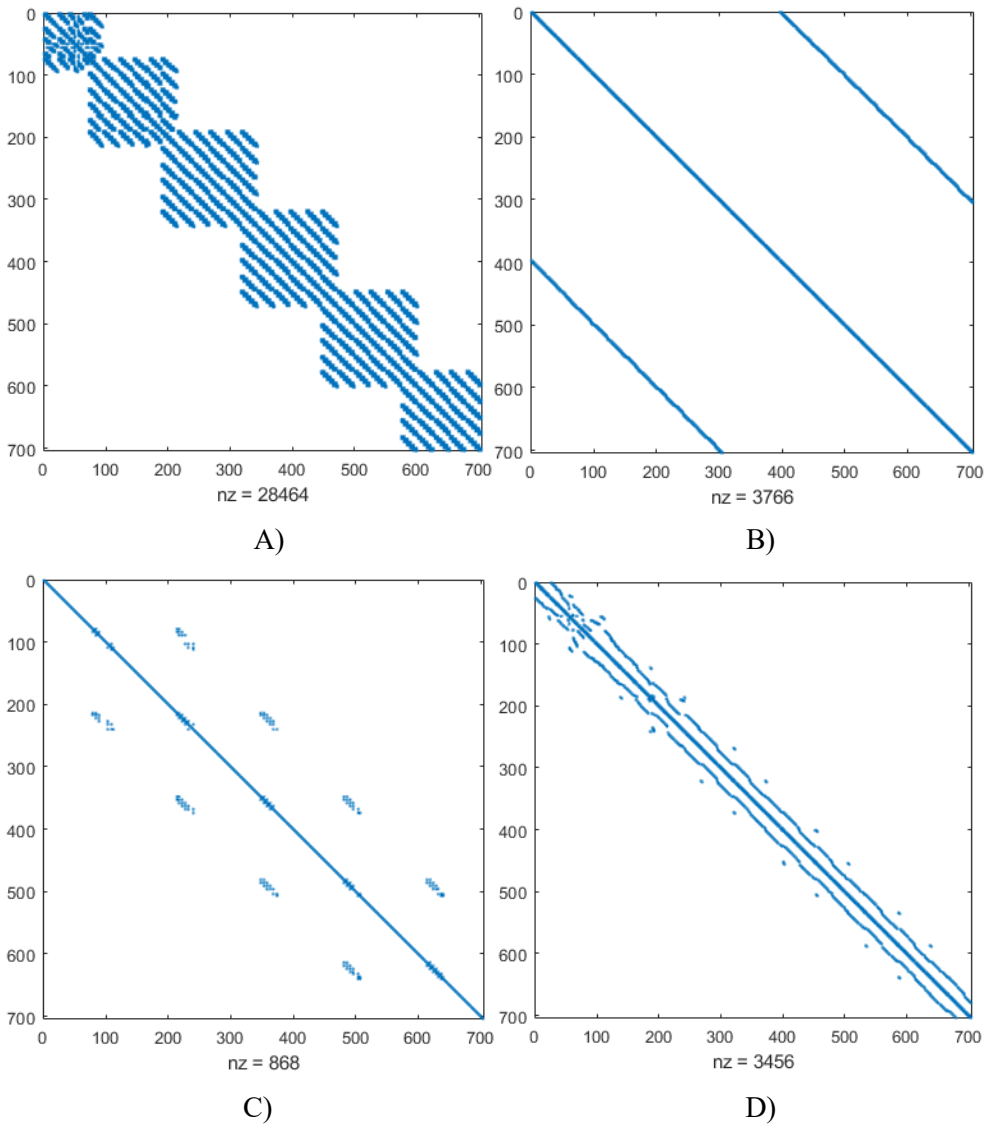


Figure 3.21: Sparsity patterns for (A) the global stiffness matrix A (nz = 28,464) corresponding to the mesh shown in Figure 3.11(c), and the associated preconditioner matrices: (B) F4R variant (nz = 3,766), (C) F4CR variant (nz = 868), and (D) F3T variant (nz = 3,456). Each preconditioner significantly reduces the number of nonzero entries compared to A, with the degree of sparsity reflecting the underlying coarsening strategy F4R and F4CR retain only the connectivity of selected coarse nodes, while F3T preserves additional couplings due to its triangular-element construction.

In one dimension, the construction of the FEM1 preconditioner simplifies significantly. Instead of forming rectangular or triangular elements, the matrix is built directly from pairs of consecutive nodes, each pair corresponding to a first-order, two-node finite element.

3.10.4 p-Multigrid Preconditioner

The theoretical background and general principles of multigrid methods, including their role in attenuating high-frequency error modes and correcting low-frequency components through coarse-grid projections, were introduced in Section 1.9.3.

Here, we detail the specific p-multigrid preconditioner employed in this work. In this approach, the mesh remains fixed while the polynomial order of the basis functions is reduced across levels, with the coarsest level consisting of first-order elements .In our setting, the coarsest level corresponds to first-order elements. At the finest level (original polynomial order of the discretization), a couple of iterations of the SSOR method are applied to the system matrix A_h with the current approximation u_h, serving as smoother to damp high-frequency (oscillatory) components of the error. The algebraic structure of the coarse-level operators generated during successive adaptive refinements, which form the foundation of this process, is presented later in Figure 3.17.

A single V-cycle in the implemented p-multigrid preconditioner proceeds as follows:

1. Pre-smoothing (Fine Level):

At the finest discretization level h, a couple of iterations of the SSOR method which was introduced in Section 3.9.5 are applied to the system.

$$A_h u_h = b_h \tag{3.148}$$

starting from the current approximation u_h. This step damps high-frequency (oscillatory) error components that the coarse grid cannot represent.

2. Residual Computation:

The residual is evaluated as

$$r_h = b_h - A_h u_h \tag{3.149}$$

3. Restriction to Coarse Level:

Both the residual and the current fine-level approximation are transferred to the coarser polynomial space via the interpolation operator I_h^{2h} (order-reduction matrix):

$$r_{2h} = \left(I_h^{2h}\right)^T r_h \tag{3.150}$$

$$u_{2h} = \left(I_h^{2h}\right)^T u_h \tag{3.151}$$

The coarse-level stiffness matrix is assembled as

$$A_{2h} = \left(I_h^{2h}\right)^T A_h I_h^{2h} \tag{3.152}$$

4. Coarse-grid Correction:

The coarse problem

$$A_{2h} e_{2h} = r_{2h} \tag{3.153}$$

is solved approximately using CG, PCG, or MPCG with an appropriate preconditioner (Jacobi or SSOR). As discussed in Sections 3.9.3.1 and 3.9.4.1, the solution could also be obtained using a nested preconditioner approach. However, in this thesis, the PCG method

is employed with a Jacobi preconditioner ($M_{\text{Jacobi}} = \text{diag}(A_{2h})$), where the preconditioner is constructed from the stiffness matrix assembled at the coarse level.

5. Prolongation to Fine Level:

The coarse correction is interpolated back to the fine space:

$$e_h = I_h^{2h} e_{2h} \tag{3.154}$$

and the fine-level solution is updated:

$$u_h \leftarrow u_h + e_h \tag{3.155}$$

6. Post-smoothing (Fine Level):

A further set of SSOR iterations are applied at the fine level to remove any high-frequency error modes introduced during interpolation. The above V-cycle is repeated until the convergence criterion:

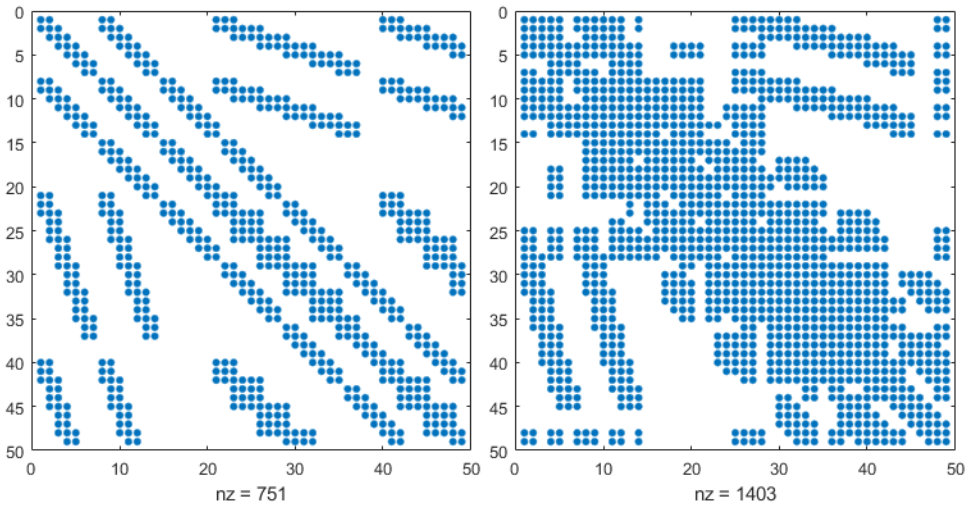
$$\left|\left|b_h - A_h u_h\right|\right|_\infty < \varepsilon_{\text{tol}} \tag{3.156}$$

is satisfied. I have to note the convergence criterion in this work is based on the L_∞ norm of the residual vector, which provides a strict measure of the largest local error. Formally, for a residual vector $r \in \mathbb{R}^n$ where $\mathbb{R}$ denotes the set of all real numbers and superscript n indicates that we are dealing with ordered tuples of n components:

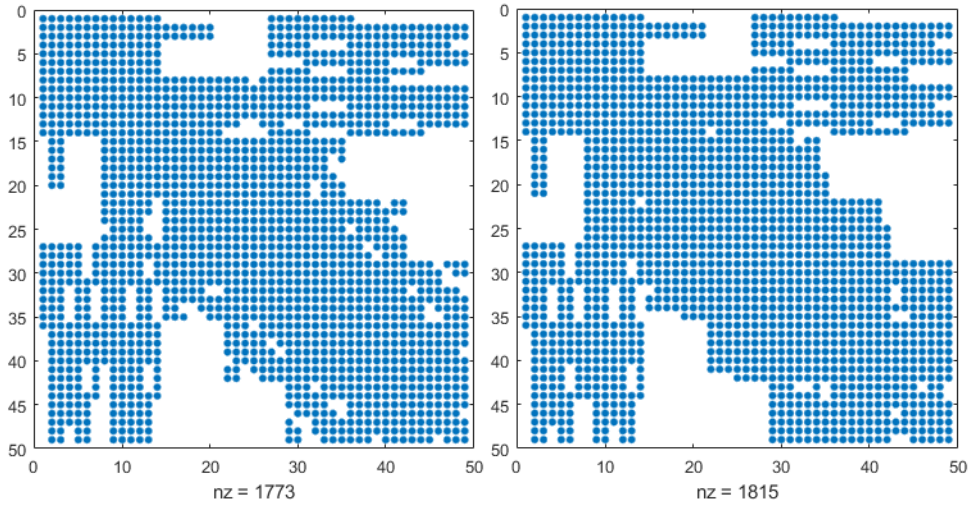
$$||r||_\infty = \max_{1 \le i \le n} ||r_i|| \tag{3.157}$$

where r_i denotes the $i-$th component of r. Convergence is declared when $||r||_\infty$, falls below a prescribed tolerance, ensuring that the largest residual across all degrees of freedom is sufficiently small. While L^2 is commonly used in iterative solver analysis as described in detail in Section 3.7.1.2, L_∞ offers a more conservative stopping criterion, making it particularly suitable for applications where strict control over the maximum pointwise error is required. This implementation ensures that both $high$- and low-frequency error components are effectively reduced, with fine-level smoothers targeting oscillatory modes and coarse-level corrections addressing the smooth residual components. As noted, convergence hinges on reducing oscillatory error at the fine level while correcting smooth error on the coarse level. Accordingly, we employ SSOR at the fine level in Steps 1 (pre-smoothing) and 6 (post-smoothing) of the V-cycle. SSOR is responsible for damping high-frequency modes before restriction and after prolongation; the formulation used here is summarized next.

Figure 3.22 shows the algebraic form of the coarse-level systems after four $p-$adaptive refinements (A–D). Each panel corresponds to the coarse-level counterpart of the solutions shown in Figure 3.14: panel (A) relates to Figure 3.11(A), panel (B) to Figure 3.11(B), and so on for panels (C) and (D). These sparsity patterns represent the coarse-level operators, not the original stiffness matrices, and reflect how the refinement process modifies the system within the $p-$multigrid cycle. This structure is essential for defining restriction and prolongation operators in the multigrid method. For further details on algebraic multigrid (AMG) see [104] and [105].



A) B)



C) D)

Figure 3.22: Sparsity patterns of the coarse-level matrices obtained after four successive p −adaptive refinements. Panels (A–D) correspond to the coarse-level counterparts of the solutions in Figure 3.14(A–D), respectively. These patterns represent the coarse operators used in the p −multigrid cycle rather than the original stiffness matrices.

Chapter 4: Results and Discussion

4.1 Problem Setup for 1D Cases

We begin by analyzing the performance of the iterative solvers in the 1D uniform mesh setting. All one-dimensional test problems are based on the steady-state Poisson equation $-\nabla^2 u(x) = f(x)$ in the domain x ∈ [0,1], subject to Dirichlet boundary conditions $U(0) = U(1) = 0$. In this study, we choose the source term as $f(x) = x^{50}$. The corresponding exact solution is $U(x) = -\frac{x^{52}}{2652} + \frac{x}{2652}$ which satisfies the prescribed boundary conditions. Throughout this section, we use the symbol $U(x)$ to denote the analytical (exact) solution in order to distinguish it clearly from the numerical approximations produced by the iterative solvers. Poisson equation is discretized using the finite element method (FEM), starting from an initial mesh of three uniformly sized elements, each with polynomial degree two. A schematic of this model problem, including the domain and boundary conditions, is shown in Figure 3.1(B). Unless otherwise stated, this setup serves as the baseline for both uniform and adaptive refinement studies, as well as for the comparison of different preconditioners. Convergence of the iterative solvers was monitored until the relative residual norm Equation (3.112) was reduced to 10^{-10}. Error norms for the computed solutions were evaluated down to 10^{-8}.

All numerical experiments, either 1D or 2D were carried out on a workstation equipped with a 12th Gen Intel(R) Core (TM) i7-12700H processor (2.70 GHz), 16 GB RAM (15.7 GB usable), running a 64-bit operating system.

Note_1: For brevity in the tables', abbreviated labels are used: *Ad* for Adaptation step, *p* for Polynomial order, *El* for number of Elements, and *Iter* for number of Iterations.

Note_2: To simplify notation, we drop the explicit word "*preconditioner"* and refer to methods only by their acronyms (e.g., SSOR, FEM1).

Note_3: All reported runtimes represent the average over seven independent solves, taken to reduce timing sensitivity and ensure a more stable performance comparison.

4.2 1D Uniform Mesh Refinement

Two uniform refinement strategies are considered: (i) *p*-refinement, where the polynomial order of all elements is increased while keeping the mesh fixed, and (ii) *h*-refinement the number of elements in the mesh is increased while keeping the same polynomial order for each element. As the mesh is subdivided, the element size decreases, and in the uniform refinement setting considered here, all elements remain of equal size. This setting provides a baseline reference for evaluating the effect of preconditioners under simple mesh configurations. In Section 3.7, the error was estimated through the decay of modal coefficients, which gave a reliable criterion for adaptive refinement. Although this method was explained in detail before and tested successfully, here we use a simpler

approach by relying on the exact solution. The main reason is to keep focused on testing the preconditioners, which are the main subject of this work. Using the exact solution also avoids the extra cost of repeating the error estimation, which helps to reduce computation time while still letting us study the solver behavior under uniform mesh refinement.

A) B)

Figure 4.1: Example of *p*-uniform refinement on a uniform mesh with three elements. (A) corresponds to polynomial degree 2, while (B) corresponds to polynomial degree 3. The blue points indicate the additional nodes introduced by the higher-order basis functions.

A) B)

Figure 4.2: Example of *h*-uniform refinement. In (A), the mesh consists of three elements with polynomial degree 2. In (B), the mesh is refined into 13 elements while maintaining the same polynomial degree 2.

4.2.1 *p*- Uniform Refinement

In this section, the solver is stopped as soon as the convergence condition in Equation (3.112) is satisfied. After that, Equation (3.69) is checked, if the error is below or equal to tolerance, the process ends, and the problem is considered converged. If not, the mesh must be refined by increasing the polynomial degree of all elements by one. Importantly, when the degree is raised, the nodes within each element are redistributed so that the spacing between them remains uniform. In other words, the additional node is not simply inserted arbitrarily, but the entire node arrangement is updated to maintain equal distances between nodes, consistent with the procedure illustrated in Figure 3.2 and described in Section 3.5. The new system is then solved again, and the procedure is repeated step by step until both criteria (Equation (3.69) and (3.112)) are satisfied.

Table 4.1 reports the iteration counts for Jacobi, SSOR, FEM1, and the combined Jacobi and FEM1 applied to uniformly refined 1D problems. A key observation is that the combined preconditioner MPCG (Jacobi+FEM1) exhibits nearly identical behavior to FEM1 alone, showing that the dominant effect comes from the FEM1 component. Compared with Jacobi, FEM1 achieves a remarkable reduction in iterations, for instance, with a 35×35 matrix, Jacobi requires 130 iterations to converge, whereas FEM1 converges in only 14 steps a reduction of about 89%.

Equally important is the growth rate of the iterations as the matrix size increases. For Jacobi, the iteration count rises sharply from 6 to 130 as the system grows from 5×5 to 35×35, while FEM1 shows only a gradual increase, from 3 to 14 iterations across the same

range. SSOR performs better than Jacobi but still exhibits a faster growth in iteration counts compared to FEM1, confirming that SSOR provides only a partial improvement in conditioning (see Table 4.3). These results highlight the superior robustness of FEM1, both alone and in combination with Jacobi, in maintaining low iteration counts and controlling the growth of computational effort as the problem size increases.

Table 4.1: Iterative performance of preconditioned CG solvers for increasing matrix sizes under p −uniform refinement, where the polynomial degree of all three elements is raised uniformly from 2 to 12 for 1D.

				Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*p*	*El*	*Iter*	*Iter*	*Iter*	*Iter*
5^2	1	2	3	6	5	3	4
8^2	2	3	3	9	8	4	5
11^2	3	4	3	12	11	5	7
14^2	4	5	3	17	14	6	8
17^2	5	6	3	23	18	7	10
20^2	6	7	3	29	23	9	11
23^2	7	8	3	37	31	10	11
26^2	8	9	3	53	42	13	13
29^2	9	10	3	69	55	12	14
32^2	10	11	3	97	77	14	14
35^2	11	12	3	130	99	14	15

From Figure 4.3 it is clear that the number of iterations increases steadily with matrix size across all four preconditioners. Jacobi and SSOR exhibit very similar behavior, both showing a nearly linear growth in iterations with increasing matrix size. In contrast, FEM1 and MPCG (Jacobi+FEM1) require noticeably fewer iterations overall and follow almost identical trends. Interestingly, for matrix sizes up to 26×26, all four methods increase with comparable slopes, but after that, FEM1 and FEM1+Jacobi essentially flatten, maintaining a nearly constant iteration count, while Jacobi and SSOR continue to rise with the same slope (1.2 roughly). This indicates that the FEM1 based approaches scale more efficiently with problem size, whereas Jacobi and SSOR become less effective as the system grows.

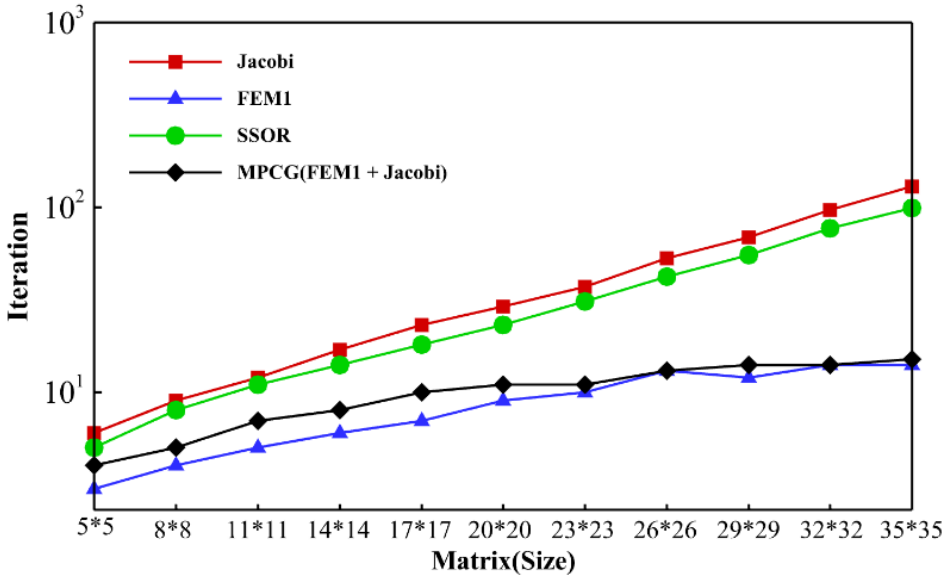


Figure 4.3: Iteration counts for different preconditioners under *p*-uniform refinement for 1D.

Table 4.2 reports on the solver runtimes for different preconditioners as the matrix size increases. Unlike the iteration counts, here the Jacobi preconditioner consistently exhibits the lowest runtime across all system sizes. In contrast, the SSOR is strongly affected by the matrix dimension, its runtime grows steeply, with the ratio between the largest and smallest case reaching more than 150. For the other preconditioners (Jacobi, FEM1, and the combined preconditioners (Jacobi+FEM1) this ratio is much smaller about 15.3, 4.3, and 9.2, respectively indicating greater reliability to mesh refinement. Among these, FEM1 shows the most stable performance, confirming its effectiveness in maintaining efficiency even as the problem size grows.

Table 4.2: Runtimes of different preconditioners with increasing matrix size in p −uniform refinement for 1D.

				Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*p*	*El*	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)
5^2	1	2	3	3.08E-05	3.35E-05	2.18E-03	2.02E-04
8^2	2	3	3	2.94E-05	5.86E-05	2.80E-03	2.40E-04
11^2	3	4	3	3.54E-05	1.16E-04	3.46E-03	3.94E-04
14^2	4	5	3	7.56E-05	1.98E-04	4.35E-03	5.22E-04
17^2	5	6	3	6.80E-05	3.13E-04	5.02E-03	6.95E-04
20^2	6	7	3	8.86E-05	4.34E-04	5.55E-03	8.45E-04
23^2	7	8	3	1.06E-04	7.56E-04	6.39E-03	8.96E-04
26^2	8	9	3	1.53E-04	1.20E-03	7.34E-03	1.24E-03

29^2	9	10	3	2.04E-04	1.91E-03	7.77E-03	1.40E-03
32^2	10	11	3	2.92E-04	3.07E-03	8.69E-03	1.48E-03
35^2	11	12	3	4.72E-04	5.06E-03	9.44E-03	1.86E-03

From Figure 4.4, we see that the SSOR maintains competitive runtime performance up to a matrix size of 26×26, ranking second only to Jacobi. Within this range, all other preconditioners (Jacobi, FEM1, and MPCG (Jacobi+FEM1)) increase runtime with nearly the same slope, while SSOR shows a steady, almost linear growth from the smallest to the largest matrix size (35×35). Beyond the size of 26×26, the behavior diverges both FEM1 and Jacobi+FEM1 exhibit nearly flat growth, with MPCG (Jacobi+FEM1) showing only a marginally higher increase compared to FEM1. In contrast, Jacobi experiences a pronounced rise in runtime, with a significantly steeper slope than in the earlier range from 5×5 to 14×14. Overall, these results suggest that as the mesh size grows in p −uniform refinement, FEM1 and Jacobi+FEM1 become increasingly effective, maintaining reliability performance, whereas Jacobi and SSOR lose efficiency, with SSOR showing persistent linear growth and Jacobi deteriorating more sharply after the critical size threshold.

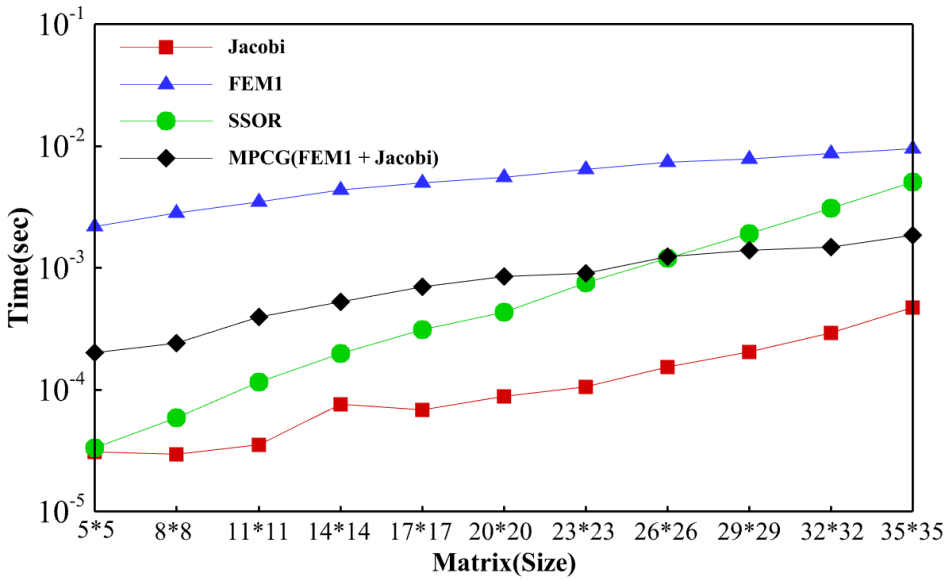


Figure 4.4: Runtime comparison of Jacobi, FEM1, SSOR, and FEM1+Jacobi preconditioners with increasing matrix size in *p*-uniform refinement for 1D.

For all cases, the condition number grows almost linearly as the matrix size increases as shown in Figure 4.5, but with different slopes. For Jacobi, SSOR, and the combined preconditioner (Jacobi+FEM1), the growth rate is about 3.2, while for FEM1 it is smaller (about 2.2), meaning FEM1 slows down growth more effectively. Up to size 17×17, Jacobi behaves very close to the unpreconditioned system, with virtually no change, apart from a negligible increase in the condition number. After this point, the difference becomes more

noticeable, the unpreconditioned system continues to grow faster (average slope about 2.5), while the Jacobi grows even faster (average slope about 2.5).

Table 4.3: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and the unpreconditioned system under *p*-uniform with 3 elements for 1D.

				UP	Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*p*	*El*	Condition Number	Condition Number	Condition Number	Condition Number	Condition Number
5^2	1	2	3	1.80E+01	1.80E+01	3.39E+00	1.46E+00	2.22E+00
8^2	2	3	13	6.07E+01	6.14E+01	8.89E+00	2.17E+00	2.53E+01
11^2	3	4	23	1.77E+02	1.80E+02	2.14E+01	3.50E+00	1.08E+02
14^2	4	5	33	5.02E+02	5.15E+02	5.28E+01	6.20E+00	4.08E+02
17^2	5	6	43	1.45E+03	1.56E+03	1.45E+02	1.22E+01	1.19E+03
20^2	6	7	53	4.37E+03	5.27E+03	4.71E+02	2.66E+01	4.63E+03
23^2	7	8	63	1.39E+04	2.03E+04	1.84E+03	6.40E+01	1.76E+04
26^2	8	9	73	4.61E+04	8.93E+04	8.26E+03	1.67E+02	1.05E+05
29^2	9	10	83	1.59E+05	4.37E+05	4.11E+04	4.64E+02	6.72E+05
32^2	10	11	93	5.61E+05	2.31E+06	2.19E+05	1.35E+03	4.21E+06
35^2	11	12	103	2.02E+06	1.30E+07	1.22E+06	4.09E+03	3.42E+07

The combined Jacobi and FEM1 reduce the condition number at first up to 17×17, but beyond this size of matrix the performance worsens, and the curve becomes concave upward, showing that it becomes less effective as the mesh is refined. SSOR provides better improvement, but as the system grows larger, its effect weakens. For example, at the largest matrix size (35×35) and smallest system (5×5), the condition number with SSOR is 1.22×10^6 and 3.39×10^0 respectively, while the unpreconditioned (UP) case is 2.02×10^6 and 1.80×10^1 respectively as reported in Table 4.3. This shows that the effectiveness of SSOR is strongly affected by the mesh size. In contrast, FEM1 consistently shows the best performance, keeping condition number much lower and growing more slowly compared to the other methods. For example, in Figure 4.6, the eigenvalue distributions of the system matrix were obtained after four refinement steps (17×17 matrix size).

In the unpreconditioned case, which corresponds to the Stiffness Matrix (A) in Figure 4.6, the eigenvalues cover a very wide interval, ranging from approximately 0.54 up to nearly 794, but they are not evenly distributed across this span. Instead, they form three main regions of concentration, one in the low range [0.54, 62.05], another between [158.6, 180], and a final cluster in the high range [780.4, 793,9]. These bands are separated by

large gaps, which means that most eigenvalues lie within narrow local groups, while only a few values are spread across the upper end of the spectrum.

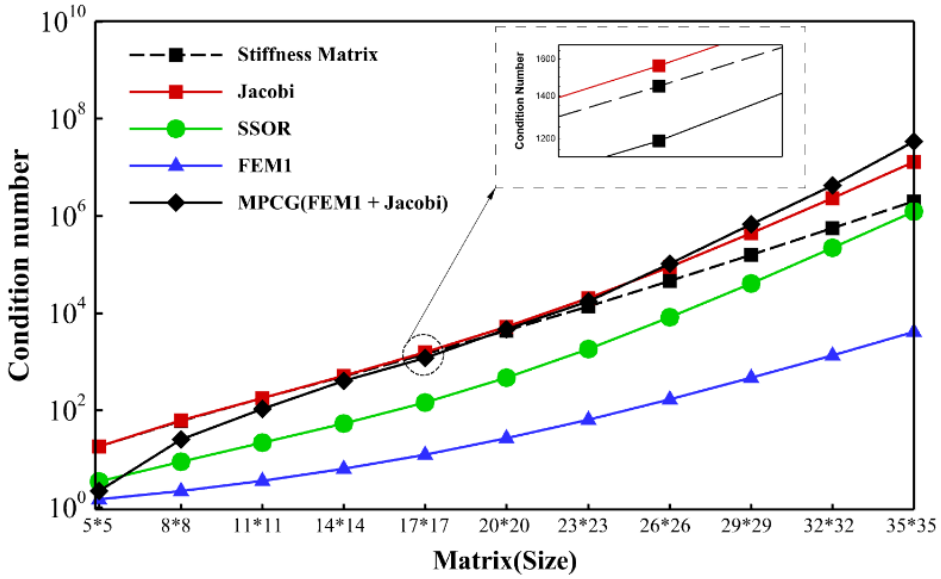


Figure 4.5: Variation of condition numbers with polynomial degree for preconditioned and Stiffness Matrix CG solvers under p −uniform refinement with three elements for 1D.

When the Jacobi preconditioner is applied (B) in Figure 4.6, the structure of the spectrum changes noticeably. The sharp separation between bands disappears, and the eigenvalues become distributed more smoothly and uniformly across the range up to about 3.8. While this reduces some of the extreme outliers, it also eliminates the compact clusters that were present before compared to part (A). The result is that the eigenvalues no longer concentrate in a few intervals but instead cover the entire range more continuously. From the point of view of iterative methods, the residual polynomial must cover the whole spread of eigenvalues, rather than just a few compact groups. In Figure 4.6(C) the SSOR preconditioner provides a more moderate modification. Here, the spectral interval is compressed to [0, 1], and some weak clustering begins to appear, with small groups of eigenvalues forming local plateaus. However, the overall spectrum still shows scattered points between these groups, so the distribution cannot be described as strongly clustered. While the spread is reduced compared to the unpreconditioned and Jacobi cases, the lack of clear grouping means that higher-degree polynomials are still required to handle the scattered distribution effectively.

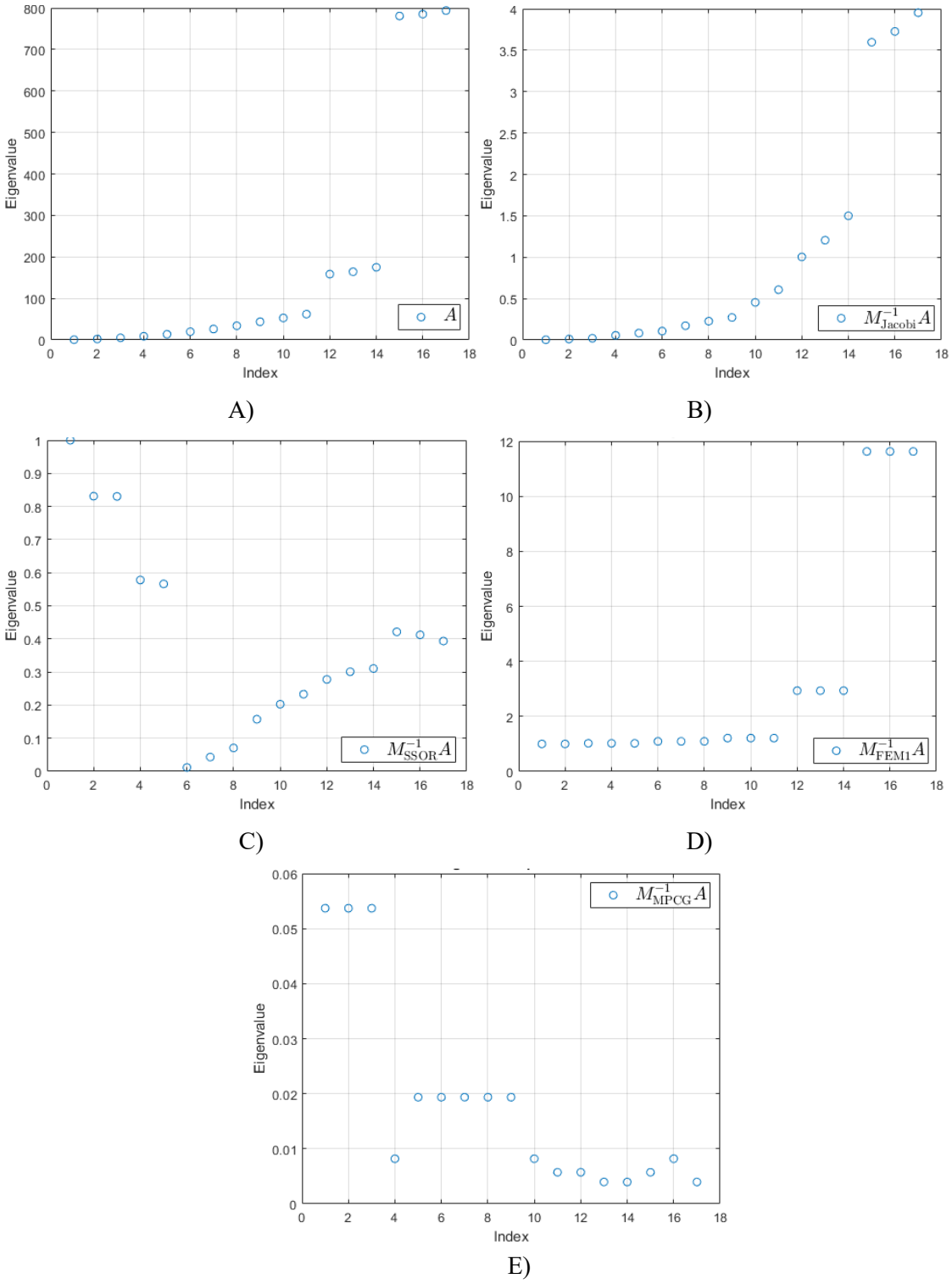
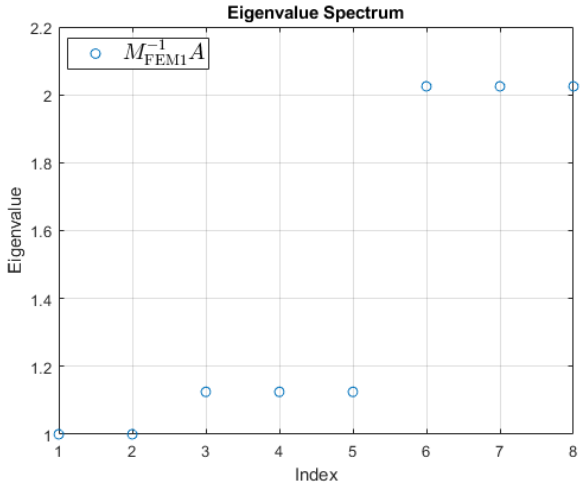
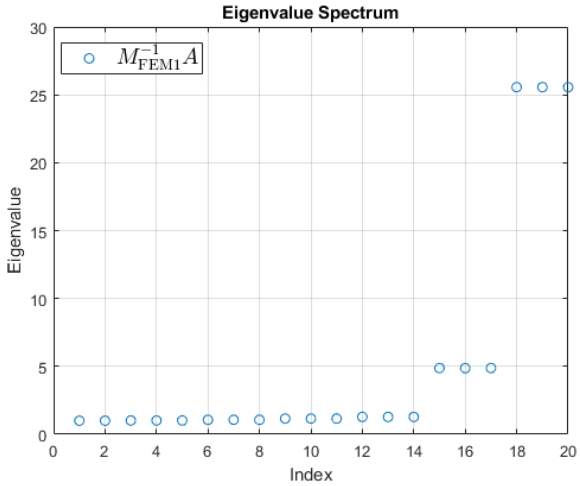


Figure 4.6: Eigenvalue distributions under different preconditioners for *p*-refinement with matrix size 17×17, obtained after four refinement steps. (A) Unpreconditioned system. (B) Jacobi preconditioner. (C) SSOR preconditioner. (D) FEM1 preconditioner. (E) MPCG preconditioner for 1D.

In contrast, the FEM1 (D) in Figure 4.6 changes the spectrum so that the eigenvalues fall into about six clear clusters. Each cluster is tight, with a gap separating it from the next, so the distribution looks almost step-like. To further illustrate this clustering effect, Figure 4.7 presents two additional cases, for element orders $p = 3$ [Figure 4.7 (A)] and $p = 7$ [Figure 4.7 (B)], the spectrum splits into roughly three and seven clusters, respectively. In this situation, CG does not need a polynomial that controls the whole interval, but only one that is small on each cluster. In other words, the number of clusters gives a rough guide for the degree of the polynomial needed, since each cluster can be covered by one root.



A)



B)

Figure 4.7: Eigenvalue distributions of the FEM1 preconditioned system under $p-$uniform refinement. (A) corresponds to the initial case with polynomial degree $p = 3$. (B) shows the refined case after uniformly increasing the polynomial degree of all elements to $p = 7$ for 1D.

This makes the cluster count more important than the overall condition number, with three and seven clusters, the method would in theory need no more than three and seven steps. In practice, however, we found that only four and nine iterations were required to reach convergence. The corresponding finite element solutions are shown in [Figure 4.8 (A)] for the case in Figure 4.7 (A) and Figure 4.8 (B) for the case in Figure 4.7 (B). As the polynomial order increases, the solution field becomes smoother, and the error decreases from 2.21×10^{-5} (for $p = 3$) to 1.65×10^{-6} (for $p = 7$). This comparison illustrates how increasing the polynomial order makes the solution field progressively smoother and closer to the exact solution.

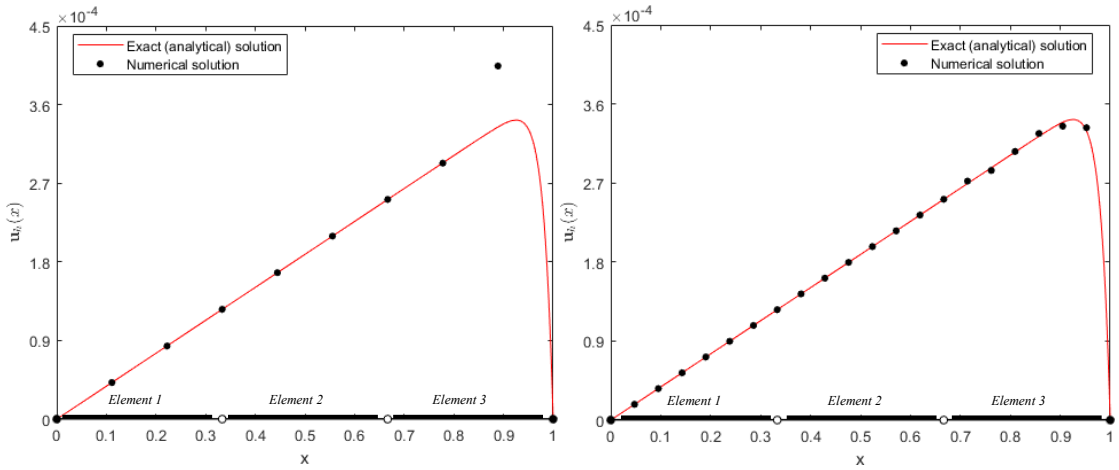


Figure 4.8: Finite element solutions under p-uniform refinement with three elements. (A) Solution with polynomial degree $P = 3$ (error = 2.21×10^{-5}). (B) Solution with polynomial degree $P = 7$ (error = 1.65×10^{-6}). In both cases, refinement is performed by uniformly increasing the polynomial order of all elements for 1D.

4.2.2 *h* - Uniform Refinement

In this refinement strategy, the polynomial degree of elements is kept fixed, while the mesh is refined by progressively adding more elements. The process begins with a coarse mesh consisting of three elements. If the error given by Equation (3.69) does not meet the tolerance, the mesh is refined by adding ten elements at each step (see Figure 4.2) with all the elements (old and new) made equal in size. Choosing a step of ten elements, rather than refining one element at a time, avoids excessive iterations with only marginal error reduction and makes the error decay trend clearer. This strategy reduces the number of refinement stages while still capturing the overall solver behavior under h −uniform refinement. For each such mesh, the solver (see Section 3.9) is run until the convergence criterion in Equation (3.112) is satisfied. After this solve, the error is re-evaluated using Equation (3.69). If the tolerance is still not met, the mesh is further refined, and the process

is repeated. Convergence is declared once Equation (3.69) is satisfied. The numerical results are presented in Table 4.4 and 4.5 and Figures 4.9 and 4.10.

In Table 4.4 the solver iteration counts are reported for different preconditioners as the matrix size increases. The Jacobi preconditioner shows the steepest growth, with the number of iterations rising from 6 to 219 as the system size increases, giving a ratio of about 34.8 between the largest and smallest matrix size. The SSOR performs better, with its iteration count increasing from 5 to 91, corresponding to a ratio of about 18.2. By contrast, the FEM1 exhibits completely reliable performance, requiring only 3 iterations across all system sizes, which confirms its robustness under h −refinement. The combined Jacobi and FEM1 also maintain low iteration count, increasing gradually in a stepwise manner, remaining at 4 iterations for small matrices size (5×5), then rising to 6 for range of matrix between 25×25 and 85×85, and eventually stabilizing at 7 for the range of matrix size between 105×105 and 205×205.

Table 4.4: Iterative performance of preconditioned CG solvers for growing matrix sizes in h −uniform refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

				Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*p*	*El*	*Iter*	*Iter*	*Iter*	*Iter*
5^2	1	2	3	6	5	3	4
25^2	2	2	13	26	18	3	6
45^2	3	2	23	46	27	3	6
65^2	4	2	33	66	36	3	6
85^2	5	2	43	88	44	3	6
105^2	6	2	53	109	52	3	7
125^2	7	2	63	131	60	3	7
145^2	8	2	73	153	68	3	7
165^2	9	2	83	175	75	3	7
185^2	10	2	93	197	83	3	7
205^2	11	2	103	219	91	3	7

The Jacobi and SSOR exhibit very similar behavior as shown in Figure 4.9. Both methods display a steep increase in iteration count as the matrix size grows whereas the FEM1 preconditioner maintains a constant iteration count. For the combined preconditioners MPCG (Jacobi+FEM1), the iteration count initially rises sharply from

matrix size 5×5 to 25×25, similar to Jacobi and SSOR. However, beyond this point, the growth stabilizes and remains nearly constant, showing that FEM1 dominates and compensates for the weaknesses of Jacobi. Meanwhile, Jacobi and SSOR continue to grow with matrix size, with SSOR performing slightly better but following the same upward trend.

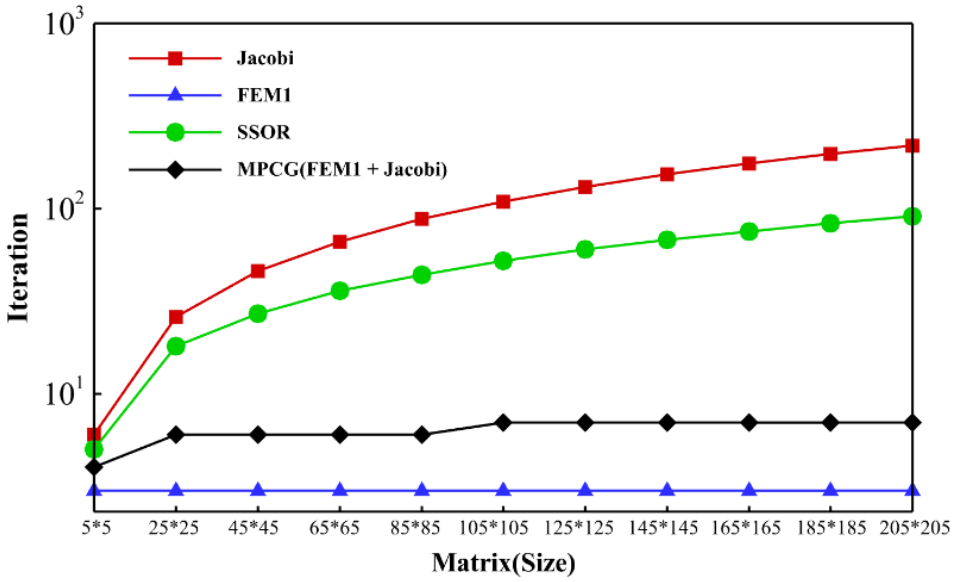


Figure 4.9: Iteration counts for different preconditioners under h −uniform refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

It is evident that Jacobi consistently requires less computational time compared to the other preconditioners (see Table 4.5). However, its sensitivity to matrix size is pronounced. In the range from5×5 to 85×85, the runtime increases from 3.08×10^{-5} to 1.24×10^{-3}, giving a ratio of 40.26 (see Figure 4.10). In contrast, in the range from 85×85 to 205×205, the runtime increases only from 1.24×10^{-3} to 6.49×10^{-3}, yielding a much smaller ratio 5.23. This indicates that Jacobi is efficient for smaller problems, but its performance deteriorates more rapidly as the system size increases.

Table 4.5: Runtimes of different preconditioners with increasing matrix size in h −uniform refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

				Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*p*	*El*	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)
5^2	1	2	3	3.08E-05	3.35E-05	2.18E-03	2.02E-04
25^2	2	2	13	7.61E-05	4.16E-04	6.56E-03	4.81E-04
45^2	3	2	23	2.16E-04	1.43E-03	1.15E-02	8.18E-04

65^2	4	2	33	4.28E-04	3.17E-03	1.60E-02	1.41E-03
85^2	5	2	43	1.24E-03	6.16E-03	2.08E-02	2.41E-03
105^2	6	2	53	1.56E-03	1.19E-02	2.53E-02	4.65E-03
125^2	7	2	63	2.04E-03	1.64E-02	3.11E-02	7.02E-03
145^2	8	2	73	3.14E-03	2.56E-02	3.77E-02	9.65E-03
165^2	9	2	83	3.82E-03	3.29E-02	4.45E-02	1.26E-02
185^2	10	2	93	4.94E-03	4.10E-02	4.89E-02	1.67E-02
205^2	11	2	103	6.49E-03	5.58E-02	5.71E-02	2.16E-02

SSOR, which can be regarded as the next simplest option after Jacobi (see Section 3.10.2), shows a more dramatic growth in runtime as the matrix size increases. At the largest system size 205×205, its runtime becomes nearly equal to that of the FEM1. By comparison, FEM1 exhibits a much smoother increase with matrix size as shown in Figure 4.10. Between 5×5 and 105×105 the growth ratio is 12.25, while from 105×105 to 205×205 it decreases to 2.25. This demonstrates that FEM1 largely preserves its efficiency even as the problem size grows. On the other hand, for MPCG (Jacobi+FEM1), the runtime increases almost linearly with matrix size, maintaining a relatively uniform slope across the entire range. From Table 4.5 it is clear that for Jacobi and SSOR, the runtime order lies between 10^{-5} and 10^{-3} for Jacobi, and up to 10^{-2} for SSOR. For FEM1 and MPCG (Jacobi+FEM1), the upper bound of the runtime order is around 10^{-2}, while the lower bounds reach approximately 10^{-3} and 10^{-4}, respectively.

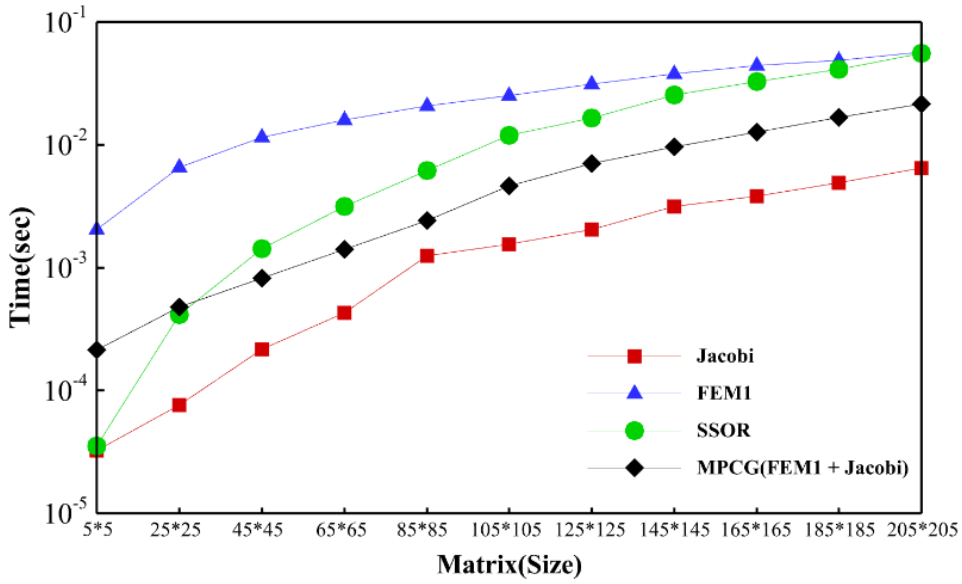


Figure 4.10: Runtime comparison of Jacobi, FEM1, SSOR, and FEM1+Jacobi preconditioners with increasing matrix size in h −uniform refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

Table 4.6 shows the condition numbers for different preconditioners under h −refinement, indicating that Jacobi, SSOR, and FEM1 follow a very similar trend, from matrix size 5×5 to 25×25, the condition number grows sharply (by a factor of about 20.4), after which the growth slows considerably, with an average increase of only 1.4 up to 205×205. This indicates that these preconditioners maintain relatively stable conditioning as the system size increases. In contrast, the combined MPCG (Jacobi+FEM1) scheme produces condition numbers that rise dramatically, reaching 1.76×10^{6} at the largest system size, while for the other three preconditioners they remain much smaller 2.31×10^{4} for Jacobi, 2.78×10^{4} for SSOR, and 1.40×10^{4} for FEM1. These patterns are also visible in Figure 4.11, where for smaller matrix (below 45×45), SSOR lies slightly below FEM1.

Table 4.6: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and the Stiffness Matrix under h −uniform refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

				UP	Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*p*	*El*	Condition Number	Condition Number	Condition Number	Condition Number	Condition Number
5^2	1	2	3	1.80E+01	1.80E+01	3.39E+00	6.00E+00	2.22E+00
25^2	2	2	13	3.64E+02	3.67E+02	4.53E+01	4.86E+01	4.34E+02
45^2	3	2	23	1.14E+03	1.15E+03	1.40E+02	1.18E+02	4.33E+03
65^2	4	2	33	2.35E+03	2.37E+03	2.86E+02	2.10E+02	1.85E+04
85^2	5	2	43	4.00E+03	4.03E+03	4.85E+02	3.23E+02	5.33E+04
105^2	6	2	53	6.07E+03	6.12E+03	7.37E+02	4.57E+02	1.23E+05
125^2	7	2	63	8.58E+03	8.65E+03	1.04E+03	6.10E+02	2.46E+05
145^2	8	2	73	1.15E+04	1.16E+04	1.40E+03	7.82E+02	4.44E+05
165^2	9	2	83	1.49E+04	1.50E+04	1.81E+03	9.71E+02	7.42E+05
185^2	10	2	93	1.87E+04	1.89E+04	2.27E+03	1.18E+03	1.17E+06
205^2	11	2	103	2.29E+04	2.31E+04	2.78E+03	1.40E+03	1.76E+06

However, beyond this range, FEM1 becomes the most effective preconditioner, consistently yielding the lowest condition numbers among all methods. The difference between SSOR and FEM1 remains small, with both outperforming Jacobi across all cases. Overall, Jacobi shows negligible effect on conditioning, whereas FEM1 demonstrates the strongest ability to improve conditioning as the mesh is refined.

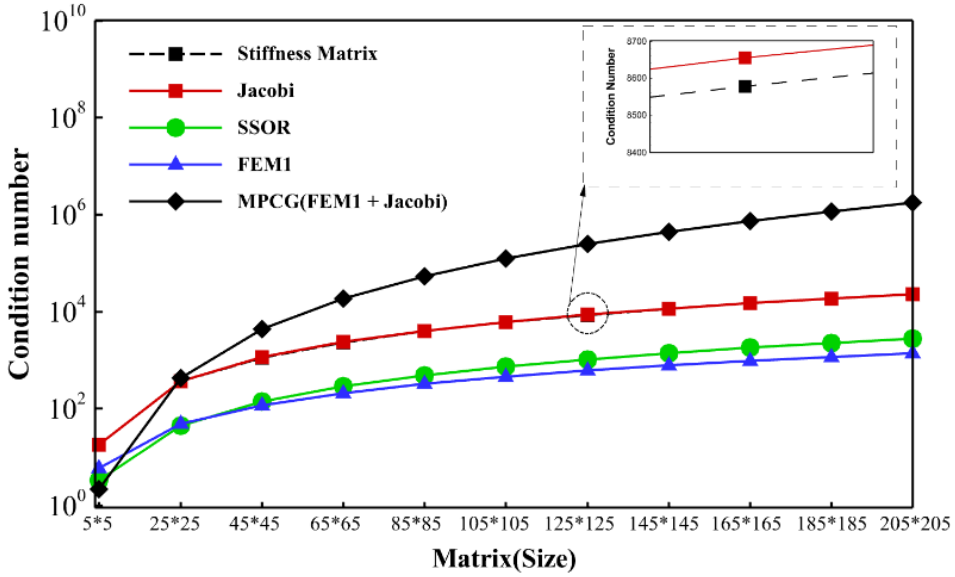
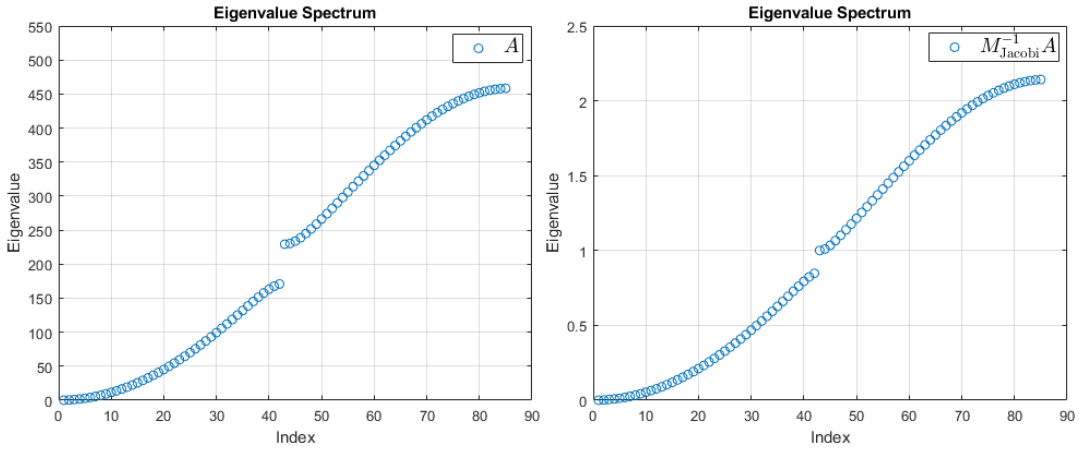


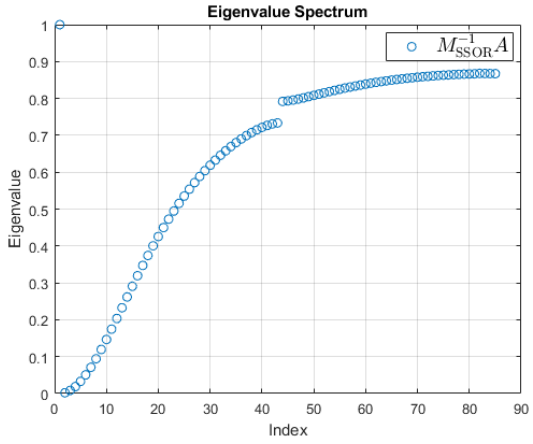
Figure 4.11: Variation of condition numbers with polynomial degree for preconditioned and Stiffness Matrix CG solvers under h −uniform refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

The eigenvalue distributions in Figure 4.12 illustrate the effect of different preconditioners on the spectral properties of the system at the fourth refinement, corresponding to a mesh of 43 elements. The corresponding finite element solutions are presented in Figure 4.13, where the improvement from refinement is clear, as the error decreases from 2.35×10^{-5}, in the initial mesh without refinement [Figure 4.13 (A)] to 1.09×10^{-8} after four refinement steps [Figure 4.13 (B)]. In (A), corresponding to the Stiffness Matrix, the eigenvalues are widely spread, which is very similar to the case with the Jacobi preconditioner in (B).

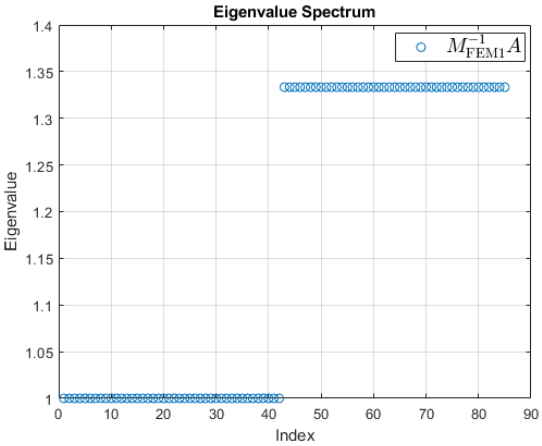


A)

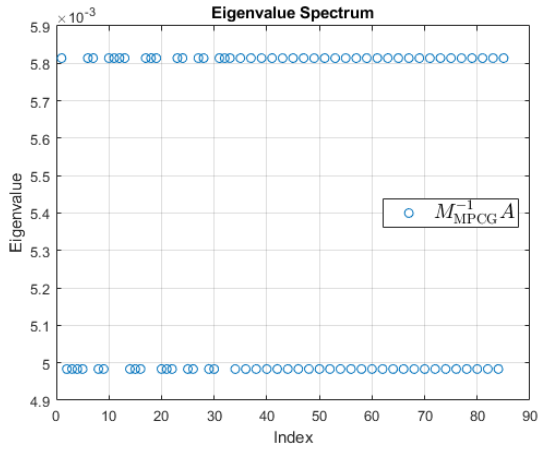
B)



C)



D)



C)

Figure 4.12: Eigenvalue distributions under different preconditioners for h −uniform refinement with matrix size 85 × 85, obtained after four refinement steps (43 elements). (A) Stiffness Matrix. (B) Jacobi preconditioner. (C) SSOR preconditioner. (D) FEM1 preconditioner. (E) MPCG preconditioner for 1D.

But in (C), eigenvalues are clustered near the upper bound of the eigenvalues (λ_{max}), despite having a somewhat larger condition number due to an isolated outlier. Such clustering is highly beneficial for CG; this algorithm can construct a relatively low-degree polynomial that quickly damps the bulk of the clustered eigenvalues, leaving only a small number of outliers to be resolved more gradually. In contrast, (B) shows a nearly uniform spread of eigenvalues over the full interval $[\lambda_{min}, \lambda_{max}]$ and since there is no clustering and the spectrum is spread over this wide range, the CG method must rely on higher-degree polynomials to approximate it effectively, which ultimately results in more iterations. Finally, in (D) for the FEM1 reveals a particularly favorable distribution, where the eigenvalues form two well-separated clusters, one near λ_{min} and the other near λ_{max}. Such a configuration is ideal for CG since in theory, convergence would occur in at most the number of distinct clusters (two in this case), while in practice about three iterations were sufficient to reach convergence in our experiments.

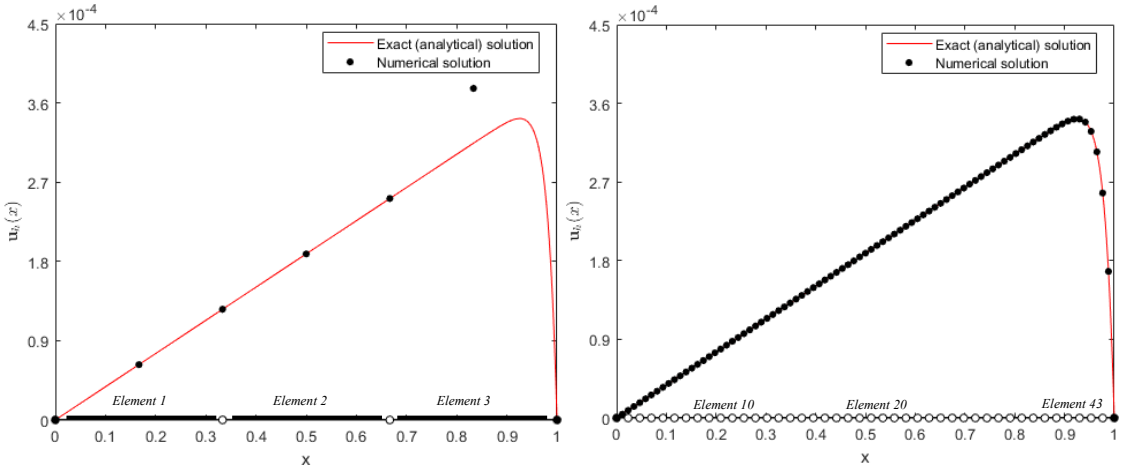


Figure 4.13: Finite element solutions under h −refinement in 1D. (A) Initial solution with 3 elements, each of polynomial degree $p = 2$ (error = 2.35×10^{-5}). (B) Solution after four refinement steps, resulting in 43 elements, each of polynomial degree $p = 2$ (error = 1.09×10^{-8}), highlighting the substantial improvement in accuracy in 1D.

4.3 1D Adaptive Mesh Refinement

In this section we investigate the three adaptive refinement strategies, namely p −adaptivity, h −adaptivity, and hp −adaptivity [18], which were each described in detail in Sections 3.8.1−3.8.3 As in the case of the one-dimensional uniform refinement study, the analysis here proceeds under the assumption that the exact solution is available. Although the error estimation procedure based on modal coefficient decay, discussed in Section 3.7.1, was successfully tested, we avoid using it here in order to reduce computational cost and to keep focus on the performance of the preconditioners. Working

with the exact solution eliminates the need for repeated error estimation while still allowing us to systematically study the solver behavior. The presentation is organized as follows: results for p −adaptivity are discussed first, followed by those for h −adaptivity, and finally the results for the combined strategy, hp −adaptivity.

4.3.1 p -, h -, hp - Refinement

In the adaptive refinement strategies, the solver must satisfy the convergence condition before the process can terminate. In addition, the elementwise error is estimated using Equation (3.64) and based on this information the mesh is adapted according to the chosen strategy.

For p −adaptivity, after evaluating all elements with Equation (3.64), those elements that fail to meet the tolerance are enriched in polynomial degree following the procedure outlined in Section 3.8.2, and the system is solved again. Similarly, in h −adaptivity, elements that do not satisfy the error criterion are subdivided according to Section 3.8.3, and the system is recomputed on the refined mesh. Finally, in the hp −adaptivity strategy, the same elementwise error evaluation is performed, but the decision of whether to refine an element by increasing its polynomial degree or by subdividing it is made according to the methodology described in Section 3.8.1. In this section, the results of all three adaptive strategies are presented side by side, so that their performance and efficiency can be directly compared.

According to the results reported in Tables 4.7, 4.8, and 4.9, the number of adaptive cycles required to reach the target tolerance depends strongly on the type of adaptivity. In the case of h −adaptivity, the solver required eleven adaptation steps before convergence, whereas in both p −adaptivity and hp −adaptivity only five steps were sufficient. In the last step of hp −adaptivity the matrix dimension reached 44×44, while in p −adaptivity it was only 16×16. Across all three strategies, the Jacobi preconditioner consistently produced the highest iteration counts, with a growth trend that was similar to SSOR. In h −adaptivity and hp −adaptivity [Figure 4.14(C)], the gap between Jacobi and SSOR increased as the matrix size grew, for instance, in h −adaptivity the difference was only one iteration at size 9×9, but it rose to eight iterations at size 25×25. Similarly, in hp −adaptivity the difference was one at size 9×9, but widened to twenty-four iterations at size 44×44. By contrast, in p −adaptivity [Figure 4.14(A)] the behavior was reversed so that with increasing matrix size, the difference between Jacobi and SSOR gradually decreased, and at size 15×15 SSOR even passed Jacobi by two iterations and also at the final step, when the size reached 16×16, the two preconditioners became equal.

Table 4.7: Iterative performance of preconditioned CG solvers for growing matrix sizes in p −adaptive mesh refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

				Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*p*	*El*	*Iter*	*Iter*	*Iter*	*Iter*
5^2	0	2	3	6	5	3	4
6^2	1st	3	3	7	6	4	4
7^2	2nd	4	3	8	7	5	5
8^2	3rd	5	3	9	8	6	5
9^2	4th	6	3	11	9	7	6
10^2	5th	7	3	12	10	9	6
11^2	6th	8	3	14	11	10	7
12^2	7th	9	3	18	16	12	7
13^2	8th	10	3	21	20	12	8
14^2	9th	11	3	26	24	14	8
15^2	10th	12	3	31	33	14	9
16^2	11th	13	3	37	37	17	9

Table 4.8: Iterative performance of preconditioned CG solvers for growing matrix sizes in h −adaptive mesh refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

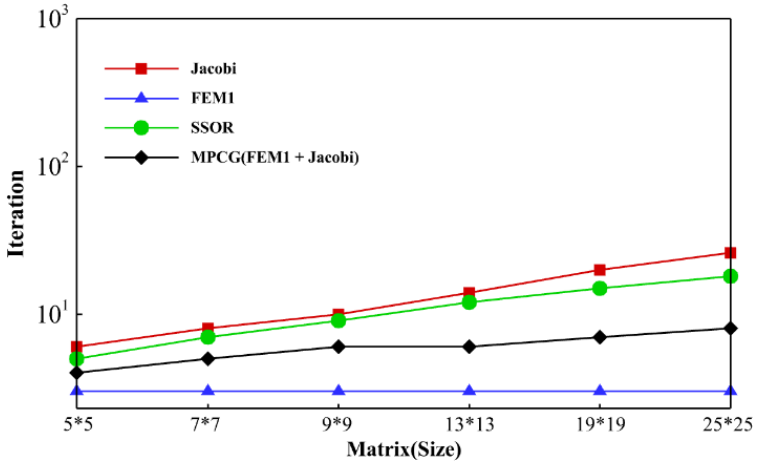
				Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*p*	*El*	*Iter*	*Iter*	*Iter*	*Iter*
5^2	0	2	3	6	5	3	4
7^2	1st	2	4	8	7	3	5
9^2	2nd	2	5	10	9	3	6
13^2	3rd	2	7	14	12	3	6
19^2	4th	2	9	20	15	3	7
25^2	5th	2	10	26	18	3	8

Table 4.9: Iterative performance of preconditioned CG solvers for growing matrix sizes in hp −adaptive mesh refinement in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

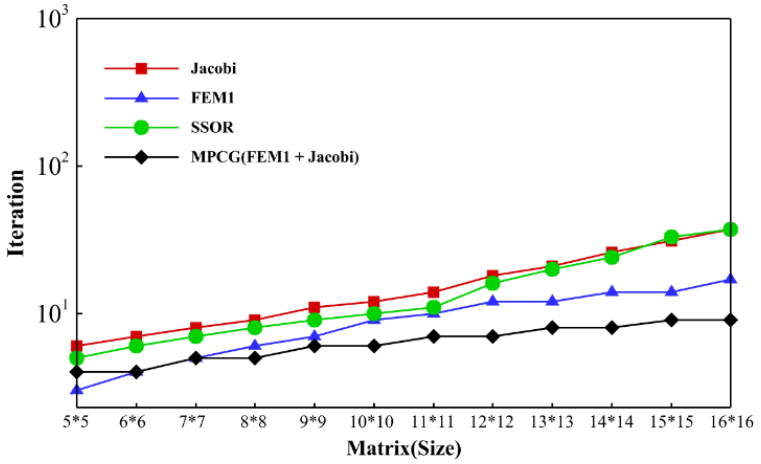
				Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*p*	*El*	*Iter*	*Iter*	*Iter*	*Iter*
5^2	0	2	3	6	5	3	4
9^2	1st	2	4	10	9	3	6
14^2	2nd	2	5	15	13	5	7
21^2	3rd	2	7	24	19	7	10
31^2	4th	2	9	39	30	8	13
44^2	5th	2	10	65	41	12	15

Figure 4.14 shows that the FEM1 was effective in all three adaptive strategies, with iteration counts generally ranging between three and seventeen and increasing only very slowly with problem size. In p −adaptivity [Figure 4.14(A)], the growth was somewhat oscillatory, while in hp −adaptivity [Figure 4.14(C)] the increase was smoother. The same qualitative behavior was observed for the combined (Jacobi+FEM1), and in fact, in p −adaptivity [Figure 4.14(A)] this combination yielded the lowest iteration counts overall. At very small sizes (e.g., 5×5 to 6×6), FEM1 alone produced the fewest iterations (see Table 4.7), but from that point onward the MPCG (Jacobi+FEM1) became the most efficient choice. In h −adaptivity, since all elements initially shared the same polynomial degree ($p = 2$), the iteration counts remained essentially flat as the matrix size increased [Figure 4.14(B)], a behavior consistent with the earlier h −uniform refinement results where increasing the number of elements did not lead to any change in iteration count (see Figure 4.9 and Table 4.4).

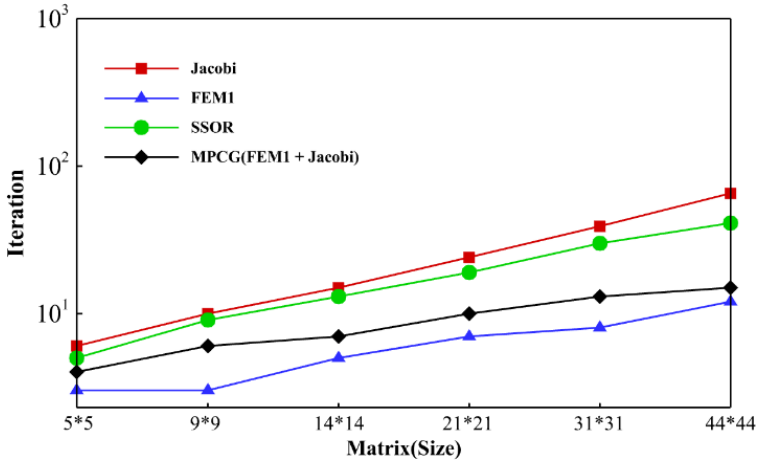
It is also worth noting that in both p −adaptivity and hp −adaptivity the FEM1 produced the lowest iteration counts, followed closely by the combined MPCG (Jacobi+FEM1). However, the number of iterations in hp −adaptivity was consistently smaller than in p −adaptivity, though hp −adaptivity generated a matrix more than three times larger in size. In other words, while the system size grew more rapidly in hp −adaptivity, the solver required fewer iterations to reach convergence compared to p −adaptivity. On the other hand, in hp −adaptivity Jacobi and SSOR showed the highest iteration counts across the entire adaptive sequence, with Jacobi in particular requiring substantially more iterations than any other method.



A)



B)



C)

Figure 4.14: Iteration counts for different preconditioners under adaptivity: (A) p −adaptivity, (B) h −adaptivity, and (C) hp −adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

Regarding the execution times reported in Tables 4.10, 4.11, and 4.12, a consistent trend can be observed across all three adaptive strategies. In general, the Jacobi preconditioner provided the shortest runtimes and can therefore be regarded as the most efficient in terms of time. Following Jacobi, the SSOR occupied second place in most cases, although there are some important exceptions at the final adaptation steps. In the case of hp −adaptivity, during the fifth as the final adaptation, where the matrix size reached 14×14, SSOR lost its second-place position and was overtaken by the combined preconditioners MPCG (Jacobi+FEM1). At this point, the runtime for SSOR was larger than the runtime for MPCG by a factor of approximately 1.6 (see Table 4.12). A similar effect was also visible in p −adaptivity, although it was much less significant. At the final adaptation step, corresponding to matrix size 16×16, SSOR again dropped to third place, but the difference compared to MPCG (Jacobi+FEM1). was only around 1.06× (Table 4.10). This shows that while Jacobi remains the fastest in almost all cases, there are situations where the relative order of the other preconditioners changes depending on the adaptation method and the matrix size.

Table 4.10: Runtimes of different preconditioners with increasing matrix size in p −adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

				Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*p*	*El*	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)
5^2	0	2	3	3.08E-05	3.35E-05	2.18E-03	2.02E-04
6^2	1st	3	3	3.25E-05	1.08E-04	2.61E-03	2.52E-04
7^2	2nd	4	3	3.71E-05	5.67E-05	2.61E-03	3.06E-04
8^2	3rd	5	3	3.78E-05	8.90E-05	2.57E-03	2.46E-04
9^2	4th	6	3	1.03E-04	1.15E-04	2.98E-03	3.73E-04
10^2	5th	7	3	4.06E-05	1.01E-04	3.11E-03	3.33E-04
11^2	6th	8	3	4.53E-05	1.31E-04	3.40E-03	4.03E-04
12^2	7th	9	3	5.21E-05	1.95E-04	3.66E-03	4.07E-04
13^2	8th	10	3	5.66E-05	2.41E-04	3.84E-03	5.00E-04
14^2	9th	11	3	7.03E-05	4.67E-04	4.56E-03	5.68E-04
15^2	10th	12	3	8.85E-05	6.18E-04	4.57E-03	6.23E-04
16^2	11th	13	3	9.62E-05	6.33E-04	4.63E-03	5.90E-04

Looking closely at p −adaptivity, the runtime increased with matrix size in a very regular and predictable manner. The growth was almost linear with a very small slope, approximately 1.2 overall, indicating that the runtime penalty for larger systems was

modest. In contrast, the behavior of hp −adaptivity was more irregular and less predictable. [Figure 4.15(C)] illustrates that SSOR presents a clear upward trend with increasing matrix size, whereas the combined preconditioner MPCG (Jacobi+FEM1) actually shows a downward trend over the adaptive sequence.

At the very smallest systems (5×5 to 9×9), MPCG (Jacobi+FEM1) was the worst-performing method in terms of time, but as the system grew larger, its performance improved significantly. By the last adaptation step, when the system size reached 44×44, the runtime of MPCG (Jacobi+FEM1) had decreased compared to the first adaptation, with an overall reduction of about 64% (see Table 4.12). This is a remarkable observation, since normally one expects runtimes to increase monotonically with system size.

The improvement suggests that the interaction between the FEM1 component of the preconditioner and the hp −adaptive structure of the mesh provided progressively better efficiency as the mesh was refined. FEM1, in contrast, consistently produced the largest runtimes across all three adaptive strategies, and in this sense, it can be considered the least efficient when runtime is the main criterion.

The only notable exception was in hp −adaptivity at very small system sizes (5×5 to 9×9), where MPCG (Jacobi+FEM1) initially performed worse than FEM1. However, this was quickly reversed as the system size increased, and for all larger problems FEM1 remained the slowest option.

Table 4.11: Runtimes of different preconditioners with increasing matrix size in h −adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

				Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*p*	*El*	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)
5^2	0	2	3	4.04E-05	4.28E-05	1.98E-03	5.09E-04
7^2	1st	2	4	2.91E-05	9.78E-05	5.74E-03	2.93E-04
9^2	2nd	2	5	3.90E-05	8.78E-05	2.75E-03	3.14E-04
13^2	3rd	2	7	4.58E-05	1.71E-04	3.65E-03	3.49E-04
19^2	4th	2	9	6.53E-05	2.56E-04	5.23E-03	4.69E-04
25^2	5th	2	10	8.02E-05	3.94E-04	6.58E-03	6.72E-04

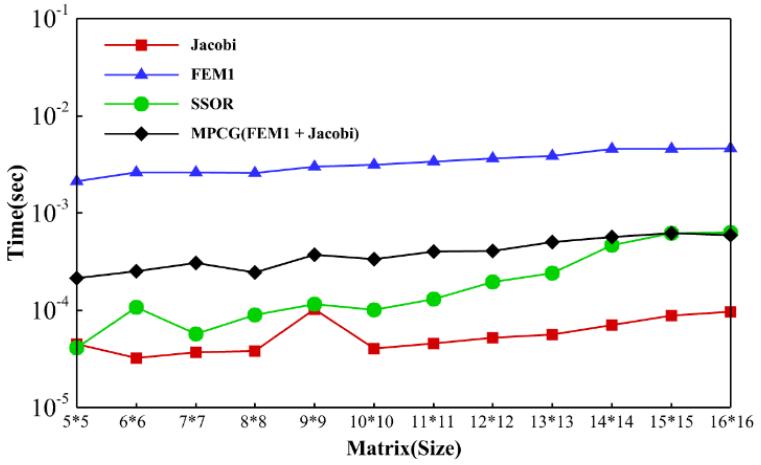
In h −adaptivity, the behavior of the runtimes was the smoothest and most uniform among the three strategies. As shown in Figure 4.15(B), the curves increase steadily with system size but without the sudden fluctuations that appear in p −adaptivity and especially hp −adaptivity. The only irregularity in h −adaptivity occurred for FEM1 at matrix size 7×7, where a small sudden jump in runtime was observed. Apart from this single anomaly, the growth remained gentle and predictable. Among the four preconditioners

tested in $h-$adaptivity, SSOR displayed the steepest increase in runtime. According to Table 4.11, the average growth ratio for SSOR was approximately 1.5, while for the other three preconditioners the ratios were all below 1.3, confirming the more rapid growth of SSOR compared to the alternatives. When comparing the three adaptive strategies in terms of the time order of magnitude, further differences become clear. For p-adaptivity and $h-$adaptivity, the orders of magnitude are mostly in the range10^{-5} to 10^{-3}, reflecting relatively low runtime growth. For $hp-$adaptivity, however, the time orders fall between 10^{-4} and 10^{-2}, which indicates that overall, the $hp-$adaptive strategy required more computational time than the other two methods.

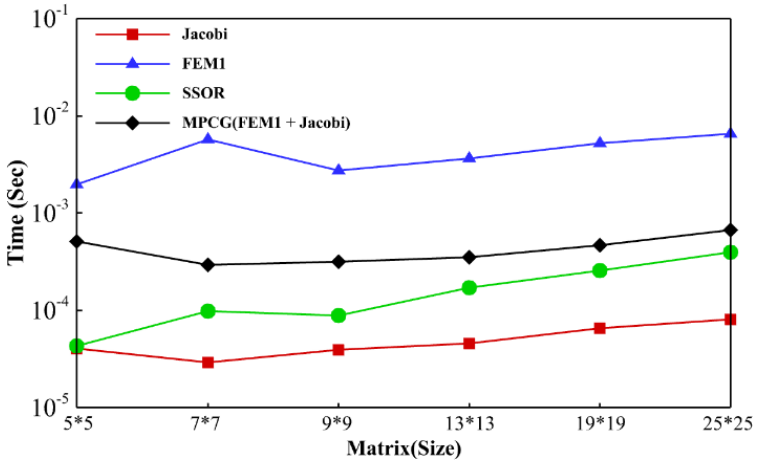
Table 4.12: Runtimes of different preconditioners with increasing matrix size in $hp-$adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

			Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*El*	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)
5^2	0	3	5.89E-04	1.31E-03	4.61E-03	6.61E-03
9^2	1st	4	1.19E-04	1.85E-04	3.08E-03	5.39E-03
14^2	2nd	5	1.68E-04	2.94E-04	4.51E-03	8.96E-04
21^2	3rd	7	5.67E-04	1.36E-03	6.77E-03	3.45E-03
31^2	4th	9	1.36E-04	1.09E-03	8.14E-03	1.51E-03
44^2	5th	10	3.04E-04	3.87E-03	1.25E-02	2.38E-03

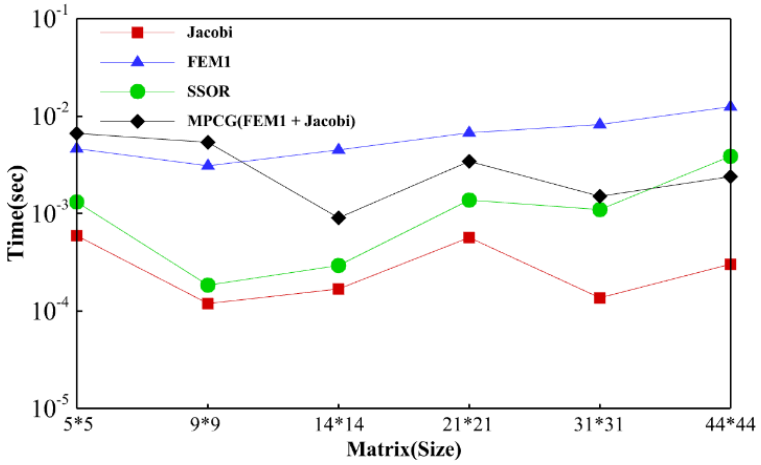
This trend can also be clearly observed in Figure 4.15, where (C), corresponding to $hp-$adaptivity, shows the steepest slope among the three strategies. Moreover, while in (A) and (B) which are related to $p-$adaptivity and $h-$adaptivity respectively all preconditioners follow a relatively parallel growth trend with only small differences, in $hp-$adaptivity the results are more varied and less predictable. For example, the SSOR shows a steadily increasing curve, while MPCG (Jacobi+FEM1) follows a generally decreasing curve, something that is not seen in the other two strategies. Finally, when comparing Figures 4.15(A) and (B), it can be seen that $h-$adaptivity demonstrates the smoothest and most regular behavior. The increase in runtime is slower, and the irregular oscillations that appear in $p-$adaptivity and $hp-$adaptivity are absent. This makes $h-$adaptivity somewhat easier to interpret and to predict in terms of computational cost. The only small irregularity, as already mentioned, is FEM1 at size 7×7. Apart from that, all four preconditioners increase smoothly in runtime with increasing system size. Among them, SSOR is the most expensive after Jacobi, with a noticeably steeper slope.



A)



B)



C)

Figure 4.15: Runtime comparison of Jacobi, FEM1, SSOR, and MPCG (FEM1+Jacobi) preconditioners with increasing matrix size under adaptivity: (A) p −adaptivity, (B) h −adaptivity, and (C) hp −adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

According to the condition numbers reported in Tables 4.13, 4.14 and 4.15, it is clear that the conditioning of the system worsens with each adaptive step, regardless of the type of adaptivity. This outcome is expected, since every adaptation effectively enlarges the global stiffness matrix and makes it "fatter," which naturally drives the condition number upward. This behavior is also visible in Table 4.16, which illustrates the sparsity of the global stiffness matrix at size 9×9 for the three adaptive strategies. In Table 4.16 (A), corresponding to p −adaptivity, the sparsity pattern is more uneven, with a denser region in the lower right corner. Based on this distribution one would expect the condition number to be higher compared to the other two cases, and indeed Table 4.13 confirms a value of 7.11E+02. In contrast, the values for h −adaptivity and hp −adaptivity reported in Tables 4.14 and 4.15 are 8.95E+01 and 1.99E+02, respectively. In Figure 4.17 (B) and (C), corresponding to $h-$ and hp −adaptivity, the sparsity structures are more balanced overall, however, the hp −adaptive case is still noticeably less uniform than the h −adaptive case. This reduced uniformity directly explains why the condition number for hp −adaptivity (1.99E+02) is higher than for h −adaptivity (8.95×10^1), even though both are significantly better conditioned than the p −adaptive case.

Table 4.13: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and unpreconditioned system (Stiffness matrix) under p −adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

				UP	Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*p*	*El*	Condition Number	Condition Number	Condition Number	Condition Number	Condition Number
5^2	0	3	3	1.80E+01	1.80E+01	3.39E+00	6.00E+00	2.22E+00
6^2	1st	4	3	3.97E+01	2.93E+01	4.77E+00	9.64E+00	1.04E+01
7^2	2nd	5	3	1.03E+02	7.08E+01	8.88E+00	1.60E+01	3.75E+01
8^2	3rd	6	3	2.65E+02	1.95E+02	1.87E+01	3.35E+01	1.50E+02
9^2	4th	7	3	7.11E+02	5.83E+02	4.73E+01	1.01E+02	4.49E+02
10^2	5th	8	3	2.03E+03	1.96E+03	1.44E+02	3.88E+02	2.14E+03
11^2	6th	9	3	6.16E+03	7.55E+03	5.46E+02	1.59E+03	9.13E+03
12^2	7th	10	3	1.97E+04	3.32E+04	2.49E+03	7.86E+03	6.19E+04
13^2	8th	11	3	6.61E+04	1.62E+05	1.29E+04	4.04E+04	3.36E+05
14^2	9th	12	3	2.28E+05	8.63E+05	7.22E+04	2.26E+05	2.47E+06
15^2	10th	13	3	8.07E+05	4.86E+06	4.26E+05	1.27E+06	1.46E+07
16^2	11th	14	3	2.90E+06	2.86E+07	2.61E+06	7.57E+06	1.09E+08

When comparing the condition number orders of magnitude, the differences between strategies become stronger. In p −adaptivity the values for the largest systems rise between 10^6 and 10^8, whereas in h −adaptivity the order remains between 10^1 and 10^3, and in hp −adaptivity it falls between $10^2$and 10^4. Although this suggests that p −adaptivity leads to much worse conditioning, for example in Table 4.7 that shows that at size 9×9, with Jacobi, only eleven iterations were required for the solver to be converged, just one more than ten iterations in $h-$ and hp −adaptivity.
The apparently small difference in iteration counts, despite a condition number almost ten times larger, is explained by the eigenvalue distributions reported in Table 4.17: at this matrix size the spectra of all three strategies appear broadly similar, so the convergence is influenced more by the clustering of eigenvalues than by the raw condition number itself.

Table 4.14: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and unpreconditioned system (Stiffness matrix) under h −adaptivity in 1D, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

				UP	Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*p*	*El*	Condition Number	Condition Number	Condition Number	Condition Number	Condition Number
5^2	0	2	3	1.80E+01	1.80E+01	3.39E+00	6.00E+00	2.22E+00
7^2	1st	3	4	4.18E+01	3.44E+01	5.69E+00	1.03E+01	1.20E+01
9^2	2nd	4	5	8.95E+01	5.34E+01	8.26E+00	2.03E+01	3.06E+01
13^2	3rd	5	7	2.31E+02	1.53E+02	2.28E+01	5.04E+01	2.74E+02
19^2	4th	6	9	5.62E+02	3.82E+02	5.13E+01	1.31E+02	1.09E+03
25^2	5th	7	10	1.19E+03	5.92E+02	7.80E+01	2.94E+02	3.24E+03

The trends in Figure 4.16 become clearer when tracking the growth of the condition number as the matrix size increases. In (A) corresponding to p −adaptivity, the slope of the increase is noticeably steeper than in (B) and (C) which are corresponding to $h-$ and $hp-$ adaptivity respectively. Within (A), over the range matrix size from 5×5 to 9×9, the unpreconditioned system (stiffness matrix) presents the largest condition numbers. Beyond this range, however, still in (A), Jacobi and MPCG (Jacobi+FEM1) overtake the unpreconditioned system and produce larger values. Up to size 14×14, again in (A), FEM1 achieves the lowest condition numbers compared to unpreconditioned system, but its curve then rises more rapidly and eventually surpasses it. Generally, (A) shows that although the preconditioners reduce the condition number, their growth rates with refinement can still exceed that of the unpreconditioned system. More specifically, the slope for the unpreconditioned system curve in (A) lies around 2.5–3.5, whereas the slopes

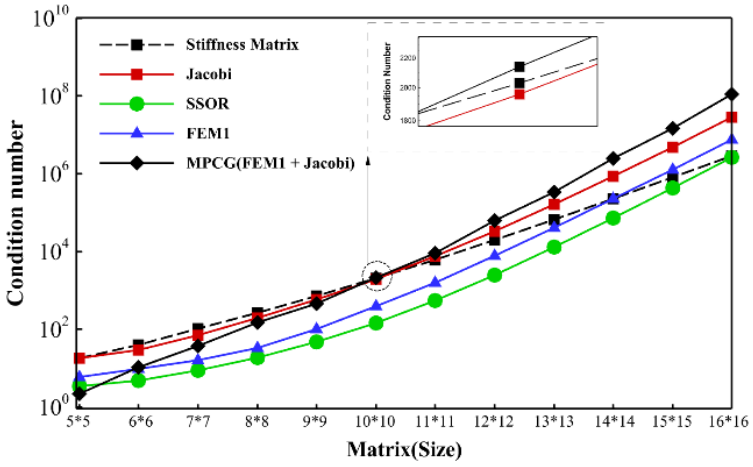
of the preconditioned curves are generally steeper (> 2.5–3.5), with the important exception of MPCG (Jacobi+FEM1), which maintains a more favorable trend.

Table 4.15: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and the unpreconditioned system (Stiffness matrix) under hp −adaptivity, starting from an initial mesh of 3 elements with polynomial degree $p = 2$.

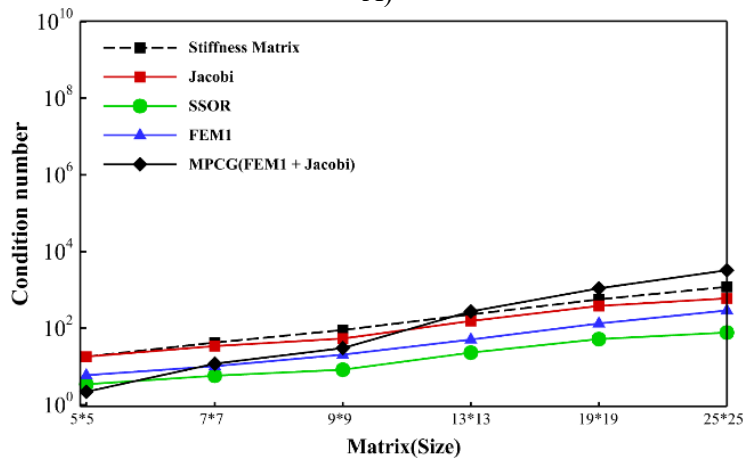
			UP	Jacobi	SSOR	FEM1	MPCG
Size of matrix	*Ad*	*El*	Condition Number	Condition Number	Condition Number	Condition Number	Condition Number
5^2	0	3	1.80E+01	1.80E+01	3.39E+00	6.00E+00	2.22E+00
9^2	1st	4	6.46E+01	6.53E+01	9.43E+00	1.36E+01	2.34E+01
14^2	2nd	5	1.99E+02	2.05E+02	2.45E+01	2.87E+01	1.65E+02
21^2	3rd	7	5.96E+02	6.34E+02	6.55E+01	6.31E+01	8.73E+02
31^2	4th	9	1.84E+03	2.05E+03	1.93E+02	1.44E+02	5.77E+03
44^2	5th	10	5.88E+03	6.93E+03	6.35E+02	3.43E+02	3.52E+04

In (B), the growth of the condition number with matrix size is nearly linear whereas (C) presents a similar overall trend, but with a slightly steeper slope. A closer comparison highlights several differences. In h −adaptivity, Jacobi consistently yields smaller condition numbers than the unpreconditioned system. By contrast, in hp −adaptivity, Jacobi produces values that are negligibly larger which are around 1.04 times higher than the unpreconditioned system. For the FEM1, both strategies show the same favorable behavior, the condition number remains below the unpreconditioned system baseline throughout the entire adaptive process. The MPCG (Jacobi+FEM1) shows mixed behavior. In h −adaptivity it produces smaller condition numbers than the unpreconditioned system (Stiffness Matrix) up to a matrix size of 13×13, beyond this point it exceeds the unpreconditioned system, although its slope of increase is noticeably gentler than in the earlier range. In hp −adaptivity, MPCG (Jacobi+FEM1) stays below the unpreconditioned system curve up to 14×14, but beyond that range it reverses and remains above for the rest of the refinement. A comparison between the FEM1 and SSOR highlights another key point in (C). In the lower matrix-size range (5×5 to 9×9), SSOR yields smaller condition numbers. However, Table 4.9 shows that SSOR nevertheless requires more iterations to converge. The reason is evident when looking specifically at the matrix size 9×9, Table 4.15 reports condition numbers of 1.36E+01 for FEM1 and 9.43E+00 for SSOR while Table 4.17 shows that under FEM1 the eigenvalues are grouped into four distinct clusters, whereas under SSOR they are spread more uniformly, with only a minor grouping near the upper bound (λ_{max}). Thus, although the SSOR achieves the smaller condition number, its uniform spectrum makes it harder for CG to reduce the error efficiently, whereas the clustered spectrum under FEM1 allows faster convergence with fewer iterations. As the

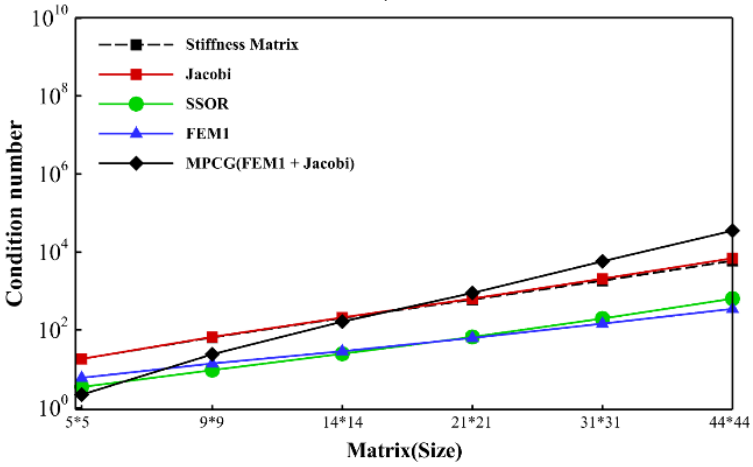
matrix size increases beyond 14×14, FEM1 not only retains this clustering advantage but also begins to produce smaller condition numbers than SSOR preconditioner.



A)



B)



C)

Figure 4.16: Variation of condition numbers with polynomial degree for preconditioned and unpreconditioned CG solvers under adaptivity, starting from 3 elements of polynomial degree 2: (A) p −adaptivity, (B) h −adaptivity, and (C) hp −adaptivity for 1D.

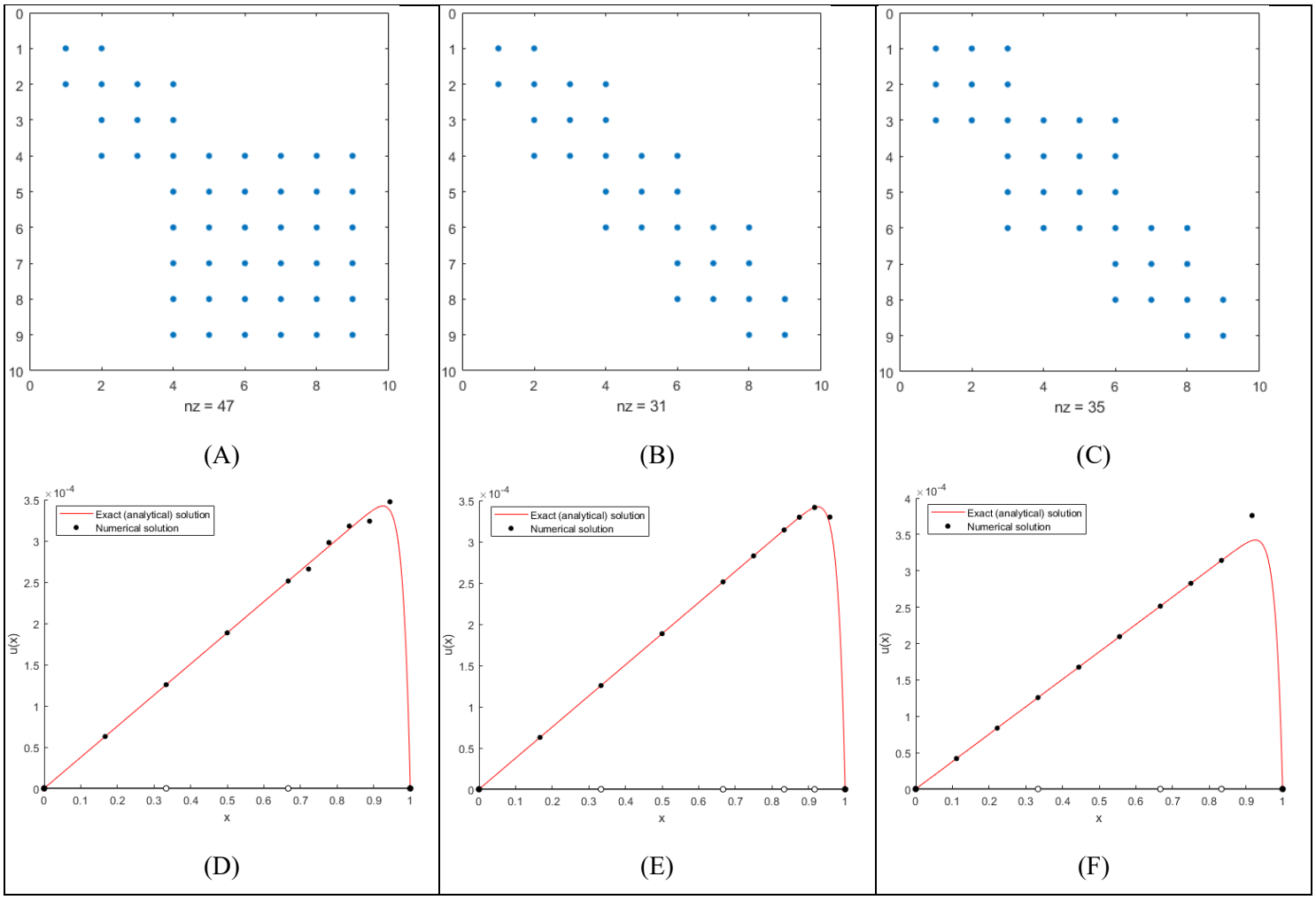


Figure 4.17: Sparsity patterns and corresponding solutions for adaptive refinements with a 9×9 stiffness matrix. (A) p −adaptivity after the 4th refinement, with its corresponding solution shown in (D). (B) h −adaptivity after the 2nd refinement, with its corresponding solution shown in (E). (C) hp −adaptivity after the 1st refinement, with its corresponding solution shown in (F). The subfigures (D–F) display the numerical solutions compared against the exact (analytical) solution for the respective adaptive cases. All refinements start from an initial mesh of 3 elements with polynomial degree $p = 2$.

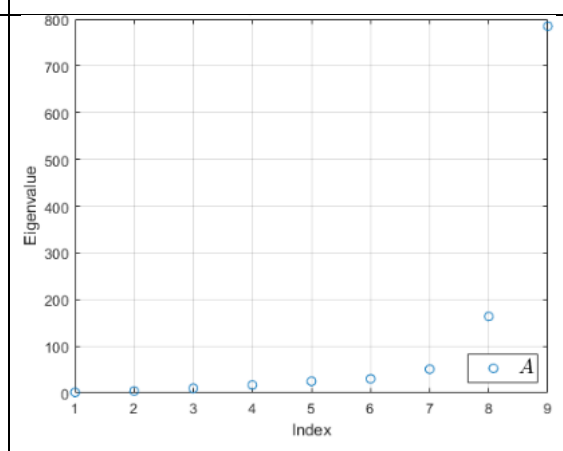
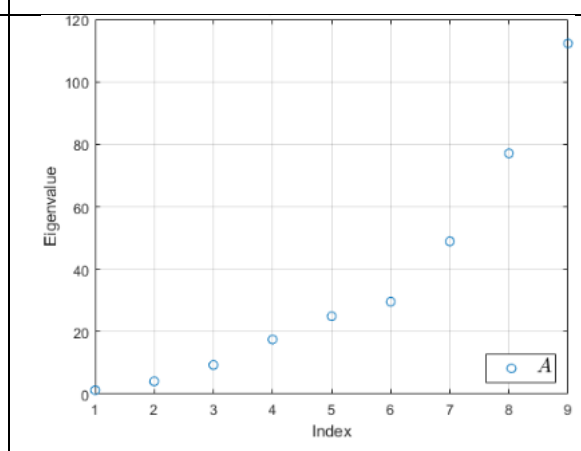
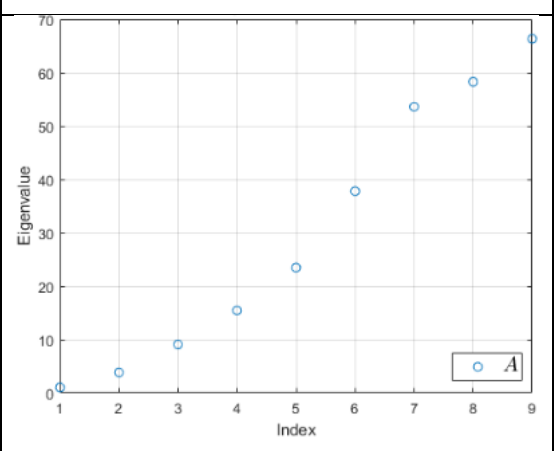
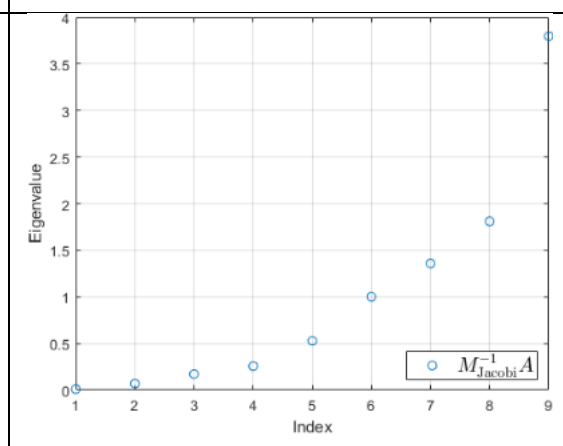
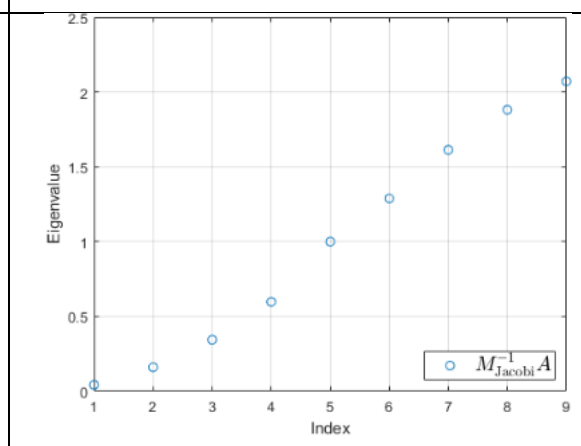
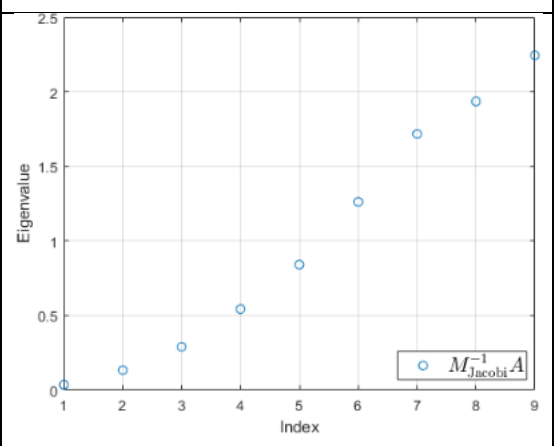
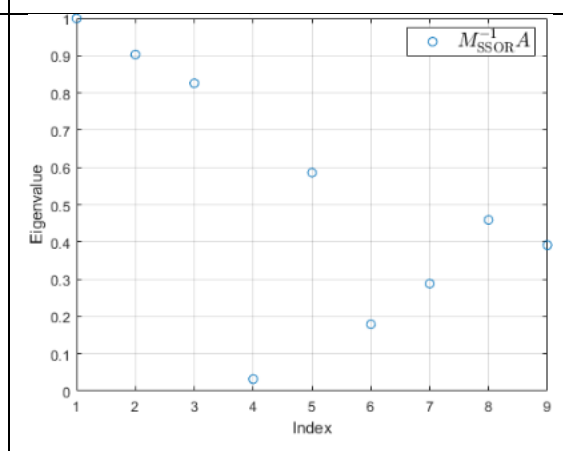
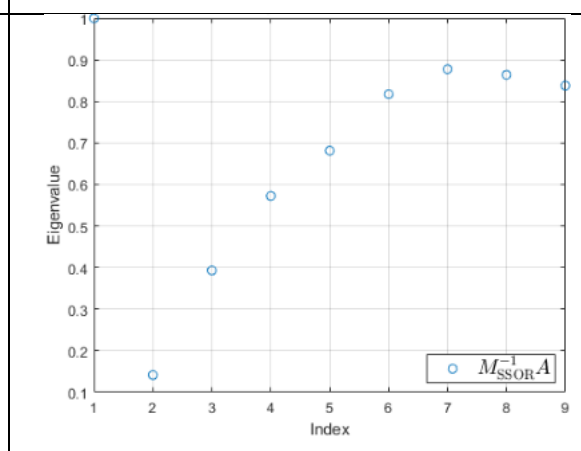
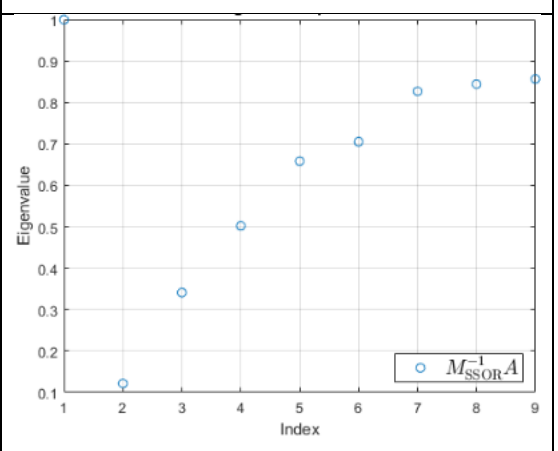
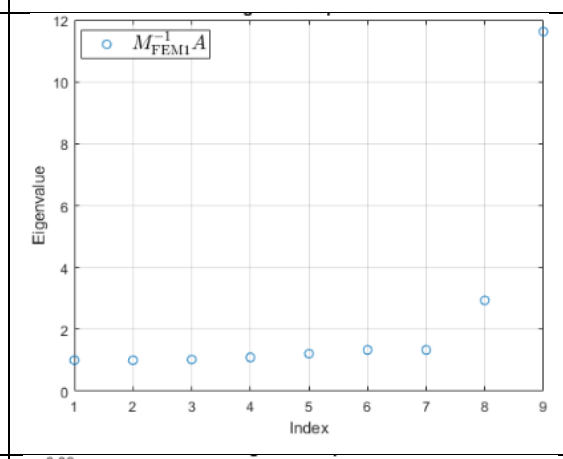
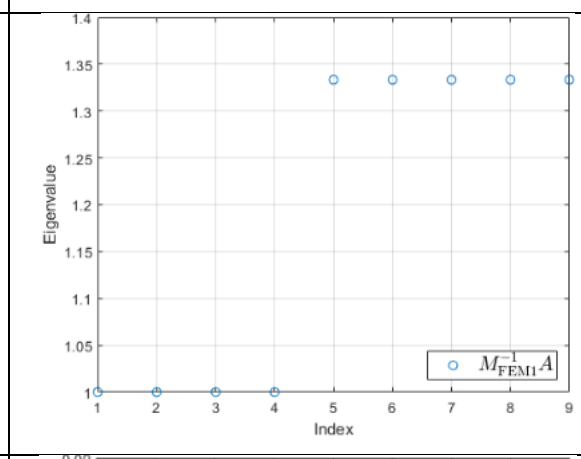
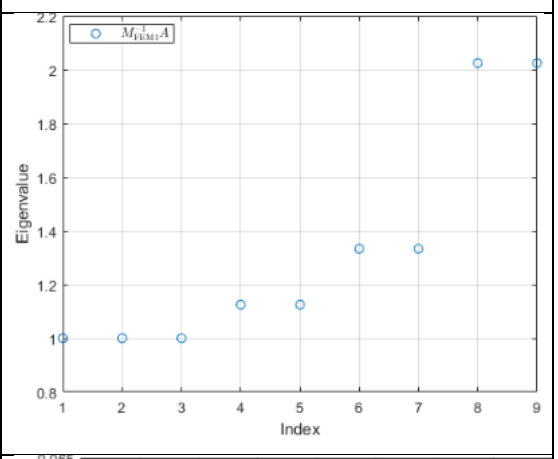
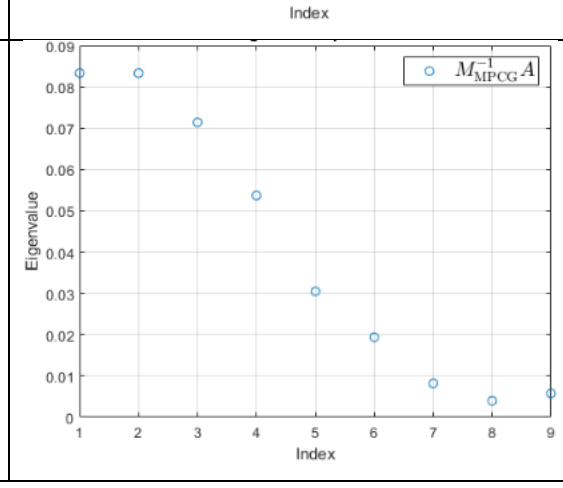
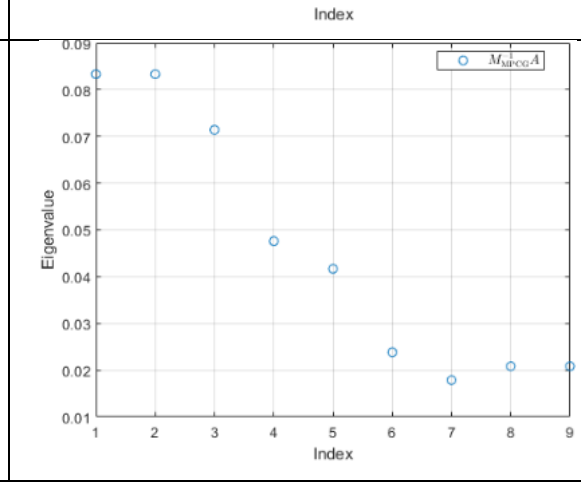
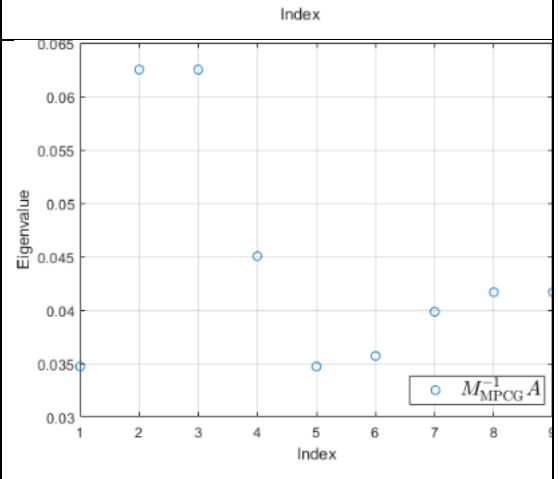
$M^{-1}A$	*Type of Adaptivity*		
	p −Adaptivity	h −Adaptivity	hp −Adaptivity
Unpreconditioned			
Jacobi			
SSOR			
FEM1			
MPCG			

Figure 4.18: Eigenvalue distributions of the preconditioned and unpreconditioned system (Stiffness Matrix) under different types of adaptivity in size of matrix 9×9. Columns correspond to (A) p −adaptivity, (B) h −adaptivity, and (C) hp −adaptivity, while rows show the spectra for the unpreconditioned system (Stiffness Matrix) and for the Jacobi, SSOR, FEM1, and MPCG preconditioners in 1D.

4.4 Problem Setup for 2D Cases

We now turn to the two-dimensional test problems in order to evaluate the performance of the iterative solvers under more complex settings. As before, the model problem is based on the steady-state Poisson equation $-\nabla^2 u(x,y) = f(x,y)$ in the domain $(x,y) \in [0,1]^2$, subject to Dirichlet boundary conditions on the entire boundary of the unit square $U(0,y) = U(1,y) = U(x,0) = U(x,1) = 0$, and $f(x,y)$ is provided in 0 . The exact form of the source term $f(x,y)$ is provided in 0, and it is chosen so that the analytical solution is known explicitly. In particular, the exact solution is given by $U(x,y) = -\big(x(1-x)^6 y(2-2y)(1-12y)\big)cos(6\pi x)$, which satisfies the prescribed boundary conditions. Throughout this section, we use the notation $U(x,y)$ to denote the analytical solution in order to distinguish it from the numerical approximations produced by the solvers. The discretization is carried out using the finite element method (FEM), starting from an initial mesh with six elements of equal size in both the $x-$ and $y-$directions, each of polynomial degree 3. A schematic for the setup of this model problem, including the domain and boundary conditions, is shown in Figure 3.1(A). This setup is used as the baseline for uniform and adaptive refinement, and for comparing different preconditioners. Illustrative examples of $p-$uniform and $h-$uniform refinement, together with the corresponding solutions of this model problem, are presented in Figures 4.17 and 4.18. Convergence of the iterative solvers was monitored until the relative residual norm Equation (3.112) was reduced to 10^{-10}. Error norms for the computed solutions were evaluated down to 10^{-6}.

Note_1: For brevity in the tables', abbreviated labels are used: *Ad* for Adaptation step, *p* for Polynomial order, *El* for number of Elements, and *Iter* for Iteration.

Note_2: To simplify notation, we drop the explicit word "*preconditioner"* and refer to methods only by their acronyms (e.g., SSOR, FEM1).

Note_3: All reported runtimes represent the average over seven independent solves, taken to reduce timing sensitivity and ensure a more stable performance comparison

4.5 2D Uniform Refinement Mesh

In the two-dimensional study, two uniform refinement strategies are examined, (i) $p-$refinement, where the polynomial degree of all elements is increased while keeping the mesh fixed, and (ii) $h-$refinement, where the number of elements in the mesh is increased while keeping the polynomial degree constant. In the latter case, refining the mesh decreases the element size equally in both directions, and under the uniform refinement considered here, all elements are subdivided evenly to maintain a balanced mesh. To evaluate solver performance, the exact solution is used directly, which reduces the computational cost and keeps the focus on analyzing the effect of different preconditioners.

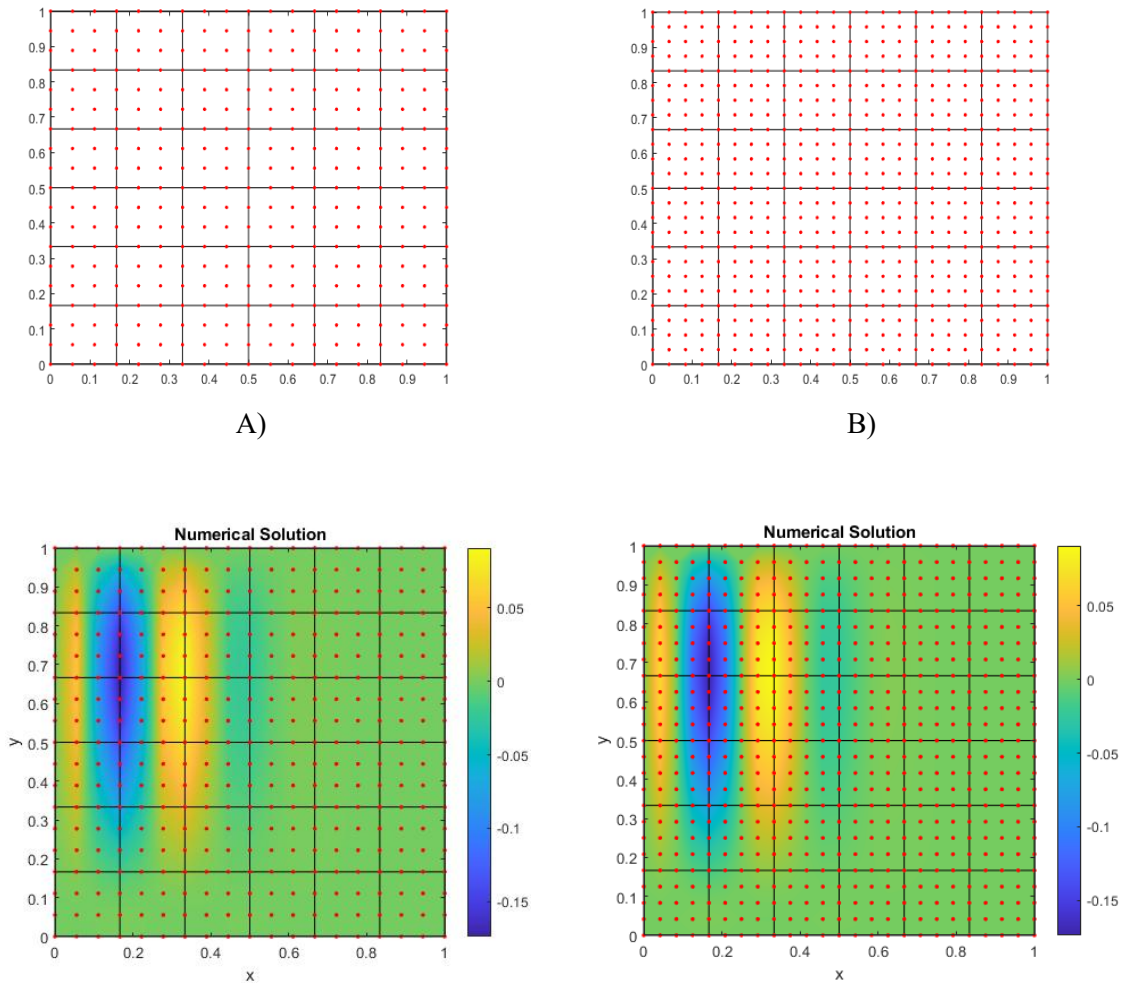


Figure 4.19: Example of p −uniform refinement on a two-dimensional uniform mesh together with the corresponding solutions of the model problem. (A) shows the mesh with polynomial degree 3, and (B) shows the mesh with polynomial degree 4, obtained by uniformly increasing the order of all elements by one. (C) and (D) display the numerical solutions for polynomial degrees 3 and 4, respectively.

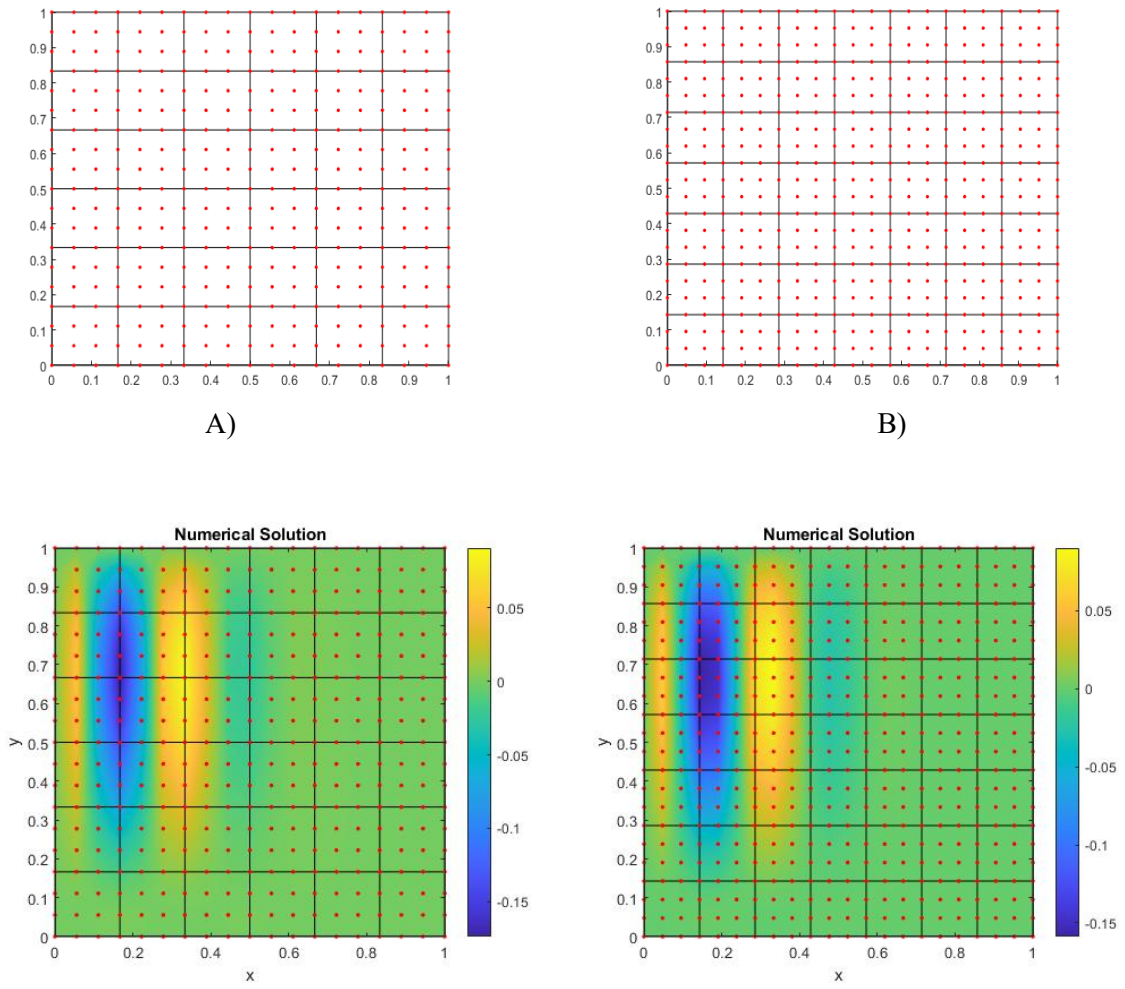


Figure 4.20: Example of h-uniform refinement on a two-dimensional mesh together with the corresponding solutions of the model problem. (A) shows the mesh with 6 elements in both the $x-$ and $y-$directions, each of polynomial degree 3. (B) shows the uniformly refined mesh with 7 elements in both directions, again with polynomial degree 3. (C) and (D) display the numerical solutions corresponding to the meshes in (A) and (B), respectively.

4.5.1 *P* - Uniform Mesh Refinement

As in the one-dimensional case discussed in Section 4.2.1, the two-dimensional $p-$refinement strategy follows the same overall procedure, but with node distribution handled in both spatial directions. The solver is advanced until the convergence condition in Equation (3.112) is satisfied. Afterward, Equation (3.69) is checked to determine whether the prescribed tolerance is achieved. If not, the polynomial degree of all elements is increased uniformly by one, and at this stage the nodes inside each element are redistributed in both $x-$ and $y-$directions so that their spacing remains uniform. This redistribution ensures that the new degrees of freedom are incorporated consistently across

the mesh, as illustrated in Figure 4.17 and explained in Section 3.4.2, and this process will be repeated until both equations mentioned above are satisfied, with Figure 3.17 reporting the first three stages of the overall p −adaptivity process.

According to the results reported in Table 4.16 for the 2D p −uniform refinement case, the behavior of the nine preconditioners is quite different. The p −multigrid method is the most stable, since after the smallest case (361×361), with seven iterations, it always converges in about 30 iterations, 30 V-cycle, independent of matrix size. Within the family of F4R, F4CR, and F3T preconditioners (see Section 3.11.3), F4R and F3T follow a very similar trend, while F4CR preconditioners require much higher iteration counts. At the largest system size (2401×2401), F4R and F3T need 174 and 170 iterations, but F4CR needs nearly seven times more. Jacobi shows the largest growth overall, while SSOR stays in the middle range.

Table 4.16: Iterative performance of preconditioned CG solvers for growing matrix sizes in p −uniform refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

				Jacobi	SSOR	F4R	F4CR	F3T	MPCG1	MPCG2	MPCG3	p_multigrid
Size of matrix	*Ad*	*p*	*El*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*
361^2	1	3	36	66	30	21	75	20	20	62	16	7
625^2	2	4	36	122	47	31	132	27	29	124	23	30
961^2	3	5	36	228	75	45	219	40	45	212	35	30
1369^2	4	6	36	448	121	69	379	62	70	340	54	30
1849^2	5	7	36	942	224	112	679	102	111	504	91	30
2401^2	6	8	36	2172	456	174	1239	170	174	662	137	30

These trends can also be seen clearly in Figure 4.21. The curves of F4R and F3T almost overlap, with a maximum difference of only 10 iterations at matrix size 1849×1849 (see Table 4.16). The line for F4CR, on the other hand, grows much faster and separates strongly from the other two. This behavior may look surprising when compared with results in Table 4.18, where the condition numbers of F3T and F4CR are reported as being much smaller than for most other preconditioners, and in Figure 4.23 their values are also very close to each other. One might expect this to result in fewer iterations for F4CR, but Table 4.17 shows that this is not the case. The reason can be explained by looking at the eigenvalue distributions in Figure 22 for the matrix size 1849×1849.

The spectrum for F3T shows a more compact and clustered distribution, with only a few distinct clusters, while in F4CR the clusters appear at higher parts of the spectrum and are more widely spread. F4R lies in between. Since CG convergence depends not only on the condition number but also on how tightly the eigenvalues are grouped, F3T gives fewer iterations than F4CR despite similar condition numbers. In other words, the better clustering and smaller number of clusters in F3T make it easier for CG to approximate with low-degree polynomials, while the wider spread of F4CR forces more iterations. The Jacobi curve sits at the top of Figure 4.12, confirming its poor performance, while SSOR remains between the better and worse groups.

The three combined preconditioners (MPCG1, MPCG2, and MPCG3) behave differently from their base versions. MPCG1 improves over F4R, MPCG3 improves over F3T, and even MPCG2 does better than F4CR alone, although at large sizes it is still far more expensive than the others. At the final matrix size, MPCG2 requires about 3.8 times more iterations than MPCG1 and 4.8 times more than MPCG3. Interestingly, its curve does not grow linearly but bends downward after size 1961×1961, and in the largest case it comes close to the SSOR curve.

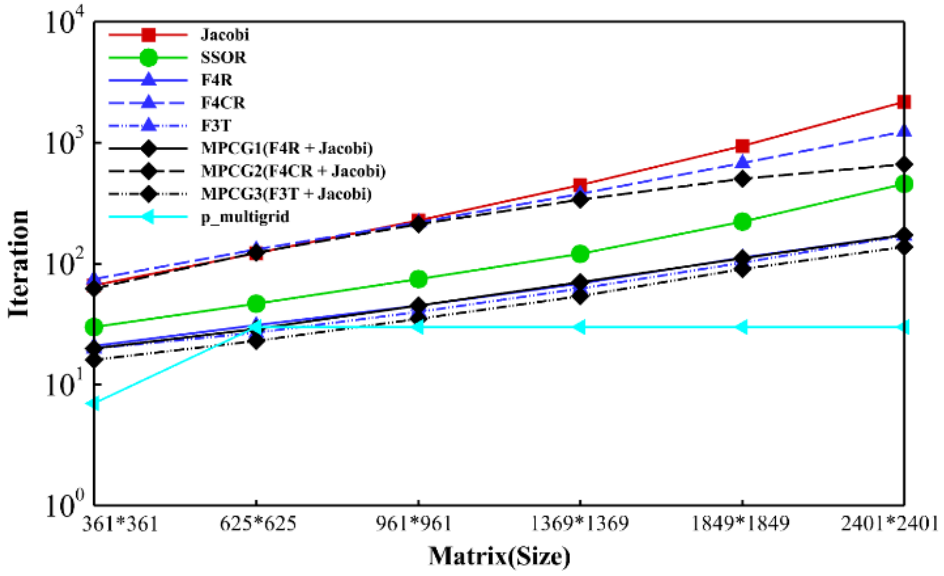


Figure 4.21: Iteration counts for different preconditioners under p −uniform refinement for 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

According to Table 4.19, the runtime of the p −multigrid preconditioner shows a very distinct pattern compared to the other methods. For small to medium system sizes, in the range from 361×361 to 625×625, its runtime is actually worse than almost all other preconditioners, with the exception of MPCG2. This behavior is also visible in Figure 4.20. However, as the mesh is refined further, the growth of runtime for p-multigrid is much slower than for the others, and by the time the largest system size (2401×2401) is reached, p-multigrid becomes the most efficient method overall. At this final size, it achieves a 12.3% reduction compared to Jacobi, which had been the fastest method up to

that point, corresponding to a factor of about 1.14× faster. This crossover can be explained by the way multigrid methods operate.

At small sizes, the extra overhead transferring between fine and coarse levels, combined with smoother applications, can make p −multigrid appear less competitive. But as the system grows, these costs are balanced by the efficiency of the multigrid cycle. High-frequency error components are quickly damped at the fine level, while low-frequency errors are reduced effectively on coarser levels, providing a good correction and a much better starting guess for the next iteration. As a result, the total runtime grows much more slowly with matrix size than for classical preconditioners.

Table 4.17: Runtimes of different preconditioners with increasing matrix size in p −uniform refinement for 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

				Jacobi	SSOR	F4R	F4CR	F3T	MPCG1	MPCG2	MPCG3	p_multigrid
Size of Matrix	*Ad*	*p*	*El*	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)
361^2	1	3	36	4.47E-03	3.51E-02	2.65E-02	2.80E-02	3.44E-02	3.37E-02	8.54E-02	3.44E-02	5.86E-02
625^2	2	4	36	2.22E-02	2.07E-01	1.63E-01	1.64E-01	2.25E-01	2.11E-01	7.32E-01	2.20E-01	2.46E-01
961^2	3	5	36	9.90E-02	1.28E+00	7.93E-01	7.54E-01	9.90E-01	1.25E+00	6.75E+00	1.18E+00	6.67E-01
1369^2	4	6	36	5.44E-01	7.63E+00	3.31E+00	3.17E+00	4.26E+00	5.37E+00	2.98E+01	5.09E+00	1.86E+00
1849^2	5	7	36	2.57E+00	5.50E+01	1.57E+01	1.33E+01	1.99E+01	2.40E+01	1.32E+02	2.41E+01	3.47E+00
2401^2	6	8	36	1.12E+01	3.26E+02	5.22E+01	4.52E+01	6.88E+01	7.97E+01	4.33E+02	8.76E+01	9.82E+00

In contrast, Jacobi and SSOR, which had the lowest runtimes at smaller sizes, both show steep slopes in Figure 20, meaning their runtimes increase rapidly as the matrix size grows. From size 961×961 onward, SSOR consistently ranks above most methods (except MPCG2), and at the largest size it comes very close to MPCG2, being only about 1.3× faster. The group of preconditioners F4CR, F4R, F3T, MPCG1, and MPCG3 all remain clustered in a narrow band, with reported runtimes between 2.65×10^{-2} and 8.76×10^{1} according to Table 4.17. Their curves in Figure 4.22 confirm this similarity, showing that their performance is relatively balanced compared to the extremes of Jacobi, SSOR, and p −multigrid.

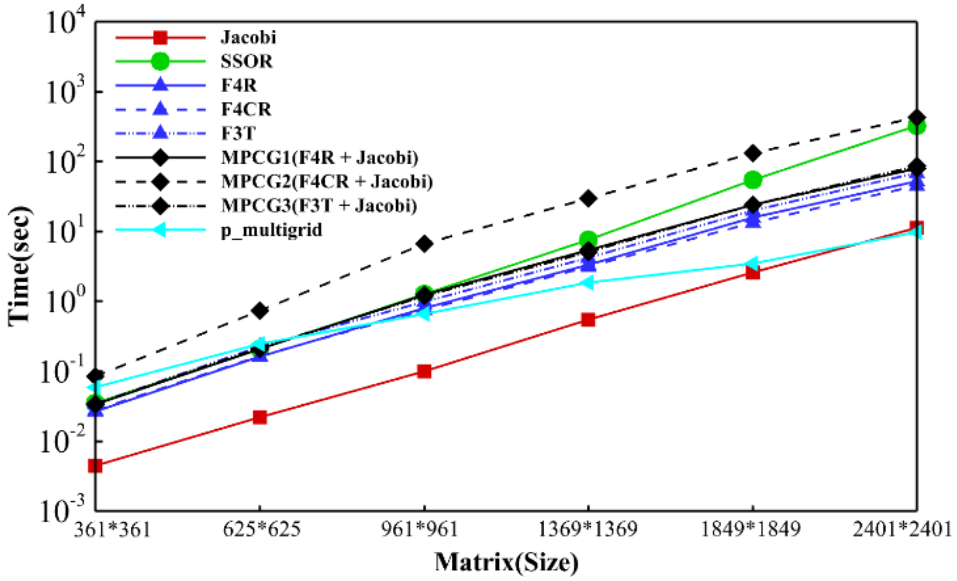


Figure 4.22: Runtime comparison of nine preconditioners with increasing matrix size in p −uniform refinement for 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

According to Table 4.18 and Figure 4.23, the condition numbers for the two preconditioners, F3T and F4CR, behave almost the same. Their variation is in the range from about 10^{0} to 10^{3}, and the two curves stay very close as the matrix size increases. What is interesting is that F3T, like F4R, is based on a first-order finite element approximation (see Section 3.10.3). The difference is that F3T is built from triangular elements, while F4R is based on square elements.

Since the whole problem is originally formed and solved using square elements, one would normally expect F4R to be closer in behavior to the system matrix and therefore to give a better approximation of its inverse. But this does not happen. Instead, F3T shows the best performance between the two, even though it is based on triangular elements. F4CR, on the other hand, are constructed from square elements but still shows similar condition number values to F3T.

The case of SSOR is also notable. Even though SSOR produced relatively high iteration counts (as discussed before), in terms of condition number it is placed in a very favorable

position. In fact, it is better than all preconditioners except F3T and F4CR. However, Figure 4.23 shows that as the matrix size increases, the slope of the SSOR curve grows faster than the others. Up to the size 1369×1369, the slope increases gradually, by about one unit each refinement step, starting from about 2.8 in the first interval (from 361 to 625). After 1369×1369, the growth becomes much sharper, and the slope increases by about three units at each refinement.

Table 4.18: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and the unpreconditioned system (Stiffness Matrix) under $p-$uniform refinement for 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

				UP	Jacobi	SSOR	F4R	F4CR	F3T	MPCG1	MPCG2	MPCG3
Size of Matrix	*Ad*	*p*	*El*	Cond#	Cond#	Cond#	Cond#	Cond#	Cond#	Cond#	Cond#	Cond#
361^2	1	3	36	2.70E+02	2.33E+02	3.07E+01	2.57E+02	7.72E+00	6.83E+00	5.53E+02	2.29E+02	5.75E+02
625^2	2	4	36	1.17E+03	1.01E+03	8.66E+01	1.15E+03	2.03E+01	1.51E+01	3.24E+03	1.25E+03	3.39E+03
961^2	3	5	36	5.21E+03	4.56E+03	2.94E+02	5.13E+03	6.12E+01	4.08E+01	8.76E+03	4.22E+03	9.19E+03
1369^2	4	6	36	2.62E+04	2.42E+04	1.31E+03	2.59E+04	2.24E+02	1.37E+02	6.90E+04	3.02E+04	7.23E+04
1849^2	5	7	36	1.54E+05	1.63E+05	9.58E+03	1.53E+05	1.00E+03	5.82E+02	1.59E+05	1.43E+05	1.66E+05
2401^2	6	8	36	1.07E+06	1.54E+06	9.56E+04	1.06E+06	5.47E+03	3.06E+03	5.57E+06	2.22E+06	5.82E+06

In the final interval, the slope reaches a value close to 10. Looking again at Table 4.18, it can be seen that for Jacobi, up to the size 1369×1369, the condition number is actually smaller than for the unpreconditioned case, although the difference is small. This is also visible in Figure 4.23. For example, at the size 961×961 the condition number for Jacobi is 4.56×10^3, while for the unpreconditioned matrix is 5.21×10^3. After this point, however, the trend reverses. With each refinement, the gap grows, and in the final case Jacobi is about 1.45 times larger than the unpreconditioned matrix which is 4.56×10^3 compared to 1.07×10^6.

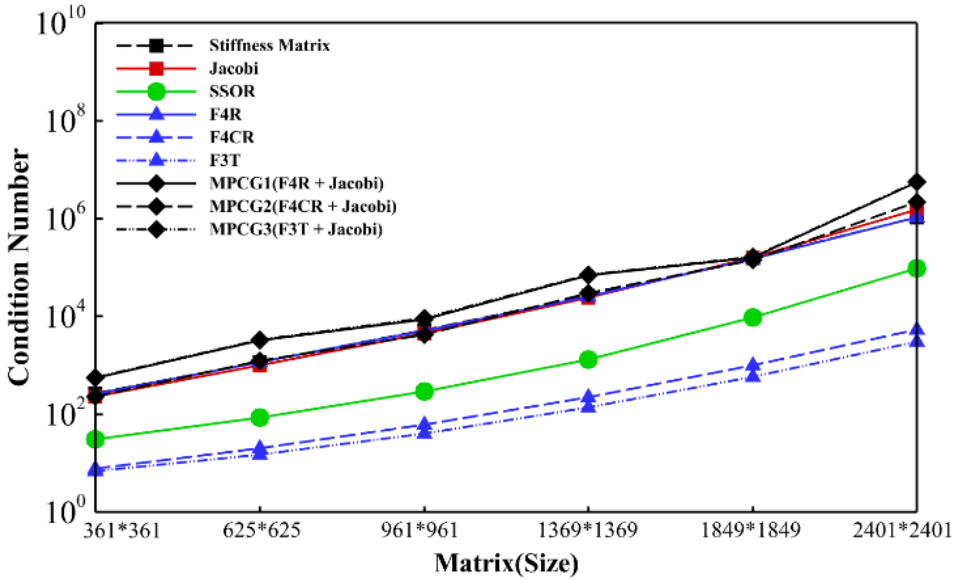


Figure 4.23: Variation of condition numbers with polynomial degree for preconditioned and unpreconditioned CG solvers under adaptivity, starting from 36 elements of polynomial degree 3 under p −uniform for 2D.

For the other preconditioners, the condition numbers all increase with matrix size, though with different slopes. The special case is MPCG1, which does not follow a simple monotonic increase. Instead, its growth is oscillatory, in some refinement steps the slope is smaller, and in others it is larger. At the matrix size 1849×1849, almost all preconditioners (except F3T and F4CR) reach very close values, clustered around 1.54×10^5. In this situation, the differences between them are minimal. The largest gap is seen with MPCG2, where the condition number of the unpreconditioned matrix is only 1.08 times larger than that of MPCG2, showing how close all the curves are at this stage.

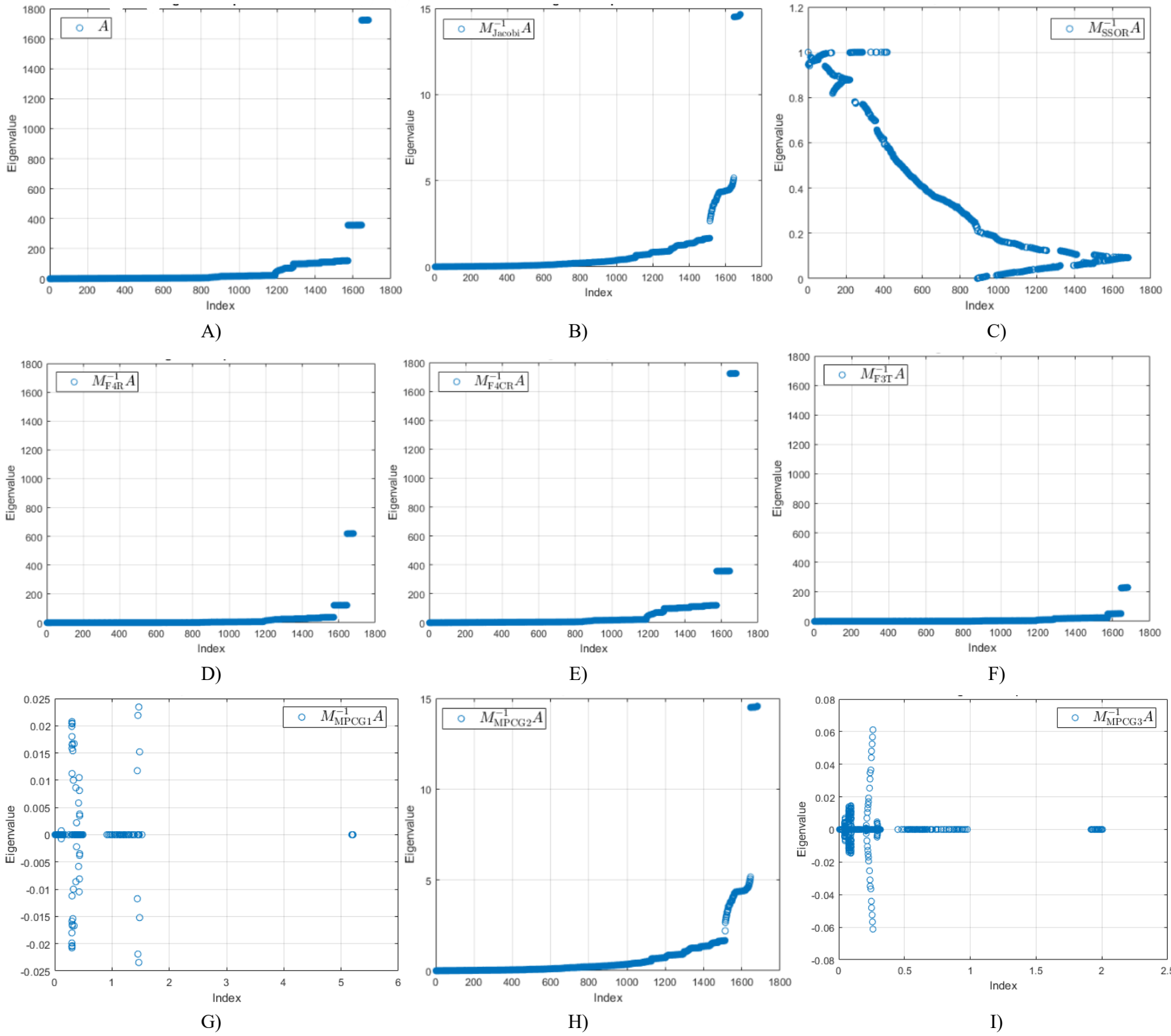


Figure 4.24: Eigenvalue distributions of the preconditioned and unpreconditioned system (Stiffness Matrix) under p −uniform of size 1849×1849 corresponding to fourth adaptation fin 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$. (A) (A) shows the unpreconditioned system A. (B) Jacobi preconditioned system $M_{Jacobi}{}^{-1}A$. (C) SSOR preconditioned system $M_{SSOR}{}^{-1}A$. (D) F4R preconditioned system $M_{F4R}{}^{-1}A$. (E) F4CR preconditioned system $M_{F4CR}{}^{-1}A$. (F) F3T preconditioned system $M_{F3T}{}^{-1}A$. (G) Multi-PCG1 preconditioned system $M_{MPCG1}{}^{-1}A$. (H) Multi-PCG2 preconditioned system $M_{MPCG2}{}^{-1}A$. (I) Multi-PCG3 preconditioned system $M_{MPCG3}{}^{-1}A$.

4.5.2 h - Uniform Mesh Refinement

As in the one-dimensional case discussed in Section 4.2.2, the two-dimensional h-refinement procedure follows the same overall principle. The solver is advanced until the convergence condition in Equation (3.112) is satisfied. Once this is achieved, Equation (3.69) is checked to determine whether the tolerance requirement is met. If it is satisfied, the process ends, and the solution is considered converged. Otherwise, the mesh is refined by subdividing the domain: one element is added in the x −direction and one element in the y −direction, so that the total number of elements grows uniformly in both spatial dimensions. Unlike the one-dimensional case, where ten elements were added after each refinement step, here the refinement proceeds in smaller increments, with exactly one element added per direction. Throughout this process the polynomial degree of the elements remains fixed at three, so only the number of elements changes. An illustration of this refinement pattern is provided in Figure 4.18, which shows how the mesh evolves under successive uniform refinements. It should be noted, however, that the computational cost of this procedure in two dimensions was extremely high. For this reason, the refinement process was carried out only up to 21 steps, beyond which the runtime became impractical. Even at this stage, the computation required more than one full day to complete, and Equation (3.69) was reduced to 5.36×10^6. This result demonstrates the prohibitively expensive nature of pure h −refinement in two dimensions.

According to Table 4.19, it can be seen that even though the mesh size grows very rapidly during refinement, the iteration remains below 350 in all cases, with the largest value occurring for the F4CR. This is a very different trend compared to the p −uniform refinement case (Table 4.16), where the Jacobi preconditioner required up to 2172 iterations at the largest system size. However, one must note that in h-uniform refinement the cost of assembling the global stiffness matrix (Equation 3.54) is very high, and the numerical integration of the right-hand side vector (F) is also much more expensive than in the p −uniform case. A closer look at Figure 4.25 shows that the preconditioners F4R, F3T, MPCG1, and MPCG3 display a slightly decreasing trend in iteration counts as the mesh size grows, while the other preconditioners, including p −multigrid, show a different behavior. In fact, p −multigrid exhibits three distinct phases: it stays constant at about 18 iterations, 18 V-cycles, until the fourth refinement, then increases gradually until around the sixteenth refinement, stabilizing again near 35 iterations, 35 V-cycles, and finally shows only a very small change from 35 to 36 after the eighteenth refinement. The initial growth phase has the steepest slope (about 2.4), but overall, the variation is minor compared to other preconditioners. This reflects a typical property of multigrid methods, where convergence is known to be almost independent of the mesh size or the polynomial degree. In practice, the iteration counts are not perfectly constant, but the overall reliability of the curve shows that the multigrid hierarchy is effectively controlling both high- and low-frequency error modes across refinements.

Table 4.19: Iterative performance of preconditioned CG solvers for growing matrix sizes in h −uniform refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

				Jacobi	SSOR	F4R	F4CR	F3T	MPCG1	MPCG2	MPCG3	p_multigrid
Size of matrix	*Ad*	*p*	*El*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*
361^2	1	3	36	66	30	21	75	20	20	62	16	7
484^2	2	3	49	77	33	21	86	20	20	80	16	9
625^2	3	3	64	88	38	20	93	19	19	93	16	12
784^2	4	3	81	98	42	20	104	19	20	104	16	15
961^2	5	3	100	108	46	20	113	19	20	120	16	17
1156^2	6	3	144	119	50	20	122	19	19	132	16	18
1369^2	7	3	168	130	54	20	132	18	19	144	16	18
1600^2	8	3	196	140	58	20	144	18	19	150	15	19
1849^2	9	3	225	151	62	20	157	18	19	161	15	21
2116^2	10	3	256	161	66	20	168	18	19	171	15	24
2401^2	11	3	289	171	69	19	180	18	18	177	15	26
2704^2	12	3	324	182	73	19	191	17	18	200	15	27
3025^2	13	3	400	192	77	19	202	17	18	203	15	29
3364^2	14	3	441	202	80	18	215	17	18	214	14	30
3721^2	15	3	484	213	84	18	230	17	18	226	14	31
4096^2	16	3	529	224	88	18	244	16	18	233	14	32
4489^2	17	3	576	234	92	18	258	16	17	241	14	35
4900^2	18	3	625	244	96	18	269	16	17	247	14	35
5329^2	19	3	679	255	99	18	286	16	17	259	14	36
5776^2	20	3	729	265	103	18	301	16	16	276	14	36
6241^2	21	3	784	275	107	18	316	16	16	286	14	36

Within the MPCG family, it is clear that MPCG2 always required the highest number of iterations, while MPCG1 and MPCG3 stayed in the narrow range of about 14 to 20 iterations. MPCG2, on the other hand, ranged between 62 and 286. A similar grouping appears in the F4R, F4CR, F3T family: F4R and F3T consistently required between 16 and 21 iterations, while F4CR was much higher, between 75 and 316. Finally, unlike in the p −uniform refinement case, Jacobi was not the worst performer here: in h −uniform refinement it occupied the third position, behind F4CR and MPCG2.

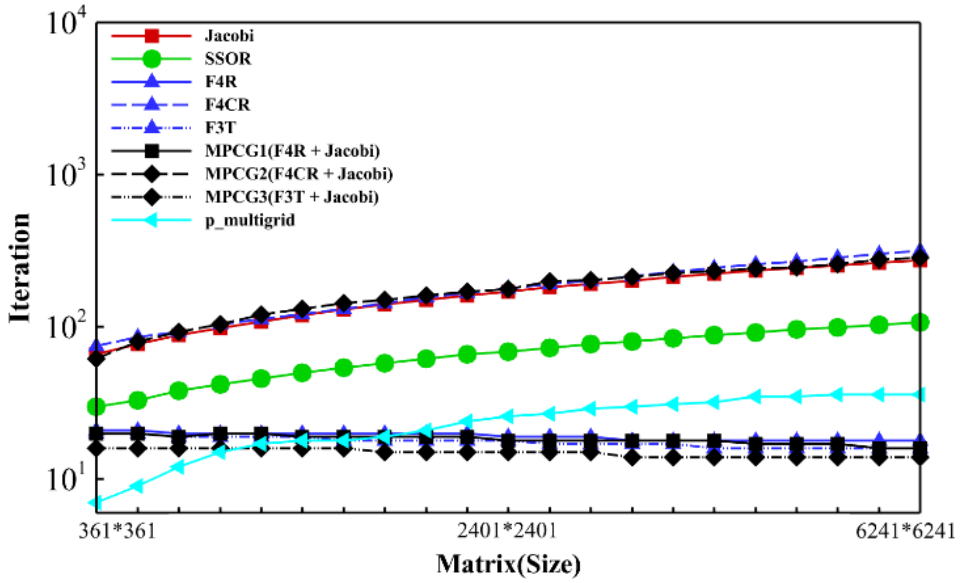


Figure 4.25: Iteration counts for different preconditioners under h −uniform refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

According to Table 4.20, the execution times during the refinement process range between the orders of 10^{-2} and 10^{2}. As shown in Figure 4.26, the preconditioners SSOR, F4R, F3T, MPCG1, and MPCG3 behave in a very similar manner. Their curves increase with almost the same slope, and the reported values remain close to each other. After the matrix size reaches around 2116×2116, the slope of these curves becomes slightly less steep, although the general trend of growth continues. Within this group, F4R stays consistently lower than the others, except at the final refinement level (matrix size 6241×6241), where MPCG1 drops below it with a runtime of 5.28E+01, as reported in Table 4.20.

Table 4.20: Runtimes of different preconditioners with increasing matrix size in h −uniform refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

				Jacobi	SSOR	F4R	F4CR	F3T	MPCG1	MPCG2	MPCG3	p_multigrid
Size of Matrix	*Ad*	*p*	*El*	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)
361^2	1	3	36	2.24E-02	3.40E-02	2.62E-02	2.98E-02	4.12E-02	3.24E-02	8.51E-02	4.83E-02	1.29E-01
484^2	2	3	49	9.65E-03	7.59E-02	6.38E-02	9.17E-02	9.12E-02	8.46E-02	2.95E-01	8.82E-02	1.19E-01
625^2	3	3	64	1.31E-02	1.26E-01	1.06E-01	1.72E-01	1.61E-01	1.32E-01	5.17E-01	1.45E-01	2.10E-01
784^2	4	3	81	2.20E-02	2.24E-01	1.82E-01	3.36E-01	2.28E-01	2.49E-01	1.01E+00	2.54E-01	3.77E-01
961^2	5	3	100	4.66E-02	4.13E-01	3.44E-01	7.11E-01	3.94E-01	5.02E-01	2.82E+00	5.03E-01	6.46E-01
1156^2	6	3	144	1.08E-01	6.92E-01	5.73E-01	1.30E+00	7.43E-01	7.61E-01	5.32E+00	8.47E-01	1.66E+00
1369^2	7	3	168	1.61E-01	1.16E+00	9.55E-01	2.43E+00	1.18E+00	1.20E+00	8.33E+00	1.30E+00	1.72E+00
1600^2	8	3	196	2.73E-01	2.00E+00	1.64E+00	4.57E+00	2.03E+00	1.97E+00	1.32E+01	2.11E+00	1.77E+00
1849^2	9	3	225	4.35E-01	3.27E+00	2.67E+00	8.30E+00	3.32E+00	3.12E+00	2.31E+01	3.29E+00	2.64E+00
2116^2	10	3	256	6.44E-01	4.88E+00	4.02E+00	1.34E+01	5.02E+00	4.53E+00	3.73E+01	4.86E+00	5.59E+00
2401^2	11	3	289	9.07E-01	7.07E+00	5.50E+00	2.04E+01	6.89E+00	6.21E+00	5.94E+01	6.70E+00	8.59E+00
2704^2	12	3	324	1.24E+00	9.29E+00	7.30E+00	2.90E+01	8.80E+00	8.24E+00	9.79E+01	9.06E+00	1.39E+01
3025^2	13	3	400	1.65E+00	1.23E+01	9.66E+00	4.07E+01	1.16E+01	1.08E+01	1.36E+02	1.19E+01	1.95E+01
3364^2	14	3	441	2.24E+00	1.65E+01	1.25E+01	5.75E+01	1.55E+01	1.43E+01	1.89E+02	1.50E+01	3.00E+01
3721^2	15	3	484	2.89E+00	2.10E+01	1.61E+01	8.14E+01	1.99E+01	1.82E+01	2.20E+02	1.94E+01	3.34E+01
4096^2	16	3	529	3.39E+00	2.48E+01	1.91E+01	1.01E+02	2.28E+01	2.15E+01	2.80E+02	2.29E+01	4.03E+01
4489^2	17	3	576	4.62E+00	3.13E+01	2.34E+01	1.37E+02	2.98E+01	2.78E+01	4.41E+02	2.89E+01	5.03E+01
4900^2	18	3	625	5.73E+00	4.20E+01	3.09E+01	1.79E+02	3.51E+01	3.31E+01	5.46E+02	3.86E+01	6.89E+01
5329^2	19	3	679	6.52E+00	5.06E+01	3.79E+01	2.38E+02	4.28E+01	4.03E+01	6.75E+02	4.66E+01	7.52E+01
5776^2	20	3	729	8.13E+00	6.15E+01	4.59E+01	3.16E+02	5.25E+01	4.69E+01	1.19E+03	5.71E+01	9.12E+01
6241^2	21	3	784	1.06E+01	7.66E+01	5.46E+01	3.97E+02	6.66E+01	5.28E+01	1.14E+03	6.42E+01	1.11E+02

The p −multigrid preconditioner shows a broadly similar behavior to this group but cannot be fully classified with them because of its different details. For example, Jacobi shows an initial drop in runtime when the matrix grows from 361×361 to 484×484, and only then begins to increase again. This behavior is not observed in the other preconditioners of the group. Although p −multigrid is generally higher than SSOR, F4R, F3T, MPCG1, and MPCG3, in refinements 7 to 9 its curve overlaps with them and even becomes the smallest at the eighth refinement (matrix size 1849×1849). After that point, however, it returns to its usual position with higher runtimes.

Among all eight preconditioners, F4CR and MPCG2 stand out as producing the largest runtimes. At the largest system size (6241×6241), their runtimes are reported as 37.4 and 107.6, respectively, compared to the Jacobi baseline. p −multigrid also shows an unusual characteristic at the very beginning: at the smallest matrix size (361×361), it already requires the largest runtime of all preconditioners, despite the fact that at this size the condition number of the system is relatively mild.

Throughout the entire process, Jacobi consistently achieves the smallest runtime. Its curve shows a slight dip after the first refinement and then takes about three refinement steps to climb back to the level of its initial runtime (see Figure 4.26). At that point, its runtime is about 1.2 times the initial value, with 2.20×10^{-2} compared to 2.24×10^{-2}at the start.

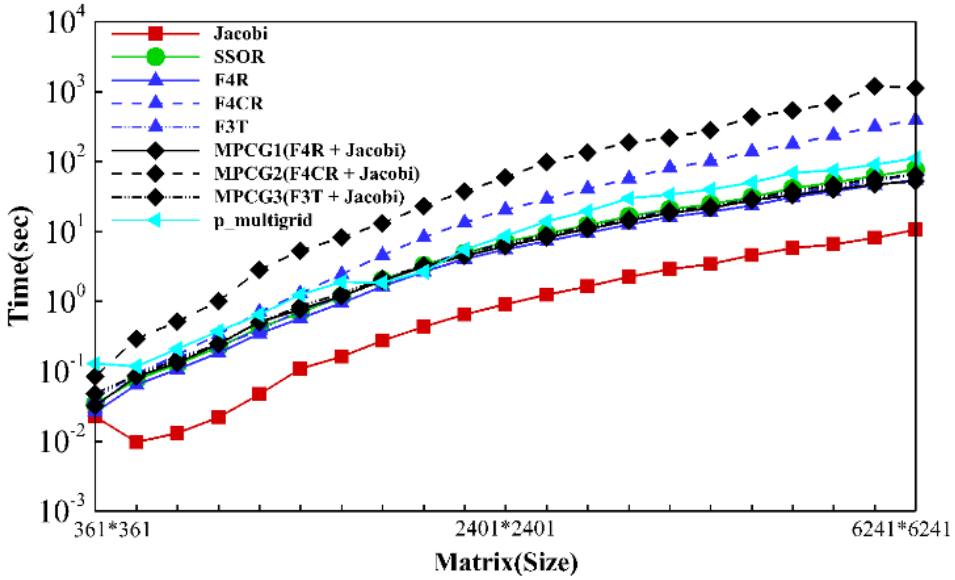


Figure 4.26: Runtime comparison of nine preconditioners with increasing matrix size in h −uniform refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

According to Table 4.21, the condition numbers reported during the refinement process show that the stiffness matrix, or the unpreconditioned case, increased from the order of 10^2 to 10^3. More precisely, the value in the last refinement step was about 18.85 times larger than in the first step (matrix size 361×361). Some preconditioners, however, showed much stronger growth. For example, F4R increased from the order of 10^2 to 10^4, with the ratio of the last to first value equal to 97.66. Similarly, MPCG2 also grew from

10^2 to 10^4, but with a much higher ratio of 324.9. The combined preconditioners MPCG1 and MPCG3 behaved even more drastically, both ranging from 10^2 to 10^5, with ratios of 341.77 and 340.9, respectively. On the other hand, F3T and F4CR remained very stable, both staying around the order of 10^0, with only tiny variations (ratios of 1.03 and 1.02). One interesting observation is that Jacobi actually produced smaller condition numbers than the unpreconditioned matrix throughout the process, which was not the case in the one-dimensional setting.

Table 4.21: Comparison of condition numbers for preconditioned CG solvers using four preconditioners and the unpreconditioned system (Stiffness Matrix) under $h-$uniform in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

				UP	Jacobi	SSOR	F4R	F4CR	F3T	MPCG1	MPCG2	MPCG3
Size of Matrix	*Ad*	*p*	*El*	Cond#	Cond#	Cond#	Cond#	Cond#	Cond#	Cond#	Cond#	Cond#
361^2	1	3	36	2.70E+02	2.33E+02	3.07E+01	2.57E+02	7.72E+00	6.83E+00	5.53E+02	2.29E+02	5.75E+02
484^2	2	3	49	3.68E+02	3.18E+02	4.15E+01	3.48E+02	7.77E+00	6.89E+00	1.02E+03	4.13E+02	1.06E+03
625^2	3	3	64	4.81E+02	4.17E+02	5.40E+01	4.53E+02	7.81E+00	6.93E+00	1.72E+03	6.93E+02	1.79E+03
784^2	4	3	81	6.09E+02	5.29E+02	6.81E+01	5.72E+02	7.83E+00	6.96E+00	2.75E+03	1.10E+03	2.85E+03
961^2	5	3	100	7.52E+02	6.54E+02	8.39E+01	7.05E+02	7.85E+00	6.98E+00	4.18E+03	1.67E+03	4.34E+03
1156^2	6	3	144	9.10E+02	7.92E+02	1.01E+02	8.53E+02	7.86E+00	6.99E+00	6.11E+03	2.43E+03	6.34E+03
1369^2	7	3	168	1.08E+03	9.43E+02	1.21E+02	1.16E+03	7.87E+00	7.00E+00	8.64E+03	3.43E+03	8.96E+03
1600^2	8	3	196	1.27E+03	1.11E+03	1.41E+02	1.59E+03	7.88E+00	7.01E+00	1.19E+04	4.70E+03	1.23E+04
1849^2	9	3	225	1.47E+03	1.28E+03	1.64E+02	2.13E+03	7.88E+00	7.02E+00	1.60E+04	6.31E+03	1.66E+04
2116^2	10	3	256	1.69E+03	1.48E+03	1.88E+02	2.80E+03	7.89E+00	7.03E+00	2.10E+04	8.31E+03	2.18E+04
2401^2	11	3	289	1.93E+03	1.68E+03	2.14E+02	3.62E+03	7.89E+00	7.03E+00	2.72E+04	1.07E+04	2.82E+04
2704^2	12	3	324	2.17E+03	1.90E+03	2.41E+02	4.61E+03	7.90E+00	7.04E+00	3.47E+04	1.37E+04	3.59E+04
3025^2	13	3	400	2.44E+03	2.13E+03	2.70E+02	5.78E+03	7.90E+00	7.04E+00	4.35E+04	1.72E+04	4.52E+04
3364^2	14	3	441	2.72E+03	2.37E+03	3.01E+02	7.17E+03	7.90E+00	7.04E+00	5.40E+04	2.13E+04	5.60E+04
3721^2	15	3	484	3.01E+03	2.63E+03	3.34E+02	8.80E+03	7.90E+00	7.04E+00	6.63E+04	2.61E+04	6.88E+04
4096^2	16	3	529	3.32E+03	2.89E+03	3.68E+02	1.07E+04	7.90E+00	7.05E+00	8.06E+04	3.17E+04	8.36E+04
4489^2	17	3	576	3.64E+03	3.18E+03	4.03E+02	1.29E+04	7.91E+00	7.05E+00	9.71E+04	3.82E+04	1.01E+05
4900^2	18	3	625	3.98E+03	3.47E+03	4.41E+02	1.54E+04	7.91E+00	7.05E+00	1.16E+05	4.56E+04	1.20E+05
5329^2	19	3	679	4.33E+03	3.78E+03	4.80E+02	1.82E+04	7.91E+00	7.05E+00	1.37E+05	5.41E+04	1.43E+05
5776^2	20	3	729	4.70E+03	4.10E+03	5.21E+02	2.14E+04	7.91E+00	7.05E+00	1.62E+05	6.36E+04	1.68E+05
6241^2	21	3	784	5.09E+03	4.44E+03	5.63E+02	2.51E+04	7.91E+00	7.05E+00	1.89E+05	7.44E+04	1.96E+05

Figure 4.27 shows this behavior clearly. Up to the seventh refinement step (matrix size 1369×1369, corresponding to a grid of 13×13 elements), F4R, Jacobi, and the stiffness matrix followed almost identical trends, both in shape and in magnitude. Starting from this step, however, the F4R curve separates and grows much faster than the other two. At refinement 7, the ratios of F4R to the stiffness matrix and Jacobi were only 1.07 and 1.2, but by the last refinement these differences had increased to 4.93 and 5.65, respectively. At the same time, the slopes of F4R and F4CR were among the mildest in Figure 4.27, so their growth was not as steep as in the combined preconditioners, even though they ended up with larger values than Jacobi.

This creates an apparent paradox. From the numbers alone, one would expect F4R to perform worse than Jacobi because its condition number kept growing larger, meaning the system became more ill-conditioned. But Table 4.19 shows that F4R actually required fewer iterations than Jacobi. Figure 4.28 helps explain this: under Jacobi, the eigenvalues are spread almost continuously, with only a small cluster forming near the maximum values. In contrast, F4R produces more than five distinct clusters, with eigenvalues packed tightly inside each group. Even though the overall condition number is worse, this clustering makes it easier for the CG polynomial to damp the error, so the solver converges in fewer iterations.

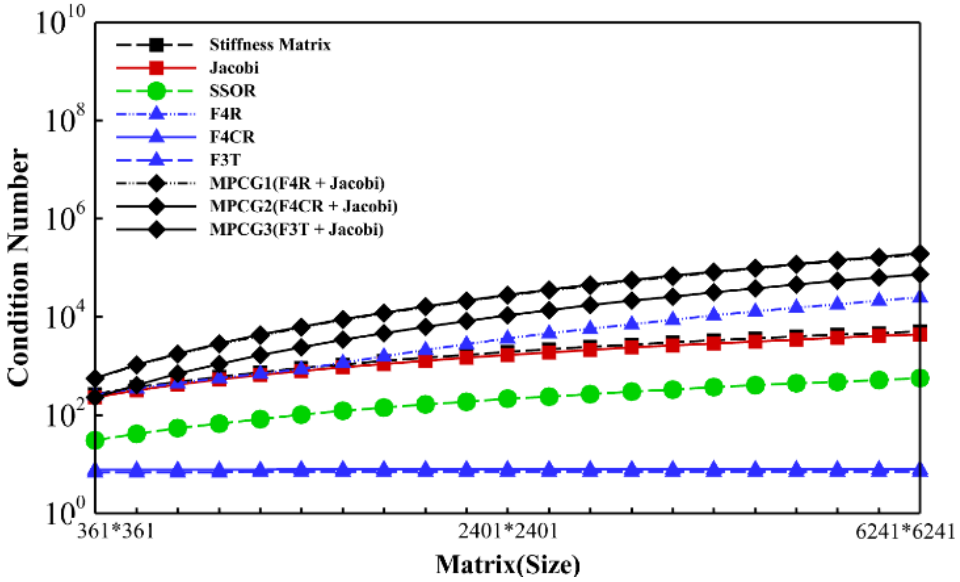


Figure 4.27: Variation of condition numbers with polynomial degree for preconditioned and unpreconditioned CG solvers under adaptivity, starting from 36 elements of polynomial degree 3 under h −uniform for 2D.

The difference between F3T and F4CR gives another good example. At refinement 7, both had very small condition numbers (7.00×10^{0} for F3T and 7.87×10^{0} for F4CR), much smaller than the unpreconditioned system (1.08×10^{3}) or Jacobi (9.43×10^{2}). However, Table 4.19 shows that F4CR required 132 iterations, while F3T only needed 18. This is about 7.3 times more iterations, even though their condition numbers were nearly

the same. Looking again at Figure 4.28, F3T shows tighter clustering of eigenvalues, with each cluster rising only slightly, while F4CR produces more clusters spread further apart. The broader spread forces CG to use higher-degree polynomials to cover all clusters, which explains the larger iteration count.

It is also worth noting that F3T and F4R are conceptually similar preconditioners: the first is built from triangular elements, and the second from quadrilateral elements. Despite this, F3T showed stronger ability to form compact eigenvalue clusters. For example, a large spectral gap appears at the high end in F4R, which is not present in F3T. This difference, combined with the slower growth of eigenvalues inside each cluster, made F3T more effective for CG. At refinement 7, even though F4R had a condition number of 1.16×10^3 (much larger than F3T's 7.00×10^0), it converged in only 20 iterations, close to F3T's 18 iterations. This comparison underlines that condition number alone cannot predict CG performance: the clustering of eigenvalues plays a central role in determining the number of iterations required.

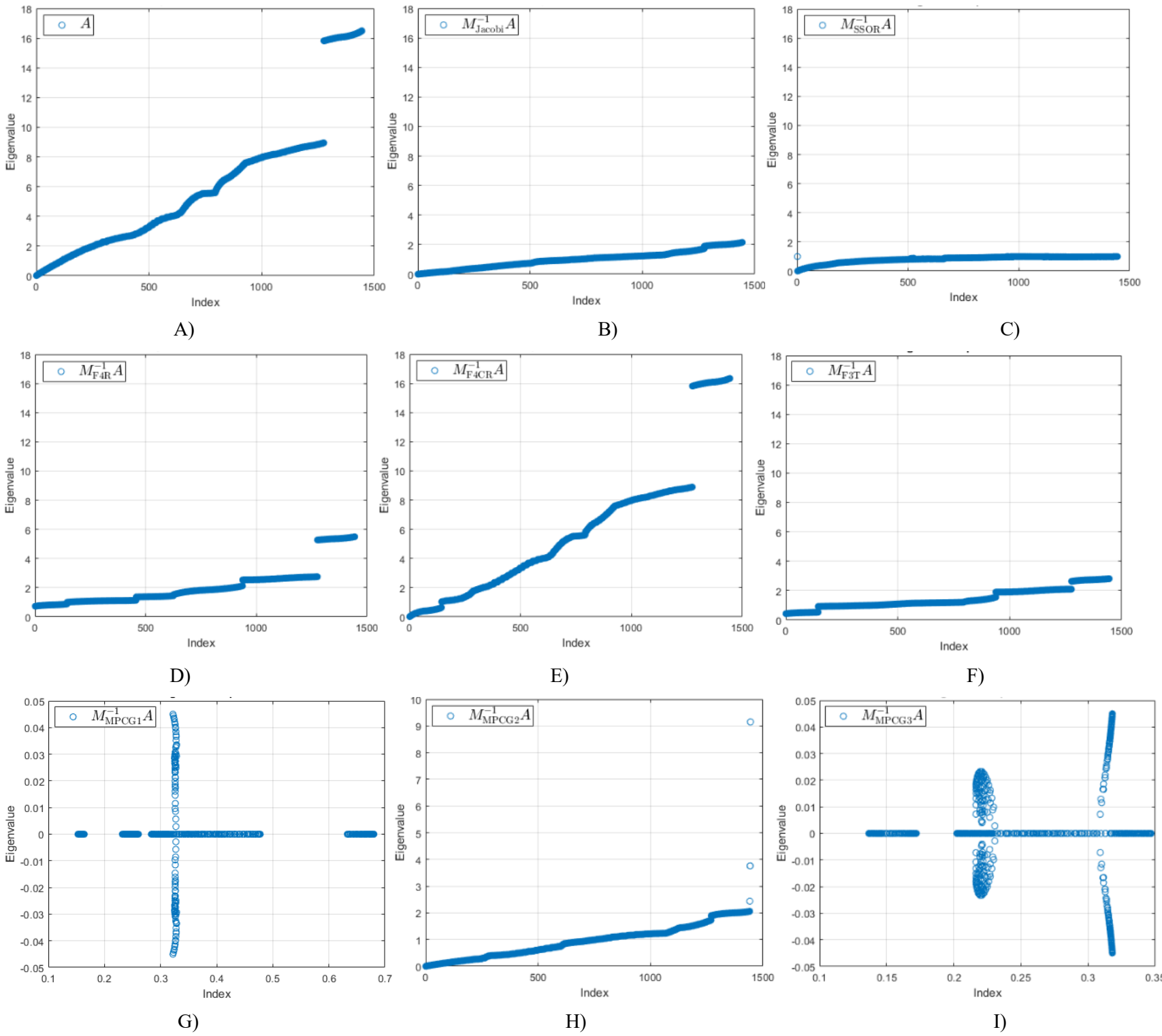


Figure 4.28: Eigenvalue distributions of the nine preconditioned and unpreconditioned system (Stiffness Matrix) under h −uniform corresponding to the 7th refinement for 2D. (A) shows the unpreconditioned system A. (B) Jacobi preconditioned system ${M_{Jacobi}}^{-1}A$. (C) SSOR preconditioned system ${M_{SSOR}}^{-1}A$. (D) F4R preconditioned system ${M_{F4R}}^{-1}A$. (E) F4CR preconditioned system ${M_{F4CR}}^{-1}A$. (F) F3T preconditioned system ${M_{F3T}}^{-1}A$. (G) Multi-PCG1 preconditioned system ${M_{MPCG1}}^{-1}A$. (H) Multi-PCG2 preconditioned system ${M_{MPCG2}}^{-1}A$. (I) Multi-PCG3 preconditioned system ${M_{MPCG3}}^{-1}A$.

4.5.3 2D Adaptive Mesh Refinement

In two dimensions, the adaptive refinement study is restricted to $p-$adaptivity, as described in Section 3.8.4.1. The overall strategy parallels the one-dimensional study (Sections 3.8.2), in the sense that the exact solution is used to evaluate errors and isolate the effect of preconditioning. However, unlike the 1D case, the 2D setting introduces additional complexity due to nonconforming interfaces between elements of differing polynomial degrees. The treatment of these interfaces through interpolation operators, discussed in Section 3.8.4.1, is an essential part of the implementation. This ensures minimizes discontinuity across element boundaries while still allowing us to systematically examine solver performance under p-adaptivity.

4.5.3.1 p - Adaptive Mesh Refinement

According to Table 4.19, the $p-$multigrid preconditioner shows the same behavior as in the p-uniform refinement case, with iteration counts independent of the element degree. This confirms that the multigrid strategy is effective in removing both high- and low-frequency errors, so the number of CG steps remains nearly constant even as the polynomial order increases. In this setting of $p-$adaptivity, the SSOR preconditioner performs much better than in the uniform case discussed earlier. Here, SSOR produces fewer iterations than all other preconditioners except $p-$multigrid. Figure 4.28, which shows the eigenvalue distribution at the first adaptive step, 568×568, helps to explain this. For SSOR, the eigenvalues are spread continuously without overlapping, which means the CG solver can be represented with a polynomial of lower degree. In contrast, for Jacobi, clear overlaps appear in the eigenvalue spectrum, forcing CG to use a higher-degree polynomial to achieve convergence.

Table 4.22: Iterative performance of preconditioned CG solvers for growing matrix sizes in $p-$adaptive refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

				Jacobi	SSOR	F4R	F4CR	F3T	MPCG1	MPCG2	MPCG3	p_multigrid
Size of matrix	*Ad*	p	*El*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*	*Iter*
361^2	0	3	36	66	30	21	75	20	20	62	16	7
568^2	1st	4	363	144	47	455	173	112	114	122	62	30
804^2	2nd	5	36	271	75	759	323	238	181	198	122	30
1001^2	3rd	6	36	504	115	1187	595	428	253	267	180	30
1091^2	4th	7	36	790	157	732	944	839	190	291	222	30
1105^2	5th	8	36	974	185	1744	1235	989	290	294	231	30

This observation is the opposite of what was seen earlier in Figure 4.24, where Jacobi showed denser clustering than SSOR. A striking result is that the F4R, F4CR, and F3T

preconditioners not only failed to reduce iteration counts but actually increased them compared to Jacobi. At the largest matrix size, 1105×1105, the iteration counts grew by factors of about 1.8, 1.3, and 1.01 respectively compared to Jacobi. By contrast, the combined MPCG preconditioners achieved significant reductions. MPCG1, MPCG2, and MPCG3 reduced the iteration counts by factors of 6.1, 4.2, and 4.2 respectively compared to their basis (F4R, F4CR, and F3T), showing that the multi-preconditioned approach is much more effective in this adaptive setting. Figure 23 highlights some of these behaviors in more detail. For example, both F4R and its combined version MPCG1 show a drop in iteration count at size 1091×1091, compared to the previous matrix size. This irregularity is not observed in all preconditioners.

In fact, F3T, Jacobi, and F4CR display almost steady growth in their iteration counts across the whole range, with slopes that remain nearly constant from one adaptive step to the next. On the other hand, preconditioners such as SSOR, MPCG2, and MPCG3 behave differently. Starting from about size 1001×1001, their slopes decrease, so that each further refinement adds fewer iterations than before.

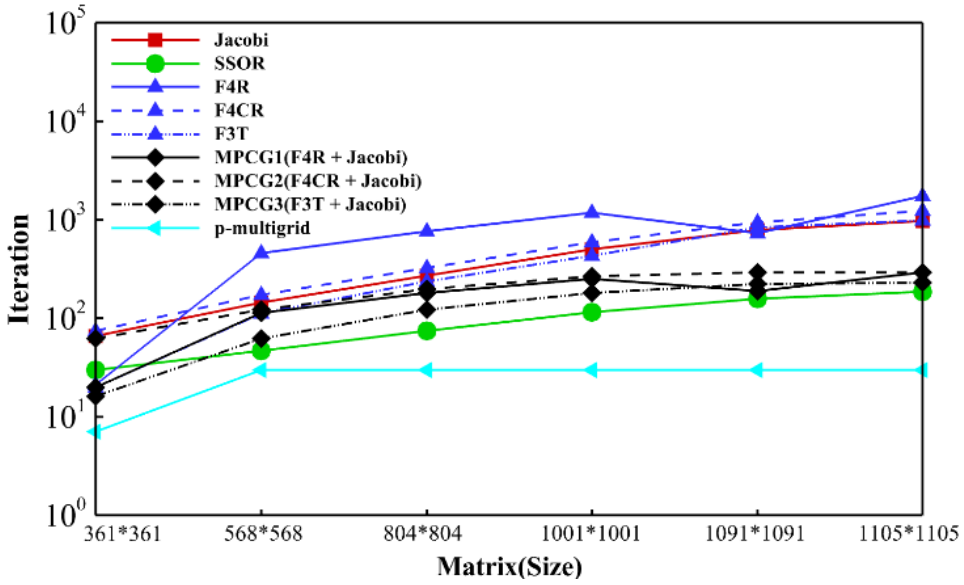


Figure 4.29: Iteration counts for nine preconditioners under p −adaptive refinement in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

According to Table 4.20, within the group of F4R, F4CR, and F3T, the F4CR reports much lower runtimes compared to the other two. At the fifth adaptive step, for example, the runtime of F4CR is 9.35×10^0, while F4R and F3T require 9.21×10^1 and 3.72×10^1, respectively. The same behavior can be seen in the combined preconditioners MPCG1, MPCG2, and MPCG3. In most cases, MPCG2 lies between the other two, except at the third and fourth adaptive steps where the order changes to MPCG3 shows the smallest runtime in the third step, and MPCG1 becomes the smallest in the fourth step.

Table 4.23: Runtimes of nine preconditioners with increasing matrix size in p −adaptivity in 2D starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

				Jacobi	SSOR	F4R	F4CR	F3T	MPCG1	MPCG2	MPCG3	p_multigrid
Size of Matrix	*Ad*	p	*El*	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)	Runtime (Second)
361^2	0	3	36	4.47E-03	3.51E-02	2.65E-02	2.80E-02	3.44E-02	3.37E-02	8.54E-02	3.44E-02	5.86E-02
568^2	1st	4	36	1.77E-02	1.67E-01	3.83E+00	2.78E-01	8.64E-01	1.38E+00	6.67E-01	6.12E-01	3.12E-01
804^2	2nd	5	36	6.51E-02	8.87E-01	1.87E+01	1.01E+00	4.07E+00	7.63E+00	4.21E+00	3.88E+00	1.07E+00
1001^2	3rd	6	36	2.33E-01	3.45E+00	5.69E+01	3.62E+00	1.35E+01	2.11E+01	1.15E+01	1.06E+01	1.73E+00
1091^2	4th	7	36	4.90E-01	9.10E+00	3.05E+01	6.91E+00	3.15E+01	1.45E+01	1.58E+01	1.72E+01	2.88E+00
1105^2	5th	8	36	6.13E-01	1.42E+01	9.21E+01	9.35E+00	3.72E+01	2.99E+01	1.62E+01	1.85E+01	3.27E+00

Figure 4.26 helps to visualize these trends. For the p −multigrid preconditioner, the slope of the runtime curve becomes smaller with each adaptive step. This indicates that once the coarser errors are removed at the lower levels, the solver only needs to handle the smoother error components at the finer levels, so the cost does not rise sharply as the matrix size grows. A similar decreasing slope can also be observed in several other preconditioners, although not in all of them. For F4R, MPCG1, and F3T, the slope remains higher, and the growth in runtime follows the matrix size more directly. Between F4R and MPCG1, the two curves look very similar in shape, but F4R is consistently about 2.5 times larger in runtime at each adaptive step.

From Table 4.20 it is also clear that F4R is the slowest among these methods, while F4CR is the fastest inside its family. The range of runtimes reported for the different preconditioners extends from about 2.65×10^{-2} up to 8.76×10^{1}, which shows how widely the computational cost can vary even under the same adaptive process.

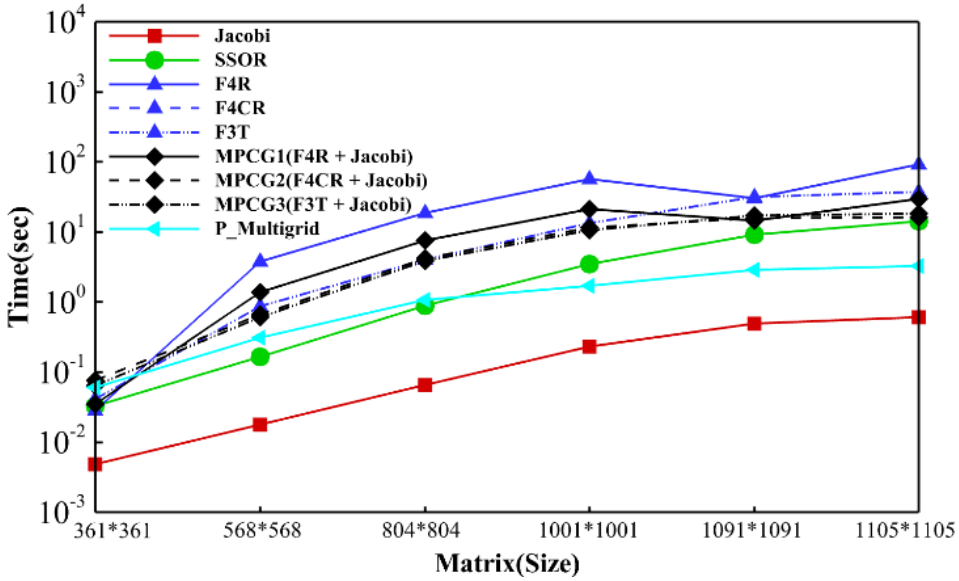


Figure 4.30: Runtime comparison of nine preconditioners with increasing matrix size under adaptivity p −adaptivity in 2D, starting from an initial mesh of 36 elements with polynomial degree $p = 3$.

Table 4.21 and Figure 4.27 show the condition numbers that are obtained with different preconditioners follow a broadly similar pattern. For most cases, the values remain in the same order of magnitude as those produced by the Jacobi preconditioner, generally within the range of 2.33×10^{2} to 1.71×10^{5}. The only noticeable exceptions are F4R, F4CR, MPCG1, and SSOR, which show deviations from this trend. In fact, almost all preconditioners except SSOR tend to generate condition numbers that are somewhat larger than Jacobi. Despite this, they do not necessarily lead to higher iteration counts. This shows that the condition number by itself is not always a reliable indicator of solver behavior, and that the distribution of eigenvalues must also be considered.

Table 4.24: Comparison of condition numbers for preconditioned CG solvers using nine preconditioners and the unpreconditioned system (Stiffness Matrix) under $p-$adaptivity starting from 36 elements of polynomial degree 3 for 2D.

				UP	Jacobi	SSOR	F4R	F4CR	F3T	MPCG1	MPCG2	MPCG3
Size of Matrix	*Ad*	*p*	*El*	Cond#	Cond#	Cond#	Cond#	Cond#	Cond#	Cond#	Cond#	Cond#
361^2	0	3	36	2.70E+02	2.33E+02	3.07E+01	2.57E+02	7.72E+00	6.83E+00	5.53E+02	2.29E+02	5.75E+02
568^2	1st	4	36	1.15E+03	9.85E+02	8.51E+01	1.42E+03	2.72E+04	1.69E+03	3.18E+04	1.26E+03	5.04E+03
804^2	2nd	5	36	5.01E+03	4.34E+03	2.79E+02	5.93E+03	6.97E+04	8.00E+03	1.02E+05	5.09E+03	1.79E+04
1001^2	3rd	6	36	2.35E+04	2.05E+04	1.10E+03	2.95E+04	2.34E+05	1.63E+04	6.45E+05	2.48E+04	4.92E+04
1091^2	4th	7	36	1.07E+05	6.46E+04	3.71E+03	1.14E+05	5.54E+04	1.05E+05	2.96E+05	7.11E+04	1.56E+05
1105^2	5th	8	36	5.70E+05	1.71E+05	1.12E+04	5.69E+05	7.57E+05	1.62E+05	1.44E+06	1.78E+05	3.14E+05

Figure 4.27 illustrates this point more clearly. For example, the F3T preconditioner achieves a smaller condition number than Jacobi only at the third adaptive step, where the difference is modest, with Jacobi being about 1.2 times larger. On other steps, F3T stays close to Jacobi but does not show a consistent advantage. In addition, the condition-number increases steadily with each adaptation step for all preconditioners, which is expected as the matrix grows larger. However, two preconditioners, MPCG1 and F4CR, display a slight anomaly, at the matrix size of 1001×1001 (the third adaptive step), their condition numbers temporarily dip before rising again in the following step. This local reduction does not change the overall trend but highlights that their spectral effects are not fully monotonic during the adaptive process.

It is also important to note that SSOR behave differently from the other methods. While most preconditioners produce condition numbers larger than Jacobi, SSOR is the only one that remains below Jacobi for part of the process. Even so, SSOR does not achieve iteration count significantly better than the others, underlining again that both the magnitude of the condition number and the clustering of eigenvalues influence convergence. Finally, aside from these specific cases, the majority of preconditioners do not create dramatic changes in the condition number compared to the unpreconditioned system, which means their effectiveness must be interpreted through a combined view of condition number growth and eigenvalue distribution rather than condition number alone.

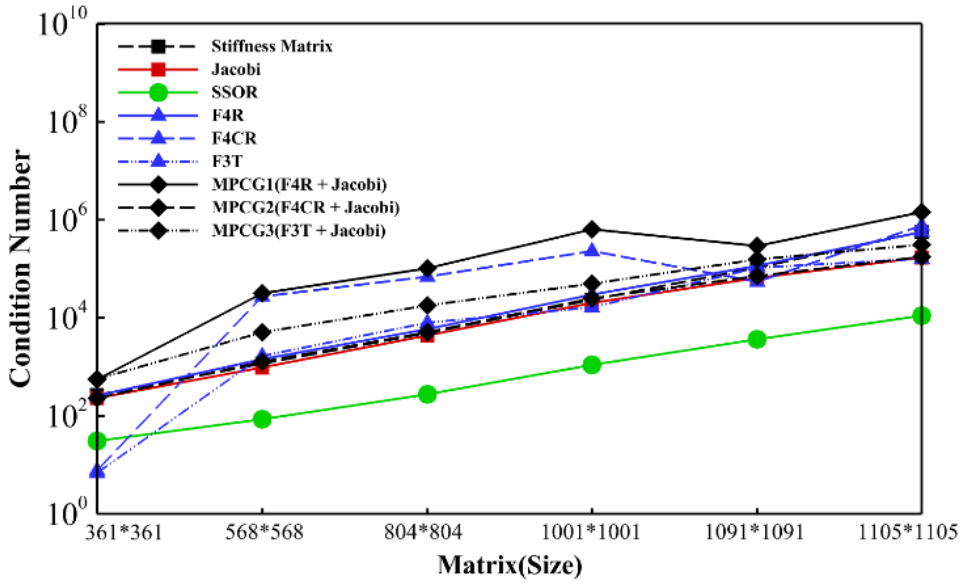


Figure 4.31: Variation of condition numbers with polynomial degree for preconditioned and unpreconditioned CG solvers under adaptivity, starting from 36 elements of polynomial degree 3 under p −adaptivity for 2D.

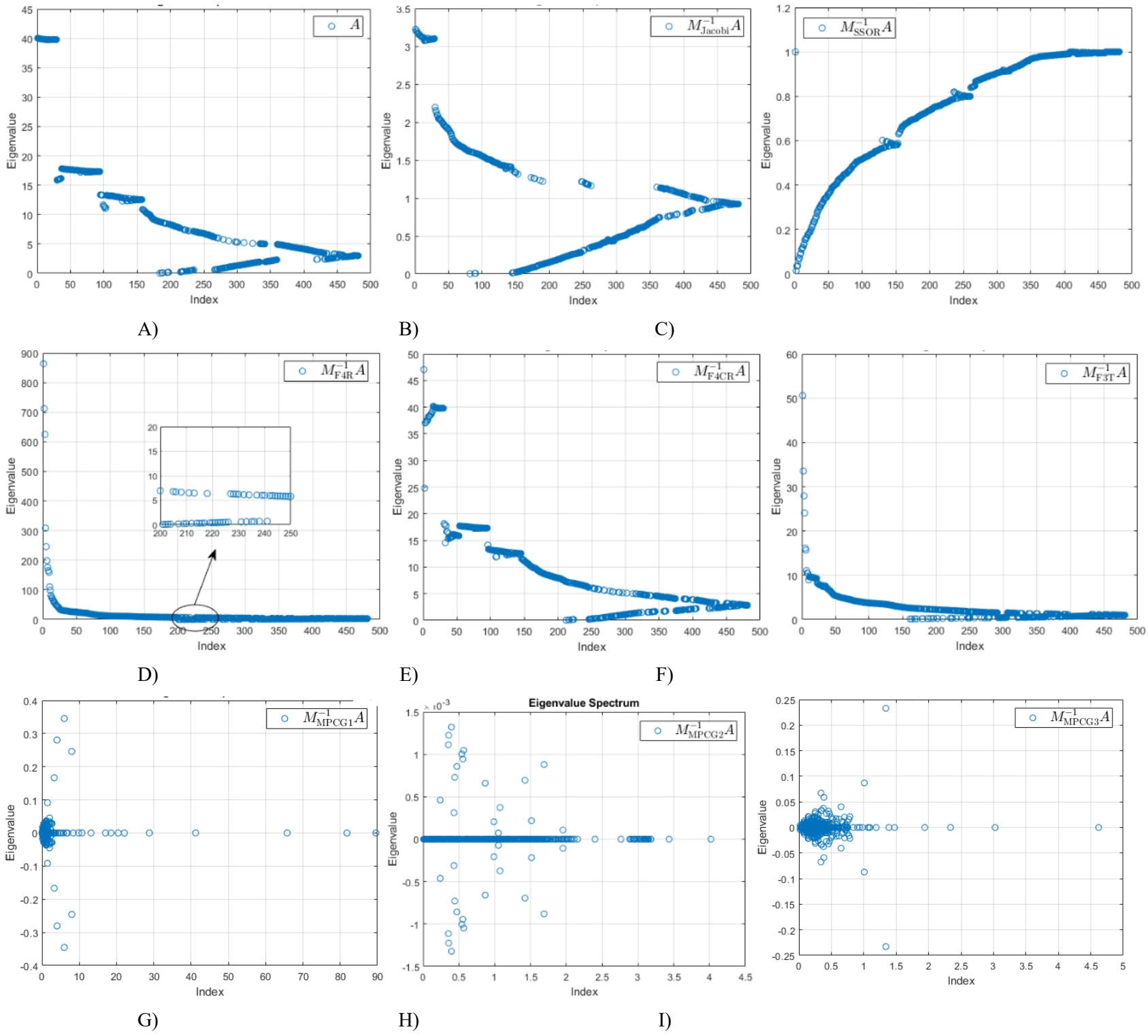


Figure 4.32: Eigenvalue distributions of the preconditioned and unpreconditioned system (Stiffness Matrix) under p −adaptivity of size 568×568 corresponding to first adaptation. (A) shows the unpreconditioned system A. (B) Jacobi preconditioned system ${M_{Jacobi}}^{-1}A$. (C) SSOR preconditioned system ${M_{SSOR}}^{-1}A$. (D) F4R preconditioned system ${M_{F4R}}^{-1}A$. (E) F4CR preconditioned system ${M_{F4CR}}^{-1}A$. (F) F3T preconditioned system ${M_{F3T}}^{-1}A$. (G) Multi-PCG1 preconditioned system ${M_{MPCG1}}^{-1}A$. (H) Multi-PCG2 preconditioned system ${M_{MPCG2}}^{-1}A$. (I) Multi-PCG3 preconditioned system ${M_{MPCG3}}^{-1}A$.

Chapter 5: Conclusion

5.1 Summary

This thesis examined the performance of preconditioned conjugate gradient (PCG) solvers under different refinement strategies for the variable order finite element methods in both one and two dimensions. The study covered uniform refinement as well as adaptive approaches ($p-$, $h-$, and $hp-$adaptivity) and compared a wide range of preconditioners. By combining exact solutions with systematic experiments, the results gave a direct view of how iteration counts, condition numbers, eigenvalue distributions, and runtime are affected by refinement and preconditioning.

One of the main findings is that condition number alone is not a reliable predictor of solver performance. While Jacobi and SSOR often reduced the condition number slightly compared to the unpreconditioned (UP) case, their eigenvalue distributions remained relatively uniform, which forced the CG method to use higher-degree residual polynomials and led to higher iteration counts. In contrast, preconditioners such as FEM1 in one dimension, and F3T or F4R in two dimensions, were able to cluster the eigenvalues into distinct groups. This clustering proved to be more important for convergence, since CG is not only affected by the condition number but also by how the eigenvalues are distributed. When the eigenvalues form clear clusters, the solver converges in fewer iterations since it is easier for the residual polynomial to handle grouped spectra.

The adaptive refinement studies also revealed clear differences between refinement types. In $h-$adaptivity, the solver required many adaptation steps and large system sizes before reaching tolerance, but the iteration counts remained relatively stable. In $p-$adaptivity, convergence was reached with far fewer adaptations, but the condition numbers grew faster, especially with higher polynomial orders. In $hp-$adaptivity, the system size became much larger than in $p-$adaptivity, but the solver still needed about the same number of iterations.

In two dimensions, the family of finite element order 1 preconditioners (F3T, F4R, and F4CR) showed different behaviors despite their conceptual similarity. F3T consistently produced tighter clustering of eigenvalues and fewer distinct clusters, which allowed for faster convergence than F4CR, even when their condition numbers were comparable. The $p-$multigrid preconditioner was especially notable for its stability, keeping the iteration nearly constant across refinements, reflecting its theoretical independence from mesh size or polynomial degree.

To keep the scope focused yet representative, one test case was used in 1D and one in 2D. The 1D Poisson case with $f(x) = (x)^{50}$ was chosen because it produces steep gradients, creating a demanding setting for both refinement and preconditioning. At the

same time, assembling the stiffness matrix for this problem requires high-order numerical integration in 1D, thereby exercising the quadrature rules at large polynomial degrees. The 2D case employed a manufactured forcing term combining trigonometric functions (sine and cosine) with high powers of x and y (see Chapter 6). This ensured an exact analytical solution while stressing high-order integration and yielding stiffness matrices with rich spectral structure both essential for evaluating solver and preconditioner behavior. Although other source terms such as oscillatory or smoother profiles could have been used, the chosen examples are sufficient to illustrate the general trends of solver and preconditioner performance. In practice, both the spectral properties of the stiffness matrix and the effects of adaptivity influence solver behavior. Therefore, the results of this study can be expected to apply beyond the specific examples considered. Furthermore, based on the results shown in Figures 4.22 and 4.30, it is possible that the $p-$multigrid preconditioner becomes increasingly advantageous for extremely large systems, given its near mesh- and order-independent iteration behavior. In addition, the trends observed in this work are highly consistent with those reported in [108], where spectral elements generally benefit from better conditioning; yet, even in such settings, simple diagonal schemes such as Jacobi remain competitive in runtime despite requiring substantially more iterations compared to FEM-based preconditioners.

Overall, the results show that condition number alone does not fully explain solver performance. Eigenvalue distribution and clustering play an equally important role, while the refinement strategy strongly influences the balance between system size, iteration count, and runtime. The comparison across one and two dimensions, and across uniform and adaptive refinements, makes clear that preconditioners must be judged not only by their impact on condition numbers, but also by how they shape the spectrum and the practical cost of solving large systems.

5.2 : Future Work

For future work, we will begin with natural extensions of the present study. The immediate next step is to implement h-adaptivity in 2D, which requires local mesh refinement guided by a posteriori error estimators, together with appropriate data structures for handling hanging nodes. Building on this, hp-adaptivity in 2D can be investigated by combining mesh refinement with local polynomial enrichment, which demands robust refinement indicators and more sophisticated preconditioning strategies.

A further stage is to insert the developed solvers into a full Navier–Stokes code. In this context, two Poisson problems arise: one for velocity and one for pressure. The pressure Poisson equation in incompressible flow is particularly challenging, as its solution is often slow to converge, making it an ideal testbed for advanced preconditioning and adaptivity.

Beyond these near-term directions, a longer-term research avenue is to investigate high-order finite element formulations in Riemannian spaces. In Cartesian meshes, curvature is

only represented indirectly by increasing the polynomial degree, whereas Riemannian metrics incorporate curvature intrinsically through geodesics. This suggests that low-order elements defined in such spaces could attain accuracy comparable to high-order Cartesian elements, thereby reducing both discretization error and computational cost.

Preliminary studies point in this direction. Early work on anisotropic mesh adaptation Vallet et al. [106] and more recent reviews on metric-based discretization Alauzet and Loseille [107] have shown that Riemannian metrics can reduce discretization error. Extending these ideas to high-order FEM would be a large-scale undertaking and would naturally require high-performance computing (HPC) resources.

Another interesting direction could be to design preconditioners inspired by turbulence filtering methods. In turbulence modeling, especially in Large Eddy Simulation, small-scale fluctuations are suppressed by applying filters, so that only the large, physically relevant structures are resolved. A similar analogy can be drawn for iterative solvers: very high condition numbers correspond to unstable, oscillatory modes in the spectrum of the stiffness matrix, which slow down convergence. The idea would be to develop a preconditioner that acts like a filter, damping or removing these problematic modes while preserving the main spectral content of the system. This could initially be explored through eigenvalue analysis, by constructing matrix-based filters that cluster the spectrum more tightly. While multigrid already reduces high-frequency error components through smoothing and coarse-grid transfer, the proposed approach would aim at a more direct filtering strategy that could be applied as a standalone operator on the stiffness matrix.

Finally, extending the study to spectral element methods, which tend to be better conditioned, could further clarify how preconditioners behave in less ill-conditioned systems.

Appendix A: Additional Material

For completeness, the explicit expression of the manufactured forcing function $f(x, y)$ used in Chapter 4 in the section 4.4 is given below. This function was designed to create a challenging test case with steep gradients and oscillatory behavior, providing a rigorous evaluation of the adaptive high-order finite element formulations methodology discussed in the thesis.

$$
\begin{aligned}
f(x,y) = {} & 48xy\cos(6\pi x)\,(x-1)^6 + 24x\cos(6\pi x)\,(2y-2)(x-1)^6 \\
& + 4x\cos(6\pi x)\,(12y-1)(x-1)^6 \\
& + 12y\cos(6\pi x)\,(2y-2)(12y-1)(x-1)^5 \\
& - 12y\pi\sin(6\pi x)\,(2y-2)(12y-1)(x-1)^6 \\
& + 30xy\cos(6\pi x)\,(2y-2)(12y-1)(x-1)^4 \\
& - 36xy\pi^2\cos(6\pi x)\,(2y-2)(12y-1)(x-1)^6 \\
& - 72xy\pi\sin(6\pi x)\,(2y-2)(12y-1)(x-1)^5
\end{aligned}
$$

Bibliography

[1]. Hekmat, M. H., Rahmanpour, M., Mahmoudi, M., and Saharkhiz, S. (2020), A genetic algorithm-based approach for numerical solution of droplet status after Coulomb fission using the energy conservation method, *Journal of Computational Applied Mechanics*, Vol. 51, No. 2, pp. 454–463.

[2]. Hekmat, M. H., and Saharkhiz, S. (2025), Impact of helically coiled shell and tube on melting and solidification of PCMs in thermal energy storage systems: A three-dimensional parametric study, *Journal of Energy Storage*, Vol. 108, Article 115172.

[3]. Hekmat, M. H., Saharkhiz, S., and Izadpanah, E. (2019), Investigation on the thermal mixing enhancement in a T-junction pipe, *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, Vol. 41, Article 276.

[4]. Fang, Q., Tsuchiya, T. and Yamamoto, T., (2002), Finite Difference, Finite Element and Finite Volume Methods Applied to Two-Point Boundary Value Problems, *Journal of Computational and Applied Mathematics*, Vol. 139, No. 1, pp. 9–19.

[5]. Yin, P., Liandrat, J. and Shen, W., (2021), A Comparison of the Finite Difference and Multiresolution Method for the Elliptic Equations with Dirichlet Boundary Conditions on Irregular Domains, *Journal of Computational Physics*, Vol. 434, Article 110207.

[6]. Luo, Z.,(2022), Computation of Two-Dimensional Poisson Equation Using the Third-Order Discrete Scheme of Finite Difference Method Based on Node Set Vector, *Journal of Physics: Conference Series*, Vol. 2381, No. 1, Article 012039.

[7]. Visbal, M. R., and Gaitonde, D. V., (2002), On the Use of Higher-Order Finite-Difference Schemes on Curvilinear and Deforming Meshes, *Journal of Computational Physics*, Vol. 181, No. 1, pp. 155–185.

[8]. Olek, C. Z., and Robert, L. T., (2013), *The Finite Element Method: Its Basis and Fundamentals*, 7th ed., Elsevier.

[9]. Sakurai, K., Aoki, T., Lee, W.-H. and Kato, K., (2002), Poisson Equation Solver with Fourth-Order Accuracy by Using Interpolated Differential Operator Scheme, *Computers and Mathematics with Applications*, Vol. 43, No. 5–6, pp. 621–630.

[10]. Tu, J., Yeoh, G. H., Liu, C., & Tao, Y. (2024). CFD techniques – The basics. In J. Tu, G. H. Yeoh, C. Liu, & Y. Tao (Eds.), *Computational Fluid Dynamics* (4th ed., pp. 153–208). Butterworth-Heinemann.

[11]. Luo, Z., Li, H., Sun, P., An, J., and Navon, I. M., (2013), A Reduced-Order Finite Volume Element Formulation Based on POD Method and Numerical Simulation for Two-Dimensional Solute Transport Problems, *Mathematics and Computers in Simulation*, Vol. 89, pp. 50–68.

[12]. Costa, R., Clain, S., Loubère, R. and Machado, G. J., (2018), Very High-Order Accurate Finite Volume Scheme on Curved Boundaries for the Two-Dimensional Steady-State Convection–Diffusion Equation with Dirichlet Condition, *Applied Mathematical Modelling*, Vol. 54, pp. 752–767.

[13]. Krivodonova, L., and Berger, M., (2006), High-Order Accurate Implementation of Solid Wall Boundary Conditions in Curved Geometries, Journal of Computational Physics, Vol. 211, No. 2, pp. 492–512.

[14]. Balakrishnan, N. and Fernandez, G., (1998), Wall Boundary Conditions for Inviscid Compressible Flows on Unstructured Meshes, *International Journal for Numerical Methods in Fluids*, Vol. 28, No. 10, pp. 1481–1501.

[15]. Galerkin, B.G., (1915), Rods and Plates, Series Occurring in Various Questions Concerning the Elastic Equilibrium of Rods and Plates, *Vestnik Inzhenerov i Tekhnikov* (Engineers and Technologists Bulletin), Vol. 19, pp. 897–908, 1915 (in Russian).

[16]. Reed, W. H. and Hill, T. R. , (1973), *Triangular Mesh Methods for the Neutron Transport Equation*, Technical Report LA-UR-73-479, Los Alamos Scientific Laboratory, Los Alamos, NM.

[17]. Cockburn, B., Hou, S. and Shu, C.-W., (1990), *The Runge–Kutta Local Projection Discontinuous Galerkin Finite Element Method for Conservation Laws. IV: The Multidimensional Case, Mathematics of Computation, Vol. 54, No. 190, pp. 545–581.*https://doi.org/10.2307/2008501.

[18]. Mavriplis, C. (1994). Adaptive mesh strategies for the spectral element method. *Computer Methods in Applied Mechanics and Engineering*, Vol.116, Nos. 1–4, pp. 77–86.

[19]. Benzi, M., (2002), Preconditioning Techniques for Large Linear Systems: A Survey, *Journal of Computational Physics*, Vol. 182, No. 2, pp. 418–477.

[20]. Li, X., Wu, J., Tai, X., Xu, J., & Wang, Y.-G. (2024). Solving a class of multi-scale elliptic PDEs by Fourier-based mixed physics informed neural networks. *Journal of Computational Physics*, Vol. 508, 113012.

[21]. Witherden, F. D., Vincent, P. E. and Jameson, A., (2016), High-Order Flux Reconstruction Schemes, in Abgrall, R. and Shu, C.-W. (eds.), *Handbook of Numerical Analysis*, Vol. 17, pp. 227–263, Elsevier. https://doi.org/10.1016/bs.hna.2016.09.010

[22]. Qin, C., Fu, W., Zhao, N., & Zhou, J. (2025). 3D adaptive finite-element forward modeling for direct current resistivity method using geometric multigrid solver. *Computers & Geosciences*, Vol. 196, 105840.

[23]. Hu, N., Guo, X.-Z., and Katz, I. N., (1997), Multi-p Preconditioners, *SIAM Journal on Scientific Computing*, Vol. 18, No. 6, pp. 1676–1697.

[24]. Xiao, J., Liu, Y., and Yi, N., (2026), High Accuracy Techniques Based Adaptive Finite Element Methods for Elliptic PDEs, *Journal of Computational and Applied Mathematics*, Vol. 472, 116799.

[25]. Berger, M., & Colella, P. (1989), Local Adaptive Mesh Refinement for Shock Hydrodynamics, *Journal of Computational Physics*, Vol. 82, No. 1, pp. 64–84

[26]. Verfürth, R., (2013), *A Posteriori Error Estimation Techniques for Finite Element Methods*, Oxford University Press, Oxford.

[27]. Babuška, I. and Suri, M., (1994), The p- and hp-Versions of the Finite Element Method: Basic Principles and Properties, *SIAM Review*, Vol. 36, No. 4, pp. 578–632.

[28]. Grätsch, T., and Bathe, K.-J., (2005), A Posteriori Error Estimation Techniques in Practical Finite Element Analysis, *Computers & Structures*, Vol. 83, Nos. 4–5, pp. 235–265. https://doi.org/10.1016/j.compstruc.2004.08.011

[29]. Heys, J. J., Manteuffel, T. A., McCormick, S. F., & Olson, L. N. (2005). Algebraic multigrid for higher-order finite elements. *Journal of Computational Physics*, 204(2), 520–532.

[30]. Babuška, I., & Suri, M. (1987). The h-p version of the finite element method with quasiuniform meshes. *ESAIM: Modélisation Mathématique et Analyse Numérique*, Vol. 21, No. 2, pp. 199–238.

[31]. Verfürth, R., (1994), A Posteriori Error Estimation and Adaptive Mesh-Refinement Techniques, *Journal of Computational and Applied Mathematics*, Vol. 50, No. 1–3, pp. 67–83.

[32]. Demkowicz, L., Kurtz, J., Pardo, D., Rachowicz, W., Paszyński, M. and Zdunek, A., (2002), A Fully Automatic hp-Adaptivity, *Journal of Scientific Computing*, Vol. 17, Nos. 1–4, pp. 127–155.

[33]. Mavriplis, C., (1990), A Posteriori Error Estimators for Adaptive Spectral Element Techniques, in Wesseling, P. (ed.), *Proceedings of the Eighth GAMM-Conference on Numerical Methods in Fluid Mechanics*, Springer Fachmedien, Wiesbaden.

[34]. Householder, A. S., (1964), *The Theory of Matrices in Numerical Analysis*, Blaisdell Publishing Company, New York.

[35]. Schwarz, H. R., (1979), the Method of Conjugating Gradients in Finite Element Applications, *Journal of Applied Mathematics and Physics (ZAMP)*, Vol. 30, No. 1, pp. 109–119.

[36]. Golub, G. H., and O'Leary, D. P., (1989), Some History of the Conjugate Gradient and Lanczos Algorithms: 1948–1976, *SIAM Review*, Vol. 31, No. 1, pp. 50–102.

[37]. Saad, Y., and van der Vorst, H. A., (2000), Iterative Solution of Linear Systems in the 20th Century, *Journal of Computational and Applied Mathematics*, Vol. 123, Nos. 1–2, pp. 1–33.

[38]. Kaneda, Y., Kawamura, H. and Sasai, M. (eds.), (2007), Frontiers of Computational Science: *Proceedings of the International Symposium on Frontiers of Computational Science 2005*, Springer, Berlin and Heidelberg.

[39]. Lanczos, C., (1952), Solution of Systems of Linear Equations by Minimized Iterations, *Journal of Research of the National Bureau of Standards*, Vol. 49, pp. 33–53. https://api.semanticscholar.org/CorpusID:7484650

[40]. Hestenes, M. R., and Stiefel, E. L., (1952), Methods of Conjugate Gradients for Solving Linear Systems, *Journal of Research of the National Bureau of Standards*, Vol. 49, pp. 409–436.

[41]. Shewchuk, J. R., (1994), *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain (Edition 1¼)*, School of Computer Science, Carnegie Mellon University.

[42]. Hestenes, M. R., and Stiefel, E., (1952), Methods of Conjugate Gradients for Solving Linear Systems, *Journal of Research of the National Bureau of Standards*, Vol. 49, pp. 409–435. https://doi.org/10.6028/jres.049.044

[43]. Lee, B., and Min, C. (2021), Optimal Preconditioners on Solving the Poisson Equation with Neumann Boundary Conditions, *Journal of Computational Physics*, Vol. 433, 110189.

[44]. MacLachlan, S. P., & Olson, L. N. (2014). Theoretical bounds for algebraic multigrid performance: review and analysis. *Numerical Linear Algebra with Applications*, Vol. 21, No. 2, 194–220.

[45]. Axelsson, O. and Barker, V., (1984), The Method of Conjugating Gradients in Finite Element Applications, *Numerische Mathematik*, Vol. 45, No. 1, pp. 1–10.

[46]. Turing, A. M., (1948), Rounding-Off Errors in Matrix Processes, *The Quarterly Journal of Mechanics and Applied Mathematics*, Vol. 1, No. 1, pp. 287–308.

[47]. Evans, D. J., (1968), The Use of Pre-Conditioning in Iterative Methods for Solving Linear Equations With Symmetric Positive Definite Matrices, *IMA Journal of Applied Mathematics*, Vol. 4, No. 3, pp. 295–314.

[48]. Chen, L., (2020), *Conjugate Gradient Methods*, University of California, Irvine, Lecture Notes, November 12.

[49]. Ahn, C. H., Chew, W. C., Zhao, J. S., and Michielssen, E., (1999), Numerical Study of Approximate Inverse Preconditioner for Two-Dimensional Engine Inlet Problems, *Electromagnetics*, Vol. 19, No. 2, pp. 131–146. https://doi.org/10.1080/02726349908908631

[50]. Jacobi, C. G. J., (1845), Über eine Neue Auflösungsart der bei der Methode der Kleinsten Quadrate Vorkommenden Linearen Gleichungen, *Astronomische Nachrichten*, Vol. 22, pp. 297–306.

[51]. Axelsson, O., (1973), A Generalized SSOR Method, *BIT Numerical Mathematics*, Vol. 13, No. 4, pp. 443–467.

[52]. Ferronato, M. (2012), Preconditioning for Sparse Linear Systems at the Dawn of the 21st Century: History, Current Developments, and Future Perspectives, *ISRN Applied Mathematics*, Vol. 2012, Article ID 127647, 49 pages.

[53]. Pestana, J., Wathen, A. J. and Heil, M., (2016), Efficient Block Preconditioning for a C^1 Finite Element Discretization of the Stokes Problem, *SIAM Journal on Scientific Computing*, Vol. 38, No. 3, pp. A1761–A1784.

[54]. Bridson, R., & Greif, C. (2005). *A multi-preconditioned conjugate gradient algorithm*. Department of Computer Science, University of British Columbia, Vancouver, BC. Retrieved from. https://www.cs.ubc.ca/~greif/pubs/mpcg.pdf

[55]. Kothari, H., Nestola, M. G. C., Favino, M. and Krause, R., (2024), *Integrating Multi-Preconditioned Conjugate Gradient with Additive Multigrid Strategy*, arXiv preprint.

[56]. Emans, M. (2010). Performance of parallel AMG-preconditioners in CFD-codes for weakly compressible flows. *Parallel Computing*, Vol. 36, Nos. 5–6, 326–338

[57]. Adams, M., and Taylor, R. L. (2000), Parallel Multigrid Solvers for 3D-Unstructured Large Deformation Elasticity and Plasticity Finite Element Problems, *Finite Elements in Analysis and Design*, Vol. 36, Issues 3–4, pp. 197–214.

[58]. Tielen, R., Möller, M., Göddeke, D., & Vuik, C. (2020). p-Multigrid methods and their comparison to h-multigrid methods within Isogeometric Analysis. *Computer Methods in Applied Mechanics and Engineering*, 372, 113347.

[59]. Zhao, L., Feng, C., Zhang, C.-S., & Shu, S. (2022). Parallel multi-stage preconditioners with adaptive setup for the black oil model. *Computers & Geosciences*, Vol. 168, 105230.

[60]. Baker, A. H., Klawonn, A., Kolev, T., Lanser, M., Rheinbach, O., & Yang, U. M. (2016). Scalability of classical algebraic multigrid for elasticity to half a million parallel tasks. In H. J. Bungartz, P. Neumann, & W. Nagel (Eds.), *Software for Exascale Computing – SPPEXA 2013–2015* (Lecture Notes in Computational Science and Engineering, Vol. 113, pp. 421–449). Cham: Springer. https://doi.org/10.1007/978-3-319-40528-5_6

[61]. Ducharne, B., and Sebald, G., (2025), Analytical Expressions of the Dynamic Magnetic Power Loss Under Alternating or Rotating Magnetic Field, *Mathematics and Computers in Simulation*, Vol. 229, pp. 340–349. https://doi.org/10.1016/j.matcom.2024.10.009

[62]. Berger, M. J. and Colella, P., (1989), Local Adaptive Mesh Refinement for Shock Hydrodynamics, *Journal of Computational Physics*, Vol. 82, No. 1, pp. 64–84.

[63]. Lu, B. Z., Zhou, Y. C., Holst, M. J. and McCammon, J. A., (2008), Recent Progress in Numerical Methods for the Poisson–Boltzmann Equation in Biophysical Applications, *Communications in Computational Physics*, Vol. 3, No. 5, pp. 973–1009.

[64]. Zheng, W., Chen, Y., and Zhou, J., (2024), A Legendre Spectral Method for Multidimensional Partial Volterra Integro-Differential Equations, *Journal of Computational and Applied Mathematics*, Vol. 436, 115302. https://doi.org/10.1016/j.cam.2023.115302

[65]. Marburg, S., (2018), Boundary Element Method for Time-Harmonic Acoustic Problems, in Kaltenbacher, M. (Ed.), *Computational Acoustics*, CISM International Centre for Mechanical Sciences, Vol. 579, Springer, Cham, pp. 43–119.

[66]. Sommerfeld, A., 1949, *Partial Differential Equations in Physics*, Vol. 1, Academic Press.

[67]. Keller, J. B., and Givoli, D., (1989), Exact Non-Reflecting Boundary Conditions, *Journal of Computational Physics*, Vol. 82, No. 1, pp. 172–192. https://doi.org/10.1016/0021-9991(89)90041-7

[68]. Dokumaci, E., (1995), Prediction of the Effects of Entropy Fluctuations on Sound Radiation from Vibrating Bodies Using an Integral Equation Approach, *Journal of Sound and Vibration*, Vol. 186, No. 5, pp. 805–819. https://doi.org/10.1006/jsvi.1995.0489

[69]. Karra, C., and Ben Tahar, M., (1997), An Integral Equation Formulation for Boundary Element Analysis of Propagation in Viscothermal Fluids, *The Journal of the Acoustical Society of America*, Vol. 102, No. 3, pp. 1311–1318. https://doi.org/10.1121/1.420050

[70]. Paltorp, M., Henríquez, V. C., Aage, N., and Andersen, P. R., (2024), A Reduced Order Series Expansion for the BEM Incorporating the Boundary Layer Impedance Condition, *Journal of Theoretical and Computational Acoustics*, Vol. 32, No. 2, 2350012. https://doi.org/10.1142/S2591728523500123

[71]. Andersen, P. R., Henríquez, V. C., and Aage, N., (2023), On the Validity of Numerical Models for Viscothermal Losses in Structural Optimization for Micro-Acoustics, *Journal of Sound and Vibration*, Vol. 547, 117455.

[72]. Shaaban, A. M., Preuss, S., and Marburg, S., (2025), Three-Dimensional Isogeometric Boundary Element Method for Acoustic Problems With Viscothermal Losses, *Computer Methods in Applied Mechanics and Engineering*, Vol. 438, Part B, 117843. https://doi.org/10.1016/j.cma.2025.117843

[73]. Jaśkowiec, J., and Pluciński, P., (2025), Orthogonalization in High-Order Finite Element Method, *Computers & Structures*, Vol. 311, 107692.

[74]. Mitchell, W. F., (2015), How High a Degree Is High Enough for High-Order Finite Elements? *Procedia Computer Science*, Vol. 51, pp. 246–255.

[75]. Atallah, N. M., Mittal, K., Scovazzi, G., and Tomov, V. Z. (2025), A High-Order Shifted Interface Method for Lagrangian Shock Hydrodynamics, *Journal of Computational Physics*, Vol. 523, 113637.

[76]. Ainsworth, M., and Jiang, S., (2019), Preconditioning the Mass Matrix for High-Order Finite Element Approximation on Triangles, *SIAM Journal on Numerical Analysis*, Vol. 57, No. 1, pp. 355–377.

[77]. Whiteley, J. P., (2017), A Preconditioner for the Finite Element Computation of Incompressible, Nonlinear Elastic Deformations, *Computational Mechanics*, Vol. 60, No. 4, pp. 683–692.*2*.

[78]. Phoon, K. K., Toh, K. C., Chan, S. H., and Lee, F. H., (2002), An Efficient Diagonal Preconditioner for Finite Element Solution of Biot's Consolidation Equations, *International Journal for Numerical Methods in Engineering*, Vol. 55, pp. 377–400.

[79]. Dutt, P., Biswas, P. and Raju, G.N., (2008), Preconditioners for Spectral Element Methods for Elliptic and Parabolic Problems, *Journal of Computational and Applied Mathematics*, Vol. 215, No. 1, pp. 152–166.

[80]. Andrej, J., Atallah, N., Backer, J.-P., Camier, J.-S., Copeland, D., Dobrev, V.A., Dudouit, Y., Duswald, T., Keith, B., Kim, D., Kolev, T.V., Lazarov, B.S., Mittal, K., Pazner, W., Petrides, S., Shiraiwa, S., Stowell, M. and Tomov, V.Z., (2024), High-Performance Finite

Elements with MFEM, *The International Journal of High Performance Computing Applications*, Vol. 38, pp. 447–467.

[81]. Orszag, S. A. (1980), Spectral Methods for Problems in Complex Geometries, *Journal of Computational Physics*, Vol. 37, No. 1, pp. 70–92.

[82]. Pazner, W., (2020), Efficient Low-Order Refined Preconditioners for High-Order Matrix-Free Continuous and Discontinuous Galerkin Methods, *SIAM Journal on Scientific Computing*, Vol. 42, No. 5, pp. A3055–A3083.

[83]. Pazner, W. and Kolev, T., (2022), Uniform Subspace Correction Preconditioners for Discontinuous Galerkin Methods with hp-Refinement, *Communications on Applied Mathematics and Computation*, Vol. 4, No. 2, pp. 697–727.

[84]. Fischer, P. F. (1997). An overlapping Schwarz method for spectral element solution of the incompressible Navier–Stokes equations. *Journal of Computational Physics*, Vol. 133, No. 1, pp. 84–101.

[85]. Deville, M. O. and Mund, E. H. (1990), Finite-Element Preconditioning for Pseudospectral Solutions of Elliptic Problems, *SIAM Journal on Scientific and Statistical Computing*, Vol. 11, No. 2, pp. 311–342.

[86]. Kalchev, D. Z. and Vassilevski, P. S. (2020), A Condensed Constrained Nonconforming Mortar-Based Approach for Preconditioning Finite Element Discretization Problems, *SIAM Journal on Scientific Computing*, Vol. 42, No. 5, pp. A3136–A3156.

[87]. Langer, U., Steinbach, O., and Yang, H. (2021), Robust Discretization and Solvers for Elliptic Optimal Control Problems with Energy Regularization, *arXiv preprint*, arXiv:2102.03515.

[88]. Pazner, W., Kolev, T., and Dohrmann, C. R. (2023), Low-Order Preconditioning for the High-Order Finite Element de Rham Complex, *SIAM Journal on Scientific Computing*, Vol. 45, No. 2, pp. A675–A702.

[89]. Kwon, J., Ryu, S., Kim, P., and Kim, S. D. (2012), Finite Element Preconditioning on Spectral Element Discretizations for Coupled Elliptic Equations, *Journal of Applied Mathematics*, Vol. 2012, Article ID 245051, pp. 1–16.

[90]. Zhou, Z. (2025), Accelerating Posterior Sampling for Scalable Gaussian Process Model, *Conference Proceedings* (in press). Available at: https://api.semanticscholar.org/CorpusID:278326952

[91]. Kim, S., St-Cyr, A., and Kim, S. D. (2013), Finite Difference Preconditioners for Legendre-Based Spectral Element Methods on Elliptic Boundary Value Problems, *Applied Mathematics*, Vol. 4, No. 8, pp. 838–847.

[92]. Collier, N. O., Dalcin, L., Pardo, D., and Calo, V. M. (2012), The Cost of Continuity: Performance of Iterative Solvers on Isogeometric Finite Elements, *arXiv preprint*, arXiv:1206.2948.

[93]. Siahkolaei, T. S., and Salkuyeh, D. K. (2019), A Preconditioned SSOR Iteration Method for Solving Complex Symmetric System of Linear Equations, *Numerical Algebra, Control and Optimization*.

[94]. Liu, C.-S., El-Zahar, E. R., and Chang, C.-W. (2023), Dynamical Optimal Values of Parameters in the SSOR, AOR, and SAOR Testing Using Poisson Linear Equations, *Mathematics*, Vol. 11, No. 18, 3828.

[95]. Alsalti-Baldellou, À., Franceschini, A., Mazzucco, G., and Janna, C. (2024), Efficient AMG Reduction-Based Preconditioners for Structural Mechanics, *Computer Methods in Applied Mechanics and Engineering*, Vol. 431, 117249

[96]. Olson, L. (2007). Algebraic multigrid preconditioning of high-order spectral elements for elliptic problems on a simplicial mesh. *SIAM Journal on Scientific Computing*, Vol. 29, No. 5, pp. 2189–2209.

[97]. Di Pietro, D. A., Matalon, P., Mycek, P., & Rüde, U. (2023). High-order multigrid strategies for hybrid high-order discretizations of elliptic equations. *Numerical Linear Algebra with Applications*, Vol. 30, No. 1, e2456

[98]. Iwamura, C., Costa, F. S., Sbarski, I., Easton, A., & Li, N. (2003). An efficient algebraic multigrid preconditioned conjugate gradient solver. *Computer Methods in Applied Mechanics and Engineering*, Vol. 192, Issues 20–21, pp. 2299–2318

[99]. Napov, A., & Notay, Y. (2014). Algebraic multigrid for moderate order finite elements. *SIAM Journal on Scientific Computing*, Vol. 36, No. 4, A1678–A1707.

[100]. Grauschopf, T., Griebel, M., & Regler, H. (1997). Additive multilevel preconditioners based on bilinear interpolation, matrix-dependent geometric coarsening and algebraic multigrid coarsening for second-order elliptic PDEs. *Applied Numerical Mathematics*, Vol. 23, No. 1, 63–95.

[101]. Eisenträger, S., Atroshchenko, E., & Makvandi, R. (2020), On the condition number of high order finite element methods: Influence of p-refinement and mesh distortion, *Computers & Mathematics with Applications*, Vol. 80(11), pp. 2289–2339.

[102]. Nguyen, H. (2010). *p-adaptive and automatic hp-adaptive finite element methods for elliptic partial differential equations* (Doctoral dissertation, University of California, San Diego). Retrieved from

[103]. Kolditz, O., (2002), Finite Volume Method, in *Computational Methods in Environmental Fluid Mechanics*, Springer, pp. 173–190.

[104]. Briggs, W. L., Henson, V. E., & McCormick, S. F. (2000). *A multigrid tutorial* (2nd ed.). Philadelphia, PA: SIAM. ISBN 978-0-89871-462-3.

[105]. Stüben, K. (2001). A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, Vol. 128, No. 1–2, pp. 281–309.

[106]. Vallet, M.-Gabrielle, Bourgault, Y., & Fortin, M. (2002), Anisotropic Mesh Adaptation: Towards User-Independent, Mesh-Independent and Solver-Independent CFD. Part III. Unstructured Meshes, *International Journal for Numerical Methods in Fluids*, Vol. 40, Nos. 3–4, pp. 313–331.

[107]. Alauzet, F., &Loseille, A. (2016), A Decade of Progress on Anisotropic Mesh Adaptation for Computational Fluid Dynamics, *Computer-Aided Design*, Vol. 72, pp. 13–39.

[108]. Wustenberg et al. (2024, June 11, Nektar++ workshop 2024), Algorithm and solver development