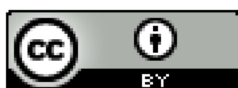
PRECONDITIONING TECHNIQUES FOR SPARSE LINEAR SYSTEMS

A DISSERTATION
SUBMITTED TO THE INSTITUTE OF COMPUTATIONAL AND
MATHEMATICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Shaked Regev
December 2022

© 2022 by Shaked Regev. All Rights Reserved.
Re-distributed by Stanford University under license with the author.



This work is licensed under a Creative Commons Attribution-3.0 United States License.
http://creativecommons.org/licenses/by/3.0/us/

This dissertation is online at: https://purl.stanford.edu/zc485bz0015

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Eric Darve, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Michael Saunders, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mary Wootters

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Lexing Ying

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format.

Abstract

We present two novel methods, SSAI and HyKKT, for sparse linear systems. The methods differ in that SSAI is meant to be an out of the box solver that is robust on many different types of Hermitian positive definite (**HPD**) linear systems and has a variant that can be used to solve general systems (including rectangular ones), but is not necessarily the best for a given linear system. This is due to the fact that SSAI in an effort to be general, ignores domain knowledge of the problem. In contrast, HyKKT is a specialized solver for very sparse symmetric indefinite linear systems with a Karush-Kuhn-Tucker (KKT) structure [100, 109]. These two methods represent the main parts of the thesis.

Chapter 2 introduces SSAI, a method for solving a Hermitian positive definite linear system $Ax = b$, where A is an explicit sparse matrix (real or complex). A sparse approximate right inverse is computed and replaced by its symmetrization M, which is used as a left-right preconditioner in a modified version of the preconditioned conjugate-gradient method (PCG), where M is modified occasionally, if necessary, to make it more positive definite. Before symmetrization, M is formed column by column and can therefore be computed in parallel with no communication except at the beginning and end. PCG requires only matrix-vector multiplications with A and M (not solving a linear system with M), and so too can be carried out in parallel. We compare it with incomplete Cholesky factorization (the gold standard for PCG) and with a direct Cholesky factorization and solve on sparse matrices from various applications and show it is robust. For least-squares problems, we implement an analogous form of preconditioned Conjugate Gradient Least-Squares (PCGLS) and show it is also robust. The contributions of the work in Chapter 2 are summarized in Section 2.2.

Chapter 3 introduces HyKKT, a solution strategy for the large indefinite linear systems arising in interior methods for nonlinear optimization. The method is suitable for implementation on hardware accelerators such as graphical processing units (GPUs). The current gold standard for sparse indefinite systems is the $\mathrm{LBL^T}$ factorization, where L is a lower triangular matrix and B is 1×1 or 2×2 block diagonal. However, this requires pivoting, which substantially increases communication cost and degrades performance on GPUs. Our approach solves a large indefinite system by solving multiple smaller positive definite systems, using an iterative solver on the Schur complement and an inner direct solve (via Cholesky factorization) within each iteration. Cholesky is stable without

pivoting, thereby reducing communication and allowing reuse of the symbolic factorization. We demonstrate the practicality of our approach on large optimal power flow problems and show that it can efficiently utilize GPUs and outperform LBL^T factorization of the full system. The contributions of the work in Chapter 3 are summarized in Section 3.2.

Acknowledgments

I am fortunate to know many people who made my PhD possible, or at least substantially more enjoyable. Although I would like to thank each and every person individually, no such list can ever be exhaustive. That being said, it doesn't hurt to try.

I thank my advisor, Prof Eric Darve, who is the one who got me interested in the world of numerical linear algebra in my first year at Stanford through his course and this alone would already be a lot. Later, as my advisor, Eric supported me throughout my research and encouraged me to pursue the avenues that interested me most, while always providing guidance, support, and constructive feedback.

My co-advisor Prof Michael Saunders took me on even though at the time I turned to him, his "last" student had just graduated. Michael's energy and optimism kept me going during difficult times during my PhD. His meticulous proofreading and suggestions were vital to my research and my writing.

I would like to thank my committee members Profs Mary Wooters and Lexing Ying for reading my thesis and providing feedback and the chair of my oral defense committee Prof Daniel Tartakovsky.

I thank my mentor Dr Slaven Peleš, for believing in my at the time yet-to-be-demonstrated software engineering capabilities to recommend that Pacific Northwest National Lab fund my PhD research. Slaven recognized my love for numerical linear algebra and constantly pushed me to develop the C++ skills to match. He offered me an endless stream of technical advice and life wisdom throughout the years.

I am grateful for Dr Kasia Świrydowicz who taught me how to navigate the intricacies of academia culture and how to efficiently debug code. Many times when I had a bug that I thought was impossible to find, Kasia proved me wrong. I have no idea what shape my code would be without her, but I'm sure it would not be nearly as good.

I am thankful for Christopher Oehmen, Cameron Rutherford, Asher Mancinelli, and all the folks at Pacific Northwest National Lab, for giving me the opportunity to work on the Exascale project and their mentorship and collaboration.

I thank my first advisor at Stanford, Prof Alison Marsden, for supporting me in my first years at Stanford, pushing me to find research topics that interest me and ultimately advising me to make

the switch to numerical linear algebra.

I thank my Master's degree advisor, Prof Oded Farago, for introducing me to research in such a way that made me love it so much.

Many ICME "elders" were a source of inspiration to me and provided constant technical and administrative advice. I'd particularly like to mention Cindy, Alex, Bazyli, Nurbek, Steven, Leopold, and Abeynaya.

I am grateful for my ICME cohort and especially appreciate cooking up a storm, going on much-needed trips, and having seven-layer inside jokes with Dan and Martin, endlessly philosophising with Phil, Aldo and Ruilin's deadpan humor, and hour-long deep conversations with Amel when we should've definitely been working. I also thank Izzy for being honorarily in my cohort and for being the most bottomless source of positivity that I have ever seen.

I was incredibly lucky to get Philipp Muscher as a roommate when I first arrived at Stanford, even though that would've never happened if he were honest with himself about his bonkers sleeping habits. Philipp taught me the ins and outs of campus life, went on trips with me, and taught me the value of spontaneity. During our time living together and even after he graduated, Philipp was always there for me and even welcomed me along with the wonderful people at the Eyja co-op when I needed a place to crash.

I thank Dana and Ilai Bistritz, my Israelis at Stanford family, for all the trips, laughs, dinners, board game nights, and conversations. They truly made the campus feel more like home and were always there to lend a helping hand or listening ear.

I thank Bernardo Casares for supporting me through my lowest lows and pushing me towards my highest highs. Bernardo always made sure I stayed in shape, both physically and mentally. He believed in me at times that I could not believe in myself. Bernardo and the rest of the Casares family even gave me the perfect end to my PhD by welcoming me to their home and allowing me to see their beautiful country Ecuador.

There are many people who but for their presence in my life, I would not have gotten into Stanford, or perhaps not finished the PhD. However, no one was quite as directly responsible as Frona and Ted Kahn. Frona and Ted urged me to shoot for the stars and helped me hone my essay writing skills. During my time at Stanford they provided me constantly with emotional and logistical support.

Growing up walking distance from my grandparents was an experience I wish for every child. Carol and Ian, my grandparents longtime friends or my "adopted-grandparents", provided me with the closest thing possible in Palo Alto. They were a constant source of positivity, warmth and inspiration.

I thank the Doitel-Dickman family for welcoming me into their home with open arms. From the first moment when Ben came to pick me, some random son of his aunt's neighbor, up for Shabbat, I could tell this family was special. Shabbat dinners with them recharged me after long weeks. Noa

and Ofer are parental figures to me, always there to listen and offer advice or help me out when I need it most. I thank Hagar, Michael, Uri, Shai, Tal, and Ayelet for bringing me into their home during my nomadic period and for our all of our fiercely competitive ticket to ride games and good times. I also thank Shai for being my mentee in what was probably the most successful summer internship for a high school student in the history of the world, or as the kids say: a massive dub.

I thank my aunts, uncles, and cousins of various degrees for their support from far and from near and for inquiring about me and being involved in my life. My uncle Maor encouraged me to apply for a PhD in the US and gave me a not so gentle nudge to hurry up and accept Stanford's insultingly late admission offer. He and my "I feel weird calling her my aunt" Kate talked me through endless hardships with work and relationships, provided me with some much needed family time in the Bay, and allowed me the privilege of being a cousin acting as an uncle to their sons Bash and Bo.

I thank my chronologically little but physically big brothers Alon and Yuval for celebrating our differences and teaching me new ways to look at the world. Alon taught me to never sell myself short and that anything is possible if you want it enough. From Yuval I learned that nothing is worth losing your cool over and that listening is at least twice as important as talking. Even though we saw each other relatively infrequently over these five years, I treasure those board game matches, hikes, and trash talk sessions and they were crucial to my sanity. Thank you for the dozens of hours spent in transit for the sole purpose of seeing me.

I thank my grandparents Zion, Ilan, Rachel, and Eva for shaping who I am today. From Zion I learned that family is everything and guests are always invited, even if implicitly. Ilan taught me that humor is the best medicine, that there is no better pass time than learning more about the world, and that you should never be ashamed of correcting course. I only regret that he is not here to see this day as I know how much it meant to him and that in a very practical sense he was the closest person to me who knew what a PhD entails. From Rachel I learned to count my blessings and that cooking is an art, before I later learned that it is also a science. Eva taught me that humor is the best coping mechanism and to listen patiently to what other people think and then do what you think is best anyway.

My parents Shira and Dror, AKA Ima and Aba, supported me from day one, in the quite literal sense. They gave me the love of learning and curiosity about the world that I have today and advised me through countless challenges. They physically and emotionally helped me move to Stanford and supported me pursuing my goals, even though it meant being far away from them. They planned family and more intimate vacations to recharge our batteries and do what we love together. My mom taught me that if you are not happy with your outcomes, you should reexamine your choices, even if you think nothing is wrong with them. My dad taught me that just because you're in the middle of a forest or climbing a mountain, doesn't mean you also can't hash out the mathematics for an interesting problem. I am always flattered by how highly he thinks of my mathematical and technical abilities, even though he is sometimes wrong. These are just examples, though basically

everything I know was taught by them directly or through the curiosity they inspired.

Last but not least, I would like to thank Ilana for being the most integral part of my life in these 5 years. Thank you for sharing our happiest moments together, for teaching me that selflessness knows no bounds, for bringing me out of my comfort zone and teaching me to properly cohabitate, for pushing me to learn how to cook, for pulling me through trying times at work and in the world, for knowing me better than I know myself, and for using that knowledge to help me in ways I could not even know to ask for. The role we play in each other's lives has shifted over the years, but our friendship has remained constant. It hasn't always been easy, but it's always been and hopefully will be worthwhile.

Contents

Abstract **iv**

Acknowledgments **vi**

1 Introduction **1**

1.1 Sparse Hermitian linear systems . . . 1

1.1.1 Direct solution methods . . . 2

1.1.2 Iterative solution methods . . . 3

1.1.3 Methods with iterative and direct components . . . 4

1.2 Sparse least-squares problems . . . 5

1.2.1 Direct solution methods . . . 5

1.2.2 Iterative solution methods . . . 5

1.2.3 Methods with iterative and direct components . . . 5

1.3 Scaling . . . 6

1.3.1 Max scaling . . . 7

1.3.2 Diagonal scaling . . . 7

1.3.3 Ruiz scaling . . . 7

1.3.4 Norm scaling . . . 8

1.4 Reordering . . . 8

1.4.1 Fill-in reduction . . . 8

1.4.2 Numerical stability via pivoting . . . 10

1.5 Preconditioning for iterative solutions of sparse HPD systems . . . 12

1.6 Direct methods for solving sparse symmetric indefinite KKT linear systems . . . 15

1.6.1 Motivation . . . 19

1.7 Parallel computing . . . 19

1.7.1 Bounds on speedup . . . 20

1.7.2 Performance limiters . . . 21

1.7.3 Types of parallelism . . . 22

1.8 Overview . . . 22

2 SSAI: A symmetric sparse approximate inverse preconditioner **23**

2.1 Overview . . . 23
2.2 Contributions . . . 24
2.3 Introduction . . . 24
2.4 A PCG algorithm for HPD linear systems . . . 26
2.4.1 Approximate solution of $Am_j = e_j$ . . . 28
2.4.2 Modified PCG . . . 29
2.5 Numerical results on HPD systems . . . 30
2.5.1 Matrices from SuiteSparse . . . 32
2.5.2 SDDM matrices . . . 33
2.5.3 Scale-up for Trefethen challenge matrices . . . 33
2.5.4 Scale-up for finite-element linear elasticity matrices . . . 34
2.5.5 Scale-up for finite-volume petroleum engineering matrices . . . 34
2.5.6 Scale-up for finite-element ice-sheet matrices . . . 34
2.6 Least-squares problems . . . 34
2.7 Sparsity structure of A . . . 36
2.7.1 Performance of ichol vs SSAI . . . 37
2.8 Parallelism and complexity . . . 38
2.9 Discussion . . . 39
2.9.1 Comparison of preconditioners for iterative methods for HPD problems . . . 40
2.9.2 Indefinite A . . . 40
2.9.3 MINRES and preconditioners for indefinite A . . . 40
2.9.4 Signature matrices . . . 41
2.10 Summary . . . 41

3 HyKKT: A Hybrid Method for Solving KKT Linear Systems **47**

3.1 Overview . . . 47
3.2 Contributions . . . 48
3.3 Introduction . . . 48
3.4 Nonlinear optimization problem . . . 50
3.5 Solving KKT linear systems . . . 52
3.6 A block 2×2 system solution method . . . 54
3.6.1 A hybrid solver with minimal regularization . . . 55
3.6.2 Guaranteed descent direction . . . 57
3.6.3 Convergence for large γ . . . 59
3.6.4 Summary of equations solved . . . 60

3.7 Practicality demonstration 60
3.7.1 γ selection 61
3.7.2 Results for larger matrices 62
3.7.3 Reordering H_γ 64
3.8 Comparison with LBL^T 64
3.9 Iterative vs. direct solve with H_δ in Algorithm 7 67
3.10 Summary 67
3.11 Appendix A: Additional OPF matrix results 71
3.12 Appendix B: HyKKT implementation 71
3.12.1 Classes 71
3.12.2 Definitions 73
3.12.3 Functions 74
3.12.4 Utilities 74
4 Summary **76**

List of Tables

2.1 Abbreviations for SuiteSparse problem fields. 31

2.2 SuiteSparse $Ax = b$, $\text{nnz}(A) > 1M$. **R** is the number of restarts in Algorithm 5 (PCG + SSAI). SSAI succeeded on all 19 problems, while ichol failed on 10, and '\' failed on 5. 43

2.3 StocF-1465 $Ax = b$ with varying `lfil` and `itmax`. **R** is the number of restarts in Algorithm 5. **Obj** $= \nu\,[\text{nnz}(A) + \text{nnz}(M)]*$`Itns` is a normalized measure of operations in PCG. Increasing `lfil` and `itmax` helps the overall performance. 44

2.4 Comparison of ichol and SSAI with rchol on SDDM systems $Ax = b$ from SuiteSparse. rchol requires A to be SDDM. A was not scaled for rchol because only shallow_water2 remains SDDM with scaling. Scaling was performed on ichol and SSAI for better efficiency. rchol performed well on these three examples (especially on ecology2), but failed on a fourth SDDM matrix (Andrews, therefore omitted). 44

2.5 Scale-up for Combinatorics $Ax = b$. PCG + SSAI required no restarts. ichol and SSAI succeeded on all 3 problems, and '\' failed on the largest 2. 44

2.6 Scale-up for ordered grid structural mechanics $Ax = b$. PCG + SSAI required no restarts. ichol and SSAI succeeded on all 3 problems, and '\' failed on the largest 1. 45

2.7 Scale-up for finite-volume $Ax = b$. PCG + SSAI required no restarts. ichol and SSAI succeeded on all 6 problems, and '\' failed on the largest 4. 45

2.8 Scale-up for '\' and SSAI on unordered grid structural mechanics $Ax = b$. SSAI succeeded on all 4 problems, ichol failed on all 4, and '\' failed on 2. 45

2.9 SuiteSparse least-squares problems $\min \|Ax - b\|$ with $A \in \mathbb{R}^{m \times n}$. SSAI and ichol are used with PCGLS. SSAI succeeded on all 9 problems, ichol failed on 4, and '\' failed on 2. 46

2.10 Comparison of preconditioners for iterative methods on HPD problems. The rchol and spaND methods have further restrictions. rchol requires the matrix to be SDD, or SDDM if we don't want to double the size of our system, and spaND requires the interfaces between groups of variables to be relatively sparse (i.e., there are only short range interactions between variables). Note that SAINV adds a matrix to A during computation of M to ensure that M is positive definite. This means that A may be modified too much or even unnecessarily unless the process is attempted multiple times. (In contrast, SSAI modifies M, not A, and only as needed.) Sparsity refers to the sparsity structure of the preconditioner and whether it is supplied by the user (fixed) or calculated within the method (flexible). Parallelism refers to the computation of the preconditioner (not its application). FSM says if it can be used with fixed-storage iterative methods like CG and MINRES. Application refers to how the preconditioner is applied. (Product preconditioners are more desirable because they can be applied in embarrassingly parallel fashion, whereas triangular solves involve waiting and communication between dependent equations.) The color blue is used to indicate a desirable property, whereas the color purple is used to indicate an undesirable property. Properties left in black are somewhere in between. 46

3.1 Notation. SP(S)D stands for symmetric positive (semi)definite. Matrix/vector norms are 2-norms unless stated otherwise. For a symmetric matrix M, we define its smallest eigenvalue restricted to the null space of a Jacobian J: $\lambda_{\min}(M_{|\text{null}(J)}) \equiv \min_{\|v\|=1,\ v\in\text{null}(J)} v^T M v$. 50
3.2 Summary of the methods used for solving various equations in the chapter. 60
3.3 Characteristics of the five tested optimization problems, each generating sequences of linear systems $K_k \Delta x_k = r_k$ (3.3) of dimension N. Numbers are rounded to 3 digits. K and M signify 10^3 and 10^6. 61
3.4 Accelerator devices and compilers used. 64
3.5 Dimensions, number of nonzeros, and factorization densities (number of nonzeros in the factors per row) for solving (3.3) directly with LBL$^{\text{T}}$ (N, nnz_L, ρ_L respectively) and for solving (3.6) with Cholesky (n_x, nnz_C, ρ_C respectively). Numbers are rounded to 3 digits. K and M signify 10^3 and 10^6. In all cases, $\rho_C < \rho_L$ and $n_x < N/2$. . . . 65

3.6 Average times (in seconds) for solving (3.3) directly on a CPU with LBL^T (via MA57 [58]) or for solving one H_δ linear system with supernodal Cholesky via Cholmod (CM) in SuiteSparse [37], or Cholesky via cuSolver [1] (CS) using the routine `csrlsvchol`, each on a CPU and on a GPU. Cholesky on a GPU is quicker than LBL^T on a CPU by an increasingly large ratio. The GPU is not actually utilized for small problems with CM, because CM has internal logic that decides what to offload to the GPU, and there is a minimum problem size for the GPU to be utilized. All runs are on Newell [128]. 67

3.7 Average times (in seconds) for solving sequences of systems (3.3) directly on a CPU with LBL^T (via MA57 [58]) or on a GPU using our hybrid method. The latter is split into forming (3.7)–(3.8), analysis and factorization phases, and multiple solves. (The routines `cusolverSpCreateCsrcholInfo`, `cusolverSpXcsrcholAnalysis`, `cusolver-SpDcsrcholFactor`, `cusolverSpDcsrcholZeroPivot`, and `cusolverSpDcsrcholSolve` are used to allow for the easy solution of systems with multiple RHS.) "Forming (3.7)–(3.8)" shows the times needed for the matrix products, not including the memory allocation phase (which is the most expensive). However, since the nonzero structure is constant in a given problem, this need only be done once. Symbolic analysis is needed only once for the whole sequence. Factorization happens once for each matrix. The solve phase is the total time for Lines 15-17 in Algorithm 7 with a CG tolerance of 10^{-12} on Line 16 plus the recovery of the solution to the original problem. The results show that HyKKT, without optimization of the code and kernels, outperforms LBL^T on the largest series (US Eastern Interconnection grid) by a factor of more than 3 on a single matrix, and more than 10 on a whole series, because the cost of symbolic analysis can be amortized over the series. All runs are on Deception [128]. 68

3.8 Densification of the problem for cases where the direct-iterative method is viable. Numbers are rounded to 3 digits. K and M signify 10^3 and 10^6. $\text{nnz}_{\text{op}}(H_\delta) = \text{nnz}(\widetilde{H}) + 2\,\text{nnz}(J) + n_x$ is the number of multiplications when H_δ is applied as an operator, and $\text{nnz}_{\text{fac}}(H_\delta) = 2\,\text{nnz}(L)$ is the number of multiplications for solving systems with H_δ. The ratio $\text{nnz}_{\text{fac}}(H_\delta)/\text{nnz}_{\text{op}}(H_\delta)$ is only about 2 in all cases. . . . 69

List of Figures

1.1 (Left) A matrix that is completely dense when factorized. (Right) A matrix with no fill-in when factorized. 9

2.1 Sparsity structures of some square HPD problems. Although exact Cholesky factors would be essentially dense (within the outermost band), ichol and SSAI both perform well. 36
2.2 Sparsity structures of A in rectangular problems, compared to the sparsity structure of $A^H A$. The structure makes it clear why an iterative method would outperform a direct method on `ch8-8-b`, and vice versa for `Hardesty`. 37
2.3 Sparsity pattern entry magnitude of `1138_bus` and `sts4098`. The larger entries are concentrated near the diagonal. On `1138_bus`, SSAI cannot choose easily among the many entries r_i of similar magnitude. 38

3.1 Optimization solver workflow showing invocation of key linear solver functions. The top feedback loop represents the main optimization solver iteration loop. The bottom feedback loop is the optimization solver control mechanism to regularize the underlying problem when necessary. 53
3.2 Illinois grid: (a) CG iterations on Eq. (3.7) with varying γ. $\gamma \geq 10^3$ gives good convergence. The mean number of iterations for $\gamma = 10^4$ is 9.4. (b) Sorted eigenvalues of $S_\gamma \equiv \gamma J_c H_\gamma^{-1} J_c^T$ in (3.9) matrix 22, for $\gamma = 10^4$. The eigenvalues are clustered close to 1. 61
3.3 Illinois grid (3.3), with varying γ in (3.6): (a) Backward error (**BE**) and (b) relative residual (**RR**). (a) $\gamma \leq 10^4$ gives results close to machine precision. (b) $\gamma \leq 10^4$ has RR $\leq 10^{-8}$. 62
3.4 South Carolina grid: δ_1 for $\gamma = 10^4$. For other values of γ the graph was similar. Except for the first few and last few matrices, $\delta \approx 1$, meaning the required regularization would make the solution too inaccurate. The value of 0 is omitted on the log-scale. . . 63

3.5 US Western Interconnection grid with $\gamma = 10^6$ in (3.6): (a) CG iterations on Eq. (3.7). The mean number of iterations is 17. (b) BE and RR for the sequence. The BE for (3.3) is less than 10^{-10}, except for matrix 4. 64
3.6 US Eastern Interconnection grid with $\gamma = 10^6$ in (3.6): (a) CG iterations on Eq. (3.7). The mean number of iterations is 13.1. (b) BE and RR for (3.3) and (3.4). The BE for (3.3) is less than 10^{-10}. 65
3.7 Illinois grid matrix 22: (a) Approximate minimum degree ordering of $\text{chol}(H_\gamma)$ is sparser than (b) Nested dissection ordering of $\text{chol}(H_\gamma)$. Both orderings are calculated in MATLAB. 66
3.8 Illinois grid (3.4) with varying γ in (3.6): (a) BE, $\gamma \le 10^4$ gives results close to machine precision. (b) RR, $\gamma \le 10^4$ has RR $\le 10^{-8}$. 70
3.9 Texas grid with $\gamma = 10^4$: (a) CG iterations on Eq. (3.7). The mean number of iterations is 11.1. (b) BE and RR for (3.3) and (3.4). The BEs are roughly machine precision and the RRs are less than 10^{-8}. 70
3.10 A summary of the workflow of the linear solver of HyKKT 72

Chapter 1

Introduction

We say that an $m \times n$ matrix A is sparse if $A_{ij} = 0$ for many pairs (i, j). Often this means that regardless of the matrix size, the average number of nonzeros per row is $O(1)$. The number could grow slowly with m or n, for example $O(\log(n))$. For purposes of solving sparse systems of equations $Ax = b$ or least-squares problems $\min \|Ax-b\|_2^2$, the matrix A is typically stored in compressed sparse row or column (**CSR** or **CSC**) format. CSR format is typically used in programming languages with row major memory sorting. In CSR, the row array contains in position k the index number of the first nonzero entry in the row k of A. The last ($n+1$-th) entry is $\text{nnz}(A)$ in 0-index storage and $\text{nnz}(A)+1$ in 1-index storage to handle the end of array edge case. The column array contains at index k the column number of the k-th nonzero entry of A. The value array contains at index k the value of the k-th nonzero entry of A. Similarly for CSC, we switch rows and columns in these definitions.

We use $i : j$ in the indexing of A to indicate all the indices between i and j. For example $A_{1:2,3:4}$ indicates the submatrix of A that includes the first and second rows of the third and fourth columns. We use : alone as shorthand for all rows (or columns). For example, $A_{1,:}$ is the entire first row of A.

Typically, solving linear systems requires preprocessing, which in almost all cases includes the scaling of A (further covered in Section 1.3) and in the case of direct solvers (or those with direct components) includes reordering the unknowns (further covered in Section 1.4.1). The number could grow slowly with m or n, for example $O(\log(n))$.

1.1 Sparse Hermitian linear systems

The majority of the thesis revolves around sparse Hermitian $n \times n$ linear systems $Ax = b$. These arise in a wide variety of scientific applications such as optimal power flow models, portfolio optimization, computational fluid dynamics, power networks, economics, material analysis, structural analysis,

statistics, circuit simulation, computer vision, model reduction, electromagnetics, acoustics, combinatorics, undirected graphs, and heat or mass transfer. We say that a system is Hermitian if $\forall i, j$ we have $A_{ij} = A^*_{ji}$, where $*$ denotes the complex conjugate. For real symmetric systems, the $*$ can be dropped. Generally speaking, these systems can be solved directly, via iterative methods, or some combination of the two.

1.1.1 Direct solution methods

Direct solution methods are typically deployed by taking a Cholesky ($A = LL^H$) (see Section 1.1.1 for full algorithm) or $A = \mathrm{LBL^T}$ (see Section 1.1.1 for full algorithm) factorization, where L is a lower triangular matrix and B is 1×1 or 2×2 block diagonal. Cholesky works only for positive definite matrices, while $\mathrm{LBL^T}$ is more general. Each factorization can be broken down into the symbolic factorization and numerical factorization. The first depends only on the nonzero structure of A, and so can be reused for problems where the structure of A (the nonzero pattern) stays the same and only the values change (such as in HyKKT in Chapter 3). Once these factorizations are obtained, multiple systems with the same A and different bs can be solved efficiently using triangular solves with L and L^H (L^T), and small 1×1 or 2×2 system solves with B. This method relies on the fact that the factorizations of A remain sparse when A is permuted symmetrically in some helpful way. For a given (i, j), when $A_{ij} = 0$ but $L_{ij} \neq 0$, we say that the (i, j) entry is the result of fill-in. When there is substantial fill-in, the computation can scale cubically in the size of the system [80, 163]. For this reason, while direct solves can be made accurate to machine precision, they may not be computationally viable. A key difference between positive definite and indefinite matrices is that the first can be factorized stably for any ordering of the rows and columns, while the second may require pivoting (switching of rows and columns) in order to preserve stability [48]. This means that when factorizing a positive definite matrix, one need only optimize over reducing the fill-in, whereas numerical considerations play a part when factorizing an indefinite matrix. In an effort to reduce the amount of pivoting and avoid pivoting (which is expensive in CSR or CSC storage), sometimes the symmetry of an indefinite matrix will be ignored and broken, so that LU factorization, where U is an upper triangular matrix, is used instead of $\mathrm{LBL^T}$.

LU factorization for general matrices

Algorithm 1 produces an in-place (overwriting A) LU factorization without pivoting (covered in Section 1.4.2). The diagonal of L is implicitly set to 1.

$\mathrm{LBL^T}$ factorization for Hermitian matrices

The recursive relations in Eq. (1.1) produce an $\mathrm{LBL^T}$ factorization without pivoting. It is related to LU factorization for Hermitian matrices with U defined as DL^H (DL^T for real matrices).

Algorithm 1 In place LU factorization with no pivoting.

```
1: for i ← 1 to n do
2:     for j ← 1 to i − 1 do
3:         A_ij ← (A_ij − A_{i,1:j−1} A_{1:j−1,j}) / A_jj
4:     end for
5:     A_{i,i:n} ← A_{i,i:n} − A_{i,1:i−1} A_{1:i−1,i:n}
6: end for
```

$$D_{jj} = A_{jj} - \sum_{k=1}^{j-1} L_{jk} D_{kk} L_{jk}^H \tag{1.1a}$$

$$L_{ij} = D_{jj}^{-1} \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} D_{kk} L_{jk}^H \right) \quad \forall i < j \tag{1.1b}$$

Note that blocks in D (and the corresponding ones in L) may be either 1×1 blocks or 2×2 blocks, where the latter are sometimes needed to preserve stability. This means j may stand for two consecutive indices.

Cholesky factorization for HPD matrices

The recursive relations in Eq. (1.2) produce a Cholesky factorization. It is related to the LBL$^{\mathrm{T}}$ factorization for HPD matrices with $L = L\sqrt{D}$, where $\sqrt{D}_{ii} = \sqrt{D_{ii}}$. Note that in the case of an HPD matrix, D is diagonal (has only 1×1 blocks) and all its entries are positive, so $\sqrt{D_{ii}}$ is well defined. The recursive formula for computing the Cholesky factorization is

$$L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk} L_{jk}^*}, \tag{1.2a}$$

$$L_{ij} = \frac{1}{L_{jj}} \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk}^* \right) \quad \forall i < j. \tag{1.2b}$$

1.1.2 Iterative solution methods

Conjugate gradient (**CG**) [92] or MINRES [122] are typical iterative solution methods for symmetric systems. These methods require only products of the matrix A with vectors, so they can be used even if A is not formed explicitly (for example, if it is a product of sparse matrices that would be dense if computed). At iteration k they search for an approximate solution $x^{(k)}$ to the system $Ax = b$ in the Krylov subspace $\mathcal{K}(A, b, k) = \mathrm{span}\{b, Ab, A^2b, \ldots, A^{k-1}b\}$. The difference is that CG minimizes the residual $r = b - Ax^{(k)}$ in the A^{-1} norm, meaning $r^T A^{-1} r$, whereas MINRES minimizes $r^T r$. If A has negative eigenvalues, $\|r^T A^{-1} r\|$ is unbounded from below, so CG can only be used for positive

definite A (or positive semi-definite A when the system is consistent) [99], but MINRES can be used with any Hermitian A. Both methods have cost $O(\text{nnz}(A))$ per iteration (though the constant for MINRES is larger). While both methods are guaranteed to find a solution in exact arithmetic in n or fewer iterations, that is usually a prohibitive cost. There are theoretical bounds on the number of iterations to convergence in terms of both the number of distinct eigenvalues of A and the condition number $\kappa(A)$:

$$\frac{\|e_k\|_A}{\|e_0\|_A} \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k, \tag{1.3}$$

where $e^k := x^{(k)} - x$ and $\kappa(A)$ is the ratio of the maximum and minimum absolute values of the eigenvalues of A. However, these bounds are loose for many As. In almost all cases, preconditioning the system, i.e., pre-multiplying it by a matrix such that the system being solved has a small condition number or few unique eigenvalues, is essential for keeping the number of iterations down. Both methods require the preconditioner to be HPD. This limitation sometimes leads to the use of GMRES [147] (which is similar to MINRES but is normally used for square unsymmetric systems and must store all Krylov vectors, not just the last three). This improves the accuracy of the method, but the storage grows linearly and the computation time per iteration grows quadratically with the iteration number.

Preconditioning is further covered in Section 1.5 and Section 2.3. SSAI in Chapter 2 is an example of a purely iterative solver, as even the preconditioner is computed iteratively.

1.1.3 Methods with iterative and direct components

There are variants for solving certain systems that combine the two methods. For instance, the solver in Chapter 3 contains an inner and outer solver. Another such variant is iterative refinement (**IR**), the process of taking the residual vector $r = b - A\tilde{x}$, where $\tilde{x}$ is the solution produced by the direct solver, and solving $A\Delta x = r$. The solution Δx is added to $\tilde{x}$. Corrections can be repeated until $\|r\|$ is smaller than some tolerance. Another version of iterative refinement takes $\tilde{x}$ as an initial guess for an iterative solver. This can improve the quality of the solution but comes at the cost of more operations. Even if the system is Hermitian, GMRES is typically used for IR due to it being more stable because it stores all of the Krylov vectors and orthogonalizes each new vector with respect to all preceding vectors. In contrast, CG and MINRES are able to limit their computational cost because they orthogonalize with respect to only the last two Krylov vectors, with loss of numerical accuracy [147]. Preconditioning can be seen as many direct solves with an inexact matrix (via an approximate inverse or an incomplete factorization where the number of fill-in entries is limited) that are used in the inner loop of an iterative solver. This is further detailed in Section 1.5 and Section 2.3.

1.2 Sparse least-squares problems

Section 2.6 covers the solution of least-squares problems $\min_x \|Ax - b\|_2$ with rectangular $A \in \mathbb{R}^{m\times n}$ and $m > n$. These problems arise, for example, in circuit simulation, computer graphics and vision, and combinatorics. Here again there are three broad classes of solution methods.

1.2.1 Direct solution methods

Direct solution methods typically apply the methods in Section 1.1 for Hermitian matrices to the normal equations $A^H Ax = A^H b$ or use QR factors of A, where Q is an orthogonal matrix and R is upper triangular (see Section 1.2.1). The disadvantage of the normal equations is that $\kappa(A^H A) = \kappa(A)^2$, which creates numerical issues due to ill-conditioning. Once QR factors are obtained, multiple systems with the same A and different bs can be solved efficiently using triangular solves with R and products with Q^H. As with Hermitian problems, keeping the factors sparse is essential. A method for doing so and obtaining an inexact factorization is detailed in [78].

QR factorization for general matrices

The QR factorization is based on computing an orthogonal basis for A, which we label as Q, and computing an upper triangular matrix R such that $A = QR$. It can be computed via the Gram-Schmidt process, Givens rotations, or Householder reflections. The reader is referred to [48, §5] for more details about these methods.

1.2.2 Iterative solution methods

Iterative solution methods typically apply the methods in Section 1.1 for Hermitian equations to the normal equations or use a least squares variant of CG (**CGLS**), which is equivalent in exact arithmetic but is more stable than applying CG to the normal equations [123, §7.1], [24, §7.4.1]. (**LSQR** and **LSMR** [70, 123] are two further least-squares solvers that are equivalent to, but more stable than, applying CG and MINRES to the normal equations.) The same preconditioning considerations apply to CGLS, LSQR, and LSMR as for CG. Section 2.6 below is an example of the use of PCGLS to solve least-squares problems.

1.2.3 Methods with iterative and direct components

As with Hermitian problems, there are variants that combine iterative and direct methods, such as IR combined with QR factorization, PCGLS, or sparse QR [78].

1.3 Scaling

The advantages of scaling a matrix A to create $\tilde{A}$ are that it provides some protection against numerical round-off errors, creates consistency, informs parameter selection, and can improve convergence. We first go into detail on these advantages and then describe some methods for scaling.

Round-off errors

Integers are stored on a computer as binary numbers, with one bit reserved for the sign. This means that with k bits, a computer can store numbers between -2^{k-1} and $2^{k-1}-1$. A floating-point number is stored on a computer as

$$\pm\left(1+\sum_{i=1}^{p-1} d_i 2^{-i}\right) 2^e, \tag{1.4}$$

where one bit is reserved for the sign, p is the precision, e is the exponent, and each d_i can be 0 or 1. While p allows for more precision, e allows for more range. If the memory allocated for each floating-point number is fixed, then there is a precision versus range tradeoff. Here we are primarily concerned with the case that p is not sufficiently large to store the number. If a number requires more digits to represent it than p, then it cannot be stored. For example, the result of $1+2^{-p}$ is 1 on a computer. This on its own does not usually create problems. However, if we then subtract 1, we get 0, even though mathematically the correct result is 2^{-p}. Rearranging the operations as $1-1+2^{-p}=2^{-p}$ does give the correct answer. If we were to divide by the result in the first case we would get undefined behavior. Furthermore, we see that addition is no longer associative, so parts of a code that rely on this can fail to produce the right results.

Consistency and parameter selection

If we solve an HPD system $\phi Ax = \phi b$ with very large (or small) ϕ, we might expect a result identical to that of the system $Ax = b$ because mathematically they are equivalent. However, issues arise during the selection of regularization parameters that will work consistently across different optimization problems. If A is close to singular, a common practice is to use $A+\delta I$ for a factorization, be it complete to solve the system directly, or incomplete to use as a preconditioner. If there has been no control over the magnitude of the entries of A, this regularization may fail to solve the factorization issue if it is too small, or completely dominate if it is too large, resulting in factors that are close to $\sqrt{\delta}I$ with minor correction terms. In Chapter 3 we first scale our systems to enable selection of parameters that can be reused across many different systems.

Convergence of iterative methods

Even elementary scaling before using an iterative method can vastly change convergence. For example on the diagonal $n \times n$ system $Ax = b$ with $A_{ii} = i$ and $A_{ij} = 0$ if $i \neq j$, CG will take n iterations to converge. However, if A is scaled symmetrically to have unit diagonal, convergence occurs in one iteration. This is a pathological example, and one would not use an iterative method in this case, but scaling decreases $\kappa(A)$ and gives better convergence bounds according to Eq. (1.3).

1.3.1 Max scaling

The goal of max scaling is to create $\tilde{A}$ such that $|\tilde{A}_{ij}| \leq 1$. To this effect we use a positive diagonal matrix D. In row-wise max scaling, this is achieved by setting $\tilde{A} = D_r A$, where $(D_r)_{ii} = (\max_{1 \leq j \leq n} |A_{ij}|)^{-1}$. In column-wise max scaling, this is achieved by setting $\tilde{A} = AD_c$, where $(D_c)_{jj} = (\max_{1 \leq i \leq n} |A_{ij}|)^{-1}$. In row-column-wise max scaling, this is achieved by setting $\tilde{A} = \sqrt{D_r} A \sqrt{D_c}$, where $(\sqrt{D})_{ii} \coloneqq \sqrt{D_{ii}}$. An advantage of max scaling is that it preserves symmetry in $\tilde{A}$, because if A is symmetric (or Hermitian) then $D_r = D_c$.

1.3.2 Diagonal scaling

Diagonal scaling is typically applied to HPD A, with $\tilde{A} = DAD$ and $D_{ii} = 1/\sqrt{A_{ii}}$, which means $\tilde{A}_{ii} = 1 \quad \forall i$. (For HPD matrices, $A_{ii} > 0 \ \forall i$, so the scaling is guaranteed to be defined.)

Proof. If $A_{ii} \leq 0$ for some i, then $e_i^T A e_i = A_{ii} \leq 0$, which contradicts A being HPD. □

Note that for HPD matrices the diagonal entries are frequently the largest in magnitude in their respective rows and columns, so max scaling and diagonal scaling are equivalent, but this is not always true. For example, the matrix $\begin{bmatrix} 1 & 2 \\ 2 & 5 \end{bmatrix}$ is HPD but the maximum entry in the first row and column is not the $(1, 1)$ entry. Interestingly, however, an off-diagonal entry cannot have maximal magnitude in both its row and its column.

Proof. Take an arbitrary Hermitian A and assume $\exists i, j$ such that $|A_{ij}| \geq A_{ii} \wedge |A_{ij}| \geq A_{jj}$. We know that the HPD structure of A is unaffected by symmetry-preserving permutations and that A is positive definite if and only if all its principal submatrices are positive definite [48]. If we permute the i-th row and column with the first, and the j-th row and column with the second, the determinant of the second principal submatrix is $A_{ii}A_{jj} - A_{ij}A_{ij}^* \leq 0$. This means A cannot be positive definite. □

1.3.3 Ruiz scaling

Ruiz scaling [146] is an iterative application of max scaling until some stopping criterion is met. Some examples of stopping criteria are that the entries of D_r and D_c are close to 1 or a maximum

number of iterations. The advantage of Ruiz scaling is that it assures that if $A_{ij} \neq 0$, then $|\tilde{A}_{ij}| \approx 1$. In contrast, max scaling guarantees only $|\tilde{A}_{ij}| \leq 1$. This is advantageous because it means the nonzero entries of $\tilde{A}$ are of the same magnitude, so floating-point operations between them are safer. The disadvantage is that Ruiz scaling is necessarily more computationally intensive than max scaling (however this is not usually the bottleneck of the computation). There is also a symmetric version of Ruiz scaling [103], which is an iterative application of symmetric max scaling.

1.3.4 Norm scaling

The goal of norm scaling is to create $\tilde{A}$ whose rows (or columns) have norm 1. This is achieved similarly to max-scaling except $(D_r)_{ii} = (\|A_{i,:}\|_2)^{-1}$ (the norm of the i-th row of A) and $(D_c)_{ii} = (\|A_{:,i}\|)^{-1}$ (the norm of the i-th column of A). Frequently the 2-norm is used. When the infinity-norm is used, this is equivalent to max scaling. The variant that scales both rows and columns can also be used with norm scaling.

1.4 Reordering

The ordering of the unknowns x in the equation $Ax = b$ is arbitrary in the sense that the solution to the system does not depend on it. Therefore we reorder or permute x if it is advantageous. If we define a permutation matrix P, some permuted version of the identity with $P^TP = I$, we can write $Ax = b$ as $PAx = Pb$ and factorize PA rather than A to reach a solution. If we define $y = Px$, we can also write it as $AP^Ty = b$, solve for y by factorizing AP^T and multiply by P^T to reach the solution x. If A is symmetric (or Hermitian) and we wish to preserve symmetry, we may write $PAP^Ty = Pb$, solve for y by factorizing PAP^T and multiply by P^T to reach the solution x. This is done because, during factorization, the permuted A may have less fill-in or be more numerically stable. For Hermitian matrices, the fill-in reduction analysis is carried out on A, whereas for non-Hermitian matrices it's carried out on $A + A^H$.

1.4.1 Fill-in reduction

Barring numerical cancellation, the rules for the creation of fill-in in L (the lower triangular factor of A) can be summarized as follows [48]:

1. $i > j$, $a_{ij} \neq 0 \implies l_{ij} \neq 0$
2. $i > j > k$, $l_{jk} \neq 0$ & $l_{ik} \neq 0 \implies l_{ij} \neq 0$

Fig. 1.1 shows a stark example of two matrices that are symbolically equivalent, but the one on the left is completely dense when factorized, and the one on the right has no fill-in. By simply moving the first variable to be last (permuting the first and n-th rows and columns), we can save

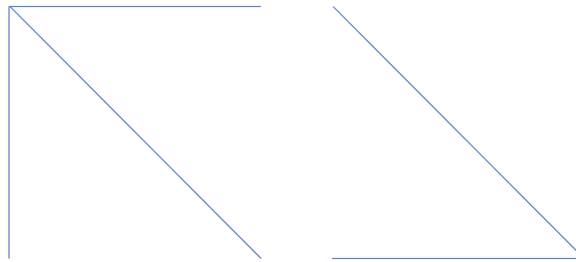


Figure 1.1: (Left) A matrix that is completely dense when factorized. (Right) A matrix with no fill-in when factorized.

substantial storage and computation. In general, the objective of finding an ordering that produces the least possible amount of fill-in poses an NP-complete problem [172]. Instead, we use heuristics that usually work well in practice. Some examples are detailed in the following sections.

Approximate minimum degree

The minimum degree algorithm follows these steps:

1. Define the graph of a square matrix A as having n (the dimension of A) nodes with an edge between node i and node j existing in the graph if $A_{ij} \neq 0$.
2. A symbolic factorization is simulated on a graph and at each factorization step, row and columns are permuted to minimize the number of off-diagonal nonzeros in the pivot row and column.

The symmetric version of the algorithm, which permutes a given row if and only if it permutes the corresponding column, is called symmetric minimum degree [143]. An important implementation detail is a tie-breaking strategy when the multiple renumberings result in the same degree. In practice, an approximate minimum degree (**AMD**) algorithm is used because of the exact minimum degree algorithm's computational cost [160].

Nested dissection

The nested dissection algorithm [76] on a matrix A consists of these steps:

1. Form an undirected graph in which nodes represent rows and columns of A, and an edge between two nodes i, j represents a nonzero entry A_{ij}.
2. Partition the graph recursively into subgraphs using separators, which are small subsets of nodes such that when they are removed, the graph can be partitioned into subgraphs with at most a constant fraction of the number of nodes.

3. An ordering of the unknowns is computed using the recursive structure of the partition: Eliminate the two subgraphs formed by removing their corresponding separator node, then eliminate the separator nodes.

Using nested dissection limits fill-in to the square of the number of separator nodes at each level of the recursive partition. For planar graphs, which can arise from the solution of sparse linear systems derived from 2D finite element method meshes, the resulting matrix has $O(n \log n)$ nonzeros and is computed in $O(n^{3/2})$ time. For 3D space graphs, which can arise from the solution of sparse linear systems derived from 3D finite element method meshes, the resulting matrix has $O(n \log^4 n)$ nonzeros (though the constant can be quite large) and is computed in $O(n^2)$ time. For arbitrary graphs there is a nested dissection that guarantees fill-in within a $O(\min\{\sqrt{d} \log^4 n, \text{nnz}^{1/4} \log^{3.5} n\})$ factor of optimal, where d is the maximum degree and nnz is the number of nonzeros [10].

Reverse Cuthill-McKee

The Cuthill–McKee algorithm (CM) [47] produces a permutation of a symmetrically structured sparse matrix into a small-bandwidth band matrix. A level structure is computed by breadth-first search. Then the nodes are numbered according to this structure, where in each level they are visited in order of their predecessor's numbering from lowest to highest. Ties are broken by node degree (from lowest to highest). The reverse CM algorithm (RCM) reverses the resulting index numbers. In practice RCM generally produces less fill-in than CM [75], so it is more often used.

1.4.2 Numerical stability via pivoting

The LU, LBL^{T}, and Cholesky factorizations all require dividing by entries on the diagonal (see Section 1.1.1). This can create numerical issues if there are very small magnitude entries on the diagonal that arise during the factorization, and will cause breakdowns if there are 0s on the diagonal. For this reason, during the factorization of A, pivoting, i.e., switching entries of A (and making the corresponding switches to the system) is essential. It should be noted that all three factorizations are stable (do not require pivoting) when performed on HPD matrices. In particular, Cholesky is stable because it is only performed on HPD matrices [48]. In this section we detail some common methods for pivoting. The first four pivoting methods break symmetry and can only be used with LU factorization. The last two methods (diagonal pivoting and MC64) maintain symmetry.

Row pivoting

During row pivoting, before computing the j-th diagonal entry of U, we find the largest magnitude entry in the j-th column of the partially factorized A (with row index at least j). We then switch its row with the j-th row and proceed.

Column pivoting

During column pivoting, before computing the j-th diagonal entry of U, we find the largest magnitude entry in the j-th row of the partially factorized A (with column index at least j). We then switch its column with the j-th column and proceed.

Full (or complete) pivoting

During full pivoting, before computing the j-th diagonal entry of U, we find the largest magnitude entry in the partially factorized A (with row and column indices at least j). We then switch its row and column with the j-th row and column and proceed.

Rook pivoting

Rook pivoting is a compromise between row or column pivoting (which may not find a large enough pivot) and full pivoting (which requires searching over the entire matrix and is computationally intensive). Before computing the j-th diagonal entry of U, we look for the largest entry in the j-th row of the partially factorized A (with column index at least j), then look for the largest entry in that entry's column of the partially factorized A (with row index at least j), and repeat alternating between columns and rows until we find an entry that is maximal in both its row and its column. We then switch its row and column with the j-th row and column and proceed.

Symmetric or diagonal pivoting

During symmetric pivoting, before computing the j-th diagonal entry of U, we find the largest magnitude entry on the diagonal of the partially factorized A (for all indices at least j). We then switch its row and column with the j-th row and column and proceed. Since the diagonal entries always stay on the diagonal, this method preserves symmetry and is suitable for use with $\mathrm{LBL}^{\mathrm{T}}$ and Cholesky (though recall that pivoting is unnecessary for Cholesky except to preserve sparsity).

MC64

MC64 [2] is a widely used algorithm using minimum weight perfect matchings to avoid small pivots. It computes a column permutation P that maximizes the product of the absolute values of the diagonal entries as well as row- and column-scaling coefficients D_l and D_r. This creates an I-matrix $A_I \equiv D_l A D_r P$, i.e. $|(A_I)|_{ii} = 1$ and $|(A_I)_{ij}| \leq 1\ \forall i, j$. While the reordering is unsymmetric and therefore not suitable for symmetric (or Hermitian) systems of equations, the scaling step remains desirable as it reduces the condition number. Therefore MC64 provides the possibility to generate a symmetric reordering and scaling. The scaling coefficients have the same properties as in the unsymmetric case but the reordering will not generally maximize the product of the diagonal entries.

1.5 Preconditioning for iterative solutions of sparse HPD systems

Sparse HPD linear systems $Ax = b$ arise in many science and engineering fields including computational fluid dynamics, power networks, economics, material analysis, structural analysis, statistics, circuit simulation, computer vision, model reduction, electromagnetics, acoustics, combinatorics, undirected graphs, and heat or mass transfer. A reliable and efficient solution method is therefore crucial. A direct solution via Cholesky factorization, detailed in Section 1.1, is efficient when the sparsity structure is such that fill-in is moderate.

Iterative solvers typically require preconditioners to limit the number of iterations for convergence. One form of preconditioning requires a sparse matrix $M \approx A^{-1}$ such that $AM \approx I$ or $MA \approx I$ [19, 68]. Such an M is a sparse approximate inverse (**SPAI**) of A, which defines a class of methods known as SPAI-type [86]. The transformed systems $AMy = b$, $x = My$ or $MAx = Mb$ have the same solution as $Ax = b$, but ideally need fewer iterations of an iterative solver. If we have $M = C^H C$ for some nonsingular C, then M is HPD and we can precondition $Ax = b$ by solving the left-right preconditioned system $CAC^H y = Cb,\ x = C^H y$. The matrices C and C^H are not needed by PCG, just products Mv for given vectors v. Alternatively, we may have $M \approx A$ (not A^{-1}). Such an M is applied by solving systems with it, via factorization and then triangular solves. This leads to a class of methods known as approximate factorizations for preconditioning.

Broadly speaking, preconditioners can be split into two different types.

Problem-specific preconditioners require knowledge of the domain in which the linear system arises and knowledge of the specific properties of the matrix A. An example of such a preconditioner for the coupled Stokes-Darcy problem with the Darcy problem in primal form is given in [96]. These preconditioners can be highly effective in solving a given problem because they incorporate more knowledge of the problem than a generic preconditioner, but may require substantial development time for any new problem. Their use is therefore limited to situations in which many problems of the same type need to be solved and the developer has sufficient understanding of both the domain and the numerical liner algebra.

Generic preconditioners require no domain knowledge and have typically few (if any) limitations, but may not be the most effective for any given problem. For the rest of this section and in Chapter 2, we limit our discussion to this type of preconditioner. A good generic preconditioner should substantially reduce the number of iterations until convergence, while working on as large of a class of problems as possible. However, there are multiple other properties that can make a preconditioner more effective.

1. A preconditioner must be sparse (or easy to apply) to be effective, but a flexible sparsity pattern can make it even more effective. Whether the preconditioner is trying to approximate the inverse of the matrix, or approximate its factors, it's possible that the important (large)

entries of the preconditioner cannot be ascertained ahead of time based on generic information. A flexible sparsity pattern allows these entries to be "captured" in the preconditioner without a priori knowledge of the location of these entries.

2. One of the big advantages of using iterative methods is that they are inherently parallel because they only require matrix-vector multiplications. However, if the computation of the preconditioner is serial (or parallel in some less efficient way), it becomes the bottle-neck in a parallel program, no matter how efficient the implementation is. We therefore prefer the computation of the preconditioner to be parallel.

3. Similarly, we would like the application of the preconditioner to be embarrassingly parallel. This means that ideally the preconditioner will be applied by matrix-vector multiplication, rather than the solution of systems (which allows for only limited parallelism).

4. The solution of HPD systems via CG or MINRES means that the storage required by the algorithm and the cost per iteration of the algorithm are fixed (and do not depend on the iteration number). If the computed preconditioner is not HPD, CG and MINRES cannot be used, meaning that GMRES must be used instead. This is undesirable as the storage of GMRES grows linearly with the iteration number, and the computation grows quadratically. We note that a different version of MINRES [15] can be used to solve HPD problems with indefinite preconditioners, but since we know that the best preconditioner (the actual inverse) is positive definite, this is not necessarily desirable. Thus, in almost all cases, we will want our preconditioner to be HPD when solving HPD systems.

5. An ideal preconditioner should work for all problems in its target class and not break down in certain instances.

Here we summarize some of the existing general-purpose preconditioning methods and explain which properties they have and which they lack.

1. Sparsified Nested Dissection (spaND) [32] is a method that computes an approximate factorization and uses it as a preconditioner. The spaND algorithm computes a nested dissection ordering, sparsifies the resulting factors by zeroing out certain entries (that would cause a lot of fill-in were they not ignored), and then uses these factors as a preconditioner for PCG. Though the spaND algorithm is highly effective, its downsides are that it requires the interfaces between groups of variables to be relatively sparse (i.e., there are only short-range interactions between variables) and that it is has only some parallelism in its computation and is applied by solving systems (rather than via matrix-vector multiplication).

2. Methods based on A-orthogonalization such as [22], which compute the factors of the orthogonalized A, are effective even on highly ill-conditioned systems and do not have a risk of

breakdown. However, their sparsity structure is fixed and they do not exhibit parallelism in the computation of the preconditioner or its application.

3. Factorized sparse approximate inverse (or FSAI) methods [104, 105, 110] calculate C such that $\widetilde{M} = C^H C$ directly. Their computation can use shared memory parallelism and they are applied via products with C and C^H. However, they have a risk of breakdown and their sparsity pattern is fixed.

4. Incomplete Cholesky factorization (ichol) [116, 117] is a popular method for computing a sparse triangular matrix L such that $LL^H \approx A$ (and $C = L^{-1}$). In this case, PCG requires triangular solves with L and L^H each iteration. A typical choice for the sparsity structure of L is that of the lower triangular part of A. The sparsity pattern can be made more flexible by setting a drop tolerance and leaving only entries with magnitudes larger than this tolerance. However, this means that the sparsity of the preconditioner (and therefore allocation) cannot be done ahead of time. ichol can be applied to any HPD system, but it is at the risk of breakdowns and it is applied as a triangular solve. There is a version of ichol that is computed with fine grained parallelism [42], which still requires communication between iterations.

5. Randomized Cholesky factorization (**rchol**) works only on symmetric diagonally dominant (**SDD**) matrices (meaning $\forall i,\ A_{ii} \geq \sum_{i \neq j} |A_{ij}|$) and may require doubling the size of the system if the matrix is not SDDM (an SDD matrix with non-positive off-diagonal entries: $A_{ij} \leq 0$ if $i \neq j$). While rchol is restricted to a smaller class of matrices, it has been shown to work better than ichol in some cases [36]. Similarly to ichol, it computes an incomplete factorization of A and has the same properties in terms of sparsity and parallelism. Theoretically it cannot break down, but Section 2.5.2 shows a practical example where it does.

6. Sparse approximate inverse (SPAI) methods [86] are highly flexible and can be used for any type of nonsingular A. They work by computing $M \approx A^{-1}$ such that $AM \approx I$ or $MA \approx I$ and applying M in preconditioned GMRES. This is done even in cases where A is HPD because the resulting M is typically not Hermitian and sometimes not positive definite (unless prohibitively many nonzero entries are allowed). Symmetry can be dealt with easily by taking $\widetilde{M} = (M + M^H)/2$, but getting a positive definite preconditioner is not so easy. This is their main disadvantage, as they have all the other desirable properties.

7. SAINV [21] is a stabilized version of SPAI (also known as AINV) for HPD matrices. SAINV works similarly to SPAI except it adds a matrix to A during computation of M to ensure that M is positive definite. This means that A may be modified too much or even unnecessarily unless the process is attempted multiple times. Its advantage over SPAI is that it allows the use of PCG or MINRES, which saves substantial computation time and memory.

8. Algebraic multigrid (AMG) methods are useful for a wide range of problems resulting from discretization of partial differential equations [27]. They attempt to solve the linear systems resulting from the finely discretized grid by using the solution of the coarsely discretized grid. AMG methods can also work well on matrices that have a specific structure such as Laplacian matrices [106]. AMG is more effective when used on matrices with a specific structure, and thus fulfil our requirement of a broad preconditioner. AMG can be used in conjunction with other preconditioners by computing them on the coarser systems and then using AMG to extrapolate them onto the finer systems. Thus AMG methods can inherit the properties of the other preconditioners.

In Chapter 2 we propose SSAI, a method most closely related to SPAI, which tries to provide the advantages of SAINV without the disadvantage of modifying A a priori and possibly incurring an unnecessary loss of accuracy, or waiting before modifying it and possibly getting a breakdown. SSAI does not work on modifying A in the computation of M, but rather first computes M and then modifies it if it sees it is not positive definite within the PCG (or MINRES) solve. Its application is extended to least squares problems in Section 2.6.

1.6 Direct methods for solving sparse symmetric indefinite KKT linear systems

This section contains partial text of our work in Swirydowicz et al. [157] originally published in *Parallel Computing* (2020). Reprinted with permission. All rights reserved.

Interior-point methods [168, 169] provide an important tool for nonlinear optimization and are used widely for engineering design [23] and discovery from experimental data [57]. Most of the computational cost for interior methods comes from solving underlying linear equations. Indeed, interior methods came to prominence with the emergence of robust sparse linear solving techniques. For a general nonconvex optimization problem, the linear systems are sparse symmetric indefinite and typically ill-conditioned. Furthermore, many implementations require the linear solver to provide matrix inertia information (the number of positive, zero, and negative eigenvalues).

There is a vast body of research on benchmarking optimization solvers for their effectiveness and performance [54, 55]. However, not much attention has been devoted to the linear solvers employed within interior methods for optimization. The literature can be divided into three major categories: (a) papers that compare different optimization solvers without isolating the linear solver performance, including parameters and algorithm selection, (b) literature focusing on linear solver selection and performance, and (c) papers focusing on linear solvers for symmetric indefinite systems.

In papers from category (a), the CUTE, CUTEr, or CUTEst [82] frameworks are often employed. For example, Schenk et al. [149] applies a combinatorial (symmetric weighted matching based) preprocessing step to lower the condition number of the matrices arising in Ipopt [170], followed by a

direct solver that uses supernodal static pivoting. The authors test this approach with CUTEr.

For category (b), authors are primarily concerned with finding suitable preconditioners [33, 121, 132] for solving the linear systems iteratively. We cannot take advantage of these results, however, because the problems described herein are extremely ill-conditioned and thus require a direct linear solver, possibly with IR, to achieve acceptable backward error levels.

For category (c), there is a long lineage of research available because the scientific community has been interested in efficient numerical linear solvers for (dense and sparse) symmetric indefinite systems dating back to the 1970s [28, 69]. Bunch and Kaufman [30] introduced the pivoting strategy currently used in LAPACK for dense symmetric indefinite systems. This scheme was later improved in [14], where the more stable *bounded Bunch-Kaufman pivoting* is proposed. Duff et al. pioneered much of the field with a series of papers focused on symmetric indefinite systems [58, 62, 63, 65, 66]. Gould et al. [83] reviewed the sparse symmetric solvers that are part of the mathematical software library HSL (formerly the Harwell Subroutine Library) and concluded that MA57 [58] is currently the best general-purpose package in HSL. Hogg et al. [94] improved the pivoting strategy of [62] for challenging matrices. Duff et al. [64] focused on pivoting strategies and new approaches designed for parallel computers. Similarly, several recent papers [59, 93, 95] have considered improved algorithms targeting modern multicore processors and GPUs. Becker et al. [17] proposed a supernode-Bunch-Kaufman pivoting strategy that enables high-performance implementations on modern multicore processors. For a survey of the literature on sparse symmetric systems, see Davis et al. [52]. For a benchmark of several sparse symmetric direct solvers, see Gould et al. [84].

Our goal in Chapter 3 is to develop a GPU-accelerated linear solver for these types of highly ill-conditioned linear systems arising from interior point methods. Here we summarize the state-of-the-art software packages that meet these criteria, along with MA57, which is considered the one of the best performing solvers for symmetric indefinite systems, even though it is currently not GPU-accelerated.

SuperLU

SuperLU is the most mature and established of the five packages evaluated. For the purpose of generating results presented in this study we used SuperLU_dist, which is accelerated by MPI and CUDA.

SuperLU is a direct linear solver for large, sparse unsymmetric systems. It employs Gaussian elimination (super-nodal) to compute the factorization

$$P_r D_r A D_c P_c = LU,$$

where D_r and D_c are diagonal matrices used to equilibrate (scale) the matrix A, and P_c, P_r are column/row permutation matrices. If the solution is not accurate enough, it follows with an IR

phase based on the Richardson iteration.

Unlike the serial version of SuperLU (which uses partial pivoting), the distributed version uses static pivoting based on graph matching [112], which has better scaling properties [142]. The pivoting strategy determines the choice of P_r. Distributed SuperLU employs an $(A^T + A)$-based sparsity ordering, which determines P_c. SuperLU reorders both columns and rows; however, the user does not have the freedom to select a particular algorithm.

STRUMPACK

STRUMPACK (STRUctured Matrix PACKage was developed by Pieter Ghysels, Xiaoye Li, Yang Liu, Lisa Claus and other contributors at Lawrence Berkeley National Laboratory. STRUMPACK is intended to solve sparse and dense systems. STRUMPACK is targeted toward matrices that exhibit an underlying low-rank structure (such as block structure) that is exploited to construct the L and U factors. Unlike in SuperLU, the computed factors can be used as a preconditioner within an iterative linear solver. A multifrontal factorization is employed.

STRUMPACK uses MC64 [61] to permute and scale matrix columns (as does SuperLU). It follows the MC64 step with a nested dissection reordering [76], which controls the amount of fill-in. The third pre-processing step is a reordering that helps reduce the ranks used for the HSS (hierarchically semi-separable) compression steps. After the pre-processing, there is a symbolic factorization phase to construct the elimination tree, and then actual multifrontal factorization. During the factorization, HSS is used to numerically compress the fronts. This strategy is typically only applied to large fronts (near the end of the factorization). Depending on the settings and the matrix, the solve can be done through back substitution combined with either IR or an iterative solver (GMRES or BiCGStab).

STRUMPACK is parallelized through MPI and CUDA. It uses cuBLAS and cuSolver, and relies on SLATE for LAPACK functions.

cuSolver

`cuSolver` is a library provided by NVIDIA and distributed with CUDA Toolkit The functions provided by the library allow the user to solve a system (or multiple systems) of linear equations $Ax = b$, where A does not have to be square. It computes a QR, LU, or LBL$^{\mathrm{T}}$ factorization of A, then performs upper and lower triangular solves. The library consists of three main components: `cuSolverDN` (for dense matrices), `cuSolverSP` (for sparse matrices) and `cuSolverRF` (for matrix series in which the matrix values change but the nonzero pattern stays the same). The library comes with many helper functions and provides several algorithms for solving linear systems (using LU, QR, Cholesky, or least squares).

We focus on `cuSolverSP` because our matrices are sparse. However, the sparse interface appears to be less mature and to have limited functionality. For example, some of the functions are not implemented to run on GPUs, and LBL$^{\mathrm{T}}$ is available only through the dense interface.

cuSolver is proprietary software and its source code is not available. The user supplies the matrix and right-hand side and the functions return a solution along with the error. The only parameter that can be chosen is the reordering with options: RCM (reverse Cuthill–McKee), AMD (approximate minimum degree), MeTiS, or none.

SSIDS

SSIDS is part of a large library known as SPRAL (Sparse Parallel Robust Algorithms Library). SPRAL was developed by the Computational Mathematics Group at Rutherford Appleton Laboratory in the UK.

Unlike SuperLU and STRUMPACK, SSIDS uses an LBL^{T} factorization to solve $Ax = b$ with A symmetric but indefinite. In a first pass, SSIDS computes the decomposition $A = PLD\,(PL)^T$, where P is a permutation matrix, L is unit lower triangular, and D is either diagonal or block diagonal with 1×1 or 2×2 blocks. A clear advantage of the LBL^{T} factorization is that it requires less storage, as only one triangular factor needs to be stored. SSIDS applies an *a posteriori* threshold pivoting [59] to implement a scalable GPU pivoting strategy.

PaStiX

PaStiX (Parallel Sparse matriX package) [91], a scientific library that provides a direct parallel linear solver for large sparse linear systems, was developed at Bordeaux University and Inria.

PaStiX uses numerical algorithms implemented in single or double precision (real or complex) using Cholesky, LBL^{T}, and LU with static pivoting (for nonsymmetric matrices having a symmetric pattern). It later refines the solutions computed by the direct method using GMRES preconditioned with the computed factors. Additionally, PaStiX provides low-rank compression methods to reduce the memory footprint and save time. For our purpose, the LBL^{T} factorization is used and it has a similar structure to the one used by SSIDS.

MA57

The MA57 package is part of HSL [3]. It was written and developed by the Computational Mathematics Group at the Science and Technology Facilities Council Rutherford Appleton Laboratory and other experts.

Similarly to SSIDS, MA57 uses the multifrontal LBL^{T} method developed in [65] for symmetric indefinite systems. The matrix can be optionally symmetrically prescaled using the MC64 subroutine and ordered using MeTiS. The user can avoid fill-in beyond that predicted by the analysis by using static pivoting, though this can cause numerical issues and breakdowns. It is currently not GPU-capable and any GPU capabilities would be substantially inhibited by pivoting.

1.6.1 Motivation

We show in Swirydowicz et al. [157] that the GPU-accelerated solvers do not work well on our highly ill-conditioned problems. All of them, except cuSolver, perform worse when a GPU is used and also perform worse than MA57 with or without a GPU. cuSolver is able to achieve some speedup on a GPU and outperforms MA57 by a factor of ≈ 1.6 when a GPU is used [38]. cuSolver is proprietary, so our goal in Chapter 3 is to build an open source code that is able to utilize GPUs better by avoiding pivoting and unnecessary operations. To this end it is crucial to avoid unstable factorizations (and stick with Cholesky) and to view the systems arising from an interior-point method applied to a given problem as a whole, rather than as separate systems of the form $Ax = b$. This means that the operations that are shared by these systems because they share a sparsity structure will only be performed once. Furthermore, our method should provide a certificate of the correctness of the inertia.

1.7 Parallel computing

In serial programming, an algorithm is constructed and implemented as a serial list of instructions. The instructions are executed on a central processing unit (**CPU**) of a single computer. Only when an instruction is finished can the next instruction be executed.

Conversely, parallel computing simultaneously employs multiple processing elements to execute a program. The algorithm is broken down into parts that can each be executed simultaneously on different processing elements. The processing elements can include resources such as one computer with several processors, multiple networked computers, hardware accelerators such as graphical processing units (**GPU**s), or a combination of them. Parallel computing is used for scientific computing applications where the problem sizes grow very large, such as fluid or structural mechanics simulations, weather simulations, or any program that requires the solving of linear systems or matrix products, which is true for almost any scientific computing application [131].

Until 2004, improvements in the frequency of instructions executed by a computer, i.e., the number of instructions executed in a given unit of time, were the main source of improvements in run time [90]. However, as the power consumption of a processor scales linearly with its frequency and hardware constraints limit the frequency of any given processor, parallel computing is necessary to solve large problems [130].

Parallel computing first became available to the masses in 1994 with the publication of message passing interface (**MPI**) 1.0 [46], but it has taken some time to become fully adopted. GPU programming is a specific type of parallel programming that uses GPUs. Initially GPUs were designed for use in computer graphics, but they have since been adopted for general-purpose (and specifically scientific) computing. This is because GPUs can perform simple operations such as multiplications or additions at a higher frequency than CPUs. The downside to GPU use is that memory allocation

and management are typically costly and that the cache size is small. This means that more care must be taken in the design of algorithms that utilize GPUs, otherwise they may perform worse than even on a single CPU.

This thesis does not employ MPI programming. Chapter 2 details an algorithm that is shown to be embarrassingly parallel (see Section 1.7.3) but is implemented in serial fashion to limit scope. Chapter 3 suggests an algorithm that can utilize hardware accelerators such as GPUs in parallel and this implementation is provided.

AMD (Advanced Micro Devices) [160] and Intel [4] have two common frameworks for GPU coding. In this thesis we use the CUDA framework by NVIDIA, which is more mature than the two previously mentioned frameworks, including the cuBLAS [5], cuSPARSE [6], and cuSolver [1] libraries because it is more mature and has the linear algebra operations required by our algorithms. We have wrapper functions around this proprietary code so that if the kernels are developed in an open source library, they can easily be replaced. We also write some custom GPU kernels (functions) when needed.

1.7.1 Bounds on speedup

Ideally the speedup using multiple processors would be linear, i.e., doubling the number of processors and keeping the problem size constant (known as strong scaling) would halve the run time, and doubling the number of processors and the problem size (known as weak scaling) would leave the run time constant. In practice, parallel programs can at best achieve nearly linear strong scaling up to a certain number of processors and then level off. However, the bound for weak scaling is more optimistic and can stay nearly linear even for a large number of processors. We define R_t to be the relative run time of the parallel program (with the serial equivalent fixed to 1), $0 \leq s \leq 1$ is the fraction of time spent on the serial part of the program, and p the number of processors. Using these definitions, Amdahl's law and Gustafson's law provide simplified models for understanding strong and weak scaling respectively.

Amdahl's law

In 1967 Gene Amdahl showed that the run time of a program is lower bounded by the part of it that must be executed serially [11]. The bound is given as

$$R_t = s + \frac{1-s}{p}. \tag{1.5}$$

Eq. (1.5) provides a grim view of parallelism. First we notice that even if $p \to \infty$, the run time cannot fall below s. If we want to achieve some run time $\tilde{R}_t > s$, we require $p = \left\lceil (1-s)/(\tilde{R}_t - s) \right\rceil$ processors. This may be very limiting unless $s \ll 1$. The solution to this issue is to not blindly increase computing resources for small problems and instead save them for large ones.

Gustafson's law

Gustafson's law assumes that the serial part of a program does not increase along with the problem size and its parallel part scales linearly with the number of processors [87] and gives the bound for weak scaling in Eq. (1.6). This assumption holds for an embarrassingly parallel program (see Section 1.7.3), but does not hold in general because of communication between processors and increased costs to the serial part due to setting them up and managing memory. However, usually a well designed program can keep communication down and have the serial part scale slowly with the problem size (such as $O(\log n)$) and the parallel part scale nearly linear with the problem size (such as $O(n \log n)$). The equation

$$R_t = \frac{1}{s + (1-s)p} \tag{1.6}$$

shows that in the regime $p \gg s/(1-s)$, with double the processors, problems of double the size can be solved in close to the same amount of time. In practice, Gustafson's assumptions hold for few problems so the weak scaling is not ideal, but his law is still instructive with regard to the proper way to use parallel computing resources.

1.7.2 Performance limiters

There are three types of limits on performance of a parallel code:

Compute bound A code is compute bound if its computations finish in a given time frame or at the expected maximum based on the architecture.

Memory bound A code is memory bound if its memory bus is frequently saturated.

Latency bound A code is latency bound if neither of the above is true. In this case the run time is dominated by the communication time between the processors.

Typically the number of computations can be easily approximated by analyzing an algorithm, so they can be minimized at the design stage. GPUs are highly efficient at arithmetic operations and thus there are few efficient algorithms that would give rise to a compute bound GPU code. Managing memory efficiently and not allocating it more frequently than necessary is essential for code design. An example of this is given in Chapter 3. Minimizing latency requires minimizing communication between processors and frequently dictates the number of processors that should be used. An example of this is Algorithm 4, where columns of a preconditioner are computed independently of each other and communication occurs only at the beginning to disperse the information and at the end to aggregate it.

1.7.3 Types of parallelism

Algorithms that exhibit some level or parallelism can be classified according to how often their subtasks synchronize or communicate with each other. An algorithm exhibits fine-grained parallelism if its subtasks communicate many times in comparison to the amount of computation; it exhibits coarse-grained parallelism if they communicate a few times in comparison to the amount of computation, and it exhibits embarrassing parallelism if they rarely or never have to communicate. Embarrassingly parallel algorithms are considered the easiest to parallelize in practice. A classic example of a parallel algorithm is a matrix-vector multiplication. Each entry in the product Ab is computed using only one row of A (and all of b) and is independent of the other entries. Thus, roughly speaking, if A has n rows, it can use n processors to cut down the run time by a factor of n (minus fixed costs of memory movement before and after each product). Or in general, with $p < n$ processors, the run time can be cut by a factor of $\lceil n/p \rceil$ by sending $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ rows of A to each processor. Additionally, parallelism may require shared memory between the processors. In the example of the matrix-vector products, b can be sent to all processors, or they can all access the same b if that is more efficient on the available hardware. However, only the relevant rows of A need to be sent to each processor (and no advantage is gained by sending more of them or accessing rows on other processors). This analysis holds true if b is small enough to be stored and transferred in memory. If b is too large, it may need to be split between multiple processors. In this case, all processors computing a given entry of Ab must communicate between them to aggregate the result and the process becomes less parallel.

1.8 Overview

The rest of the thesis is organized as follows. Chapter 2 details SSAI, a symmetric sparse approximate inverse preconditioner for the iterative solvers CG, MINRES, CGLS, LSQR, and LSMR. Chapter 3 details HyKKT, A Hybrid Direct-Iterative Method for Solving KKT Linear Systems. Finally, Chapter 4 summarizes the contributions of the thesis, discusses possible future work, and discusses the tradeoff between specialized and generic solvers.

Chapter 2

SSAI: A symmetric sparse approximate inverse preconditioner for the conjugate gradient methods PCG and PCGLS

This chapter contains a modified version of the text in [139]. This work is currently in review. All rights reserved.

2.1 Overview

We propose a method for solving a Hermitian positive definite linear system $Ax = b$, where A is an explicit sparse matrix (real or complex). A sparse approximate right inverse is computed and replaced by its symmetrization M, which is used as a left-right preconditioner in a modified version of the preconditioned conjugate-gradient method (PCG), where M is modified occasionally, if necessary, to make it more positive definite. Before symmetrization, M is formed column by column and can therefore be computed in parallel with no communication except at the beginning and end. PCG requires only matrix-vector multiplications with A and M (not solving a linear system with M), and so too can be carried out in parallel. We compare it with may factorization (the gold standard for PCG) and with a direct Cholesky factorization on sparse matrices from various applications and show it is robust. For least-squares problems, we implement an analogous form of preconditioned Conjugate Gradient Least-Squares (PCGLS) and show it is also robust.

2.2 Contributions

- We improved upon an existing algorithm for computing a sparse approximate inverse to an SPD matrix to use as a preconditioner.
- We developed a novel method to ensure the resulting preconditioner is positive definite and showed that it is robust in practice.
- We proved a theoretical result showing that symmetric indefinite systems cannot be preconditioned with an indefinite preconditioner using fixed storage methods.
- We developed software implementing the algorithm and it is freely available at [134].

2.3 Introduction

We consider linear systems $Ax = b$, where $A \in \mathbb{C}^{n\times n}$ is a Hermitian positive definite (HPD) matrix (real or complex), and $x, b \in \mathbb{C}^n$. We also consider least-squares problems $\min \|Ax - b\|_2$, where $A \in \mathbb{C}^{m\times n}$. In both cases, we assume A is an explicit, sparse matrix.

HPD systems are common in many fields including computational fluid dynamics, power networks, economics, material analysis, structural analysis, statistics, circuit simulation, computer vision, model reduction, electromagnetics, acoustics, combinatorics, undirected graphs, and heat or mass transfer. A reliable and efficient solution method is therefore crucial. Solving directly (for example with sparse Cholesky factorization) may be computationally prohibitive. Iterative methods are then preferred, especially when they use the sparsity structure of the matrix [80, 163]. For HPD systems, the conjugate gradient method (CG) [92] and preconditioned CG (PCG) are such methods.

In general, the rate of convergence of iterative methods for solving $Ax = b$ depends on the condition number of A and the clustering of its eigenvalues. These bounds are more descriptive when A is normal, as in our case [67]. Preconditioning requires a matrix $M \approx A^{-1}$ such that $AM \approx I$ or $MA \approx I$ [19]. The transformed systems $AMy = b$, $x = My$ or $MAx = Mb$ have the same solution as $Ax = b$ but are typically better conditioned. One strategy for approximating A^{-1} is to choose a sparse M to minimize the Frobenius norm of $AM - I$ or $MA - I$ [16].

As noted by Chow and Saad [43] for general square A,

$$\|AM - I\|_F^2 \equiv \sum_{j=1}^{n} \|Am_j - e_j\|_2^2 \tag{2.1}$$

(where m_j and e_j are column j of M and I) and each m_j can be computed separately by solving n least-squares problems $\min \|Am_j - e_j\|_2^2$, ideally in parallel. For HPD A, our algorithm SSAI approximately solves $Am_j = e_j$ by a Jacobi-like method that changes one element of m_j at a time.

We also study algorithm SSAI_LS for approximately solving each least-squares problem; it is the same as SSAI except for one line in the inner loop.

If we have $\widetilde{M} = C^H C$ for some nonsingular C, then $\widetilde{M}$ is HPD and we can precondition $Ax = b$ by solving the left-right preconditioned system

$$CAC^H y = Cb, \; x = C^H y. \tag{2.2}$$

The matrices C and C^H are not needed by PCG, just products $\widetilde{M}v$ for given vectors v.

Incomplete Cholesky factorization [116, 117] is a popular method for computing a sparse triangular matrix L such that $LL^H \approx A$ (and $C = L^{-1}$ in Eq. (2.2)). In this case, PCG requires triangular solves with L and L^H each iteration. A typical choice for the sparsity structure of L is that of the lower triangular part of A. The cost of each PCG iteration with LL^H preconditioner is then about twice that of CG (assuming serial computation). Choosing a denser L is undesirable unless it substantially decreases the number of iterations. There are some methods such as [32] that successfully use a denser L, which SSAI also allows, but to limit the scope of the chapter, we focus on $\text{nnz}(L) \approx \text{nnz}(A)/2$ (with the exception of Table 2.3). Choosing a sparser L will generally lead to more iterations, while not changing the asymptotic cost of each iteration (decreasing it by at most a factor of 2). Therefore, we restrict our discussion to the ichol factorization [116, 117] with no fill-in (sometimes referred to as IC(0)), its modified version (michol), and a randomized version (rchol) [36]. All three versions omit elements of the exact L. michol compensates the diagonal for the dropped elements by ensuring $Ae = LL^H e$, where e is an n-vector of 1s. rchol works only on symmetric diagonally dominant (SDD) matrices (meaning $\forall i$, $A_{ii} \geq \sum_{i \neq j} |A_{ij}|$) and may require doubling the size of the system if the matrix is not SDDM (an SDD matrix with non-positive off-diagonal entries: $A_{ij} \leq 0$ if $i \neq j$). While rchol is restricted to a smaller class of matrices, it (theoretically) cannot break down, and has been shown to work better than ichol in some cases [36].

Alternatives to incomplete Cholesky factorization include sparse approximate inverse preconditioning (SAINV in Benzi et al. [21] and SPAI in Grote and Huckle [86]) and factorized sparse approximate inverse preconditioning (FSAI in Kolotilina and Yeremin [104] and implemented for example in FSAIPACK [97]). For sparse HPD systems we propose SSAI, a symmetric sparse approximate inverse preconditioner. It is an inherently parallel SPAI algorithm adapted from the left-preconditioner described for GMRES [147] in [148]. Some other SPAI algorithms such as those described in [41, 43] were implemented, but they were not as fast in preliminary tests because they require matrix-vector products at each iteration, as well as a dropping scheme. In contrast, SSAI has scalar-vector products in its iterations, does not compute quantities that are dropped, and does not require the pre-selection of a sparsity pattern or dropping tolerance. However, it can only be used with SPD matrices. To this end, we ensure that the resulting preconditioner is SPD in Algorithm 5 (by changing M to $M + \gamma I$ for some positive γ) so that it can be used with PCG as opposed to only with GMRES. It's worth noting that this adjustment for an SPD preconditioner can be used

with any symmetric preconditioner, including those from [41, 43]. SSAI improves on SPAI by using the symmetry and positive definiteness of A and allowing the solver to be PCG. Its advantages over FSAI are that it does not need prescription of a sparsity pattern, requires communication only twice: at the beginning and end of the algorithm (like SPAI), and like GSPAI-Adaptive [53] is not at risk of breakdowns. We note that one can implement SSAI in a manner that preconditions the system being solved with the approximate M computed until that iteration and still solves for the columns in parallel, but this requires communication at each iteration and under- performs the unpreconditioned version of SPAI-type methods for SPD matrices [43].

The chapter is organized as follows. Section 2.4 details the algorithms for computing our preconditioner $\widetilde{M} \approx A^{-1}$ and solving $Ax = b$ with a modified version of PCG. Section 2.5 compares a MATLAB implementation of $\widetilde{M}$ with ichol, michol, rchol and sparse Cholesky (via MATLAB's '\' operator) on a wide variety of square matrices from different applications from the SuiteSparse Matrix Collection [49, 50] and other sources. We focus on MATLAB comparisons because the main goal of this chapter is to suggest a new algorithm and not focus on differences in implementation. We note that a parallel ichol factorization was developed by Chow and Patel [42], but our focus here is on serial experiments, with some discussions of parallelism. SSAI is computed in an embarrassingly parallel fashion, i.e., completely without communication except once at the beginning and the end, whereas the fine-grained parallel ichol in [42] requires communication between iterations. Section 2.5.2 compares ichol and SSAI with rchol, a method specific to SDDM matrices. For least-squares problems, Section 2.6 develops a modified version of preconditioned Conjugate Gradient Least-Squares (PCGLS), to solve some examples from SuiteSparse. Matrices in Section 2.5, 2.6 are chosen to cover a wide array of fields and problem sizes, and several scale-up tests. Section 2.7 examines sparsity structures of square and rectangular systems and their impact on choosing solution methods. Section 2.8 discusses complexity and parallel implementation. Section 2.9 discusses results and future directions. Section 2.9.1 summarizes the similarities and differences of preconditioners for HPD problems. For symmetric indefinite systems, Section 2.9.3 shows that an indefinite preconditioner $M = C^T C$ can be used if and only if C is available separately. Thus, SSAI is not useful in this situation (because it exists as M, not as $C^T C$).

2.4 A PCG algorithm for HPD linear systems

For SSAI and comparisons with existing methods, we start by diagonally scaling A using Algorithm 2 (unless otherwise noted). Each off-diagonal entry is scaled by the square root of the product of the diagonal entries in its row and column. Thus, $A_{ii} = 1$ and $|A_{ij}| \lesssim 1$. The symmetry and unit diagonal of the scaled A are not affected by round-off. Algorithm 2 takes $O(k)$ time, where $k \equiv \text{nnz}(A)$ is the number of nonzeros in A. It could be parallelized (less trivially than the rest of SSAI) but its cost is negligible. The system to be solved becomes $DADy = Db$, $x = Dy$. From now on we assume

Algorithm 2 Diagonal scaling of A

1: Define a diagonal matrix D with $D_{ii} = 1/\sqrt{A_{ii}}$
2: $A \leftarrow D * \texttt{tril}(A) * D$ (the strictly lower triangular part of A)
3: $A \leftarrow A + A^H + I$

Algorithm 3 Algorithm 3 [148] Parameters (typical values not mentioned): `lfil` , `itmax`

1: $M \leftarrow 0_{n \times n}$
2: **for** $j \leftarrow 1$ `to` n **do**
3: $m_j \leftarrow M * e_j$
4: **for** $k \leftarrow 1$ to `itmax` **do**
5: $r = e_j - Am_j$
6: $i = \texttt{indmax}(|r|)$ (index of maximum element-wise absolute value)
7: $\alpha = r_i / a_{i,i}$
8: $m_j \leftarrow m_j + \alpha e_i$
9: **if** $\text{nnz}(m) \geq$ `lfil` **then**
10: break
11: **end if**
12: **end for**
13: **end for**

$Ax = b$ has been scaled in this way.

The outer loop of Algorithm 4 (SSAI) computes a sparse vector m_j as an approximate solution of $Am_j = e_j$ for each j. The main differences between Algorithm 4 and the method of [148] are application to PCG (not GMRES), simplification of the for loops, parameter selection (`lfil` and `itmax`), and the fact that the symmetrization on line 16 is mandatory instead of optional. The inner loop of Algorithm 3 in [148] computes the residual $r_j = e_j - Am_j$, but Algorithm 4 here updates $r \leftarrow r - \delta * a_i$ very cheaply.

Theorem 1. *The two algorithms are equivalent in exact arithmetic, assuming A is diagonally scaled. (If not, $\delta \leftarrow r[i]/a_{ii}$ in SSAI instead of line 7 makes them equivalent.)*

Proof. Both algorithms initialize r to e_j when working on column j. If we denote the value of a variable z at iteration k by $z^{(k)}$, [148] sets $r^{(k+1)} = e_j - Am_j^{(k+1)} = e_j - A(m_j^{(k)} + m_j^{(k+1)} - m_j^{(k)}) = e_j - A(m_j^{(k)}) - A(e_i \delta^{(k)}) = r^{(k)} - \delta^{(k)} a_i$ as in Algorithm 4 line 12. □

The inner loop of Algorithm 4 changes one entry of m_j by the amount $\delta = r[i]$, the largest element of the residual vector $r = e_j - Am_j$, and limits the number of nonzeros per column to `lfil`. Note that line 12 sets $r[i] = 0$, so the next i will be different.

We must choose `itmax` $\geq$ `lfil` for SSAI to work, but not so large that the preconditioner is too expensive to compute. (In our experiments, we set `itmax` = 2*`lfil`.) We construct M column by column (not row by row) because this leads to efficient memory access in MATLAB.

Algorithm 4 SSAI: left-right HPD preconditioner $\texttt{lfil} = \texttt{ceil}(\text{nnz}(A)/n)$, $\texttt{itmax} = 2*\texttt{lfil}$

```
1: M ← 0_{n×n}
2: for j ← 1 to n do
3:    m ← 0_{n×1}
4:    r ← e_j (n × 1)
5:    for k ← 1 to itmax do
6:       i = indmax(|r|) (index of maximum element-wise absolute value)
7:       δ ← r[i]
8:       m[i] ← m[i] + δ
9:       if nnz(m) ≥ lfil then
10:         break
11:      end if
12:      r ← r − δ*a_i
13:   end for
14:   m_j ← m
15: end for
16: M̃ ← (M + M^H)/2
```

Lines 6–8 of Algorithm 4 generate the entries of m_j with the largest magnitude. M itself has previously been used with GMRES, but PCG requires a Hermitian preconditioner. If M is a good right preconditioner, i.e., it minimizes Eq. (2.1) in some subspace, we know that M^H minimizes $\|M^H A - I\|_F$ in the same subspace, and so is a good left preconditioner. Thus, the Hermitian matrix $\widetilde{M} = (M + M^H)/2$ should be a good left-right preconditioner.

If M is PD, it follows that M^H is PD and $\widetilde{M}$ is HPD. However, M may not be PD. Grote and Huckle [86] prove that if $\|r_j\|_2 \equiv \|Am_j - e_j\|_2 < \epsilon$ and p denotes the maximum number of nonzeros in any column of the residual, the eigenvalues of AM lie inside a disk of radius $\sqrt{p}\epsilon$, centered around 1. As $p \le n$, taking $\epsilon = 1/\sqrt{n}$, changing line 5 of Algorithm 4 to "While $\|r_j\|_2 \ge \epsilon$ do", and removing lines 9–11, guarantees M will be PD [86], but would substantially increase computation time. Fortunately, Algorithm 4 usually produces an HPD $\widetilde{M}$. In Algorithm 5 we use an alternative and less expensive way of ensuring $\widetilde{M}$ is HPD, and this has proved effective in the numerical tests.

If $\widetilde{M} = C^H C$, it is natural to ask whether $CAC^H \approx I$. The eigensystem of $\widetilde{M}$ proves there is a unique HPD $\widetilde{C}$ such that $\widetilde{M} = \widetilde{C}^2$. Further, for a Hermitian A, $A\widetilde{C}^2 = I \Rightarrow \widetilde{C}A\widetilde{C} = I$. By continuity, we can expect that minimizing $\|A\widetilde{M} - I\|_F^2$ tends to minimize $\|\widetilde{C}A\widetilde{C} - I\|_F^2$.

For simplicity, the rest of the chapter uses M in place of $\widetilde{M}$.

2.4.1 Approximate solution of $Am_j = e_j$

Jacobi's method [80] applies to HPD systems. Algorithm 4 applies up to `itmax` iterations of a Jacobi-like method to the jth linear system $Am_j = e_j$ (with A_{ii} assumed to be 1 after scaling). Instead of updating all elements of the solution at each iteration (like Jacobi), Algorithm 4 computes only one element at each iteration and limits the number of iterations to keep m_j sparse. A bound

on the size of the residual r after k steps of the inner loop follows from Theorem 4.1 in [148]: $\|r_k\|_{A^{-1}} \leq (1 - \frac{1}{n\kappa(A)})^{k/2}\|e_j\|_{A^{-1}}$, where $\kappa(A)$ is the condition number of A and $\|x\|_A \equiv x^H Ax$. This provides a weak theoretical bound for convergence, but in practice the algorithm performs much better.

A variation on SSAI is to apply coordinate descent to the function $\frac{1}{2}\|r\|_2^2$ within the inner loop. For each k, this leads to the 1-variable least-squares problem $\min_\delta \|a_i\delta - r\|$, which changes line 7 of Algorithm 4 from $\delta \leftarrow r[i]$ to $\delta \leftarrow a_i^T r/\|a_i\|^2$. We name this method SSAI_LS. The squared norms can be precomputed from A, but the new line of code is significantly more expensive than line 7 of SSAI, and safeguards are needed to choose a different i in the next inner loop (because the largest $r[i]$ may be for the same i, but $a_i^T r$ will now be zero). Salkuyeh and Toutounian [148] prove that each iteration of SSAI reduces the quantity $\|r\|_{A^{-1}}$, whereas SSAI_LS reduces $\|r\|$ itself. Numerical comparisons are therefore of interest. We found in [137] that SSAI_LS tends to produce an M that is closer to being SPD but requires more iterations by the $Ax = b$ solver (in this case preconditioned MINRES).

In [136] we give MATLAB code for both versions of SSAI and for several other preconditioners, along with drivers that use MINRES with each preconditioner to solve a given HPD system $Ax = b$. Two of the preconditioners apply 2 and 3 iterations of MINRES (respectively) to the problem $\min \|Am_j - e_j\|$ for each j. (These preconditioners and their drivers allow A to be definite or indefinite.) On HPD systems we find that SSAI in Algorithm 4 is usually the most effective preconditioner. The numerical results below therefore focus on SSAI, but the other preconditioners in [136] may be useful in specific cases.

2.4.2 Modified PCG

Algorithm 5 makes use of the computed preconditioner $\widetilde{M}$ (now M) in a modified form of PCG. We use a matrix-vector multiplication instead of a linear system solve because $M \approx A^{-1}$ (not A itself). The algorithm is equivalent to applying CG to system Eq. (2.2) with $M = C^H C$. In exact arithmetic, PCG is guaranteed to converge in at most n iterations, but this number is usually much less than n with any reasonable preconditioner. We can safely set the stop condition of the loop to n, knowing that the break conditions will usually stop the loop. We store our initial guess as x_0 and use PCG to solve $Adx = r_0$, ending with $x = x_0 + dx$. If x_0 is a very good guess, the small steps defining dx won't be lost to round-off error.

A further modification is to detect indefiniteness or near-singularity in M (lines 18–20) and restart PCG with an updated x_0 and a more HPD M. The normalized inner product of r with M is $\hat{\rho} \equiv (r^H Mr)/\|r\|_2^2$. If $\hat{\rho}$ is small or negative, M must be close to singular or indefinite. With `tolM` defined to be the minimal $\hat{\rho}$ that we accept, we modify M in a linear fashion to increase $\hat{\rho}$, update x_0, and restart PCG. The parameters `tolM` and δ were chosen experimentally, but the same values were used for all problems.

Algorithm 5 Modified PCG Typical parameters: $\texttt{tolM} = 10^{-2}$, $\delta = 10$

1: $r \leftarrow b - A{*}x_0,\ \ dx \leftarrow 0_{n\times 1}$
2: $p \leftarrow z \leftarrow M{*}r$
3: $\rho_{new} \leftarrow \texttt{real}(z^H{*}r)$
4: **for** $j \leftarrow 1$ `to` `n` **do**
5:     $\texttt{q} \leftarrow A{*}p$
6:     $\beta \leftarrow \texttt{real}(p^H{*}q)$
7:     **if** $\beta \leq 0$ **then**
8:         break; (A is not PD)
9:     **end if**
10:     $\rho \leftarrow \rho_{new},\ \ \alpha \leftarrow \rho/\beta$
11:     $dx \leftarrow dx + \alpha{*}p$
12:     $r \leftarrow r - \alpha{*}q$
13:     **if** $\|r\|_2/\|b\|_2 < \texttt{tol}$ **then**
14:         break
15:     **end if**
16:     $z \leftarrow M{*}r$
17:     $\rho_{new} \leftarrow \texttt{real}(z^H{*}r),\ \ \hat{\rho} = \rho_{new}/\|r\|_2^2$
18:     **if** $\hat{\rho} < \texttt{tolM}$ **then**
19:         Restart at line 1 with $x_0 \leftarrow x_0 + dx,\ \ \gamma = \delta{*}(\texttt{tolM} - \hat{\rho}),\ \ M \leftarrow M + \gamma{*}I$
20:     **end if**
21:     $p \leftarrow z + (\rho_{new}/\rho){*}p$
22: **end for**
23: $x = x_0 + dx$

The matrix-vector product in line 16 can be parallelized. Benzi et al. [21] introduce a sparse approximate inverse of the Cholesky factor, again allowing a parallel product. However, their method does not parallelize the computation of the preconditioner, and should not be expected to work when Cholesky fails. (SSAI does not depend on Cholesky factorization.) Benzi and Tůma [22] later introduce a method that does not depend on the Cholesky factors, but rather on A-orthogonalization (A-orth.). This method has the advantage of always succeeding, but it cannot be computed or applied in parallel. Some known methods such as [148] aim to compute a preconditioner to an HPD matrix, but use it on the left or right with more general methods such as GMRES, because the resulting matrix AM or MA is not guaranteed to be Hermitian. These methods are far less efficient than PCG, which is tailored for HPD matrices and has fixed storage.

2.5 Numerical results on HPD systems

We compare the performance of SSAI (Algorithm 4) with the ichol, michol, and rchol (in Section 2.5.2 only) preconditioners and the intrinsic backslash (\) solver , using MATLAB on a serial PC. For all methods except rchol, we apply Algorithm 2 to HPD matrices A to obtain a scaled matrix A with unit diagonal. We then attempt to solve $Ax = b$, where $b = Aw$ and $w^T = [1, 2, \ldots, n]/n$. The first

Table 2.1: Abbreviations for SuiteSparse problem fields.

Field Name	**Abbreviation**	**Field Name**	**Abbreviation**
Acoustics	ACO	Combinatorial problems	COMB
Circuit Simulation	CS	Computational Fluid Dynamics	CFD
Computer Graphics and Vision	CGV	Electromagentics	EM
General Least-Squares	GLS	Geomechanics	GM
Power Network	PN	Structural Mechanics	SM
Thermal Problem	TP	Undirected Graph	UG

two sections use matrices from the SuiteSparse collection, and the other sections contain scale-up tests for problems submitted to SuiteSparse and also available in [135].

Note that the MATLAB implementation of SSAI handles both real and complex systems (without change), but SuiteSparse contains only one complex HPD matrix `mhd1280`. Timing results were not included because of the matrix's size, but all methods worked.

ichol and michol are computed by the built-in function `ichol(A)` with `opts.type = 'nofill'` and `opts.michol = 'off'` and `'on'` respectively. rchol is computed with the package from [35]. We implement SSAI as in Algorithm 4 and use it with PCG as in Algorithm 5. For ichol, michol, and rchol, a simpler form of PCG replaces the product with M by two triangular solves (corresponding to the ichol factors). Backslash computes the Cholesky factors of A if possible, and then two triangular solves. Otherwise, it tries MATLAB's `ldl` function (LDL^H with possibly indefinite D) or sparse LU factorization. Note that decompositions take far more time than triangular solves. If there are multiple right-hand sides, it is best to compute the decomposition explicitly using $\tilde{A} =$ `decomposition(A,'chol')` and $x = \tilde{A}\backslash b$ for each b.

In Algorithm 4 (SSAI) we use $\texttt{itmax} = 2{*}\texttt{lfil}$ and $\texttt{lfil} = \texttt{ceil}(\text{nnz}(A)/n)$, giving $\text{nnz}(M) \approx \text{nnz}(A)$. These values were chosen to be roughly consistent with ichol, and for the same reason of not substantially affecting the run-time of each PCG iteration. It is possible that different values would give better results, including selecting `lfil` on a column by column basis, but here we focus on the method itself and not optimizing on the edges.

In Algorithm 5 (modified PCG) we use $\texttt{tol} = 10^{-8}$, $\texttt{tolM} = 10^{-2}$, $\delta = 10$. Final relative residuals $\|b - Ax\|_2/\|b\|_2$ were verified to be below `tol`.

In the tables of results, michol is omitted because when it worked, the number of iterations was very similar to that of ichol, but it was less robust, frequently encountering a non-positive pivot even when ichol worked seamlessly. ichol and '\' are also omitted if they failed. rchol is omitted for non-SDD matrices. T_1 and T_2 are the times (in seconds) for algorithms 4 and 5. For '\', only total time is reported as T_2. For readability, Table 2.1 stores a key for the fields of the matrices. Additionally, both n and nnz are rounded after 3 digits, with K denoting thousands and M denoting millions. Here and in the other tables, $\text{nnz}(M)$ refers to $2 * \text{nnz}(L)$ for ichol and rchol.

The MATLAB code for algorithms 4 and 5 is available from [134].

2.5.1 Matrices from SuiteSparse

Results are presented in order of increasing nnz(A) as a proxy for problem difficulty. The number of iterations reported includes iterations before and after M was changed. The number of changes to M is labeled **R**. Evidently with very few corrections to M, SSAI gives an effective HPD preconditioner. Table 2.2 gives results for matrices where nnz(A) > 1M. Only SSAI succeeded on all 19 problems (ichol fails on 10 and '\' fails on 5) and it is the most easily parallelized method. Here and in other cases, when ichol failed it was because it encountered a non-negative pivot, and when '\' failed it was because it ran out of memory. The memory capacity could be increased, but the point is for a given memory size, and increasing problem size, '\' will (almost) always run out of memory long before any iterative method. Along with the scale-up tests in the following sections, this table illustrates that for large matrices that are not bandwidth-limited, SSAI is the most robust of the methods. There is a way to ensure that ichol succeeded with the '`diagcomp`' option with parameter α, which calculates ichol on `A` + α`*diag(diag(A))`. (For scaled A, this is equivalent to `A` + α`*speye(n)`.) However, there is no foolproof way for selecting α. On the matrices tested, $\alpha = 0.1$ did not work but $\alpha = 1$ did. In the latter case, the PCG time T_2 was substantially larger than with SSAI. MATLAB advises that α = `max(sum(abs(A))./diag(A)) - 2` guarantees `A` + α`*diag(diag(A))` is diagonally dominant—a sufficient condition for ichol to succeed, but a too conservative one. It led to $\alpha \approx 10$ and even more iterations. We recommend ichol be used only when it works with $\alpha = 0$.

We see also that the cost per iteration of PCG + SSAI is always less than PCG + ichol. Additionally, the SSAI preconditioner is applied as a product and can be parallelized more efficiently. It follows that if the number of PCG + ichol iterations is comparable to PCG + SSAI, which indeed occurs frequently, SSAI would be preferred even when the other methods work. Note that ichol and '\' are intrinsic MATLAB functions coded in `C` and can be expected to outperform M-files such as algorithms 4 and 5, or PCG with ichol. The disparity between run times of intrinsic and `.m` functions in MATLAB can be 3 orders of magnitude [12, 107]. Thus, although T_1 is often significant for SSAI, if efficiently implemented it could outperform ichol on large systems (given that usually nnz(M) < 2∗nnz(L)). Also, the T_1 cost is less significant when there are multiple bs.

The three largest SuiteSparse matrices are Queen_44147, Bump_2911, and Flan_1565, with n = 4M, 2.9M, and 1.6M. A comparison with FSAIE-Comm [110] showed SSAI taking about a third as many iterations (1504:4755, 784:2206, 1425:4578). The FSAIE preconditioner is guaranteed to be SPD, but PCG/SSAI restarted only once on Queen_44147 to obtain an SPD $M + \gamma I$, and the other two problems required no restarts. This motivates a more optimized (and parallel) implementation of SSAI.

For StocF-1465 (which took many PCG iterations and had a very sparse M), Table 2.3 tests if larger values of `lfil` and `itmax` give a better M. Monitoring nnz(M) shows that Algorithm 4 terminates mostly because of `itmax` and not `lfil`. To avoid discrepancies in run-time unrelated to the algorithm, we define a measure **Obj** = $\nu\,[\text{nnz}(A) + \text{nnz}(M)]$ ∗ `Itns` that roughly represents

the number of operations completed by PCG, where ν is a constant chosen such that $\mathbf{Obj} = 1$ for `lfil` $=$ `ceil`$(\text{nnz}(A)/n) = 15$. We see that the number of iterations is monotonically decreasing and **Obj** has a downward trend. We conclude that for some applications, tweaking the parameters `lfil` and `itmax` can be advantageous.

2.5.2 SDDM matrices

There are only four SDDM matrices in our SuiteSparse test set (table 2.2 and smaller problems), and only one remained SDDM after scaling (shallow_water2). Table 2.4 compares ichol and SSAI with rchol on three of the matrices, using $b = Aw$ before A and b were scaled for ichol and SSAI. (This is different from Table 2.2, where we assume the scaled matrix DAD is the actual "A", and $b = Aw$ is defined from the scaled A.) Note that scaling A to DAD does not affect definiteness, but it can destroy or create the SDDM property required by rchol. Thus, the units of A and x must be chosen carefully before rchol is applied. Also, rchol forms the Laplacian matrix $\begin{pmatrix} A & -Ae \\ -e^TA & e^TAe \end{pmatrix}$, factorizes it, and takes the top-left $n \times n$ block of the $(n+1) \times (n+1)$ factor. On the matrix Andrews, rchol failed because the $n \times n$ block was singular to machine precision. (This is an example of a theoretical property not holding up numerically.) As the goal of Table 2.4 is to make comparisons with rchol, we omit Andrews from the table. We tuned ichol and SSAI to have $\text{nnz}(M)$ similar to rchol, though this does not optimize their performance. ichol and rchol performed well on all three problems. SSAI was comparable on the first two problems, but needed more PCG iterations on ecology2.

2.5.3 Scale-up for Trefethen challenge matrices

Table 2.5 shows scale-up results for a challenge matrix [161, 162] with increasing prime numbers on the main diagonal and 1s wherever $|i - j| = 2^N$ for some non-negative integer N. The original matrix has $n = 20,000$, and the challenge is to compute $e_1^T A^{-1} e_1$. SuiteSparse contains case $n = 2,0000$. We generated two much larger matrices of the same form. They remind us that iterative methods are essential for systems that have substantial fill-in in their Cholesky factors. For ichol and SSAI, T_2 scales linearly with n (with SSAI being twice as fast as ichol). On these problems, '\' scales poorly because the bandwidth grows linearly with n. It failed on even the moderately sized Trefethen_200000. The PCG iterations remain roughly constant because the matrix becomes more diagonally dominant near the bottom right corner, and the solution vector is also concentrated at the bottom. We checked all methods on the original Trefethen problem by solving with $b = e_1$. Their performance was similar. For each n we recovered the first ten digits of the solution: $0.7250783462, 0.7250809785, 0.7250812561$ (found by all methods that worked). The $n = 20,000$ value agrees with [161]. A later publication by some of the challenge participants [26] found the same values with PCG + a diagonal preconditioner for the largest three problems. We

note that PCG with `tol` $= 10^{-11}$ took 14 iterations with a diagonal preconditioner [26], but only 6 with SSAI. The relatively high cost of calculating M suggests that for one b, it may not be worth the trouble. However, if we were interested in multiple bs, such as finding A^{-1}, we would only need to find M once, making it negligible in the overall cost. The number of iterations needed to solve each system $b = e_j$ is constant empirically and analytically [26]. In fact, with the matrix becoming more diagonally dominant near the bottom, the number of iterations decreases with j (for example: $n = 20,000$, $j = 20,000$ converged in 2 iterations). $\mathrm{nnz}(A) = O(n \log(n))$ means that PCG with any of the three preconditioners can find A^{-1} in $O(n^2 \log(n))$ time, as matrix-vector products with A (or M) take up most of the computation. SSAI has the smallest constant.

2.5.4 Scale-up for finite-element linear elasticity matrices

Table 2.6 shows scale-up results for a structural mechanics problem that results from discretizing a cube using quadratic hexahedral finite elements. All elements are cubes, leading to a relatively well-conditioned A. We show results for $8, 16, 32$ elements per direction. As in Table 2.5, '\' failed, though later because the bandwidth of A grows less rapidly with n compared to the Trefethen problems. Both iterative methods work well. SSAI needs more PCG iterations but less PCG time. The PCG iterations roughly double as the number of elements increases—a small price for 2X the resolution in all three directions.

2.5.5 Scale-up for finite-volume petroleum engineering matrices

Table 2.7 shows scale-up results for SPE10, an important benchmark for testing solver methods in petroleum engineering [44]. Examples were constructed by Klockiewicz [102] using the methods in [115] for a petroleum reservoir simulation. Here again, '\' failed for the larger problems, while both iterative methods work well. As with the preceding finite-element matrices, ichol needs fewer PCG iterations but more PCG time. For ichol, $\mathrm{nnz}(M) = 2 * \mathrm{nnz}(L)$. For SSAI, $\mathrm{nnz}(M) \approx 1.1 * \mathrm{nnz}(A)$.

2.5.6 Scale-up for finite-element ice-sheet matrices

Table 2.8 shows scale-up results for a model of ice flow in Antarctica [32, 159]. SSAI is the only method that works on all cases, with '\' failing for the larger ones and ichol failing on all of them. For SSAI, $\mathrm{nnz}(M) \approx 0.3\text{–}0.4\,\mathrm{nnz}(A)$, leaving room for a denser M.

2.6 Least-squares problems

We consider least-squares problems $\min_x \|Ax - b\|_2$ with rectangular $A \in \mathbb{R}^{m \times n}$ and $m > n$. If A is sparse and $\tilde{A} \equiv A^H A$ is not too dense, a naive approach is to apply PCG + SSAI to the normal equation $A^H A x = A^H b \equiv \tilde{A} x = \tilde{b}$. The system is HPD iff A has full column rank. Otherwise, $\tilde{A}$

Algorithm 6 Modified PCGLS Typical parameters: `tolM` $= 10^{-2}$, $\delta = 10$

1: $r \leftarrow b - A{*}x_0,\ \ dx \leftarrow 0_{n\times 1}$
2: $t \leftarrow A^H{*}r$
3: $u \leftarrow w \leftarrow M{*}t$
4: $\gamma_{new} = t^H{*}w$
5: **for** $j \leftarrow 1$ `to n` **do**
6: $\quad q \leftarrow A{*}u,\ \ \gamma \leftarrow \gamma_{new}$
7: $\quad \alpha \leftarrow \gamma/\|q\|_2^2$
8: $\quad dx \leftarrow dx + \alpha{*}u$
9: $\quad r \leftarrow r - \alpha{*}q$
10: $\quad t \leftarrow A^H{*}r$
11: $\quad$ **if** $\|t\|_2/\|b\|_2 <$ `tol` **then**
12: $\quad\quad$ break
13: $\quad$ **end if**
14: $\quad w \leftarrow M{*}t$
15: $\quad \gamma_{new} = t^H{*}w,\ \ \hat\gamma = \gamma_{new}/\|t\|_2^2$
16: $\quad$ **if** $\hat\gamma <$ `tolM` **then**
17: $\quad\quad$ Restart at line 1 with $x_0 \leftarrow x_0 + dx,\ \ M \leftarrow M + (\delta{*}(\texttt{tolM} - \hat\gamma)){*}I$
18: $\quad$ **end if**
19: $\quad u \leftarrow w + (\gamma_{new}/\gamma){*}u$
20: **end for**
21: $x = x_0 + dx$

is Hermitian positive semi-definite (and singular) but a solution always exists, i.e., the system is compatible.

For the normal equations, Hestenes and Stiefel [92, §10] gave a special form of CG now known as CGLS [123, §7.1], [24, §7.4.1]. If a preconditioner $M \approx (A^HA)^{-1}$ is known, preconditioned CGLS can be implemented as shown in Algorithm 6 (modified PCGLS), with operators A and M. Some previous implementations of PCGLS work in this way (e.g., Regularization Tools [88] and IRtools [74]), excluding the restarts in lines 16–18. We note that when M is available in factored form ([74, 88] use $M = L^{-T}L^{-1}$ with L triangular), it should be preferable numerically to work with CGLS and operator AL^{-1}, rather than with PCGLS. This is done with ichol in [173] and for SSAI and ichol in Table 2.9.

When A is rectangular and sparse, '\' uses a sparse QR factorization $AP = QR$ to solve $Ax \approx b$ directly without forming A^HA (where P is a column permutation to preserve sparsity, $Q^HQ = I$, and R is upper triangular). In general, if $\tilde A$ is sparse, the high efficiency of sparse Cholesky compared to sparse QR makes $x = \tilde A\backslash\tilde b$ outperform $x = A\backslash b$, as in the examples here. Thus, we use $\tilde A$ and $\tilde b$ for all methods.

Using the same methods and comparisons as in Section 2.5, we compile results for SSAI on least-squares problems. Normalizing each column of A gives $\tilde A$ a unit diagonal, and then we can apply algorithms 4 and 5. We set $b = A^Te$ and use $\tilde b = A^Hb$, which is scaled with the same diagonal matrix used to normalize A.

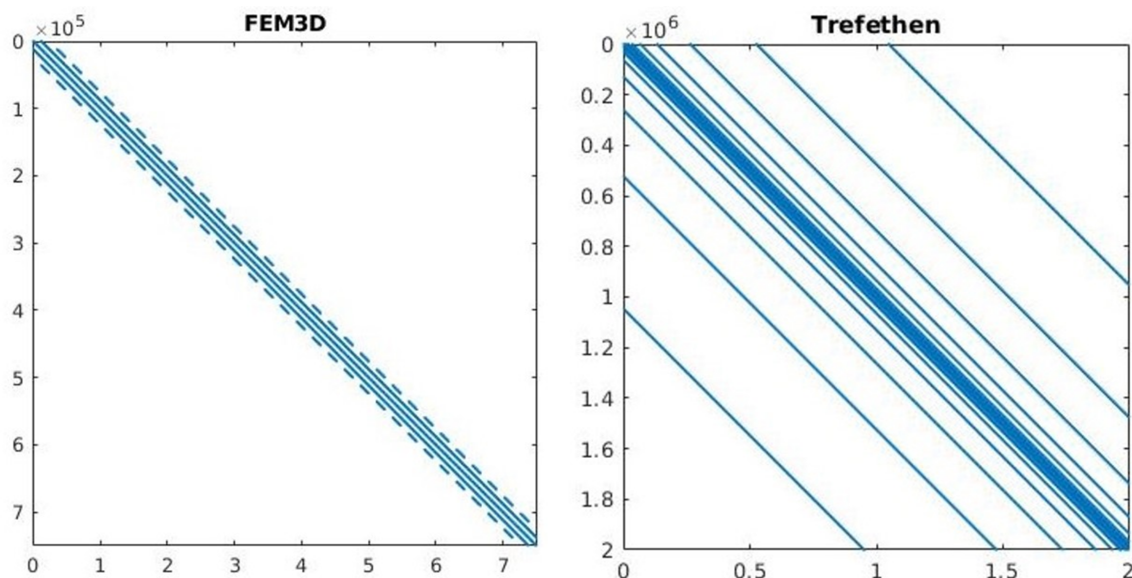


Figure 2.1: Sparsity structures of some square HPD problems. Although exact Cholesky factors would be essentially dense (within the outermost band), ichol and SSAI both perform well.

Table 2.9 shows results for matrices [49] ordered by increasing $\text{nnz}(A^HA)$, but keeps problems of the same type together. As in Section 2.5, when a method failed, it is omitted from the table for that matrix.

We see that SSAI is again the most robust of the three methods. It is the only method to work on `ch8-8-b5`, and it outperforms the other methods sometimes even when they work. ichol and SSAI's T_2 scales linearly on `ch8-8-b`-type problems. SSAI was twice as fast as ichol until ichol failed.

Section 2.7 explains the seemingly surprising result of '\' working well on the large `Hardesty3`, but not at all on the substantially smaller `ch8-8-b4` and `ch8-8-b5`.

Our experiments with PCGLS have obtained their preconditioner M by applying Algorithm 4 to A^HA. When A contains some relatively dense rows, special methods are needed to avoid forming A^HA (and SSAI is not practical). Several such methods have been given by Scott and Tůma [150–152].

2.7 Sparsity structure of A

We examine the sparsity structure of some of our matrices in figures 2.1–2.2, where for rectangular matrices (last two) we look at the sparsity of A^HA in addition to A. We look at the largest matrix in each of four classes. The sparsity structure of smaller matrices is essentially identical. The banded structure of A^HA is apparent in `Hardesty`, allowing a dense band solver (or sparse Cholesky) to

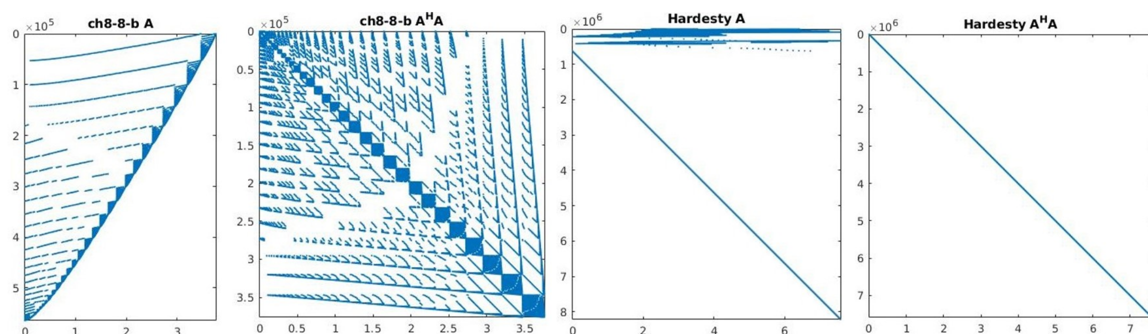


Figure 2.2: Sparsity structures of A in rectangular problems, compared to the sparsity structure of A^HA. The structure makes it clear why an iterative method would outperform a direct method on `ch8-8-b`, and vice versa for `Hardesty`.

solve the problem easily. PCGLS+SSAI iterations scale as $\sim 50n^{0.3}$, which could be reasonable were it not for the superiority of direct methods on banded problems. `Trefethen` and `ch8-8-b` are difficult for direct solvers because of the substantial fill-in. However, the low magnitude of most of the omitted entries allows PCG + SSAI to converge with a constant number of iterations on these problems. `FEM3D` has the classic sparsity structure of finite-element matrices. It is less banded than `Hardesty`, but substantially more banded than `Trefethen` and `ch8-8-b`. It is a middle ground of sorts, with '\' scaling poorly but not quite as poorly as for `Trefethen` and `ch8-8-b`. PCG + SSAI iterations scale as $\sim n^{0.3}$ (very low), making it a good choice on these problems, while leaving room for a different iterative solver to do better.

In Fig. 2.2, note that the `Hardesty3` matrix A appears to have a significant number of relatively dense rows, suggesting that the special methods of Scott and Tůma [150–152] might be needed to avoid a dense A^HA. However, the nonzeros in A are integers and there must be exact cancellation when A^HA is formed, as it proves to be banded as shown. `Hardesty3` is a deceptive test matrix.

2.7.1 Performance of ichol vs SSAI

To gain understanding of how ichol and SSAI compare, we examine two small matrices: `1138_bus` (where ichol substantially outperformed SSAI) and `sts4098` (where SSAI worked well but ichol failed). Timing results are not included because of the matrices' small size. The inverses of both matrices are fully dense.

For complete Cholesky factorization of `1138_bus`, $\text{nnz}(L) \approx 38.3K$ means that the factor stays relatively sparse and the error introduced by ichol in dropping the fill-in does not create a negative pivot to break the factorization. Fig. 2.3 shows the sparsity pattern. As the entries are similar in magnitude, SSAI introduces significant error when ignoring many somewhat small ones.

For complete Cholesky factorization of `sts4098`, $\text{nnz}(L) \approx 4.68M$ means the factor is essentially

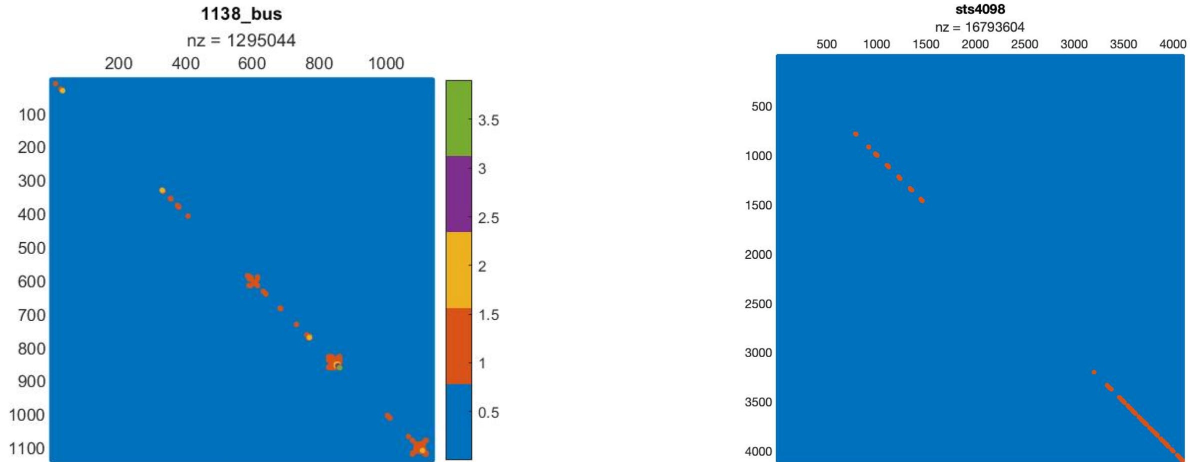


Figure 2.3: Sparsity pattern entry magnitude of `1138_bus` and `sts4098`. The larger entries are concentrated near the diagonal. On `1138_bus`, SSAI cannot choose easily among the many entries r_i of similar magnitude.

dense (about half of the entries are nonzero). ichol fails on this matrix because it throws away almost all entries of the complete factor, and eventually enough error is introduced to make a pivot negative. SSAI is able to find the larger entries and prioritize them in the factorization. This can be done with ichol with fill-in as well, but an entry must be calculated before the decision is made to remove it from the factorization, and this is very costly.

2.8 Parallelism and complexity

Parallelism requires examination of run-time dependence on n, $k \equiv \text{nnz}(A)$, and number of processors p. We can assume that communication between processors is negligible for $p \ll n$, but verification of this assumption and the following idealized analysis remains for future work. For a nonsingular A, $n \leq k \leq n^2$. The outer loop of Algorithm 4 is trivially parallelizable on up to n machines because each column is computed separately. Symmetrization of M (Line 16) could be parallelized with greater effort, but it is only an $O(k)$ operation occurring once. Lines 6–12 are executed $O(k)$ times, requiring $O(k/n)$ arithmetic operations each time. Line 14 runs n times with $O(k/n)$ operations each time. The time to compute M is therefore $T(A) \equiv O(k^2/(np))$.

Algorithm 5 is dominated by matrix-vector products that require $O(k)$ operations. In exact arithmetic, the loop runs at most n times and at least once. It must happen sequentially. On parallel machines, matrix-vector products can be done row by row, so with p machines they take $O(k/p)$ time. The norm on line 13 is an $O(n/p)$ operation because the contribution of each processor

must be summed. Therefore, the run-time for PCG on p machines is $\tau(A) \equiv O(kn/p)$. The analysis of Algorithm 6 is similar with A replaced by $A^H A$.

When $\tau \gtrsim T$ it could help to use larger `lfil` and `itmax` in SSAI. This was true on `StocF-1465` (which needed the most PCG iterations among the SuiteSparse matrices).

The complexities $T(A)$ and $\tau(A)$ should be taken into account when we construct M and when we try to bound the size of A itself (e.g., if it comes from finite-element discretization). An extensive study such as for FSAI [105] is needed to verify how these theoretical results for parallel machines hold up in practice.

2.9 Discussion

The numerical results show that even on serial machines, SSAI is competitive with ichol preconditioners and direct methods for large HPD linear systems whose Cholesky factors are denser than A. The total PCG + SSAI iterations scales sublinearly with n for all problems, and for some problem classes of increasing dimension is even constant. Moreover, PCG + SSAI is the most robust of the four methods, and guarantees a solution in a wider range of cases. For small systems, the overhead of computing the preconditioner is considerable, although this is partly due to inefficient implementation compared to built-in functions. The overhead for computing M is less significant when there are several right-hand sides or several similar matrices, such that the preconditioner M can be reused. Although there is no 'one size fits all' preconditioner, our results suggest that SSAI for HPD systems is suitable if some of the following criteria are met: n or $k \equiv \text{nnz}(A)$ is large, A arises from finite-element schemes, A is not banded and factors of A would be too dense, or parallelization is essential. For general systems or least-squares problems, PCGLS + SSAI can be effective methods. Further considerations then are the dimensions m, n and the sparsity of $A^H A$. CG normally terminates when the residual norm is sufficiently small (line 13 of Algorithm 5), which means the conjugate residual method (CR [92]) and MINRES can always terminate sooner than CG [71]. Thus, CR + SSAI or MINRES + SSAI are viable alternatives to PCG + SSAI if precautions are added to increase the positive definiteness of M and restart (lines 17–20 of Algorithm 5), as has been done in the MINRES drivers of [136].

A future research direction would be to find a more efficient SSAI-type algorithm to derive a preconditioner for more general matrices; for example, a SSAI-type preconditioner for LSQR and LSMR [70, 123] on square or rectangular systems, where $A^H A$ is not explicitly formed. For symmetric indefinite systems, we can hope to do better.

Another direction is to make parallel implementations of SSAI for CPUs and GPUs. A recent work on GSAI-Adaptive [73] found that SPAI algorithms for unsymmetric $Ax = b$ can be effectively implemented on GPUs. A natural continuation of that work and ours would be to implement SSAI in a similar way, in order to take advantage of the symmetry and positive definiteness of SPD problems.

2.9.1 Comparison of preconditioners for iterative methods for HPD problems

Table 2.10 gives a high-level comparison of different preconditioners for solving HPD systems. This table motivates an optimized implementation of SSAI because it is an otherwise competitive method. Naturally, different methods will work better for different problems, so there is no clear winner. Rather, this table should be used as a guide to which methods to try when attempting to solve a given problem based on the available matrix data and hardware.

2.9.2 Indefinite A

Any Hermitian A has a decomposition $PAP^T = LDL^T$ for some permutation P, where L is lower-triangular and D is block-diagonal [29]. If $\tilde{A} \approx A$ and a sparse decomposition $P\tilde{A}P^T = LDL^T$ is available (when factorizing A would be too expensive), we can obtain an HPD preconditioner $M = L|D|L^T$ by setting $D_{ii} = |D_{ii}|$ for 1×1 blocks and replacing each 2×2 block B by $V|\Lambda|V^H$, where $B = V\Lambda V^H$ is the eigensystem of B [77]. Greif, et al. [85] give a comprehensive C++ package SYM-ILDL for computing such preconditioners for indefinite and skew-symmetric A, for use with MINRES and SQMR.

If $\tilde{A}$ is quasidefinite [165], a sparse Cholesky-type factorization $P\tilde{A}P^T = LDL^T$ exists for any permutation P with D diagonal and indefinite, and we would use $M = L|D|L^T$.

These "incomplete LDL" methods can be very effective, but are unlike the SSAI approach.

2.9.3 MINRES and preconditioners for indefinite A

Theorem 2. *An indefinite left-right preconditioner cannot be used with MINRES-type methods [40, 122] on indefinite A without explicit knowledge of the preconditioner's factors. (Note that if the factors are known explicitly, QMR [72] can be used.)*

Proof. MINRES solves $Ax = b$ with Hermitian A. As real symmetric (RS) matrices are a subset of both Hermitian matrices and complex symmetric (CS) matrices, we restrict our attention to RS nonsingular A. Suppose we have an RS nonsingular preconditioner M with eigensystem $M = QDQ^T = Q\sqrt{D}\sqrt{D}Q^T = Q\sqrt{D}Q^TQ\sqrt{D}Q^T = CC$, where $Q^TQ = I$, $\sqrt{D}_{ii} = \sqrt{D_{ii}}$, and $C = Q\sqrt{D}Q^T$. If M is SPD, $\sqrt{D}$ is real-valued, $C = C^H = C^T$, and the system can be rewritten as in Eq. (2.2). However, if M is indefinite, $\sqrt{D}$ has some pure real and pure imaginary diagonal entries, and $C = C^T \neq C^H$. We therefore rewrite $Ax = b$ as

$$CAC^Ty = Cb, \quad x = C^Ty, \tag{2.3}$$

where Eq. (2.3) is not Hermitian but rather CS(!). □

Hence, an (indefinite) approximate inverse for indefinite A is not usable as a preconditioner. We note that an indefinite M can be used as a preconditioner if A is HPD [15], but as we know that the actual inverse is HPD, an approximate inverse preconditioner of this form is not necessarily desirable.

We conclude that for SSAI on indefinite A, knowing an indefinite $M = C^T C$ (where C is complex) such that $CAC^T \approx I$ is insufficient to precondition the problem. We must obtain the factor C explicitly and solve Eq. (2.3), or abandon the left-right approximate inverse method as a preconditioner for a symmetric indefinite system.

2.9.4 Signature matrices

A nonsingular matrix A is *Hermitian quasidefinite* (HQD) if for any permutation P, the factorization $PAP^T = LDL^H$ exists with L lower triangular and D nonsingular, indefinite, and diagonal (as opposed to D being block-diagonal).

Theorem 3. *If A is symmetric and indefinite, the existence of an HPD $\widetilde{M} = T^H T$ such that $TAT^H = S$, where S is a signature matrix (a diagonal matrix with entries ± 1), is equivalent to A being HQD.*

Proof. If A is HQD, then for any permutation P we have $PAP^T \equiv \tilde{A} = LDL^H \Rightarrow L^{-1}\tilde{A}L^{-H} = D$. If we define $\widetilde{D}$ such that $\widetilde{D}_{ii} = 1/\sqrt{|D_{ii}|}$, we have $\widetilde{D}L^{-1}\tilde{A}L^{-H}\widetilde{D} = \widetilde{D}D\widetilde{D} \Rightarrow TAT^H = S$, where $T \equiv \widetilde{D}L^{-1}P$ and $S \equiv \widetilde{D}D\widetilde{D}$.

Conversely, $TAT^H = S \Rightarrow PAP^T = L\widetilde{D}^{-1}S\widetilde{D}^{-1}L^H = LDL^H$. As L is lower triangular and $\widetilde{D}^{-1}S\widetilde{D}$ is diagonal, this means A is HQD. □

Thus, the strategy of finding $\widetilde{M} = T^H T$ such that $TAT^H = S$ is useful for quasidefinite matrices, and *only* for them. While SSAI is effective for HPD systems, finding an analogous HPD preconditioner for indefinite systems for use with MINRES remains an open question, but the Algorithm 5 approach of restarting PCG with $M \leftarrow M + \gamma I$ has already proved helpful for MINRES in the MINRES drivers of [136].

2.10 Summary

SSAI is a new method for obtaining an HPD preconditioner for HPD systems $Ax = b$. The preconditioner M does not require a prescription of sparsity patterns or reordering of the unknowns, its computation and application are embarrassingly parallel, it can be made as sparse or dense as we wish via a few parameters, and PCG and MINRES can easily request diagonal modification to ensure that M is HPD. We show how SSAI compares with MATLAB's $A\backslash b$ and ichol on many matrices from SuiteSparse. On the largest examples, SSAI is the only successful method. Out

of the 65 problems tested, SSAI succeeded on all of them, while ichol failed on 26 and '\' failed on 17. We also show how the method can be used with PCGLS to solve general rectangular systems $\min \|Ax - b\|$. Section 2.9 includes new theoretical results about methods for preconditioning indefinite and quasidefinite systems.

MATLAB code for algorithms 4–6 (SSAI, PCG, PCGLS) is available from [134]. The implementations are like ichol in being applicable to a wide range of problems without careful choice of parameters for each problem. Analogous preconditioners for MINRES (SSAI, Mminres2, Mminres3, Mchol, Mldl) are given in [136], including drivers that restart MINRES with $M \leftarrow M + \gamma I$ when necessary, as we do here within PCG.

Acknowledgements

We thank Eric Darve, Bazyli Klockiewicz, and Leopold Cambier for consultation and data regarding the SPE10 and Antarctica matrices. We also recognize the invaluable resource that the SuiteSparse collection [49] represents.

Table 2.2: SuiteSparse $Ax = b$, nnz$(A) > 1M$. **R** is the number of restarts in Algorithm 5 (PCG + SSAI). SSAI succeeded on all 19 problems, while ichol failed on 10, and '\' failed on 5.

Field	**Matrix**	n	nnz(A)	nnz(M)	**Method**	**Itns**	**R**	T_1	T_2
EM	2cubes_sphere	102K	1.65M	1.75M	ichol	9	-	3.00e−2	9.39e−2
				-	\	-	-	-	2.86e+0
				1.98M	SSAI	10	0	1.82e+1	7.36e−2
ACO	qa8fm	66K	1.66M	1.73M	ichol	6	0	1.20e−2	4.34e−2
				-	\	-	-	-	1.08e+0
				1.71M	SSAI	11	0	3.07e+1	6.59e−2
CFD	cfd2	123K	3.09M	-	\		-	-	2.15e+0
				3.24M	SSAI	1235	1	5.32e+1	1.38e+1
CFD	parabolic_fem	526K	3.67M	4.2M	ichol	697	-	3.61e−2	1.76e+1
				-	\	-	-	-	1.57e+0
				3.69M	SSAI	892	0	3.63e+1	1.95e+1
UG	pdb1HYS	36.4K	4.34M	4.38M	ichol	276	-	1.26e−1	3.57e+0
				-	\		-	-	7.40e−1
				4.8M	SSAI	511	2	7.75e+1	6.46e+0
PN	ecology2	1M	5M	6M	ichol	1717	-	3.94e−2	7.44e+1
				-	\	-	-	-	1.94e+0
				5M	SSAI	3248	0	4.58e+1	7.15e+1
EM	tmt_sym	727K	5.08M	5.81M	ichol	1043	-	4.87e−2	39.9e+1
				-	\	-	-	-	2.49e+0
				5.38M	SSAI	1976	0	4.90e+1	5.95e+1
SM	boneS01	127K	5.52M	-	\	-	-	-	1.81e+0
				6.31M	SSAI	651	0	8.17e+1	8.69e+0
CS	G3_circuit	1.59M	7.66M	9.25M	ichol	564	-	9.44e−2	3.91e+1
				-	\	-	-	-	8.41e+0
				8.39M	SSAI	1128	0	8.48e+1	5.47e+1
TP	thermal2	1.23M	8.58M	9.81M	ichol	1814	-	1.68e−1	1.49e+2
				-	\	-	-	-	4.24e+0
				9.22M	SSAI	2417	0	8.66e+1	1.25e+2
SM	hood	221K	9.9M	-	\		-	-	1.03e+0
				8.98M	SSAI	568	1	1.88e+2	1.82E+1
SM	crankseg_2	63.8K	14.1M	-	\		-	-	2.52e+0
				13.1M	SSAI	161	3	4.21e+2	8.36e+0
CFD	StocF-1465	1.47M	21M	9.66M	SSAI	7822	1	3.85e+2	5.20e+2
SM	Fault_639	639K	27.2M	21.6M	SSAI	660	1	5.03e+2	5.27e+1
SM	boneS10	915K	40.9M	-	\	-	-	-	1.45e+1
				46.1M	SSAI	9351	0	5.72e+2	9.60e+2
SM	bone010	987K	47.9M	-	\	-	-	-	1.93e+2
SM				52.4M	SSAI	3146	0	6.95e+2	3.69e+2
SM	Flan_1565	1.56M	114M	116M	ichol	1947	-	2.28e+0	7.23e+2
				124M	SSAI	1425	0	1.88e+3	3.75e+2
GM	Bump_2911	2.91M	128M	89.9M	SSAI	784	0	2.35e+2	2.32e+2
SM	Queen_4147	4.15M	317M	142M	SSAI	1504	1	7.10e+3	8.36e+2

Table 2.3: StocF-1465 $Ax = b$ with varying `lfil` and `itmax`. **R** is the number of restarts in Algorithm 5. **Obj** $= \nu\,[\text{nnz}(A) + \text{nnz}(M)] *$ `Itns` is a normalized measure of operations in PCG. Increasing `lfil` and `itmax` helps the overall performance.

`lfil`	`itmax`	nnz(M)	nnz(M) + nnz(A)	**Itns**	**R**	**Obj**
20	40	11M	32M	7690	1	1.03
30	60	13.3M	34.3M	6766	1	0.97
40	80	15.3M	36.3M	6072	1	0.92
50	100	17M	38M	5487	1	0.87
60	120	18.5M	39.5M	5263	1	0.87
70	140	20M	41M	5156	1	0.88
80	160	21.3M	42.3M	4660	2	0.82
90	180	22.6M	43.6M	4631	1	0.84
100	200	23.8M	44.8M	4315	1	0.81

Table 2.4: Comparison of ichol and SSAI with rchol on SDDM systems $Ax = b$ from SuiteSparse. rchol requires A to be SDDM. A was not scaled for rchol because only shallow_water2 remains SDDM with scaling. Scaling was performed on ichol and SSAI for better efficiency. rchol performed well on these three examples (especially on ecology2), but failed on a fourth SDDM matrix (Andrews, therefore omitted).

Field	**Matrix**	n	nnz(A)	nnz(M)	**Method**	**Itns**	T_1	T_2
OPT	torsion1	40K	198K	395K	ichol	9	3.60e−3	2.00e−2
				386K	rchol	23	6.44e−2	4.78e−2
				362K	SSAI	29	3.57e+0	2.78e−2
CFD	shallow_water2	81.9K	328K	491K	ichol	8	5.91e−3	2.71e−2
				491K	rchol	16	9.88e−2	6.03e−2
				472K	SSAI	15	4.46e+0	2.85e−2
PN	ecology2	1M	5M	20.9M	ichol	455	2.84e−1	3.25e+1
				19.8M	rchol	52	2.00e+0	3.98e+0
				20.9M	SSAI	2311	3.68e+2	8.78e+1

Table 2.5: Scale-up for Combinatorics $Ax = b$. PCG + SSAI required no restarts. ichol and SSAI succeeded on all 3 problems, and '\' failed on the largest 2.

Matrix	n	nnz(A)	nnz(M)	**Method**	**Itns**	T_1	T_2
Trefethen_20000	20K	554K	574K	ichol	5	5.73e−3	1.60e−2
			-	\	-	-	5.95e+0
			575K	SSAI	3	5.27e+0	1.03e−2
Trefethen_200000	200K	6.88M	7.08M	ichol	4	9.97e−2	1.94e−1
			7.14M	SSAI	3	7.62e+1	9.90e−2
Trefethen_2000000	2M	81.8M	83.8M	ichol	3	2.21e+0	2.80e+0
			83.9M	SSAI	2	9.58e+2	1.02e+0

Table 2.6: Scale-up for ordered grid structural mechanics $Ax = b$. PCG + SSAI required no restarts. ichol and SSAI succeeded on all 3 problems, and '\' failed on the largest 1.

Matrix	n	nnz(A)	nnz(M)	**Method**	**Itns**	T_1	T_2
FEM3D_3	10.1K	1.02M	1.12M	ichol	19	4.79e−2	6.03e−2
			-	\	-	-	5.46e−1
			1.2M	SSAI	25	2.00e+1	6.15e−2
FEM3D_4	89.3K	10.5M	10.6M	ichol	35	5.00e−1	1.17e+0
			-	\	-	-	1.03e+2
			12.3M	SSAI	45	2.29e+2	9.65e−1
FEM3D_5	750K	94.9M	95.6M	ichol	67	4.52e+0	1.97e+1
			110M	SSAI	81	2.35e+3	1.61e+1

Table 2.7: Scale-up for finite-volume $Ax = b$. PCG + SSAI required no restarts. ichol and SSAI succeeded on all 6 problems, and '\' failed on the largest 4.

Matrix	n	nnz(A)	nnz(M)	**Method**	**Itns**	T_1	T_2
lin4E5	422K	2.91M	3.33M	ichol	445	2.33e−2	1.12e+1
			-	\	-	-	8.76e+0
			3.23M	SSAI	661	3.77e+1	7.71e+0
lin1E6	1.12M	7.78M	8.9M	ichol	1138	8.54e−2	7.58e+1
			-	\	-	-	1.03e+2
			8.76M	SSAI	1742	1.01e+2	6.18e+1
lin2E6	2.1M	14.6M	16.7M	ichol	1793	187e−1	2.24e+2
			16.3M	SSAI	2716	1.89e+2	1.82e+2
lin4E6	4.1M	28.5M	32.6M	ichol	2007	2.66e−1	5.06e+2
			31.8M	SSAI	2942	3.66e+2	3.78e+2
lin8E6	8M	55.8M	63.8M	ichol	2957	5.88e−1	2.75e+3
			61.9M	SSAI	4176	7.25e+2	1.18e+3
lin1E7	16M	112M	128M	ichol	2934	1.11e+0	4.48e+4
			124M	SSAI	4205	1.45e+3	2.34e+4

Table 2.8: Scale-up for '\' and SSAI on unordered grid structural mechanics $Ax = b$. SSAI succeeded on all 4 problems, ichol failed on all 4, and '\' failed on 2.

Matrix	n	nnz(A)	nnz(M)	**Method**	**Itns**	**R**	T_1	T_2
ant16_5	630K	29.7M	-	\	-	-	-	1.33e+1
			8.7M	SSAI	5307	1	5.65e+2	2.78e+2
ant16_10	1.15M	57.5M	-	\	-	-	-	1.22e+2
			21.9M	SSAI	8969	2	1.11e+3	8.77e+2
ant8_5	2.52M	120M	52.9M	SSAI	6328	3	2.33e+2	1.36e+3
ant8_10	4.62M	232M	92.4M	SSAI	12055	2	4.66e+3	4.99e+3

Table 2.9: SuiteSparse least-squares problems $\min \|Ax - b\|$ with $A \in \mathbb{R}^{m \times n}$. SSAI and ichol are used with PCGLS. SSAI succeeded on all 9 problems, ichol failed on 4, and ‘\’ failed on 2.

Matrix	**Field**	m	n	nnz(A)	nnz($A^H A$)	nnz(M)	**Method**	**Itns**	**R**	T_1	T_2
image_interp	CGV	240K	120K	712K	1.56M	-	\		-	-	5.18e−1
						1.54M	SSAI	5941	0	2.77e+1	2.29e+1
sls	GLS	1.75M	62.7K	6.8M	4.72M	4.78M	ichol	43	-	4.97e−1	2.68e−1
						-	\	-	-	-	1.81e+1
						7.3M	SSAI	75	2	3.36e+2	3.84e+0
LargeRegFile	CS	2.11M	801K	4.94M	6.38M	7.18M	ichol	24	-	1.45e+2	1.66e+0
						-	\	-	-	-	4.01e+0
						7.38M	SSAI	33	0	76.3e+1	1.23e+0
Rucci1	GLS	1.98M	110K	7.79M	9.75M	9.86M	ichol	90	-	3.99e−1	5.26e+0
						-	\	-	-	-	9.53e+0
						4.13M	SSAI	3557	2	2.06e+2	1.19e+2
ch8-8-b3	COMB	118K	18.8K	470K	1.43M	1.45M	ichol	8	-	4.39e−2	6.76e−2
						-	\	-	-	-	1.09e+2
						995K	SSAI	7	0	3.30e+1	2.61e−2
ch8-8-b4	COMB	376K	118K	1.88M	7.64M	7.76M	ichol	9	-	2.45e−1	3.22e−1
						4.59M	SSAI	9	0	1.70e+2	1.21e−1
ch8-8-b5	COMB	564K	376K	3.39M	17.3M	17.3M	SSAI	5	1	2.30e+2	3.26e−1
Hardesty2	CGV	930K	304K	4.02M	3.94M	-	\		-	-	1.37e+0
						3.96M	SSAI	2871	0	6.91e+1	4.39e+1
Hardesty3	CGV	8.22M	7.59M	40.4M	98.6M	-	\		-	-	4.81e+1
						98.4M	SSAI	7226	0	1.82e+3	2.13e+3

Table 2.10: Comparison of preconditioners for iterative methods on HPD problems. The rchol and spaND methods have further restrictions. rchol requires the matrix to be SDD, or SDDM if we don’t want to double the size of our system, and spaND requires the interfaces between groups of variables to be relatively sparse (i.e., there are only short range interactions between variables). Note that SAINV adds a matrix to A during computation of M to ensure that M is positive definite. This means that A may be modified too much or even unnecessarily unless the process is attempted multiple times. (In contrast, SSAI modifies M, not A, and only as needed.) Sparsity refers to the sparsity structure of the preconditioner and whether it is supplied by the user (fixed) or calculated within the method (flexible). Parallelism refers to the computation of the preconditioner (not its application). FSM says if it can be used with fixed-storage iterative methods like CG and MINRES. Application refers to how the preconditioner is applied. (Product preconditioners are more desirable because they can be applied in embarrassingly parallel fashion, whereas triangular solves involve waiting and communication between dependent equations.) The color blue is used to indicate a desirable property, whereas the color purple is used to indicate an undesirable property. Properties left in black are somewhere in between.

Method	**Sparsity**	**Parallelism**	**FSM**	**Breakdown risk**	**Application**
A-orth.	Fixed	No	Yes	No	Solve
FSAI	Fixed	shared memory	Yes	Yes	Product
ichol	Fixed	fine-grained	Yes	Yes	Solve
rchol	Random	shared memory	Yes	Yes	Solve
SAINV	Flexible	embarrassing	Yes	No	Product
SPAI	Flexible	embarrassing	No	No	Product
spaND	Flexible	shared memory	Yes	No	Solve
SSAI	Flexible	embarrassing	Yes	No	Product

Chapter 3

HyKKT: A Hybrid Direct-Iterative Method for Solving KKT Linear Systems

This chapter contains the full text of [138]. This work was originally published by Optimizations Methods and Software. Reprinted with permission. All rights reserved.

3.1 Overview

We propose a solution strategy for the large indefinite linear systems arising in interior methods for nonlinear optimization. The method is suitable for implementation on hardware accelerators such as graphical processing units (GPUs). The current gold standard for sparse indefinite systems is the $\mathrm{LBL^T}$ factorization , where L is a lower triangular matrix and B is 1×1 or 2×2 block diagonal. However, this requires pivoting, which substantially increases communication cost and degrades performance on GPUs. Our approach solves a large indefinite system by solving multiple smaller positive definite systems, using an iterative solver on the Schur complement and an inner direct solve (via Cholesky factorization) within each iteration. Cholesky is stable without pivoting, thereby reducing communication and allowing reuse of the symbolic factorization. We demonstrate the practicality of our approach on large optimal power flow problems and show that it can efficiently utilize GPUs and outperform $\mathrm{LBL^T}$ factorization of the full system.

3.2 Contributions

- We introduced an approach of solving linear systems arising from optimization problems that solves the set of linear systems as a whole, rather than focusing on each system individually.
- We developed a new GPU-capable hybrid direct-iterative algorithm to solve KKT linear systems which outperforms LBL^T by a factor of 10 on the largest matrices tested.
- We proved important convergence theorems regarding the algorithm.
- We developed software implementing the algorithm and it is freely available online.

3.3 Introduction

Interior methods for nonlinear optimization [31, 81, 166, 170] are essential in many areas of science and engineering. They are commonly used in model predictive control with applications to robotics [155], autonomous cars [171], aerial vehicles [98], combustion engines [101], and heating, ventilation and air-conditioning systems [113], to name a few. Interior methods are also used in public health policy strategy development [9, 154], data fitting in physics [140], genome science [13, 167], and many other areas.

Most of the computational cost within interior methods is in solving linear systems of Karush-Kuhn-Tucker (KKT) type [120]. The linear systems are sparse, symmetric indefinite, and usually ill-conditioned and difficult to solve. Furthermore, implementations of interior methods for nonlinear optimization, such as the filter-line-search approach in Ipopt [170] and HiOp [126], typically expect the linear solver to provide the matrix inertia (number of positive, negative and zero eigenvalues) to determine if the system should be regularized. (Otherwise, interior methods perform curvature tests to ensure descent in a certain merit function [39].) Relatively few linear solvers are equipped to solve KKT systems, and even fewer to run those computations on hardware accelerators such as graphic processing units (GPUs) [157].

At the time of writing, six out of the ten most powerful computers in the world have more than 90% of their compute power in hardware accelerators [8]. Hardware accelerator technologies are becoming ubiquitous in off-the-shelf products, as well. In order to take advantage of these emerging technologies, it is necessary to develop fine-grain parallelization techniques tailored for high-throughput devices such as GPUs.

For the sparse systems of the interest of this manuscript, pivoting becomes extremely expensive, as data management takes a large fraction of the total time compared to computation [60]. Unfortunately, LBL^T factorization is unstable without pivoting. This is why LBL^T approaches, typically used by interior methods for nonlinear problems on CPU-based platforms [158], have not performed as well on hardware accelerators [157]. Some iterative methods such as MINRES and

SYMMLQ [122] for general symmetric matrices can make efficient (albeit memory bandwidth limited) use of GPUs because they only require matrix-vector multiplications at each iteration, which can be highly optimized [18], but they are not efficient when the number of iterations to converged solution becomes large. Another approach for better-conditioned KKT systems is using a modified version of the preconditioned conjugate gradient (**PCG**) with implicit-factorization preconditioning [56]. The ill-conditioned nature of our linear systems means that iterative methods alone are not practical [122, 129].

We propose a hybrid direct-iterative method for solving KKT systems that is suitable for execution on hardware accelerators. The method only requires direct solves using a Cholesky factorization, as opposed to LBL^T, which means it avoids pivoting. We provide preliminary test results that show the practicality of our approach. Our test cases are generated by optimal power flow analysis [34, 108, 118] applied to realistic power grid models that resemble actual grids, but do not contain any proprietary data [114]. These systems are extracted from optimal power flow analysis using Ipopt [170] with MA57 as its linear solver. Solving such sequences of linear problems gives us an insight in how our linear solver behaves within an interior method. Using these test cases allowed us to assess the practicality of our hybrid approach without interfacing the linear solver with an optimization solver. We want to point out that our method is not specific to power grids. Rather, power grids are representative of very sparse and irregular systems commonly found in engineering disciplines[133].

The chapter is organized as follows. Table 3.1 defines our notation. Section 3.4 describes the optimization problem being solved. Section 3.5 defines the linear systems that arise when an interior method is applied to the optimization problem. In Section 3.6, we derive HyKKT, a novel hybrid direct-iterative algorithm that utilizes the block structure of the KKT linear systems, and prove convergence properties for the algorithm. Numerical tests in Section 3.7 show the accuracy of HyKKT on realistic systems, using a range of algorithmic parameter values. Section 3.8 compares our C++ and CUDA implementation to MA57 [58]. Section 3.9 explains our decision to use a direct solver in the inner loop of our algorithm. In Section 3.10, we summarize our main contributions and results. Section 3.11 provides supplemental figures for Section 3.7.

Table 3.1: Notation. SP(S)D stands for symmetric positive (semi)definite. Matrix/vector norms are 2-norms unless stated otherwise. For a symmetric matrix M, we define its smallest eigenvalue restricted to the null space of a Jacobian J: $\lambda_{\min}(M_{|\text{null}(J)}) \equiv \min_{\|v\|=1,\ v\in\text{null}(J)} v^T M v$.

Variable	Properties	Functions	Meaning
M	symmetric matrix	$\lambda_{\max}(M)$, $\lambda_{\min}(M)$	largest, smallest (most negative) eigenvalues
M	SPSD matrix	$\lambda_{\min *}(M)$	smallest nonzero eigenvalue
M	SPD matrix	$\kappa(M) = \lambda_{\max}(M)/\lambda_{\min}(M)$	condition number
J	rectangular matrix	$\text{null}(J)$	nullspace
M, J	symmetric, rectangular	$\lambda_{\min}(M_{|\text{null}(J)})$	smallest eigenvalue value restricted to null(J)
x	vector, $x > 0$	$X \equiv \text{diag}(x)$	diagonal matrix X with $X_{ii} = x_i$
e_p	vector		p-vector of 1s

3.4 Nonlinear optimization problem

We consider constrained nonlinear optimization problems of the form

$$\min_{x\in\mathbb{R}^{n_x}} \quad f(x) \tag{3.1a}$$

$$\text{s.t.} \quad c(x) = 0, \tag{3.1b}$$

$$d(x) \geq 0, \tag{3.1c}$$

$$x \geq 0, \tag{3.1d}$$

where x is an n_x-vector of optimization parameters, $f: \mathbb{R}^{n_x} \to \mathbb{R}$ is a possibly nonconvex objective function, $c: \mathbb{R}^{n_x} \to \mathbb{R}^{m_c}$ defines m_c equality constraints, and $d: \mathbb{R}^{n_x} \to \mathbb{R}^{m_d}$ defines m_d inequality constraints. (Problems with more general inequalities can be treated in the same way.) Functions $f(x)$, $c(x)$ and $d(x)$ are assumed to be twice continuously differentiable. Interior methods enforce bound constraints (3.1d) by adding barrier functions to the objective (3.1a):

$$\min_{x\in\mathbb{R}^{n_x},\, s\in\mathbb{R}^{m_d}} \quad f(x) - \mu \sum_{j=1}^{n_x} \ln x_j - \mu \sum_{i=1}^{m_d} \ln s_i,$$

where the inequality constraints (3.1c) are treated as equality constraints $d(x) - s = 0$ with slack variables $s \geq 0$. The barrier parameter $\mu > 0$ is reduced toward zero using a continuation method to obtain a solution x that is close to the solution of (3.1) to within a solver tolerance.

Interior methods are most effective when exact first and second derivatives are available, as we assume for $f(x)$, $c(x)$, and $d(x)$. We define $J_c(x) = \nabla c(x)$ and $J_d(x) = \nabla d(x)$ as the sparse Jacobians for the constraints. The solution of a barrier subproblem satisfies the nonlinear equations

$$\nabla f(x) + J_c^T y_c + J_d^T y_d - z_x = 0 \tag{3.2a}$$
$$-y_d - z_s = 0 \tag{3.2b}$$
$$c(x) = 0 \tag{3.2c}$$
$$d(x) - s = 0 \tag{3.2d}$$
$$X z_x = \mu e_{n_x} \tag{3.2e}$$
$$S_s z_s = \mu e_{m_d}, \tag{3.2f}$$

where x and s are primal variables, y_c and y_d are Lagrange multipliers (dual variables) for constraints (3.2c)–(3.2d), and z_x and z_s are Lagrange multipliers for the bounds $x \geq 0$ and $s \geq 0$. The conditions $x > 0$, $s > 0$, $z_x > 0$, and $z_s > 0$ are maintained throughout, and the matrices $X \equiv \operatorname{diag}(x)$ and $S_s \equiv \operatorname{diag}(s)$ are SPD.

Analogously to [170], at each continuation step in μ we solve nonlinear equations (3.2) using a variant of Newton's method. Typically z_x and z_s are eliminated from the linearized version of (3.2) by substituting the linearized versions of (3.2e) and (3.2f) into the linearized versions of (3.2a) and (3.2b), respectively, to obtain a smaller *symmetric* problem. Newton's method then calls the linear solver to solve a series of linearized systems $K_k \Delta x_k = r_k$, $k = 1, 2, \ldots$, of block 4×4 form

$$\overbrace{\begin{bmatrix} H + D_x & 0 & J_c^T & J_d^T \\ 0 & D_s & 0 & -I \\ J_c & 0 & 0 & 0 \\ J_d & -I & 0 & 0 \end{bmatrix}}^{K_k} \overbrace{\begin{bmatrix} \Delta x \\ \Delta s \\ \Delta y_c \\ \Delta y_d \end{bmatrix}}^{\Delta x_k} = \overbrace{\begin{bmatrix} \tilde{r}_x \\ r_s \\ r_c \\ r_d \end{bmatrix}}^{r_k}, \tag{3.3}$$

where index k denotes optimization solver iteration (including continuation step in μ and Newton iterations), each K_k is a KKT-type matrix (with saddle-point structure), vector Δx_k is a search direction[1] for the primal and dual variables, and r_k is derived from the residual vector for (3.2) evaluated at the current value of the primal and dual variables (with $\|r_k\| \to 0$ as the method converges):

$$\tilde{r}_x = -(\nabla f(x) + J_c^T y_c + J_d^T y_d - \mu X^{-1} e_{n_x}),$$
$$r_s = y_d + \mu S_s^{-1} e_{m_d}, \quad r_c = -c(x), \quad r_d = s - d(x).$$

[1]Search directions are defined such that $x_{k+1} = x_k + \alpha \Delta x$ for some linesearch steplength $\alpha > 0$.

With $Z_x \equiv \text{diag}(z_x)$ and $Z_s \equiv \text{diag}(z_s)$, the sparse Hessian

$$H \equiv \nabla^2 f(x) + \sum_{i=1}^{m_c} y_{c,i} \nabla^2 c_i(x) + \sum_{i=1}^{m_d} y_{d,i} \nabla^2 d_i(x),$$

$D_x \equiv X^{-1} Z_x$ is a diagonal $n_x \times n_x$ matrix, $D_s \equiv S_s^{-1} Z_s$ is a diagonal $m_d \times m_d$ matrix, J_c is a sparse $m_c \times n_x$ matrix, and J_d is a sparse $m_d \times n_x$ matrix. We define $m \equiv m_c + m_d$, $n \equiv n_x + m_d$, and $N \equiv m + n$.

Interior methods may take hundreds of iterations (but typically not thousands) before they converge to a solution. All K_k matrices have the same sparsity pattern, and their nonzero entries change slowly with k. An interior method can exploit this by reusing output from linear solver functions across multiple iterations k:

- Ordering and symbolic factorization are needed only once because the sparsity pattern is the same for all K_k.
- Numerical factorizations can be reused over several adjacent Newton's iterations, e.g., when an inexact Newton solver is used within the optimization algorithm.

Operations such as triangular solves have to be executed at each iteration k.

The workflow of the optimization solver with calls to different linear solver functions is shown in Fig. 3.1 (where $K_k \Delta x_k = r_k$ denotes the linear system to be solved at each iteration). The main *optimization* solver loop is the top feedback loop in Fig. 3.1. It is repeated until the solution is optimal or a limit on optimization iterations is reached. At each iteration, the residual vector r_k is updated. Advanced implementations have control features to ensure stability and convergence of the optimization solver. The lower feedback loop in Fig. 3.1 shows linear system regularization by adding a diagonal perturbation to the KKT matrix. One such perturbation removes singularity [170, Sec. 3.1], which happens if there are redundant constraints. The linear solver could take advantage of algorithm control like this and request matrix perturbation when beneficial.

3.5 Solving KKT linear systems

LBL^T factorization via MA57 [58] has been used effectively for sparse symmetric systems on traditional CPU-based platforms. Parallel and GPU-accelerated direct solvers such as SuperLU [7, 111], STRUMPACK [144, 156], and PaStiX [91, 124] exist for general symmetric indefinite systems (although the first two are designed for general systems). However, these software packages on GPU are designed to take advantage of dense blocks of the matrices in denser problems and do not perform well on our systems of interest, which do not yield these dense blocks [89, 157].

The fundamental issue with using GPUs for LBL^T is that this factorization is not stable without pivoting [80]. Pivoting requires considerable data movement and, as a result, a substantial part of the

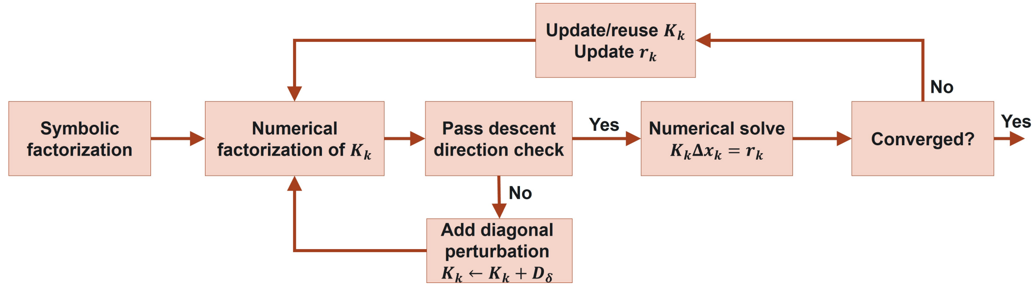


Figure 3.1: Optimization solver workflow showing invocation of key linear solver functions. The top feedback loop represents the main optimization solver iteration loop. The bottom feedback loop is the optimization solver control mechanism to regularize the underlying problem when necessary.

run time is devoted to memory access and communication. Any gains from the hardware acceleration of floating-point operations are usually outweighed by the overhead associated with permuting the system matrix during the pivoting. This is especially burdensome because both rows and columns need to be permuted in order to preserve symmetry [60]. While any of the two permutations can be performed efficiently on its own with an appropriate data structure for the system's sparse matrix (e.g., compressed sparse row storage for row permutations and compressed sparse column storage for column permutations), swapping rows and columns simultaneously is necessarily costly.

Here we propose HyKKT, a method that uses sparse Cholesky factorization on the (1,1) block (suitably modified) and an iterative solver on the Schur complement. Cholesky factorization is advantageous because it is stable without pivoting and can use GPUs efficiently compared to LBL^T [141]. In addition, the symmetric ordering to retain sparsity in the factors (also known as symbolic factorization) can be established at the beginning of the optimization without considering numerical values, and hence its cost is amortized over the optimization iterations.

To make the problem smaller, we eliminate $\Delta s = J_d \Delta x - r_d$ and $\Delta y_d = D_s \Delta s - r_s$ from (3.3) to obtain the 2×2 system [125, Sec. 3.1]

$$\begin{bmatrix} \widetilde{H} & J_c^T \\ J_c & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y_c \end{bmatrix} = \begin{bmatrix} r_x \\ r_c \end{bmatrix}, \quad \widetilde{H} \equiv H + D_x + J_d^T D_s J_d, \tag{3.4}$$

where $r_x = \tilde{r}_x + J_d^T(D_s r_d + r_s)$. This reduction requires block-wise Gaussian elimination with block pivot $\begin{bmatrix} D_s & -I \\ -I & \end{bmatrix}$, which is ill-conditioned when D_s has large elements, as it ultimately does. Thus, system (3.4) is smaller but more ill-conditioned. After solving (3.4), we compute Δs and Δy_d in turn to obtain the solution of (3.3).

3.6 A block 2×2 system solution method

Let Q be any SPD matrix. Multiplying the second row of (3.4) by $J_c^T Q$ and adding it to the first row gives a system of the form

$$\begin{bmatrix} H_\gamma & J_c^T \\ J_c & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y_c \end{bmatrix} = \begin{bmatrix} \hat{r}_x \\ r_c \end{bmatrix}, \qquad H_\gamma = \widetilde{H} + J_c^T Q J, \tag{3.5}$$

where $\hat{r}_x = r_x + J_c^T Q r_c$. The simplest choice is $Q = \gamma I$ with $\gamma > 0$:

$$H_\gamma = \widetilde{H} + \gamma\, J_c^T J_c. \tag{3.6}$$

When H_γ is SPD, its Schur complement $S \equiv J_c H_\gamma^{-1} J_c^T$ is well defined, and (3.5) is equivalent to

$$S \Delta y_c = J_c H_\gamma^{-1} \hat{r}_x - r_c, \tag{3.7}$$

$$H_\gamma \Delta x = \hat{r}_x - J_c^T \Delta y_c. \tag{3.8}$$

This is the approach of Golub and Greif [79] for saddle-point systems (which have the structure of the 2×2 system in (3.4)). Golub and Greif found experimentally that $\gamma = \|\widetilde{H}\|/\|J_c\|^2$ made H_γ SPD and better conditioned than smaller or larger values. We show in Theorems 5 and 7 that for large γ, the condition number $\kappa(H_\gamma)$ increases as $\gamma \to \infty$, but $\kappa(S)$ converges to 1 as $\gamma \to \infty$. Corollary 1 shows there is a finite value of γ that minimizes $\kappa(H_\gamma)$. This value is probably close to $\gamma = \|\widetilde{H}\|/\|J_c\|^2$.

Our contribution is to combine the system reduction in [125] (from (3.3) to (3.4)) with the method of [79] for changing (3.4) to (3.5) to solve an optimization problem consisting of a series of block 4×4 systems using a GPU implementation of sparse Cholesky factorization applied to H_γ. A method for regularizing is suggested (though it has not been needed in practice), and practical choices for parameters are given based on scaled systems. Also, important convergence properties of the method are proven in Theorems 4–7.

If (3.5) or H_γ are poorly conditioned, the only viable option may be to ignore the block structure in (3.5) and solve (3.3) with an $\mathrm{LBL^T}$ factorization such as MA57 (likely without help from GPUs). This is the fail-safe option. Otherwise, we require H_γ to be SPD (for large enough γ) and use its Cholesky factorization to apply the conjugate gradient method (CG) [92] or MINRES to (3.7) with or without a preconditioner. We note that TriCG and TriMR [119] could be used on (3.5), but TriCG requires that the (2,2) block be made slightly negative definite, and such regularization is typically not otherwise needed in practice. If the R part of QR factors of J_c^T is not too dense, we could use

$$M \equiv (J_c J_c^T)^{-1} J_c H_\gamma J_c^T (J_c J_c^T)^{-1}$$

as a multiplicative preconditioner for CG on (3.7). This gives the exact solution if the RHS is orthogonal to null(J_c) or J_c is square, which is not possible in our case. In our experiments, solutions with the preconditioner M lagged slightly behind those without it, but both take $O(1)$ iterations. We proceed without preconditioning (3.7).

3.6.1 A hybrid solver with minimal regularization

Typically, (3.4) starts with an SPD $\widetilde{H}$ and full-rank J_c. As the optimization iterations progress, $\widetilde{H}$ may become indefinite and J_c's rank may shrink (at least numerically). This means the system becomes singular and must be regularized. We seek a small regularization to avoid changing too much the solution of the equivalent system (3.5).

An SPD H_γ guarantees that $\widetilde{H}$ is SPD on null(J_c), a requirement at the solution of the optimization problem.

Theorem 4. *Assume J_c has full row rank. Then for large enough γ ($\gamma \geq \gamma_{\min}$ for some finite $\gamma_{\min}$), $H_\gamma = \widetilde{H} + \gamma J_c^T J_c$ is uniformly SPD (i.e., $\lambda_{\min}(H_\gamma) \geq \epsilon > 0$) if and only if $\widetilde{H}$ is positive definite on null(J_c) (i.e., $\lambda_{\min}(\widetilde{H}_{|null(J_c)}) > 0$).*

Proof. Assume $\widetilde{H}$ is positive definite on null(J_c). Consider a unit vector v; we decompose v as $v = v_0 + v_1$ with v_0 in null(J_c) and v_1 orthogonal to null(J_c). Consider

$$v^T H_\gamma v = v^T \widetilde{H} v + v_1^T (\gamma J_c^T J_c) v_1$$

When $v_1 = 0$ then $v^T H_\gamma v = v_0^T \widetilde{H} v_0 \geq \lambda_{\min}(\widetilde{H}_{|\mathrm{null}(J_c)})$. Since $v^T H_\gamma v$ is continuous with respect to v_1, there exists some α such that if $\|v_1\| \leq \alpha$ then

$$v^T H_\gamma v \geq \frac{1}{2} \lambda_{\min}(\widetilde{H}_{|\mathrm{null}(J_c)})$$

Assume now that $\|v_1\| > \alpha$. Then:

$$v^T H_\gamma v = v^T \widetilde{H} v + v_1^T (\gamma J_c^T J_c) v_1 \geq$$
$$\lambda_{\min}(\widetilde{H}) + \gamma \|v_1\|^2 \lambda_{\min *}(J_c^T J_c) > \lambda_{\min}(\widetilde{H}) + \gamma \alpha \lambda_{\min *}(J_c^T J_c)$$

Since $\alpha \lambda_{\min *}(J_c^T J_c) > 0$, the lower bound goes to $+\infty$ when $\gamma \to \infty$. So we can find $\gamma_{\min}$ such that if $\gamma \geq \gamma_{\min}$ then

$$v^T H_\gamma v \geq \frac{1}{2} \lambda_{\min}(\widetilde{H}_{|\mathrm{null}(J_c)})$$

In summary, in all cases, we guarantee that

$$v^T H_\gamma v \geq \frac{1}{2} \lambda_{\min}(\widetilde{H}_{|\mathrm{null}(J_c)})$$

if $\gamma \geq \gamma_{\min}$.

Conversely, assume H_γ is SPD. For any nonzero vector v_0 in null(J_c), we have $v_0^T \widetilde{H} v_0 = v_0^T H_\gamma v_0 > 0$. This direction does not require J_c to have full row rank. □

In practice one would use $\gamma \geq 0$. In our application we set $\gamma = 10^4$–10^6. However, γ cannot be made arbitrarily large without increasing $\kappa(H_\gamma)$ when J_c is rectangular, as in our case. There must be an ideal intermediate value of γ. We use H_γ SPD as a proxy for $\widetilde{H}$ being SPSD on null(J_c), keeping in mind that if it does not hold even for a very large γ in (3.6), H_γ is singular and needs to be modified.

Theorem 5. *If J_c has full row rank with more columns than rows, $\widetilde{H}$ is symmetric and positive definite on null(J_c), and $H_\gamma = \widetilde{H} + \gamma J_c^T J_c$, there exists $\gamma_{\max} \geq \max(\gamma_{\min}, 0)$ such that for $\gamma \geq \gamma_{\max}$, $\kappa(H_\gamma)$ increases linearly with γ.*

Proof.

$$\begin{aligned}
\lambda_{\max}(H_\gamma) \equiv \max_{\|v\|_2=1} v^T H_\gamma v &\leq \max_{\|v\|_2=1} v^T \widetilde{H} v + \max_{\|v\|_2=1} v^T(\gamma J_c^T J_c) v \\
&= \lambda_{\max}(\widetilde{H}) + \gamma \lambda_{\max}(J_c^T J_c), \\
\lambda_{\max}(H_\gamma) \geq \min_{\|v\|_2=1} v^T \widetilde{H} v &+ \max_{\|v\|_2=1} v^T(\gamma J_c^T J_c) v = \lambda_{\min}(\widetilde{H}) + \gamma \lambda_{\max}(J_c^T J_c).
\end{aligned}$$

Hence $\lambda_{\min}(\widetilde{H}) + \gamma\lambda_{\max}(J_c^T J_c) \leq \lambda_{\max}(H_\gamma) \leq \lambda_{\max}(\widetilde{H}) + \gamma\lambda_{\max}(J_c^T J_c)$, meaning $\lambda_{\max}(H_\gamma) \propto \gamma$ for large enough γ (defined as $\gamma \geq \gamma_{\max} \geq \max(\gamma_{\min}, 0)$). Similarly,

$$\lambda_{\min}(H_\gamma) \equiv \min_{\|v\|_2=1} v^T H_\gamma v \leq \max_{\|v\|_2=1} v^T \widetilde{H} v + \min_{\|v\|_2=1} v^T(\gamma J_c^T J_c) v.$$

Since J_c has a nontrivial null-space, there exists $v \neq 0$ such that $v^T(\gamma J_c^T J_c)v = 0$ and therefore if $\gamma \geq 0$ then

$$\min_{\|v\|_2=1} v^T(\gamma J_c^T J_c) v = 0.$$

Therefore

$$\lambda_{\min}(H_\gamma) \leq \max_{\|v\|_2=1} v^T \widetilde{H} v = \lambda_{\max}(\widetilde{H}).$$

Recall from Theorem 4 that if $\gamma \geq \gamma_{\min}$, $\lambda_{\min}(H_\gamma) \geq \epsilon > 0$. Thus $\epsilon \leq \lambda_{\min}(H_\gamma) \leq \lambda_{\max}(\widetilde{H})$ independent of γ. So $\kappa(H_\gamma) = \lambda_{\max}(H_\gamma)/\lambda_{\min}(H_\gamma) \propto \gamma$ for $\gamma \geq \gamma_{\max}$. □

Corollary 1. *Among γs such that H_γ is SPD ($\gamma \geq \gamma_{\min}$), $\kappa(H_\gamma)$ is minimized for some $\gamma \in [\gamma_{\min}, \gamma_{\max}]$.*

In practice, the optimizer may provide systems where $\widetilde{H}$ is not SPD on null(J_c). In this case we can regularize H_γ by using $H_\delta = H_\gamma + \delta_1 I$ instead. Unlike γ, the introduction of δ_1 changes

the solution of the system, so it is essential to keep δ_1 as small as possible. If H_γ is not SPD, we set $\delta_1 = \delta_{\min}$, a parameter for some minimum value of regularization. If H_δ is still not SPD, we double δ_1 until it is. This ensures we use the minimal value of regularization (to within a factor of 2) needed to make H_δ SPD.

If δ_1 proves to be large, which can happen before we are close to a solution, it is essential for the optimizer to be informed to allow it to modify the next linear system. When the optimizer nears a solution, δ_1 will not need to be large.

In our tests, δ_1 starts at 0 for the next matrix in the sequence, but is set back to its previous value if the factorization fails.

We also set $\delta_{\max}$, the maximum allowed δ_1 before we resort to $\mathrm{LBL^T}$ factorization of K_k or return to the optimization solver.

If J_c has low row rank, S in (3.7) is SPSD. In this case, CG will succeed if (3.7) is consistent. Otherwise, it will encounter a near-zero quadratic form. We then restart CG on the regularized system $(J_c H_\delta^{-1} J_c^T + \delta_2 I)\Delta y = J_c w - r_c$. In this way, δ_1 regularizes the $(1,1)$ block and δ_2 regularizes the $(2,2)$ block.

To ensure we can judge the size of parameters γ and $\delta_{\min}$ relative to system (3.5), we first scale (3.4) with a symmetrized version of Ruiz scaling [146].

Algorithm 7 is a generalization of the Uzawa iteration [164] for KKT systems with a (1,1) block that is not necessarily positive definite. It gives a method for solving a sequence of systems (3.7)–(3.8) with $Q = \gamma I$ used in the calculation of H_γ. The workflow is similar to Fig. 3.1 except only H_δ is factorized. On lines 16–19, H_δ^{-1} is applied by direct triangular solves with the Cholesky factors L, L^T of H_δ. Each iteration of the CG solve on line 17 requires multiplication by J_c^T, applying H_δ^{-1}, multiplying by J_c, and adding a scalar multiple of the original vector. Section 3.9 shows why complete Cholesky factorization of H_δ was chosen.

The rest of the section discusses other important bounds that γ must obey. However, selecting an ideal γ in practice is difficult and requires problem heuristics (like $\gamma = \|\widetilde{H}\|/\|J\|^2$ in [80]) or trial and error.

3.6.2 Guaranteed descent direction

Optimization solvers typically use an $\mathrm{LBL^T}$ factorization to solve the $N \times N$ system (3.3) at each step because (with minimal extra computation) it supplies the inertia of the matrix. A typical optimization approach treats each of the four 2×2 block of (3.3) as one block, accounting for the possible regularization applied to H_γ. We mean that the (1,1) block $H_K \equiv \begin{bmatrix} H + D_x + \delta_1 I & \\ & D_s \end{bmatrix}$ is a Hessian of dimension n, and the (2,1) block $J_K \equiv \begin{bmatrix} J_c & \\ J_d & -I \end{bmatrix}$ represents the constraint Jacobian and has dimensions $m \times n$. The inertia being $(n, m, 0)$ implies that (a) H_K is uniformly SPD on $\mathrm{null}(J_K)$ for all k, meaning $v^T H_K v \geq \epsilon > 0$ for all vectors v satisfying $J_K v = 0$, iterations k, and some

Algorithm 7 Using CG on Schur complement to solve the block system (3.5) by solving (3.7)–(3.8). H_δ is a nonsingular perturbation of $H_\gamma = \widetilde{H} + \gamma J_c^T J_c$. Typical parameters: $\gamma = 10^4$, $\delta_{\min} = \delta_2 = 10^{-9}$, $\delta_{\max} = 10^{-6}$.

1: **for** each matrix in the sequence such as in (3.5) **do**
2: $\quad\delta_1 \leftarrow 0$
3: $\quad H_\delta = H_\gamma$ (γ used in the calculation of H_γ)
4: $\quad$Try $LL^T =$ `chol`(H_δ) (Fail ← False if factorized, True otherwise)
5: $\quad$**while** Fail and $\delta_1 <= \delta_{\max}/2$ **do**
6: $\quad\quad$**if** $\delta_1 == 0$ **then**
7: $\quad\quad\quad\delta_1 \leftarrow \delta_{\min}$
8: $\quad\quad$**else**
9: $\quad\quad\quad\delta_1 = \delta_{\min} \leftarrow 2\delta_{\min}$
10: $\quad\quad$**end if**
11: $\quad\quad H_\delta = H_\gamma + \delta_1 I$
12: $\quad\quad$Try $LL^T =$ `chol`(H_δ) (Fail ← False if factorized, True otherwise)
13: $\quad$**end while**
14: $\quad$**if** Fail==False **then**
15: $\quad\quad$Direct solve $H_\delta w = \hat{r}_x$
16: $\quad\quad$**if** CG on $(J_c H_\delta^{-1} J_c^T)\Delta y_c = J_c w - r_c$ produces a small quadratic form **then**
17: $\quad\quad\quad$CG solve $(J_c H_\delta^{-1} J_c^T + \delta_2 I)\Delta y_c = J_c w - r_c$ (perturbed (3.7))
18: $\quad\quad$**end if**
19: $\quad\quad$Direct solve $H_\delta \Delta x = \hat{r}_x - J_c^T \Delta y_c$ (perturbed (3.8))
20: $\quad$**else**
21: $\quad\quad$Use LBL$^\text{T}$ to solve (3.3) or return problem to optimization solver
22: $\quad$**end if**
23: **end for**

positive constant ϵ; (b) KKT matrix in (3.3) is nonsingular. Together these conditions are sufficient to ensure a descent direction and fast convergence in the optimization algorithm [120, 170]. We show that Algorithm 7 ensures properties (a) and (b) without computing the inertia of the KKT matrix.

Theorem 6. *If Algorithm 7 succeeds with $\delta_2 = 0$, it provides a descent direction for the interior method for large enough γ.*

Proof. By construction, H_δ is uniformly SPD (because the Cholesky factorization was successful for all iterations of the solver until this point) and the KKT system in (3.5) is nonsingular (because J_c has full row rank, or $\delta_2 I$ was added to S to shift it from SPSD to SPD). Therefore, Algorithm 7 provides a descent direction for (3.5) even if regularization is added. The rest of the proof assumes $\delta_2 = 0$, and we return to the other case later. Since J_c has full row rank, $H + D_x + \delta_1 I$ is nonsingular by assumption, all block rows in (3.3) have full rank internally, and Gaussian elimination cannot cause cancellation of an entire block row, we conclude that (3.3) is nonsingular. Let $H_{K\gamma} \equiv H_K + \gamma_K J_K^T J_K$ (for $\gamma_K \geq \gamma$ used in Algorithm 7). For any nonzero vector $u^T = (u_1^T, u_2^T)$ with u_1 and u_2 of

dimensions n_x and m_d,

$$u^T H_{K\gamma} u = u_1^T (H + D_x + \gamma_K J_c^T J_c) u_1 + u_2^T D_s u_2 + \gamma_K w^T w,$$

where $w = J_d u_1 - u_2$. If $w = 0$, then $u_2 = J_d u_1$, which means $u^T H_{K\gamma} u \geq u_1^T H_\gamma u_1 \geq \epsilon > 0$ and the proof is complete (with $\gamma_K = \gamma$). Otherwise, $\gamma_w \equiv \gamma_K w^T w > 0$. So for large enough γ_K, $u^T H_{K\gamma} u \geq \epsilon > 0$. Applying Theorem 4 to (3.3) with H_K and J_K replacing $\widetilde{H}$ and J_c shows that H_K is positive definite on null(J_K). □

We note that δ_1 corresponds to the so-called primal regularization of the filter line-search algorithm [170]. Under this algorithm, whenever δ_1 becomes too large, one can invoke the feasibility restoration phase of the filter line-search algorithm [170] as an alternative to performing the LBL^{T} factorization on the CPU. Feasibility restoration "restarts" the optimization at a point with more favorable numerical properties. We also note that when δ_1 is sufficiently large, the curvature test used in [39] should be satisfied. Hence, inertia-free interior methods have the global convergence property without introduction of other regularization from the outer loop.

The δ_2 regularization is a numerical remedy for low-rank J_c, caused by redundant equality constraints. This regularization is similar to the so-called dual regularization used in Ipopt [170] and specifically addresses the issue of rank deficiency; however, there is no direct analogue from a δ_2 regularization in (3.5) to Ipopt's dual regularization in (3.3) and neither of the two heuristics guarantees a descent direction for (3.3). Given the similarity between the two heuristics, we believe that the δ_2 regularization can be effective within the filter line-search algorithm.

When Algorithm 7 is integrated with a nonlinear optimization solver such as [168] and [127], the while loop (Line 5) in Algorithm 7 can be removed and the computation of δ_1 and δ_2 can be decided by the optimization algorithm. The development of robust heuristics that allow Algorithm 7's δ_1 and δ_2 regularization within the filter line-search algorithm will be subject of future work.

3.6.3 Convergence for large γ

In Section 3.6.1 we showed that H_γ is SPD for large enough γ, and that γ should not be so large that the low-rank term $\gamma(J_c^T J_c)$ makes H_γ ill-conditioned. Here we show that in order to decrease the number of CG iterations, it is beneficial to increase γ beyond what is needed for H_γ to be SPD.

Theorem 7. *In exact arithmetic, for $\gamma \gg 1$, the eigenvalues of $S_\gamma \equiv \gamma S$ converge to 1 with an error term that decays as $1/\gamma$.*

Proof. By definition,

$$S_\gamma = \gamma J(\widetilde{H} + \gamma J_c^T J_c)^{-1} J_c^T = J_c \left(\frac{1}{\gamma} \widetilde{H} + J_c^T J_c \right)^{-1} J_c^T. \tag{3.9}$$

Table 3.2: Summary of the methods used for solving various equations in the chapter.

Equation	Method	Comments
(3.3)	MA57 (LBL$^{\text{T}}$)	Practical, but not for GPU
(3.3)	HyKKT	Solved via subsystems (3.7) and (3.8)
(3.7)	CG	S applied as an operator
(3.8)	Cholesky	Also used for inner solves in (3.7)

Since $\widetilde{H}$ is nonsingular and J_c has full row rank by assumption, the Searle identity $(A+BB^T)^{-1}B^T = A^{-1}B(I+B^TA^{-1}B)^{-1}$ [153] with $A \equiv \widetilde{H}/\gamma$ and $B \equiv J_c^T$ gives

$$S_\gamma = \gamma J_c\widetilde{H}^{-1}J_c^T(I+\gamma J_c\widetilde{H}^{-1}J_c^T)^{-1} = (I+C)^{-1}, \quad C \equiv \frac{1}{\gamma}\left(J_c\widetilde{H}^{-1}J_c^T\right)^{-1}.$$

For $\gamma \to \infty$, $|\lambda_i(C)| \ll 1\ \forall i$. The identity $(I+C)^{-1} = \lambda_{k=0}^{\infty}(-1)^kC^k$, which applies when $|\lambda_i(C)| < 1\ \forall i$, gives

$$S_\gamma = \sum_{k=0}^{\infty}(-1)^kC^k = I - C + O\left(\frac{1}{\gamma^2}\right). \tag{3.10}$$

□

A different proof of an equivalent result is given by Benzi and Liu [20, Lemma 3.1].

Corollary 2. *The eigenvalues of S are well clustered for $\gamma \gg 1$, and the iterative solve in Algorithm 7 line 16 converges quickly.*

In Section 3.7, we show that our choices of $\gamma = 10^4$–10^6 are large enough for the arguments to hold. This explains the rapid convergence of CG. We also show that our transformation from (3.3) to (3.5) and our scaling choice are stable on our test cases, by measuring the error for the original system (3.3).

3.6.4 Summary of equations solved

Table 3.2 details various equations solved and the methods used to solve them.

3.7 Practicality demonstration

We demonstrate the practicality of Algorithm 7 using five series of linear problems [114] generated by the Ipopt solver performing optimal power flow analysis on the power grid models summarized in Table 3.3. We compare our results with a direct solve of (3.3)via MA57's LBL$^{\text{T}}$ factorization [58].

Table 3.3: Characteristics of the five tested optimization problems, each generating sequences of linear systems $K_k \Delta x_k = r_k$ (3.3) of dimension N. Numbers are rounded to 3 digits. K and M signify 10^3 and 10^6.

Name	$N(K_k)$	$\mathrm{nnz}(K_k)$
South Carolina grid	56K	411K
Illinois grid	4.64K	21.6K
Texas grid	55.7K	268K
US Western Interconnection grid	238K	1.11M
US Eastern Interconnection grid	1.64M	7.67M

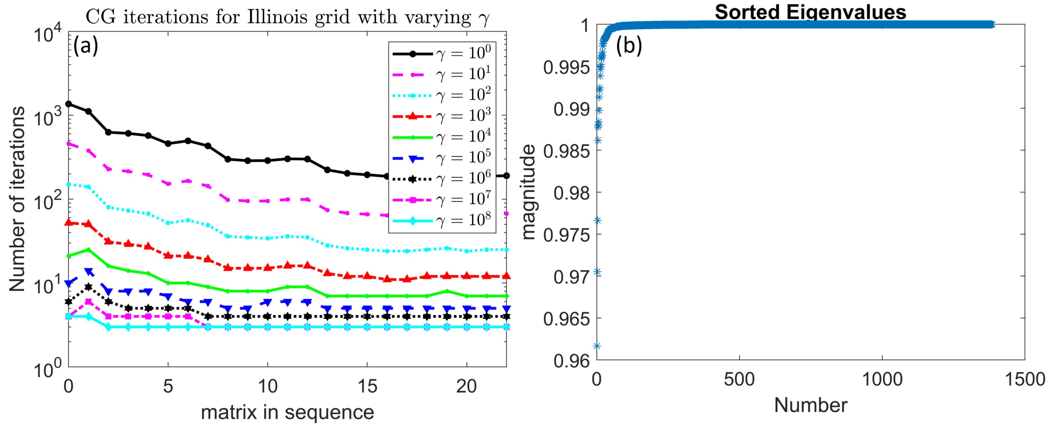


Figure 3.2: Illinois grid: (a) CG iterations on Eq. (3.7) with varying γ. $\gamma \geq 10^3$ gives good convergence. The mean number of iterations for $\gamma = 10^4$ is 9.4. (b) Sorted eigenvalues of $S_\gamma \equiv \gamma J_c H_\gamma^{-1} J_c^T$ in (3.9) matrix 22, for $\gamma = 10^4$. The eigenvalues are clustered close to 1.

3.7.1 γ selection

We use Algorithm 7 with CG as the iterative method on line 16. A larger γ in $H_\gamma = \widetilde{H} + \gamma J_c^T J_c$ may improve CG convergence but make H_γ more ill-conditioned and increase the error in the solution of (3.3). We therefore run some preliminary tests to find a suitable γ before selecting $\delta_{\min}$ and testing other problems, and we require CG to solve accurately (stopping tolerance 10^{-12} for the relative residual norm). We start with one of the smaller problems, the Illinois grid, to eliminate some values of γ and see if the condition number of K_k for each matrix in the sequence affects the needed iterations or regularization. We run these tests with $\delta_{\min} = 10^{-10}$.

Figure 3.2(a) shows that for the Illinois grid sequence, values of $\gamma \geq 10^4$ give CG convergence in approximately 10 iterations for every matrix in the sequence. For the last matrix we found that eigenvalues of S in (3.7) are very well clustered and the condition number of S is $\kappa \approx 1.04$, as shown in Fig. 3.2(b). This guarantees and explains the rapid convergence because for CG applied to a

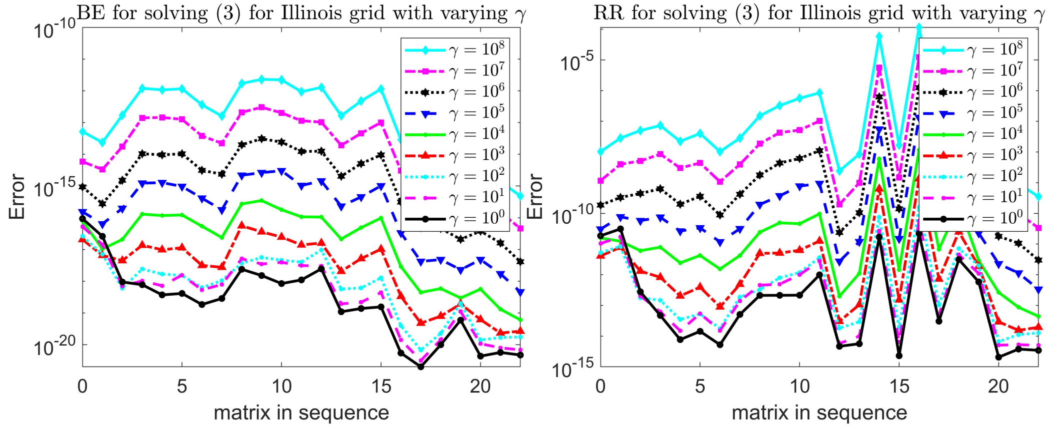


Figure 3.3: Illinois grid (3.3), with varying γ in (3.6): (a) Backward error (**BE**) and (b) relative residual (**RR**). (a) $\gamma \leq 10^4$ gives results close to machine precision. (b) $\gamma \leq 10^4$ has RR $\leq 10^{-8}$.

general SPD system $Ax = b$,

$$\frac{\|e_k\|_A}{\|e_0\|_A} \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k, \tag{3.11}$$

where $e_k = x - x_k$ is the error in an approximate solution x_k at iteration k [80].

For the last few matrices, $\gamma = 10^8$ is the only value requiring $\delta_1 > 0$. The final value of δ_1 was $16\delta_{\min} = 1.6 \cdot 10^{-9}$. No important information was gleaned from $\texttt{cond}(K_k)$. For all other values of γ, $\delta_1 = 0$ for the whole sequence. For a system $Ax = b$ and an approximate solution $\tilde{x} \approx x$, we define the backward error **BE** as $\|A\tilde{x} - b\|/(\|A\|\,\|\tilde{x}\| + \|b\|)$ and the relative residual **RR** as $\|A\tilde{x} - b\|/\|b\|$. As is common practice, we use $\|A\|_\infty$ to estimate $\|A\|_2$, which is too expensive to calculate directly. $\|A\|_\infty$ always provides an upper bound for $\|A\|_2$, but in practice is quite close to the actual value. Note that MA57 always has a BE of order machine precision. Figure 3.3 shows the (a) BE and (b) RR for system (3.3) for varying γ. Results for the BE and RR of system (3.4) are not qualitatively different and are given in Section 3.11. One conclusion is that increasing γ to reduce CG iterations can be costly for the accuracy of the solution of the full system. Based on the results of this section, γ in the range 10^2–10^6 gives reasonable CG iterations and final accuracy. For other matrices, we present a selected γ in this range that produced the best results.

3.7.2 Results for larger matrices

Solving larger (and perhaps more poorly conditioned) problems brings about new computational challenges and limits the amount of time any particular task can take. We wish to set $\delta_{\min}$ small enough to avoid over-regularizing the problem, and large enough to eliminate wasteful iterations and numerical issues. We want $\delta_{\max}$ small enough to recognize that we have over-regularized (and should

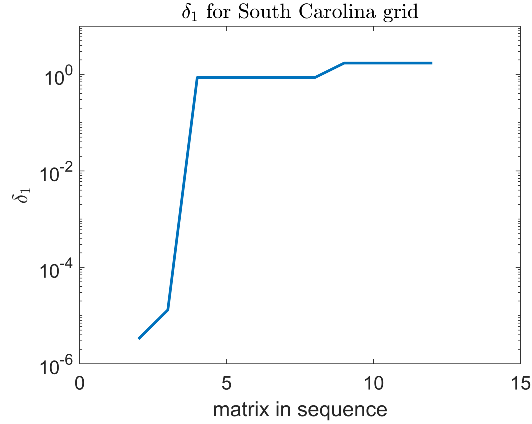


Figure 3.4: South Carolina grid: δ_1 for $\gamma = 10^4$. For other values of γ the graph was similar. Except for the first few and last few matrices, $\delta \approx 1$, meaning the required regularization would make the solution too inaccurate. The value of 0 is omitted on the log-scale.

try a different method), but large enough to allow for reasonable regularization. In our numerical tests, we use $\delta_{\min} = 10^{-10}$ and $\delta_{\max}$ large enough so that δ_1 can increase until $H_\delta = \widetilde{H} + \delta_1 I$ is SPD. This informs the parameter selection for the next system.

Figure 3.4 shows that the South Carolina grid matrices as currently constructed cannot benefit from this method. They need $\delta_1 > 1$ to make $H_\gamma + \delta_1 I$ SPD, which on a scaled problem means as much weight is given to regularization as to the actual problem. Algorithm 7 succeeds on the other matrix sequences, at least for certain γs, and needs no regularization ($\delta_1 = \delta_2 = 0$).

For the US Western Interconnection grid, Fig. 3.5(a) shows a CG convergence graph and (b) shows several types of error. For the US Eastern Interconnection grid, Fig. 3.6(a) shows a CG convergence graph and (b) shows several types of error. Figures for the Texas grid are given in Section 3.11, as they do not provide more qualitative insight. Convergence occurs for all matrix sequences in less than 20 iterations on average. The BE for (3.3) is consistently less than 10^{-8} and, with two exceptions in the US Western Interconnection grid, is close to machine precision. Results for the US Eastern Interconnection grid show that the method does not deteriorate with problem size, but rather there are some irregular matrices in the US Western Interconnection grid.

The results in this section suggest that $\delta_{\min}$ in the range 10^{-8} down to 10^{-10} is reasonable for any $\gamma \leq 10^8$. There is no clear choice for $\delta_{\max}$, but a plausible value would be $\delta_{\max} = 2^{10}\delta_{\min} \approx 1000\delta_{\min}$. This way we are guaranteed that the regularization doesn't take over the problem, and the number of failed factorizations is limited to 10, which should be negligible in the total solution times for a series of ≈ 100 problems.

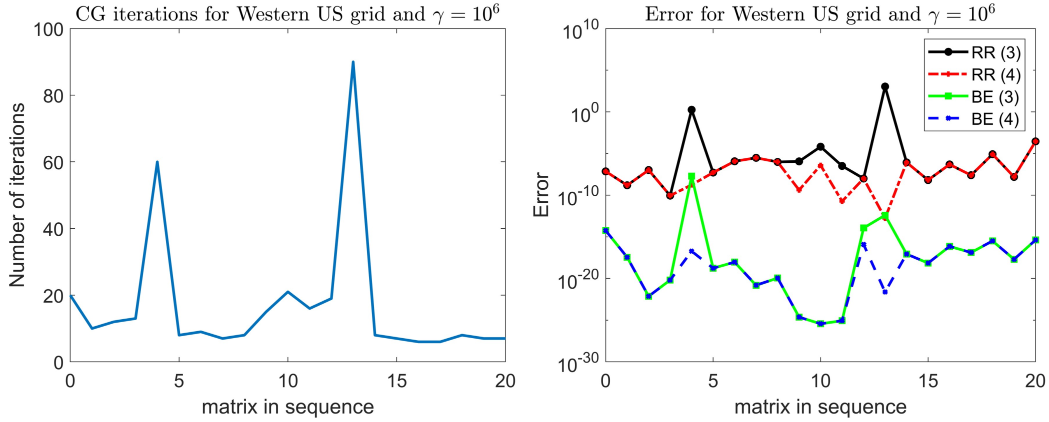


Figure 3.5: US Western Interconnection grid with $\gamma = 10^6$ in (3.6): (a) CG iterations on Eq. (3.7). The mean number of iterations is 17. (b) BE and RR for the sequence. The BE for (3.3) is less than 10^{-10}, except for matrix 4.

Table 3.4: Accelerator devices and compilers used.

Machine name	Host processor	Accelerator device	Host compiler	Device compiler
Newell	IBM Power9	NVIDIA Volta 100	GCC 7.4.0	CUDA 10.2
Deception	AMD EPYC 7502	NVIDIA Ampere 100	GCC 7.5.0	CUDA 11.1

3.7.3 Reordering H_γ

The efficiency of sparse Cholesky factorization $PH_\gamma P^T = LL^T$ depends greatly on the row/column ordering defined by permutation P. Figure 3.7 compares the sparsity of L, corresponding to H_γ of matrix 22 in the Illinois grid sequence, obtained from two choices of P: approximate minimum degree (**AMD**) and nested dissection. (The data in this section are generated via MATLAB.) We see that AMD produces a sparser L ($17,413$ nonzeros vs. $20,064$).

Reverse Cuthill-McKee and no ordering gave $46,527$ and $759,805$ nonzeros respectively. Recall that the sparsity structure is identical for matrices from the same family and similar for other matrix families. As expected, AMD was the sparsest ordering tested for other matrix families. Thus, AMD is our reordering of choice. Ordering is a one-time cost performed during the optimization problem setup.

3.8 Comparison with LBL^T

We compare HyKKT with a direct solve of (3.3) using MA57's LBL^T factorization [58] with default settings. All testing in this section is done using a prototype C++/CUDA code on a single GPU device. Reference MA57 solutions were computed on a CPU. Further details on computational platforms used are given in Table 3.4.

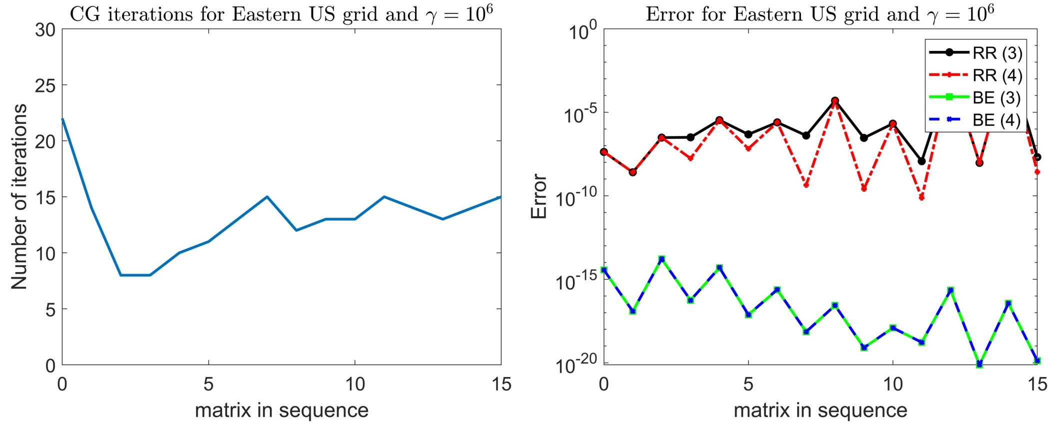


Figure 3.6: US Eastern Interconnection grid with $\gamma = 10^6$ in (3.6): (a) CG iterations on Eq. (3.7). The mean number of iterations is 13.1. (b) BE and RR for (3.3) and (3.4). The BE for (3.3) is less than 10^{-10}.

Table 3.5: Dimensions, number of nonzeros, and factorization densities (number of nonzeros in the factors per row) for solving (3.3) directly with $\mathrm{LBL}^{\mathrm{T}}$ (N, nnz_L, ρ_L respectively) and for solving (3.6) with Cholesky (n_x, nnz_C, ρ_C respectively). Numbers are rounded to 3 digits. K and M signify 10^3 and 10^6. In all cases, $\rho_C < \rho_L$ and $n_x < N/2$.

Abbreviation	N	nnz_L	ρ_L	n_x	nnz_C	ρ_C
Illinois	4.64K	94.7K	20.4	2.28K	34.9K	15.3
Texas	55.7K	2.95M	52.9	25.9K	645K	24.9
Western US	238K	10.7M	44.8	116K	2.23M	19.2
Eastern US	1.64M	85.4M	52.1	794K	17.7M	22.3

The factorization density is $\rho_L = (2\,\mathrm{nnz}(L) + \mathrm{nnz}(D))\,/N$. We define ρ_C analogously for the Cholesky factors of H_γ in (3.5), with $\mathrm{nnz}(D) = 0$ and dimension n_x. Note that ρ gives the average number of nonzeros in the factorization per row or column. Table 3.5 shows that the Cholesky factor of H_γ is usually less dense than the $\mathrm{LBL}^{\mathrm{T}}$ factor for (3.3), even though (3.3) is sparser than (3.5).

Table 3.6 shows the solve times on the Newell computing cluster [128]. The main trend is that as the problem size increases, GPUs using cuSolver [1] increasingly outperform equivalent computations on a CPU.

Supernodal Cholesky via Cholmod in Suitesparse [37] does not perform well on GPUs for these test cases, but performs better than cuSolver on CPUs. This matches literature showing that multifrontal or supernodal approaches are not suitable for very sparse and irregular systems, where the dense blocks become too small, leading to an unfavorable ratio of communication versus computation [25, 51, 89]. This issue is exacerbated when supernodal or multifrontal approaches are used for fine-grain parallelization on GPUs [157]. HyKKT becomes better when the ratio of $\mathrm{LBL}^{\mathrm{T}}$ to Cholesky factorization time grows, because factorization is the most costly part of linear solvers and HyKKT has more (but smaller and less costly) system solves.We note that Cholmod runs on

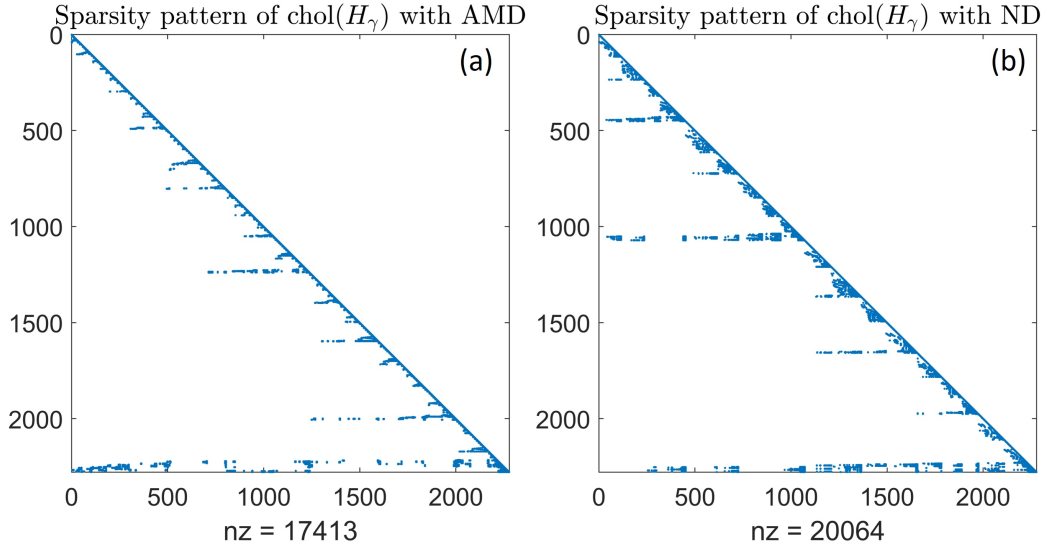


Figure 3.7: Illinois grid matrix 22: (a) Approximate minimum degree ordering of chol(H_γ) is sparser than (b) Nested dissection ordering of chol(H_γ). Both orderings are calculated in MATLAB.

the CPU and offloads computation to the GPU. In contrast, HyKKT can run completely on the GPU after one time (per optimization problem) costs of memory allocation, nonzero structure of matrix-matrix products computation, AMD, and symbolic analysis for Cholesky factorization.

Table 3.7 compares a direct solve of (3.3) using MA57's LBL^T factorization [58] and HyKKT, broken down into forming (3.7)–(3.8), symbolic analysis of H_δ, factorization of H_δ, CG on (3.7), and the subsequent full solution recovery on Deception [128]. The times for regularization (which was not needed in our case but timed nonetheless) were 3 orders of magnitude smaller than forming (3.7)–(3.8) and were therefore not included in the table.

When Cholesky is used, symbolic analysis is needed only for the first matrix in the sequence because pivoting is not a concern. As problems grow larger, the solve phase becomes a smaller part of the total run time. Also, HyKKT increasingly outperforms MA57. The run time is reduced by a factor of more than 3 on one matrix from the US Eastern Interconnection grid and more than 10 on the whole series, because the analysis cost can be amortized over the entire optimization problem. Furthermore, although only one GPU is used for all computations, we see that the computation costs increases sublinearly with the problem size. This occurs because the GPU is being under utilized and will naturally be limited at some point. This motivates using HyKKT for even larger problems.

Another advantage of HyKKT is that it solves systems that are less than half the size of the original one, though it does have to solve more of them. Notably, the LBL^T factorization may require pivoting during the factorization, whereas Cholesky does not. With MA57, all our test cases required substantial permutations even with a lax pivot tolerance of 0.01 (and a tolerance as large as 0.5 may be required to keep the factorization stable). This means that for our systems, LBL^T factorization

Table 3.6: Average times (in seconds) for solving (3.3) directly on a CPU with LBL^T (via MA57 [58]) or for solving one H_δ linear system with supernodal Cholesky via Cholmod (CM) in SuiteSparse [37], or Cholesky via cuSolver [1] (CS) using the routine `csrlsvchol`, each on a CPU and on a GPU. Cholesky on a GPU is quicker than LBL^T on a CPU by an increasingly large ratio. The GPU is not actually utilized for small problems with CM, because CM has internal logic that decides what to offload to the GPU, and there is a minimum problem size for the GPU to be utilized. All runs are on Newell [128].

Name	MA57	CM CPU	CM GPU	CS CPU	CS GPU
Illinois	$7.35 \cdot 10^{-3}$	$1.74 \cdot 10^{-3}$		$2.25 \cdot 10^{-3}$	$5.80 \cdot 10^{-3}$
Texas	$1.24 \cdot 10^{-1}$	$3.42 \cdot 10^{-2}$		$5.67 \cdot 10^{-2}$	$4.79 \cdot 10^{-2}$
Western US	$4.30 \cdot 10^{-1}$	$1.02 \cdot 10^{-1}$		$1.89 \cdot 10^{-1}$	$1.59 \cdot 10^{-1}$
Eastern US	$4.34 \cdot 10^{0}$	$1.08 \cdot 10^{0}$	$3.65 \cdot 10^{0}$	$2.52 \cdot 10^{0}$	$6.12 \cdot 10^{-1}$

requires considerable communication and presents a major barrier for GPU programming. On the other hand, the direct solve is generally more accurate by 2–3 orders of magnitude.

3.9 Iterative vs. direct solve with H_δ in Algorithm 7

We may ask if forming $H_\delta \equiv H + D_x + J_d^T D_s J_d + \gamma J_c^T J_c + \delta_1 I$ and its Cholesky factorization $H_\delta = LL^T$ is worthwhile when it could be avoided by iterative solves with H_δ. Systems (3.7)–(3.8) require two solves with H_δ and an iterative solve with $S = J_c H_\delta^{-1} J_c^T + \delta_2 I$, which includes "inner" solves with H_δ until CG converges. As we see in Section 3.7, between 6 and 92 solves with H_δ are needed in our cases (and possibly more in other cases). Further, the inner iterative solves with H_δ would degrade the accuracy compared to inner direct solves or would require an excessive number of iterations. Therefore, for iterative solves with H_δ to be viable, the direct solve would have to cause substantial densification of the problem (i.e., the Cholesky factor L would have to be very dense). Let $\text{nnz}_{\text{op}}(H_\delta) = \text{nnz}(\widetilde{H}) + 2\,\text{nnz}(J) + n_x$ be the number of multiplications when H_δ is applied as an operator, and $\text{nnz}_{\text{fac}}(H_\delta) = 2\,\text{nnz}(L)$ be the number of multiplications for solving systems with H_δ. These values (generated in MATLAB) and their ratio are given in Table 3.8. The ratio is always small and does not grow with problem size, meaning L remains very sparse and the factorization is efficient. As the factorization dominates the total time of a direct solve with multiple right-hand sides, this suggests that performing multiple inner iterative solves is not worthwhile.

3.10 Summary

Following the approach of Golub and Greif [79], we have developed a novel direct-iterative method for solving saddle point systems, and shown that it scales better with problem size than LBL^T on systems arising from optimal power flow [114]. The method is tailored for execution on hardware accelerators where pivoting is difficult to implement and degrades solver performance dramatically.

Table 3.7: Average times (in seconds) for solving sequences of systems (3.3) directly on a CPU with LBL^T (via MA57 [58]) or on a GPU using our hybrid method. The latter is split into forming (3.7)–(3.8), analysis and factorization phases, and multiple solves. (The routines `cusolverSpCreateCsrcholInfo`, `cusolverSpXcsrcholAnalysis`, `cusolverSpDcsrcholFactor`, `cusolverSpDcsrcholZeroPivot`, and `cusolverSpDcsrcholSolve` are used to allow for the easy solution of systems with multiple RHS.) "Forming (3.7)–(3.8)" shows the times needed for the matrix products, not including the memory allocation phase (which is the most expensive). However, since the nonzero structure is constant in a given problem, this need only be done once. Symbolic analysis is needed only once for the whole sequence. Factorization happens once for each matrix. The solve phase is the total time for Lines 15-17 in Algorithm 7 with a CG tolerance of 10^{-12} on Line 16 plus the recovery of the solution to the original problem. The results show that HyKKT, without optimization of the code and kernels, outperforms LBL^T on the largest series (US Eastern Interconnection grid) by a factor of more than 3 on a single matrix, and more than 10 on a whole series, because the cost of symbolic analysis can be amortized over the series. All runs are on Deception [128].

Name	MA57	Hybrid Direct-Iterative Solver			
		Forming (3.7),(3.8)	Analysis	Factorization	Total solves
Illinois	$6.24 \cdot 10^{-3}$	$1.07 \cdot 10^{-3}$	$4.32 \cdot 10^{-3}$	$3.67 \cdot 10^{-4}$	$8.70 \cdot 10^{-3}$
Texas	$1.00 \cdot 10^{-1}$	$1.52 \cdot 10^{-3}$	$3.29 \cdot 10^{-2}$	$8.53 \cdot 10^{-4}$	$1.02 \cdot 10^{-1}$
Western US	$3.38 \cdot 10^{-1}$	$1.98 \cdot 10^{-3}$	$1.76 \cdot 10^{-1}$	$7.76 \cdot 10^{-4}$	$1.43 \cdot 10^{-1}$
Eastern US	$3.48 \cdot 10^{0}$	$5.58 \cdot 10^{-3}$	$7.79 \cdot 10^{-1}$	$8.83 \cdot 10^{-4}$	$3.25 \cdot 10^{-1}$

To solve KKT systems of the form (3.3), Algorithm 7 presents a method with an outer iterative solve and inner direct solve. The method assumes $H_\gamma = H + D_x + J_d^T D_s J_d + \gamma J_c^T J_c$ is SPD (or almost) for some $\gamma \geq 0$, and if necessary, uses the minimal amount of regularization $\delta_1 \geq 0$ (to within a factor of 2) to ensure $H_\delta = H_\gamma + \delta_1 I$ is SPD. We proved that as γ grows large, the condition number of H_γ grows linearly with γ, and the eigenvalues of the iteration matrix S converge to $1/\gamma$ ((3.10)). These results provide some heuristics for choosing γ, and explain why CG on the Schur complement system (3.7) converges rapidly.

On several sequences of systems arising from applying an interior method to OPF problems, the number of CG iterations for solving (3.7) was less than 20 iterations on average, even though no preconditioning was used.

Four of the five matrix series were solved with $\delta_1 = \delta_2 = 0$, and the BE for the original system (3.3) was always less than 10^{-8}. The efficiency gained by using a Cholesky factorization (instead of LBL^T) and avoiding pivoting is demonstrated in Table 3.5. Even though H_γ in (3.5) is denser than K_k in (3.3), its factors are sparser. Table 3.7 shows that HyKKT, when it succeeds, has better scalability than LBL^T and is able to utilize GPUs. This is the most substantial result of our chapter. For the fifth series (smaller than 2 of the others) $\delta_2 = 0$ worked, but δ_1 had to be of order 1 and no accurate solution could be obtained. The development of robust heuristics to select δ_1 and δ_2, and to integrate with the filter line-search algorithm, will be subject of future work.

The fact that the Cholesky factors are scarcely denser than the original matrix suggests that

Table 3.8: Densification of the problem for cases where the direct-iterative method is viable. Numbers are rounded to 3 digits. K and M signify 10^3 and 10^6. $\mathrm{nnz_{op}}(H_\delta) = \mathrm{nnz}(\widetilde{H}) + 2\,\mathrm{nnz}(J) + n_x$ is the number of multiplications when H_δ is applied as an operator, and $\mathrm{nnz_{fac}}(H_\delta) = 2\,\mathrm{nnz}(L)$ is the number of multiplications for solving systems with H_δ. The ratio $\mathrm{nnz_{fac}}(H_\delta)/\mathrm{nnz_{op}}(H_\delta)$ is only about 2 in all cases.

Name	$\mathrm{nnz_{op}}(H_\delta)$	$\mathrm{nnz_{fac}}(H_\delta)$	ratio
Illinois	20.5K	34.9K	1.70
Texas	249K	646K	2.59
Western US	1.05M	2.23M	2.11
Eastern US	7.23M	17.7M	2.45

not much could be gained by using nullspace methods [145] for the four sequences we were able to solve, as those require sparse LU or QR factorization of J_c^T, which is typically less efficient than sparse Cholesky factorization of H_δ. For the fifth sequence, and sequences similar to it, an efficient nullspace method may be better than the current fail-safe $\mathrm{LBL^T}$ factorization of the 4×4 system (3.3).

We limit the scope of the chapter to the development and implementation of the linear solver and do not include integrating it into an optimization solver such as HiOp. This requires features such as dealing with a rank-deficient J_c via regularization, and balancing the internal linear solver regularization with that of the optimization solver. We note that the linear systems being solved in iterations after the first one depend on the linear solver used in previous iterations and that this could impact the convergence of the optimization solver. Additionally, developing a better method of parameter selection, especially for γ, is essential for reaching HyKKT's full potential.

Acknowledgements

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations (the Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem—including software, applications, hardware, advanced system engineering, and early testbed platforms—to support the nation's exascale computing imperative.

We thank Research Computing at Pacific Northwest National Laboratory (PNNL) for computing support. We are also grateful to Christopher Oehmen and Lori Ross O'Neil of PNNL for critical reading of the manuscript and for providing helpful feedback. Finally, we would like to express our gratitude to Stephen Thomas of National Renewable Energy Laboratory for initiating discussion that motivated Section 3.9.

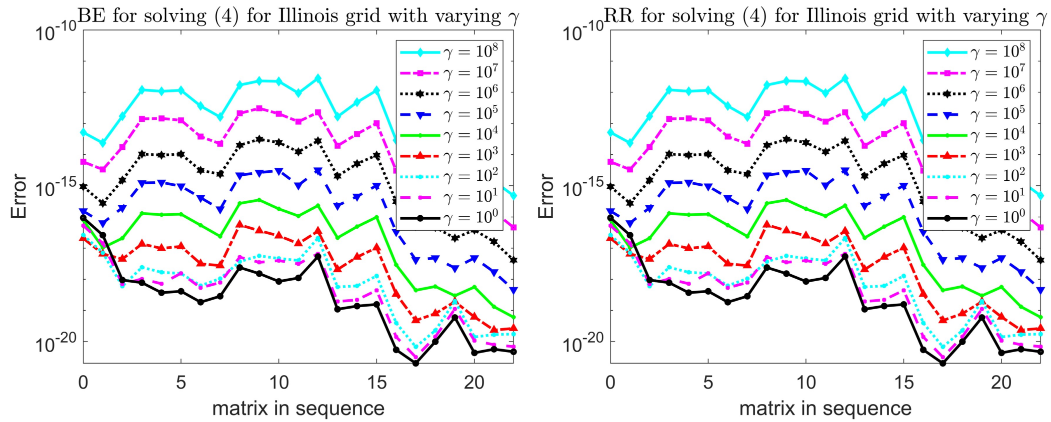


Figure 3.8: Illinois grid (3.4) with varying γ in (3.6): (a) BE, $\gamma \leq 10^4$ gives results close to machine precision. (b) RR, $\gamma \leq 10^4$ has RR $\leq 10^{-8}$.

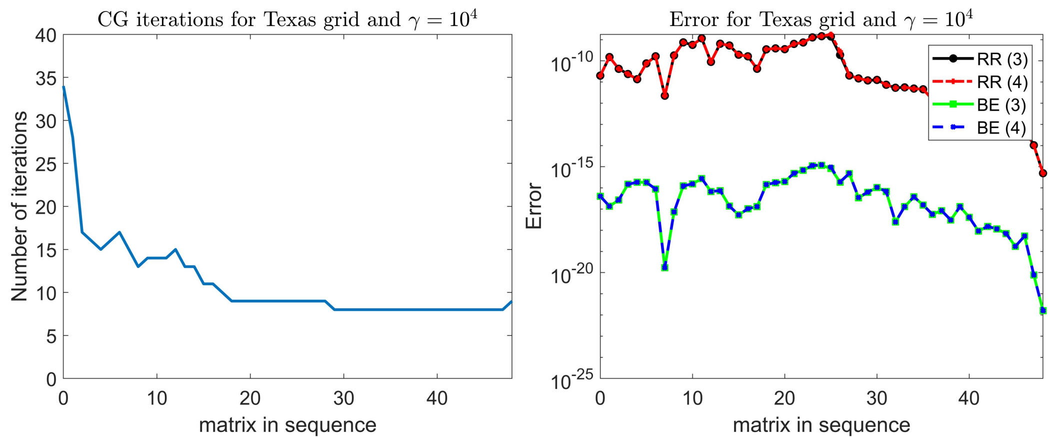


Figure 3.9: Texas grid with $\gamma = 10^4$: (a) CG iterations on Eq. (3.7). The mean number of iterations is 11.1. (b) BE and RR for (3.3) and (3.4). The BEs are roughly machine precision and the RRs are less than 10^{-8}.

3.11 Appendix A: Additional OPF matrix results

Figure 3.8 shows BE and RR in (3.4) for varying γ for the Illinois grid. For $1 \leq \gamma \leq 10^4$ the results are accurate.

CG convergence for the Texas grid with $\gamma = 10^4$ is given in Fig. 3.9(a), while (b) shows that the solution is very accurate and all errors are smaller than 10^{-8}.

3.12 Appendix B: HyKKT implementation

Figure 3.10 details the steps of the HyKKT algorithm. This workflow assumes that γ is selected once and kept constant for different problems because choosing the ideal γ is difficult and requires access to the eigenvalues which are prohibitive to compute. To deal with this, the regularization parameters δ_1 and δ_2 are used as described in Algorithm 7. The code will become open-sourced in 2023. Several custom classes and function files were created to allow for the full functionality of HyKKT. The code was developed with the guiding principles of object-oriented programming, removing details unnecessary for the user to be familiar with from their control (simplifying the CUDA API, which sometimes requires a long list of inputs), and modularizing the code so that no operations are repeated unnecessarily (such as anything related to the nonzero structure of the matrix). The following subsections give an overview of each file, for specific details about given functions see [provide open source link when it's published].

3.12.1 Classes

CholeskyClass

This class computes and stores the symbolic (`cusolverSpXcsrcholAnalysis`) and numeric (`cusolverSpDcsrcholFactor`) factorization of H_γ (or H_δ when regularized). The factorization is then used to solve systems of the form $H_\gamma x = b$ such as the inner solve of the Schur Complement conjugate gradient solve or the system $H_\gamma \Delta x = \hat{r_x} - J_c^T \Delta y_c$ with (`cusolverSpDcsrcholSolve`). The numeric factorization is reused for all these systems, such that only a triangular solve is performed each time a system is solved. The symbolic factorization is reused between iterations of the optimization solver, since the sparsity structure is maintained. In this fashion, no operation is performed more times than it need be. All actions are performed on the device.

HyKKTSolver

This class contains all the other classes and the relevant data to use HyKKT to solve Eq. (3.3). Additionally, it contains methods to set up the linear systems being solved Eq. (3.7) and Eq. (3.8), to recover the solution of system Eq. (3.3) from the solution of Eq. (3.5). Allocation operations are

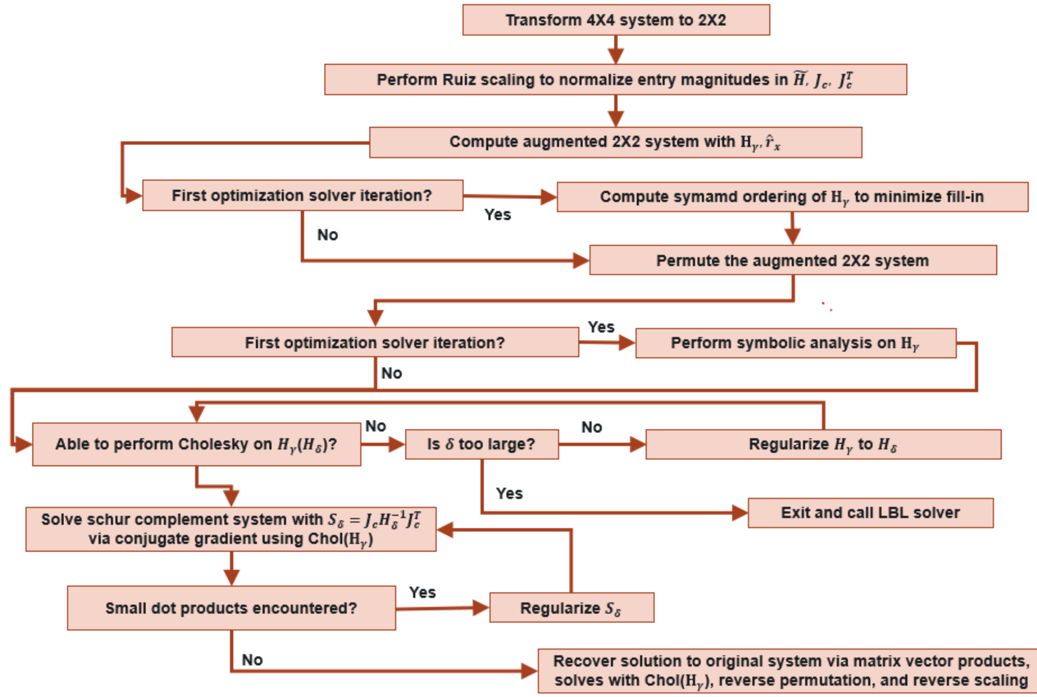


Figure 3.10: A summary of the workflow of the linear solver of HyKKT

performed during the first optimization solver iteration on the host. Subsequently, all computations are performed on the device including in the first iteration.

MMatrix

This structure can store a matrix in both COO and CSR format. It is useful to HyKKT because Matrix Market matrices are in COO format, so this is the format to read them into. If the matrix is symmetric, its upper triangular part is stored implicitly. This is all unpacked into the full explicit CSR format necessary for storing permuted matrices (permutation is done to reduce fill-in) and using them efficiently. The structure is on the host.

PermClass

This class computes and stores the permutation for the system Eq. (3.5). The permutation is calculated based on minimizing the fill-in of the Cholesky factorization of H_γ with symmetric approximate minimum degree. The equivalent permutations to the row and column arrays of the sparse blocks are calculated and the column permutation is stored to use on the value array. These actions are performed on the host and only during the first iteration of the optimization solver. At each iteration, only the value arrays of the block matrices are permuted, since the other arrays are unchanged due

to a constant sparsity pattern. This is performed on the device. Permutation is essential to reduce fill-in of the Cholesky factorization, which reduces the time and storage required for its computation.

RuizClass

This class performs Ruiz scaling of the system Eq. (3.5) on the device using the custom kernels defined in Section 3.12.3. This is done to ensure all entries in the matrix are $O(1)$ to aid in parameter selection portability between different problems. It also includes a method to get max_d, a variable that stores the applied scaling so that the solution to the scaled system can be scaled back to the solution of the original system.

SchurComplementConjugateGradient

This class calculates the solution to the system Eq. (3.7) via Chronopoulos-Gear CG [45]. This method is used because it allows for more parallelization and synchronization than standard CG [45]. S is applied by multiplying by J_c^T, solving a system with H_γ (using its precomputed Cholesky factorization), and finally multiplying by J_c. Operations are performed on the device.

SpgemmClass

This class allocates the memory necessary for matrix-matrix sums and products, which are used at various stages of the HyKKT algorithm. The sparsity structure is repeated between optimization solver iterations so the symbolic sum or product is computed only once and then reused. The numerical sums and products are computed at each iteration on the device using functions in matrix_matrix_ops.

3.12.2 Definitions

constants

Defines floating-point numerical constants 0, 1, and -1, which are useful in matrix and vector operations..

cusparse_params

Defines parameters used by cusparse in function calls such as data, index, and algorithm types.

3.12.3 Functions

matrix_vector_ops

Defines wrappers to CUDA functions for matrix-vector products, and adds kernels (GPU functions) used by RuizClass, for finding maximum row entries in a block matrix, and performing the appropriate Ruiz scaling. Additionally, it has custom kernels for scaling a matrix by a diagonal matrix, regularizing it, or multiplying by a constant. All calculations are performed on the device.

matrix_matrix_ops

Defines matrix-matrix functions for products and sums used by SpGEMM class. The functions are wrappers for `SpGEMM` and `cusparseDcsrgeam2` functions for matrix-matrix multiplies and sums respectively. The wrapper functions simplify away the CUDA API from the user and provide the same output with fewer inputs than the original function. Additionally, the memory allocation and symbolic sums and products are carried out only during the first optimization solver iteration (or first linear solve of Eq. (3.3)) and are used during subsequent iterations. It should be noted that SpGEMM in CUDA does have the ability to perform the operation $C = \alpha AB + \beta C$, but this requires C to be of the same sparsity structure as AB or else an all 0 matrix. This would effectively mean that the symbolic product would have to be computed and C appropriately padded before the function were called. We decided to use the `cusparseDcsrgeam2` functions to work around this. These operations are carried out on the device (GPU).

permcheck

Contains custom kernels implementing the methods of PermClass to permute block matrices (by row, column, or both simultaneously) in order to reduce fill-in during factorization and to limit storage usage and compute time. It also contains a custom kernel to map an existing permutation of a sparse matrix to a matrix of values. This is done so that the previous permutations can be used between optimization solver iterations and only the value array need be recalculated.

vector_vector_ops

Contains wrappers for methods to sum two vectors, take their dot product, or scale a vector on the device. All these operations are essential at various stages of the HyKKT solver.

3.12.4 Utilities

cuda_check_errors

Defines a wrapper function for CUDA calls to check they are successful before continuing with the program.

cuda_memory_utils

Contains wrapper functions to allocate, copy, or display matrices and vectors to supply the user with a more convenient API than CUDA.

cusparse_utils

Defines functions to create native cusparse handle, vector, and matrix objects and ones to transpose or display cusparseSpMatDescr_t matrices.

input_functions

Defines functions used to populate an MMatrix from a Matrix Market file. It can read a matrix stored in COO format in a Matrix Market file, and move a matrix stored in COO storage to CSR storage, with the option of filling in the implicitly stored symmetric entries if applicable. This is done because when reordering the unknowns, we can no longer count on the fact that the lower triangular part stores all the nonzero entries, unless they were all stored explicitly (in the upper triangular part) to begin with. This reading and populating is performed on the host.

Chapter 4

Summary

In this thesis we develop and implement two methods, SSAI and HyKKT, that combine elements of both iterative and direct solvers for linear equations. SSAI was left as a high-level language (Matlab) solver because low-level software development, including GPU capabilities, was prioritized on HyKKT due to the more promising nature of the preliminary results and the direct applicability to a problem at hand. However, as we detail in Section 2.8 and elsewhere in Chapter 2, there is no fundamental algorithmic difficulty with implementing SSAI in an embarrassingly parallel fashion.

The main takeaway of this thesis is the extent to which a solver designed specifically for a given application such as HyKKT can greatly outperform a solver that has to deal with different types of systems, such as MA57, and save both time and computational resources. The associated cost is in the development of the specialized algorithm and writing the corresponding code.

In the case of HyKKT, the development of the specialized software required knowledge of the specific block structure of the matrices arising from optimal power flow models, the high level of sparsity, and the fact that the sparsity structure is unchanged between iterations of the optimization solver. The last point is crucial because it is a paradigm in the solution of linear systems that when there are multiple systems $Ax = b$, if A changes then the systems are treated completely disjointly. HyKKT shifts away from this paradigm by exploiting the fact that the entirety of the memory allocation, reordering, and symbolic analysis can be done only once in the entire solution of an optimization problem, instead of $\approx$ 50–100 times, once for each iteration until convergence of the optimization solver. This brings substantial computational savings because these operations are the more costly ones and the other operations can be efficiently offloaded onto a GPU. It is possible that with more specialized kernels, larger problems, and better GPUs, HyKKT will outperform LBL^T by even larger margins. Furthermore, if for a given class of optimization problems the existence or non-existence of dependencies between two variables does not change, even if the nature of the dependency does, the more expensive parts of the HyKKT solver can be used over this entire class. In certain industrial applications where thousands of these problems are solved on a rolling basis,

the memory allocation and symbolic analysis time can effectively be washed out.

In contrast, a solver like SSAI is mainly useful if one is trying to solve only one problem at a time, or multiple problems without a clear through line between them. In these cases, the time and resources required for software development may far outweigh any gains made in computational time. Its strength is not necessarily on any given problem, but rather in that any (HPD) problem can be solved with it and usually quite competitively.

Future directions of research include developing a C++ version of SSAI with GPU and parallel computing capabilities. For HyKKT, the current version may be improved by creating more tailored kernels, expanding the code to allow the use of multiple CPUs and GPUs, and fully integrating it into an optimization solver in a way that cuts down on the extraneous memory allocation in the optimization software in addition to what's been done in the linear solver. Furthermore, both SSAI and HyKKT have parameters that might be set in a more efficient or accurate way than what we did here, which was experimentally coming up with one value that worked well for all problems.

Bibliography

[1] https://docs.nvidia.com/cuda/cusolver/index.html.

[2] https://www.hsl.rl.ac.uk/catalogue/ma57.html.

[3] https://www.hsl.rl.ac.uk/catalogue/ma57.html.

[4] https://intel.github.io/clGPU/docs/iclBLAS/html/.

[5] https://docs.nvidia.com/cuda/cublas/index.htmll.

[6] https://docs.nvidia.com/cuda/cusparse/index.html.

[7] https://portal.nersc.gov/project/sparse/superlu/.

[8] Top500 List, June 2021. https://www.top500.org/lists/top500/2021/06/.

[9] D. Acemoglu, V. Chernozhukov, I. Werning, and M. D. Whinston. Optimal targeted lockdowns in a multi-group sir model (working paper no. 27102). *National Bureau of Economic Research*, 10:w27102, 2020. Retrieved from http://www.nber.org/papers/w27102doi.

[10] Ajit Agrawal, Philip Klein, and R. Ravi. Cutting down on fill using nested dissection: Provably good elimination orderings. In Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, pages 31–55, New York, NY, 1993. Springer New York.

[11] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[12] T. Andrews. Computation time comparison between MATLAB and C++ using launch windows, 2012.

[13] Mirela Andronescu, Anne Condon, Holger H. Hoos, David H. Mathews, and Kevin P. Murphy. Computational approaches for RNA energy parameter estimation. *RNA*, 16(12):2304–2318, 2010.

[14] Cleve Ashcraft, Roger G. Grimes, and John G. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Anal. Appl.*, 20(2):513–561, 1998.

[15] H. Avron, A. Gupta, and S. Toledo. Solving Hermitian positive definite systems using indefinite incomplete factorizations. *J. Comput. Appl. Math.*, 243:126–138, 2012.

[16] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, Cambridge, 1994.

[17] Dulceneia Becker, Marc Baboulin, and Jack Dongarra. Reducing the amount of pivoting in symmetric indefinite systems. Technical report, INRIA, 2011.

[18] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[19] M. Benzi. Preconditioning techniques for large linear systems: A survey. *J. Comput. Phys.*, 182:418–447, 2002.

[20] M. Benzi and J. Liu. Block preconditioning for saddle point systems with indefinite (1,1) block. *Intern. J. of Computer Mathematics*, 84(8):1117–1129, 2007.

[21] M. Benzi, C. D. Meyer, and M. Tuma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Sci. Comput.*, 17:1135–1149, 1996.

[22] M. Benzi and M. Tuma. A robust incomplete factorization preconditioner for positive definite matrices. *Numer. Algor.*, 10(5):385–400, 2003.

[23] Lorenz T. Biegler. A survey on sensitivity-based nonlinear model predictive control. *IFAC Proceedings Volumes*, 46(32):499–510, 2013.

[24] Å. Björck. *Numerical Methods for Least Squares Problems.* SIAM, Philadelphia, 1996.

[25] Joshua Dennis Booth, Sivasankaran Rajamanickam, and Heidi Thornquist. Basker: A threaded sparse LU factorization utilizing hierarchical parallelism and data layouts. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 673–682. IEEE, 2016.

[26] F. Bornemann, D. Laurie, S. Wagon, and J. Waldvogel. *The SIAM 100-Digit Challenge.* SIAM, Philadelphia, PA, 2004.

[27] Brandt. Multi-level adaptive solutions to boundary-value problems. 31(138), May 1977.

[28] J. R. Bunch and B. N. Parlett. Direct methods for solving symmetric indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 8(4):639–655, 1971.

[29] J. R. Bunch and B. N. Parlett. Direct methods for solving symmetric indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 8(4):639–655, 1971.

[30] James R. Bunch and Linda Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Math. Comp.*, pages 163–179, 1977.

[31] Richard H. Byrd, Jorge Nocedal, and Richard A. Waltz. *KNITRO: An Integrated Package for Nonlinear Optimization*, pages 35–59. Springer US, Boston, MA, 2006.

[32] L. Cambier, C. Chen, E. G. Boman, S. Rajamanickam, R. S. Tuminaro, and E. Darve. An algebraic sparsified nested dissection algorithm using low-rank approximations. *SIAM J. Matrix Anal. Appl.*, 41(2):715–746, 2020.

[33] Luciana Casacio, Christiano Lyra, Aurelio Ribeiro Leite Oliveira, and Cecilia Orellana Castro. Improving the preconditioning of linear systems from interior point methods. *Computers and Operations Research*, 85:129–138, 2017.

[34] Sambuddha Chakrabarti, Matt Kraning, Eric Chu, Ross Baldick, and Stephen Boyd. Security constrained optimal power flow via proximal message passing. In *2014 Clemson University Power Systems Conference*, pages 1–8. IEEE, 2014.

[35] C. Chen, T. Liang, and G. Biros, 2020. https://github.com/ut-padas/rchol.

[36] C. Chen, T. Liang, and G. Biros. RCHOL: Randomized Cholesky factorization for solving SDD linear systems. arXiv preprint arXiv:2011.07769, 2020.

[37] Y. Chen, T.A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Softw.*, 35(3):22:1–22:14, 2008.

[38] N. Chiang, S. Peles, C. G. Petra, S. Regev, K. Swirydowicz, and J. Wang. Exasgd: 2021 kernel thrust activities. Technical Report LLNL-TR-828464, Lawrence Livermore National Lab, 2021.

[39] Nai-Yuan Chiang and Victor M. Zavala. An inertia-free filter line-search algorithm for large-scale nonlinear programming. *Comput. Optim. Appl.*, 64(2):327–354, 2016.

[40] S.-C. T. Choi, C. C. Paige, and M. A. Saunders. MINRES-QLP: a Krylov subspace method for indefinite or singular symmetric systems. *SIAM J. Sci. Comput.*, 33:1810–1836, 2011.

[41] E. Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J. Sci. Comput.*, 21:1804–1822, 2000.

[42] E. Chow and A. Patel. Fine-grained parallel incomplete LU factorization. *SIAM J. Sci. Comput.*, 37(2):C169–C193, 2015.

[43] E. Chow and Y. Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM J. Sci. Comput.*, 19:995–1023, 1998.

[44] M. Christie and M. Blunt, et al. Tenth SPE comparative solution project: A comparison of upscaling techniques. *Soc. Petrol. Eng. J.*, 4, 2001.

[45] A.T. Chronopoulos and C.W. Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25(2):153–168, 1989.

[46] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI message passing interface standard. In Karsten M. Decker and René M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 213–218, Basel, 1994. Birkhäuser Basel.

[47] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. ACM '69, page 157–172, New York, NY, USA, 1969. Association for Computing Machinery.

[48] E. Darve and M. Wootters. *Numerical Linear Algebra with Julia*. Other Titles in Applied Mathematics Series. SIAM, Philadelphia, 2021.

[49] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38, 2011.

[50] T. A. Davis, Y. Hu, and S. Kolodziej. SuiteSparse matrix collection. https://sparse.tamu.edu/, 2015–present.

[51] T. A. Davis and E. Palamadai Natarajan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 37(3):1–17, 2010.

[52] Timothy A. Davis, Sivasankaran Rajamanickam, and Wissam M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 2016.

[53] M. M. Dehnavi, D. M. Fernández, J. Gaudiot, and D. D. Giannacopoulos. Parallel sparse approximate inverse preconditioning on graphic processing units. *IEEE Trans. on Parallel and Distributed Systems*, 24(9):1852–1862, 2013.

[54] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2, Ser. A):201–213, 2002.

[55] Elizabeth D. Dolan, Jorge J. Moré, and Todd S. Munson. Benchmarking optimization software with COPS 3.0. Technical report, Argonne National Laboratory, 2004.

[56] H. Sue Dollar, Nicholas Gould, Wil Schilders, and Andrew Wathen. Implicit-factorization preconditioning and iterative solvers for regularized saddle-point systems. *SIAM J. Matrix Anal. Appl.*, 28:170–189, 01 2006.

[57] Zhijun Duan, Mirela Andronescu, Kevin Schutz, Sean McIlwain, Yoo Jung Kim, Choli Lee, Jay Shendure, Stanley Fields, C Anthony Blau, and William S Noble. A three-dimensional model of the yeast genome. *Nature*, 465(7296):363–367, 2010.

[58] I. S. Duff. MA57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Math. Softw.*, 30(2):118–144, 2004.

[59] Iain Duff, Jonathan Hogg, and Florent Lopez. A new sparse symmetric indefinite solver using a posteriori threshold pivoting. *NLAFET Working Note*, 2018.

[60] Iain Duff, Jonathan Hogg, and Florent Lopez. A new sparse LBL^T solver using a posteriori threshold pivoting. *SIAM J. Optim.*, 42(2):C23–C42, 2019.

[61] Iain Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl.*, 22:973–996, 2001.

[62] Iain S. Duff, Nick I. M. Gould, John K. Reid, Jennifer A. Scott, and Kathryn Turner. The factorization of sparse symmetric indefinite matrices. *IMA J. Numer. Anal.*, 11(2):181–204, 1991.

[63] Iain S. Duff and Stéphane Pralet. Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM J. Matrix Anal. Appl.*, 27(2):313–340, 2005.

[64] Iain S. Duff and Stéphane Pralet. Towards stable mixed pivoting strategies for the sequential and parallel solution of sparse symmetric indefinite systems. *SIAM J. Matrix Anal. Appl.*, 29(3):1007–1024, 2007.

[65] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Trans. Math. Softw.*, 9(3):302–325, 1983.

[66] Iain S. Duff, John K. Reid, N. Munksgaard, and Hans B. Nielsen. Direct solution of sets of linear equations whose matrix is sparse, symmetric and indefinite. *IMA J. Numer. Anal.*, 23(2):235–250, 1979.

[67] M. Embree. How descriptive are GMRES convergence bounds? Technical report, Oxford University Computing Laboratory, 1999.

[68] Massimiliano Ferronato. Preconditioning for sparse linear systems at the dawn of the 21st century: History, current developments, and future perspectives. *ISRN Applied Mathematics*, 2012, 12 2012.

[69] R. Fletcher. Factorizing symmetric indefinite matrices. *Linear Algebra and its Applications*, 14(3):257–272, 1976.

[70] D. C.-L. Fong and M. A. Saunders. LSMR: An iterative algorithm for least-squares problems. *SIAM J. Sci. Comput.*, 33(5):2950–2971, 2011.

[71] D. C.-L. Fong and M. A. Saunders. CG versus MINRES: An empirical comparison. *SQU J. Sci.*, 17(1):44–62, 2012. http://stanford.edu/group/SOL/reports/SOL-2011-2R.pdf.

[72] R. W. Freund and N. M. Nachtigal. A new Krylov-subspace method for symmetric indefinite linear systems. *IMACS*, pages 1253–1256, 1994.

[73] J. Gao, Q. Chen, and G. He. A thread-adaptive sparse approximate inverse preconditioning algorithm on multi-GPUs. *Parallel Computing*, 101:102724, 2021. http://www.sciencedirect.com/science/article/pii/S0167819120301083.

[74] S. Gazzola, P. C. Hansen, and J. G. Nagy. IR Tools: a MATLAB package of iterative regularization methods and large-scale test problems. *Numer. Algor.*, 81(3):773–811, 2019.

[75] A. George, G. Alan, and J.W.H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall series in computational mathematics. Prentice-Hall, 1981.

[76] Alan George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10(2):345–363, 1973.

[77] P. E. Gill, W. Murray, D. B. Ponceleón, and M. A. Saunders. Preconditioners for indefinite systems arising in optimization. *SIAM J. Matrix Anal. Appl.*, 13:292–311, 1992.

[78] Abeynaya Gnanasekaran and Eric Darve. Hierarchical orthogonal factorization: Sparse square matrices. *SIAM J. Matrix Anal. Appl.*, 43(1):94–123, 2022.

[79] G. H. Golub and C. Greif. On solving block-structured indefinite linear systems. *SIAM J. Sci. Comput.*, 6(24):2076–2092, 2003.

[80] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. The Johns Hopkins University Press, Baltimore, fourth edition, 2013.

[81] Jacek Gondzio. HOPDM (version 2.12) – a fast LP solver based on a primal-dual interior point method. *European Journal of Operational Research*, 85(1):221–225, 1995.

[82] Nicholas I. M. Gould, Dominique Orban, and Philippe L. Toint. CUTEst: a constrained and unconstrained testing environment with safe threads for mathematical optimization. *Comput. Optim. Appl.*, 60(3):545–557, 2015.

[83] Nicholas I. M. Gould and Jennifer A. Scott. A numerical evaluation of HSL packages for the direct solution of large sparse, symmetric linear systems of equations. *ACM Trans. Math. Softw.*, 30(3):300–325, 2004.

[84] Nicholas I. M. Gould, Jennifer A. Scott, and Yifan Hu. A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Trans. Math. Softw.*, 33(2):10–es, 2007.

[85] C. Greif, S. He, and P. Liu. SYM-ILDL: Incomplete LDL^T factorization of symmetric indefinite and skew-symmetric matrices. arXiv preprint arxiv.org/abs/1505.07589, 2016.

[86] M. J. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput.*, 18:838–853, 1997.

[87] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, may 1988.

[88] P. C. Hansen. Regularization Tools version 4.0 for MATLAB 7.3. *Numer. Algor.*, 46:189–194, 2007.

[89] Kai He, Sheldon X-D Tan, Hai Wang, and Guoyong Shi. GPU-accelerated parallel sparse LU factorization method for fast circuit analysis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(3):1140–1150, 2015.

[90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2006.

[91] Pascal Hénon, Pierre Ramet, and Jean Roman. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing*, 28(2):301–321, 2002.

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

[93] J. D. Hogg and J. A. Scott. An indefinite sparse direct solver for large problems on multicore machines. Technical Report RAL-TR-2010-011, Rutherford Appleton Laboratory, Chilton, 2010.

[94] Jonathan D. Hogg and Jennifer A. Scott. Pivoting strategies for tough sparse indefinite systems. *ACM Trans. Math. Softw.*, 40(1):1–19, 2013.

[95] Jonathan D. Hogg and Jennifer A. Scott. Compressed threshold pivoting for sparse symmetric indefinite systems. *SIAM J. Matrix Anal. Appl.*, 35(2):783–817, 2014.

[96] Karl Erik Holter, Miroslav Kuchta, and Kent-Andre Mardal. Robust preconditioning for coupled Stokes–Darcy problems with the Darcy problem in primal form. *Computers and Mathematics with Applications*, 91:53–66, 2021. Robust and Reliable Finite Element Methods in Poromechanics.

[97] C. Janna, M. Ferronato, F. Sartoretto, and G. Gambolati. FSAIPACK: A software package for high-performance Factored Sparse Approximate Inverse preconditioning. *ACM Trans. Math. Softw.*, 41(2):10:1–10:26, 2015.

[98] Juan Jerez, Sandro Merkli, Samir Bennani, and Hans Strauch. FORCES-RTTO: A tool for on-board real-time autonomous trajectory planning. In *10th International ESA Conference on Guidance, Navigation and Control Systems*, pages 1–22. ESA Salzburg, Austria, 2017.

[99] E. F. Kaasschieter. A practical termination criterion for the conjugate gradient method. *BIT*, 28(2):308–322, jun 1988.

[100] William Karush. Minima of functions of several variables with inequalities as side conditions, 1939.

[101] Martin Keller, Severin Geiger, Marco Günther, Stefan Pischinger, Dirk Abel, and Thivaharan Albin. Model predictive air path control for a two-stage turbocharged spark-ignition engine with low pressure exhaust gas recirculation. *International Journal of Engine Research*, 21(10):1835–1845, 2020.

[102] B. Klockiewicz and E. Darve. Sparse hierarchical preconditioners using piecewise smooth approximations of eigenvectors. arXiv preprint arXiv:1907.03406v1, 2019.

[103] Philip A. Knight, Daniel Ruiz, and Bora Uçar. A symmetry preserving algorithm for matrix scaling. *SIAM J. Matrix Anal. Appl.*, 35(3):931–955, 2014.

[104] L. Yu. Kolotilina and A. Yu. Yeremin. Factorized sparse approximate inverse preconditionings. I: Theory. *SIAM J. Matrix Anal. Appl.*, 14:45–58, 1993.

[105] L. Yu. Kolotilina and A. Yu. Yeremin. Factorized sparse approximate inverse preconditioning II: Solution of 3D FE systems on massively parallel computers. *Int. J. High Speed Comput.*, 7:191–215, 1995.

[106] Tristan Konolige and Jed Brown. A parallel solver for graph laplacians. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[107] J. Kouatchou. Basic comparison of Python, Julia, R, MATLAB and IDL, 2016. https://modelingguru.nasa.gov/docs/DOC-2625.

[108] Drosos Kourounis, Alexander Fuchs, and Olaf Schenk. Toward the next generation of multiperiod optimal power flow solvers. *IEEE Transactions on Power Systems*, 33(4):4005–4014, 2018.

[109] H. W. Kuhn and A. W. Tucker. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability, 1950*, pages 481–492, Berkeley and Los Angeles, 1951. University of California Press.

[110] S. Laut, M. Casas, and R. Borrel. Communication-aware sparse patterns for the factorized approximate inverse preconditioner, https://dl.acm.org/doi/abs/10.1145/3502181.3531472, 2022. Presented at the 31st International Symposium on High-Performance Parallel and Distributed Computing.

[111] Xiaoye S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.

[112] Xiaoye S. Li and James W. Demmel. A scalable sparse direct solver using static pivoting. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing 1999 (San Antonio, TX)*, page 10. SIAM, Philadelphia, 1999.

[113] Yudong Ma, Jadranko Matuško, and Francesco Borrelli. Stochastic model predictive control for building HVAC systems: Complexity and conservatism. *IEEE Transactions on Control Systems Technology*, 23(1):101–116, 2014.

[114] J. Maack and S. Abhyankar. ACOPF sparse linear solver test suite, 2020. github.com/NREL/opf_matrices.

[115] A. M. Manea, J. Sewall, H. A. Tchelepi, et al. Parallel multiscale linear solver for highly detailed reservoir models. *Soc. Petrol. Eng. J.*, 21:2–62, 2016.

[116] T. A. Manteufel. An incomplete factorization technique for positive definite linear systems. *Math. Comp.*, 34:473–497, 1980.

[117] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.*, 31:148–162, 1977.

[118] Daniel K Molzahn, Florian Dörfler, Henrik Sandberg, Steven H Low, Sambuddha Chakrabarti, Ross Baldick, and Javad Lavaei. A survey of distributed optimization and control algorithms for electric power systems. *IEEE Transactions on Smart Grid*, 8(6):2941–2962, 2017.

[119] Alexis Montoison and Dominque Orban. TriCG and TriMR: Two iterative methods for symmetric quasi-definite systems. *SIAM J. Sci. Comput.*, 43(4):302–325, 2021.

[120] J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer Series in Operations Research. Springer Verlag, New York, second edition, 2006.

[121] Jerome W. O'Neal. *The Use of Preconditioned Iterative Linear Solvers in Interior-Point Methods and Related Topics.* PhD dissertation, Georgia Institute of Technology, 2005.

[122] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12:617–629, 1975.

[123] C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Softw.*, 8:43–71, 1982.

[124] http://pastix.gforge.inria.fr/files/README-txt.html.

[125] C. Petra, B. Gavrea, M. Anitescu, and F. Potra. A computational study of the use of an optimization-based method for simulating large multibody systems. *Optimization Methods & Software*, 24(6):871–894, 2009.

[126] Cosmin G. Petra and Nai-Yuan Chiang. HiOp – User Guide. Technical Report LLNL-SM-743591, Lawrence Livermore National Laboratory. https://github.com/LLNL/hiop/blob/develop/doc/hiop_usermanual.pdf.

[127] Cosmin G. Petra, Nai-Yuan Chiang, Slaven Peles, Asher Mancinelli, Cameron Rutherford, Jake K. Ryan, and Michel Schanen. HPC solver for nonlinear optimization problems, 2017. https://github.com/LLNL/hiop/tree/master.

[128] PNNL machines. https://www.pnnl.gov/capabilities/advanced-computer-science-visualization-data.

[129] Anna Pyzara, Beata Bylina, and Jarosław Bylina. The influence of a matrix condition number on iterative methods' convergence. In *Proceedings of the Federated Conference on Computer Science and Information System*, pages 459–464. IEEE, 2011.

[130] Jan M. Rabaey. *Digital Integrated Circuits: A Design Perspective*. Prentice-Hall, Inc., USA, 1996.

[131] T. Rauber and G. Rünger. *Parallel Programming: for Multicore and Cluster Systems*. Springer Berlin Heidelberg, 2013.

[132] Tim Rees and Chen Greif. A preconditioner for linear systems arising from interior point optimization methods. *SIAM J. Sci. Comput.*, 29:1992–2007, 2007.

[133] Tyrone Rees and Jennifer Scott. A comparative study of null-space factorizations for sparse symmetric saddle point systems: A comparative study of null-space factorizations. *Numerical Linear Algebra with Applications*, 25:e2103, 06 2017.

[134] S. Regev, 2019. https://github.com/shakedregev/SSAI.

[135] S. Regev, 2019. https://tinyurl.com/matfiles.

[136] S. Regev and M. A. Saunders. minres20: MATLAB software for MINRES and several preconditioners, 2020. http://stanford.edu/group/SOL/software/minres.

[137] S. Regev and M. A. Saunders. SSAI and SSAI_LS: Sparse approximate inverse preconditioners for CG and MINRES, http://stanford.edu/group/SOL/talks/22householder-saunders.pdf, 2022. Presented at the XXI Householder Symposium on Numerical Linear Algebra, Selva di Fasano (Br), Italy.

[138] Shaked Regev, Nai-Yuan Chiang, Eric Darve, Cosmin G. Petra, Michael A. Saunders, Kasia Świrydowicz, and Slaven Peleš. HyKKT: a hybrid direct-iterative method for solving KKT linear systems. *Optimization Methods & Software*, xx:24pp, 2022.

[139] Shaked Regev and Michael Saunders. SSAI: A symmetric sparse approximate inverse preconditioner for the conjugate gradient method. Working paper, 2019.

[140] P. Reinert, Hermann Krebs, and Evgeny Epelbaum. Semilocal momentum-space regularized chiral two-nucleon potentials up to fifth order. *The European Physical Journal A*, 54(5):1–49, 2018.

[141] Steven C. Rennich, Darko Stosic, and Timothy A. Davis. Accelerating sparse Cholesky factorization on GPUs. *Parallel Computing*, 59:140–150, 2016. Theory and Practice of Irregular Applications.

[142] Edward Jason Riedy. *Making Static Pivoting Scalable and Dependable*. PhD thesis, University of California, Berkeley, 2010.

[143] Donald J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In Ronald C. Read, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, 1972.

[144] François-Henry Rouet, Xiaoye S. Li, Pieter Ghysels, and Artem Napov. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Trans. Math. Softw.*, 42(4), 2016.

[145] J. Rozložník. *Saddle-point Problems and Their Iterative Solution.* Cham: Birkhäuser, 2018.

[146] D. Ruiz. A scaling algorithm to equilibrate both rows and columns norms in matrices. Technical Report RAL-TR-2001-034, Rutherford Appleton Laboratory, 2001.

[147] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for nonsymmetric linear systems. *SIAM J. Sci. and Statist. Comput.*, 7:856–869, 1986.

[148] D. K. Salkuyeh and F. Toutounian. A new approach to compute sparse approximate inverse of an SPD matrix. *IUST - Int. J. Eng. Sci.*, 15:87–95, 2004.

[149] Olaf Schenk, Andreas Wächter, and Michael Hagemann. Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale

nonconvex interior-point optimization. *Comput. Optim. Appl.*, 36(2-3):321–341, April 2007.

[150] J. Scott and M. Tuma. Solving mixed sparse-dense linear least squares by preconditioned iterative methods. *SIAM J. Sci. Comput.*, 39(6):A2422–A2437, 2017.

[151] J. Scott and M. Tuma. A Schur complement approach to preconditioning sparse linear least-squares problems with some dense rows. *Numer. Algor.*, 79(1):1147–1168, 2018.

[152] J. Scott and M. Tuma. Sparse stretching for solving sparse-dense linear least-squares problems. *SIAM J. Sci. Comput.*, 41(3):A1604–A1625, 2019.

[153] S. R. Searle. *Matrix Algebra Useful for Statistics*. John Wiley and Sons, Hoboken, NJ, 1982.

[154] Cristiana J Silva and Delfim FM Torres. Optimal control for a tuberculosis model with reinfection and post-exposure interventions. *Mathematical Biosciences*, 244(2):154–164, 2013.

[155] Jean-Pierre Sleiman, Jan Carius, Ruben Grandia, Martin Wermelinger, and Marco Hutter. Contact-implicit trajectory optimization for dynamic object manipulation. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6814–6821. IEEE, 2019.

[156] https://portal.nersc.gov/project/sparse/strumpack/.

[157] K. Świrydowicz, E. Darve, J. Maack, S. Regev, M. A. Saunders, S. J. Thomas, and S. Peleš. Linear solvers for power grid optimization problems: a review of GPU-accelerated linear solvers. *Parallel Computing*, (submitted), 2020.

[158] Byron Tasseff, Carleton Coffrin, Andreas Wächter, and Carl Laird. Exploring benefits of linear solver parallelism on modern nonlinear optimization applications, 2019.

[159] I. K. Tezaur, M. Perego, A. G. Salinger, R. S. Tuminaro, and S. F. Price. Albany/FELIX: a parallel, scalable and robust, finite element, first-order Stokes

approximation ice sheet solver built for advanced analysis. *Geosci. Model. Dev.*, 8:1197–1220, 2015.

[160] Rue Toulouse, Patrick Amestoy, Tim Davis, Iain Duff, and Patrick Amestoy. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17, 01 1998.

[161] L. N. Trefethen. A hundred-dollar, hundred-digit challenge. *SIAM News*, 35:65, 2002.

[162] L. N. Trefethen. The SIAM 100-dollar, 100-digit challenge, 2002. https://people.maths.ox.ac.uk/trefethen/hundred.html.

[163] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, 1983.

[164] H. Uzawa. *Iterative Methods for Concave Programming*. Johns Hopkins Studies in the Mathematical Sciences. Stanford University Press, Stanford, CA, 1958.

[165] R. J. Vanderbei. Symmetric quasidefinite matrices. *SIAM J. Optim.*, 5:100–113, 1995.

[166] Robert J. Vanderbei. LOQO: An interior point code for quadratic programming. *Optimization Methods & Software*, 11(1):451–484, 1999.

[167] Nelle Varoquaux, Ferhat Ay, William Stafford Noble, and Jean-Philippe Vert. A statistical approach for inferring the 3D structure of the genome. *Bioinformatics*, 30(12):i26–i33, 2014.

[168] A. Wächter and L. T. Biegler. Line search filter methods for nonlinear programming: Motivation and global convergence. *SIAM J. Optim.*, 16(1):1–31, 2005.

[169] Andreas Wächter and Lorenz T. Biegler. Line search filter methods for nonlinear programming: Local convergence. *SIAM J. Optim.*, 16(1):32–48, 2005.

[170] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106(1):25–57, 2006.

[171] Allen Wang, Ashkan Jasour, and Brian C Williams. Non-Gaussian chance-constrained trajectory planning for autonomous vehicles under agent uncertainty. *IEEE Robotics and Automation Letters*, 5(4):6041–6048, 2020.

[172] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Discr. Meth.*, 2, 03 1981.

[173] J. Zhang and T. Xiao. A multilevel block incomplete Cholesky preconditioner for solving normal equations in linear least squares problems. *J. Appl. Math. and Comput.*, 11(1):59–80, 2003.