Making Static Pivoting Scalable and Dependable

by

Edward Jason Riedy

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor James Demmel, Chair
Professor Katherine Yelick
Professor Sanjay Govindjee

Fall 2010

Making Static Pivoting Scalable and Dependable

Abstract

Making Static Pivoting Scalable and Dependable

by

Edward Jason Riedy

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor James Demmel, Chair

Solving square linear systems of equations $Ax = b$ is one of the primary workhorses in scientific computing. With asymptotically and practically small amounts of extra calculation and higher precision, we can render solution techniques *dependable*. We produce a solution with tiny error for almost all systems where we should expect a tiny error, and we correctly flag potential failures.

Our method uses a proven technique: iterative refinement. We extend prior work by applying extra precision not only in calculating the residual $b - Ay_i$ of an intermediate solution y_i but also in carrying that intermediate solution y_i. Analysis shows that extra precision in the intermediate solutions lowers the limiting backward error (measuring perturbations in the initial problem) to levels that produce a forward error (measuring perturbations in the solution) not much larger than the precision used to store the result. We also demonstrate that condition estimation is not necessary for determining success, reducing the computation in refinement substantially.

This basic, dependable solver applies to typical dense LU factorization methods using partial pivoting as well as methods that risk greater failure by choosing pivots for non-numerical reasons. Sparse factorization methods may choose pivots to promote structural sparsity or even choose pivots *before* factorization to decouple the phases. We show through experiments that solutions using these restrictive pivoting methods still have small error so long as an estimate of factorization quality, the growth factor, does not grow too large. Our refinement algorithm dependably flags such failures. Additionally, we find a better choice of heuristic for sparse static pivoting than the defaults in Li and Demmel's SuperLU package.

Static pivoting in a distributed-memory setting needs an algorithm for choosing pivots that does not rely on fitting the entire matrix into one memory space. We investigate a set of algorithms, Bertsekas's auction algorithms, for choosing a static pivoting via maximum weight perfect bipartite matching. Auction algorithms have a natural mapping to distributed memory computation through their bidding mechanism. We provide an analysis of the auction algorithm fitting it comfortably in linear optimization theory and characterizing

approximately maximum weight perfect bipartite matches. These approximately maximum weight perfect matches work well as static pivot choices and can be computed much more quickly than the exact maximum weight matching.

Finally, we consider the performance of auction algorithm implementations on a suite of real-world sparse problems. Sequential performance is roughly equivalent to existing implementations like Duff and Koster's MC64, but varies widely with different parameter and input settings. The parallel performance is even more wildly unpredictable. Computing approximately maximum weight matchings helps performance somewhat, but we still conclude that the performance is too variable for a black-box solution method.

To Nathan.

Contents

List of Figures

List of Tables

List of Listings

Acknowledgments

Here I come with a little bit of help
Here I come with a little bit of help
Here I come with a little bit of help
Here I come with a little bit of ...

First and foremost, my wife Nathan Strange supported this work even while I fell apart.

There are a host of other people involved over the time spent with this windmill. I no doubt will forget many but will place some names into the corner of the Engineering Library.

Who do I thank? Who should I not thank?

At Berkeley: my adviser Dr. James Demmel and my committee member Dr. Katherine Yelick for many lessons; external member Dr. Sanjay Govindjee for listening, reading, and putting up with the reminder of times long past; Dr. W. Kahan for being Dr. W. Kahan (and suggesting the term "dependable"); Dr. Beresford Parlet for encouraging my voice; Dr. David Culler for introducing me to a particular massively multithreaded architecture; and many more.

People of particular note, many part of the Berkeley diaspora, and with full recognition that I'll miss people who should be acknowledged: Dr. Mark Hoemmen for the regular reminders; Dr. Rich Vuduc for the faith; Dr. David Bader for looking the other way; Bekah Simone and Samantha 'Slaughter' Anne for tolerating my working at their workplace and possibly scaring the customers; James Hsiao for remembrances; Dr. Tim Davis and Dr. Joe Wilson for not shooting me down even when I may have deserved it; Jim Hranicky, Dave Lopata, Brian Smith, Mike Cerrato, and really the whole UF CIS/CISE consultant crew at the time for helping me learn far more low-level system information than people ever should know; and Data Guthrie Martin for reminding me of the fun in words.

Institutions to which I owe thanks include: University of California in Berkeley, CA, where I served my graduate school years; Virginia Intermont College in Bristol, VA, where I taught basic mathematics as an adjunct; CERFACS in Toulouse, France, which I visited corroborating the nigh hopelessness of distributed bipartite matching; and Georgia Institute of Technology in Atlanta, GA, where I am an official Research Scientist and have occasionally borrowed computation resources.

There are far too many coffee shops, bars, and other places where by hook or by crook I managed non-negligible amounts of thesis-related work: Panama Bay Coffee, Concord, CA; Zazzy's Coffee House, Abingdon, VA; Java J's, Bristol, VA; Free Speech Movement Cafe, Berkeley, CA; Brewed Awakening, Berkeley, CA; Cafe Strada, Berkeley, CA; Octane Coffee, Atlanta, GA; Cypress Street Pint and Plate, Atlanta, GA; Corner Tavern, East Point, GA; Midtown Tavern, Atlanta, GA; Malaprop's Bookstore/Cafe, Asheville, NC; Le Chat d'Oc,

Toulouse, France; Z'otz Cafe, New Orleans, LA; Hartsfield-Jackson Atlanta International Airport, Atlanta, GA; Tri-Cities Regional Airport, Blountville, TN; and BWI Thurgood Marshall Airport, MD.

Software whose development communities I thank includes GNU Emacs, LaTeX, GNU Octave, and GNU R[85] (particularly the ggplot2 graphics system[102] used throughout). The proprietary SpiderOak backup system is lending me some temporary piece of mind as well.

I won't die with a little bit of help
I won't die with a little bit of help
I won't die with a little bit of help
I won't die with a little bit of ...

— *for squirrels,* superstar

Star.

Chapter 1

Overview

1.1 Making static pivoting scalable and dependable

The ultimate goal of this thesis is to provide a fully automatic solution of $Ax = b$ for use *within* other computations, providing predictable and small errors upon success while clearly indicating failures. Within that goal, we want to choose pivots within sparse LU factorization statically for use in distributed memory factorization. A static pivot order provides static data dependencies for the numerical factorization, and the static dependencies open the throttle for a faster numerical factorization.

For the goal of solving $Ax = b$ for use within other routines, we provide a variation of iterative refinement[18, 103] with

- clearly defined error thresholds for successfully computed solutions,
- clear signaling of computations that likely do not meet the thresholds, and
- relevant diagnostic interpretations of many failures.

Appropriate ways to convey diagnostic information to programs is future work. Here we provide the first step by ascribing failures to likely causes.

1.1.1 Dependability

We provide a *dependable* algorithm in Part I.

Dependable Returns a result with small error often enough that you expect success with a small error, and clearly signals results that likely contain large errors.

Chapter 2 defines the error measures of interest.

Chapter 3 provides and analyzes the iterative refinement algorithm. We use twice the input precision both within residual computations as well as for carrying intermediate solutions.

Our refinement algorithm requires only $O(N^2 \cdot k)$ higher-precision floating-point operations after $O(N^3)$ operations in the working precision, where N is the size of the matrix and there are k right-hand sides. Applying extra precision in both places permits both perturbations (backward errors) and solution error (forward errors) that are a small multiple of the input precision for systems that are not too difficult to solve. Difficult systems are those so ill conditioned that the matrix is within a few rounding errors of being singular. We also consider low-precision factorizations like those used with GPGPU and other accelerators. We duplicate others' backward error analysis results [69] while explaining and diagnosing the solution error limitations; future work will validate our algorithm on these architectures. Chapter 2 provides more detail on the norms involved in measuring a system's conditioning. We detect difficult systems by tracking ratios of intermediate quantities.

Chapter 4 validates refinement's dependability against artificial dense test systems extended from Demmel et al. [35]. We consider real and complex systems in both single and double precision. For both, we examine the iteration counts as a measure of performance.

Chapter 5 validates refinement against practical sparse matrices of modest size. We extend the results to explore restricted and perturbed factorizations. Sparse *LU* factorization codes often restrict the pivot selection to preserve sparsity. Threshold pivoting, a common choice[49], heuristically chooses the pivot of least degree from those within some threshold of the unrestricted, partial pivoting choice. We validate also against threshold pivoting, properly detecting the additional failures to converge. Static pivoting used within distributed-memory parallel *LU* factorization[71] fixes the pivot order and perturbs the pivot should it prove too small. We improve the perturbation heuristic and validate our algorithm against static pivoting, again detecting the additional failures when the perturbations prove too large.

1.1.2 Scalability

Bertsekas's auction algorithm[12] for computing a maximum weight complete bipartite matching is the basis for choosing the static pivot order. Part II extends the auction algorithm for MPI-based, distributed memory platforms. Chapter 6 introduces the matching problem underneath our pivot choice. Chapter 6 examines the problem in greater detail, using linear optimization and linear algebra to simplify existing results.

Chapter 7 details the auction algorithm within the linear optimization framework. We provide previously unpublished details necessary for effective implementation on real data, like correct definitions for vertices of degree one. We also improve on the successive approximation algorithm with two new heuristics. The first dynamically reduces the approximation factor to the actually *achieved* approximation (Section 7.5). The second applies a single pass of price reduction before determining the achieved approximation (Section 7.7).

Chapter 8 examines the performance of the auction algorithm. The performance is, unfortunately, unpredictable. While the implementation is memory-scalable on distributed memory machines, the results show extreme performance variability across our moderately sized test sparse matrices. The extreme variability renders this a special-purpose tool and

not a good fit for blind use in the dependable solver we want to produce.

1.2 Contributions

- Enhancements to iterative refinement:
  - Analysis and experiments justify dependability criteria for the forward error that do not depend on explicit condition estimation (Chapters 3 through 5).
    - Given careful computation of the residual, the forward relative error in *working precision* ε_w is at most a small multiple of ε_w in all cases where the refinement step size converges to a slightly smaller multiple of ε_w and that *also* have a corresponding backward relative error *of the solution in extended precision* of a small multiple of ε_w^2. See Equation (3.6.12).
    - We use a constant c_N for a given system to encapsulate round-off effects related to the matrix's size. Let $c_N = \max\{8, \sqrt{N}\}$ for dense systems to permit both one bit of error and a double-rounding in converting an extended-precision solution back to working precision. For sparse cases, c_N can be replaced by the square root of the largest row degree of the filled factorization. However, the experiments in Chapter 4 and Chapter 5 show no noticeable dependence of error measures on the systems' size.
    - Relying on refinement's convergence does flag a small number of successful cases as failures even when the relevant condition number is below $1/\varepsilon_f$, where ε_f denotes the precision used to factor the matrix. However, we never let a failed case pass as successful.
  - Similar experiments demonstrate that iterative refinement remains dependable applied with factorizations containing intentional errors, *e.g.* restricted pivoting or perturbed factorizations. These less-accurate factorizations have fewer successful results.
  - Using column-relative perturbations rather than SuperLU's default norm-based perturbations permits refinement to reach backward stable results for sparse systems and small forward error so long as the conditioning and element growth[1] are not too severe. The refinement algorithm remains dependable.
- Distributed matching algorithm:
  - A memory-scalable auction algorithm can compute a useful static pivoting. However, Chapter 8 shows that performance is wildly unpredictable.

[1] Sometimes called pivot growth, but we wish to emphasize that the growth is in matrix elements and not just the pivots.

- A touch of linear algebra and linear programming greatly simplifies the analysis of Bertsekas's auction algorithm (Chapter 6).
- We include two new heuristics. The first dynamically reduces the weight of the approximately maximum weight perfect matching to a computed bound on the *achieved* approximation (Section 7.5) rather than an arbitrary multiple of an iteration's initial factor. The second applies a single pass of price reduction before determining the achieved approximation (Section 7.7). These help "easy" problems finish rapidly and occasionally help when chasing the final matching edge.
- Auction algorithms can compute approximately maximum weighted maximal matchings. A large approximation factor smooths out a few of the wild performance problems, and the approximate maximum weight maximal matchings appear not to affect iterative refinement.

1.3 Notation

This thesis is unfortunately heavy on notation. A quick summary:

- The square linear system of n equations and n variables with one right-hand side under consideration is $Ax = b$. Results are repeated for multiple right-hand sides treating each column independently.
- The vector y is a computed solution to $Ax = b$.
- Quantities at different iterations of an algorithm are denoted by a subscript, so y_i is the computed solution during the i^{th} iteration.
- Matrix and vector elements are referred to with parentheses. $A(i, j)$ refers to the element at the i^{th} row and j^{th} column.
- Many Octave [51] and Matlab [74] notations and conventions are used. For example, $|A|$ is an element-wise absolute value. Rather than prefixing the operation with a dot, however, we denote an elementwise division of vectors v and w by v/w.
- Diagonal scaling matrices are denoted as D_w, where w is a scalar or vector along the diagonal. At times, the diagonal quantity is assumed and the subscript refers to the iteration.
- Any norm without a qualifier is an ∞ norm, although we try to add the ∞ explicitly.
- Division by zero is handled specially in computations related to error measures. To handle exact zeros, we let $0/0 = 0$ while anything else divided by zero diverges to ∞. Section 3.9 provides the rationale. When comparing scaling factors, however, $0/0 = 1$.

Part I

Dependability: Iterative Refinement for $Ax = b$

Chapter 2

Defining accuracy and dependability

2.1 Introduction

Given the ubiquitous linear system $Ax = b$ where A is a square, $n \times n$ matrix A and b is a $n \times m$ matrix, we want the $n \times m$ solution x if one exists. We want an accurate solution, and we want it quickly. The limitations of finite precision, however, require a few trade-offs.

Finding a solution given a dense matrix A has the same complexity as matrix multiplication [36], and practical algorithms require $O(n^3)$ operations. Computing x with exact arithmetic is prohibitively expensive; the bit complexity of individual operations will grow exponentially with n. Even the cost of computing tight bounds around x using interval arithmetic over finite precision is NP-hard [89], and computing looser intervals increases the cost of the $O(n^3)$ computation [27].

So for common uses and large systems, we should assume only finite-precision arithmetic with all its limitations [59, 34]. But we still want as much accuracy or as small an error as is reasonable. There are many different measures of accuracy and many different expectations on "reasonable". We define measures of accuracy through the rest of this chapter. "Reasonable" is up to the reader and user. We assume that a dependable solver that does not significantly decrease the performance of standard methods, particularly LU factorization, is reasonable. Adding a small multiple of $O(n^2)$ computation and memory traffic will not effect the performance on large systems.

Our term "dependable" admits a somewhat fuzzy definition:

Definition 2.1.1: A **dependable** solver for $Ax = b$ returns a result x with small error often enough that you expect success with a small error, and clearly signals results that likely contain large errors.

Chapter 3's algorithm delivers small errors for almost all systems that are not too ill-conditioned and do not encounter too-large element growth during LU factorization. Table 2.1 depicts the different situations that may occur with our dependable refinement algorithm. Chapters 4 and 5 provide results with dense and sparse test systems, respectively. A

true forw. error	cond. num.	Alg. reports	Likeliness given err & cond.
$\leq C \cdot \varepsilon_w$	$\leq 1/\varepsilon_w$	success	Very likely
		failure	Somewhat rare
$> C \cdot \varepsilon_w$	$\leq 1/\varepsilon_w$	success	May occur, not yet seen
		failure	Practically certain
$\leq C \cdot \varepsilon_w$	$> 1/\varepsilon_w$	success	Whenever feasible
		failure	Very likely
$> C \cdot \varepsilon_w$	$> 1/\varepsilon_w$	success	May occur, not yet seen
		failure	Practically certain

Table 2.1: Conditions, where ε_w denotes the precision to which the solution is returned and C is a small constant related to the size of the system. This table is somewhat generic, and the condition number in question is the one relating the forward error to an achieved tiny backward error. We declare cases "rare" when they should never happen but may conceivably occur. For example, we have seen no cases where the relevant condition number is $> 1/\varepsilon_w$ and iterative refinement converges to an incorrect solution while declaring success. This table does not depict formal probabilities but instead intuitive interpretation.

dependable solver should be useful to other software and not just users or numerical analysts. Either the solution has a small error, or there is a clear indication that the solver likely failed. Figure 2.1 shows graphically what we intend for a solver based on LU factorization targeting a small forward error in the solution x. The final errors are small (in blue) or the algorithm determines that the result may be untrustworthy (in red).

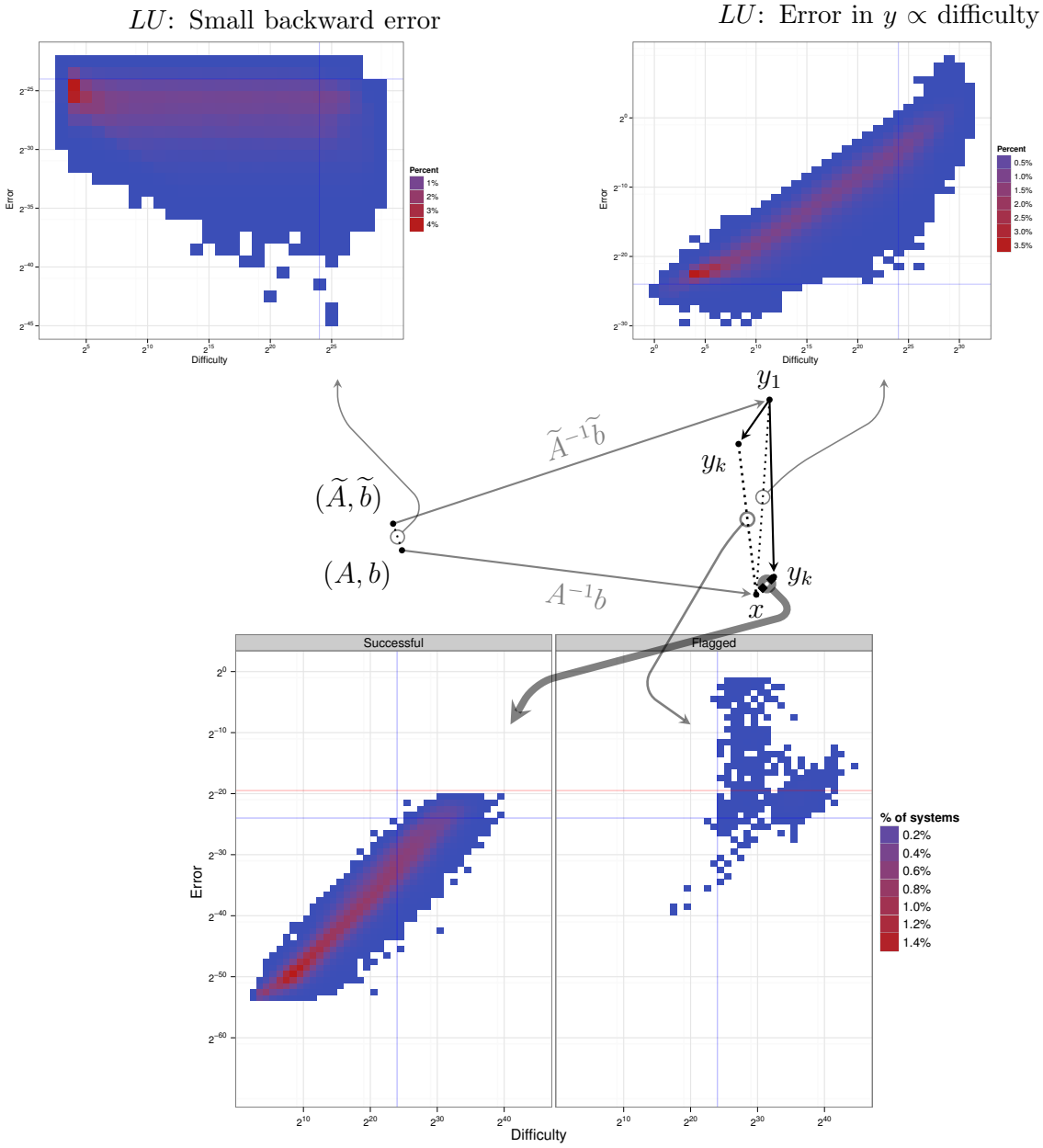
We do permit a slight possibility of catastrophic mistake in a dependable solver. It is possible that a low-rank A may encounter numerical errors that lead to a factorization and a solution that happens to satisfy $Ax = b$. We will touch on techniques for preventing these mistakes in Chapter 4.

2.2 Measuring errors

Let $Ax = b$ be the system of equations with $n \times n$ matrix A and $n \times 1$ column vectors x and b. The vector x is the true solution. Given a computed solution y, we measure error in two different ways: *backward* error and *forward* error. Backward error measures a distance to the nearest system exactly satisfied by y, $(A + \delta A)y = (b + \delta b)$. Forward error measures a distance between the true solution x and the computed solution y. Each of these distances can be measured different ways, and we will list our used measures shortly. We will consider only single solutions. We treat multiple right-hand sides independently rather than as a block.

The majority of this section includes standard definitions available in the literature [59, 34]. We do introduce an unusual variant, a column-relative backward error, that will prove useful

Figure 2.1: Gaussian elimination with partial pivoting almost always ensures a small residual and backward error, but the forward error in the computed y grows proportionally with the *difficulty* (condition number, see Section 2.3). Iterative refinement produces solutions either declared successful with small forward error or are flagged with no statement about the forward error. The horizontal axis shows an approximate *difficulty*, and the vertical axis shows error. The horizontal lines denote the machine precision (blue) and an error bound (red). The vertical blue line is the limit where we no longer expect convergence, although it still happens. Further details are in Chapter 4.



Refined: Accepted with small errors in y, or flagged with unknown error.

for making iterative refinement's forward error dependable in Chapter 4.

Both error measures have applications. Backward error is most useful when the initial system is approximate; comparing the backward error to the approximation's size provides a measure of trust that the solution is "good enough" relative to the input approximation. If the data was produced from physical measurements with known tolerances, a backward error within those tolerances implies the solution is consistent with our best knowledge of the physical system. In a numerical optimization algorithm, a small backward error implies the solution fits within a trust region, *etc.* The forward error makes no assumptions about the quality of inputs A and b. The forward error is appropriate for general-purpose libraries and dynamic environments that cannot know users' intentions.

On top of the choice of forward or backward error, each error can be measured differently. Two measurements are common, normwise and componentwise. We will introduce a third, a column-wise measurement for backward error. Both can use different compatible matrix and vector norms, but we will use the ∞-norm for almost all results:

$$\|v\|_\infty = \max_i |v(i)| \text{ for a vector, and}$$
$$\|A\|_\infty = \max_i \sum_j |A(i,j)| \text{ for a matrix.}$$

As described in Chapter 1.3, parentheses (*e.g.* $v(i)$, $A(i,j)$) select a particular entry of a vector or matrix, and the absolute value $|A|$ applies elementwise. If we let 1_c denote the vector of all 1s conformal to A's columns, then $\|A\|_\infty = \max_i |A|1_c$.

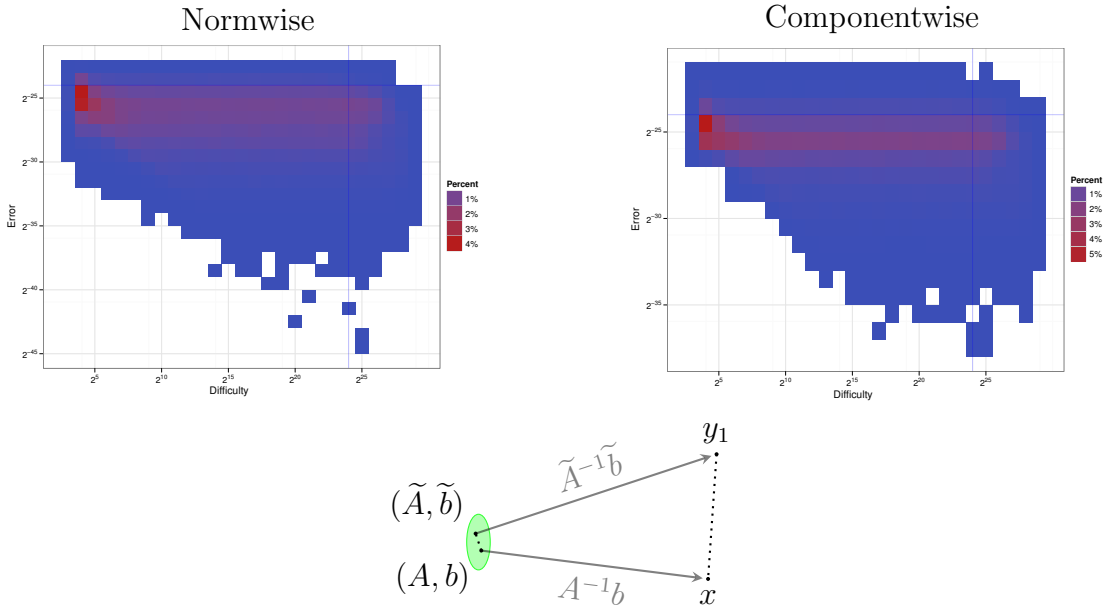
2.2.1 Backward errors

The first error measurement, and in many senses the easiest to minimize, is the *relative backward error*, which we define in this section. The backward error of a computed solution y measures the distance from the actual system $Ax = b$ to the closest perturbed system $(A + \delta A)y = b + \delta b$ exactly satisfied by y. The distance is a function of the perturbations δA and δb and corresponds to the green circled segment in Figure 2.2. Rearranging the perturbed system produces $\delta Ay - \delta b = b - Ay$, so we expect a relationship between backward errors and the residual $r = b - Ay$.

The backward error can be *computed* given any vector y, unlike the forward error discussed later. We can measure directly backward errors.

Note that we never will equate backward error with measurement errors. Having a backward error smaller than measurement or modeling error validates results *after* computation. Too often, however, measurement errors are used to justify sloppy numerical methods. Stating that a solution is "as good as the data deserves" implies that you consider the data you have, not the data you want or data nearby that you call your own. Admittedly there always are engineering trade-offs guided by the expected accuracy and performance of algorithms. But rather than finding the fastest algorithm with hopefully acceptable error, we concentrate on

Figure 2.2: Backward errors measure the distance from (A, b) to $(A + \delta A, b + \delta b)$, circled here. The density plots show the backward errors in solving $Ax = b$ using LU factorization and partial pivoting **before refinement** for Chapter 4's single-precision, real test systems. The horizontal axis is an estimate of the problem's difficulty.



constructing a dependable, accurate algorithm that also is fast. We leave interpreting fuzzy blobs and blurry images to the Weekly World News [1].

2.2.2 Normwise backward errors

The first backward error in plotted in Figure 2.2 is the normwise backward error.
Definition 2.2.1: The normwise relative backward error is

$$\begin{aligned} \text{nberr}(A, y, b) = \min_{\delta A, \delta b, \epsilon} & \epsilon \\ \text{such that } & (A + \delta A)y = b + \delta b, \\ & \|\delta A\|_\infty \leq \epsilon \|A\|_\infty, \text{ and} \\ & \|\delta b\|_\infty \leq \epsilon \|b\|_\infty. \end{aligned} \tag{2.2.1}$$

Definition 2.2.1 is a convex optimization problem with a closed-form solution. The derivation requires extra machinery, specifically vector duals and derivatives of norms, that carry us too far afield. Rigal and Gaches [88] provide a first-principles derivation, and Higham [59] provides an *ansatz* result. Both produce the same solution. First, define the diagonal scaling matrix

$$D_{\text{nberr}} = (\|A\|_\infty \|y\|_\infty + \|b\|_\infty) I = D_{\|A\|_\infty \|y\|_\infty + \|b\|_\infty}.$$

We frequently will use the notation $D_x = xI$ or $D_v = \text{diag}(v)$ to identify scaling matrices with scalar or vector quantities x and v. These scaling matrices will express *scaled* ∞*-norms*. With the scaling matrix D_{nberr},

Theorem 2.2.1:

$$\text{nberr}(A, y, b) = \frac{\|r\|_\infty}{\|A\|_\infty \|y\|_\infty + \|b\|_\infty} = \|D_{\text{nberr}}^{-1} r\|_\infty. \tag{2.2.2}$$

Higham [59] provides a more general form that measures δA and δb against arbitrary, non-negative matrices E and f. The result is the same after substituting E and f for A and b in the denominator. We will not need the general normwise form.

Note that the normwise backward error is sensitive to both row and column scaling. We can multiply by diagonal matrices D_v and D_w to produce a "new" system $A_s x_s = (D_v A D_w)(D_w^{-1} x) = (D_v b) = b_s$ and "new" computed solution $y_s = D_w^{-1} y$. The normwise backward error can be arbitrarily different for the scaled and original systems. In a sense, the normwise backward error considers only the largest entries in A and b, and scaling both rows and columns can change which entries are the largest.

The dependency on row scaling is somewhat expected; the normwise backward error measures $\|r\|_\infty$, and that depends on the row scaling. But the dependence on column scaling is unfortunate. Mathematically, r is invariant to column scaling. And the LU factorization used to solve $Ax = b$ also is column-scaling invariant (with two minor caveats). A measurement for the error after using an LU factorization to solve $Ax = b$ should respect that invariance. We use column-scaling invariance frequently in later chapters to estimate scaled norms and will suggest a column-relative error to replace the normwise relative backward error shortly.

The two minor caveats do not remove our need for a column-invariant measure. The first caveat mentioned earlier is that an arbitrary column scaling could introduce over- or underflow errors when storing the scaled A_s. In practice, the scaling factors can be limited to ensure this never happens. The second minor caveat is that LU factorization could use accelerated, Strassen-like [97, 38] matrix multiplication, where the errors are not scaling-invariant. Our analytical use of column scaling to achieve componentwise results applies the scaling *implicitly*, so Strassen-like methods do not affect our key results. Assumptions built into Chapter 4's entry growth estimates could be violated with accelerated matrix multiplication. We will discuss this possibility further in Chapter 4. Demmel et al. [36] also discusses scaling and Strassen-like algorithms.

2.2.3 Componentwise backward errors

Unlike the normwise error above, the more general form of expressing the *componentwise relative backward error* will prove very useful. The general componentwise relative backward error measures each component of δA and δb against non-negative E and f:

Definition 2.2.2: The general componentwise relative backward error of a computed solution y to $Ax = b$ measured against non-negative matrix E and non-negative vector f is

$$\begin{aligned} \operatorname{cberr}_{E,f}(A, y, b) = \min_{\delta A, \delta b, \epsilon} & \epsilon \\ \text{such that } & (A + \delta A)y = b + \delta b, \\ & |\delta A| \leq \epsilon E, \text{ and} \\ & |\delta b| \leq \epsilon f. \end{aligned} \tag{2.2.3}$$

Oettli and Prager [80] provide the original, clear proof of the following result, and again Higham [59] provides an *ansatz* result. This is a linear optimization problem that can be solved by standard techniques [19] to prove

Theorem 2.2.2:

$$\begin{aligned} \operatorname{cberr}_{E,f}(A, y, b) &= \max_i \frac{|r(i)|}{(E\,|y| + f)(i)} \\ &= \left\| \frac{r}{E\,|y| + f} \right\|_\infty = \left\| D^{-1}_{(E\,|y|+f)} r \right\|_\infty, \end{aligned} \tag{2.2.4}$$

where D_v represents a diagonal matrix with vector v along its diagonal, and we define vector division as componentwise division.

Here and in other computations related to errors, we define $0/0 = 0$. Section 3.9 explains why this is the appropriate choice within solving $Ax = b$.

Specializing Theorem 2.2.2 to measure perturbations relative to $|A|$ and $|b|$ provides the componentwise backward error used throughout later experiments and results,

Corollary 2.2.3:

$$\begin{aligned} \operatorname{cberr}(A, y, b) &= \left\| \frac{r}{|A|\,|y| + |b|} \right\|_\infty = \left\| D^{-1}_{(|A|\,|y|+|b|)} r \right\|_\infty \\ &= \left\| D^{-1}_{\operatorname{cberr}} r \right\|_\infty. \end{aligned} \tag{2.2.5}$$

2.2.4 Columnwise backward error

We introduce another error measurement with great trepidation. Two measurement choices often are one too many. The normwise backward error's dependence on the column scaling makes it less useful for exploring the column-scaling-invariant solution's behavior. Also, as we will see in Chapter 5, normwise error measurements are not very useful for sparse systems.

Scaling the columns of A alters the magnitude of components of the solution by

$$(AD)(D^{-1}x) = b.$$

A measure of error invariant to column scaling reflects properties of the standard LU factorization algorithm and should provide information regardless of the relative component

sizes in the solution. To handle sparse systems well, we want an error measure that allows for perturbations in A while also providing useful information about smallest components of y. We also would like to avoid artificial thresholds proposed for other backward errors in sparse systems [9].

To define an appropriate columnwise backward error, recall the generic componentwise backward error from Equation (2.2.4),

$$\operatorname{cberr}_{E,f}(A, y, b) = \max_i \frac{|r(i)|}{(E\,|y| + f)(i)} = \left\| D^{-1}_{(E\,|y|+f)} r \right\|_\infty .$$

We chose $f = |b|$ as in cberr but pick a matrix E that allows more perturbations than does $|A|$.

To maintain column-scaling invariance, each column of E must be determined only by the corresponding column of A; the entries $E(:, j)$ are a function of $A(:, j)$ only. The simplest choice, and the path we follow, is to replicate a single scalar across each column of E. Obeying the analyst's penchant for sprinkling norms everywhere, we let $E(:, j) = \|A(:, j)\|_\infty = \max |A(:, j)|$.

Definition 2.2.3: The *columnwise backward error* $\operatorname{colberr}(A, y, b)$ is defined through Figure 2.2.2 with the matrices E and f where

$$E(i, j) = \max_k |A(k, j)| \quad \text{and} \quad f = |b|.$$

The above definition of E is the same as $E = 1_r \max |A|$.[1]

To compute the columnwise backward error,

Corollary 2.2.4:

$$\begin{aligned} \operatorname{colberr}(A, y, b) &= \left\| \frac{r}{(1_r \max |A|)\,|y| + |b|} \right\|_\infty = \left\| D^{-1}_{(1_r \max |A|)\,|y|+|b|} r \right\|_\infty \\ &= \left\| D^{-1}_{\operatorname{colberr}} r \right\|_\infty . \end{aligned} \tag{2.2.6}$$

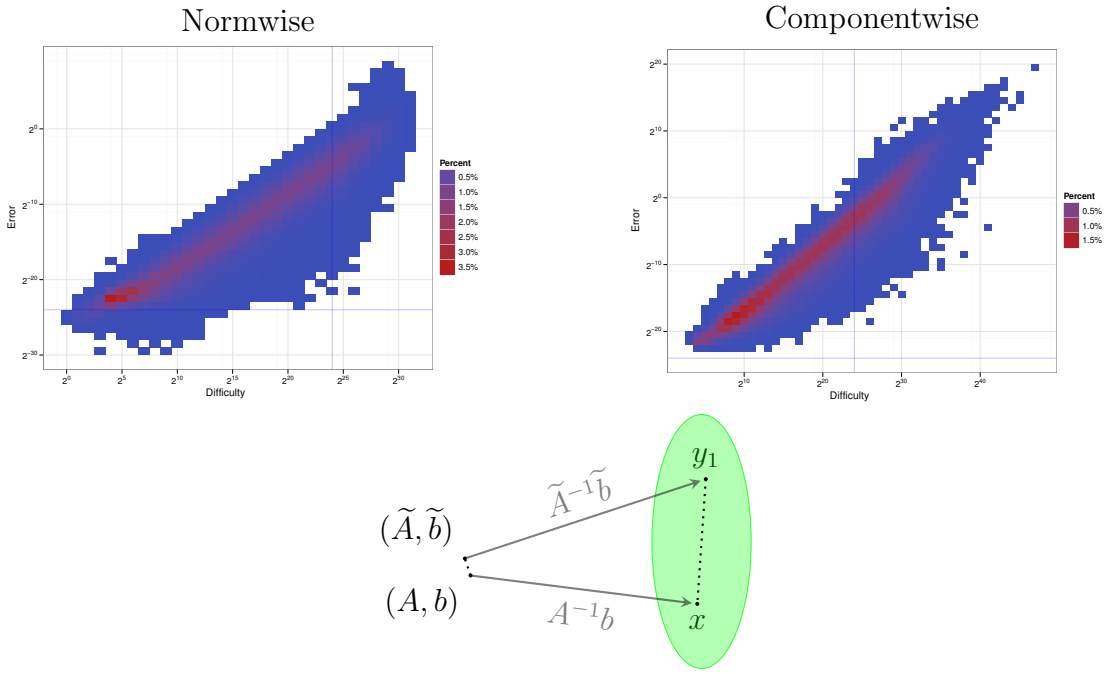
We leave $(1_r \max |A|)\,|y|$ unsimplified to keep it dimensionally commensurate with $|b|$. An implementation in a language that permits adding scalars to every component of vectors can simplify the term to $(\max |A|)|y|$. This columnwise backward error is not comparable with the normwise backward error.

2.2.5 Forward errors

The forward errors measure the error $e = y - x$, the distance between the computed solution y and the true solution x. Generally, these errors cannot be computed because they require knowledge of the exact solution. Outside testing situations like this thesis, there is little point in computing a solution when you have the true solution. Forward errors are simple to describe and understand, see Figure 2.3.

[1] In Octave, E = **ones**(**size**(A,1),1) ∗ **max**(**abs**(A)).

Figure 2.3: Forward errors measure the distance from x to $y = x + e$, circled here. The density plots show the backward errors in solving $Ax = b$ using LU decomposition and partial pivoting **before refinement** for Chapter 4's single-precision, real test systems. The horizontal axis is an estimate of the problem's difficulty.



Measuring the forward error against the true solution x is useful for analysis where the solution can remain implicit. Programs typically lack the imagination for handling implicit quantities and are limited to explicit quantities like the computed solution y.

So we are faced with the following four ways to measure the error e:

- normwise with respect to y,

$$\text{nferr}(x, y) = \|D^{-1}_{\|y\|_\infty} e\|_\infty \equiv \|D^{-1}_{\text{nferr},y} e\|_\infty; \tag{2.2.7}$$

- normwise with respect to x,

$$\text{nferr}_x(x, y) = \|D^{-1}_{\|x\|_\infty} e\|_\infty \equiv \|D^{-1}_{\text{nferr},x} e\|_\infty; \tag{2.2.8}$$

- componentwise with respect to y,

$$\text{cferr}(x, y) = \|D^{-1}_{|y|} e\|_\infty \equiv \|D^{-1}_{\text{cferr},y} e\|_\infty; \text{ and} \tag{2.2.9}$$

- componentwise with respect to x,

$$\text{cferr}(x, y) = \|D^{-1}_{|x|} e\|_\infty \equiv \|D^{-1}_{\text{cferr},x} e\|_\infty. \tag{2.2.10}$$

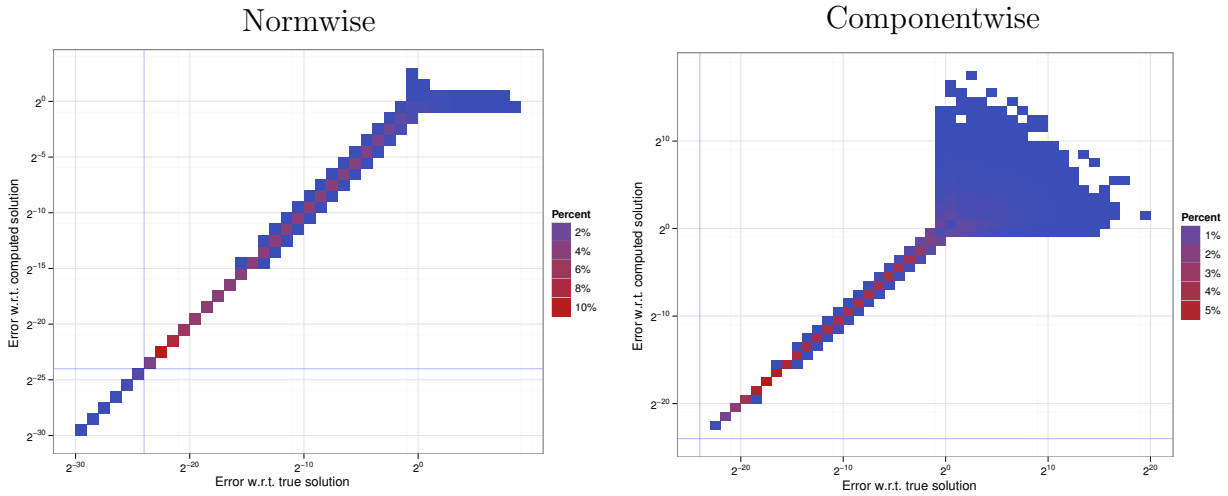
Because the end user of our algorithm has no recourse to the true solution x, our experimental results will use the definitions with respect to y from Equation (2.2.7) and Equation (2.2.9). The analysis, however, will consider convergence to the true solution x and measure against x using Equation (2.2.8) and Equation (2.2.10). Measuring with respect to x differs significantly only when the scaling matrix has diagonal entries near zero. For the normwise error, there is no important difference for our test sets as verified in Figure 2.4.

For the componentwise error, however, the difference can be huge. Many algorithms, including ours, tend to produce its largest componentwise errors in the smallest components of x. Our test generator never produces a zero entry in x, but our algorithm applied to the test set occasionally computes y vectors with zero entries. Reporting a small componentwise error on a vector with zero entries may mislead users, especially those trying to make sense of results in a hurry. We will always be clear about the forward error being evaluated.

2.3 Estimating the forward error and a system's conditioning

Backward errors are computable from the data in hand: A, y, and b. Forward errors involve the unknown true solution x. We use a relationship between backward and forward errors to bound the forward error. That relationship, which depends on the condition number, also measures the "difficulty" of solving a given system accurately. Condition numbers

Figure 2.4: Initial errors measured with respect to the true solution x (vertical axis) or the final computed solution y (horizontal) show little difference for normwise errors on our test set.



summarize the worst-case sensitivity of the solution in response to perturbations of the problem. Rounding errors serve as perturbations, so a very sensitive problem is deemed more difficult.

Recall that a backward error measures a perturbation to the data, and a forward error measures a perturbation to the solution. The sensitivity of the solution to perturbations in the problem is that problem's conditioning and is summarized in a scalar *condition number*. This section derives upper bounds on various condition numbers. Condition numbers depend on the norms used for the system's domain (x) and range (r).

Nothing presented in this section is fundamentally new. Condition numbers related to the normwise forward error are well-described in Higham [59], Demmel [34], Golub and Van Loan [54]. Our presentation here unifies results for normwise and componentwise forward error but does not prove that our expressions are the true condition numbers. The "condition numbers" derived are upper-bounds. We do not prove here that the upper bounds are achievable. The referenced texts provide full derivations for conditioning relating to the normwise forward error.

The condition number relationship always holds regardless of how a solution has been obtained. Unfortunately, the forward error estimate derived from the condition number can be very pessimistic. The over-estimate combined with the expense of estimating condition numbers is one reason why Chapter 4's refinement algorithm avoids condition estimation.

For our uses an upper-bound of the following form suffices, where B is the backward error and F is the forward error:

$$\text{forward error} \leq \text{condition number relating } B \text{ and } F \times \text{backward error.}$$

Assume throughout that all quantities are invertible. Let D_F be the scaling factor for the forward error $\|D_F^{-1}e\|_\infty$ with $e = y - x$. Let D_B be the scaling factor for the backward error $\|D_B^{-1}r\|_\infty$. Then the expressions take the form

$$\|D_F^{-1}e\|_\infty \leq \text{condition number} \times \|D_B^{-1}r\|_\infty.$$

Theorem 2.3.1: Let y be a computed solution to the system $Ax = b$ with error $e = y - x$ and residual $r = b - Ay$. Assume A is invertible. If scaling matrices D_F and D_B define a forward error measure $\|D_F^{-1}e\|$ and a backward error measure $\|D_B^{-1}r\|$, respectively, then for any consistent matrix and vector norms,

$$\|D_F^{-1}e\| \leq \|D_F^{-1}A^{-1}D_B\| \, \|D_B^{-1}r\|. \tag{2.3.1}$$

Proof. Expressing the backward error $(A+\delta A)y = b+\delta b$ in terms of the residual, $\delta Ay - \delta b = r$, and substituting $Ax = b$ shows that $Ae = A(y - x) = \delta b - \delta Ay = -r$. We have assumed A is invertible, so $e = -A^{-1}r$. Multiplying both sides by D_F^{-1} and introducing $I = D_B\, D_B^{-1}$ leads to $D_F^{-1}e = -\left(D_F^{-1}A^{-1}D_B\right)\left(D_B^{-1}r\right)$. Taking norms proves Equation (2.3.1). □

We have not shown that the bound always is achievable, but treating this upper-bound as a condition number is conservative. With this breach of rigor, we define condition numbers for this thesis.

Definition 2.3.1: $\|D_F^{-1}A^{-1}D_B\|_\infty$ is the *condition number* of A with respect to the given forward and backward error measures.

Theorem 2.3.1 is generic, but we only use the ∞-norm form.

2.3.1 Conditioning of the normwise forward error

The most commonly used condition numbers relate to the normwise forward error. We reproduce the standard expressions from our general framework. The normwise forward error $\text{nferr}(x, y)$ is defined by $D_F = D_{\text{nferr},y} = \|y\|_\infty I$. Substituting $\|y\|_\infty I$ into Equation (2.3.1),

$$\text{nferr}(x, y) \leq \|A^{-1}D_B\|_\infty \cdot \frac{1}{\|y\|_\infty} \cdot \|D_B^{-1}r\|_\infty$$

To relate $\text{nferr}(x, y)$ to the normwise backward error $\text{nberr}(y)$, we substitute $D_B = D_{\text{nberr}} = \text{diag}(\|A\|_\infty \|y\|_\infty + \|b\|_\infty)$ to obtain

$$\text{nferr}(x, y) \leq \|A^{-1}\|_\infty \left(\|A\|_\infty + {}^{\|b\|_\infty}/_{\|y\|_\infty}\right) \cdot \text{nberr}(y) \tag{2.3.2}$$

To relate $\text{nferr}(x, y)$ to the general componentwise backward error $\text{cberr}_{E,f}(y)$, we substitute $D_B = \text{diag}(E\,|y| + f)$ to obtain

$$\text{nferr}(x, y) \leq \frac{\|A^{-1} \cdot (E\,|y| + f)\|_\infty}{\|y\|_\infty} \cdot \text{cberr}_{E,f}(y). \tag{2.3.3}$$

The condition number is the sensitivity as the perturbations bounded by $E \to 0$ and $f \to 0$. As stated, we are not being rigorous, so here we simply substitute x for y to obtain upper-bounds on the true condition numbers. But even with our sloppiness, Equation (2.3.2) provides the standard normwise-normwise condition number

$$\kappa(A, x, b) = \frac{\|A^{-1}\|_\infty \|b\|_\infty}{\|x\|_\infty} + \|A^{-1}\|_\infty \|A\|_\infty. \tag{2.3.4}$$

Taking advantage of the fact that $\|Mz\|_\infty = \||Mz|\|_\infty = \||M|z\|_\infty$ when $z \geq 0$, we replace A^{-1} with $|A^{-1}|$ in Equation (2.3.3) to obtain the standard normwise-componentwise condition number

$$\text{cond}(A, x, b) = \frac{\||A^{-1}| \ (|A| \, |x| + |b|)\|_\infty}{\|x\|_\infty}. \tag{2.3.5}$$

The literature cited above proves these actually *are* the relevant condition numbers and not just upper bounds. For the columnwise backward error, the normwise-columnwise condition number taken from Equation (2.3.3) is

$$\text{colcond}(A, x, b) = \frac{\|A^{-1} \cdot ((1_r \max |A|) \, |x| + |b|)\|_\infty}{\|x\|_\infty}. \tag{2.3.6}$$

Approximating to within a factor of two, the expressions simplify to the more commonly used condition numbers

$$\kappa(A) = \|A^{-1}\|_\infty \|A\|_\infty, \tag{2.3.7}$$

$$\text{cond}(A, x) = \frac{\||A^{-1}| \, |A| \, |x|\|_\infty}{\|x\|_\infty}, \text{ and} \tag{2.3.8}$$

$$\text{colcond}(A, x) = \frac{\|A^{-1} \cdot (1_r \max |A|) \, |x|\|_\infty}{\|x\|_\infty}. \tag{2.3.9}$$

We also use a form of $\text{cond}(A, x)$ that removes the dependency on x,

$$\text{cond}(A) = \text{cond}(A, 1_c) = \left\| |A^{-1}| \, |A| \right\|_\infty. \tag{2.3.10}$$

Given that we can at best approximate the condition numbers inexpensively and that Chapter 4's algorithm does not rely on condition number estimates, we use the simpler expressions $\kappa(A)$, $\text{cond}(A, x)$, and $\text{cond}(A)$ where possible.

Are these error bounds tight enough for practical use? Comparing the estimates with actual errors in Chapter 4's test cases, Figures 2.5 and 2.6 show that the median upper bound overestimation is 11× when using $\kappa(A)$ and 4.3× using $\text{cond}(A)$. The worst case overestimation on problems by factors of 9.5×10^5 and 3.4×10^5, respectively, suggest these bounds cause alarm when not warranted. Our data contains underestimates by at most a factor of two for problems with bounds below one, which is expected from approximating $\kappa(A, x, b)$

Figure 2.5: Examining $\text{nferr}(x, y) \leq \kappa(A) \cdot \text{nberr}(y)$ (before refinement): The median error over-estimation ratio when the bound is at most one is ≈ 11 or $\approx 2^{3.5}$. The worst case for problems with a bound below one is a ratio of $\approx 9.5 \times 10^5$. There are no severe underestimates when the bound is below one; the only overestimates are within a factor of two (the vertical red line) and occur from approximating $\kappa(A, x, b)$ by $\kappa(A)$.

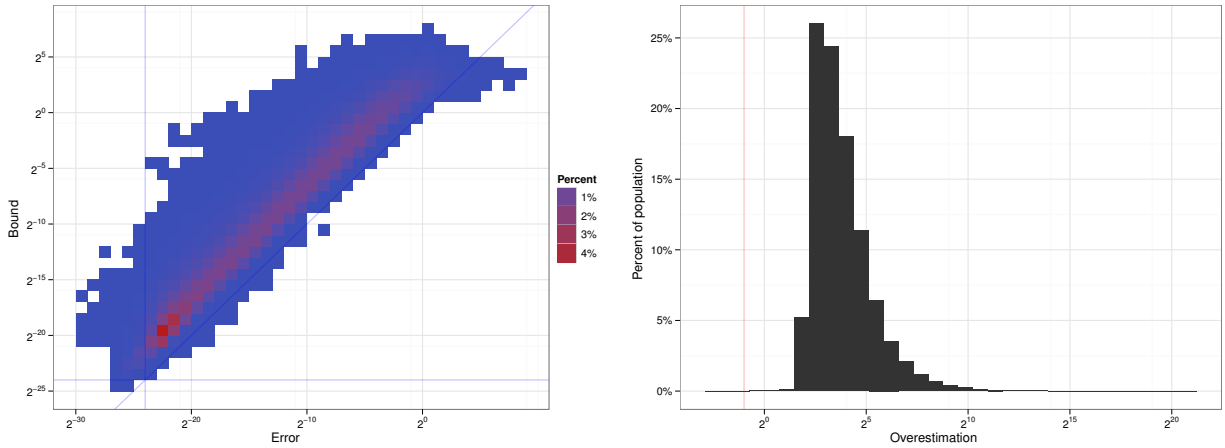
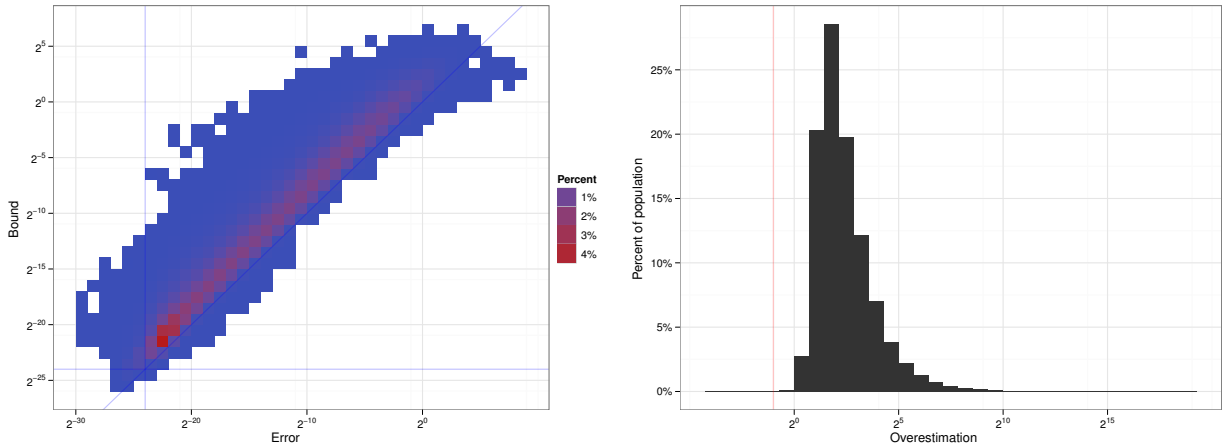


Figure 2.6: Examining $\text{nferr}(x, y) \leq \text{cond}(A) \cdot \text{cberr}(y)$ (before refinement): The median error over-estimation ratio of ≈ 4.3 or $\approx 2^{2.1}$. The worst case for those with bound below one is $\approx 3.4 \times 10^5$. No underestimates beyond a factor of two (the vertical red line) appear for problems with bound below one, and that factor is expected.



by $\kappa(A)$ and $\text{cond}(A, x, b)$ by $\text{cond}(A)$. The Higham and Tisseur [62] norm approximation algorithm with three test vectors does not contribute noticeably to the underestimation.

Later we use refinement to drive the errors below $n\varepsilon_w$ and do not need condition numbers for error estimates. LAPACK uses the relation with $\text{cond}(A, x, b)$ to provide error bounds in the expert $Ax = b$ drivers x`GESVX` [6]. Experience with users shows that providing too-large error bounds is counter productive. The error bound is ignored even when the bound could signal a true problem.

We use condition numbers to summarize the worst-case difficulty of solving a linear system. The final algorithm will not rely on computing estimates. Computing condition number estimates adds additional $O(N^2)$ working-precision operations per right-hand side.

2.3.2 Conditioning of the componentwise forward error

The generic framework above reproduces the standard results with respect to normwise forward error. We also can relate backward errors to the *componentwise* forward error. The resulting expression supports the componentwise condition number assumed in Demmel et al. [37, 35].

Again, we are not rigorous and simply substitute the appropriate scaling matrices even when some component of x or y is zero. Figure 2.4 borrows traditional results to demonstrate that our results are the condition numbers, but we do not offer an elementary proof.

Note that the condition number is a property of the system $Ax = b$ and does not depend on the computed solution y. So we use $\text{cferr}_x(x, y)$ (relative to x) rather than the $\text{cferr}(x, y)$ (relative to y) measurement in this analysis.

The componentwise forward error $\text{cferr}_x(x, y)$ is defined using $D_F = D_{\text{nferr},x} = D_{|x|}$. We ignore the possibility of zero components and assume the inverses of A and $D_{|x|}$ are defined. Substituting,

$$\text{cferr}_x(x, y) \leq \|D_{|x|}^{-1} A^{-1} D_B\|_\infty \, \|D_B^{-1} r\|_\infty$$

To relate the componentwise forward error $\text{cferr}_x(x, y)$ to the normwise backward error $\text{nberr}(y)$, we substitute $D_B = D_{\text{nberr}} = \text{diag}(\|A\|_\infty \|y\|_\infty + \|b\|_\infty)$ to obtain

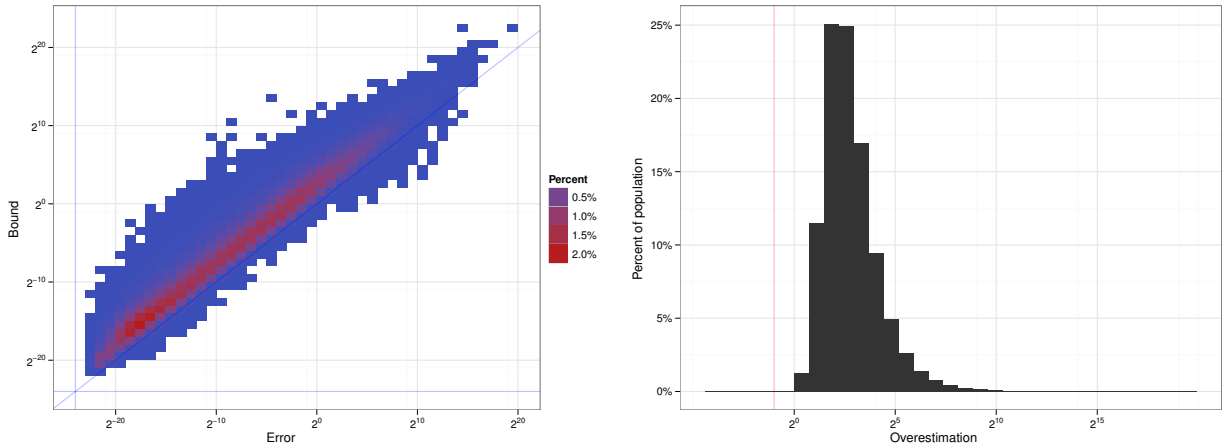
$$\text{cferr}_x(x, y) \leq \|D_{|x|}^{-1} A^{-1}\|_\infty \cdot (\|A\|_\infty \|y\|_\infty + \|b\|_\infty) \cdot \text{nberr}(y). \tag{2.3.11}$$

Substituting $D_B = \text{diag}(E\,|y| + f)$ for the general componentwise backward error $\text{cberr}_{E,f}(y)$ provides

$$\text{cferr}_x(x, y) \leq \left\| D_{|x|}^{-1} A^{-1} \cdot (E\,|y| + f) \right\|_\infty \cdot \text{cberr}_{E,f}(y). \tag{2.3.12}$$

A few substitutions and rearrangements produce condition numbers very similar to those of Section 2.3.1. First, we substitute $\|A\|_\infty \|y\|_\infty = \|AD_{\|y\|_\infty}\|_\infty$ and rearrange $D_{|x|}^{-1} A^{-1} = (AD_{|x|})^{-1}$. We specialize to $E = |A|$ and $f = |b|$. Following the chain

Figure 2.7: Examining $\text{cferr}_x(x, y) \leq \text{cond}(AD_{|x|}) \cdot \text{cberr}(y)$: The median over-estimate is a factor of ≈ 6 or $\approx 2^{2.6}$. The worst case for well-conditioned problems is $\approx 6.6 \times 10^6$. Not including the right-hand side in $\text{cond}(AD_{|x|})$ again induces underestimates by at most a factor of two (the vertical red line).



$|A|\,|x| = |A|\,|D_{|x|} \cdot \text{sign}(x)| = |A|\,D_{|x|}\,|\text{sign}(x)| = |AD_{|x|}|\,|\text{sign}(x)| = |AD_{|x|}|$ produces the componentwise condition numbers

$$\text{c}\kappa(A, x, b) = \|(AD_{|x|})^{-1}\|_\infty \cdot \left(\|AD_{\|x\|_\infty}\|_\infty + \|b\|_\infty\right), \tag{2.3.13}$$

$$\text{ccond}(A, x, b) = \left\|\left(|(AD_{|x|})^{-1}| \cdot |AD_{|x|}| \cdot |\text{sign}\, x|\right) + |(AD_{|x|})^{-1}|\,|b|\right\|_\infty, \text{ and} \tag{2.3.14}$$

$$\text{ccolcond}(A, x, b) = \left\|(AD_{|x|})^{-1} \cdot \left((1_r \max |AD_{|x|}|) + |b|\right)\right\|_\infty. \tag{2.3.15}$$

Note that $|\text{sign}(x)| = 1_c$, but using $\text{sign}(x)$ shows the similarity between solving $Ax = b$ for x and $(AD_{|x|})z = b$ for $z = \text{sign}(x)$.

And again, loosening our bound by a factor of two permits simplifying to

$$\text{c}\kappa(A, x) = \|(AD_{|x|})^{-1}\|_\infty \, \|AD_{|x|}\|_\infty, \tag{2.3.16}$$

$$\text{ccond}(A, x) = \left\||(AD_{|x|})^{-1}| \cdot |AD_{|x|}|\right\|_\infty = \text{cond}(AD_{|x|}), \text{ and} \tag{2.3.17}$$

$$\text{ccolcond}(A, x) = \left\||(AD_{|x|})^{-1}| \cdot (1_r \max |AD_{|x|}|\right\|_\infty. \tag{2.3.18}$$

The second, Equation (2.3.17), is equal to the expression $\text{cond}(AD_x)$ derived in Demmel et al. [37, 35] by considering the normwise error in $(AD_x)z = b$ where $z = D_x^{-1}x = 1_c$.

Figure 2.7 shows that $\text{cond}(AD_{|x|}) \cdot \text{cberr}(y)$ does provide an upper-bound for $\text{cferr}_x(x, y)$. But again the upper bound is too loose for practical use. The worst case on problems with bound below one overestimates the error by a factor of 6.6×10^6.

2.4 From normwise results to componentwise results

In the previous section, we see that the normwise and componentwise forward errors differ only in the scaling factor applied to the right of A. From an implementation point of view, that one difference leads to a generalization that makes supporting both measures simple.

Here we explain the reason in more detail and discuss why using the absolute value, $D_{|x|}$, may be preferable over D_x for analysis. Ultimately there is no serious difference between the two choices. Using $D_{|x|}$ will make a few results more obvious. We explain our choice here only because it differs from previously published results. Because our focus is analysis, we measure errors relative to x rather than the computed y. Also, we assume that norms are *absolute*, so $\|A\| = \| \, |A| \, \|$.

Consider $Ax = b$ where $b(i) = \text{sum}(A(i,:))$ and $x = 1_c$. Then $D_{\|x\|_\infty} = D_{|x|} = D_x = I_c$, an identity matrix commensurate with the columns of A. The normwise and componentwise errors in a computed y coincide,

$$\begin{aligned} \text{nferr}_x(x,y) &= \|D^{-1}_{\|x\|_\infty}(y-x)\| = \|y-x\| = \|D^{-1}_{|x|}(y-x)\| \\ &= \text{cferr}_x(x,y). \end{aligned} \tag{2.4.1}$$

This relationship holds for any system where $|x(i)| = 1$ and $b = Ax$. The resulting scaling matrix is diagonal with diagonal entries of absolute value 1, and the absolute norm annihilates any resulting sign change, real or complex.

Equation (2.4.1) suggests that we can obtain a componentwise error by transforming the solution for a general $Ax = b$. There are two obvious transformations, either $Ax = (AD_x)1_c = b$ or $Ax = (AD_{|x|})(\text{sign}(x)) = b$. For the latter, remember that $\|\text{sign}(x)\|_\infty = 1$. Demmel et al. [37, 35]'s work on iterative refinement applies the former without discussion.

This work prefers the latter, using $D_{|x|}$ to scale the results. Ultimately, there is no serious difference. Using the absolute value makes a few results obvious from the beginning.

- The resulting formulas are identical using either D_x or $D_{|x|}$, so the sign of a floating-point zero must not matter.
- The error is a measure of distance and does not depend on the direction from which the computed y approaches x.
- The scaling matrix $D_{|x|}$ passes through absolute values without change.

There is one important caveat related to common implementations of complex arithmetic. Most software uses an $|x| \approx \text{abs1}(x) = |\text{real}(x)| + |\text{imag}(x)|$ that is cheaper to compute than $|x| = \sqrt{\text{real}(x)^2 + \text{imag}(x)^2}$. This approximate absolute value squares the circle, so using abs1 within ∞-norms introduces a $\sqrt{2}$ approximation factor,

$$\frac{\|x\|_\infty}{\sqrt{2}} \le \sum_i \text{abs1}(x_i) \le \sqrt{2}\|x\|_\infty.$$

We absorb the factor of $\sqrt{2}$ into other arithmetic parameters; see Section 3.3.1.

2.5 A note on our abuse of "condition number"

Condition numbers summarize the worst case *achievable* sensitivity to perturbations of a problem. We have not shown the quantities called condition numbers above are achievable. Unlike Demmel et al. [39, 37, 35], we do not rely on condition number estimation for determining success. Our "condition numbers" are used only for plotting results and for discussion.

For our purposes, condition numbers represent the difficulty of a problem. Rounding errors within backwards-stable algorithms are represented by perturbations to the initial data. The condition number is the worst-case magnification of those perturbations into the solution's error. In Chapter 5, we see that well-conditioned systems are solved poorly when the algorithm introduces too-large perturbations through element growth, so we cannot rely solely on condition numbers to indicate potentially large errors.

Estimating the condition numbers that depend on the solution x require an estimation for each x. This becomes computationally expensive for multiple right-hand-sides and moderately sized systems [37]. Demmel et al. [31] conjectures that guaranteed condition estimation must be as computationally costly as matrix multiplication, and Demmel et al. [36] reduces matrix factorization algorithms to the complexity of matrix multiplication. These computational complexity results imply that *guaranteed* condition estimation may remain computationally expensive relative to the LU factorization used to solve the system. Applying the practical $O(n^2)$ algorithms [25, 62, 57] for each error we wish to return greatly increases the operational cost and memory traffic for iterative refinement.

Referring to conditioning for a problem requires nailing down two aspects: the perturbation structure and the measure used on the errors. There is much related work for different structured perturbations [56]. Here we consider only the simplest of perturbations, uniform perturbations across A and b. Higham and Higham [56] shows that respecting symmetry introduces at most a factor of two, and that is sufficient for plotting purposes. Chapter 5 introduces a simple structural element to capture the non-zero structure of L and U, but that is relatively unimportant.

How we measure the backward error perturbation and the forward error accuracy is quite important. This work focuses on the infinity norm, but there is more than just the norm. Numerically scaling the matrix A may produce a more accurate factorization $PD_rAD_c = LU$. The scaling essentially changes the norms on the domain and range. A solution *in that scaled norm* may be more accurate, but that is not the user's requested norm. Many packages, including LAPACK[6], report the condition number of the *scaled* matrix $A_s = D_rAD_c$. This condition number is slightly deceptive; it relates backward error perturbations to forward accuracy in the *scaled* norms and not the user's norm. Section 3.8 describes how we incorporate numerical scaling into our refinement algorithm.

Our dense cases used initially for demonstrating dependable refinement in Chapter 4 do not stress the difference between the scaled and unscaled cases. Some of the real-world sparse matrices in Chapter 5 are very sensitive to the numerical scaling of A. The conditioning

Error kind	Measure	Scaling matrix
Backward D_B	Normwise	$D_{\text{nberr}} = \text{diag}(\|A\|_\infty \|y\|_\infty + \|b\|_\infty)$
	Columnwise	$D_{\text{colberr}} = \text{diag}((1_r \max \|A\|) \|y\| + \|b\|)$
	Componentwise	$D_{\text{cberr}} = \text{diag}(\|A\| \|y\| + \|b\|)$
Forward D_F	Normwise	$D_{\text{nferr},x} = \text{diag}(\|x\|_\infty)$
	Componentwise	$D_{\text{cferr},x} = \text{diag}(\|x\|)$

Table 2.2: Error measures. The backward error measures scale the norm of the residual $r = b - Ay$ as $\|D_B^{-1} r\|_\infty$, and the forward error measures scale the norm of the error $e = y - x$ as $\|D_F^{-1} e\|_\infty$.

changes drastically, which does affect accuracy within the numerical factorization. More importantly for the final result, numerical scaling alters element growth. We revisit the issue in the later chapters.

2.6 Summary

Table 2.2 summarizes the different error measures, and Table 2.3 provides the condition numbers relating the different error measures. In later chapters, we use the columnwise-normwise condition number Equation (2.3.9) to depict the difficulty of the normwise forward error and the componentwise-componentwise condition number Equation (2.3.17) for the difficulty of the componentwise forward error. Chapter 4 finds an approximate relationship between $\text{cond}(A^{-1})$ and the difficulty of refining the componentwise backward error. Chapter 5 will explore the relationship between conditioning and numerical scaling further.

With these measures of errors and difficulty, we can set up the expectations for a dependable solver. For each error of interest, the following must hold:

- When delivering a result declared successful for a not-too-ill conditioned system (estimated condition number $< 1/\varepsilon_w$), the result must be accurate. The error must be at most a modest multiple of ε_w.
- Nearly all not-too-ill conditioned systems must produce accurate results.
- Any too-large error must be flagged, regardless of the system's conditioning.

The behavior of Chapter 4's refinement algorithm also assists in diagnosing *why* a large error occurs.

Backward error	Forward error	Condition number
Normwise	Normwise	$\kappa(A) = \|A^{-1}\|_\infty \|A\|_\infty$
	Componentwise	$\text{cond}(A,x) = \frac{\left\|\|A^{-1}\|\,\|A\|\,\|x\|\right\|_\infty}{\|x\|_\infty}$
Columnwise	**Normwise**	$\text{colcond}(A,x) = \frac{\left\|A^{-1}\cdot(1_r \max\|A\|)\,\|x\|\right\|_\infty}{\|x\|_\infty}$
	Componentwise	$\text{ccolcond}(A,x) = \left\|\|(AD_{\|x\|})^{-1}\| \cdot (1_r \max\|AD_{\|x\|}\|)\right\|_\infty$
Componentwise	Normwise	$\text{c}\kappa(A,x) = \|(AD_{\|x\|})^{-1}\|_\infty \|AD_{\|x\|}\|_\infty$
	Componentwise	$\text{ccond}(A,x) = \left\|\|(AD_{\|x\|})^{-1}\| \cdot \|AD_{\|x\|}\|\right\|_\infty = \text{cond}(AD_{\|x\|})$

Table 2.3: Condition numbers relating backward to forward errors. Bold denotes the condition number used for a forward error's difficulty in later error plots.

Chapter 3

Design and analysis of dependable iterative refinement

3.1 High-level algorithm

Iterative refinement of the square system of n equations $Ax = b$ is a very well-known version of Newton's method[18, 103, 76]. Listing 3.1 provides the top-level function implementing our iterative refinement algorithm in Octave. The remainder of this chapter fills in the details for the backwards and forward termination and success criteria, check_berr_criteria and check_ferr_criteria respectively. The following questions are addressed:

- What final accuracy can we expect in each error measure?
- Under what conditions can that accuracy be achieved?
- How can we detect a failure to reach the target forward error accuracy?

The short answers are that with extra precision within the residual computation and used for storing and computing with the intermediate solution, we can achieve accuracy that is a small multiple of the best possible for the input, working precision. The conditions are that the system $Ax = b$ not be too ill-conditioned, and we detect that through monitoring the backward error directly and the step size. The step size serves as an indirect measure of the forward error.

Section 3.3 establishes the error model used throughout the remaining analysis. Section 3.4 describes the general structure for all the recurrences. The backward and forward errors are covered by Sections 3.5 and 3.6. Section 3.7 defines the check_berr_criteria and check_ferr_criteria routines we use for extended-precision refinement. Section 3.8 covers numerical scaling for reducing somewhat artificial problems of measurement. Potential failure modes of our algorithm wrap up the chapter in Section 3.9.

```
function [yout, success] = itref (A, x1, b, MAXITER = 100)
### Function File [xout, success] = itref (A, x1, b[, MAXITER = 100])
### Refine an initial solution x1 to A*x=b. The result xout contains
### the refined solutions, and success is an opaque structure to be
### queried for the status of xout.
  nrhs = dim (x1,2);
  xout = zeros (dim (x1, 1), nrhs);
  for j = 1:nrhs,
    state = initial_refinement_state ();
    y{1} = widen (x1(:,j)); # Store solutions to extra precision ε_x.
    for i = 1:MAXITER,
      ## Compute this step's residual: r_i = b − Ay_i+δr_i.
      ## Intermediates are computed in extra precision ε_r.
      r{i} = widen_intermediates b − A*y{i};

      state = check_berr_criteria (state, i, r, A, y, b);
      if all_criteria_met (state), break; endif

      ## Compute the increment: (A+δA_i)δy_i = r_i.
      dy{i} = A \ r{i};

      state = check_ferr_criteria (state, i, y, dy);
      if all_criteria_met (state), break; endif

      ## Update the solution: y_{i+1} = y_i + dy_i+δy_i.
      y{i+1} = y{i} + dy{i};
    endwhile
    yout(:,j) = y{i}; # Round to working precision.
    success{j} = state_success (state);
  endfor
endfunction
```

Listing 3.1: Iterative refinement of $Ax = b$ in Octave. This chapter determines how the criteria functions work. The commands widen and widen_intermediates potentially increase the precision used. The underlined mathematical expressions are the error terms defined in Section 3.3.2.

```
function state = check_ferr_criteria (state, i, y, dy)
### For Wilkinson's iterative refinement algorithm.
  normdy = norm (dy{i});
  if normdy / state.prevnormdy > 0.5 || normdy / norm (y{i}, inf) <= 2*eps,
    state.done = 1;
  endif
  state.prevnormdy = normdy;
endfunction
```

Listing 3.2: For Wilkinson's iterative refinement, widen and check_berr_criteria from Listing 3.1 are empty operations, and check_ferr_criteria is defined below.

3.2 Related work

(This section is substantially derived from Demmel et al. [35].)

Extra precise iterative refinement was proposed in the 1960s. Bowdler et al. [18] presents the Algol programs that perform the LU factorization, the triangular solutions, and the iterative refinement using $\varepsilon_r = \varepsilon_w^2$. Listing 3.2 defines the support routine check_ferr_criteria determining termination in Bowdler et al. [18]'s refinement. Only the incremental step dy factors into termination in this algorithm.

In Bowdler et al. [18]'s basic solution method for computing the step size $Ady_i = r_i$, Wilkinson uses the Crout algorithm for LU factorization. The inner products are performed in extra precision. The Crout algorithm cannot easily exploit the memory hierarchy and is slow on current systems. As noted in Sections 3.5 and 3.6, higher precision inner products in the LU factorization only affect the rate of convergence and chance for success but not the limiting accuracy.

There is no error analysis in Bowdler et al. [18]. But Wilkinson [103] analyzes the convergence of the refinement procedure in the presence of round-off errors from a certain type of scaled fixed point arithmetic. Moler extends Wilkinson's analysis to floating point arithmetic. Moler accounts for the rounding errors in the refinement process when the working precision is ε_w and the residual computation is in precision ε_r, and derives the following error bound [76, Eq. (11)]:

$$\frac{\|y_i - x\|_\infty}{\|x\|_\infty} \leq [\sigma\kappa_\infty(A)\varepsilon_w]^i + \mu_1\varepsilon_w + \mu_2 n\kappa_\infty(A)\varepsilon_r \ ,$$

where σ, μ_1, and μ_2 are functions of the problem's dimension and condition number as well as refinement's precisions. Moler comments that "[if] A is not too badly conditioned" (meaning that $0 < \sigma\kappa_\infty(A)\varepsilon_w < 1$), the convergence will be dominated by the last two terms, and μ_1 and μ_2 are usually small. Furthermore, when ε_r is much smaller than ε_w (e.g., $\varepsilon_r \leq \varepsilon_w^2$), the limiting accuracy is dominated by the second term. When $\varepsilon_r \leq \varepsilon_w^2$, the stopping criterion he uses is $\|dy_i\|_\infty \leq \varepsilon_w\|y_1\|_\infty$. As for the maximum number of iterations, he suggests using the value near $-\log_{10}\varepsilon_r \approx 16$.

The use of higher precision in computing x was first presented as an exercise in Stewart [96, p. 207-208]. Stewart suggests that if x is accumulated in higher and higher precision, say in $\varepsilon_w, \varepsilon_w^2, \varepsilon_w^3, \ldots$ precisions, the residual will get progressively smaller. Eventually the iteration will give a solution with any desired accuracy. Kiełbasiński [66] proposes an algorithm called binary cascade iterative refinement. In this algorithm, GEPP and the first triangular solve for y_0 are performed in a base precision. Then during iterative refinement, both r_i and y_{i+1} are computed in increasing precision. Furthermore, the correction dy_i is also computed in increasing precision by using the same increasing-precision iterative refinement process. Kiełbasiński [66] "cascades" the precisions recursively. Kiełbasiński analyzes the algorithm and shows that with a prescribed accuracy for x, you can choose a maximum precision required to stop the iteration. This algorithm requires arbitrary precision arithmetic, often implemented in software and considered too slow for wide use. We are not aware of any computer program that implements this algorithm.

A very different approach towards guaranteeing accuracy of a solution is to use interval arithmetic techniques [92, 93]. Interval techniques provide guaranteed bounds on a solution's error. However, intervals alone do not provide a more accurate solution. Intervals indicate when a solution needs improving and could guide application of extra precision. We will not consider interval algorithms further, although they are an interesting approach. We do not have a portable and efficient interval BLAS implementation and so cannot fairly compare our estimates with an interval-based algorithm.

Björck [17] surveys the iterative refinement for linear systems and least-squares problems, including error estimates using working precision or extra precision in residual computation. Higham [59] gives a detailed summary of various iterative refinement schemes which have appeared through history. Higham also provides estimates of the limiting normwise and componentwise error. The estimates are not intended for computation but rather to provide intuition on iterative refinement's behavior. The estimates involve quantities like $\||A^{-1}| \cdot |A| \cdot |x|\|_\infty$, which we encounter again in Section 3.5.

Until recently, extra precise iterative refinement was not adopted in such standard libraries as LINPACK [41] and later LAPACK [6] because there was no portable way to implement extra precision when the working precision was already the highest precision supported by the compiler. Therefore, the LAPACK expert driver routines x`GESVX` and support routine x`GERFS` provided only the working precision iterative refinement routines ($\varepsilon_r = \varepsilon_w$). Since iterative refinement can always ensure backward stability, even in working precision [59, Theorem 12.3], the LAPACK refinement routines use the componentwise backward error in the stopping criteria. Listing 3.3 provides details on LAPACK's working-precision stopping criteria.

Demmel et al. [35] describes LAPACK's newer, extra-precise refinement algorithms, x`GERFSX`. The x`GERFSX` algorithm does not exactly fit into Listing 3.1's framework. The logic is more convoluted and separates the normwise (`x-state`) and componentwise (`z-state`) considerations; see Figure 3.1. There is a somewhat artificial "unstable" stage for the componentwise solution we replace with more well-understood conditions on the backward error for all forward errors. Also, the x`GERFSX` routines use extra precision in carrying the solution

```
function state = (state, i, r, A, y, b)
### For LAPACK's xGERFS iterative refinement algorithm.
  ## Compute the componentwise backward error.
  safe1 = (size (A, 1) + 1) * realmin;
  safe2 = safe1 / eps;
  numer = abs (r);
  denom = abs (A) * abs (y) + abs (b);
  mask = denom < safe2;
  denom(mask) += safe1;
  numer(mask) += safe1;
  cberr = max (numer ./ denom);
  ## Terminate if the error stops decreasing or is tiny.
  if cberr / state.prevcberr > 0.5 || cberr <= eps,
    state.done = 1;
  endif
  state.prevcberr = cberr;
endfunction
```

Listing 3.3: For LAPACK's iterative refinement routines xGERFS used in the expert drivers xGESVX, widen and check_ferr_criteria from Listing 3.1 are empty operations, and check_berr_criteria is defined below.

only when an estimated condition number κ_s times the ratio of largest to smallest magnitude entries in the solution is at least $1/n\varepsilon_w$.

Our analysis and algorithm below simplify this logic considerably. None of the forward errors are considered "stable" until the backward error has fallen to $O(\varepsilon_w^2)$. The solution always is carried to extra precision. The error estimate in Demmel et al. [35] is reduced to a binary indicator of successfully delivering a small error or not. And the final condition estimations not shown in Figure 3.1 are replaced the final convergence criteria and a rough estimate of the worst-case componentwise forward error conditioning. The remainder of this chapter details the new algorithm.

3.3 Error model

3.3.1 Floating-point error model

The higher-level error model is based on a standard floating-point relative error model as in Demmel [34], Higham [59]. Each arithmetic operation produces a result that has a small relative error, for example

$$a \oplus b = (a + b)(1 + \delta),$$

where $|\delta| \leq \varepsilon$. Here ε characterizes the precision of the floating-point result. Table 3.1 provides common precision values. We assume that underflow is not a significant contributor to any results; see further discussion in Section 3.9.

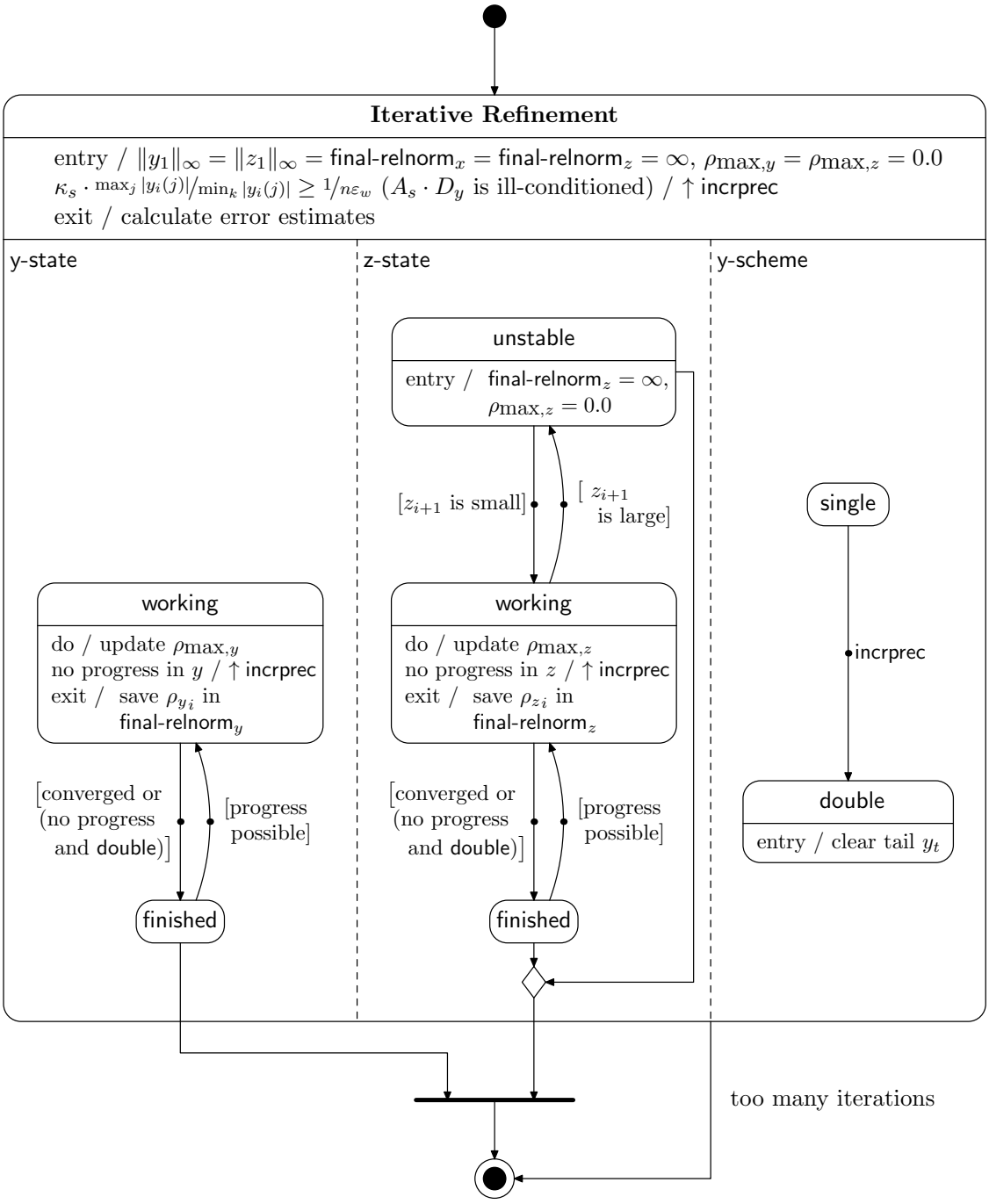


Figure 3.1: Overall statechart in UML 1.4 notation [79] for the algorithm in LAPACK's xGERFSX. Here, $dz_i = dy_i/y_i$ with some care to compute $0/0 = 0$. The ratios ρ_x and ρ_z track $\|dy_i\|_\infty/\|dy_{i-1}\|_\infty$ and $\|dz_i\|_\infty/\|dz_{i-1}\|_\infty$, respectively. If either ratio is above a pre-set value (0.5 or 0.9), the iteration has stopped making progress. If either value $\|dy_i\|_\infty$ or $\|dz_i\|_\infty$ falls below $n\varepsilon_w$, the iteration has converged. The final-relnorm variables produce an error estimate.

Name	Symbol	Value
IEEE-754 single (bin32)	ε_s	$2^{-24} \approx 6 \times 10^{-8}$
IEEE-754 double (bin64)	ε_d	$2^{-53} \approx 1.1 \times 10^{-15}$
Intel 80-bit	ε_{de}	$2^{-64} \approx 5.4 \times 10^{-20}$
doubled double	ε_{dd}	$2^{-106} \approx 1.2 \times 10^{-32}$
IEEE-754 quadruple (bin128)	ε_d	$2^{-113} \approx 9.6 \times 10^{-35}$

Table 3.1: Common floating-point precisions. Note that the doubled double precision does not round to the representable number nearest the true result. Modeling the precision as 2^{-106} suffices for our uses of some precision at most ε_d^2.

We use subscripts to denote the different precisions in Table 3.1 as well as different uses. Each of the matrix and vector operations in Listing 3.1 can operate at a different precision internally. We call the storage format of the input and output solutions the *working precision* ε_w. The solution of $Ady_i = r_i$ is performed in the factorization precision ε_f. The residual $r_i = b - Ay_i$ is computed internally in precision ε_r before being stored to ε_w. The intermediate solution is carried to precision ε_x. When appropriate, the entire intermediate solution is used in the residual calculation and not just the solution rounded to the working precision.

The doubled double precision is a software emulation of a type with precision $\leq \varepsilon_d^2$. While not a good choice for general-purpose computations, doubled double works well within bulk linear algebra operations like the BLAS[43, 42]. See Hida et al. [55] for information about doubled double and quad double arithmetic.

These precision definitions need to be multiplied by $2\sqrt{2}$ for complex arithmetic. Otherwise, all the analysis below holds as well for complex arithmetic as for real arithmetic.

3.3.2 Matrix operation error model

The floating-point implementation of each of the residual, solve, and step increments in Listing 3.1 incur errors related to the precisions. We rely on the standard error analysis of each step, with the error terms underlined in Listing 3.1. To recap,

- the i^{th} residual is computed as $r_i = b - Ay_i + \delta r_i$ with error term δr_i,
- the i^{th} step is computed with backward error $(A + \delta A_i)dy_i = r_i$ with error term δA_i, and
- the $(i+1)^{\text{st}}$ solution is computed as $y_{i+1} = y_i + dy_i + \delta y_i$ with error term δy_i.

None of these error terms are new. All are derived in any linear algebra text. Assuming we solve a dense system $Ady_i = r_i$ with LU factorization,

$$|\delta r_i| \leq (1 + n_d)\varepsilon_r \left(|A|\,|y_i| + |b|\right) + \varepsilon_w |r|, \tag{3.3.1}$$

$$|\delta A_i| \leq 3n_d |L|\,|U| \leq p(n_d) g 1_r \max|A|, \text{ and} \tag{3.3.2}$$

$$|\delta y_i| \leq \varepsilon_x |y_i|. \tag{3.3.3}$$

Here $p(n_d)$ is a second-degree polynomial in a size parameter n_d, and g is the element growth factor to be described shortly. The element growth is defined against the maximum column entries to permit matrices with zero entries. Chapter 4 details the changes this relationship implies.

The n_d term is a slight twist on the size of the system, n. In nearly all cases, a factor of n causes a severe *overestimate* of the true error. We give ourselves a bit of freedom in picking a better dimensional estimate. For dense problems, $n_d = \sqrt{n}$ seems appropriate. For sparse problems, n_d is the square root of the longest operation chain length. The sparse residual calculation's n_d is one more than the largest number of entries in a row of A (the largest row degree), limited to at most $\sqrt{n}$. For common implementations of direct solvers for $Ady_i = r_i$, n_d is the largest row degree in the sum of the lower- and upper-triangular factors plus the largest row degree in A, again limited to at most $\sqrt{n}$. This analysis improves the $3n$ factor in the traditional version of Equation (3.3.2) to $2n$, but both are severe overestimates.

In many ways, the precision ε times a running accumulation of the same operations applied to their operands' absolute values would be a much better approximation than simply using either n or n_d [59]. While floating-point operations are "free" on many current processors, they are not *that* free. Also, the software infrastructure to support such running error analysis easily does not exist.

The element growth factor g measures the largest elements combined during the Schur complements during factorization. We follow common practice and estimate g given the final factors rather than monitoring g throughout factorization. With partial pivoting as in LAPACK, g is a function of the entries in U and A. For unrestricted partial pivoting, we consider the improved, column-relative entry growth measure recently adopted by LAPACK,

$$g = \max_j \frac{\max_i |U(i,j)|}{\max_i |A(i,j)|} \tag{3.3.4}$$

computed by comparing the maximum entries per column separately, then the maximum over all those. For dense problems, this is almost always less than 100. Examples exist with growth as large as 2^{n-1}, however[44].

When we go looking at oncoming traffic[1] and use restricted pivoting in later chapters, we generalize Equation 3.3.4 to include L. The expression becomes more complex, but still is

[1] At least one numerical analyst has suggested that someone is likely to be run over by a bus before encountering an unexpectedly large element growth.

easily computable:

$$g_c = \max_j \frac{(\max_{1\le k\le j} \max_i |L|(i,k)) \cdot (\max_i |U|(i,j))}{\max_i |A|(i,j)}. \tag{3.3.5}$$

This generalized element growth factor will indicate potential problems with Chapter 5's restricted pivoting strategies.

3.4 General structure of analysis

The analysis of refinement here focuses on what can happen when things go *well*, while also collecting the assumptions necessary. We begin with LAPACK x`GERFS` refinement algorithm and the XBLAS/LAPACK algorithm in Demmel et al. [37]. These two share a common recurrence structure we generalize. Later sections apply the recurrence to other quantities.

LAPACK's x`GERFS` routines monitor a perturbed componentwise backward error. The x`GERFS` routines terminate when one of the following conditions hold:

1. the componentwise backward error does not decrease by a factor of two,
2. the same error is at most ε_w, or
3. a fixed number of steps (five) have passed.

In this section, we are interested in the first condition. We model the first condition by the recurrence

$$\text{cberr}(y_i) \le 0.5 \cdot \text{cberr}(y_{i-1}) + \text{ error in computing cberr}(y_i). \tag{3.4.1}$$

We will expand the last error term in Section 3.5. For now, we are interested in the form of Equation (3.4.1). At each step, components of $|r_i|$ are perturbed by an additional $(n+1)\lambda(1+\varepsilon_w)$ beyond the rounding errors in computing r_i to protect against rounding small entries to zero [34], where λ is the underflow threshold. The source code only adds that perturbation when it might have an effect, but we can ignore that detail for the immediate analysis and its effect on block-diagonal systems

Demmel et al. [37] refines the forward error and monitors the per-iteration step sizes $\|D_{\|y\|_\infty i}^{-1} dy_i\|_\infty$ and $\|D_{|y|_i}^{-1} dy_i\|_\infty$. The routine applies essentially the same criteria as x`GERFS` to these quantities, targeting the forward error rather than the backward error. The extra-precise refinement routine requires that the next relative step sizes decrease to either 0.5 or 0.9 of the current relative step size for the conservative or aggressive settings, respectively. We model the aggressive, componentwise-relative step size by

$$\|D_{|y|_i}^{-1} dy_{i+1}\|_\infty \le 0.9 \cdot \|D_{|y|_i}^{-1} dy_i\|_\infty + \text{ error in computing } \|D_{|y|_i}^{-1} dy_i\|_\infty, \tag{3.4.2}$$

with the other settings and norms modeled similarly.

Each of these refinement algorithms follow a recurrence relationship,

$$z_{i+1} \leq \rho_i z_i + t_i, \tag{3.4.3}$$

where z_i denotes some measure, ρ_i is the ratio of (ideally) decrease, and t_i is a perturbation to each step. Additionally, Section 3.6 shows that an upper bound on forward error also follows this recurrence. This section examines this recurrence relationship to set expectations and determine assumptions. The goals are to determine the termination criteria related to t_i and dependability criteria from monitoring of measured ρ_i. We will apply these criteria to both forward and backward errors.

Later sections will substitute z_i with measures of errors and also dy_i once Section 3.6.1 relates dy_i to the forward error. The t_i is an upper bound on the error perturbations, typically proportional to $(n+1)\cdot(\varepsilon_x + \varepsilon_r)$ with a per-iteration measure of ill-scaling. An upper bound on ρ_i is related to the problem's conditioning and measures of ill-scaling.

During refinement, we have z_i, z_{i+1} in hand and can compute the upper bound of t_i. We do not have ρ_i and wish to avoid the expensive process of estimating the relevant condition number. Our first assumption is that ρ_i and t_i do not change much between iterations.

We are interested in a bound for z_i when rate of decrease slows and also in bounding ρ_i as a measure of difficulty. If the error z_i reaches an expected small multiple of t_i, we will assume the problem never crossed a region of high difficulty and declare success. This is in contrast to explicit condition estimation in our prior work [37].

This does induce a failure mode, see Section 3.9, where the error is "accidentally" too small. For the backward error, the higher-precision computation of the residual r renders this failure mode exceptionally rare, and even more careful computation can avoid it without explicitly perturbing the result by the error term in Equation (3.3.1) [34]. When inferring the forward error from the solution's change between iterations, however, this could cause acceptance of a solution with large error. Section 3.9 discusses this and other issues more thoroughly.

Assume the recurrence terminates with with $z_{i+1} = c_i z_i$. Refinement will enforce a bound on c_i by requiring some progress, and will terminate when $c_i \geq \underline{c}$. Then

$$c_i z_i \leq \rho_i z_i + t_i,$$

and

$$z_i \leq \frac{t_i}{c - \rho_i} \tag{3.4.4}$$

We expect z_i to converge to a multiple of t_i, and want to target small multiples of t_i by controlling c_i for a range of expected ρ_i values. Let $\rho_i \leq \bar{\rho}$ for all i, where we relate $\bar{\rho}$ to various condition measures and factorization errors in later sections. Also assume $\bar{\rho} < c_i$. Then

$$z_i \leq \frac{t_i}{c_i - \bar{\rho}},$$

Consider LAPACK's default backward error refinement in x`GESVX`. There z_i is the componentwise backward error. We can roughly model LAPACK's refinement, which targets $c \geq \underline{c} = 0.5$ and has $t_i \approx (n+1)\varepsilon_w$ unless there are tiny components in r_i, by

$$z_i \lessapprox \frac{(n+1)\varepsilon_w}{0.5 - \bar{\rho}}.$$

For systems with $\bar{\rho} \leq 0.25$, $z_i \leq 4(n+1)\varepsilon_w$ at termination. An appropriate $\bar{\rho}$ here is $\bar{\rho} \approx p(n_d)\varepsilon_w g \operatorname{cond}(A, x)$, so systems not too ill-conditioned, that do not experience too-great element growth, and are not massive should terminate with a small componentwise backward error.

If we want $z_i = t_i$ at convergence, c must be at least one, which opens the door to the error growing. If $z_i \leq 10t_i$, or within one decimal digit of the perturbation t_i, $c < 1$ for $\bar{\rho} < 0.9$. For one bit, $z_i \leq 2t_i$, and $c < 1$ for $\bar{\rho} > .5$. So there is an expected trade-off between accuracy we can achieve and the convergence ratio.

Consider the XBLAS/LAPACK [37] conservative ratio setting ($\underline{c} = 0.5$) and factor of 10 safety ratio in conditioning ($\bar{\rho} \leq .1$). Then $z_i \leq 2.5t_i$ when refinement stops. Pushing to $\underline{c} = 0.9$, $z_i \leq 1.25t_i$. In that case, z_i is a measure of dy_i. Section 3.6.1 related this step size to the forward error. Overall, our recurrence model matches well the results in our related prior work.

For a *lower* bound on $\bar{\rho}$ given quantities we have in-hand,

$$\hat{\rho} = \frac{z_{i+1} - t_i}{z_i} = c - \frac{t_i}{z_i} \leq \bar{\rho}. \tag{3.4.5}$$

Here c is computed directly as z_{i+1}/z_i. While a low value of $\hat{\rho}$ is not a guarantee that $\bar{\rho}$ itself is small, a large value before convergence alerts us to potential conditioning and accuracy problems. This is why larger values of c are considered risky.

3.5 Refining backward error

To establish such a recurrence for the backward error, chain together the effects of one step of refinement:

$$\begin{aligned} r_1 &= b - Ay_1 + \delta r_1, \\ (A + \delta A_1)dy_1 &= r_1, \\ y_2 &= y_1 + dy_1 + \delta y_2, \text{ and} \\ r_2 &= b - Ay_2 + \delta r_2. \end{aligned}$$

Each step simply shifts the indices, so we drop the step number from the subscripts.

Expanding r_2 yields

$$\begin{aligned} r_2 &= b - Ay_2 + \delta r_2 \\ &= b - A(y_1 + dy_1 + \delta y_2) + \delta r_2 \\ &= (b - Ay_1) - Ady_1 + \delta r_2 - A\delta y_2 \\ &= r_1 - \delta r_1 - A(A + \delta A_1)^{-1} r_1 + \delta r_2 - A\delta y_2 \\ &= \left(I - A(A + \delta A_1)^{-1}\right) r_1 - \delta r_1 + \delta r_2 - A\delta y_2 \\ &= \delta A_1 (A + \delta A_1)^{-1} r_1 - \delta r_1 + \delta r_2 - A\delta y_2. \end{aligned} \tag{3.5.1}$$

Let D_1 and D_2 be the scaling matrices for the backward error of interest. Introducing these into Equation (3.5.1) relates the steps' backward errors. Scaling,

$$\begin{aligned} D_2^{-1} r_2 = D_2^{-1} \delta A_1 (A + \delta A_1)^{-1} D_1 D_1^{-1} r_1 \\ - D_2^{-1} D_1 D_1^{-1} \delta r_1 + D_2^{-1} \delta r_2 - D_2^{-1} A \delta y_2. \end{aligned}$$

Taking norms provides a recurrence of the form in Equation (3.4.3),

$$\begin{aligned} \|D_2^{-1} r_2\|_\infty \le \|D_2^{-1} \delta A_1 (A + \delta A_1)^{-1} D_1\|_\infty \|D_1^{-1} r_1\|_\infty \\ + \|D_2^{-1} D_1\|_\infty \|D_1^{-1} \delta r_1\|_\infty + \|D_2^{-1} \delta r_2\|_\infty + \|D_2^{-1} A \delta y_2\|_\infty. \end{aligned}$$

Here $\rho_1 = \|D_2^{-1} \delta A_1 (A + \delta A_1)^{-1} D_1\|_\infty$ and $z_1 = \|D_2^{-1} D_1\|_\infty \|D_1^{-1} \delta r_1\|_\infty + \|D_2^{-1} \delta r_2\|_\infty + \|D_2^{-1} A \delta y_2\|_\infty$.

3.5.1 Establishing a limiting accuracy

The terms involving δr_1, δr_2, and δy_2 form t_i in Equation (3.4.3) and establish the lower bound for the terminal error $\|D_i^{-1} r_i\|_\infty$. Note that this limiting accuracy is for the *solution carried to precision* ε_x and not the solution returned in precision ε_w. Rounding to ε_w forces a backward error of at least ε_w, and that is the backward error the calling routine will see. The possibly extended-precision intermediate solution, however, permits us to achieve better than expected *forward* error.

We bound the terms involving δr_1 and δr_2 using Equation (3.3.1) and that $D_1^{-1}(|A|\,|y_i| + |b|) \le 1$ for all backward errors of interest,

$$\begin{aligned} \|D_i^{-1} \delta r_i\|_\infty &\le \|D_i^{-1} \left((n_d + 1)\varepsilon_r \left(|A|\,|y_i| + |b|\right) + \varepsilon_w |r_i|\right)\|_\infty \\ &\le (n_d + 1)\varepsilon_r + \varepsilon_w \|D_i^{-1} r_i\|_\infty \end{aligned} \tag{3.5.2}$$

The term involving the update error δy_2 we bound using Equation (3.3.3),

$$\|D_2^{-1} A \delta y_2\|_\infty \le \varepsilon_x \|D_2^{-1} A |y_2|\|_\infty.$$

Because D_2^{-1} and $|y_2|$ are non-negative, then,

$$\|D_2^{-1}A\delta y_2\|_\infty \leq \varepsilon_x \|D_2^{-1}|A||y_2|\|_\infty.$$

Using $Ay_2 = b - r_2 + \delta r_2$ and Equation (3.5.2) along with $\|D_i^{-1}b\|_\infty \leq 1$ for our backward error scaling factors,

$$\begin{aligned}\|D_2^{-1}A\delta y_2\|_\infty &\leq \varepsilon_x \|D_2^{-1}\left(|b| + |r_2| + |\delta r_2|\right)\|_\infty \\ &\leq \varepsilon_x \|D_2^{-1}b\|_\infty + \varepsilon_x \|D_2^{-1}r_2\|_\infty + \varepsilon_x \|D_2^{-1}\delta r_2\|_\infty \\ &\leq \varepsilon_x + (n_d+1)\varepsilon_r\varepsilon_x + (\varepsilon_x + \varepsilon_x\varepsilon_w)\|D_2^{-1}r_2\|_\infty \end{aligned} \tag{3.5.3}$$

Expanding the recurrence in Equation (3.4.3),

$$\begin{aligned}(1-\tilde{\varepsilon})\|D_2^{-1}r_2\|_\infty \leq \left(\|D_2^{-1}\delta A_1(A+\delta A_1)^{-1}D_1\|_\infty + \varepsilon_w\|D_2^{-1}D_1\|_\infty\right)\|D_1^{-1}r_1\|_\infty \\ +(n_d+1)(1+\|D_2^{-1}D_1\|_\infty)\varepsilon_r + \varepsilon_x + (n_d+1)\varepsilon_r\varepsilon_x\end{aligned}$$

where $\tilde{\varepsilon} = \varepsilon_w + \varepsilon_x + \varepsilon_x\varepsilon_w$. Using $\tilde{\varepsilon} \approx \varepsilon_w + \varepsilon_x$, expanding $(1-\tilde{\varepsilon})^{-1} \approx 1 + \tilde{\varepsilon}$, and ignoring second-order effects ($\varepsilon_x\varepsilon_r$) throughout yields the recurrence we will consider,

$$\begin{aligned}\|D_2^{-1}r_2\|_\infty \lessapprox \left((1+\tilde{\varepsilon})\|D_2^{-1}\delta A_1(A+\delta A_1)^{-1}D_1\|_\infty + \varepsilon_w\|D_2^{-1}D_1\|_\infty\right)\|D_1^{-1}r_1\|_\infty \\ +(n_d+1)(1+\|D_2^{-1}D_1\|_\infty)\varepsilon_r + \varepsilon_x\end{aligned} \tag{3.5.4}$$

We will return to the ratio

$$\rho_i = (1+\tilde{\varepsilon})\|D_{i-1}^{-1}\delta A_i(A+\delta A_i)^{-1}D_i\|_\infty + \varepsilon_w\|D_{i-1}^{-1}D_i\|_\infty \tag{3.5.5}$$

after examining the terminal backward error when refinement succeeds.

The t_i term determining the limiting accuracy then is

$$t_i = (n_d+1)(1+\|D_{i-1}^{-1}D_i\|_\infty)\varepsilon_r + \varepsilon_x. \tag{3.5.6}$$

Only the term $\|D_{i-1}^{-1}D_i\|_\infty$ depends on the current iteration. This term is a rough measure of ill-scaling encountered between iterations. Consider the normwise backward error,

$$\begin{aligned}\|D_2^{-1}D_1\|_\infty &= \frac{\|E\|_\infty\|y_1\|_\infty + \|b\|_\infty}{\|E\|_\infty\|y_2\|_\infty + \|b\|_\infty} \\ &\leq \frac{\|y_1\|_\infty}{\|y_2\|_\infty} + 1 \\ &\lessapprox \frac{\|dy_2\|_\infty}{\|y_2\|_\infty} + 2.\end{aligned}$$

If y_2 is somewhat close to a solution, we can expect $\|dy_2\|_\infty \ll \|y_2\|_\infty$. The componentwise and columnwise relationships are similar, so we approximate

$$t_i \approx 3(n_d + 1)\varepsilon_r + \varepsilon_x. \tag{3.5.7}$$

and monitor $\|D_{i-1}^{-1} D_i\|_\infty$ for potential failures. These do not occur in any of our tests, although they could occur when some component of $|A||x| + |b|$ is nearly zero.

Thus:

Theorem 3.5.1: For a given $\underline{c}$ and $\bar{\rho}$, the terminal backward error satisfies

$$\|D_i^{-1} r_i\|_\infty \leq (\underline{c} - \bar{\rho})^{-1} \left(3(n_d + 1)\varepsilon_r + \varepsilon_x\right)$$

so long as the following assumptions hold:

1. $\|D_{i-1}^{-1} D_i\|_\infty \leq 2$, and
2. the ratio ρ_i in Equation (3.5.5) stays less than one.

The first assumption is *measurable and testable* without significant computational cost. Monitoring $\|D_{i-1}^{-1} D_i\|_\infty$ requires storing an additional vector per solution the worst case, however. We infer violations of the second assumption from measuring $c_i = {}^{z_{i+1}}/_{z_i}$ during refinement. So long as c_i does not exceed $\underline{c}$ before z_{i+1} reaches the expected bound, we accept the answer. This does open the door to possible failure modes, discussed in Section 3.9.

Table 3.2 shows the backward errors we expect to achieve in our model starting from single precision and a system of dimension around 100. We assume that the assumptions of Theorem 3.5.1 hold. Note that *both* computation of the residual and carrying of the solution must be done in double precision to achieve a double-precision result. However, the factorization precision does not enter into the terminal precision. Our results match well with other results and experience [37, 58, 69], although often still overestimates.

3.5.2 Considering the ratio ρ_i between iterates

We at best can estimate the ratio from Equation (3.5.5),

$$\rho_i = (1 + \tilde{\varepsilon})\|D_2^{-1}\delta A_1 (A + \delta A_1)^{-1} D_1\|_\infty + \varepsilon_w \|D_2^{-1} D_1\|_\infty.$$

The limiting accuracy in Theorem 3.5.1 often is an overestimate, so the t_i/z_i term in Equation (3.4.5) will produce an underestimate of ρ_i. As described previously, we rely on monitoring c_i to infer when ρ_i becomes unacceptably large.

This section sets up an interpretation to help diagnose why iteration may stop prematurely. Assuming A is invertible,

$$\begin{aligned}
\|D_2^{-1}\delta A_1 (A + \delta A_1)^{-1} D_1\|_\infty &= \|D_2^{-1}\delta A_1 A^{-1} (I + \delta A_1 A^{-1})^{-1} D_1\|_\infty \\
&= \|D_2^{-1}\delta A_1 A^{-1} D_2 D_2^{-1} D_1 D_1^{-1} (I + \delta A_1 A^{-1})^{-1} D_1\|_\infty \\
&\leq \|D_2^{-1}\delta A_1 A^{-1} D_2\|_\infty \|D_2^{-1} D_1\|_\infty \|(I + D_1^{-1}\delta A_1 A^{-1} D_1^{-1})^{-1}\|_\infty.
\end{aligned}$$

Description	ε_r	ε_x	$\underline{c}$	$\bar{\rho}$	Bound
Single-precision	ε_s	ε_s	0.5	0.1	$85\varepsilon_s \approx$ 5.1e-6
	ε_s	ε_s	0.9	0.1	$42.5\varepsilon_s \approx$ 2.5e-6
Double residual	ε_d	ε_s	0.5	0.1	$2.5\varepsilon_s \approx$ 1.5e-7
	ε_d	ε_s	0.9	0.1	$1.25\varepsilon_s \approx$ 7.5e-8
Double both	ε_d	ε_d	0.5	0.1	$85\varepsilon_d \approx$ 9.4e-15
	ε_d	ε_d	0.9	0.1	$42.5\varepsilon_d \approx$ 4.7e-15

Table 3.2: Terminal precisions with $n_d = 10$ (so a system of dimension ≈ 100). Note that *both* $\varepsilon_r \le \varepsilon_w^2$ and $\varepsilon_x \le \varepsilon_w^2$ to achieve double precision: the residual must be computed in double precision, and the solution must be carried to double precision. The factorization precision does not directly influence the limiting accuracy but rather the conditions for reaching this limit.

We relate this to the problem's conditioning by using $|\delta A_1| \le p(n_d) g \varepsilon_f 1_r \max |A|$ and ignoring second-order effects in ε_f. Then

$$\|D_2^{-1}\delta A_1 (A + \delta A_1)^{-1} D_1\|_\infty \lessapprox \|D_2^{-1} D_1\|_\infty \cdot p(n_d) g \varepsilon_f \operatorname{ccolcond}(A^{-1}, \operatorname{diag} D_2). \tag{3.5.8}$$

With additional manipulation, we can replace the column-wise condition number ccolcond with others and absorb a factor of two to replace the c-prefixed condition numbers with their single-argument counterparts.

Ultimately, Equation (3.5.8) provides a few possible explanations when refinement fails to converge:

- The problem is too ill-conditioned around the current iterate.
- The factorization encounters large element growth.
- The backward error measure changes too drastically between iterations.
- The system is too large for the factorization precision.

The second and third reasons can be monitored, but failure can occur through a combination of all four reasons.

3.6 Refining forward error

The backward error is computable, but the forward error requires the unknown true solution and is not. This section shows that dy_i tracks an upper bound for the forward error. Then dy_i follows a recurrence we can analyze as in Section 3.4. Here D_i refers to the scale measuring the *forward* error. And unlike the backward error, the errors in the factorization, Equation (3.3.2),

will enter into the forward error. The bound from Equation (3.3.2) influences even the limiting accuracy.

Assuming convergence to a single solution given enough time,

$$\|D_i^{-1}e_i\|_\infty = \|D_i^{-1}(y_i - x)\|_\infty = \|D_i^{-1}\sum_{j=i}^{\infty} dy_j\|_\infty \leq \sum_{j=i}^{\infty}\|D_i^{-1}dy_j\|_\infty$$

If $\|D_i^{-1}dy_j\|_\infty$ decreases with ratio ρ,

$$\|D_i^{-1}e_i\|_\infty \leq \frac{1}{1-\rho}\|D_i^{-1}dy_i\|_\infty, \tag{3.6.1}$$

so we expect that $\|D_i^{-1}dy_i\|_\infty$ does track $\|D_i^{-1}e_i\|_\infty$.

A different analysis by Demmel et al. [35] infers that both decrease by nearly the same ratio, considering that

$$\begin{aligned}\|D_2^{-1}e_2\|_\infty &\leq \|D_2^{-1}(A + \underline{\delta A_2})\delta A_2 D_1\|_\infty \|D_1^{-1}e_1\|_\infty + ..., \text{ and}\\ \|D_2^{-1}dy_2\|_\infty &\leq \|D_2^{-1}(A + \underline{\delta A_1})\delta A_2 D_1\|_\infty \|D_1^{-1}dy_1\|_\infty + ...\end{aligned}$$

differ only by the perturbation to A.

3.6.1 Tracking the forward error by the step size

The error and step size are related through Equation (3.3.2) and a re-expression of Equation (3.3.1),

$$\begin{aligned}r_1 &= b - Ay_1 + \delta r_1 = Ax - Ay_1 + \delta r_1 = -Ae_1 + \delta r_1, \text{ and}\\ r_1 &= (A + \delta A_1)dy_1.\end{aligned}$$

Combining these,

$$e_1 = -A^{-1}(A + \delta A_1)dy_1 + A^{-1}\delta r_1.$$

Scaling and taking norms,

$$\|D_1^{-1}e_1\|_\infty \leq (1 + \|D_1^{-1}A^{-1}\delta A_1 D_1\|_\infty)\|D_1^{-1}dy_1\|_\infty + \|D_1^{-1}A^{-1}\delta r_1\|_\infty. \tag{3.6.2}$$

By Equation (3.6.2), the error is related to the step by a (hopefully small) multiplicative factor and an additive perturbation. The additive term satisfies

$$\begin{aligned}\|D_1^{-1}A^{-1}\delta r_1\|_\infty &\leq \left\|D_1^{-1}A^{-1}\left((n_d+1)\varepsilon_r(|A|\,|y_1| + |b|) + \varepsilon_w|r_1|\right)\right\|_\infty\\ &\leq (n_d+1)\varepsilon_r\left\|D_1^{-1}|A^{-1}|\left(|A|\,|y_1| + |b|\right)\right\|_\infty\\ &\quad + \varepsilon_w\|D_1^{-1}A^{-1}D_{B1}\|_\infty\|D_{B1}^{-1}r_1\|_\infty\\ &\leq (n_d+1)\varepsilon_r\,\mathrm{ccond}(A, y_1, b) + \varepsilon_w\cdot\mathrm{cond}\cdot\mathrm{berr},\end{aligned} \tag{3.6.3}$$

where *cond* and *berr* are determined by the arbitrary scaling D_{B1}.

The second term in Equation (3.6.3) should be at most ε_w *if the backward error converges.* This provides one of our tests for relying on estimating the forward error through the step size. The first term also should be at most ε_w unless the system is too ill-conditioned or too large.

For LU, $|\delta A_1| \leq p(n_d)g\varepsilon_f 1_r \max|A|$, and the ratio

$$\|D_1^{-1}A^{-1}\delta A_1 D_1\|_\infty \leq p(n^2)g\varepsilon_f \operatorname{colcond}(A)$$

If $p(n_d)g\varepsilon_f \operatorname{colcond}(A) \leq c/\varepsilon_w$, then

$$\|D_1^{-1}e_1\|_\infty \lessapprox (1+c)\|D_1^{-1}dy_1\|_\infty + 2\varepsilon_w \tag{3.6.4}$$

most likely is a large overestimate of $\|D_1^{-1}e_1\|_\infty$. Equation (3.6.4) shows that unless the problem is too ill-conditioned, $\|D_i^{-1}dy_i\|_\infty$ tracks $\|D_i^{-1}e_i\|_\infty$. If the step dy converges to $c'\varepsilon_w$, then the forward error converges to a slightly larger $(c'(1+c)+2)\varepsilon_w$.

3.6.2 Step size

Assuming the step size tracks the error well enough, we expand across iterations to link dy_2 to dy_1. The end result will not be as straight-forward as Theorem 3.5.1 and relies on A being invertible. Although the result could be phrased similarly, approximating for a few specific cases is more useful. We consider the following precision settings:

- fixed-precision refinement with $\varepsilon_w = \varepsilon_f = \varepsilon_x = \varepsilon_r$;
- extended-precision refinement with $\varepsilon_w = \varepsilon_f$, $\varepsilon_x, \varepsilon_r \leq \varepsilon_w^2$, and
- refinement after a reduced-precision factorization with $\varepsilon_f > \varepsilon_w$.

Expanding dy_2 yields

$$\begin{aligned}
dy_2 &= (A+\delta A_2)^{-1}r_2 \\
&= (A+\delta A_2)^{-1}(b - Ay_2 + \delta r_2) \\
&= (A+\delta A_2)^{-1}(b - A(y_1 + dy_1 + \delta y_2) + \delta r_2) \\
&= (A+\delta A_2)^{-1}(b - Ay_1 - Ady_1 - A\delta y_2 + \delta r_2) \\
&= (A+\delta A_2)^{-1}(r_1 - \delta r_1 - Ady_1 - A\delta y_2 + \delta r_2) \\
&= (A+\delta A_2)^{-1}((A+\delta A_1)dy_1 - Ady_1 - A\delta y_2 - \delta r_1 + \delta r_2) \\
&= (A+\delta A_2)^{-1}\delta A_1 dy_1 - (A+\delta A_2)^{-1}(A\delta y_2 + \delta r_2 - \delta r_1).
\end{aligned}$$

Following the now-common pattern, scaling by the appropriate error measure produces

$$\begin{aligned}
D_2^{-1}dy_2 = D_2^{-1}(A+\delta A_2)^{-1}\delta A_1 D_2 D_2^{-1} D_1 D_1^{-1} dy_1 - \\
D_2^{-1}(A+\delta A_2)^{-1}(A\delta y_2 + \delta r_2 - \delta r_1).
\end{aligned}$$

Note that we introduce a factor of $D_2^{-1}D_1$ to relate $D_2^{-1}dy_2$ to $D_1^{-1}dy_1$. Taking norms,

$$\begin{aligned}\|D_2^{-1}dy_2\|_\infty = \|D_2^{-1}(A+\delta A_2)^{-1}\delta A_1 D_2\|_\infty \|D_2^{-1}D_1\|_\infty \|D_1^{-1}dy_1\|_\infty + \\ \|D_2^{-1}(A+\delta A_2)^{-1}A\delta y_2\|_\infty + \\ \|D_2^{-1}(A+\delta A_2)^{-1}\delta r_2\|_\infty + \|D_2^{-1}(A+\delta A_2)^{-1}\delta r_1\|_\infty.\end{aligned} \tag{3.6.5}$$

Again, we consider the additive perturbations to determine t_i. First consider the contribution of δy_2,

$$\begin{aligned}\|D_2^{-1}(A+\delta A_2)^{-1}A\delta y_2\|_\infty &= \|D_2^{-1}(A+\delta A_2)^{-1}AD_2D_2^{-1}\delta y_2\|_\infty \\ &\le \|D_2^{-1}(A+\delta A_2)^{-1}AD_2\|_\infty \|D_2^{-1}\delta y_2\|_\infty \\ &\le \|D_2^{-1}(A+\delta A_2)^{-1}AD_2\|_\infty \cdot \varepsilon_x.\end{aligned}$$

For the first term,

$$\begin{aligned}D_2^{-1}(A+\delta A_2)^{-1}AD_2 &= D_2^{-1}(I+A^{-1}\delta A_2)^{-1}D_2 \\ &= (I+D_2^{-1}A^{-1}\delta A_2 D_2)^{-1}.\end{aligned}$$

Now we assume $D_2^{-1}A^{-1}\delta A_2 D_2$ is relatively small. Similar to the analysis in Higham [59, 58], let there be some matrix F_2 such that $|F_2| \le \varepsilon_f p(n_d) g |A^{-1}|\,|A|$ and $(I+A^{-1}\delta A_2)^{-1} = I+F_2$. Then

$$\|D_2^{-1}(A+\delta A_2)^{-1}AD_2\|_\infty \le 1 + \varepsilon_f p(n_d) g \operatorname{cond}(A, y_2).$$

Ultimately, the contribution of δy_2 is bounded by

$$\|D_2^{-1}(A+\delta A_2)^{-1}A\delta y_2\|_\infty \le (1+\varepsilon_f p(n_d) g \operatorname{cond}(A, y_2))\varepsilon_x. \tag{3.6.6}$$

One interpretation of Equation (3.6.6) is that carrying the solution to extra precision ε_x *may* help with forward accuracy when the system is ill-conditioned beyond the expected limits of ε_f.

The term depending on δr_2 is

$$\begin{aligned}\|D_2^{-1}(A+\delta A_2)^{-1}\delta r_2\|_\infty &= \|D_2^{-1}(I+A^{-1}\delta A_2)^{-1}A^{-1}\delta r_2\|_\infty \\ &= \|(I+D_2^{-1}A^{-1}\delta A_2 D_2)^{-1}D_2^{-1}A^{-1}\delta r_2\|_\infty \\ &\lessapprox (1+\|D_2^{-1}A^{-1}\delta A_2 D_2\|_\infty)\|D_2^{-1}A^{-1}\delta r_2\|_\infty \\ &\le (1+\|D_2^{-1}A^{-1}\delta A_2 D_2\|_\infty)\cdot \\ &\quad \left\|D_2^{-1}A^{-1}\left((n_d+1)\varepsilon_r(|A|\,|y_2|+|b|)+\varepsilon_w|r_2|\right)\right\|_\infty \\ &\le (1+\|D_2^{-1}A^{-1}\delta A_2 D_2\|_\infty)\left((n_d+1)\varepsilon_r \operatorname{ccond}(A, y_2)\right. \\ &\quad \left.+\varepsilon_w \cdot \operatorname{cond}_2 \cdot \operatorname{berr}_2\right). \\ &\le (1+\varepsilon_f p(n_d) g \operatorname{cond}(A, y_2))\left((n_d+1)\varepsilon_r \operatorname{ccond}(A, y_2)\right. \\ &\quad \left.+\varepsilon_w \cdot \operatorname{cond}_2 \cdot \operatorname{berr}_2\right).\end{aligned} \tag{3.6.7}$$

The condition number and backward error can be chosen arbitrarily as in Equation (3.6.3). When considering the normwise forward relative error, $\operatorname{ccond}(A, y_2)$ can be replaced by $c\kappa(A, y_2)$ by using normwise bounds for δr_2. The contribution of $\varepsilon_w \cdot \operatorname{cond}_2 \cdot \operatorname{berr}_2$ should be small if the backward error converges below ε_w as discussed above. However, that leaves two precisions with balancing condition numbers. If we assume $\varepsilon_r \leq \varepsilon_w^2$ and the condition numbers are $\leq 1/\varepsilon_w$, then this contribution is $\approx (1 + \varepsilon_f/\varepsilon_w) \cdot \varepsilon_w$. If $\varepsilon_f \approx \sqrt{\varepsilon_w}$ for a low-precision factorization as in Langou et al. [69], this forward error bound still will be limited to around $\sqrt{\varepsilon_w}$. For that case, condition estimation combined with the backward error will provide better evidence of success.

The contribution of δr_1 is similar with an additional scaling by $\|D_2^{-1} D_1\|_\infty$,

$$\|D_2^{-1}(A + \delta A_2)^{-1}\delta r_1\|_\infty \leq \|D_2^{-1} D_1\|_\infty (1 + \varepsilon_f p(n_d) g \operatorname{cond}(A, y_2)) \left((n_d + 1)\varepsilon_r \operatorname{ccond}(A, y_1) + \varepsilon_w \cdot \operatorname{cond}_1 \cdot \operatorname{berr}_1\right). \tag{3.6.8}$$

The interpretation here is similar.

Altogether, the t_i term for dy_i is the expression

$$\begin{aligned} t_i = [1 + \varepsilon_f p(n_d) g \operatorname{cond}(A, y_2)] \; [\varepsilon_x + \varepsilon_w (\operatorname{cond}_2 \operatorname{berr}_2 \\ + \|D_2^{-1} D_1\|_\infty \operatorname{cond}_1 \operatorname{berr}_1) + (n_d + 1)\varepsilon_r (\operatorname{ccond}(A, y_2) + \\ \|D_2^{-1} D_1\|_\infty \operatorname{ccond}(A, y_1))\big] . \end{aligned} \tag{3.6.9}$$

Assume that $\operatorname{cond}_i \operatorname{berr}_i \leq 1$ by the backward error's convergence, and that $\|D_2^{-1} D_1\|_\infty \approx 1$ by stability near convergence of the solution[2] which can be monitored. Then terminal error depends on the largest of ε_x, ε_w, and $\varepsilon_r \cdot \operatorname{ccond}(A, y_i)$, multiplied by an accuracy measure of the factorization. To understand the terminal forward error, consider a few possible precision settings.

3.6.3 Fixed-precision refinement in double precision

For typical fixed-precision refinement in double precision, $\varepsilon_f = \varepsilon_r = \varepsilon_x = \varepsilon_w = \varepsilon_d$. Section 3.5 established that we expect the backward error to converge to a multiple of ε_d. So we do not expect to achieve results better than $\varepsilon_d \operatorname{cond}(A, x)$. Optimistically dropping all second-order effects, assuming stability of $\|D_2^{-1} D_1\|_\infty$, and keeping enough terms to interpret, we find the expected expression

$$(\underline{c} - \bar{\rho})\|D_i^{-1} dy_i\|_\infty \lessapprox 2(3\varepsilon_d + 2(n_d + 1)\varepsilon_d \operatorname{ccond}(A, y_i)), \tag{3.6.10}$$

where $\underline{c}$ and $\bar{\rho}$ are as defined in Section 3.4. By Equation (3.6.4),

$$(\underline{c} - \bar{\rho})\|D_i^{-1} e_i\|_\infty \lessapprox 2(5 + 2(n_d + 1) \operatorname{ccond}(A, y_i))\varepsilon_d.$$

[2] In this case, we let $0/0 = 1$ as opposed to the zero used elsewhere.

The step size bound and forward error bound increase linearly with the conditioning of the system. This bound is not an improvement on the standard bounds in Demmel [34] or Higham [59], but the similar form helps validate our analysis.

3.6.4 Extended and slightly extended precision refinement

Using double-extended precision of the Intel style[3], consider slightly extended-precision refinement with $\varepsilon_f = \varepsilon_w = \varepsilon_d$ and $\varepsilon_r = \varepsilon_x = \varepsilon_{de} = 2^{-10}\varepsilon_d$. Then

$$(\underline{c} - \bar{\rho})\|D_i^{-1} dy_i\|_\infty \lessapprox 2((2 + 2^{-10})\varepsilon_d + 2^{-9}(n_d + 1)\varepsilon_d \operatorname{ccond}(A, y_i)), \tag{3.6.11}$$

and

$$(\underline{c} - \bar{\rho})\|D_i^{-1} e_i\|_\infty \lessapprox 2((4 + 2^{-10}) + 2^{-9}(n_d + 1) \operatorname{ccond}(A, y_i))\varepsilon_d.$$

This is not appreciably different than the fixed-precision result beyond the factor of 2^{-9} before the condition number. That factor shifts the error growth slightly with respect to conditioning. For some applications, this may have been useful when double-extended precision ran at approximately the same speed (or faster!) than double precision. On current high-performance Intel and AMD architectures, however, double precision vector operations run 2×-4× faster than the scalar floating-point operations and require less memory traffic.

The shift in the condition number is far more dramatic when $\varepsilon_x \leq \varepsilon_d^2$, as in Demmel et al. [37, 35]. Then

$$(\underline{c} - \bar{\rho})\|D_i^{-1} dy_i\|_\infty \lessapprox 2((2 + \varepsilon_d)\varepsilon_d + (n_d + 1)\varepsilon_d^2 \operatorname{ccond}(A, y_i)),$$

and

$$(\underline{c} - \bar{\rho})\|D_i^{-1} e_i\|_\infty \lessapprox 2((4 + \varepsilon_d) + (n_d + 1)\varepsilon_d \operatorname{ccond}(A, y_i))\varepsilon_d.$$

If $\operatorname{ccond}(A, y_i) \leq 1/(\bar{\rho}\varepsilon_d)$,

$$\|D_i^{-1} e_i\|_\infty \lessapprox 2(4 + \bar{\rho}(n_d + 1))\varepsilon_d \cdot (\underline{c} - \bar{\rho})^{-1}, \tag{3.6.12}$$

matching well the experimental results achieved in Demmel et al. [37, 35] with $n_d = \sqrt{n}$, $\underline{c} = 0.5$, and a safety factor or $\bar{\rho} = 0.1/2 = 0.05$ (explained below).

The primary benefit to extending the precision lies in driving the backward error below ε_w^2 when the system is not too ill-conditioned ($O(1/\varepsilon_w)$). Then the forward error is at most $O(\varepsilon_w)$ regardless of other considerations. Telescoping the precision as in Kiełbasiński [66] requires also extending the working precision along with the other precisions. Nguyen and Revol [78] examine refinement carrying both to double precision in the context of verifiable computation. Their verification encounters similar issues with extremely ill-conditioned systems and fails to validate the solution in that regime.

[3]Deprecated in future architectures, alas.

3.6.5 Lower-precision and perturbed factorizations

The rise of fast, single-precision computation in GPUs returned interest in iterative refinement for providing double-precision backward error from a single-precision factorization [69]. Similar techniques have been applied to out-of-core sparse matrix factorizations [63]. Perturbing a factorization to avoid tiny pivots [82, 60, 71, 48, 95, 53] similarly reduces the precision, and threshold pivot choices to maintain sparsity increase element growth [72, 61].

For a single-precision factorization as in Langou et al. [69], $\varepsilon_f = \varepsilon_s$ and $\varepsilon_w = \varepsilon_x = \varepsilon_r = \varepsilon_d$. Assuming that $\text{cond}_i \cdot \text{berr}_i \leq 1$ and $\varepsilon_d \, \text{ccond}(A, y_2) \leq 1$,

$$\|D_i^{-1} dy_i\|_\infty \lessapprox (1 + \varepsilon_s p(n_d) g \, \text{cond}(A, y_i)) \, (2 + 2n_d)\varepsilon_d.$$

So the bound on dy, and hence e_i, is a small multiple of ε_d until $p(n_d) g \, \text{cond}(A, y_i)$ grows beyond $1/\varepsilon_s$. Requiring convergence similar to Equations (3.6.10) and (3.6.11) likely will succeed only when $p(n_d) g \, \text{cond}(A, y_i) < 1/\varepsilon_s$. Beyond that point, we should not expect a double-precision *forward error*. Extending the other precisions does not affect the $\varepsilon_s p(n_d) g \, \text{cond}(A, y_i)$ term but does lower the $2 + 2n_d$ term. This analysis does not support double-precision forward error with extended precision when the factorization is of lower precision and the system is even moderately well-conditioned.

Threshold pivoting for sparse matrix factorization [49] may cause a similar effect even with $\varepsilon_f = \varepsilon_d$ by inflating the element growth term g. The growth shifts the effective conditioning of the system, $p(n_d) g \, \text{cond}(A, y_2)$, making double-precision forward error less likely. Perturbing a factorization to preserve sparsity effectively reduces the precision and increases ε_f by an amount proportional to the perturbation and again may affect the forward error adversely. Chapter 5 discusses these mechanisms in more detail and provides numerical results.

3.6.6 Ratio of decrease in dy

As with the backward error, we rely on the decrease in dy to accept or reject the results. Taking the first term of Equation (3.6.5)'s ratio,

$$\begin{aligned}
\|D_2^{-1}(A + \delta A_2)^{-1} \delta A_1 D_2\|_\infty &= \|D_2^{-1}(I + A^{-1}\delta A_2)^{-1} A^{-1} \delta A_1 D_2\|_\infty \\
&\leq \|D_2^{-1}(I + A^{-1}\delta A_2)^{-1} D_2\|_\infty \|D_2^{-1} A^{-1} \delta A_1 D_2\|_\infty \\
&\lessapprox (1 + p(n_d) g \varepsilon_f \, \text{ccond}(A, y_2)) \cdot p(n_d) g \varepsilon_f \, \text{ccond}(A, y_2)
\end{aligned}$$

using the same techniques as with the backward error decrease. Then

$$\rho_2 \lessapprox (1 + p(n_d) g \varepsilon_f \, \text{ccond}(A, y_2)) \cdot p(n_d) g \varepsilon_f \, \text{ccond}(A, y_2) \|D_2^{-1} D_1\|_\infty$$

in the general recurrence (Equation (3.4.3)). Assuming $p(n_d) g \, \text{ccond}(A, y_i) \leq 1/\varepsilon_f$ in the first term, and assuming $\|D_2^{-1} D_1\|_\infty \approx 1$,

$$\rho_i \lessapprox 2 p(n_d) g \varepsilon_f \, \text{ccond}(A, y_i).$$

Targeting $\bar{\rho} = 0.05$ should accept systems within a factor of 10 of the "ill-conditioned" boundary $p(n_d)g\,\mathrm{ccond}(A, y_i) \leq 1/\varepsilon_f$ as in Demmel et al. [37, 35]. With $\underline{c} = 0.5$, the factor of $(\underline{c} - \bar{\rho})^{-1}$ in bounds above is less than 2.25. With the aggressive $\underline{c} = 0.9$, the factor is less than 1.25.

3.7 Defining the termination and success criteria

Given the above analysis, we define check_berr_criteria and check_ferr_criteria routines in Listings 3.4 and 3.5, respectively. These routines assume extended precision for *both* the internal solution ($\varepsilon_x \leq \varepsilon_w^2$) and residual computation ($\varepsilon_r \leq \varepsilon_w^2$). In each, we assume only a single definition of forward or backward error from Table 2.2 is targeted. Supporting multiple errors is straight-forward within each routine. We also must choose which backward error to use for stability in Listing 3.5 For all our later experiments, we base stability off the columnwise backward error in Equation (2.2.6).

The backward error is available by direct computation. For estimating the forward error on success, we over-estimate the relationship between the error and step size implied by Equation (3.6.12). The error estimate is the bound from Equation (3.6.9) doubled to conservatively over-estimate Equation (3.6.4)'s relationship. On success, this is always a small multiple of the working precision, so an error estimate may be uninteresting. [37, 35] track the ratios c_i and use the geometric assumption in Equation (3.6.1). However, the error estimate is only considered trust-worthy if the system is not too ill-conditioned, and the error in that case always is small. We avoid the issues related to starting and stopping tracking the ratio but give up error estimates that worked surprisingly well even for ill-conditioned systems.

As a substitute for monitoring $\|D_{i-1}^{-1} D_i\|_\infty$ during the iteration for the forward errors to catch wide swings in magnitude, we check that ${}^{\min_j |D_i(j,j)|}/_{\max_j |D_i(j,j)|} \geq \varepsilon_f c_i$ afterward. Checking once afterward avoids $O(n)$ expense per iteration and, more importantly, $O(n)$ extra storage. This check is trivial for the normwise forward error but provides a rough-enough estimate of the componentwise forward error's difficulty from $\mathrm{ccond}(A, x) \leq \mathrm{cond}(A) \cdot {}^{\max|x|}/_{\min|x|}$. This will forgo some successes but suffices to prevent missing errors in small components. Solutions with zero components always report componentwise failure, which may be inappropriate for block-structured matrices. Future work may investigate limiting the check to non-zero solution components, but our conservative approach is safe.

3.8 Incorporating numerical scaling

Systems $Ax = b$ passed to solving software often are constructed such that different components have different measurement scales. For example, a large chemical vat may be connected to a small valve. The vastly different scales confuse many error measures and condition

```
function state = check_berr_criteria (state, i, r, A, y, b)
### Check the backward error criteria for the error computed by
### compute_berr. Assumes the measure does not change drastically,
### ‖D_{i-1}^{-1} D_i‖_∞ ≈ 1, and that refinement uses ε_x ≤ ε_w^2 and ε_r ≤ ε_w^2.
  c_lower = 0.9; # Aggressive progress setting.
  rho_upper = 0.1; # Factor of 10 safety bound.
  n_d = sqrt (size (A, 1));

  r = r{i};
  S = state.berr;

  if isfield (S, "success"),
    return; # Already finished.
  endif

  if isdouble (A),
    rprec = 2**-106;
    xprec = 2**-106;
  else
    rprec = 2**-53;
    xprec = 2**-53;
  endif

  berr_bnd = (c_lower - rho_upper)**-1 * (3 * (n_d + 1) * rprec + xprec);

  berr = compute_berr (r, A, y, b);
  ## Computes:
  ## - norm (abs(r) ./ (abs(A) * abs(y) + abs(b)), inf) for componentwise,
  ## - norm(r, inf) / (norm(A,inf) * norm(y,inf) + norm(b,inf)) for normwise, or
  ## - norm (abs(r) ./ (max(A) * abs(y) + abs(b))) for columnwise.
  if berr <= berr_bnd,
    S.success = 1;
  else
    prev_berr = state.berr;
    c = berr / prev_berr;
    if c >= c_lower,
      S.success = 0;
    endif
  endif
  S.berr = berr;
endfunction
```

Listing 3.4: Backward error termination criteria.

```
function state = check_ferr_criteria (state, i, y, dy)
### Check the forward error criteria using the step size from
### compute_step_size (y, dy). Assumes the measure does not change
### drastically, ||D_{i-1}^{-1} D_i||_inf ~= 1, and that refinement uses eps_x <= eps_w^2 and
### eps_r <= eps_w^2. Also assumes the system is not too ill-conditioned by
### requiring that the backward error converge.
  c_lower = 0.9; # Aggressive progress setting.
  rho_upper = 0.1; # Factor of 10 safety bound.
  n_d = sqrt (size (A, 1));

  if !isfield (state.berr, "success"),
    return; # Backward error has not converged.
  endif
  dy = dy{i};
  S = state.step;
  if isfield (S, "success"),
    return; # Already finished.
  endif
  if isdouble (A),
    wprec = 2**-53;
  else
    wprec = 2**-24;
  endif
  step_bnd = (c_lower - rho_upper)**-1 * 2 * (2 + rho_upper * (n_d + 1)) * wprec;
  step = compute_step_size (y, dy);
  ## For normwise error, step = norm(dy,inf)/norm(y,inf).
  ## For componentwise error, step = norm(dy./y, inf), with some
  ## care for 0/0 = 0.
  prev_step = state.step;
  c = step / prev_step;
  if step <= step_bnd,
    ## As a substitute for monitoring ||D_{i-1}^{-1} D_i||_inf ~= 1,
    ## check min diag D_i / max diag D_i >= eps_f * c.
    ## Note: The check is trivial for the normwise forward error.
    S.success = check_relative_size (y);
    S.ferr_est = 2 * step_bnd;
  elseif c >= c_lower,
    S.success = 0;
  endif
  S.step = step;
endfunction
```

Listing 3.5: Forward error termination criteria. The forward error on success is estimated as twice the larger of the step and the working precision, similar to Equation (3.6.12).

numbers. A single millimeter error in the vat may go without notice, while a millimeter error in the valve may be critical. A normwise error would treat all millimeter errors as the same. Measuring the valve in millimeters and the vat in meters would ameliorate the issue, but that might cause problems for the user-interface up in the software stack.

Software can cope with some of these problems through numerical scaling. The system $Ax = b$ is scaled to $A_s x_s = b_s$ using diagonal scaling factors R and C by $A_s = RAC$, $b_s = Rb$, and $x_s = C^{-1}x$. This effectively scales the range and domain norms.

Conditioning in the unscaled system $Ax = b$ is never changed by such numerical scaling, so the limiting precisions for refinement do not change significantly. The quality of the factorization, and hence rate of decrease and success rate, can be improved greatly when working with A_s. The linear solves inside factorization may work with a more well-conditioned system, and scaling often reduces element growth. Reducing element growth will prove pivotal[4] with static pivoting.

Our different error and step measurements have different levels of invariance to these scaling factors. Componentwise measures are scale-invariant and need not change at all. Normwise measures must be measured in the user's original norms and require slight changes to the refinement algorithm. Listings 3.6, 3.7, and 3.8 update Listings 3.1, 3.4, and 3.5, respectively, to incorporate scaling. The values used within the iteration are kept in the scaled space.

We assume the scaling does not cause new, harmful underflow or overflow. Such cases can be detected and handled with care if necessary, but doing so significantly complicates the presentation. Computations of the backward errors compensating for underflow while recognizing the exact zeros that occur in block matrices become quite baroque.

The simple numerical strategy implemented in LAPACK's *xyy*`EQUB` works well for little cost. First, the column scaling C is computed such that the largest entry in magnitude of each column in AC is one. Then the rows of AC are scaled such that the largest magnitude of each is one. That suffices to avoid overflow in many situations and often reduces the entry growth. This equilibration does not fix all cases of ill-scaling. Consider the ill-scaled matrix

$$\begin{bmatrix} 0 & G & G \\ G & g & 0 \\ G & 0 & g \end{bmatrix},$$

where G is extremely large and g extremely small. Our equilibration retains this matrix's ill-scaling and ill-conditioning, while scaling

$$\begin{bmatrix} 1/G & & \\ & 1/g & \\ & & 1/g \end{bmatrix} \begin{bmatrix} 0 & G & G \\ G & g & 0 \\ G & 0 & g \end{bmatrix} \begin{bmatrix} 1/G & & \\ & 1 & \\ & & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

would remove all numerical issues.

[4] Pun intended.

```
function [yout, success] = itref_scaled (R, As, C, x1, bs, MAXITER = 100)
### Function File [xout, success] = itref (A, x1, b[, MAXITER = 100])
### Refine an initial solution x1 to A*x=b. The result xout contains
### the refined solutions, and success is an opaque structure to be
### queried for the status of xout. The SCALED system As xs = bs is
### used internally, but all termination decisions are made in the user's
### original norm.
  nrhs = dim (x1,2);
  xout = zeros (dim (x1, 1), nrhs);
  for j = 1:nrhs,
    state = initial_refinement_state ();
    y{1} = C \ widen(x1); # Store solutions to extra precision ε_x.
    i = 1;
    while i < MAXITER,
      ## Compute this step's residual: r_i = b − A y_i + δr_i.
      ## Intermediates are computed in extra precision ε_r.
      r{i} = widen_intermediates b − A*y{i};
      state = check_berr_criteria_sc (state, i, rs, R, As, C, y, bs);
      if all_criteria_met (state), break; endif
      ## Compute the increment: (A + δA_i)δy_i = r_i.
      dy{i} = A \ r{i};
      state = check_ferr_criteria_sc (state, i, y, dy, C);
      if all_criteria_met (state), break; endif
      ## Update the solution: y_{i+1} = y_i + dy_i + δy_i.
      y{i+1} = y{i} + dy{i};
      i += 1;
    endwhile
    yout(:,j) = C * y{i}; # Round to working precision.
    success{j} = state_success (state);
  endfor
endfunction
```

Listing 3.6: Iterative refinement of $Ax = b$ in Octave incorporating numerical scaling $RAC = A_s$, $Rb = b_s$. The scaled system $A_s x_s = b_s$ is used internally, but the termination criteria and error estimates are computed in the user's original norm.

```
function state = check_berr_criteria_sc (state, i, rs, R, As, C, y, bs)
### Check the SCALED backward error criteria for the error computed by
### compute_berr_sc. Assumes the measure does not change drastically,
### ‖D_{i-1}^{-1} D_i‖_∞ ≈ 1, and that refinement uses ε_x ≤ ε_w^2 and ε_r ≤ ε_w^2.
  c_lower = 0.9; # Aggressive progress setting.
  rho_upper = 0.1; # Factor of 10 safety bound.
  n_d = sqrt (size (A, 1));

  rs = rs{i};
  S = state.berr;

  if isfield (S, "success"),
    return; # Already finished.
  endif

  if isdouble (A),
    rprec = 2**-106;
    xprec = 2**-106;
  else
    rprec = 2**-53;
    xprec = 2**-53;
  endif

  berr_bnd = (c_lower - rho_upper)**-1 * (3 * (n_d + 1) * rprec + xprec);

  berr = compute_berr_sc (rs, R, As, C, y, bs);
  ## Computes:
  ## - norm (abs(rs) ./ (abs(As) * abs(ys) + abs(bs)), inf) for componentwise,
  ## - norm(R\rs, inf) / (norm(R\As/C,inf) * norm(C*y,inf) + norm(R\bs,inf)) for normwise, or
  ## - norm (abs(R\rs) ./ (max(R\A) * abs(y) + abs(R\bs))) for columnwise.
  if berr <= berr_bnd,
    S.success = 1;
  else
    prev_berr = state.berr;
    c = berr / prev_berr;
    if c >= c_lower,
      S.success = 0;
    endif
  endif
  S.berr = berr;
endfunction
```

Listing 3.7: *Scaled* backward error termination criteria.

```
function state = check_ferr_criteria_sc (state, i, y, dy, C)
### Check the SCALED forward error criteria using the step size from
### compute_step_size_sc (y, dy). Assumes the measure does not change
### drastically, ||D_{i-1}^{-1} D_i||_inf ~= 1, and that refinement uses eps_x <= eps_w^2 and
### eps_r <= eps_w^2. Also assumes the system is not too ill-conditioned by
### requiring that the backward error converge.
  c_lower = 0.9; # Aggressive progress setting.
  rho_upper = 0.1; # Factor of 10 safety bound.
  n_d = sqrt (size (A, 1));

  if !isfield (state.berr, "success"),
    return; # Backward error has not converged.
  endif
  dy = dy{i};
  S = state.step;
  if isfield (S, "success"),
    return; # Already finished.
  endif
  if isdouble (A),
    wprec = 2**-53;
  else
    wprec = 2**-24;
  endif
  step_bnd = (c_lower - rho_upper)**-1 * 2 * (2 + rho_upper * (n_d + 1)) * wprec;
  step = compute_step_size_sc (y, dy, C);
  ## For normwise error, step = norm(C*dy,inf)/norm(C*y,inf).
  ## For componentwise error, step = norm(dy./y, inf), with some
  ## care for 0/0 = 0.
  if step <= step_bnd,
    S.success = check_relative_size (C*y);
    S.ferr_est = 2 * max (step, wprec);
  else
    prev_step = state.step;
    c = step / prev_step;
    if c >= c_lower,
      S.success = 0;
    endif
  endif
  S.step = step;
endfunction
```

Listing 3.8: *Scaled* forward error termination criteria.

3.9 Potential for failure and diagnostics

In this section, we collect all the assumptions made in our refinement routine. Each assumption brings with it the potential for failure. Some failure modes can be monitored and reported. Others, like those related to conditioning, can be at best estimated within reasonable computational limits.

We make the following over-all assumptions about refinement:

- Underflow is ignorable.
- A related assumption is that the residual is computed accurately enough that accidental cancellation can be ignored.
- The recurrence terms ρ_i and t_i change little between each iteration.
- The scaling factors defining error measure change little between each iteration,

$$\|D_{i-1}^{-1} D_i\|_\infty \approx 1.$$

- True zero components converge to zero often enough that we can define $0/0 = 0$ where necessary and suffer infinities from rare failures.

These assumptions affect both the backward and forward error.

If the input is within the range of normalized floating-point numbers, Demmel [32] shows that underflow does not contribute substantially to the typical error bounds for solving $Ax = b$. Underflow could affect refinement by causing a too-small residual scaled back up by a tiny denominator. With extra precision, the former is unlikely. With numerical scaling, the latter is extremely unlikely. By not considering underflow in the residual, we avoid issues with block matrices and zero solution components that caused arbitrarily large backward errors in earlier versions of LAPACK. The extra precision also renders accidental cancellation unlikely. Using an accurate dot product like the one in Ogita et al. [81] eliminates the problem of cancellation altogether.

The assumptions about slow changes in the various terms are perhaps the most dangerous. The changes in $\|D_{i-1}^{-1} D_i^{-1}\|_\infty$ can be monitored, although at $O(N)$ storage cost. Monitoring that change would also handle changes in the t_i terms. Enforcing an upper bound on the permitted ratio c_i should keep fluctuations in ρ_i under control. Our experiments do not monitor these changes and do not seem worse for that lack.

True solutions to linear systems may have exact zero components. These appear in optimization applications when solving for directional derivatives at optimal or saddle points, in physical models where forces or currents are balanced, *etc.* Consider the exact zero solution to $Ax = 0$. If factorization of A_s succeeds, the initial solve calculates $y = 0$ exactly. The first residual $r_1 = 0$, so the step $dy_1 = 0$. When calculating divisions like $|dy_i|/|y_i|$, our implementation tests for zeros and substitutes the result $0/0 = 0$. $Ax = b$ where A is the

identity matrix and $b = [100]^T$ is a sparse right-hand side. The initial solution $y_1 = b$. Again, $r_1 = 0$ and $dy_1 = 0$. The divisions ${}^{|dy_1(1)|}/_{|y_1(1)|} = 0/1 = 0$ and ${}^{|dy_1(2)|}/_{|y_1(3)|} = 0/0 \equiv 0$ lead to immediate convergence.

Beyond these, we also make assumptions about the forward error.

- There is a single solution to $Ax = b$, or equivalently that A is invertible.
- If the backward error converges, the system is stable enough for forward error to make progress or fail when appropriate.
- The system solved at each iteration is never so ill-conditioned that the step size is too large compared to the forward error.
- The errors in factorization are relatively small, so $(I - A\delta A)^{-1} = I + F$ results in a relatively small F.

A need not be invertible for the backward error to fall to zero. In that case, we have some vector in the null space. A must be invertible to find a solution with known forward error. LAPACK's *xyy*`SVX` drivers compute $\kappa(A_s)$, the normwise condition number of the numerically scaled matrix, to check that A is invertible. Our non-perturbed test cases are invertible by construction. Our singular but perturbed sparse systems do not converge to a successful forward error, but we recommend testing at least $\kappa(A_s)$ when using intentionally perturbed factorizations. That adds five to ten matrix-vector products and system solutions to the total solution time, however. We find this extra check unnecessary in our partial pivoting test cases.

When faced with Rump's outrageously ill-conditioned matrices [91] and random x, our algorithm either successfully solved the systems ($O(\varepsilon_w)$ errors and bounds) or correctly reported failure. Consider also Example 2.6 from Demmel [34], modified for single-precision IEEE754 arithmetic. The example involves the exactly singular matrix

$$A = \begin{bmatrix} 3 \cdot 2^7 & -2^7 & 2^7 \\ 2^{-7} & 2^{-7} & 0 \\ 2^{-7} & -3 \cdot 2^{-7} & 2^{-7} \end{bmatrix}.$$

We compute and store $b = A \cdot [1, 1 + \varepsilon_w, 1]^T$ as single-precision data. Factorization succeeds in single precision without equilibration, and refinement produces a result with small backward error. However, the step size never becomes small enough for the forward error to be accepted.

If the step size is large even when the error is small, we may fail to recognize a successful solution and orbit around that solution. This is a failure of opportunity, but not one of accepting an incorrect result. Large factorization errors can cause this phenomenon as well. Our later sparse systems simply fail outright when the perturbations or element growth are too large.

3.9.1 Diagnostics

What occurs when the algorithm fails to produce successful errors? The possible failure modes in this section combined with the assumptions in Sections 3.5 and 3.6 provide a short list of diagnostics about *why* refinement failed. Here we briefly discuss the diagnostics. A simple and useful method for providing these diagnostics to routines and users is outside our scope. Developing such methods for different users is a problem worthy of future investigation.

The following are non-exclusive causes of failure to converge:

- the system is ill-conditioned,
- the factorization is poor,
- the solution has relatively tiny components (for componentwise forward error), or
- the maximum number of iterations is exhausted.

If refinement terminates because the per-step ratio $c > \underline{c}$, we assume one of the first two reasons applies. If the element growth is substantially large, say $1/\sqrt{\varepsilon_f}$, some diagnostic about the factorization quality should be triggered. Otherwise some diagnostic tool should assume ill-conditioning but possibly a good factorization.

If iteration terminates without converging or declaring a lack of progress, we also can assume some moderate ill-conditioning. The element growth again helps identify likely poor factorizations.

Checking if the solution has relatively tiny components is straight-forward and indicates possible poor column scaling.

Chapter 4

Iterative refinement for dense, square linear systems

4.1 Generating dense test systems

We validate our iterative refinement algorithm from Chapter 3 against actual (although artificial) test systems $Ax = b$.[1] This section expands the test generator in Demmel et al. [35, 37] for complex systems and double precision. The generator was implemented with the other authors; this section serves as reference and does not discuss new material. This section can be skipped if you trust us to test a reasonable range of problems from "easy" or well-conditioned to "difficult" or ill-conditioned. Section 4.2 presents the results on 30×30 systems.

4.1.1 Generating dense test systems

The generated dense test systems are spread across a range from very "easy" systems to very "difficult" ones. The easy systems have reliably accurate factorizations of A and evenly scaled solutions x. The difficult systems combine a matrix A that is nearly singular for the working precision with a solution x containing entries of vastly different magnitude. Testing with nearly singular matrices demonstrates that our refinement algorithm does not flag incorrect solutions as correct. And testing solutions with relatively tiny entries demonstrates that refinement achieves componentwise accuracy.

The method for generating these test systems is somewhat ornate. Listing 4.1 provides an Octave version of the top-level routine. The gen_dense_test_sys routine generates the matrix A with Listing 4.3's gen_dense_test_mat. The resulting matrix may be horribly ill-conditioned and difficult to solve accurately for a true solution. With a possibly ill-conditioned matrix A

[1] The driver codes and some analysis codes are available through the git meta-project currently at `http://lovesgoodfood.com/jason/cgit/index.cgi/thesis-dist/` .

```
function [A,x,b,xt] = gen_dense_test_sys (assingle=true, asreal=true, \
                                          N=30, nrhs = 4, \
                                          nrandx = 2)
### Function File [A,x,b] = gen_dense_test_sys (assingle, asreal, N, nrhs, nrandx)
### Generate a dense test system A*x = b. Default values: N = 30,
### nrhs = 4, nrandx = 2, assingle = true, and asreal = true.
### See also: gen_dense_mat
  if (N < 3), error ("Only N>=3 supported."); endif
  A = gen_dense_test_mat (N, assingle, asreal);
  Amax = max(max(abs(A)));
  if isempty (nrandx), nrandx = nrhs; endif
  b = gen_dense_test_rhs (A, assingle, asreal, nrhs, nrandx);
  ## We assume the only error in accurate_solve() is the
  ## final rounding to the output precision.
  try
    [x, xt] = accurate_solve (A, b);
  catch
    x = []; xt = [];
    warning (lasterr);
  end_try_catch
endfunction
```

Listing 4.1: Generating dense test systems $Ax = b$. The real accurate_solve factors A with at least quadruple precision and applies refinement with at least octuple-precision residuals and intermediate solutions.

```
function b = gen_dense_test_rhs (A, assingle=true, asreal=true, \
                                  nrhs = 4, nrandx = 2)
  N = size (A, 1);
  if assingle, logcond_max = 24; else logcond_max = 53; endif
  b = zeros (N, nrhs);
  if assingle, b = single (b); endif
  for k = 1:nrhs,
    if k <= nrandx,
      ## Pick a random target solution "condition" number.
      target_xcond = 2^(logcond_max * rand ()^2);
      ## Generate an initial random x
      xmode = 1 + floor (5 * rand ());
      x = xLATM1 (xmode, target_xcond, rand () < .5, N,
                  assingle, asreal);
      if xmode <= 4,
        ## The largest entry is 1 in these modes. That is a
        ## rather artificial value and may be "too easy" for
        ## normwise convergence. Multiply x by a random
        ## number in (1/2, 3/2) to fill out the number.
        x *= .5 + rand ();
      endif
      ## Generate b with the temporary x.
      b(:,k) = A * x;
    else
      ## Generate a random b, possibly with small entries.
      target_bcond = 2^(logcond_max * rand ()^2);
      ## The mode is fixed; entries are in (1/target_bcond,
      ## 1) with uniformly distributed logarithms.
      b(:, k) = xLATM1 (5, target_bcond, rand () < .5, N, \
                        assingle, asreal);
    endif
  endfor
endfunction
```

Listing 4.2: Generating dense test right-hand sides b for $Ax = b$.

in hand, gen_dense_test_sys calls gen_dense_test_rhs to generate right-hand sides b. Two different styles are included. One generates b randomly, and the other generates a random x and computes $b = Ax$. The former may produce larger or more ill-scaled solutions x than the latter. The latter, generating a temporary x, will be projected against any near-null space of A and test smaller solutions.

We then rely on accurate_solve to produce x from A and b. The x, stored in x and xt, is provided doubled-double. The accurate_solve routine signals an error if A is numerically singular to at least twice the working precision. The low-level test generator solves $Ax = b$ with quad-double [55] precision.

The matrix-generating routine gen_dense_test_mat in Listing 4.3 begins by chosing a random target difficulty (2-norm condition number) with a uniformly distributed base-2 logarithm in

```
function A = gen_dense_test_mat (N=100, assingle=true, asreal=true)
### Function File A = gen_dense_test_mat (N, assingle, asreal)
### Generate a dense test matrix for testing A*x = b solvers.
### Default values: N = 100, assingle = true, asreal = true
### See also: gen_dense_sys
  if (N < 3), error ("Only N>=3 supported."); endif

  ## Pick a random target condition number.
  if assingle, logcond2_max = 26; else logcond2_max = 56; endif
  target_cond2 = 2^(logcond2_max * rand ());

  ## Generate a diagonal from one of four modes.
  mode = 1 + floor (4 * rand ());
  D = xLATM1 (mode, target_cond2, rand () < .5, N,
              assingle, asreal);

  ## Force a leading section of the matrix to be ill-conditioned.
  piv = 3*rand ();
  if piv < 1,
    piv = 3;
  elseif piv < 2,
    piv = floor (N/2);
  else
    piv = N;
  endif
  ## There is only one large singular value in mode 1
  if mode != 1 && piv > 1 && piv < N,
    [ignore, idx] = sort (abs (D), "descend");
    k = idx(1);
    if k > 1,
      tmp = D(k); D(k) = D(1); D(1) = tmp;
      ## In case the switch impacts the later switches...
      idx(idx == 1) = k;
    endif
    k = idx(N);
    if k > 2,
      tmp = D(k); D(k) = D(2); D(2) = tmp;
      idx(idx == 2) = k;
    endif
    k = idx(2);
    if k > 2,
      tmp = D(k); D(k) = D(piv); D(piv) = tmp;
    endif
  endif

  A = full (diag (D));
  if assingle, A = single (A); endif
  ## Multiply the leading block by a random unitary mtx. on the right
  A(1:piv,1:piv) = A(1:piv,1:piv) * gen_rand_Q (piv, assingle, asreal);
  ## Multiply the trailing block by a random unitary mtx. on the right
  A(piv+1:N,piv+1:N) = A(piv+1:N,piv+1:N) * gen_rand_Q (N-piv, assingle, asreal);
  ## Multiply all of A a random unitary mtx. on the left
  A = gen_rand_Q (N, assingle, asreal)' * A;
endfunction
```

Listing 4.3: Generating dense test matrices for $Ax = b$.

$(1, 2^{26})$ for single precision and $(1, 2^{56})$ for double precision. The testing code then generates a vector using LAPACK's x`LATM1`, translated to Octave in Listing 4.4. x`LATM1` produces a vector with different patterns of large and small entries depending on the input mode, which gen_dense_test_mat randomizes. x`LATM1` also can randomly switch real entries' signs or rotate complex entries around a random angle according to targetsign.

After generating the random vector, gen_dense_test_mat swaps the largest and smallest magnitude entries into the first two positions and swap the second-largest magnitude entry into the piv entry where piv is randomly chosen from 3, $\lfloor n/2 \rfloor$, and n. This will produce an A such that the leading 1:piv columns are ill-conditioned and the trailing 1+piv:N columns are not entirely negligible.

To generate a full matrix while preserving both the difficulty (2-norm condition number) as well as the column split at piv, gen_dense_test_mat finishes by converting the vector to a diagonal matrix and then applying unitary transforms on the left and right. The transforms take the form

$$A = {Q_L}^H D \begin{bmatrix} Q_{R1} & 0 \\ 0 & Q_{R2} \end{bmatrix}.$$

The right unitary transform is split into two diagonal blocks, Q_{R1} for the leading piv columns and Q_{R2} for the trailing N−piv columns.

4.1.2 The generated dense test systems

Figures 4.1, 4.2, and 4.3 show histograms of our test systems' difficulties by different measures. There are one million systems generated for each working precision, single and double, and each entry type, real and complex.

The value of $\kappa_\infty(A)$ plotted in Figure 4.1 is roughly the inverse of the normwise distance from A to the nearest singular matrix [33]. Figure 4.1 shows that the tests include from 7% to 15% of matrices that are "numerically singular" to working precision. Using extra precision, the test generator's accurate solver indicates these matrices actually are not singular. They do, however, produce a few factorization failures in working precision, and those failures are useful for testing.

Figure 4.2 reflects the expected difficulty of solving $Ax = b$ for a normwise accurate x when provided a backward-stable method. Similarly, Figure 4.3 reflects the expected difficulty of solving $Ax = b$ for an x accurate in *every component* in the same situation.

The skew towards difficult componentwise problems in Figure 4.3 is expected. Our tests' x_0-generation is designed to make obtaining componentwise accuracy more difficult. Generating x_0 from x`LATM1` produces solutions with varying magnitudes.

Table 4.1 provides the percentage of "difficult" cases in each precision.

```
function D = xLATM1 (mode, targetcond, togglesign, N, assingle, asreal)
### Function File D = xLATM1 (mode, targetcond, togglesign, N, assingle, asreal)
### Emulate LAPACK's xLATM1 vector generators.
  D = zeros(N,1);
  ## Generate D according to the mode.
  switch abs(mode)
    case {1}
      D(1) = 1; D(2:N) = 1/targetcond;
    case {2}
      D(1:N-1) = 1; D(N) = 1/targetcond;
    case {3}
      D = targetcond.^(-(0:N-1)/(N-1));
    case {4}
      D = 1 - (0:N-1)/(N-1) * (1 - 1/targetcond);
    case {5}
      ## Randomly in (1/targetcond, 1) with uniformly
      ## distributed logarithms.
      D = targetcond.^-rand (N,1);
    case {6}
      error ("Unused in our tester.");
    otherwise
      error ("Unrecognized mode value.");
  endswitch
  ## Force into a column vector, and cast the diagonal to single if
  ## necessary.
  D = D(:);
  if assingle, D = single (D); endif
  ## Randomly toggle the signs / spin complex entries
  if togglesign,
    if asreal,
      D .*= 1 - 2*floor (rand (N,1));
    else
      args = 2*pi*rand (N,1);
      D .*= cos (args) + i*sin (args);
    endif
  endif
  ## If the mode is negative, reverse D.
  if mode < 0, D = D(N:-1:1); endif
endfunction
```

Listing 4.4: Generate a random vector with certain value distributions, equivalent to LAPACK's xLATM1 test routines.

Figure 4.1: The generated systems $Ax = b$ include 7% to 15% nearly-singular matrices as shown by this histogram of $\kappa_\infty(A)$ for 1 million generated tests, computed in working precision. The blue line, $\kappa_\infty(A) = \varepsilon_w$, is the boundary where matrices to the right could be perturbed to singular by a ε_w-sized perturbation (normwise). The percentages give the fraction on either side of that line.

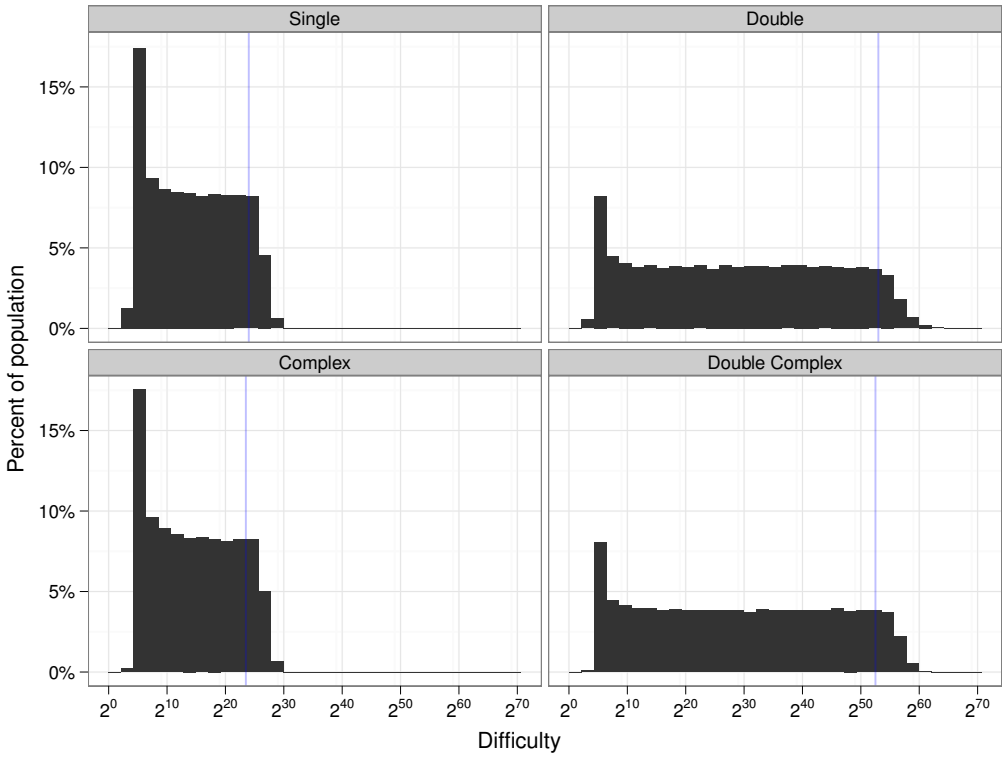


Figure 4.2: This histogram of $\kappa(A, x, b)$ for 1 million generated tests, computed in working precision, shows that 10% to 21% of generated systems are "difficult" to solve for an accurate x normwise. Solving $Ax = b$ with a backwards-stable method implies nothing about the error in the largest component of x for systems to the right of the vertical blue line, $\kappa(A, x, b) \geq 1/\varepsilon_w$.

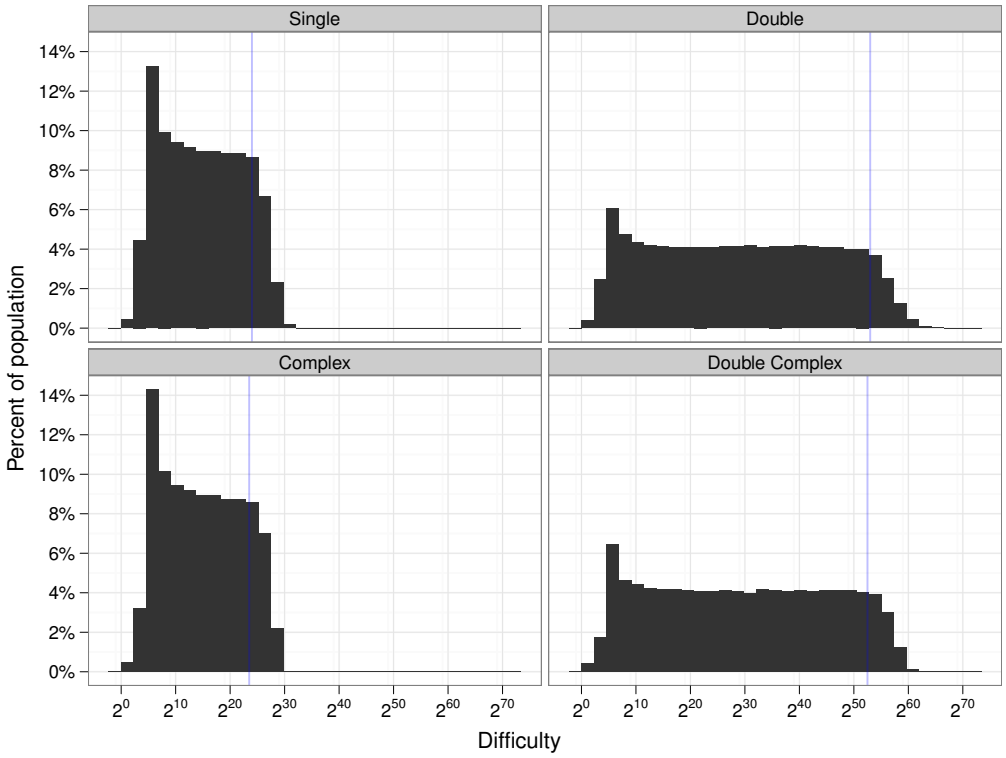
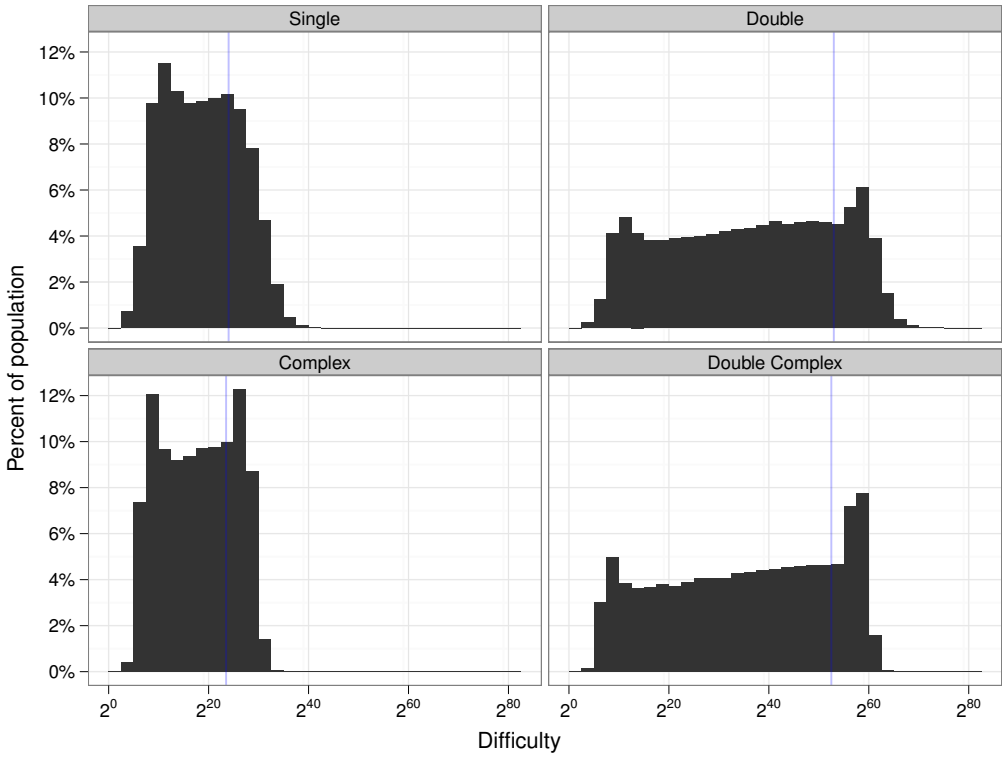


Figure 4.3: Our test generator is skewed strongly towards producing difficult componentwise problems as shown by the histogram of $\mathrm{ccond}(A, x, b)$ for 1 million generated tests, computed in working precision. Componentwise accuracy requires that *every* component of the computed solution is accurate. Generating x_0 from x`LATM1` systems with relatively small solution components and skews the data towards difficult. The vertical blue line separates well- from ill-conditioned, or $\mathrm{ccond}(A, x, b) \geq 1/\varepsilon_w$. From 20% to 30% are "difficult" to solve with high componentwise accuracy.



Precision	Kind	% Difficult
Single	"Numerically singular"	18.7%
	Normwise forward error	12.5%
	Componentwise forward error	29.5%
Complex	"Numerically singular"	20.3%
	Normwise forward error	15.0%
	Componentwise forward error	29.4%
Double	"Numerically singular"	10.0%
	Normwise forward error	7.0%
	Componentwise forward error	21.2%
Double Complex	"Numerically singular"	11.1%
	Normwise forward error	8.7%
	Componentwise forward error	21.7%

Table 4.1: Percentage of "difficult" cases in our generated dense test suite. The forward error difficulties depend on the entire system, and our generator produces some right-hand sides that push the relevant condition number just below the $1/\varepsilon_f$ threshold.

4.2 Results

The shortened results: Using extra precision within iterative refinement provides a dependable solution. The real question is how long you must wait for that solution. For large systems solved only a few times, the cost is utterly dominated by the factorization's $O(N^3)$ growth. A dependable solution is *practically free* by using targeted extra precision. For smaller systems, pre-factored systems, and systems solved many times, limiting the iteration counts to small numbers (Table 4.2) and removing Demmel et al. [37]'s condition estimators makes dependable solutions quite inexpensive.

In more detail, we applied Chapter 3's algorithm *without* scaling as given in Listing 3.1 to Section 4.1's test systems. Refinement targeted the most ambitious result, a small componentwise forward error. The generated test matrices already are well-scaled compared to practical matrices. Including numerical scaling here does not affect results significantly, and leaving out numerical scaling provides more stress on the basic refinement algorithm. Chapter 5's sparse results do include simple numerical scaling.

Reducing the backward error from ε_w to ε_w^2 with the slowest decrease permitted (a factor of 0.9) requires $\lceil \log_{0.9} 2^{-24} \rceil = 158$ iterations for single precision and $\lceil \log_{0.9} 2^{-53} \rceil = 349$ for double precision. This is far too high a limit for practical use. The first batch of results in Section 4.2.1 permits up to n iterations, where $n = 30$ is the dimension of the generated test system. This also is too high a limit to be practical because it increases the asymptotic complexity of refinement to $O(n^3)$. This large limit shows that experiments to 158 or 349 iterations are not necessary. Section 4.2.2 limits the number of iterations to 5 for single

Precision	Limit
single	5
double	10
single complex	7
double complex	15

Table 4.2: Imposed iteration limits per precision

precision and 7 for single complex. Doubling the precision roughly doubles the number of steps necessary in the worst case, so Section 4.2.2 limits double precision to 10 iterations and double complex to 15. Table 4.2 summarizes the imposed limits.

We do not include results from larger systems, but the iteration limits applied in Section 4.2.2 appear sufficient for n up to 1000 at least. As the dimension increases, sampling truly difficult systems becomes vastly more expensive.

Each estimated condition number requires at least two solutions with A and two with A^T. Estimating normwise and componentwise forward error condition numbers for each right-hand side would add at least eight solutions of a linear system to the cost. For dense matrices, this may equate to two to four refinement steps including the cost of the extended residual. Even for double complex, a significant number of 30×30 systems are solved in fewer than four steps, and condition estimates can double the running time of refinement. As the dimension increases, the cost is dominated by factorization, but parallel and sparse implementations benefit from not solving by A^T and keeping solve steps to a minimum.

Eliminating *all* conditioning considerations appears impossible. Testing that

$$\frac{\min_j |D_i(j,j)|}{\max_j |D_i(j,j)|} \geq \varepsilon_f c_i$$

is crucial for rendering a dependable componentwise solution. Without that test, some ill-conditioned systems produced componentwise errors above Equation (3.6.12)'s bound but were flagged as accurate.

4.2.1 Permitting up to n iterations

Figure 4.4 provides the error results for single precision when refinement targets the componentwise forward error. Each box in the plot corresponds to a termination state for the componentwise forward error (right label) and an error measure (top label). The termination states are explained in Table 4.3, and the error measures are described in Table 4.4. Only the "Converged" state is accepted as an accurate componentwise forward error; the other states are flagged as potentially inaccurate for the componentwise forward error measure. "Iteration Limit" applies when the system still is making progress at the large iteration limit (30).

State	Description
Converged	Here $\|D_i^{-1} dy_i\|_\infty \leq$ Equation (3.6.4)'s bound and the solution is accepted as having tiny componentwise forward error.
No Progress	The system has not converged and $\|dy_i/dy_{i+1}\|_\infty > c = 0.9$.
Unstable	The componentwise backward error did not achieve Equation (3.5.7)'s limit before the iteration limit.
Iteration Limit	None of the above applied at the iteration limit, so the system was still making slow progress.

Table 4.3: Description of the termination state labels along the right of the error plots.

Error Measure	Description	Difficulty
nberr	Normwise backward error computed with Equation (2.2.2) *using the doubled-precision solution.*	$g\kappa(A, x)$
colberr	Columnwise backward error computed with Equation (2.2.6) *using the doubled-precision solution.*	$g\kappa(A, x)$
cberr	Componentwise backward error computed with Equation (2.2.5) *using the doubled-precision solution.*	$g\kappa(A, x)$
nferr	Normwise forward error computed with Equation (2.2.7) using the true solution computed by Section 4.1.1's generator.	$g\,\mathrm{colcond}(A, x)$
nferrx	Normwise forward error relative computed by Equation (2.2.8), measured relative to the refined result rather than the typically unknown true solution.	$g\,\mathrm{colcond}(A, x)$
cferr	Componentwise forward error from Equation (2.2.9).	$g\,\mathrm{ccond}(A, x)$
cferrx	Componentwise forward error from Equation (2.2.10), relative to the refined solution.	$g\,\mathrm{ccond}(A, x)$

Table 4.4: Description of the error labels along the top of the error plots. The difficulties are the product of Equation (3.3.5)'s growth factor g and the condition numbers from Table 2.3. We use $\kappa(A, x)$ rather than $\mathrm{ccolcond}(A^{-1}, x)$ for the backward errors.

Precision	Accepted	> 30 iterations	Unstable	No progress
Single	94.17%	1.97%	3.60%	0.25%
Single complex	95.27%	1.30%	3.05%	0.37%
Double	95.27%	1.02%	2.27%	1.45%
Double complex	96.94%	0.43%	1.86%	0.77%

Table 4.5: Population breakdown for refinement termination when iteration is permitted up to n steps and the algorithm considers $\mathrm{cberr}(A, y_i, b)$ for stability and $\|D_{|y|_i}^{-1} dy_i\|_\infty$ for convergence. All accepted solutions have error below the forward componentwise error bound.

Precision	Accepted		> 30 iterations		Unstable		No progress	
	well	ill	well	ill	well	ill	well	ill
Single	79.21%	**14.96%**	**0.17%**	1.80%	**0.95%**	2.66%	**0.02%**	0.23%
Single complex	76.45%	**18.82%**	**0.01%**	1.30%	**0.71%**	2.34%	**0.00%**	0.37%
Double	86.66%	**8.60%**	**0.06%**	0.95%	**0.43%**	1.84%	**0.54%**	0.91%
Double complex	85.48%	**11.46%**	**0.00%**	0.43%	**0.36%**	1.50%	**0.05%**	0.72%

Table 4.6: Population breakdown for refinement termination against conditioning when iteration is permitted up to n steps and the algorithm considers $\mathrm{cberr}(A, y_i, b)$ for stability and $\|D_{|y|_i}^{-1} dy_i\|_\infty$ for convergence. All accepted solutions have error below the forward componentwise error bound. A column is labeled "well" when the product of the element growth $g_c = \max_j \frac{\left(\max_{1 \le k \le j} \max_i |L|(i,k)\right) \cdot (\max_i |U|(i,j))}{\max_i |A|(i,j)}$, the dimensional factor $\sqrt{n}$, and the condition number for the componentwise forward error $\mathrm{ccond}(A, x) = \left\| |(AD_{|x|})^{-1}| \cdot |AD_{|x|}| \right\|_\infty$ is below $1/\varepsilon_f$. Here $\varepsilon_f = 2^{-24}$ for single precision, $\sqrt{2} \cdot 2^{-24}$ for single complex precision, 2^{-53} for double precision, and $\sqrt{2} \cdot 2^{-53}$ for double complex precision. The first bold column shows the population of samples we gain over Demmel et al. [37] by using the convergence criteria rather than a condition estimator. The later bold columns are the populations we *lose* by not accepting only because the estimated difficulty is small.

The boxes in each error plot show a two-dimensional histogram of the error measure against the corresponding difficulty. The population counts within the histogram are a function of our test generator. The boundaries of the populated regions are more useful to verify that refinement works. The horizontal red line shows Chapter 3's bound for each error measure. There must be no points above that line in the "Converged" state for the target error measures. The horizontal blue lines denote precisions, the residual computation precision ε_r for the backward errors and the returned result precision ε_w for the forward error. Note that the backward errors are computed from the doubled-precision solution maintained during refinement and *not* the returned, working-precision solution. The vertical blue line denotes $1/\varepsilon_f$, the difficulty at which the factorization may break down.

Table 4.4 provides the difficulty measure for each error measure; the difficulty multiplies the growth factor from Equation (3.3.5) by the condition number corresponding to the error measure. We use $\kappa(A, x)$ rather than Equation (3.5.8)'s ccolcond(A^{-1}, x) for the backward errors because estimating the former on Chapter 5's sparse tests is more tractable. The boxes show all the samples within a given convergence category. So a box with the top label "nberr" and right label "No Progress" shows the normwise backward error for all those systems that failed to converge to a tiny componentwise forward error due to lack of progress.

Table 4.5 breaks down the later figures into population percentages. Each entry provides the percentage of our test cases for the given precision that terminate in the top state. Over 94% are accepted and have small componentwise forward error. Table 4.6 further breaks down the results by the estimated difficulty used in [37] to determine acceptance. We gain more than 8% of the population in additional (and correct) acceptance, while losing less than 1%.

Interpreting Figure 4.5, we see that refining the doubled-precision solution drives the backward errors to their limit of nearly $\varepsilon_r \leq \varepsilon_w^2$, and hence the forward error for sufficiently well-conditioned problems remains below the forward error limit. Basing acceptance on the final state rather than condition estimation provides a good solution to some systems that appear ill-conditioned. The componentwise forward errors do approach dangerously close to their limit.

The systems that are not-too-ill-conditioned and fail to make progress have relatively small components in the solution. They were generated by choosing a random b vector rather than a random x and were not in Demmel et al. [37]'s test suite. All those declared unstable or hit the iteration limit are within a factor of 10 of being considered ill-conditioned.

Single-precision complex results are summarized in Figure 4.5. None of the systems fail to make progress, possibly because of the slightly looser convergence bound. The precisions used in complex bounds are larger than the corresponding real precisions by a factor of $\sqrt{2}$ to account for the extra operations. Another possibility is that the test generator simply does not hit those "difficult" cases and only "easy" or "catastrophic" cases. In some ways, complex systems are equivalent to real systems of twice the dimension. Sampling difficult cases is more difficult.

Figure 4.6 shows double precision and Figure 4.7 shows double complex precision results.

All validate the method across multiple precisions. Tests with larger systems are time-consuming and present less interesting results; the test generator does not sample enough difficult larger systems.

4.2.2 Limiting the number of iterations

Figures 4.8, 4.9, 4.10, and 4.11 take a similar form to the error plots but summarize the number of iterations required for reaching different error targets. In these plots, each column of boxes is independent. The two-dimensional histograms count the final number of iterations for each system targeting the error measure along the top and terminating in the state on the right. The horizontal red line shows our eventual iteration limits (Table 4.2).

Figure 4.9 for single-precision complex results shows an oddity in the normwise forward error. The "easy" systems that converged but take longer than expected terminate as unstable when targeting the componentwise forward error. Similar issues appear in Figure 4.11 for double-precision complex systems. The tiny handful of systems (around 10) in each that fail to converge within our limits are worth further consideration but might not present a strong argument for raising the limit.

Given these error limits, Figures 4.12, 4.14, 4.13, and 4.15 show the effects enforcing the iteration limit. More systems hit the iteration limit, but none are accepted with componentwise forward error larger than the bound.

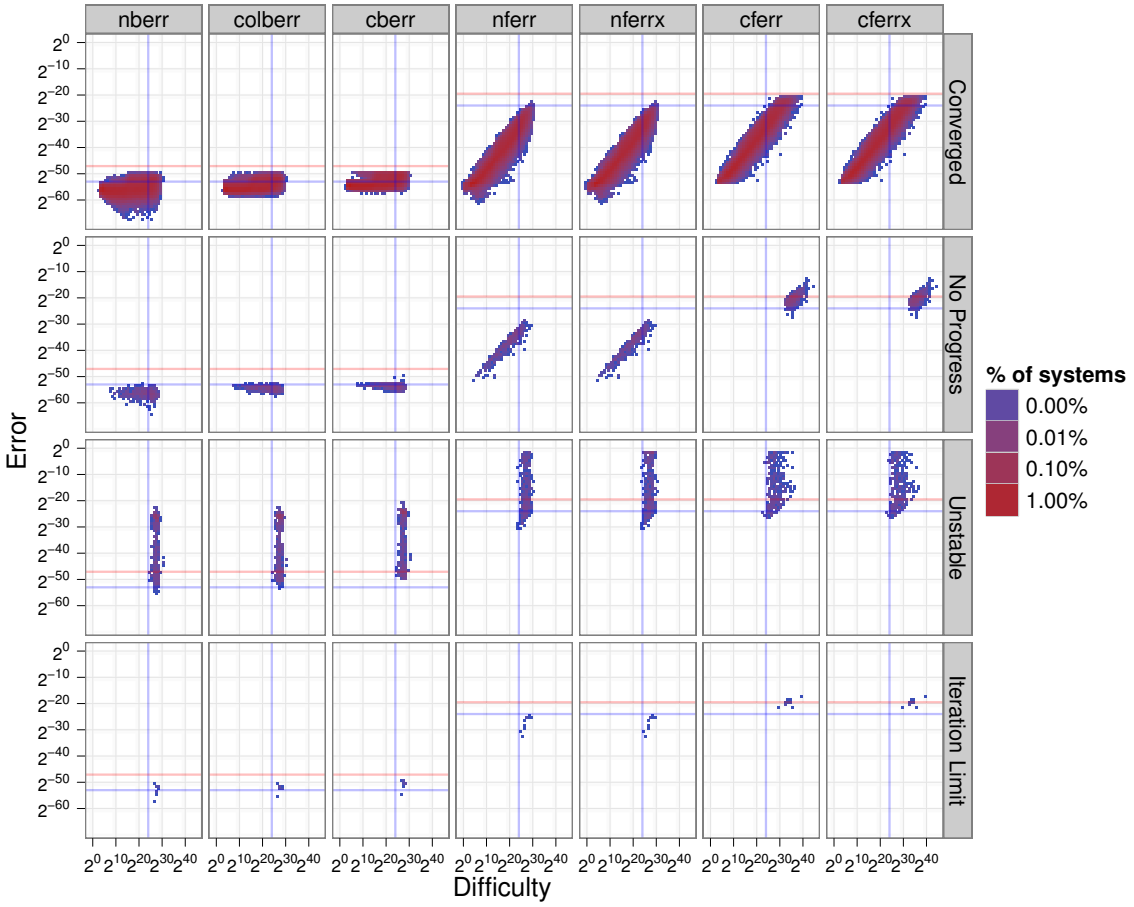


Figure 4.4: Two-dimensional histograms showing the error populations relative to the system's difficulty in *single* precision when iteration is permitted up to n steps and the algorithm considers $\mathrm{cberr}(A, y_i, b)$ for stability and $\|D_{|y|}{}_i^{-1} dy_i\|_\infty$ for convergence. The box labels on the right show the convergence status for the componentwise forward error. The labels on the top denote the error metric. A system appears in every box across a row but only once per column. The difficulty for each column is the product of the element growth $g_c = \max_j \frac{\left(\max_{1\le k\le j} \max_i |L|(i,k)\right)\cdot\left(\max_i |U|(i,j)\right)}{\max_i |A|(i,j)}$, the dimensional factor $\sqrt{n}$, and the relevant condition number. The condition number used for all the backward errors here is $\kappa_\infty(A^{-1}) = \kappa_\infty(A)$. The condition number for the normwise forward error is $\mathrm{colcond}(A, x) = \frac{\left\|A^{-1}\cdot(1_r \max|A|)\,|x|\right\|_\infty}{\|x\|_\infty}$, and the condition number for the componentwise forward error is $\mathrm{ccond}(A, x) = \left\||(AD_{|x|})^{-1}| \cdot |AD_{|x|}|\right\|_\infty$. The horizontal blue lines denote the relevant precision ($\varepsilon_r = 2^{-53}$ for backward errors, $\varepsilon_w = 2^{-24}$ for forward errors). Horizontal red lines indicate bounds. The vertical blue line denotes $1/\varepsilon_w = 1/\varepsilon_f$, where ε_f is the precision used to factor A.

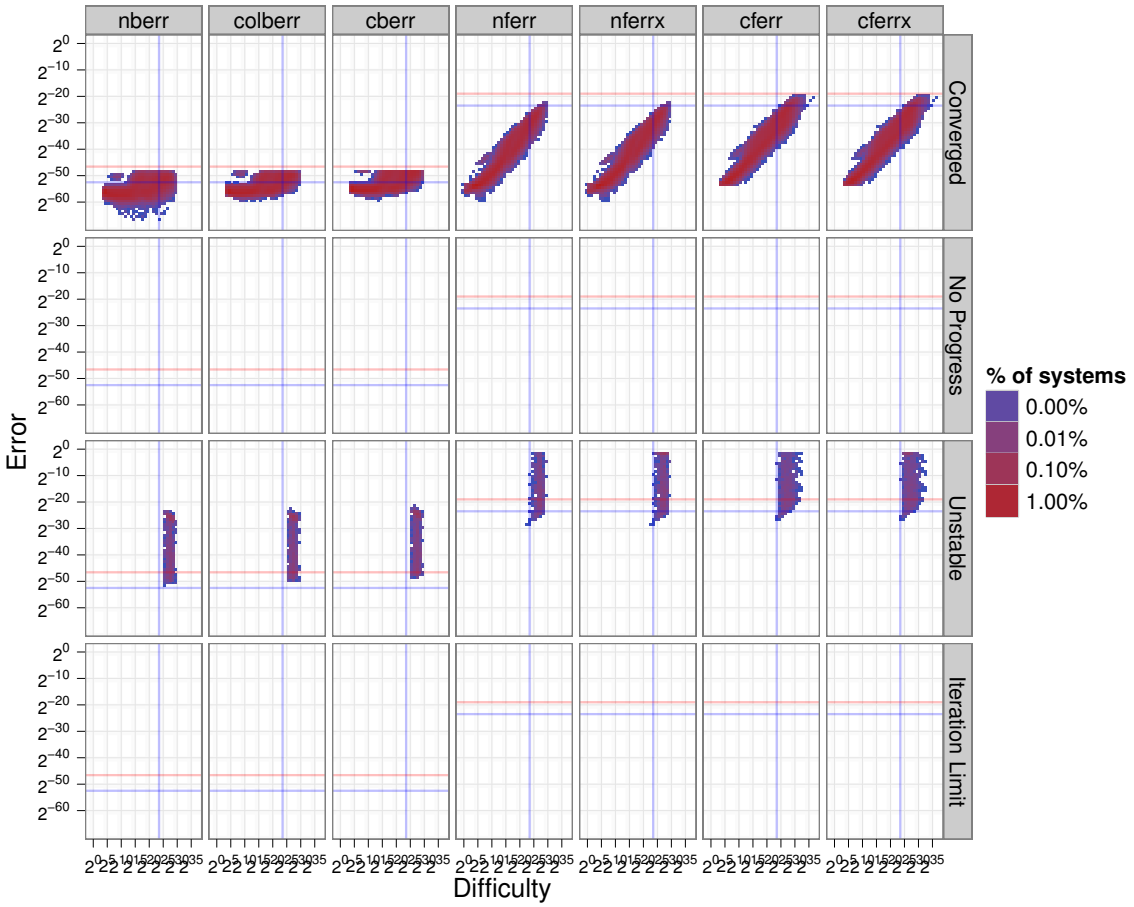


Figure 4.5: Two-dimensional histograms showing the error populations relative to the system's difficulty in *single complex* precision when iteration is permitted up to n steps and the algorithm considers $\mathrm{cberr}(A, y_i, b)$ for stability and $\|D_{|y|}{}_i^{-1} dy_i\|_\infty$ for convergence. The box labels on the right show the convergence status for the componentwise forward error. The labels on the top denote the error metric. A system appears in every box across a row but only once per column. The difficulty for each column is the product of the element growth $g_c = \max_j \frac{\left(\max_{1\le k\le j} \max_i |L|(i,k)\right)\cdot\left(\max_i |U|(i,j)\right)}{\max_i |A|(i,j)}$, the dimensional factor $\sqrt{n}$, and the relevant condition number. The condition number used for all the backward errors here is $\kappa_\infty(A^{-1}) = \kappa_\infty(A)$. The condition number for the normwise forward error is $\mathrm{colcond}(A, x) = \frac{\left\|A^{-1}\cdot(1_r \max |A|)\,|x|\right\|_\infty}{\|x\|_\infty}$, and the condition number for the componentwise forward error is $\mathrm{ccond}(A, x) = \left\|\,|(AD_{|x|})^{-1}| \cdot |AD_{|x|}|\,\right\|_\infty$. The horizontal blue lines denote the relevant precision ($\varepsilon_r = \sqrt{2}\cdot 2^{-53}$ for backward errors, $\varepsilon_w = \sqrt{2}\cdot 2^{-24}$ for forward errors). Horizontal red lines indicate bounds. The vertical blue line denotes $1/\varepsilon_w = 1/\varepsilon_f$, where ε_f is the precision used to factor A.

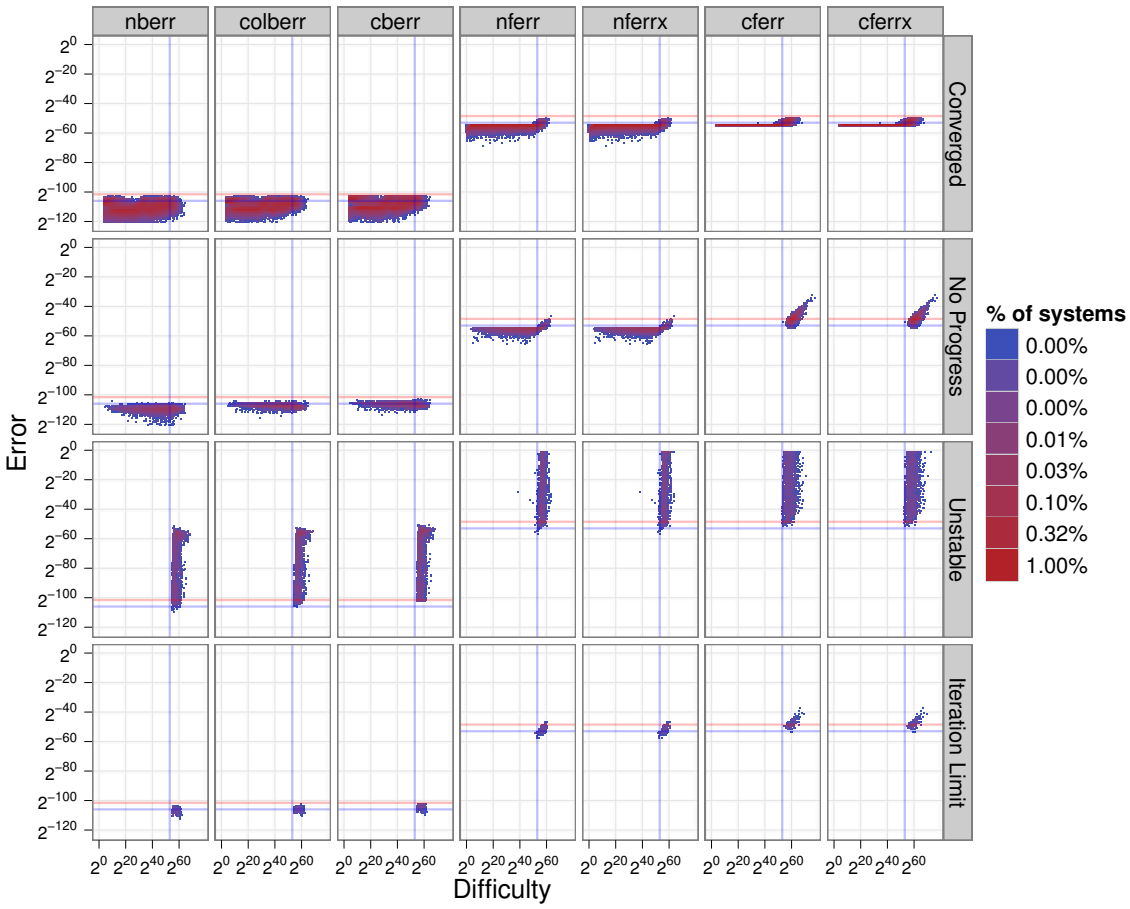


Figure 4.6: Two-dimensional histograms showing the error populations relative to the system's difficulty in *double* precision when iteration is permitted up to n steps and the algorithm considers $\mathrm{cberr}(A, y_i, b)$ for stability and $\|D_{|y|}{}_i^{-1} dy_i\|_\infty$ for convergence. The box labels on the right show the convergence status for the componentwise forward error. The labels on the top denote the error metric. A system appears in every box across a row but only once per column. The difficulty for each column is the product of the element growth $g_c = \max_j \frac{\left(\max_{1\le k\le j} \max_i |L|(i,k)\right)\cdot\left(\max_i |U|(i,j)\right)}{\max_i |A|(i,j)}$, the dimensional factor $\sqrt{n}$, and the relevant condition number. The condition number used for all the backward errors here is $\kappa_\infty(A^{-1}) = \kappa_\infty(A)$. The condition number for the normwise forward error is $\mathrm{colcond}(A, x) = \frac{\left\|A^{-1}\cdot(1_r \max|A|)\,|x|\right\|_\infty}{\|x\|_\infty}$, and the condition number for the componentwise forward error is $\mathrm{ccond}(A, x) = \left\|\,|(AD_{|x|})^{-1}| \cdot |AD_{|x|}|\,\right\|_\infty$. The horizontal blue lines denote the relevant precision ($\varepsilon_r = 2^{-106}$ for backward errors, $\varepsilon_w = 2^{-53}$ for forward errors). Horizontal red lines indicate bounds. The vertical blue line denotes $1/\varepsilon_w = 1/\varepsilon_f$, where ε_f is the precision used to factor A.

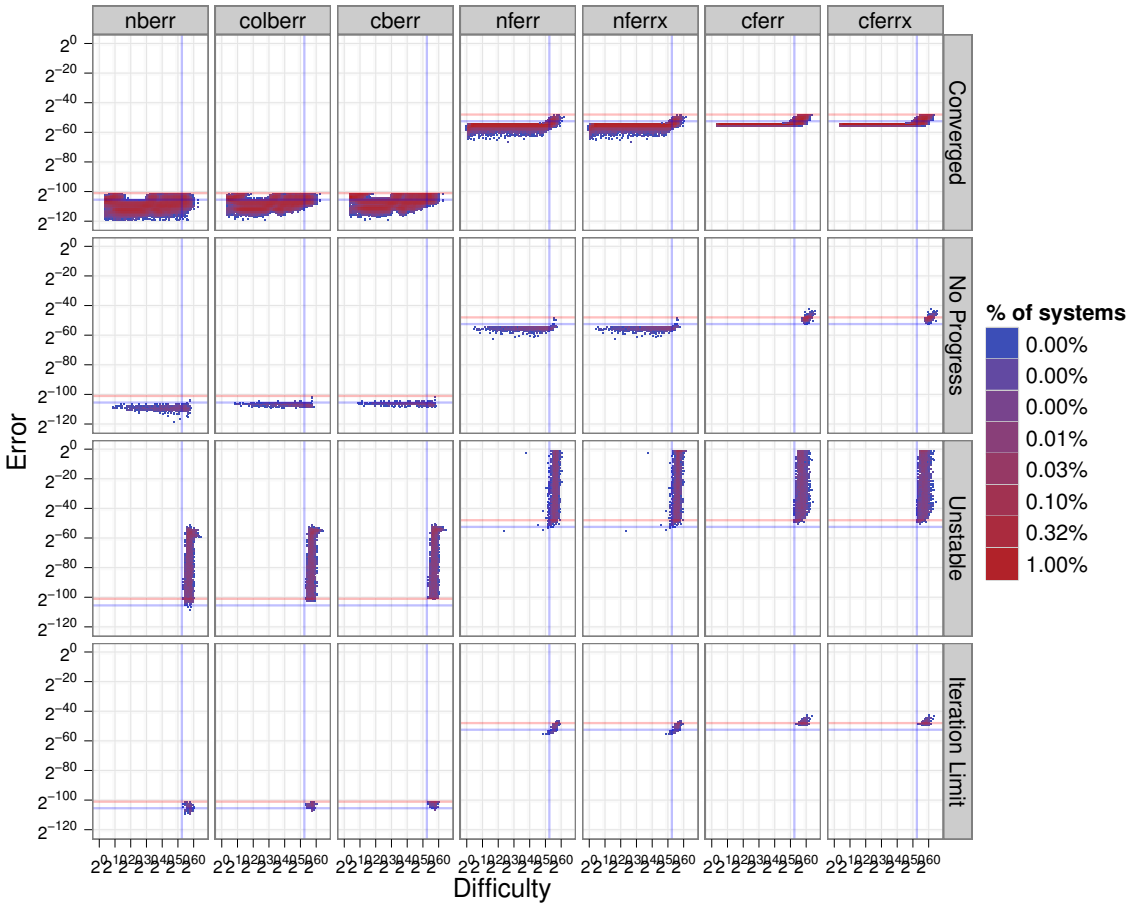


Figure 4.7: Two-dimensional histograms showing the error populations relative to the system's difficulty in *double complex* precision when iteration is permitted up to n steps and the algorithm considers $\mathrm{cberr}(A, y_i, b)$ for stability and $\|D_{|y|_i}^{-1} dy_i\|_\infty$ for convergence. The box labels on the right show the convergence status for the componentwise forward error. The labels on the top denote the error metric. A system appears in every box across a row but only once per column. The difficulty for each column is the product of the element growth $g_c = \max_j \frac{\left(\max_{1\le k\le j} \max_i |L|(i,k)\right)\cdot\left(\max_i |U|(i,j)\right)}{\max_i |A|(i,j)}$, the dimensional factor $\sqrt{n}$, and the relevant condition number. The condition number used for all the backward errors here is $\kappa_\infty(A^{-1}) = \kappa_\infty(A)$. The condition number for the normwise forward error is $\mathrm{colcond}(A, x) = \frac{\left\|A^{-1}\cdot(1_r \max |A|)\,|x|\right\|_\infty}{\|x\|_\infty}$, and the condition number for the componentwise forward error is $\mathrm{ccond}(A, x) = \left\|\,|(AD_{|x|})^{-1}| \cdot |AD_{|x|}|\,\right\|_\infty$. The horizontal blue lines denote the relevant precision ($\varepsilon_r = \sqrt{2}\cdot 2^{-106}$ for backward errors, $\varepsilon_w = \sqrt{2}\cdot 2^{-53}$ for forward errors). Horizontal red lines indicate bounds. The vertical blue line denotes $1/\varepsilon_w = 1/\varepsilon_f$, where ε_f is the precision used to factor A.

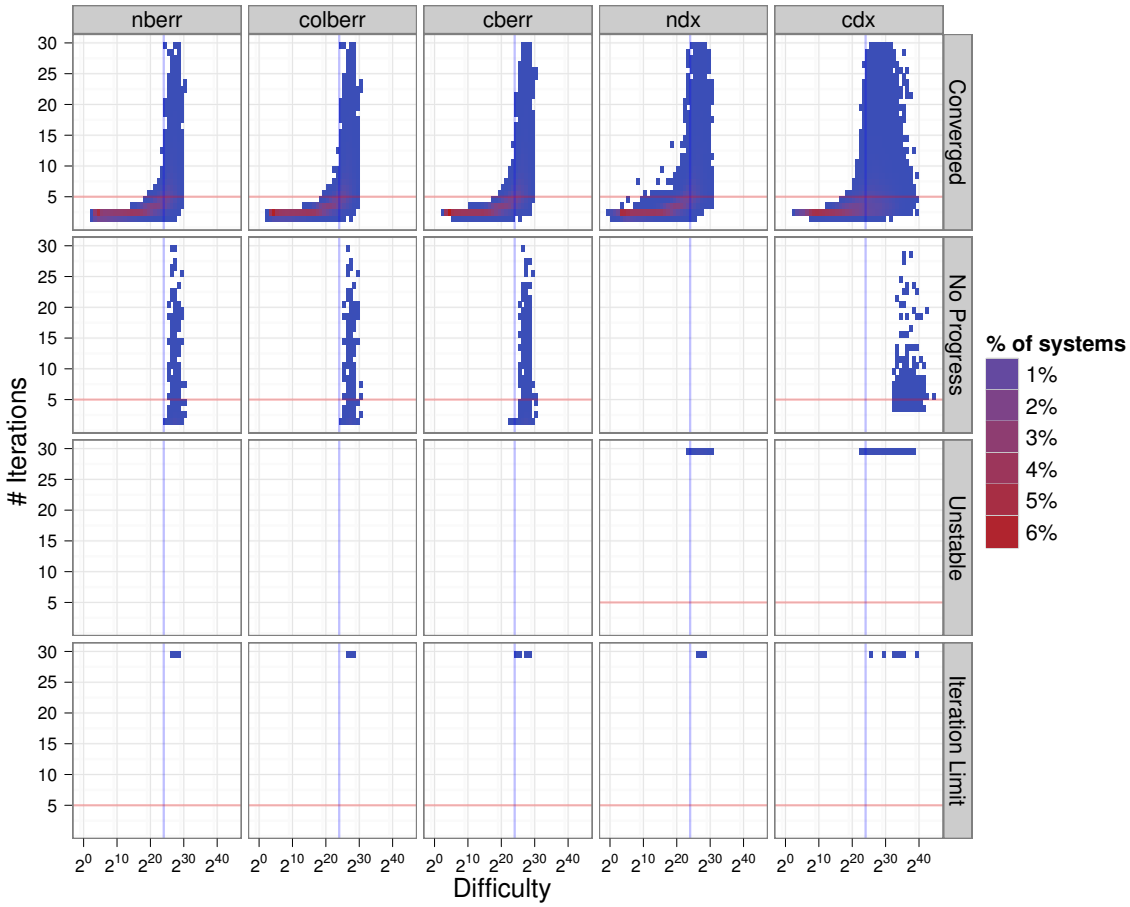


Figure 4.8: Two-dimensional histograms showing the iteration counts relative to the system's difficulty in *single* precision when iteration is permitted up to n steps. The box labels on the right show the convergence status for the componentwise forward error. The labels on the top denote the metric being monitored; each column can be considered a separate run of refinement targeting convergence of the top label. Here, "ndx" implies monitoring $\|dy_i\|_\infty / \|y_i\|_\infty$ and "cdx" implies monitoring $\|dy_i / y_i\|_\infty$. The difficulty for each column is the product of the element growth $g_c = \max_j \frac{\left(\max_{1 \le k \le j} \max_i |L|(i,k)\right) \cdot \left(\max_i |U|(i,j)\right)}{\max_i |A|(i,j)}$, the dimensional factor $\sqrt{n}$, and the relevant condition number. The condition number used for all the backward errors here is $\kappa_\infty(A^{-1}) = \kappa_\infty(A)$. The condition number for the normwise forward error is $\operatorname{colcond}(A, x) = \frac{\left\| A^{-1} \cdot (1_r \max |A|) \, |x| \right\|_\infty}{\|x\|_\infty}$, and the condition number for the componentwise forward error is $\operatorname{ccond}(A, x) = \left\| |(AD_{|x|})^{-1}| \cdot |AD_{|x|}| \right\|_\infty$. The horizontal red line in the lower plot shows where the iteration will be cut in Section 4.2.2. The vertical blue line denotes $1/\varepsilon_w = 1/\varepsilon_f$, where $\varepsilon_f = 2^{-24}$ is the precision used to factor A.

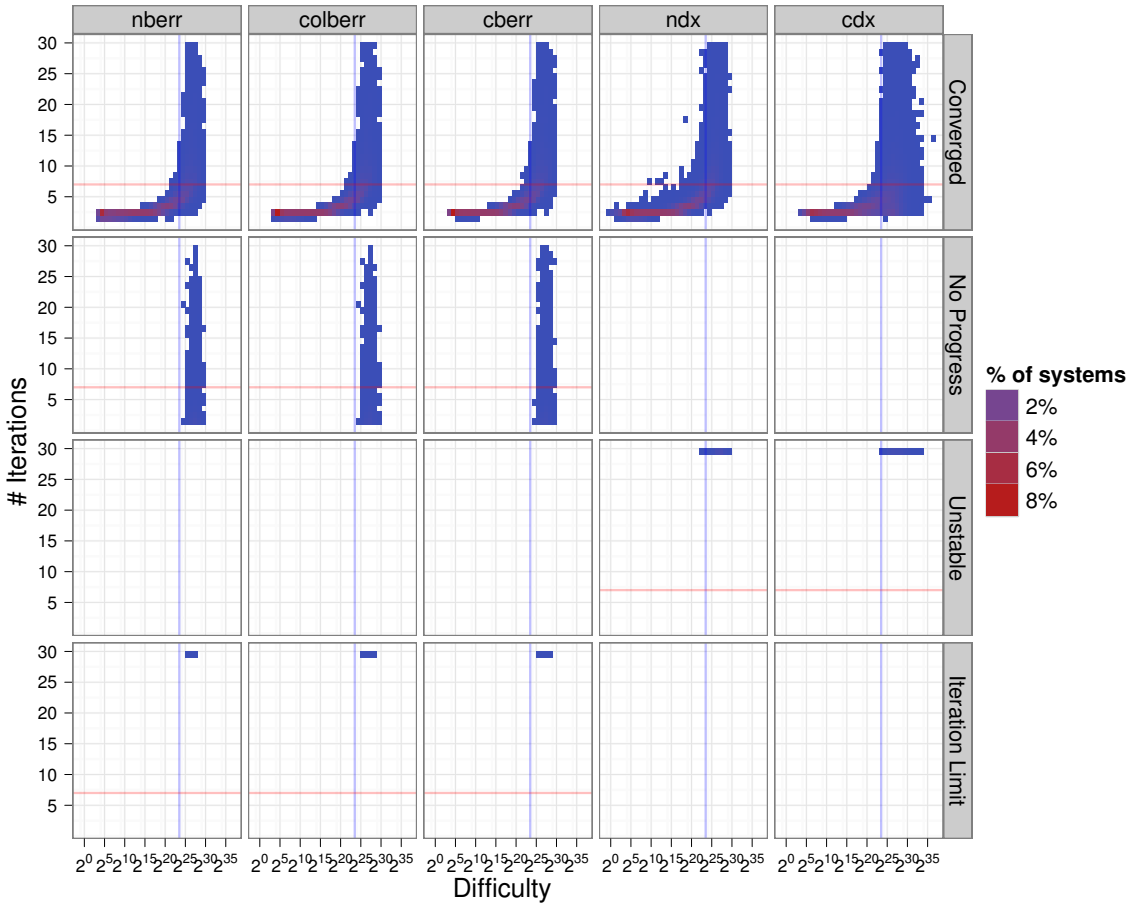


Figure 4.9: Two-dimensional histograms showing the iteration counts relative to the system's difficulty in *single complex* precision when iteration is permitted up to n steps. The box labels on the right show the convergence status for the componentwise forward error. The labels on the top denote the metric being monitored; each column can be considered a separate run of refinement targeting convergence of the top label. Here, "ndx" implies monitoring $\|dy_i\|_\infty / \|y_i\|_\infty$ and "cdx" implies monitoring $\|dy_i / y_i\|_\infty$. The difficulty for each column is the product of the element growth $g_c = \max_j \frac{\left(\max_{1 \le k \le j} \max_i |L|(i,k)\right) \cdot \left(\max_i |U|(i,j)\right)}{\max_i |A|(i,j)}$, the dimensional factor $\sqrt{n}$, and the relevant condition number. The condition number used for all the backward errors here is $\kappa_\infty(A^{-1}) = \kappa_\infty(A)$. The condition number for the normwise forward error is $\operatorname{colcond}(A, x) = \frac{\left\| A^{-1} \cdot (1_r \max |A|) \, |x| \right\|_\infty}{\|x\|_\infty}$, and the condition number for the componentwise forward error is $\operatorname{ccond}(A, x) = \left\| |(AD_{|x|})^{-1}| \cdot |AD_{|x|}| \right\|_\infty$. The horizontal red line in the lower plot shows where the iteration will be cut in Section 4.2.2. The vertical blue line denotes $1/\varepsilon_w = 1/\varepsilon_f$, where $\varepsilon_f = \sqrt{2} \cdot 2^{-24}$ is the precision used to factor A.

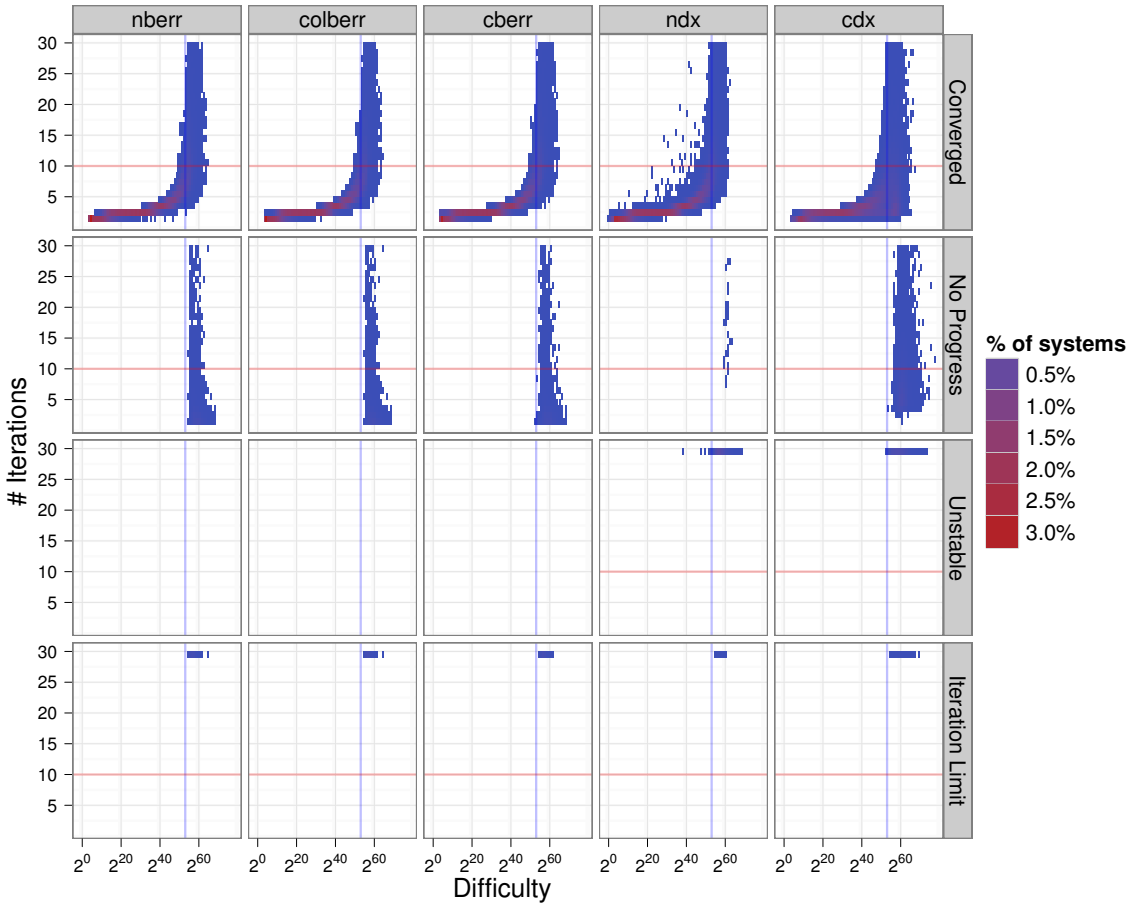


Figure 4.10: Two-dimensional histograms showing the iteration counts relative to the system's difficulty in *double* precision when iteration is permitted up to n steps. The box labels on the right show the convergence status for the componentwise forward error. The labels on the top denote the metric being monitored; each column can be considered a separate run of refinement targeting convergence of the top label. Here, "ndx" implies monitoring $\|dy_i\|_\infty / \|y_i\|_\infty$ and "cdx" implies monitoring $\|dy_i/y_i\|_\infty$. The difficulty for each column is the product of the element growth $g_c = \max_j \frac{\left(\max_{1\le k\le j} \max_i |L|(i,k)\right)\cdot\left(\max_i |U|(i,j)\right)}{\max_i |A|(i,j)}$, the dimensional factor $\sqrt{n}$, and the relevant condition number. The condition number used for all the backward errors here is $\kappa_\infty(A^{-1}) = \kappa_\infty(A)$. The condition number for the normwise forward error is $\operatorname{colcond}(A,x) = \frac{\left\|A^{-1}\cdot(1_r \max|A|)\,|x|\right\|_\infty}{\|x\|_\infty}$, and the condition number for the componentwise forward error is $\operatorname{ccond}(A,x) = \left\||(AD_{|x|})^{-1}|\cdot|AD_{|x|}|\right\|_\infty$. The horizontal red line in the lower plot shows where the iteration will be cut in Section 4.2.2. The vertical blue line denotes $1/\varepsilon_w = 1/\varepsilon_f$, where $\varepsilon_f = 2^{-53}$ is the precision used to factor A.

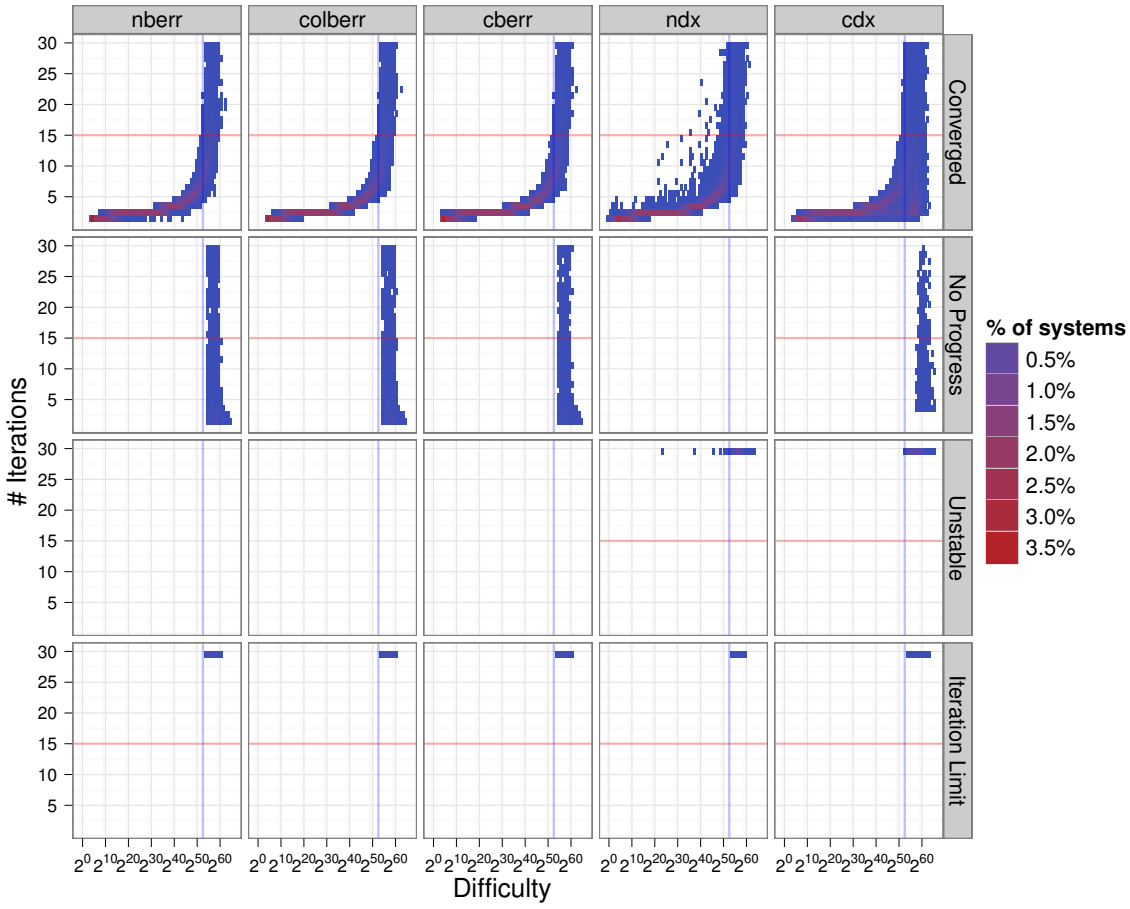


Figure 4.11: Two-dimensional histograms showing the iteration counts relative to the system's difficulty in *double complex* precision when iteration is permitted up to n steps. The box labels on the right show the convergence status for the componentwise forward error. The labels on the top denote the metric being monitored; each column can be considered a separate run of refinement targeting convergence of the top label. Here, "ndx" implies monitoring $\|dy_i\|_\infty/\|y_i\|_\infty$ and "cdx" implies monitoring $\|dy_i/y_i\|_\infty$. The difficulty for each column is the product of the element growth $g_c = \max_j \frac{\left(\max_{1\le k\le j} \max_i |L|(i,k)\right)\cdot\left(\max_i |U|(i,j)\right)}{\max_i |A|(i,j)}$, the dimensional factor $\sqrt{n}$, and the relevant condition number. The condition number used for all the backward errors here is $\kappa_\infty(A^{-1}) = \kappa_\infty(A)$. The condition number for the normwise forward error is $\operatorname{colcond}(A,x) = \frac{\left\|A^{-1}\cdot(1_r \max|A|)\,|x|\right\|_\infty}{\|x\|_\infty}$, and the condition number for the componentwise forward error is $\operatorname{ccond}(A,x) = \left\||(AD_{|x|})^{-1}|\cdot|AD_{|x|}|\right\|_\infty$. The horizontal red line in the lower plot shows where the iteration will be cut in Section 4.2.2. The vertical blue line denotes $1/\varepsilon_w = 1/\varepsilon_f$, where $\varepsilon_f = \sqrt{2}\cdot 2^{-53}$ is the precision used to factor A.

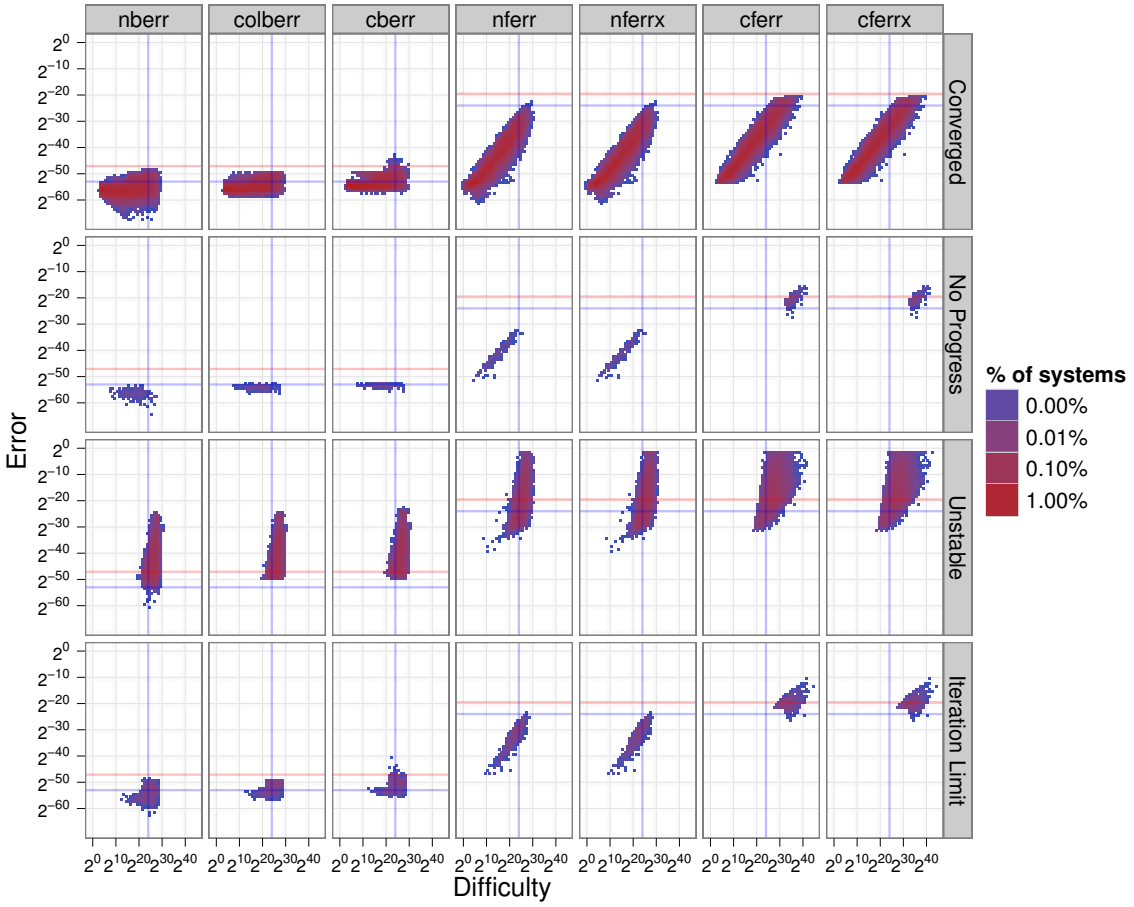


Figure 4.12: Two-dimensional histograms showing the error populations relative to the system's difficulty in *single* precision when iteration is permitted up to *five* steps and the algorithm considers $\mathrm{cberr}(A, y_i, b)$ for stability and $\|D_{|y|}{}_i^{-1} dy_i\|_\infty$ for convergence. The box labels on the right show the convergence status for the componentwise forward error. The labels on the top denote the error metric. A system appears in every box across a row but only once per column. The difficulty for each column is the product of the element growth $g_c = \max_j \frac{\left(\max_{1\le k\le j} \max_i |L|(i,k)\right)\cdot\left(\max_i |U|(i,j)\right)}{\max_i |A|(i,j)}$, the dimensional factor $\sqrt{n}$, and the relevant condition number. The condition number used for all the backward errors here is $\kappa_\infty(A^{-1}) = \kappa_\infty(A)$. The condition number for the normwise forward error is $\mathrm{colcond}(A, x) = \frac{\left\|A^{-1}\cdot(1_r \max|A|)\,|x|\right\|_\infty}{\|x\|_\infty}$, and the condition number for the componentwise forward error is $\mathrm{ccond}(A, x) = \left\|\,|(AD_{|x|})^{-1}| \cdot |AD_{|x|}|\,\right\|_\infty$. The horizontal blue lines denote the relevant precision ($\varepsilon_r = 2^{-53}$ for backward errors, $\varepsilon_w = 2^{-24}$ for forward errors). Horizontal red lines indicate bounds. The vertical blue line denotes $1/\varepsilon_w = 1/\varepsilon_f$, where ε_f is the precision used to factor A.

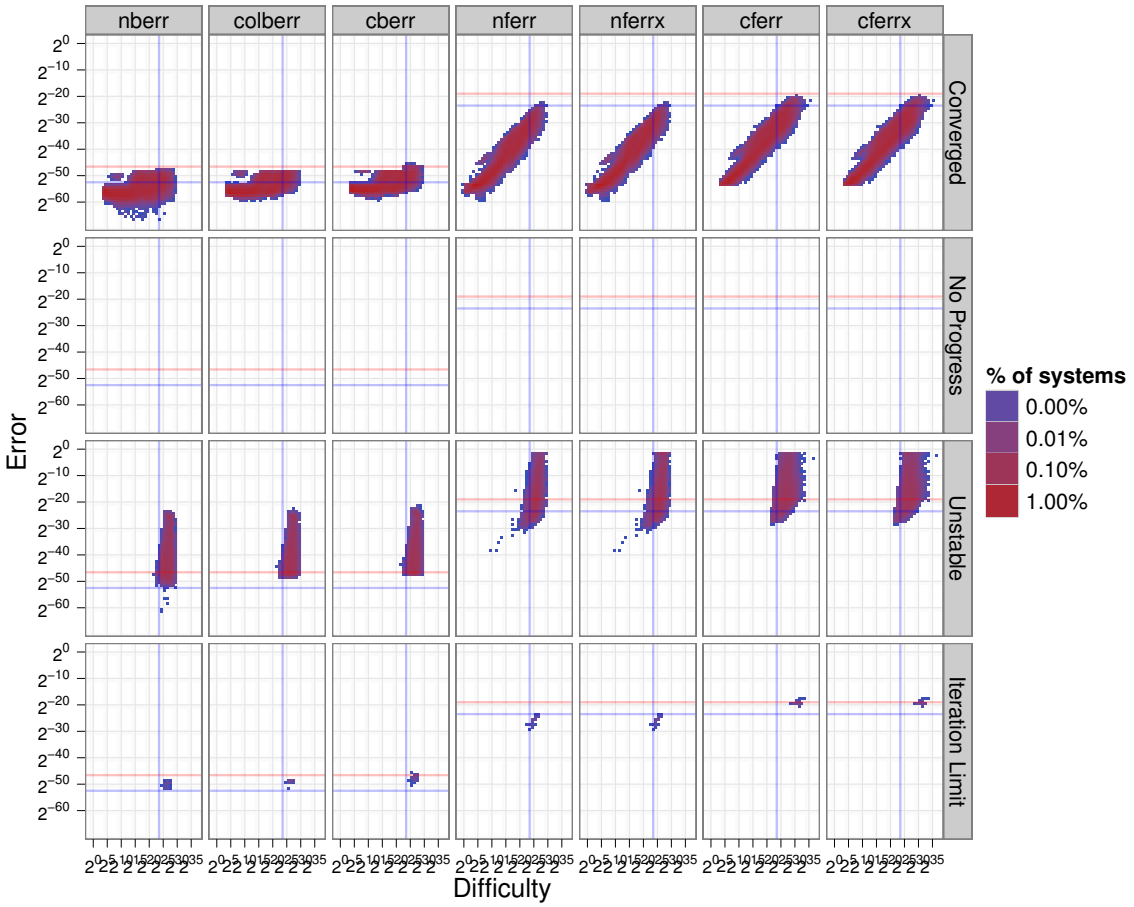


Figure 4.13: Two-dimensional histograms showing the error populations relative to the system's difficulty in *single complex* precision when iteration is permitted up to *seven* steps and the algorithm considers $\mathrm{cberr}(A, y_i, b)$ for stability and $\|D_{|y|}{}_i^{-1} dy_i\|_\infty$ for convergence. The box labels on the right show the convergence status for the componentwise forward error. The labels on the top denote the error metric. A system appears in every box across a row but only once per column. The difficulty for each column is the product of the element growth $g_c = \max_j \frac{\left(\max_{1\le k\le j} \max_i |L|(i,k)\right)\cdot\left(\max_i |U|(i,j)\right)}{\max_i |A|(i,j)}$, the dimensional factor $\sqrt{n}$, and the relevant condition number. The condition number used for all the backward errors here is $\kappa_\infty(A^{-1}) = \kappa_\infty(A)$. The condition number for the normwise forward error is $\mathrm{colcond}(A, x) = \frac{\left\|A^{-1}\cdot(1_r \max|A|)\,|x|\right\|_\infty}{\|x\|_\infty}$, and the condition number for the componentwise forward error is $\mathrm{ccond}(A, x) = \left\|\,|(AD_{|x|})^{-1}| \cdot |AD_{|x|}|\,\right\|_\infty$. The horizontal blue lines denote the relevant precision ($\varepsilon_r = \sqrt{2}\cdot 2^{-53}$ for backward errors, $\varepsilon_w = \sqrt{2}\cdot 2^{-24}$ for forward errors). Horizontal red lines indicate bounds. The vertical blue line denotes $1/\varepsilon_w = 1/\varepsilon_f$, where ε_f is the precision used to factor A.

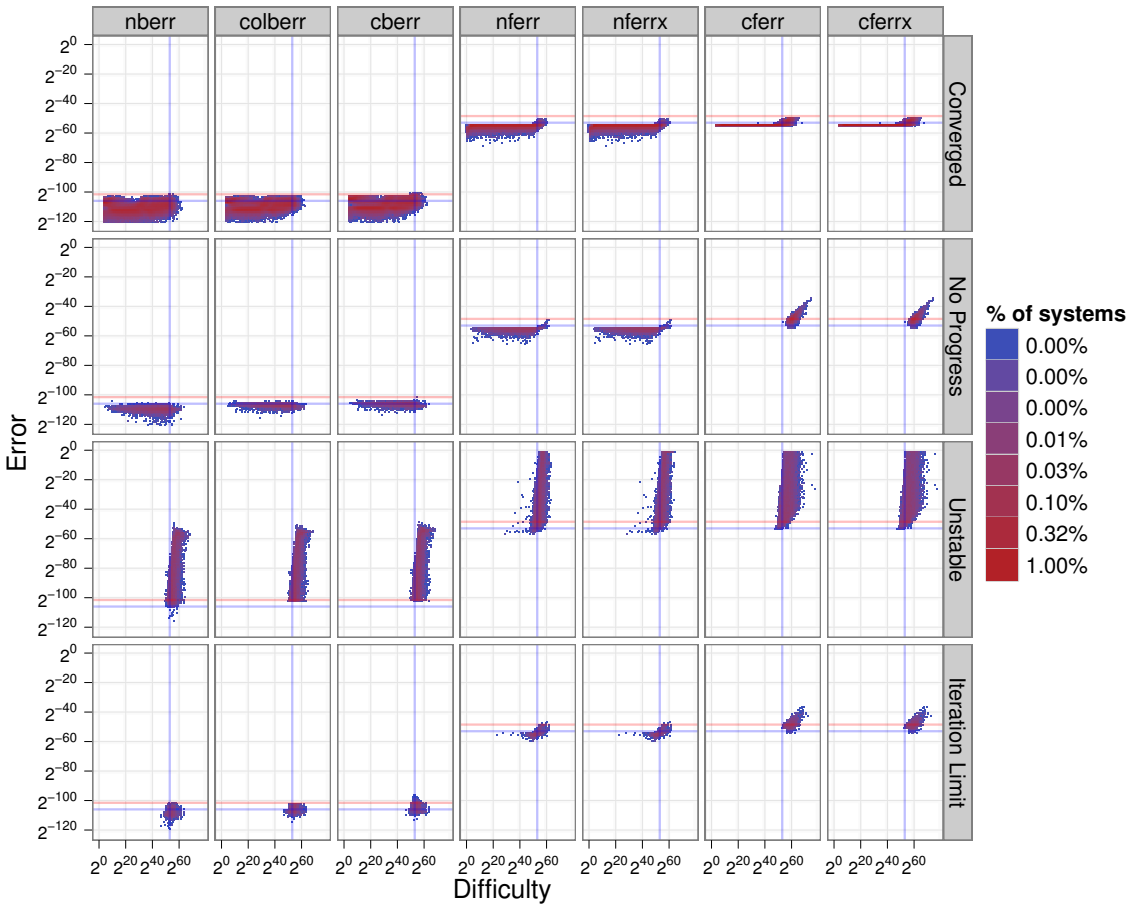


Figure 4.14: Two-dimensional histograms showing the error populations relative to the system's difficulty in *double* precision when iteration is permitted up to *ten* steps and the algorithm considers $\mathrm{cberr}(A, y_i, b)$ for stability and $\|D_{|y|}{}_i^{-1} dy_i\|_\infty$ for convergence. The box labels on the right show the convergence status for the componentwise forward error. The labels on the top denote the error metric. A system appears in every box across a row but only once per column. The difficulty for each column is the product of the element growth $g_c = \max_j \frac{\left(\max_{1\le k\le j} \max_i |L|(i,k)\right)\cdot\left(\max_i |U|(i,j)\right)}{\max_i |A|(i,j)}$, the dimensional factor $\sqrt{n}$, and the relevant condition number. The condition number used for all the backward errors here is $\kappa_\infty(A^{-1}) = \kappa_\infty(A)$. The condition number for the normwise forward error is $\mathrm{colcond}(A, x) = \frac{\left\|A^{-1}\cdot(1_r \max|A|)\,|x|\right\|_\infty}{\|x\|_\infty}$, and the condition number for the componentwise forward error is $\mathrm{ccond}(A, x) = \left\|\,|(AD_{|x|})^{-1}| \cdot |AD_{|x|}|\,\right\|_\infty$. The horizontal blue lines denote the relevant precision ($\varepsilon_r = 2^{-106}$ for backward errors, $\varepsilon_w = 2^{-53}$ for forward errors). Horizontal red lines indicate bounds. The vertical blue line denotes $1/\varepsilon_w = 1/\varepsilon_f$, where ε_f is the precision used to factor A.

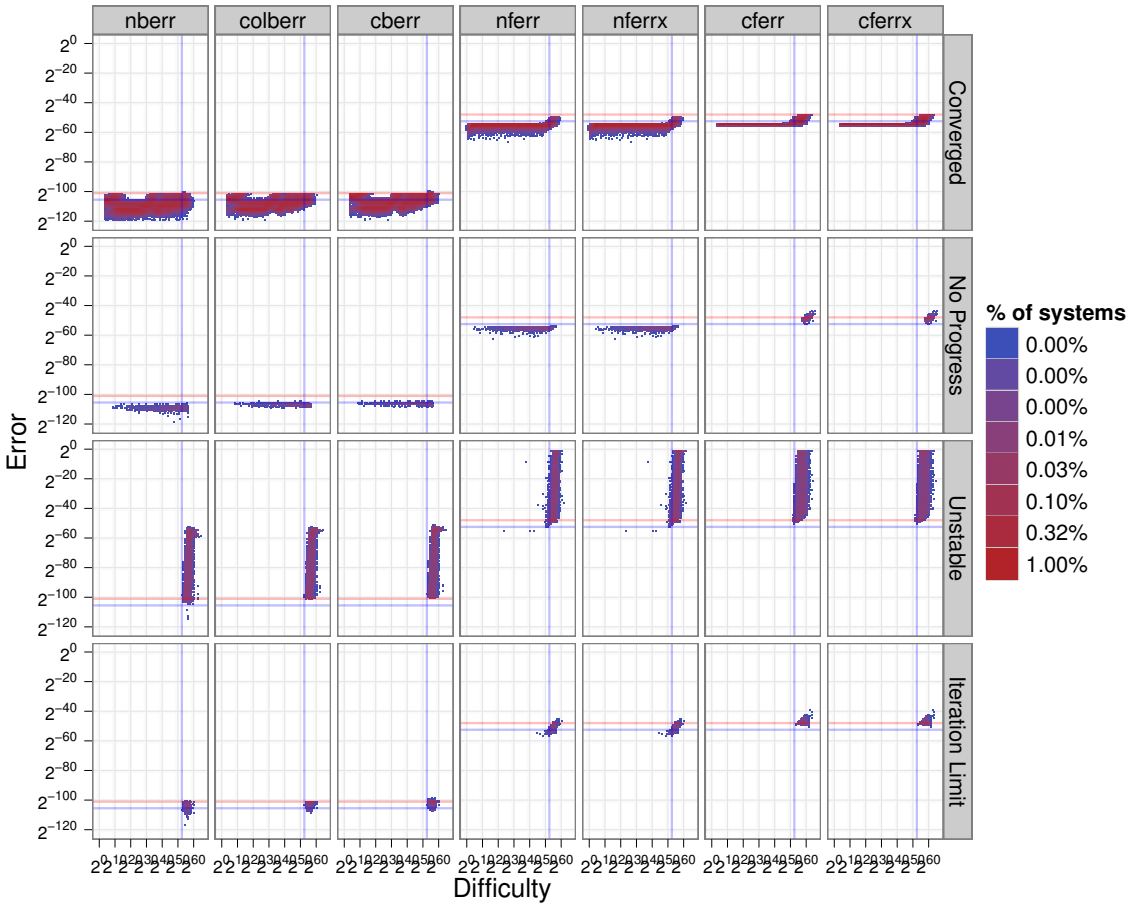


Figure 4.15: Two-dimensional histograms showing the error populations relative to the system's difficulty in *double complex* precision when iteration is permitted up to *15* steps and the algorithm considers $\mathrm{cberr}(A, y_i, b)$ for stability and $\|D_{|y|_i}^{-1} dy_i\|_\infty$ for convergence. The box labels on the right show the convergence status for the componentwise forward error. The labels on the top denote the error metric. A system appears in every box across a row but only once per column. The difficulty for each column is the product of the element growth $g_c = \max_j \frac{\left(\max_{1\le k\le j} \max_i |L|(i,k)\right)\cdot\left(\max_i |U|(i,j)\right)}{\max_i |A|(i,j)}$, the dimensional factor $\sqrt{n}$, and the relevant condition number. The condition number used for all the backward errors here is $\kappa_\infty(A^{-1}) = \kappa_\infty(A)$. The condition number for the normwise forward error is $\mathrm{colcond}(A, x) = \frac{\left\|A^{-1}\cdot(1_r \max |A|)\,|x|\right\|_\infty}{\|x\|_\infty}$, and the condition number for the componentwise forward error is $\mathrm{ccond}(A, x) = \left\|\,|(AD_{|x|})^{-1}| \cdot |AD_{|x|}|\,\right\|_\infty$. The horizontal blue lines denote the relevant precision ($\varepsilon_r = \sqrt{2}\cdot 2^{-106}$ for backward errors, $\varepsilon_w = \sqrt{2}\cdot 2^{-53}$ for forward errors). Horizontal red lines indicate bounds. The vertical blue line denotes $1/\varepsilon_w = 1/\varepsilon_f$, where ε_f is the precision used to factor A.

Chapter 5

Considerations for sparse systems

5.1 Introduction

One of the primary contributions in Chapter 4 is removing the condition estimators from Demmel et al. [37]'s iterative refinement algorithm. This is important for solving with sparse matrices, particularly on distributed memory machines. The second dominant cost after factorization is solution with triangular matrices[3]. Because of fill-in, numerical entries required to express L and U but are not in the sparse input matrix A, computing solutions naturally is more expensive than computing the residual. Common data structures used in distributed, unsymmetric factorization are not amenable to solving with a matrix's transpose. Condition estimation algorithms like Higham and Tisseur [62] rely on solving with A and A^T.

This chapter applies our refinement algorithm to sparse test systems. Section 5.2 lists the matrices chosen from Davis [29] and describes how the systems are generated. Section 5.3 applies the refinement algorithm including numerical scaling from Section 3.8 to sparse systems solved using unrestricted partial pivoting as well as threshold partial pivoting often used to reduce fill-in. Section 5.4 uses refinement to clean up after a much more aggressive pivoting scheme, static pivoting, that chooses pivots before factorization and perturbs the matrix should a pivot be too small. Static pivoting decouples the symbolic and numeric work in unsymmetric sparse LU factorization.

The end results:

- Iterative refinement remains dependable. All failures are caught.
- The growth factor (Equation (3.3.5)) plays an important role in a problem's difficulty.
- When threshold pivoting and static pivoting do not incur a large growth factor, iterative refinement produces the expected small errors.
- A column-relative static pivoting heuristic performs better than the default heuristic in SuperLU[71].

5.2 Generating sparse test systems

Table 5.1 lists the sparse matrices used for testing our refinement algorithm. These matrices were selected from existing publications comparing factorization accuracy [3, 71, 4] and were obtained from the UF Sparse Matrix Collection[29]. The sizes range from tiny (gre_115 has 115 rows) to moderately large (pre2 has over 650 000 rows). The majority are fully unsymmetric but there are eight symmetric and three structurally symmetric but numerically unsymmetric cases. A few matrices that differ only by scaling are included to test the numerical scaling.

Table 5.1: Sparse systems used for testing iterative refinement. **Bold** matrices are symmetric, ***Bold, italicized*** entries are *structurally* symmetric but not numerically symmetric. A ‡ marks the two matrices that have sample right-hand sides.

Group	Name	Dim	NENT	kind
Zitney	hydr1	5308	22680	chemical process simulation problem
Zitney	rdist1	4134	94408	chemical process simulation problem
Zitney	extr1	2837	10967	chemical process simulation problem
Zitney	rdist3a	2398	61896	chemical process simulation problem
Zitney	rdist2	3198	56834	chemical process simulation problem
Zitney	hydr1c	5308	22592	chemical process simulation problem sequence
Zitney	extr1b	2836	10965	chemical process simulation problem sequence
Zitney	radfr1	1048	13299	chemical process simulation problem
Hollinger	g7jac200	59310	717620	economic problem
Hollinger	mark3jac140sc	64089	376395	economic problem
Hollinger	g7jac200sc	59310	717620	economic problem
GHS_indef	**bmw3_2**	227362	11288630	structural problem
FIDAP	ex11	16614	1096948	computational fluid dynamics problem
FIDAP	ex19	12005	259577	computational fluid dynamics problem
Sanghavi	ecl32	51993	380415	semiconductor device problem
Wang	***wang4***	26068	177196	semiconductor device problem
GHS_psdef	**bmwcra_1**	148770	10641602	structural problem
GHS_psdef	**hood**	220542	9895422	structural problem
Norris	stomach	213360	3021648	2D/3D problem
Norris	torso1	116158	8516500	2D/3D problem

Continued on next page

Table 5.1: Sparse systems used for testing iterative refinement. **Bold** matrices are symmetric, ***Bold, italicized*** entries are *structurally* symmetric but not numerically symmetric. A ‡ marks the two matrices that have sample right-hand sides.

Group	Name	Dim	NENT	kind
HB	jpwh_991	991	6027	semiconductor device problem
HB	gemat11	4929	33108	power network problem sequence
HB	lnsp3937	3937	25407	computational fluid dynamics problem
HB	lns_131	131	536	computational fluid dynamics problem
HB	***orsreg_1***	2205	14133	computational fluid dynamics problem
HB	lnsp_131	131	536	computational fluid dynamics problem
HB	fs_541_2	541	4282	subsequent 2D/3D problem
HB	mcfe	765	24382	2D/3D problem
HB	lns_511	511	2796	computational fluid dynamics problem
HB	lns_3937	3937	25407	computational fluid dynamics problem
HB	psmigr_2	3140	540022	economic problem
HB	pores_2	1224	9613	computational fluid dynamics problem
HB	gre_115	115	421	directed weighted graph
HB	psmigr_3	3140	543160	economic problem
HB	mahindas‡	1258	7682	economic problem
HB	gre_1107	1107	5664	directed weighted graph
HB	orani678‡	2529	90158	economic problem
HB	psmigr_1	3140	543160	economic problem
HB	**saylr4**	3564	22316	computational fluid dynamics problem
HB	lnsp_511	511	2796	computational fluid dynamics problem
HB	west2021	2021	7310	chemical process simulation problem
Garon	garon2	13535	373235	computational fluid dynamics problem
Goodwin	goodwin	7320	324772	computational fluid dynamics problem
Vavasis	av41092	41092	1683902	2D/3D problem
Bai	tols4000	4000	8784	computational fluid dynamics problem
Bai	olm5000	5000	19996	computational fluid dynamics problem
Bai	dw8192	8192	41746	electromagnetics problem

Continued on next page

Table 5.1: Sparse systems used for testing iterative refinement. **Bold** matrices are symmetric, ***Bold, italicized*** entries are *structurally* symmetric but not numerically symmetric. A ‡ marks the two matrices that have sample right-hand sides.

Group	Name	Dim	NENT	kind
Bai	af23560	23560	460598	computational fluid dynamics problem
DNVS	**ship_003**	121728	3777036	structural problem
Graham	graham1	9035	335472	computational fluid dynamics problem
ATandT	onetone1	36057	335552	frequency-domain circuit simulation problem
ATandT	pre2	659033	5834044	frequency-domain circuit simulation problem
ATandT	twotone	120750	1206265	frequency-domain circuit simulation problem
ATandT	onetone2	36057	222596	frequency-domain circuit simulation problem
Grund	bayer01	57735	275094	chemical process simulation problem
Mallya	lhr71c	70304	1528092	chemical process simulation problem
Shyy	shyy161	76480	329762	computational fluid dynamics problem

Given a matrix A from Table 5.1, we generate four random right-hand sides b as in Section 4.1.1. Two of the right-hand sides are generated randomly, and two are generated by multiplying a random vector by A. Two test systems, orani678 and mahindas, come with problem-specific sparse right-hand sides. We randomly select two of these right-hand sides for each of the two systems and include them in our results.

True solutions are generated with partial pivoting in Toledo and Uchitel [98]'s out-of-core TAUCS code modified to use the author's dn-arith package.[1] The dn-arith package uses doubled arithmetic on top of the `long double` type in Standard C[2]. On Intel and AMD hardware and typical platform settings[2], the `long double` type is an 80-bit-wide format implementing an IEEE754[64] double-extended format. Doubled `long double` arithmetic has at 15 bits of range and least 128 bits of precision, giving better than quadruple precision accuracy up to the issues of doubled precision discussed earlier.

All matrix factorizations are ordered by Davis et al. [30]'s column approximate minimum degree ordering. This is not always the best ordering for sparsity, but it suffices to demonstrate that refinement remains dependable. The systems also are scaled numerically as in LAPACK's x`GEQUB` before computing the true solution. Detailed in listing 5.1, the scaling first divides

[1]Available from the author, currently at `http://lovesgoodfood.com/jason/cgit/index.cgi/dn-arith/` .

[2]Notably not Windows, which restricts the use of hardware precision.

```
function [Sr, As, Sc] = xGEEQUB (A)
### Function File A = xGEEQUB (A)
### Scale an input matrix A by powers of two along the rows
### and columns. The rows first are divided by the largest
### power of two no larger than the largest entry's magnitude.
### The columns then are scaled similarly. The output
### satisfies Sr * A * Sc = As.
  Sr = diag (2.^-floor (log2 (full (max (abs(A))))));
  As = Sr * A;
  Sc = diag (2.^-floor (log2 (full (max (abs(A), [], 2)))));
  As = As * Sc;
endfunction
```

Listing 5.1: Numerical scaling as in LAPACK's x`GEQUB`.

each row by its largest magnitude entry rounded down to a power of two, and then divides each column by its largest magnitude entry also rounded down to a power of two. The largest magnitude value in each row and column is between $1/2$ and 2.

The sparse test systems are spread across the difficulty range with respect to both normwise (Figure 5.1) and componentwise (Figure 5.2) forward error results.

5.3 Refinement after partial and threshold partial pivoting

Partial pivoting selects the largest magnitude element remaining in the factored column as the pivot. Threshold partial pivoting[72] loosens the requirement and selects a sparsity-preserving pivot (least Markowitz cost[73]) with magnitude within a threshold factor of the largest magnitude. Partial pivoting can be considered threshold partial pivoting with a threshold of one. Partial pivoting almost always produces factors with moderate element growth, while threshold partial pivoting risks inducing larger growth.

Using UMFPACK[28] via GNU Octave[51] followed by extra-precise iterative refinement, we achieve dependable solutions and accurate solutions for all systems that are not too ill-conditioned and which do not suffer extreme element growth.[3]

Including partial pivoting, we experiment with four thresholds: 1.0, 0.9, 0.5, and 0.1.

5.4 Cleaning up after static pivoting

Static pivoting is a very aggressive form of pivoting used to decouple highly parallel numerical factorization from the symbolic work[71]. Static pivoting with perturbations has been used for maintaining symmetry in indefinite sparse matrix factorizations[95, 53], proposed as a

[3]UMFPACK supports threshold partial pivoting and is the default solver in GNU Octave.

Figure 5.1: This histogram of $\kappa(A, x, b)$ summarizes the normwise sensitivity of the 232 sparse test systems to perturbations. We expect good solutions for systems to the left of the vertical blue line, $\kappa(A, x, b) \geq 1/\varepsilon_w = 2^{53}$, so long as the factorization does not incur large element growth. There are 12 extreme systems with condition numbers beyond the plot to the right.

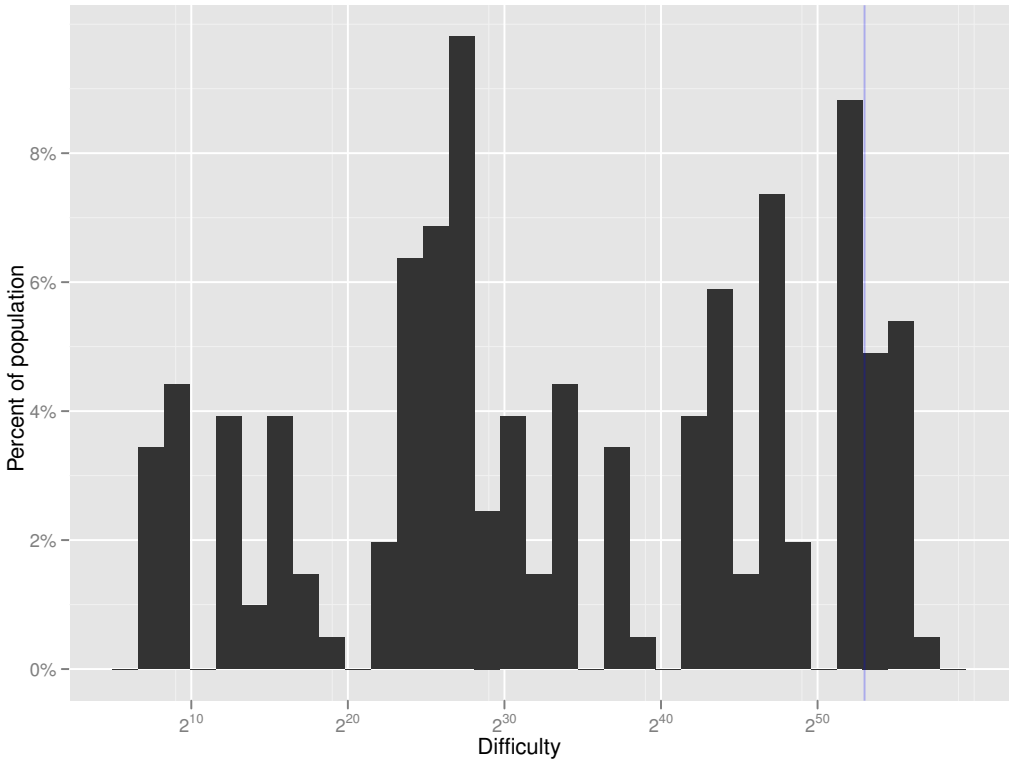
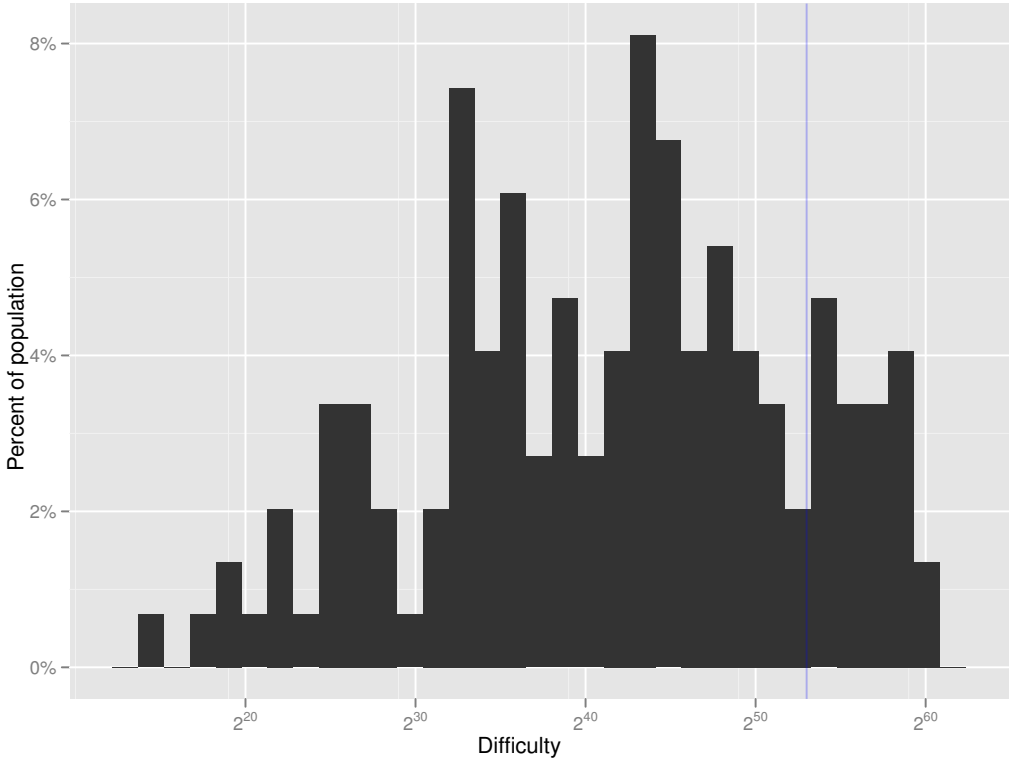


Figure 5.2: This histogram of $\mathrm{ccond}(A, x, b)$ summarizes the componentwise sensitivity of the 232 sparse test systems to perturbations. We expect good solutions for systems to the left of the vertical blue line, $\mathrm{ccond}(A, x, b) \geq 1/\varepsilon_w = 2^{53}$, so long as the factorization does not incur large element growth. There are 16 extreme systems with condition numbers beyond the plot to the right.



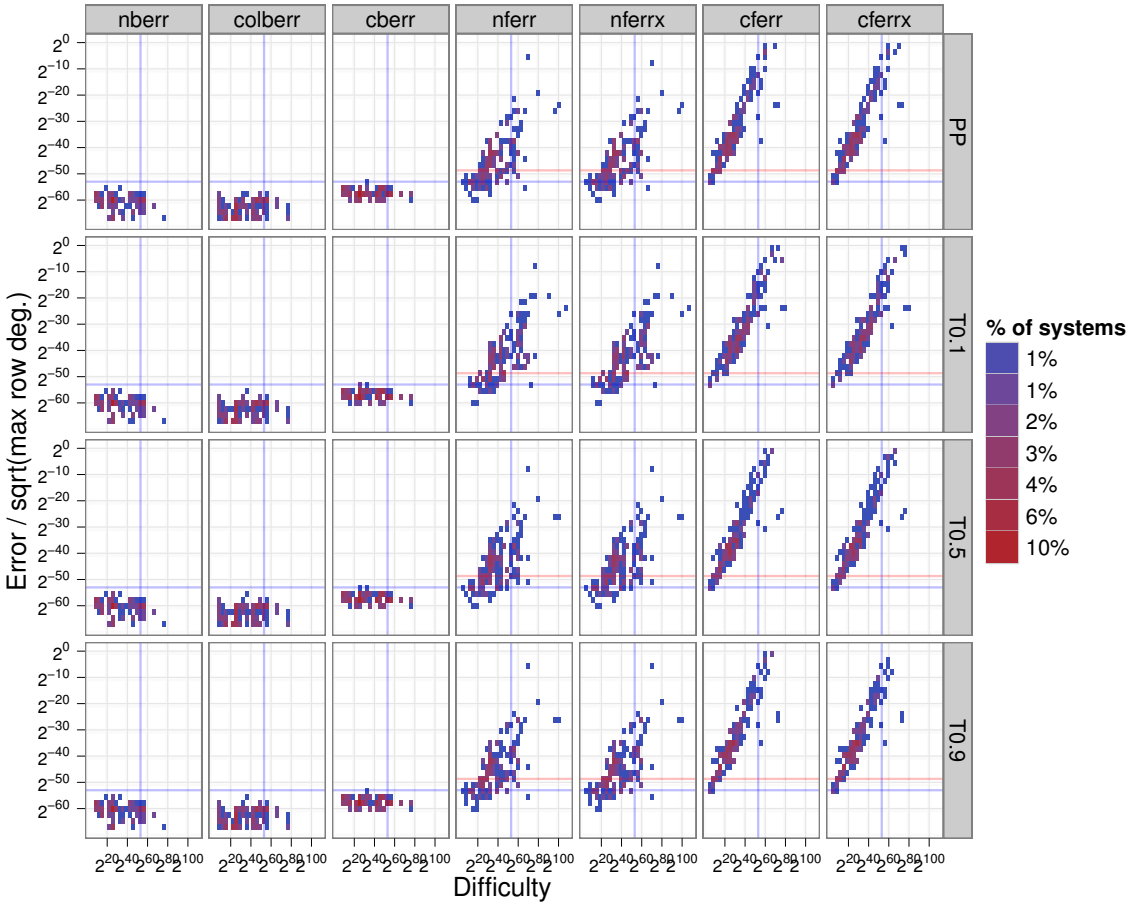


Figure 5.3: Initial errors by pivoting threshold after solving by sparse LU factorization. A threshold of one means unrestricted partial pivoting. A threshold of 0.5 means factorization chooses a pivot with least row degree having magnitude at least 0.5 of the column's largest magnitude.

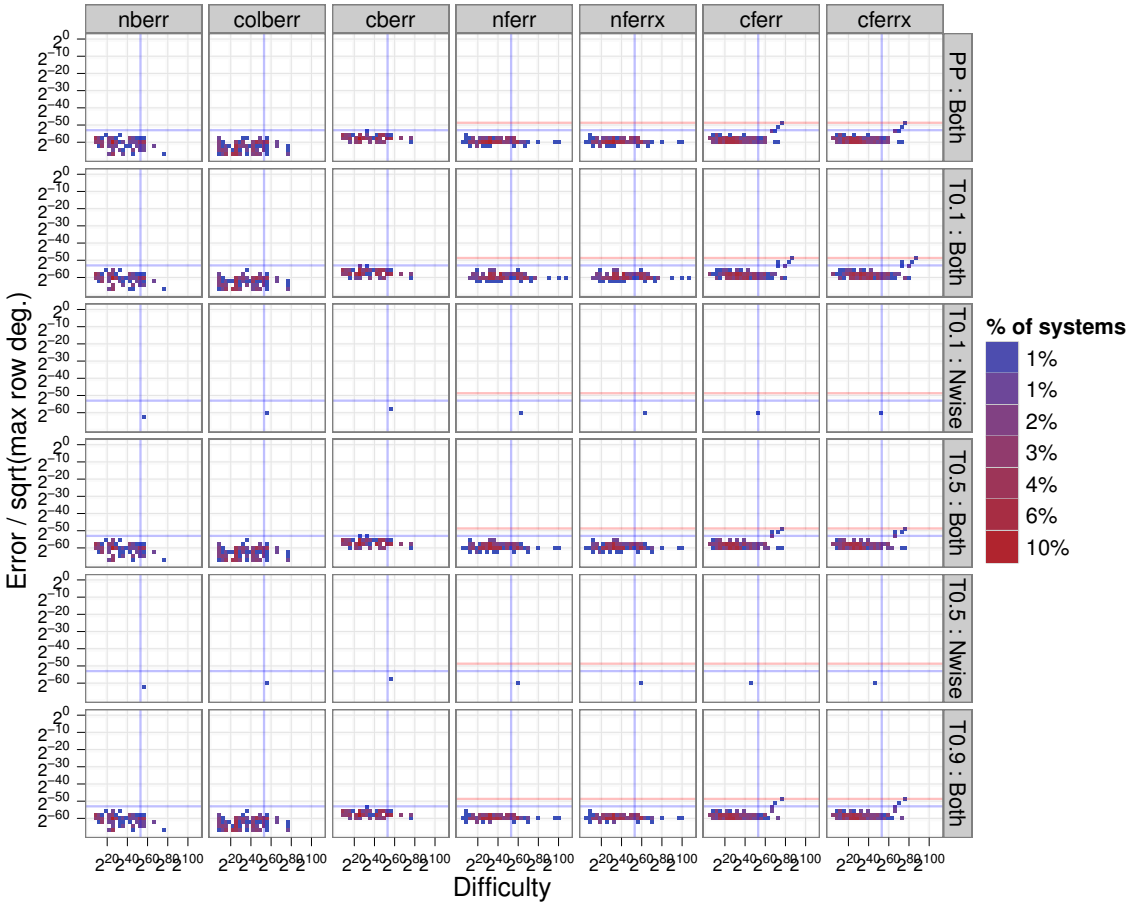


Figure 5.4: Errors in accepted solutions after refinement by pivoting threshold after solving by sparse LU factorization. A solution can be accepted normwise when the componentwise result is not trusted or accepted for both when both are trusted. Note that all solutions are accepted at least normwise. A threshold of one means unrestricted partial pivoting. A threshold of 0.5 means factorization chooses a pivot with least row degree having magnitude at least 0.5 of the column's largest magnitude.

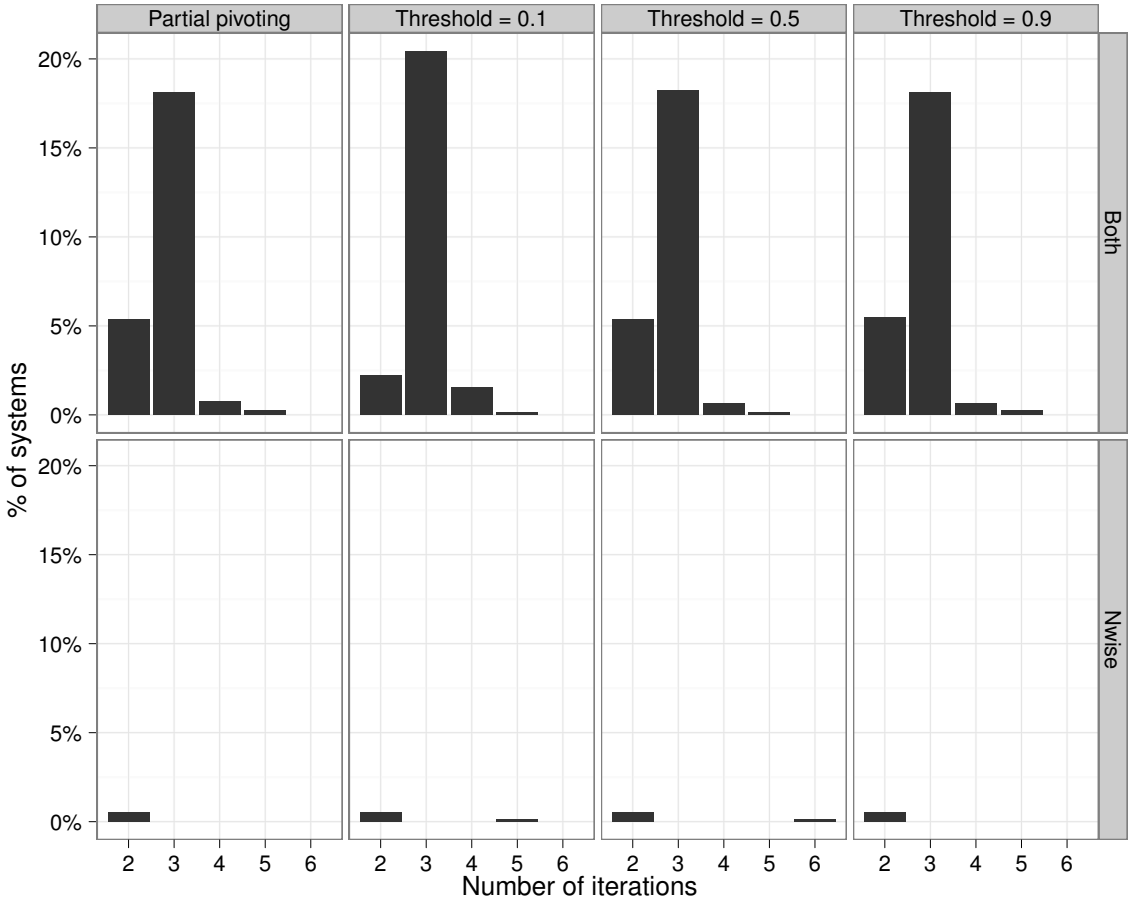


Figure 5.5: Iterations required by refinement by pivoting threshold and acceptance level after solving by sparse LU factorization. A solution can be accepted normwise when the componentwise result is not trusted or accepted for both when both are trusted. Note that all solutions are accepted at least normwise. A threshold of one means unrestricted partial pivoting. A threshold of 0.5 means factorization chooses a pivot with least row degree having magnitude at least 0.5 of the column's largest magnitude.

Figure 5.6: Iterations required by refinement by threshold

general pivoting scheme[82], and even included as a textbook exercise[96]. The symbolic ordering and factorization is more difficult to implement in parallel, but it also requires less memory and often runs more quickly than the numerical factorization[3, 71]. At the time of writing, machines with over 40 GiB of main memory can be rented on-demand from Amazon[65] for $1.50 an hour, and that amount of memory is sufficient for the symbolic factorization of every matrix in the UF Collection with sufficient care.

Static pivoting selects large elements to use as pivots ahead of time, before any numerical factorization occurs. Part II investigates one method for choosing pivots that we take as a given here. If some pivot is found to be too small and hence likely to cause intolerable element growth, the pivot is perturbed by adding a quantity small relative to the expected backward error. Ideally, preserving sparsity also limits the number of arithmetic operations affecting each matrix element, so the large magnitude diagonal entries should remain large.

The perturbation strategy we use increases a tiny pivots' magnitude to the threshold of tininess while maintaining its sign. We use three different definitions of tininess for a matrix A and precision parameter γ:

- SuperLU: $\|A\|_1 \cdot \gamma$,
- column-relative: $\max_i |A(i,j)| \cdot \gamma$,
- diagonal-relative: $|A(j,j)| \cdot \gamma$.

The SuperLU heuristic perturbs by an amount relative to the expected backward error. Li and Demmel [71] take $\gamma = \sqrt{\varepsilon_f} = \sqrt{2^{-53}}$, resulting in a half-precision factorization in the worst case (see Section 3.6.5 for implications). The author's experiments with $\gamma = 2^{-10}\sqrt{\varepsilon_f}$ [86] performed better primarily by reducing the number of perturbations.

To validate dependable iterative refinement, we use three different settings for γ, $2^{-26} \approx \sqrt{\varepsilon_f}$, 2^{-38}, and $2^{-43} = 2^{10}\varepsilon_f$. Each test case is scaled numerically according to Listing 5.1. We factor using a version of TAUCS modified to support static pivoting. Figures 5.7, 5.14, and 5.21 show the initial errors after factoring with the SuperLU, column-relative, and diagonal-relative heuristics respectively. The column- and diagonal-relative heuristics perform fewer and smaller perturbations, and that appears in better initial results. Table 5.2 shows that the diagonal- and column-relative perturbation heuristics provide more successful solutions than SuperLU's default heuristic. Figures 5.8, 5.9, and 5.10 show the results *after* refining for the componentwise forward error using the SuperLU perturbation heuristic for different levels of γ. Figures 5.15, 5.16, and 5.17 show the results *after* refining for the componentwise forward error using the column-relative perturbation heuristic for different levels of γ. Figures 5.22, 5.23, and 5.24 show the results *after* refining for the componentwise forward error using the diagonal-relative perturbation heuristic for different levels of γ. All accepted results are below the error threshold. The only "well-enough" conditioned matrices that fail suffer large element growth from the static pivot selection except for the cases with sparse right-hand sides. These cases give up on the componentwise backward error nearly

Level and heuristic	Result		
	Trust both	Trust nwise	Reject
$2^{-43} = 2^{10} \cdot \varepsilon_f$			
SuperLU	42.9%	8.0%	49.0%
Column-relative	55.7%	5.7%	38.6%
Diagonal-relative	55.8%	5.9%	38.3%
$2^{-38} =\approx 2^{-12} \cdot \sqrt{\varepsilon_f}$			
SuperLU	36.6%	6.7%	56.6%
Column-relative	52.4%	6.5%	41.2%
Diagonal-relative	53.7%	7.2%	39.1%
$2^{-26} \approx \sqrt{\varepsilon_f}$			
SuperLU	32.4%	4.0%	63.6%
Column-relative	42.2%	4.2%	53.6%
Diagonal-relative	47.4%	4.7%	47.9%

Table 5.2: Success rates regardless of difficulty for each perturbation level and heuristic show that both the column- and diagonal-relative heuristics perform better than the SuperLU heuristic. Smaller perturbations perform better than larger, although the payoff appears to level off shortly below $\sqrt{\varepsilon_f}$. Plots show that almost all of the rejections are appropriate; rejected systems are "too difficult" relative to the factorization precision ε_f and the perturbation size. Smaller perturbations accept a wider range of difficulties.

immediately. Using the column-relative backward error should fix these cases. Figures 5.11, 5.12, and 5.13 show the sometimes large iteration counts using the SuperLU heuristic with different γ settings. Figures 5.18, 5.19, and 5.20 show the sometimes large iteration counts using the column-relative heuristic with different γ settings. Figures 5.25, 5.26, and 5.27 show the sometimes large iteration counts using the diagonal-relative heuristic with different γ settings. These are iteration counts for the componentwise forward error. Users wishing only a normwise backward error near ε_w need wait only two to four iterations.

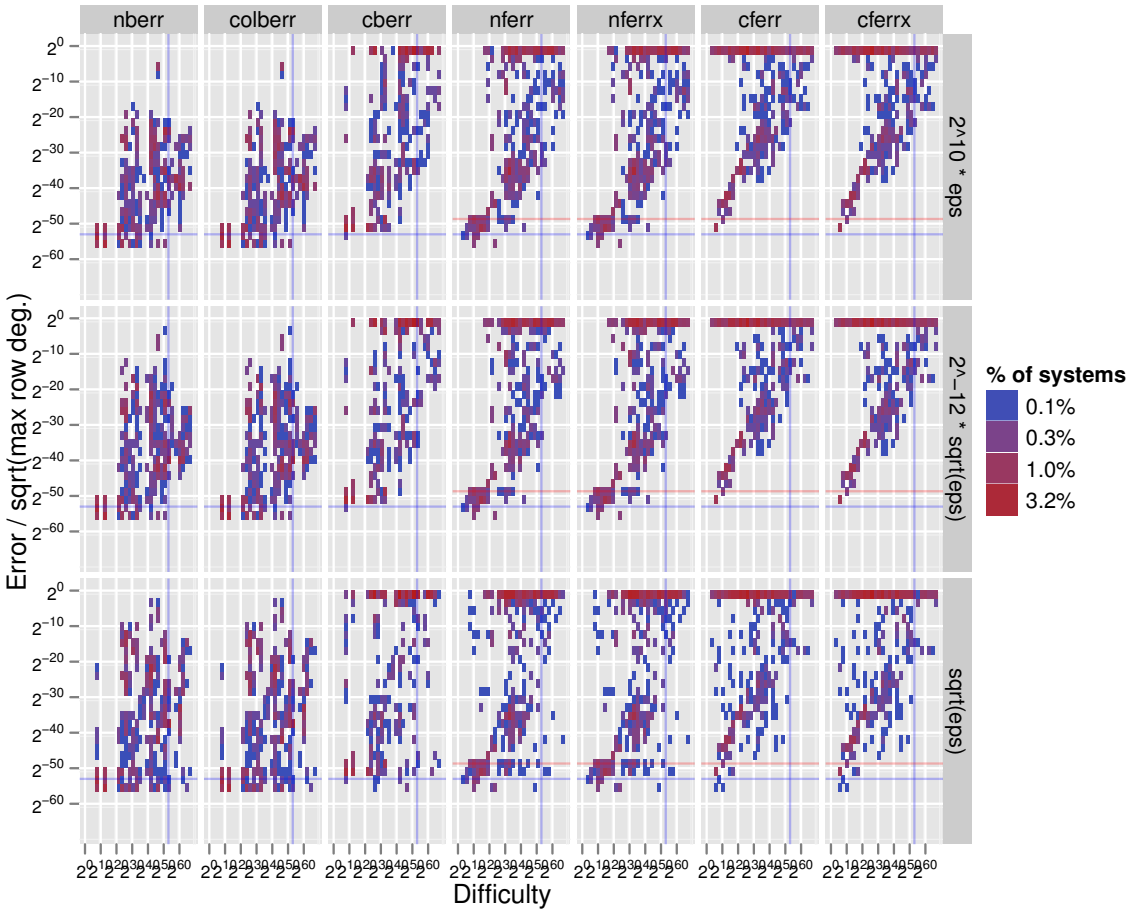


Figure 5.7: Initial errors by γ for the SuperLU heuristic. Here γ denotes a multiple of $\|A\|_1$ below which pivots are perturbed until their magnitude reaches $\gamma\|A\|_1$, and eps denotes the double-precision factorization (eps $= 2^{-53}$). The label sqrt(eps) denotes $2^{-}26 \approx \sqrt{2^{-53}}$.

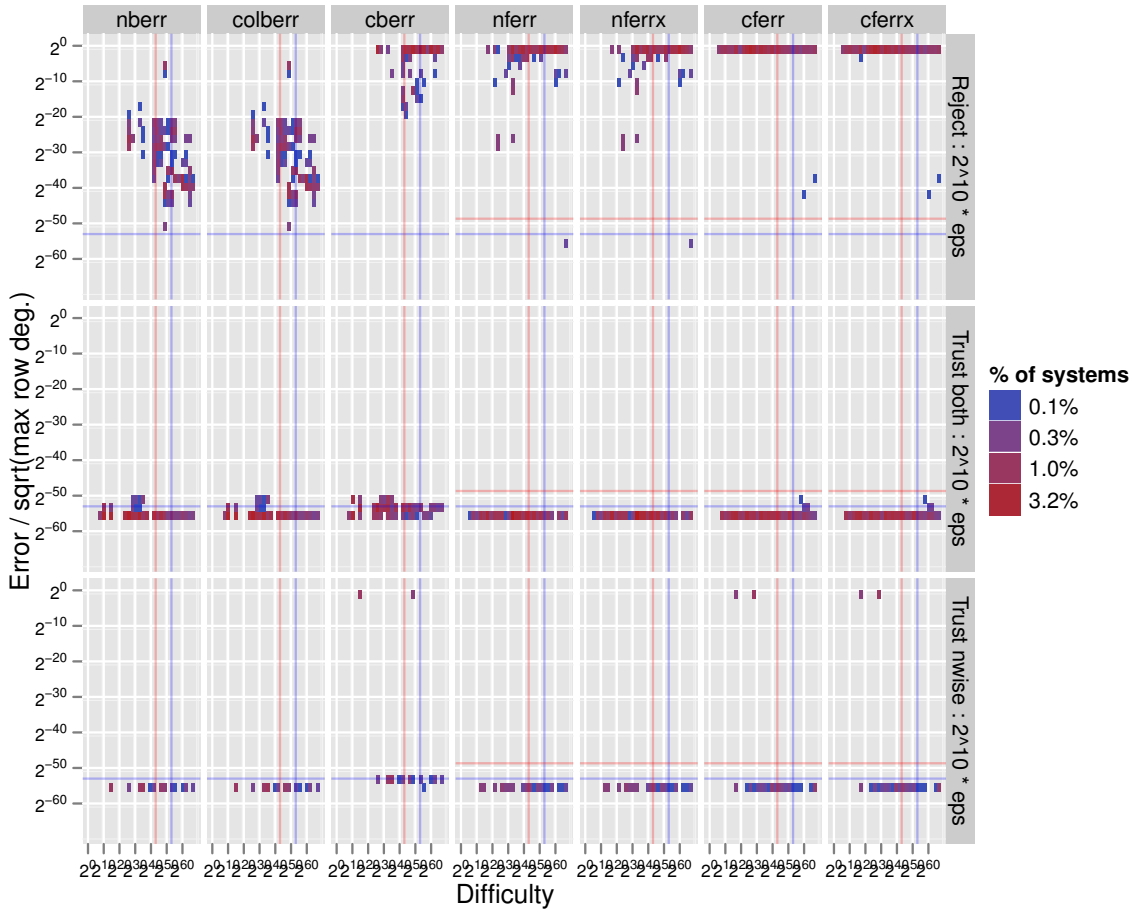


Figure 5.8: Errors after refinement for perturbations of size up to $\gamma = 2^{10}\varepsilon_f = 2^{-43}$ for the SuperLU heuristic show that refinement renders static pivoting dependable. The errors are scaled by the square-root of the maximum row degree in $L + U$ to normalize by the size c_N. The vertical blue line is $1/\varepsilon_f = 2^{53}$, and the vertical red line is $1/\gamma$.

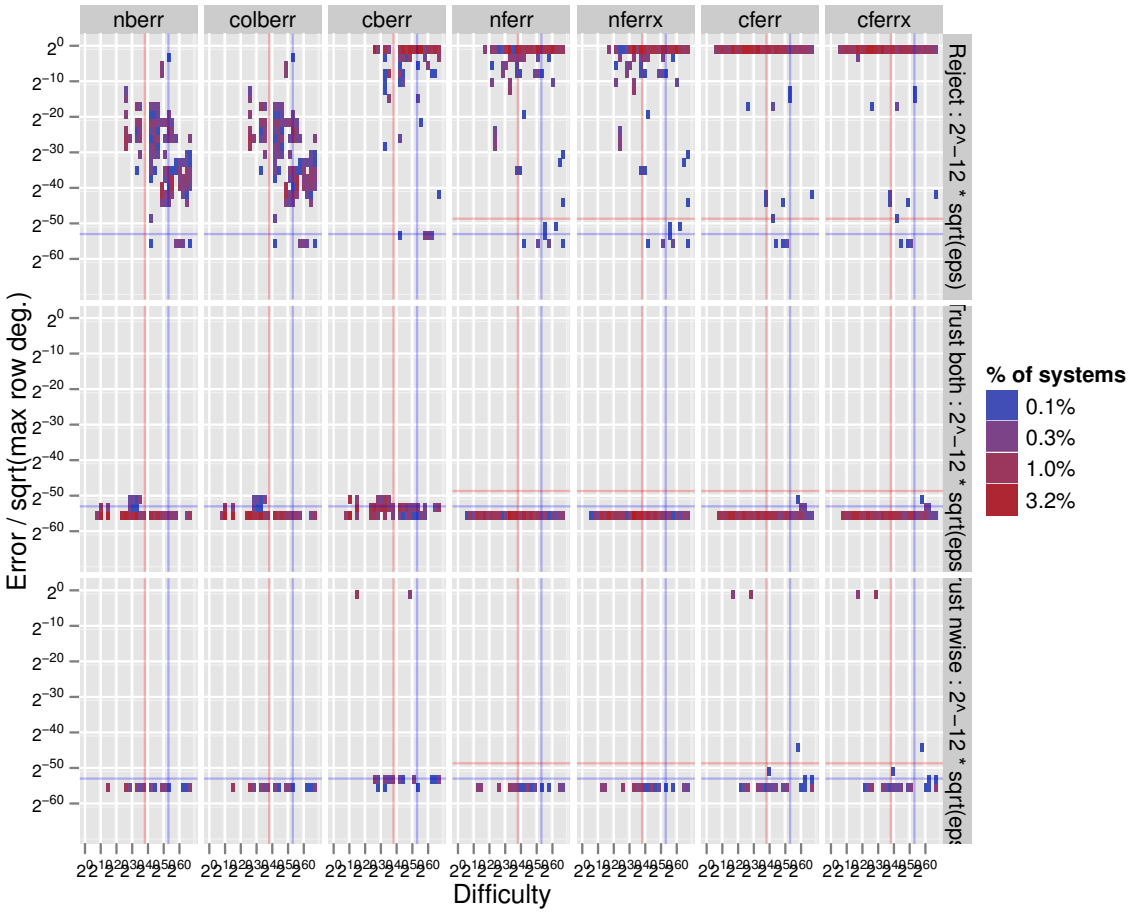


Figure 5.9: Errors after refinement for perturbations of size up to $\gamma = 2^{-38}$ for the SuperLU heuristic show that refinement renders static pivoting dependable. The errors are scaled by the square-root of the maximum row degree in $L + U$ to normalize by the size c_N. The vertical blue line is $1/\varepsilon_f = 2^{53}$, and the vertical red line is $1/\gamma$.

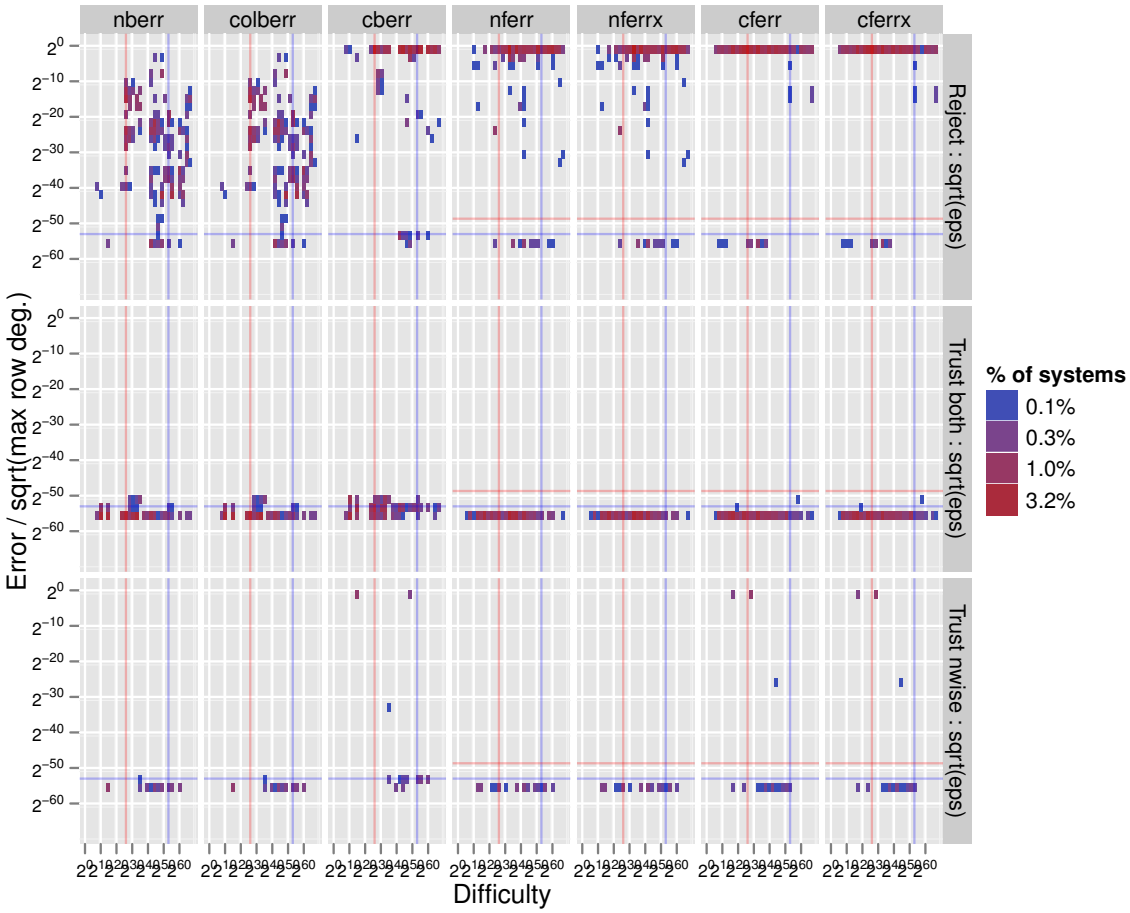


Figure 5.10: Errors after refinement for perturbations of size up to $\gamma = 2^{-26} \approx \sqrt{\varepsilon_f}$ for the SuperLU heuristic show that refinement renders static pivoting dependable. The errors are scaled by the square-root of the maximum row degree in $L + U$ to normalize by the size c_N. The vertical blue line is $1/\varepsilon_f = 2^{53}$, and the vertical red line is $1/\gamma$.

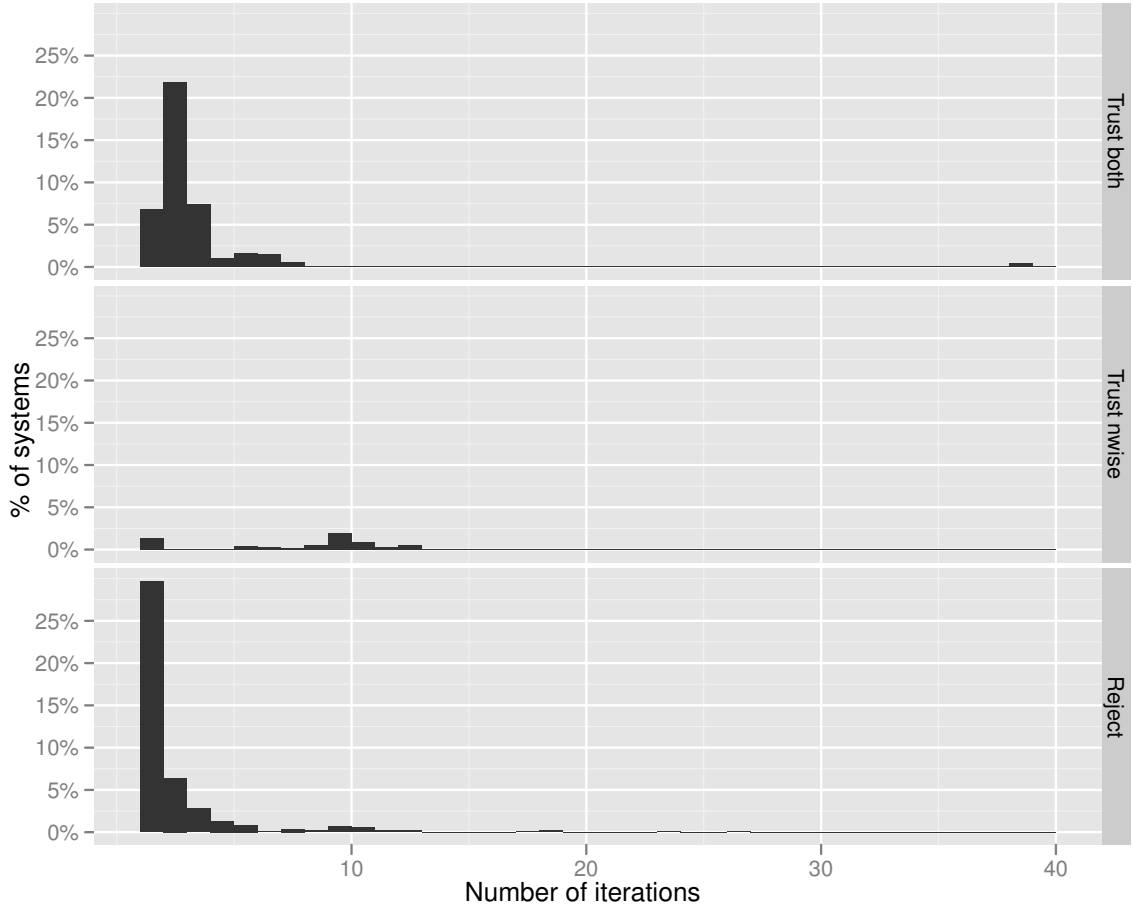


Figure 5.11: Iterations required by refinement for $\gamma = 2^{10}\varepsilon_f = 2^{-43}$ for the SuperLU heuristic. The right-hand label is the final status of the solution.

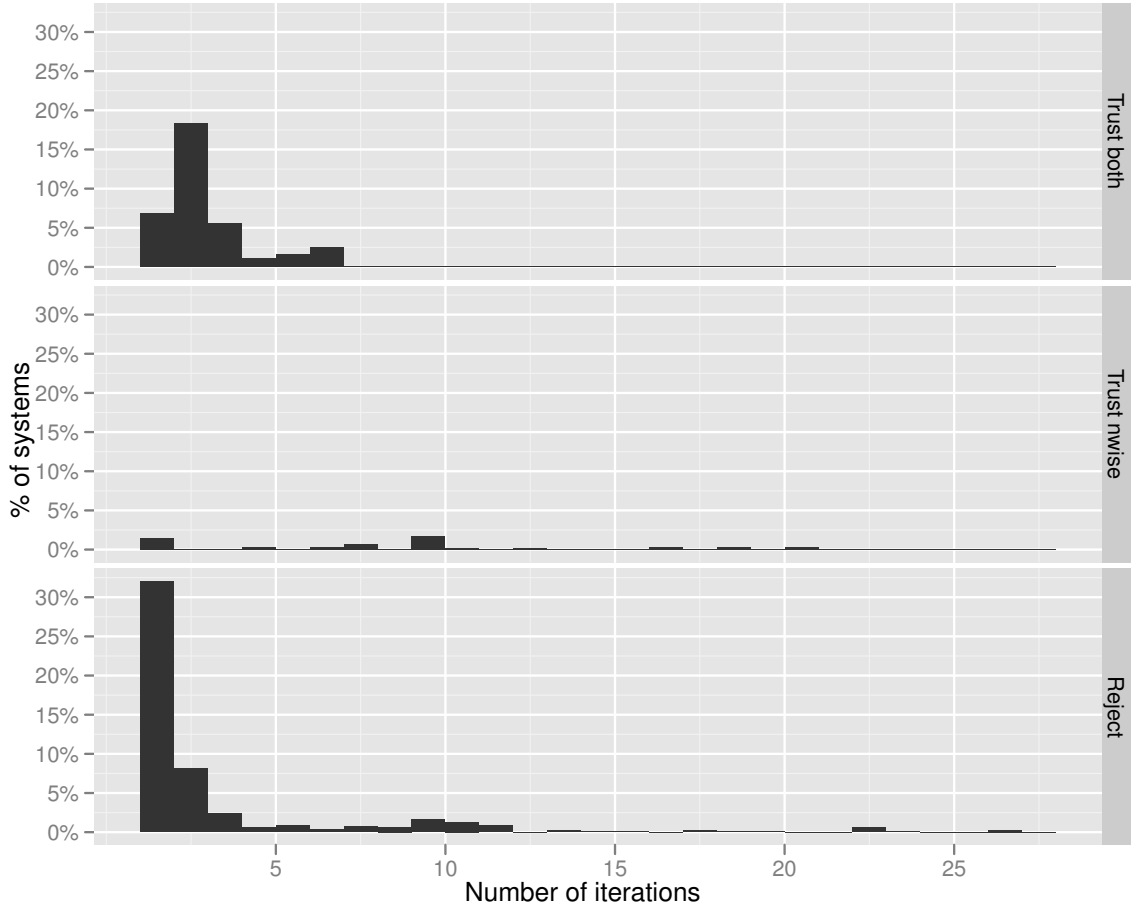


Figure 5.12: Iterations required by refinement by $\gamma = 2^{-38}$ for the SuperLU heuristic. The right-hand label is the final status of the solution.

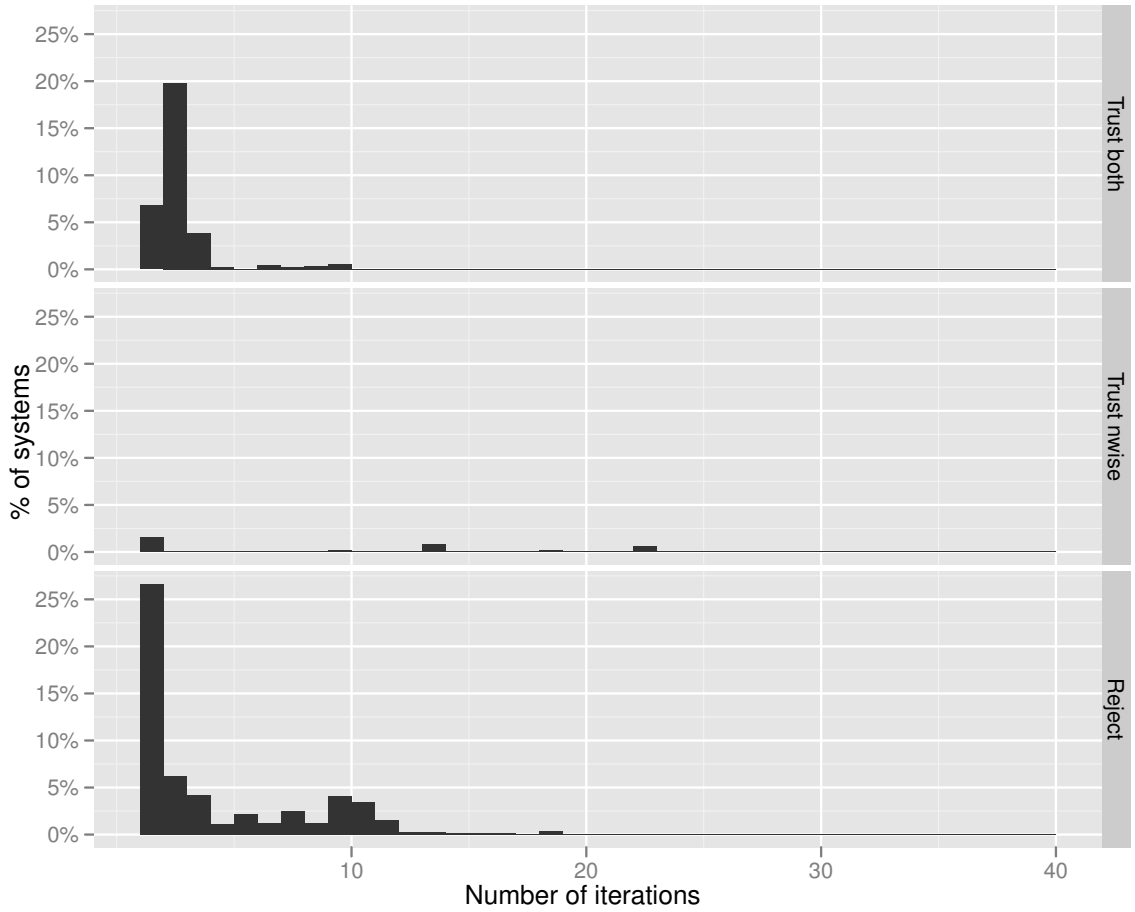


Figure 5.13: Iterations required by refinement by $\gamma = 2^{-26} \approx \sqrt{\varepsilon_f}$ for the SuperLU heuristic. The right-hand label is the final status of the solution.

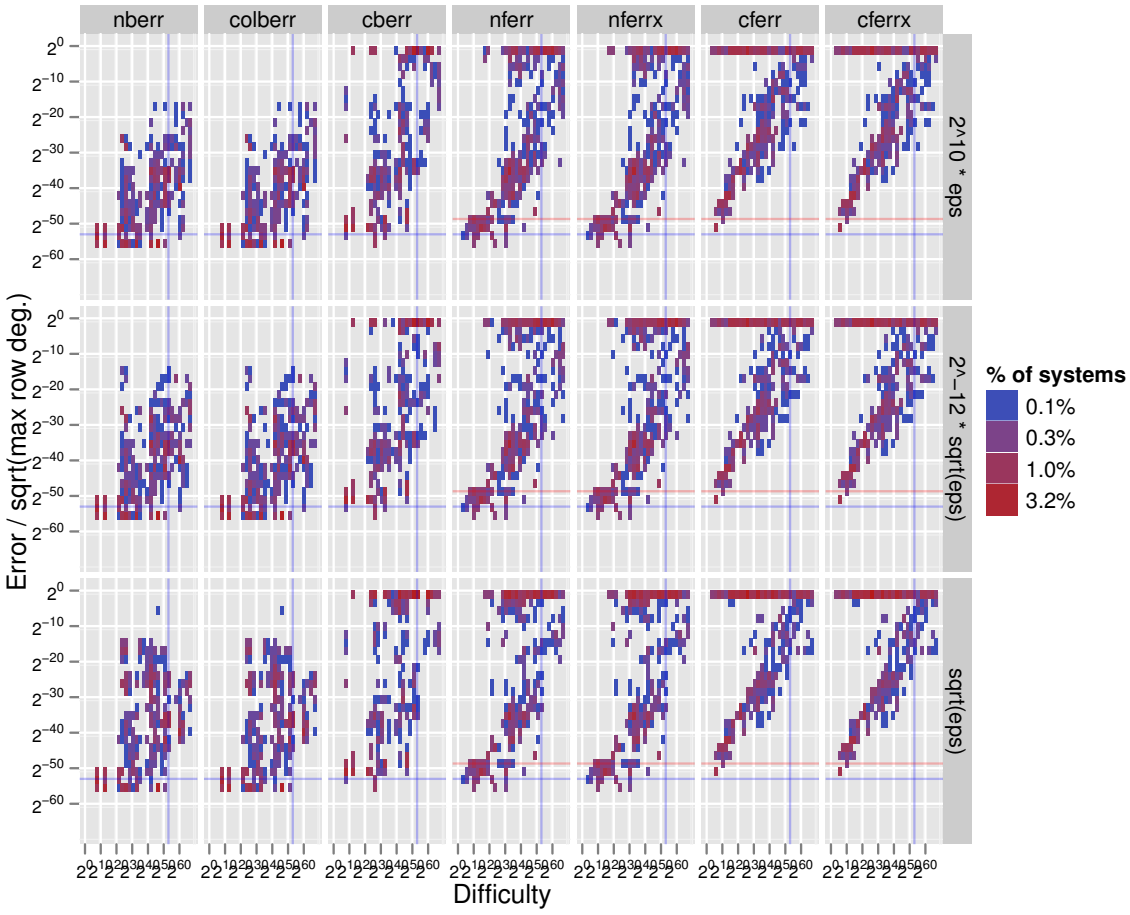


Figure 5.14: Initial errors by γ for the column-relative heuristic. Here γ denotes a multiple of $\|A\|_1$, and eps denotes the double-precision factorization (eps $= 2^{-53}$). The label sqrt(eps) denotes $2^{-}26 \approx \sqrt{2^{-53}}$.

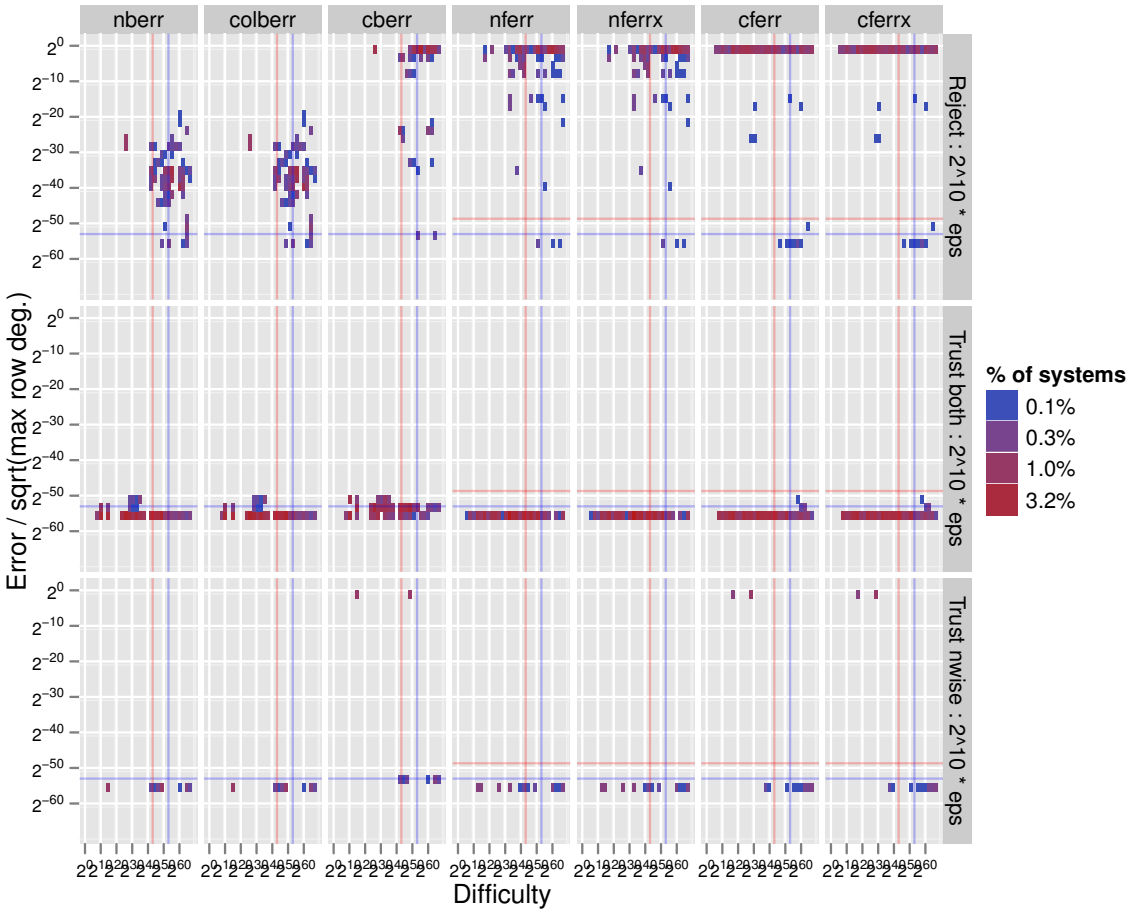


Figure 5.15: Errors after refinement for perturbations of size up to $\gamma = 2^{10}\varepsilon_f = 2^{-43}$ for the column-relative heuristic show that refinement renders static pivoting dependable. The errors are scaled by the square-root of the maximum row degree in $L + U$ to normalize by the size c_N. The vertical blue line is $1/\varepsilon_f = 2^{53}$, and the vertical red line is $1/\gamma$.

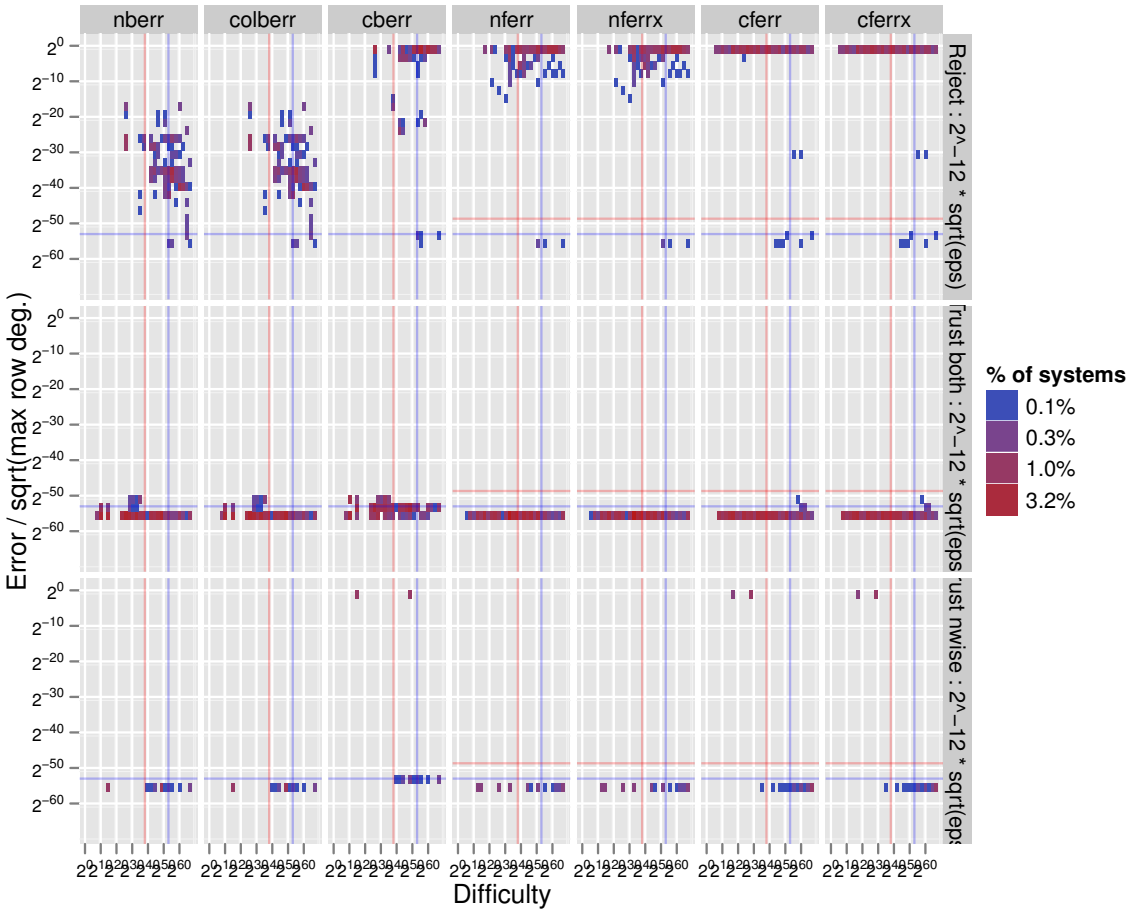


Figure 5.16: Errors after refinement for perturbations of size up to $\gamma = 2^{-38}$ for the column-relative heuristic show that refinement renders static pivoting dependable. The errors are scaled by the square-root of the maximum row degree in $L + U$ to normalize by the size c_N. The vertical blue line is $1/\varepsilon_f = 2^{53}$, and the vertical red line is $1/\gamma$.

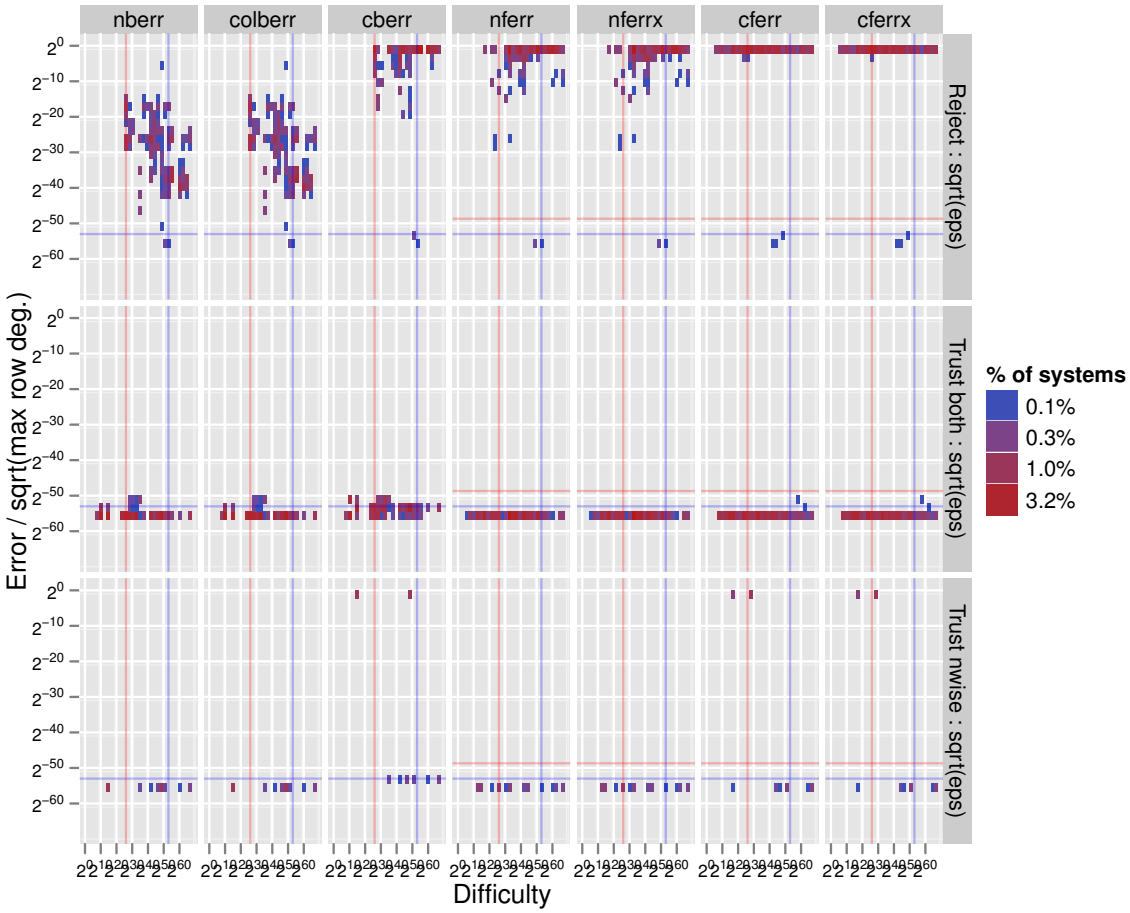


Figure 5.17: Errors after refinement for perturbations of size up to $\gamma = 2^{-26} \approx \sqrt{\varepsilon_f}$ for the column-relative heuristic show that refinement renders static pivoting dependable. The errors are scaled by the square-root of the maximum row degree in $L + U$ to normalize by the size c_N. The vertical blue line is $1/\varepsilon_f = 2^{53}$, and the vertical red line is $1/\gamma$.

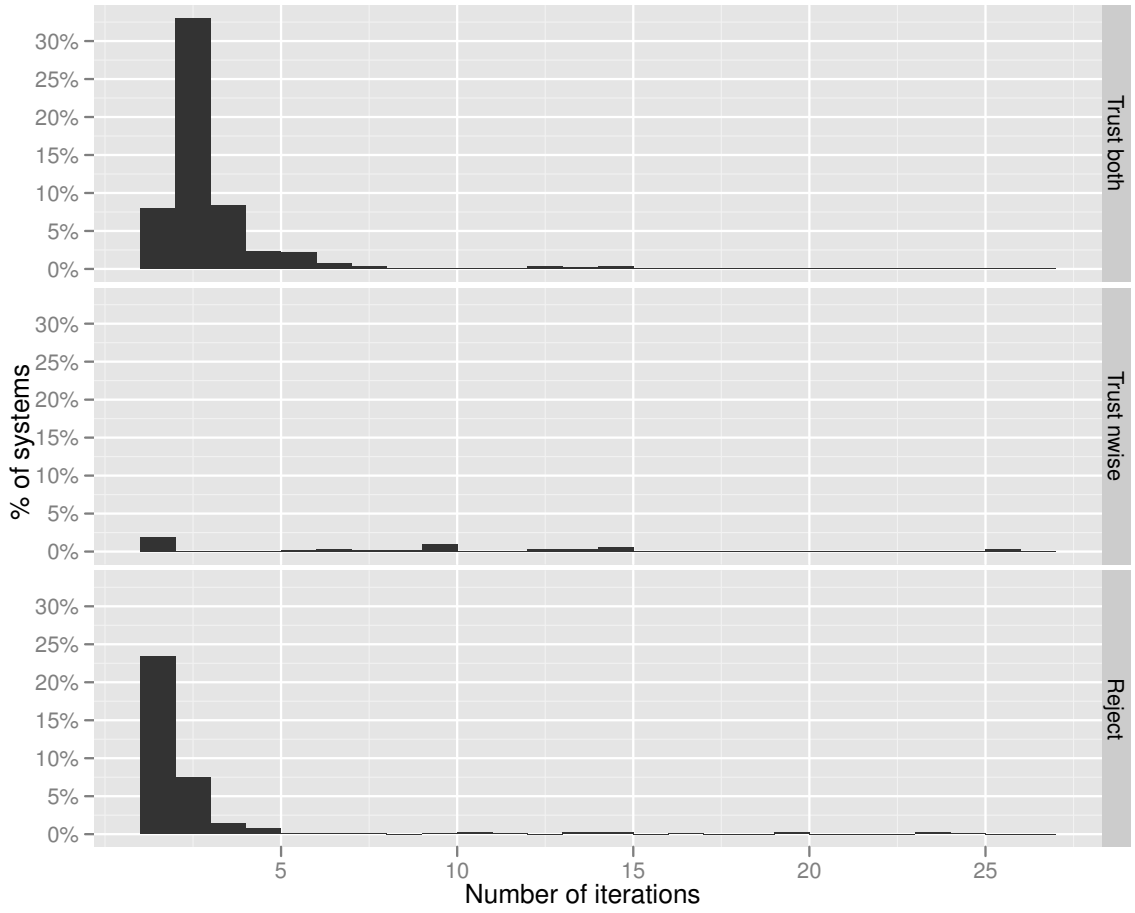


Figure 5.18: Iterations required by refinement for $\gamma = 2^{10}\varepsilon_f = 2^{-43}$ for the column-relative heuristic. The right-hand label is the final status of the solution.

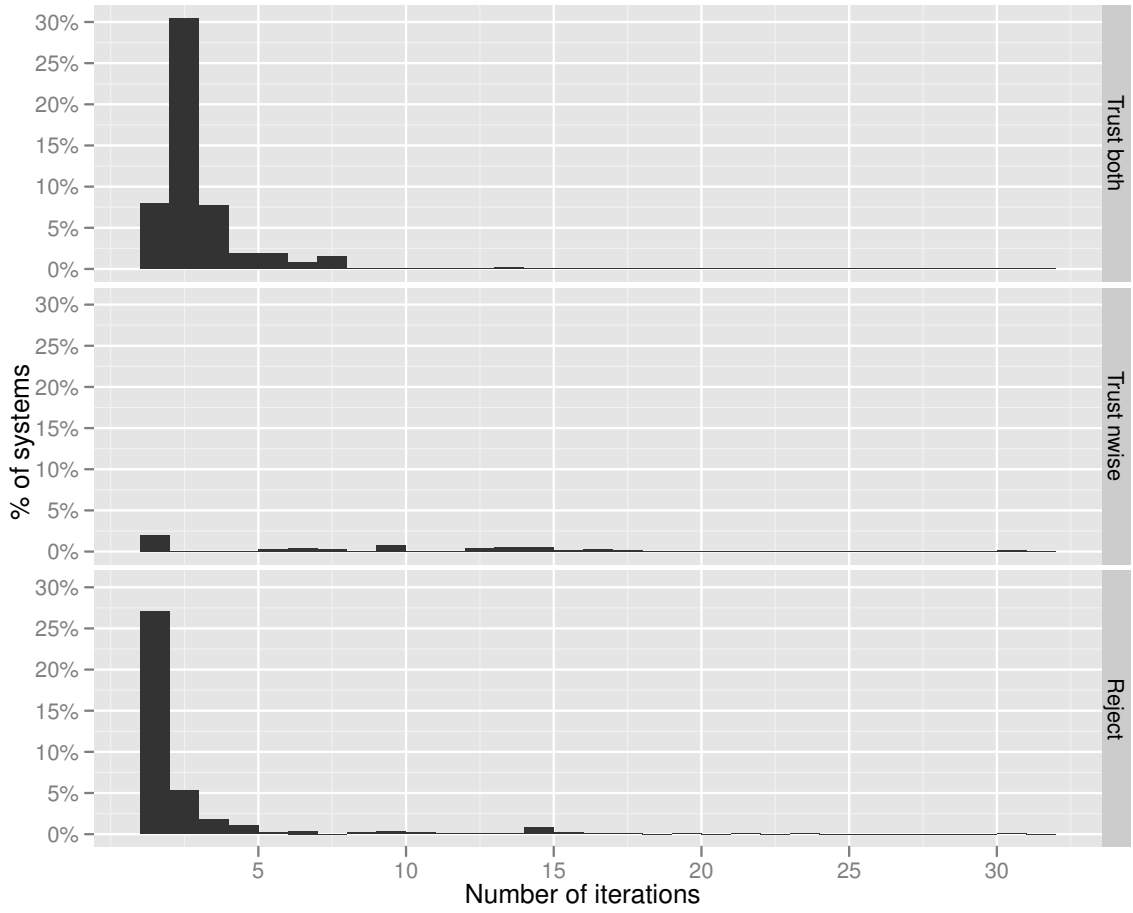


Figure 5.19: Iterations required by refinement by $\gamma = 2^{-38}$ for the column-relative heuristic. The right-hand label is the final status of the solution.

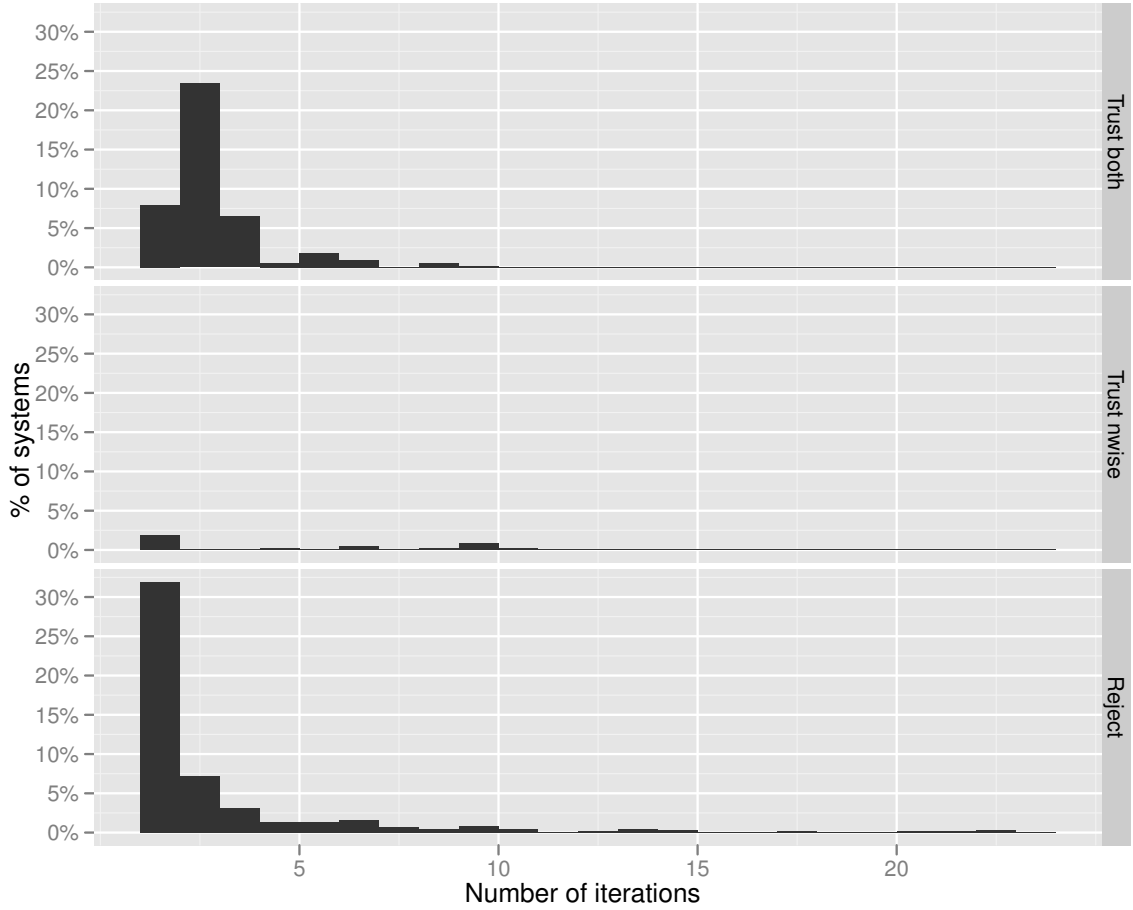


Figure 5.20: Iterations required by refinement by $\gamma = 2^{-26} \approx \sqrt{\varepsilon_f}$ for the column-relative heuristic. The right-hand label is the final status of the solution.

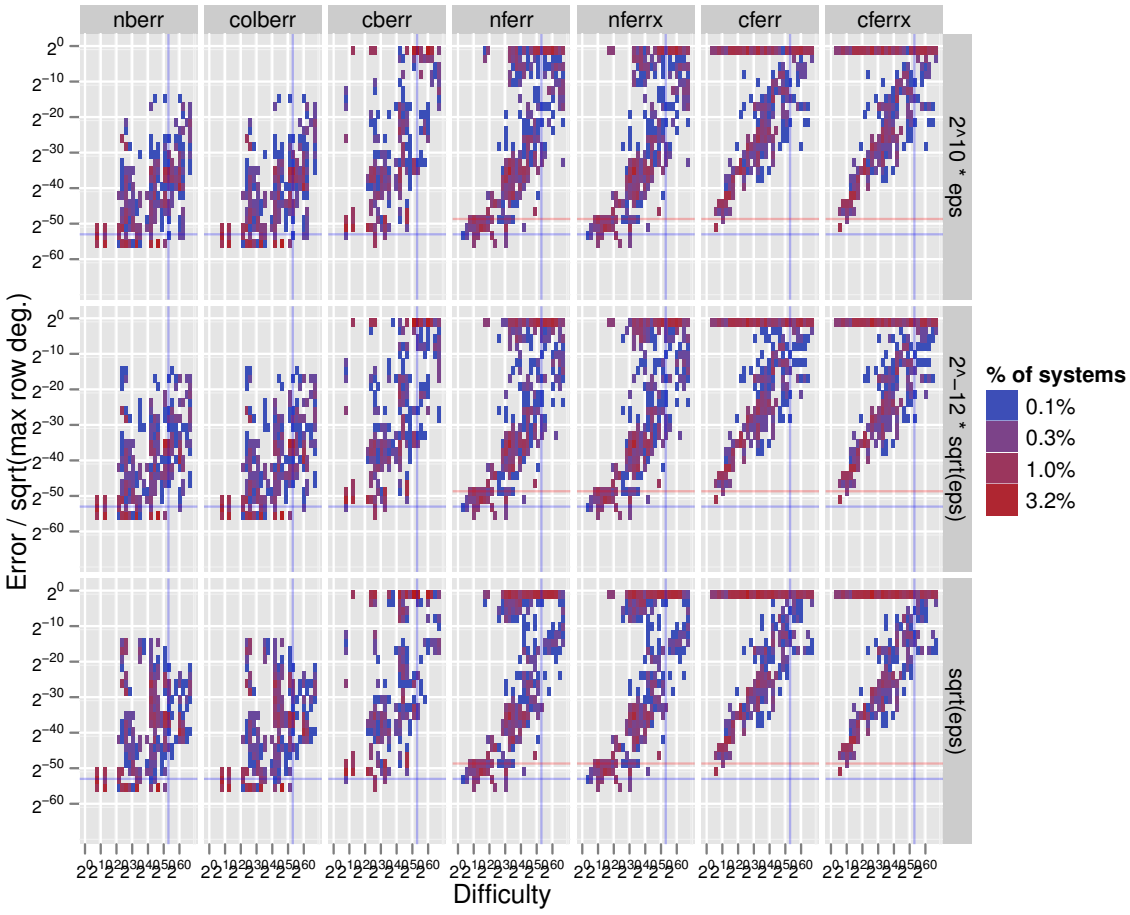


Figure 5.21: Initial errors by γ for the diagonal-relative heuristic. Here γ denotes a multiple of $\|A\|_1$, and eps denotes the double-precision factorization (eps $= 2^{-53}$). The label sqrt(eps) denotes $2^{-}26 \approx \sqrt{2^{-53}}$.

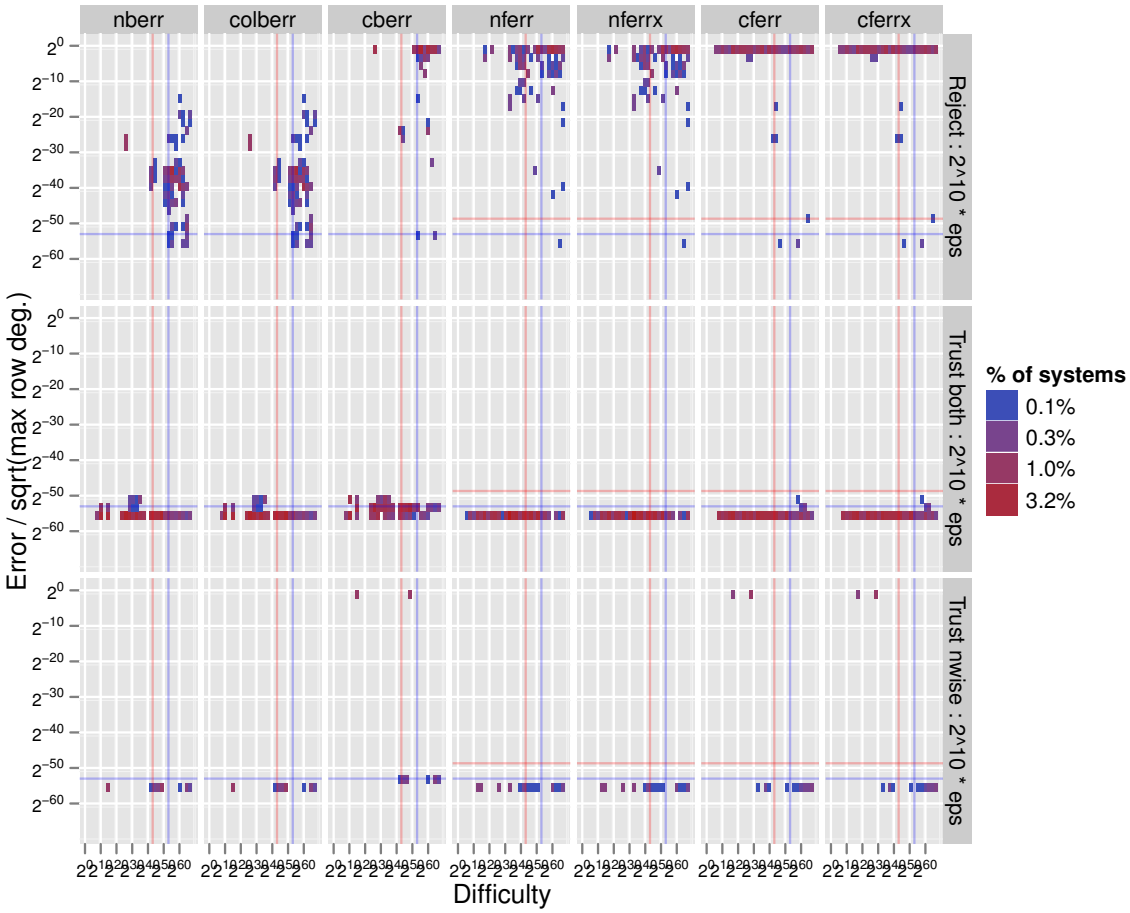


Figure 5.22: Errors after refinement for perturbations of size up to $\gamma = 2^{10}\varepsilon_f = 2^{-43}$ for the diagonal-relative heuristic show that refinement renders static pivoting dependable. The errors are scaled by the square-root of the maximum row degree in $L + U$ to normalize by the size c_N. The vertical blue line is $1/\varepsilon_f = 2^{53}$, and the vertical red line is $1/\gamma$.

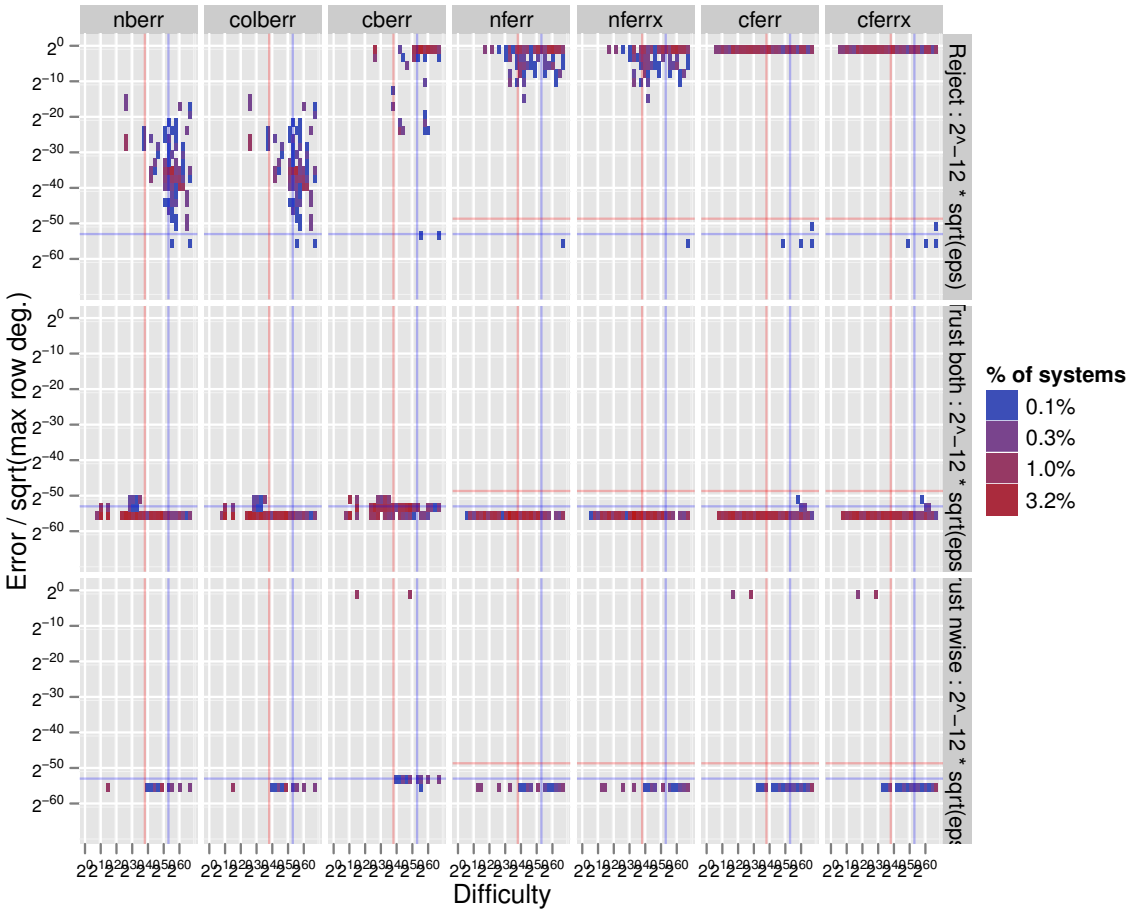


Figure 5.23: Errors after refinement for perturbations of size up to $\gamma = 2^{-38}$ for the diagonal-relative heuristic show that refinement renders static pivoting dependable. The errors are scaled by the square-root of the maximum row degree in $L + U$ to normalize by the size c_N. The vertical blue line is $1/\varepsilon_f = 2^{53}$, and the vertical red line is $1/\gamma$.

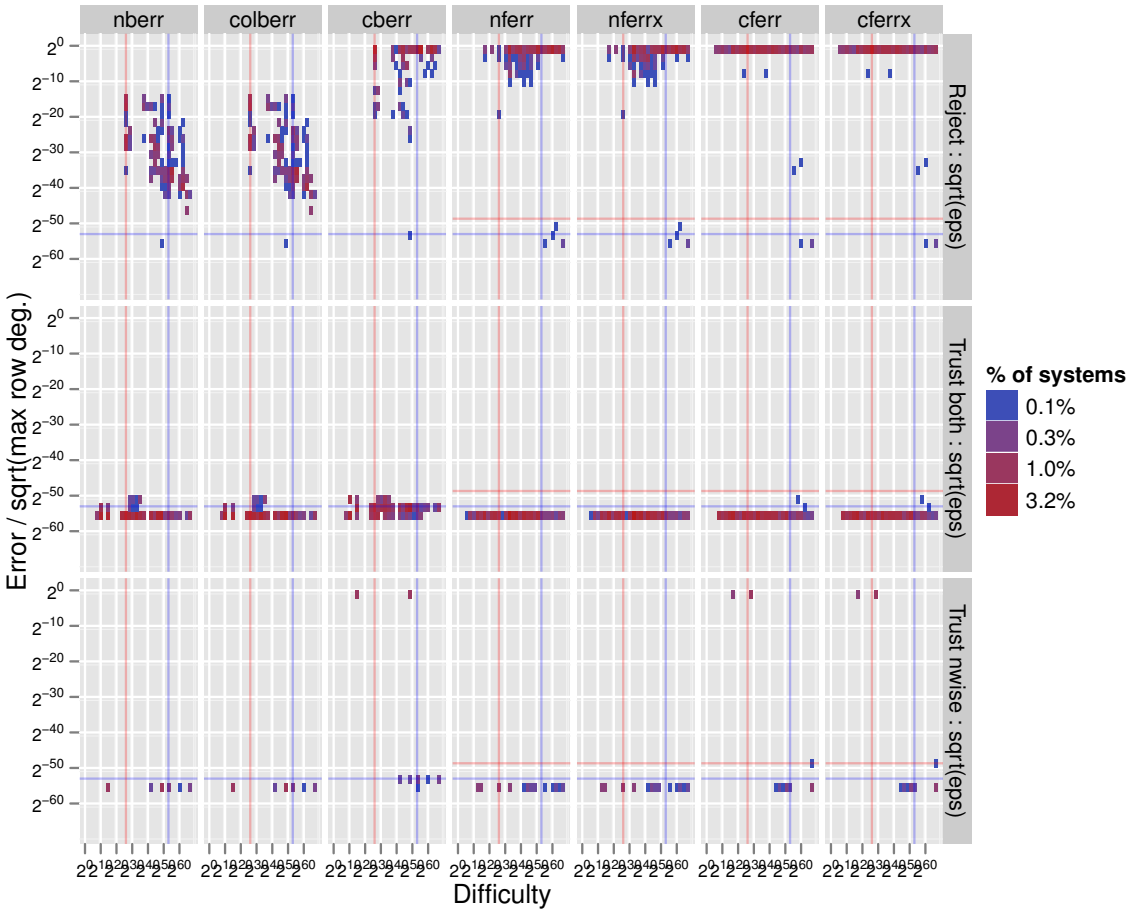


Figure 5.24: Errors after refinement for perturbations of size up to $\gamma = 2^{-26} \approx \sqrt{\varepsilon_f}$ for the diagonal-relative heuristic show that refinement renders static pivoting dependable. The errors are scaled by the square-root of the maximum row degree in $L + U$ to normalize by the size c_N. The vertical blue line is $1/\varepsilon_f = 2^{53}$, and the vertical red line is $1/\gamma$.

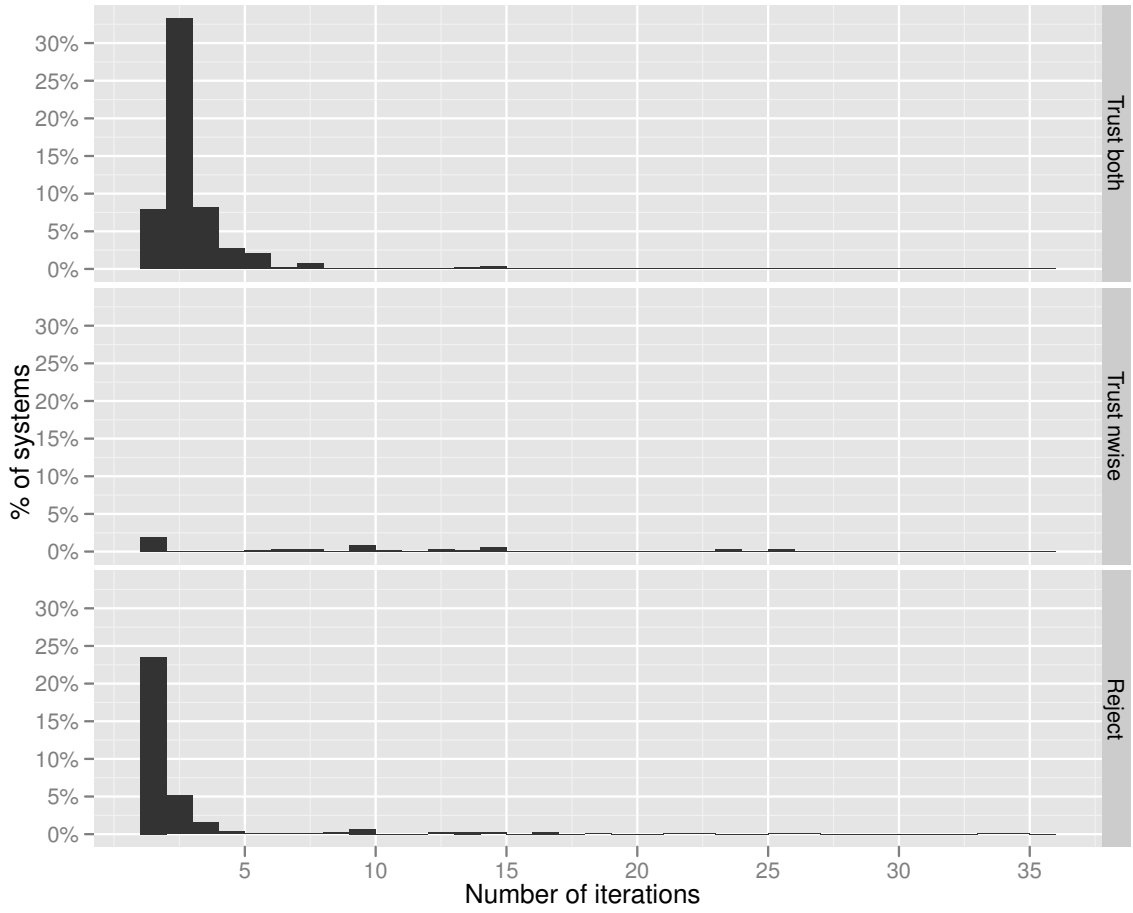


Figure 5.25: Iterations required by refinement for $\gamma = 2^{10}\varepsilon_f = 2^{-43}$ for the diagonal-relative heuristic. The right-hand label is the final status of the solution.

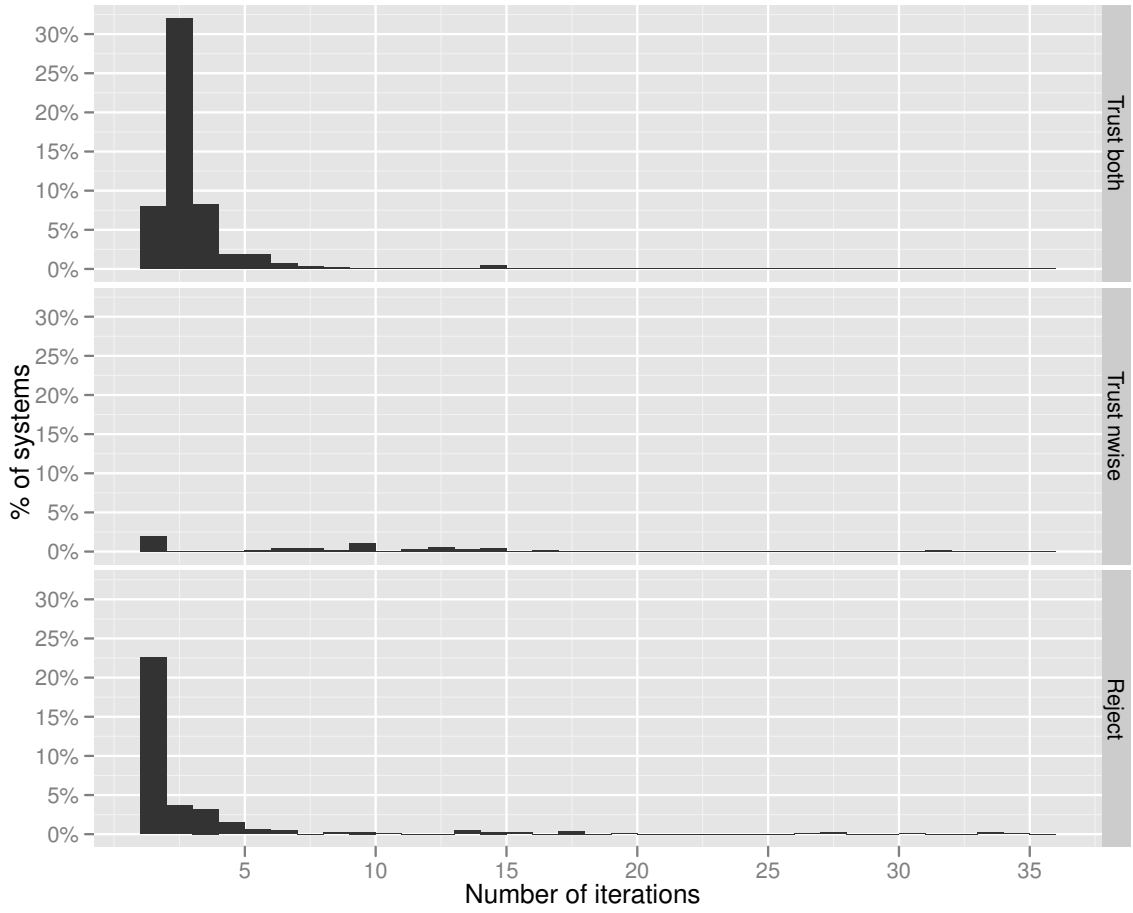


Figure 5.26: Iterations required by refinement by $\gamma = 2^{-38}$ for the diagonal-relative heuristic. The right-hand label is the final status of the solution.

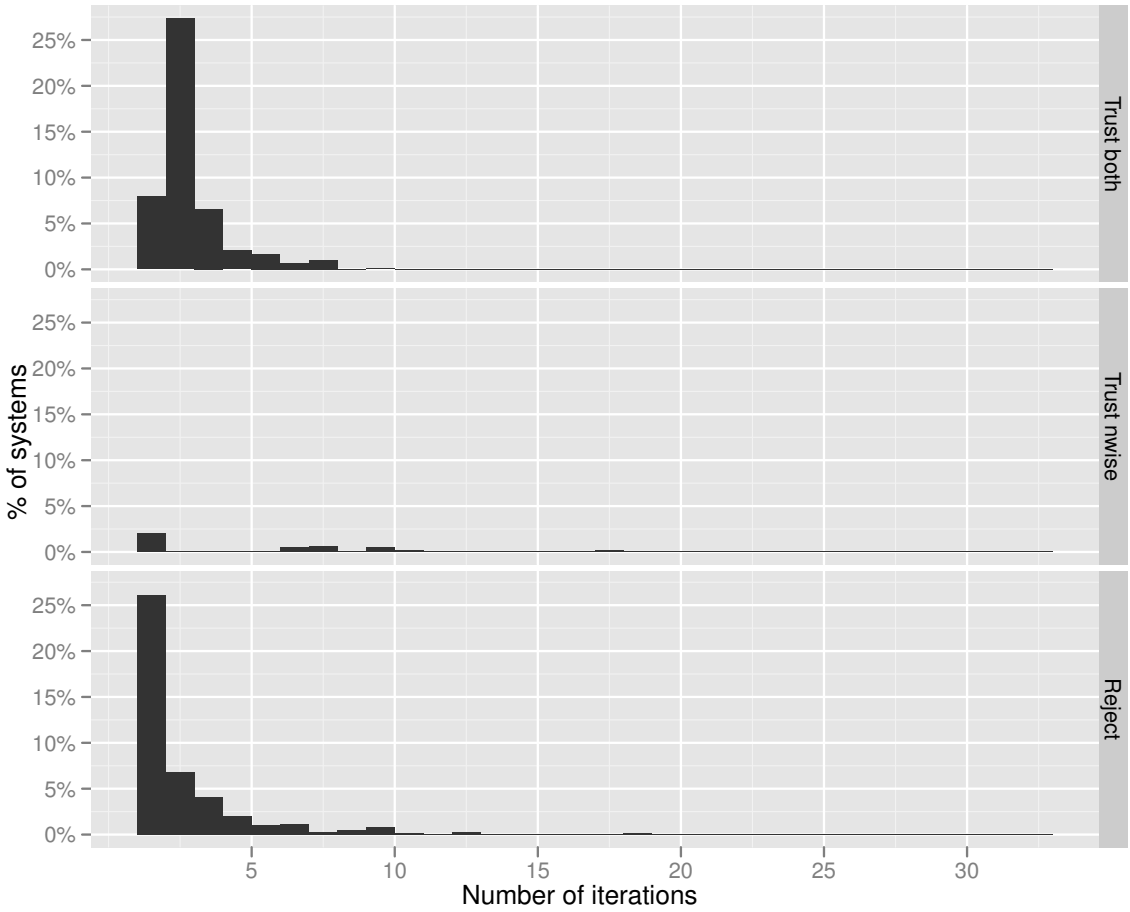


Figure 5.27: Iterations required by refinement by $\gamma = 2^{-26} \approx \sqrt{\varepsilon_f}$ for the diagonal-relative heuristic. The right-hand label is the final status of the solution.

Part II

Scalability: Distributed Bipartite Matching for Static Pivoting

Chapter 6

Matching and linear assignment problems for static pivot selection

6.1 Introduction

Scientific computing applies computational tools to scientific investigations. The tools generate data from models and then analyze data for conclusions. Many of these computational tools are *numerical*. In other words, the majority of the computer work consists of arithmetic operations on floating-point numbers. And many of these numerical tools can take advantage of large parallel computers for solving large problems. In general, the more numerical the tool, the more likely someone has used parallel computers to attack large problems.

But some large problems require less numerical arithmetic and more *symbolic* data manipulation. For example, searching protein databases for alignment matches interleaves bursts of calculation with large amounts of data manipulation[23]. Problems which involve searching for structures often have irregular structure themselves, and following the irregular structure requires symbolic work. Routines which support numerical computations, like those which assign processors to tasks [21], are highly symbolic. And within linear algebra, manipulations on general sparse matrices require significant efficient, irregular, and symbolic work to make sparsity worth-while [3].

The irregular symbolic calculations need to run in the same parallel and distributed environment as the numerical work does. Our goal herein is to examine one highly symbolic kernel from sparse matrix algorithms, to provide a parallel implementation, and to analyze its parallel performance. Weighted bipartite matching is used to pre-process matrices for parallel factorization [71], find appropriate basis vectors for optimization [83], and permute sparse matrices to block-triangular form [84]. More generally, weighted bipartite matching applies to many of the searching problems above. It also provides a central building-block for approximation algorithms related to many NP-complete problems.

Our central goal is to provide a distributed memory algorithm and implementation for

weighted bipartite matching. The matching will be used in SuperLU[71] for selecting static pivots. Section 5.4 demonstrates that we can produce dependable solutions with static pivoting. This part demonstrates a memory-scalable algorithm for choosing those pivots.

Previous work on parallel matching algorithms have found at most moderate speed-ups, and the algorithms assume that the entire graph is available on all processors. We generalize and implement the auction algorithm[12] for distributed computation. This algorithm gives an implementor a wide range of options for splitting up the computation, and we take full advantage to hide latencies and avoid synchronization. In the end, we find moderate speed-ups for many sparse problems, although the algorithm is plagued by horrible and unpredictable slow-downs on some problems.

6.2 Bipartite graphs and matrices

Throughout, let $G = \{\mathcal{R}, \mathcal{C}; \mathcal{E}\}$ be a bipartite graph with vertex sets $\mathcal{R}$ and $\mathcal{C}$ and edge set $\mathcal{E} \subset \mathcal{R} \times \mathcal{C}$. The vertices are separately enumerated, and we refer to them by number. So $\mathcal{R} = \{0, 1, \ldots, m-1\}$ and $\mathcal{C} = \{0, 1, \ldots, n-1\}$ with cardinalities $|\mathcal{R}| = m$ and $|\mathcal{C}| = n$. Unless otherwise noted, bipartite graphs are 'square', and $|\mathcal{R}| = |\mathcal{C}| = n$. We can assume non-square graphs have $n < m$ by transposing (swapping $\mathcal{R}$ and $\mathcal{C}$) if necessary.

All graphs we consider will have the same $\mathcal{R}$ and $\mathcal{C}$ as vertex sets. We abbreviate edges: $ij \in \mathcal{E}$ with $i \in \mathcal{R}$ and $j \in \mathcal{C}$. We say ij covers vertices i and j, and that i and j are adjacent to ij and each other. Also, we say i is in $\mathcal{E}$ if any edge in $\mathcal{E}$ covers i. Vertices with names based off i (*e.g.* i_1, i') will always be in $\mathcal{R}$, and j in $\mathcal{C}$. Because we use the same vertex sets for all graphs, an edge set $\mathcal{S} \subset \mathcal{R} \times \mathcal{C}$ often will be referred to as a graph itself, meaning the graph $\{\mathcal{R}, \mathcal{C}; \mathcal{S}\}$.

With any edge set (graph) $\mathcal{E}$ on $\mathcal{R}$ and $\mathcal{C}$ we associate a $\{0,1\}$-matrix $A(\mathcal{E})$, a specialized *adjacency matrix*. A standard adjacency matrix would have dimension $2n \times 2n$, but a bipartite graph admits a compressed, $n \times n$ representation. Vertices from the set $\mathcal{R}$ correspond to rows, and those from $\mathcal{C}$ to columns. We will often refer to vertex i as a row and vertex j as a column. Let $A = A(\mathcal{E})$ be the adjacency representation of G. The entry at the i^{th} row and j^{th} column of A is denoted $A(i, j)$, and the adjacency representation of $\mathcal{E}$ is

$$A(i,j) = \begin{cases} 1 & \text{if} \quad ij \in \mathcal{E}, \quad \text{and} \\ 0 & \text{otherwise.} \end{cases}$$

We will often conflate a graph, its edge set, and its adjacency matrix. Given a matrix A from edge set $\mathcal{E}$, saying $ij \in A$ is equivalent to $A(i, j) = 1$ and $ij \in \mathcal{E}$.

Given a graph $G = \{\mathcal{R}, \mathcal{C}; \mathcal{E}\}$, a *matching* on G is an edge set $\mathcal{M} \subset \mathcal{E}$ such that any vertex in G is covered by at most one edge in $\mathcal{M}$. If $ij \in \mathcal{M}$, then no other ij' or $i'j$ is in $\mathcal{M}$. If $|\mathcal{M}| \equiv \min\{|\mathcal{R}|, |\mathcal{C}|\} = \min\{m, n\}$, we call $\mathcal{M}$ *complete*. A complete matching on a square graph is also called *perfect*. A matching that is not complete is *partial*, although we

may refer also to a matching as partial to indicate that we do not know if it is complete or not. A vertex appearing in $\mathcal{M}$ is considered *covered* or *matched* by $\mathcal{M}$. A vertex not in $\mathcal{M}$ is *uncovered* or *exposed*.

The matrix representation $M = A(\mathcal{M})$ has additional structure beyond that of a general edge set's representation. Because $\mathcal{M}$ is a matching, any unit entry is the *only* unit entry on its row or column. If $\mathcal{M}$ is a perfect matching, then $|\mathcal{M}| = n$ and M has exactly n unit entries, one per row and one per column. M is thus a *permutation matrix*. If B is some matrix with rows from $\mathcal{R}$ and columns from $\mathcal{C}$, then the action $M^T B$ serves to relabel the rows to match the columns. If $ij \in M$, the j^{th} diagonal entry of $M^T B$ is $B(i, j)$.

If 1_r is an m-long column vector of unit entries, and 1_c is a similar n-long column vector, then the vectors $M1_c$ and $M^T 1_r$ for a general matching M are $\{0, 1\}$-vectors. If M is a perfect matching, then

$$\begin{aligned} M1_c &= 1_r, \quad \text{and} \\ M^T 1_r &= 1_c. \end{aligned}$$

We use the subscripts to denote the vertex set from which the vector indices are drawn. So $v_r = M1_c$ implies that $\forall i \in \mathcal{R}, v_r(i) = \sum_{j \in \mathcal{C}} M(i, j)1_c(j) \in \{0, 1\}$. An expression like $1_r^T 1_c$ is ill-formed with this convention; we would be conflating vertices from $\mathcal{R}$ and $\mathcal{C}$. However, $1_r 1_c^T$ is a perfectly acceptable matrix of all ones with rows drawn from $\mathcal{R}$ and columns from $\mathcal{C}$.

6.3 Maximum cardinality matchings

How do we know if a bipartite graph G admits a complete matching? There is a classical theorem of Hall [40] stating that every set of vertices on one side having at least as many neighbors on the other is necessary and sufficient. But rather than forming every possible set and testing its neighbors, we will use another classical result relating matchings and certain paths in G.

First, we construct a partial matching X with the simple, greedy algorithm in Listing 6.1. Listings use the programming language Python [99] as an executable pseudocode. Arrays are from Numeric Python [45] and are passed by reference. Data structures and non-obvious syntax will be explained in code comments, lines beginning with #, or the quoted strings after definitions.

Given a partial matching, we extend its cardinality through augmenting paths. An *alternating path* is a path in G which starts at any unmatched vertex and which alternates non-matching and matching edges. Alternating paths are cycle-free. An *augmenting path* is an alternating path which ends at an unmatched vertex. Another classical result relates augmenting paths to matchings.

Theorem 6.3.1 (Berge, Norman and Rabin)**:** A matching X is maximum if and only if it admits no augmenting paths. [70]

```
def greedy_matching (A):
    """Construct a partial matching for the bipartite
    graph represented by A. A is a sparse matrix; the
    explicitly stored entries are the edges of the graph.

    Returns an A.nrows()−long array X_r, where j == X_r[i]
    means ij is a matching edge. If X_r[i] == −1, then i
    has no match.
    """
    m = A.ncols()
    # fill X_r with −1
    X_r = −ones( A.nrows() )
    # iterate over j in [0,m), excluding m
    for j in range(m):
        # iterate over tuples of (i, A(i,j)) in the
        # jth column of A
        for (i, _) in A.col(j):
            if X_r[i] < 0:
                X_r[i] = j
                break
    return X_r
```

Listing 6.1: Greedy bipartite matching

If an augmenting path begins at a $j \in \mathcal{C}$, it must end at some $i \in \mathcal{R}$. Given an augmenting path P, we can increase the size of the matching X by one. To do so, remove from X each of the matching edges in P and add the non-matching edges, as shown in Listing 6.2.

So to find a maximum cardinality matching, we take some partial matching and find and apply a sequence of augmenting paths. Finding an augmenting path is easy. Starting from any unmatched $j \in \mathcal{C}$, look at all its neighbors in $\mathcal{R}$. If any such i is unmatched, we have a trivial augmenting path. If all such i are matched, pick one and let j' be its match. We can consider (j, i, j') a potential two steps of an augmenting path. Adding ij and removing ij' from X would leave j' free, so continue searching as if j' were free. One algorithm following these ideas is Duff's MC21 [46].

This algorithm has a worst-case complexity of $O(n|\mathcal{E}|)$, so $O(n^3)$ if $\mathcal{E} = \mathcal{R} \times \mathcal{C}$. Practically, sparse graphs, those with $|\mathcal{E}| = O(n)$ see linear run times. More complicated algorithms apply flow-based ideas to maximum cardinality matching. The best such algorithm, by Edmonds and Karp [52], runs in $O(\sqrt{n}|\mathcal{E}|)$ time.

Note that applying the first two steps along an augmenting path, adding the first edge and (implicitly) removing the second, leaves a matching and a shorter augmenting path. So long as we know that (j, i, j') appear along some augmenting path, we can remove ij' from and add ij to X before knowing the rest of the path. This property, that an augmenting path can be partially applied, appears again in the auction algorithm.

Theorem 6.3.2: Let M be a matching which admits an augmenting path P starting from some $j \in \mathcal{C}$. Let the first steps of P be (j, i, j'). Augmenting M with these steps and removing them from P produces a new matching M' and path P'. Then P' is an augmenting path for

```
def augment_path (X_r, j, i_list):
    """Augment the matching X_r. The augmenting path
    begins at column j. The rows are given in i_list;
    the last row must be unmatched.
    """
    for i in i_list:
        # j should be -1 only at termination
        assert -1 != j
        nextj = X_r[i]
        X_r[i] = j
        j = nextj
    # the last i should be unmatched
    assert -1 == j

def r_augment_path (X_r, j, i_list):
    """Recursive version of augment_path for exposition.
    After a single augmentation, X_r is still a matching,
    and the tail of i_list, i_list[1:], is again an
    augmenting path.
    """
    if -1 == j or i_list.empty(): return
    i = i_list[0]
    nextj = X_r[i]
    X_r[i] = j
    return r_augment_path (X_r, nextj, i_list[1:])
```

Listing 6.2: Augment a matching

M' starting from $j' \in \mathcal{C}$.

6.4 From combinatorics to linear optimization

So far, we have approached matching problems from a traditional, *combinatorial* view. This view is sufficient for finding maximum cardinality matchings. To find weighted matchings, we need machinery from a more general mathematical optimization framework.

As noted in the previous section, if X is a perfect matching, it satisfies the following constraints:

$$X 1_c = 1_r, \tag{6.4.1}$$

$$X^T 1_r = 1_c, \quad \text{and} \tag{6.4.2}$$

$$X \geq 0, \tag{6.4.3}$$

where the inequality is applied element-wise. These constraints define the *assignment polytope*. Bipartite matching problems will be stated as optimization problems over this polytope.

To support these optimization problems, we introduce two classical results:

Theorem 6.4.4 (Birkhoff)**:** The vertices of the assignment polytope are the $n \times n$ permutation matrices. [16, 100, 50]

```
def maxcard_mc21 (A):
    """Find a maximum cardinality matching on A.
    Algorithm due to Duff, 1981.
    """
    (m,n) = A.dims()
    X_r = -ones( (m) )
    # mark which vertices have been seen so far. If
    # seen[i] >= seen_phase, we've seen i in this pass.
    seen = zeros( (m) )
    seen_phase = 0
    def recurse (j):
        # Look for an unmatched i first.
        for i in A.colind(j):
            if X_r[i] < 0:
                X_r[i] = j
                return 1
        # If all i are matched, recurse along their
        # match edges.
        for i in A.colind(j):
            if seen[i] >= seen_phase: continue
            seen[i] = seen_phase
            jnew = X_r[i]
            if recurse (jnew):
                # toggle along the augmenting path
                X_r[i] = j
                return 1

    for j0 in xrange(n):
        # essentially, clear the seen flags
        seen_phase += 1
        recurse (j)
    return X_r
```

Listing 6.3: Maximum cardinality matching: MC21

Theorem 6.4.5: Extreme values of a linear optimization problem occur at the vertices of the constraint polytope. [19]

Together, these theorems state that any linear objective over the constraints (6.4.1)–(6.4.3) will achieve its extreme value at a permutation matrix. If the unit entries in the permutation matrix correspond to edges in G, then we have a perfect matching optimizing the objective.

With $X = A(\mathcal{M})$ representing a matching $\mathcal{M}$ on G, and A representing G itself, the number of edges in $\mathcal{M}$ is

$$|\mathcal{M}| = \sum_{ij \in \mathcal{E}} X(i,j) = \operatorname{Tr} A^T X.$$

This function is linear in X, suggesting the following optimization problem:

> **Optimization Problem 6.4.6** (Maximum Cardinality Matching)**.** Let A be the $n \times n$ matrix representing $G = \{\mathcal{R}, \mathcal{C}; \mathcal{E}\}$ with $A(i,j) = 1$ if $ij \in \mathcal{E}$ and 0 otherwise. Then the maximum cardinality matching problem is to
>
> $$\begin{aligned} \underset{X \in \Re^{n \times n}}{\text{maximize}} \quad & \operatorname{Tr} A^T X && \text{(6.4.6a)} \\ \text{subject to} \quad & X 1_c = 1_r, && \text{(6.4.6b)} \\ & X^T 1_r = 1_c, \text{ and} && \text{(6.4.6c)} \\ & X \geq 0. && \text{(6.4.6d)} \end{aligned}$$

By Theorem 6.4.5, the maximum occurs at a vertex. And Theorem 6.4.4 states that the vertex is a permutation matrix. But is X necessarily a complete matching on G? Not if G has no complete matching. Consider a graph with no edges, $\mathcal{E} = \emptyset$. Then *any* permutation matrix achieves the maximum of (6.4.6a), which happens to be zero. The maximum is indeed the size of the maximum cardinality matching, and we can find that matching by defining $\mathcal{M} = \{ij \mid ij \in X \quad \text{and} \quad ij \in \mathcal{E}\}$.

6.5 Related work

We are not the first to tackle parallel weighted bipartite matching, also known as the linear assignment problem. Parallel approaches exist even for the auction algorithm we implement[11, 15, 101], but none run on modern, distributed-memory computers. The only distributed memory auction algorithm we can find[22] has been tested only on artificial examples yet still suffers the same performance issues we find in Chapter 8.5. That algorithm also does not consider approximating the solution.

The auction algorithm we choose is not the only approach, either. The first proposed algorithm in von Neumann [100] required exponential time. The first polynomial-time algorithm is the Kuhn-Munkres Hungarian algorithm[67], noted to be $O(n^4)$ on dense instances

by Munkres [77], later tuned to $O(n^3)$ by Edmonds and Karp [52]. For dense instances, this remains an excellent method although as expensive asymptotically as matrix factorization. Amestoy et al. [5] attempt to short-cut the Kuhn-Munkres algorithm by determining feasible dual variables through numerical equilibration algorithms[90] then checking if there exists a greedy matching. The approach has some success but is not universal.

The Hungarian algorithm is a dual-based algorithm (see the optimization framework in Section 6.4) that works by extending augmenting paths. Working more directly with augmenting paths while manipulating the primal and dual leads to algorithms like those in MC64[47]. Parallel augmenting path approaches for older architectures also exist[104].

Flipping the problem into an equivalent flow- or circulation-based problem opens the door to parallel flow methods. Some have been published for shared memory machines[7], but distributed memory approaches for weighted (capacitated in the flow terminology) problems seem non-existent. More recent approaches to the unweighted problem deploy interesting heuristics to balance the communication and computation time[68], but it is unclear if the method can adapt to the weighted case.

Burkard and Çela [20] surveys the wide, wide variety of algorithms. We cannot cover every method here, but we do note that simplex methods tend to perform very poorly on weighted bipartite matching problems. Interior point algorithms are interesting and a potential route for the future, but converting fractional solutions into the integer solutions we need is challenging[10].

6.6 The benefit matrix

If the maximum value achieved by Problem 6.4.6 is not n, then we know that G has no perfect matching. If the non-zero entries of A were arbitrary real-valued weights rather than ones, the result would not be as clear-cut. Anticipating our problems, consider a matrix B defined with entries

$$B(i,j) = \begin{cases} 1 & \text{if} \quad ij \in \mathcal{E}, \quad \text{and} \\ -\infty & \text{otherwise.} \end{cases}$$

The product $B^T X$ involves $0 \cdot -\infty$, and so is generally undefined. However, X represents an edge set. We define $\operatorname{Tr} B^T X = \sum_{ij \in X} B(i,j)$ to avoid the problems with $0 \cdot -\infty$. Alternatively, a non-existent edge does not contribute to a weight, so we could simply define $0 \cdot -\infty = 0$. This definition will be consistent with all our uses.

Consider maximizing $\operatorname{Tr} B^T X$ over the assignment polytope. By construction, the maximum is $-\infty$ if and only if G has no perfect matching. This is true so long as all the entries in B corresponding to edges in $\mathcal{E}$ are finite.

Assume we are given a real valued weighting function $b(i,j)$ defined on the edges in $\mathcal{E}$.

Then the *benefit matrix* is

$$B(i,j) = \begin{cases} b(i,j) & \text{if} \quad ij \in \mathcal{E}, \quad \text{and} \\ -\infty & \text{otherwise.} \end{cases} \tag{6.6.1}$$

This is equivalent to extending b's domain to all $\mathcal{R} \times \mathcal{C}$ by setting $b(i,j) = -\infty$ when $ij \notin \mathcal{E}$.

Note that we can allow $b(i,j) = -\infty$ if $ij \in \mathcal{E}$, 'deleting' the edge ij, refusing to permit ij in the final matching. This can be quite useful for implementations. To simplify discussions, we assume that $b(i,j)$ is always finite when $ij \in \mathcal{E}$. However, setting $b(i,j) = \infty$ does *not* force (i,j) to be in a complete matching. The edge ij may not occur in any complete matching on G. Setting $b(i,j) = \infty$ also hides the benefits of other edges, as $\operatorname{Tr} B^T X = \infty$ whenever $ij \in \mathcal{M}$. The auction algorithm in Chapter 7 instead uses an infinite dual variable to force an edge into a matching.

6.7 The linear assignment problem

We want to find a 'best' matching on a bipartite graph G. To find a 'best' matching, we need a way to rate matchings. We choose a simple weighting: the sum of benefits $b(i,j)$ for each $ij \in \mathcal{M}$. This *benefit function* b provides a real-valued weight for each edge $ij \in \mathcal{E}$. Representing a complete matching as a permutation matrix X (Section 6.2) and defining a *benefit matrix* from b and G (Section 6.6) leads to a well-known linear optimization problem, the *linear assignment problem*. Notationally this problem is a slight change to Problem 6.4.6, replacing the 0-1 adjacency matrix A with a real-valued benefit matrix B.

> **Optimization Problem 6.7.1** (The Linear Assignment Problem, LAP)**.** Let B be an $n \times n$ matrix with entries in $\Re \cup \{-\infty\}$, and let 1_r and 1_c be n-long column vectors with unit entries. The *linear assignment problem*, or *LAP*, is to
>
> $$\begin{aligned} \underset{X \in \Re^{n \times n}}{\text{maximize}} \quad & \operatorname{Tr} B^T X && \text{(6.7.1a)} \\ \text{subject to} \quad & X 1_c = 1_r, && \text{(6.7.1b)} \\ & X^T 1_r = 1_c, \text{ and} && \text{(6.7.1c)} \\ & X \geq 0. && \text{(6.7.1d)} \end{aligned}$$

As stated, the LAP is always feasible, even when G has no complete matchings. We define B in Section 6.6 so that $\operatorname{Tr} B^T X = -\infty$ when G has no complete matching, and we will call this an insoluble problem rather than an infeasible one. Every solution to the LAP is a permutation matrix, but not every permutation matrix corresponds to a matching on G.

Theorem 6.7.2: Let B be defined from G and b in Section 6.6, let X_* be a solution to the LAP. Then X_* represents a complete matching on G if and only if $\operatorname{Tr} B^T X_* > -\infty$.

Proof. By Theorems 6.4.5 and 6.4.4, the solution X_* is a permutation matrix. We first demonstrate that $\operatorname{Tr} B^T X_* > -\infty$ implies that X_* represents a complete matching.

Treat $B^T X_*$ as a relabeling (or equivalently let $0 \cdot -\infty = 0$). The j^{th} diagonal entry of $B^T X_*$ is $B(i,j)$ where $X_*(i,j) = 1$. The sum of the diagonal entries is $> -\infty$ if and only if each $B(i,j) > -\infty$. By construction, this only occurs when $b(i,j)$ is defined, and hence $ij \in \mathcal{E}$. So the n unit entries in the permutation matrix X_* coincide with edges of $\mathcal{E}$, and X_* represents a complete matching.

Now let X_* be a solution to the LAP that also represents a complete matching $\mathcal{M}$. Then $\operatorname{Tr} B^T X_* = \sum_{ij \in \mathcal{M}} b(i,j)$. $\mathcal{M} \subset \mathcal{E}$, so each of the $b(i,j)$ with $ij \in \mathcal{M}$ is finite and $\operatorname{Tr} B^T X_* > -\infty$. □

We know that the LAP is always feasible, so by strong duality there is a dual problem with the same extreme value [19]. We derive the dual and two equivalent dual problems in Section 6.8. We want to re-use results from linear optimization theory, so in Section 6.9 we rephrase the LAP and its dual in *standard form*.

Given a permutation matrix X with $\operatorname{Tr} B^T X > -\infty$, how do we know if X is a solution to the linear assignment problem? Unweighted bipartite matching shows optimality through the lack of augmenting paths, but here we are given a complete matching and need to determine if it is a 'best' matching. Linear optimization theory provides the Karush-Kuhn-Tucker conditions [19], and Section 6.10 applies these conditions to our problem.

The LAP also permits a useful relaxation. Allowing the dual variables to deviate from the best possible by an additive factor of μ leads to matchings within an additive factor of $\mu(n-1)$ from the LAP optimum. Section 6.11 describes a relaxed optimality condition for testing a matching and its dual variables. The section also presents the relaxed problem as a barrier function form of the LAP, so a solution to the relaxed problem is along the LAP's central path.

Finally, section 6.12 applies augmenting paths to show that a soluble problem has bounded dual variables. The optimality conditions and boundedness will be useful in showing auction algorithm termination in Section 7.4.

6.8 The dual problem

The LAP is a feasible, linear optimization problem, and so it has a feasible, linear dual problem [19]. The dual problem is a new linear optimization problem with variables corresponding to the original, *primal* problem's constraints. In our case, the dual problem will not have tight equality constraints like (6.7.1b) and (6.7.1c), and so the dual variables will be easier for algorithms to manipulate.

The dual variables are the multipliers in the Lagrangian formulation of Problem 6.7.1. Let π_c and p_r be n-long column vectors, and let K be an $n \times n$ matrix. Then the LAP's

Lagrangian is

$$\begin{aligned}\mathcal{L}(X, \pi_c, p_r, K) &= \operatorname{Tr} B^T X - p_r^T (X 1_c - 1_r) - \pi_c^T (X^T 1_r - 1_c) + \operatorname{Tr} K^T X \\ &= \operatorname{Tr} \left(B - p_r 1_c^T - 1_r \pi_c^T + K\right)^T X + 1_r^T p_r + 1_c^T \pi_c.\end{aligned}$$

The assignment problem is linear and feasible. If X_* achieves the optimum value, then

$$\operatorname{Tr} B^T X_* = \min_{\pi_c, p_r, K} \max_X \mathcal{L}(X, \pi_c, p_r, K)$$

Maximizing over the primal variable X produces a function of π_c, p_r, and K that is $< \infty$ only under strict conditions. Because we know the optimum is $< \infty$, the conditions must hold. The maximization produces

$$\sup_X \mathcal{L}(X, \pi_c, p_r, K) = \begin{cases} 1_c^T \pi_c + 1_r^T p_r & \text{when } B - p_r 1_c^T - 1_r \pi_c^T + K = 0, \text{ and} \\ \infty & \text{otherwise.} \end{cases}$$

This maximization removes X from the problem, leaving the dual variables. We know the primal (the LAP) is feasible and achieves its optimum value, so the dual must as well. To find the optimum objective value, minimize over the Lagrangian dual variables. We state the minimization as another linear optimization problem.

Optimization Problem 6.8.1 (Full Dual Problem)**.** Given an assignment problem 6.7.1, its full dual problem is to

$$\begin{aligned} \underset{p_r, \pi_c, K}{\text{minimize}} \quad & 1_r^T p_r + 1_c^T \pi_c && \text{(6.8.1a)} \\ \text{subject to} \quad & p_r 1_c^T + 1_r \pi_c^T = B + K \quad \text{and} && \text{(6.8.1b)} \\ & K \geq 0. && \text{(6.8.1c)} \end{aligned}$$

The full dual above will be useful for putting the problem into standard form and for obtaining the optimality conditions. The $n \times n$ matrix K is unnecessary for computation. Additionally, as we will see in Section 6.10 below, K is dense, so maintaining K would be expensive. $K \geq 0$, so $p_r 1_c^T + 1_r \pi_c^T = B + K$ is equivalent to $p_r 1_c^T + 1_r \pi_c^T \geq B$, eliminating a variable and a constraint. By a traditional abuse of terminology, we refer to this simpler problem as the dual for the LAP.

Optimization Problem 6.8.2 (The Dual Problem)**.** The following problem is called the *dual problem* for the linear assignment problem 6.7.1 on page 126:

$$\begin{aligned} \underset{p_r, \pi_c}{\text{minimize}} \quad & 1_r^T p_r + 1_c^T \pi_c && \text{(6.8.2a)} \\ \text{subject to} \quad & p_r 1_c^T + 1_r \pi_c^T \geq B. && \text{(6.8.2b)} \end{aligned}$$

This problem is equivalent to 6.8.1.

We can remove the variable π_c, as well. By the constraint (6.8.2b), we have that $\pi_c(j) \geq B(i,j) - p_r(i)$ for all $i \in \mathcal{R}$. Minimizing $\pi_c(j)$ for each j also minimizes $1_c^T \pi_c$ and hence the dual's objective. The least possible $\pi_c(j)$ is

$$\pi_c(j) = \max_{i \in \mathcal{R}} B(i,j) - p_r(i). \tag{6.8.3}$$

Incorporating this directly into the objective gives an unconstrained problem.

Optimization Problem 6.8.4 (Unconstrained Dual Problem)**.** The unconstrained optimization problem

$$\underset{p_r}{\text{minimize}} \quad 1_r^T p_r + \sum_{j \in \mathcal{C}} \max_{i \in \mathcal{R}} (B(i,j) - p_r(j)) \tag{6.8.4a}$$

is dual to the LAP, and also equivalent to both 6.8.1 and 6.8.2.

This problem no longer appears linear, but defining π_c implicitly with (6.8.3) will be useful algorithmically. We will only need to store p_r, and the constraint (6.8.2b) in the dual will always be satisfied exactly regardless of arithmetic accuracy.

6.9 Standard form

Results from linear optimization theory apply to one of many *standard forms*. We use the same standard form as Boyd and Vandenberghe [19], Arbel [8], and others. The standard form problem is a *minimization* problem over a *vector* variable $\widetilde{x}$ subject to a linear equality and positivity constraints.

Optimization Problem 6.9.1 (Standard Form)**.** A linear optimization problem in standard form is to

$$\begin{aligned} \min_{\widetilde{x}} \quad & \widetilde{c}^T \widetilde{x} && \text{(6.9.1a)} \\ \text{subject to} \quad & \widetilde{A}\widetilde{x} = \widetilde{b}, \text{ and} && \text{(6.9.1b)} \\ & \widetilde{x} \geq 0. && \text{(6.9.1c)} \end{aligned}$$

The LAP, however, is a *maximization* problem over a *matrix* variable X. Two operations help map the LAP into standard form: the Kronecker product and vec. The vec operation maps $n \times n$ matrices onto n^2-long column vectors by stacking the columns. If $v = \operatorname{vec} V$, then $v((j-1) * n + i) = V(i,j)$. The Kronecker product of two matrices A and B is

$$A \otimes B = \begin{pmatrix} a(1,1)B & a(1,2)B & \cdots \\ a(2,1)B & a(2,2)B & \\ \vdots & & \end{pmatrix}.$$

Equating the LAP's objective (6.7.1a) with the standard problem's objective, we have $\operatorname{Tr} B^T X = \widetilde{c}^T \widetilde{x}$, so

$$\begin{aligned} \widetilde{c} &= \operatorname{vec} B, \quad \text{and} \\ \widetilde{x} &= \operatorname{vec} X. \end{aligned}$$

The positivity constraint translates immediately.

We now combine the remaining two equality constraints, $X 1_c = 1_r$ and $X^T 1_r = 1_c$, into a single, linear constraint $\widetilde{A}\widetilde{x} = \widetilde{b}$. Expanding the constraints in terms of $\widetilde{x}$ gives

$$\begin{aligned} \sum_j \widetilde{x}(n \cdot (j-1) + i) &= 1, \quad \forall i \in \mathcal{R} \quad \text{and} \\ \sum_i \widetilde{x}(n \cdot (j-1) + i) &= 1 \quad \forall j \in \mathcal{C}. \end{aligned}$$

If we set $\widetilde{b} = [1_r^T, 1_c^T]^T$ and

$$\widetilde{A} = \begin{pmatrix} 1_c^T \otimes I_n \\ I_n \otimes 1_r^T, \end{pmatrix}$$

then the constraint becomes $\widetilde{A}\widetilde{x} = \widetilde{b}$, and we have the problem in standard form. The constraint matrix $\widetilde{A}$ has rank $n-1$, which is appropriate because a permutation on n items has $n-1$ degrees of freedom. Note that $\widetilde{A}$ is also the *vertex-edge* matrix representation of a complete bipartite graph on $\mathcal{R}$ and $\mathcal{C}$. The first n rows correspond to vertices in $\mathcal{C}$, the second to $\mathcal{R}$, and the n^2 columns to the edges between them.

The standard form primal has a dual.

> **Optimization Problem 6.9.2** (Standard Form Dual)**.** The standard form dual to 6.9.1 is to
>
> $$\max_{\widetilde{y},\widetilde{k}} \; -1^T \widetilde{y} \tag{6.9.2a}$$
>
> $$\text{subject to } \widetilde{A}^T \widetilde{y} - \widetilde{k} + \widetilde{c} = 0, \text{ and} \tag{6.9.2b}$$
>
> $$\widetilde{k} \geq 0. \tag{6.9.2c}$$

Associating variables, we see that

$$\begin{aligned} \widetilde{y} &= [-p_r^T, -\pi_c^T]^T, \quad \text{and} \\ \widetilde{k} &= \operatorname{vec} K. \end{aligned}$$

6.10 Optimality conditions

To be an optimal solution for the primal and dual problems 6.7.1 and 6.8.2, a point (X, p_r, π_c) must satisfy a set of conditions known as the the Karush-Kuhn-Tucker conditions [19, 8].

The simplest of these conditions state that the solution must be feasible in both primal and dual, and that the Lagrangian's gradient must vanish at the solution. These conditions are trivial to satisfy for the LAP. The remaining condition, complementary slackness, is more interesting.

Complementary slackness states that at each coordinate either the dual or primal inequality is tight. In the standard form primal and dual, we have that $\widetilde{x} \odot \widetilde{k} = 0$, where $\odot$ denotes the element-wise or Hadamard product. Translating this back to our original problems, complementary slackness holds $X \odot K = 0$ for an optimal solution, . Substituting for K gives the *CS condition*

$$X \odot (p_r 1_c^T + 1_r \pi_c^T - B) = 0. \tag{6.10.1}$$

One consequence of Equation (6.10.1) is that an optimal K may have up to $n^2 - n$ non-zero entries, which is why we prefer the dual problem 6.8.2 without K.

With the implicit π_c defined in Equation (6.8.3), $\pi_c(j) = \max_i B(i,j) - p_r(i)$, we see an edge ij is in an optimum matching only when i achieves that maximum. If we call $p_r(i)$ the *price* of i, then $\pi_c(j)$ the *profit* of j and a matching edge must maximize j's profit. This observation becomes an important heuristic in Chapter 7's auction algorithm.

6.11 A relaxed assignment problem

If X represents some complete matching with dual variables p_r and π_c, how far is $\operatorname{Tr} B^T X$ from the optimum $\operatorname{Tr} B^T X_*$? The primal approaches its maximum from below, and the dual its minimum from above, so we have

$$\operatorname{Tr} B^T X \leq \operatorname{Tr} B^T X_* = 1_r^T p_r^* + 1_c^T \pi_c^* \leq 1_r^T p_r + 1_c^T \pi_c.$$

Now say X satisfies the *relaxed complementary slackness condition* for positive real μ,

$$X \odot \left((p_r - \mu 1_r) 1_c^T + 1_r \pi_c^T - B \right) \leq 0.$$

In other words, $p_r(i)$ exceeds the 'best' by up to μ. This implies that the inner product is non-positive, or

$$\operatorname{Tr}(p_r 1_c^T + 1_r \pi_c^T - \mu 1_r 1_c^T - B)^T X \leq 0.$$

Separating terms and rearranging gives

$$\operatorname{Tr}(p_r 1_c^T + 1_r \pi_c^T - \mu 1_r 1_c^T)^T X \leq \operatorname{Tr} B^T X.$$

The LAP equality constraints simplify the left side. We have

$$\begin{aligned} \operatorname{Tr}(p_r 1_c^T)^T X &= p_r^T X 1_c = p_r^T 1_r, \\ \operatorname{Tr}(1_r \pi_c^T)^T X &= 1_r^T X \pi_c = 1_c^T \pi_c, \quad \text{and} \\ -\mu \operatorname{Tr}(1_r 1_c^T)^T X &= -\mu 1_r^T X 1_c = -n\mu. \end{aligned}$$

And hence

$$1_r^T p_r + 1_c^T \pi_c \leq \operatorname{Tr} B^T X + n\mu. \tag{6.11.1}$$

Theorem 6.11.2: If a matching X satisfies the relaxed CS condition

$$X \odot \left((p_r - \mu 1_r)1_c^T + 1_r \pi_c^T - B\right) \leq 0, \tag{6.11.3}$$

then $\operatorname{Tr} B^T X$ is within an additive factor $n\mu$ of the LAP optimum value.

Proof. Combining the above, we have

$$\operatorname{Tr} B^T X \leq B^T X_* \leq 1_r^T p_r^* + 1_c^T \pi_c^* \leq 1_r^T p_r + 1_c^T \pi_c \leq \operatorname{Tr} B^T X + n\mu.$$

□

We can sharpen the bound to $(n-1)\mu$ either through examining the auction algorithm of Bertsekas [13] or by noting a relationship to a *barrier function* form. We informally follow the latter. First note that because $X(i,j) \in \{0,1\}$, μ can move to the right of the relaxed CS condition,

$$X \odot \left(p_r 1_c^T + 1_r \pi_c^T - B\right) \leq \mu 1_r 1_c^T.$$

Now we replace the matrix inequality in the LAP with a *barrier function*, an approximation to the indicator function for $X \geq 0$. Let $[\log X]$ be the matrix produced by applying log element-wise to X, and extend $\log x = -\infty$ when $x = 0$. Given a positive, real parameter μ, we use $\mu[\log X]$ as a barrier function.

Optimization Problem 6.11.4 (Barrier Function LAP)**.** For a given barrier parameter $\mu > 0$, the barrier function LAP is to

$$\begin{aligned}
\underset{X \in \Re^{n \times n}}{\text{maximize}} \quad & \operatorname{Tr} B^T X + \mu \sum_{i \in \mathcal{R}, j \in \mathcal{C}} [\log X](i,j) && \text{(6.11.4a)} \\
\text{subject to} \quad & X 1_c = 1_r, \quad \text{and} && \text{(6.11.4b)} \\
& X^T 1_r = 1_c. && \text{(6.11.4c)}
\end{aligned}$$

The objective is nonlinear, but the barrier problem is still a convex optimization problem. Casting the problem into a standard form and applying the existing theory [8] gives the following CS condition for the barrier problem:

$$X \odot \left(p_r 1_c^T + 1_r \pi_c^T - B\right) = \mu 1_r 1_c^T.$$

Any X satisfying this barrier CS condition lies on the *central path* and is off the optimum value of the non-barrier problem by at most an additive factor of $\operatorname{rank}(\widetilde{A})\mu = \mu(n-1)$. A solution satisfying the relaxed CS of Equation (6.11.3) is closer to optimum, hence also within $\mu(n-1)$.

6.12 A soluble problem has bounded dual variables

A soluble problem has a finite optimal value, so it's reasonable to believe that the dual variables p_r and π_c are also finite. This result will be useful in proving termination and bounding the complexity of various algorithms.

To find a bound, we need the concept of an *alternating path* from a $j \in \mathcal{C}$ in a partial matching $\mathcal{M}$. An alternating path as defined in Chapter 6.3 is a walk of adjacent edges starting from some unmatched vertex j and terminating at an unmatched vertex $i \in \mathcal{R}$. The walk crosses unmatched and matched edges alternately. The first edge is not in $\mathcal{M}$, the second is a matching edge in $\mathcal{M}$, the third is not, and so on.

If G has a complete matching, any incomplete matching $\mathcal{M}$ has an alternating path [40]. Flipping the edges along an alternating path increases $|\mathcal{M}|$ by 1 and keeps $\mathcal{M}$ a matching. We use this property to show that if an incomplete matching $\mathcal{M}$ represented by X has alternating paths, then the prices $p_r(i)$ of matched vertices $i \in \mathcal{R}$ are bounded. The bound is a function of the range of B's entries, the parameter μ, and the size of the problem. Some algorithms start from an arbitrary initial p_r, so we also incorporate the prices of the unmatched i.

Theorem 6.12.1: Let X be an incomplete matching on G with the associated dual variable p_r that satisfies the relaxed CS condition (6.11.3). If G has a complete matching, then

$$p_r(i) \leq \bar{p}_r + (n-1)(\mu + \bar{B})$$

for any i starting an augmenting path in X. Here $\bar{B} = |\max_{i,j} B(i,j) - \min_{i,j} B(i,j)|$ is the range of B's entries and $\bar{p}_r = \max_{\text{unmatched } i} p_r(i)$ is the largest price of an unmatched vertex $i \in \mathcal{R}$.

Proof. If X is incomplete and G has a complete matching, then there is an augmenting path from any unmatched $j \in \mathcal{C}$ to an unmatched $i \in \mathcal{R}$[40]. Let $(j_1, i_1, j_2, i_2, \ldots, j_t, i_t)$ be such a path, with edges $i_k j_{k+1} \in X$. We bound $p_r(i_1)$ and then apply this bound to all possible augmenting paths.

By the relaxed CS condition, the matching edges satisfy

$$p_r(i_k) - \mu + \pi_c(j_{k+1}) - B(i_k, j_{k+1}) \leq 0.$$

The dual feasibility constraint (6.8.2b) bounds the unmatched edges by

$$B(i_{k+1}, j_{k+1}) \leq p_r(i_{k+1}) + \pi_c(j_{k+1}).$$

Adding the two inequalities and rearranging to bound $p_r(i_k)$, we have

$$p_r(i_k) \leq p_r(i_{k+1}) + \mu + B(i_k, j_{k+1}) - B(i_{k+1}, j_{k+1}).$$

Adding all such inequalities over the entire path gives

$$p_r(i_1) \leq \bar{p}_r + t\left(\mu + \bar{B}\right).$$

The path can be at most $n-1$ steps long, so this in turn is bounded by

$$p_r(i_1) \leq \bar{p}_r + (n-1)\left(\mu + \bar{B}\right).$$

The augmenting path was chosen arbitrarily, so this bound applies to *all* augmenting paths. □

6.13 Manipulating the benefit matrix

The linear assignment problem's structure allows some manipulations of the benefit matrix without changing the optimizing permutations. The first polynomial-time assignment algorithm, the Hungarian algorithm defined in Kuhn [67], relies on basic manipulations. Other algorithms' performance depend on the benefit matrix's range; transformations to reduce that range are beneficial.

Clearly, any positive scalar multiple αB of the benefit matrix will have the solution with a similarly scaled objective value. Rank-1 perturbations of B that changes all columns or all rows equally just shift the objective function without moving the maximum permutation:
Theorem 6.13.1: Given two n-vectors u and v, the same permutation matrices maximize the linear assignment problem for benefit matrices $B \in \Re^{n\times n}$ and $B + u1_c^T + 1_r v^T$.

Proof. Adding a constant to the objective function does not alter the points where it is maximized, so the theorem is equivalent to demonstrating that the additional $u1_c^T + 1_r v^T$ terms shift the objective by a constant.

Expanding the objective function with cost matrix $B + u1_c^T + 1_r v^T$ yields

$$\begin{aligned}\operatorname{Tr}(B + u1_c^T + 1_r v^T)^T X &= \operatorname{Tr} B^T X + \operatorname{Tr} 1_c u^T X + \operatorname{Tr} v(1_r^T X) \\ &= \operatorname{Tr} B^T X + \operatorname{Tr} u^T X 1_c + \operatorname{Tr} v(X^T 1_r)^T \\ &= \operatorname{Tr} B^T X + u^T(X1_c) + \operatorname{Tr}(X^T 1_r)v^T.\end{aligned}$$

X is doubly stochastic, so $X1_c = 1_r$ and $X^T 1_r = 1_c$. The expanded objective function is

$$\operatorname{Tr}(B + u1_c^T + 1_r v^T)^T X = \operatorname{Tr} B^T X + u^T 1_r + v^T 1_c.$$

The term $u^T 1_r + v^T 1_c$ is constant with respect to X. □

This result holds only for square assignment problems. Non-square problems with $n < m$ permit only the 1_c term. If c_m is the smallest finite entry of B, then $B - c_m 1_r 1_c^T \geq 0$ for $ij \in \mathcal{E}$, so we can assume the finite benefits are non-negative.

Chapter 7

The auction algorithm

7.1 Introduction

To make the linear assignment problem 6.7.1 on page 126 more concrete, we turn to an algorithm for solving it. Dimitri Bertsekas's *auction algorithm* [12] finds an optimum complete matching through a competitive bidding process. Unmatched vertices $j \in \mathcal{C}$ corresponding to columns in B look through their adjacent rows $i \in B.\mathrm{col}(j)$ to find the most profitable match given current prices in p_r. The auction algorithm performs well in practice and can be generalized to asynchronous, parallel environments. This chapter explains the basic, sequential auction algorithm.

The basic auction algorithm is given in Listings 7.1 through 7.3, again using Python [99] as executable pseudocode. It takes as input a benefit matrix B constructed as in Section 6.6, holding information about the weights $b(i, j)$ and the connectivity of the bipartite graph G. The auction algorithm also takes an optimality parameter μ, and the algorithm delivers a matching within $\mu(n-1)$ of the optimum matching.

To summarize, the auction algorithm takes an unmatched $j \in \mathcal{C}$, finds a bid that will match j to some $i \in \mathcal{R}$, and then records the bid. Vertex j wins i by raising the price above that bid by any previously matched j', if one exists. The old match j' is now gathered with the remaining unmatched vertices, and the algorithm repeats until all vertices are matched through a winning bid. The algorithm can also terminate because the dual variable p_r, stored in the variable price, exceeds the bound in Theorem 6.12.1.

This chapter explains the auction algorithm, starting in Section 7.2 with the bid finding process. Infinite prices and nodes with only one adjacent edge require some care as explained in Section 7.3. Optimality and termination are proven in Section 7.4. We exploit the relationship with barrier methods in Section 7.5 to find a μ-scaling variation which should be more efficient. A basic algorithmic complexity result is provided in Section 7.6, and more ornate results from the literature are described. Then we present auction variations. The first is a combined forward-reverse auction which will perform well sequentially. Then Section 7.8

exploits the auction algorithm's freedom to find blocked auctions. Section 7.9 extends the block auctions to distributed parallel auctions, including restrictions on how the input graph is distributed.

7.2 Finding and placing bids

To find a bid from the vertices adjacent to j, we give to each such k a *value* $v(k) = B(k,j) - p_r(j)$. If $v(i) = \max_k v(k)$, then ij would satisfy the CS condition (6.10.1), as well as the relaxed condition (6.11.3). The bid will match j to some such i (there may be many). Let $v(i')$ be the second-largest value. Then incrementing the price by up to $v(i) - v(i')$ will leave i the best choice for j, and an additional increment of μ ensures that the algorithm avoids infinite loops. The new price for row i is

$$\begin{aligned} p_r(i) &:= p_r(i) + v(i) - v(i') + \mu \\ &= p_r(i) + B(i,j) - p_r(i) - v(i') + \mu \\ &= B(i,j) - v(i') + \mu. \end{aligned}$$

The remainder of this section explains *why* this bidding process works.

Bidding serves two purposes: entering new $\mathcal{R}$ vertices into the matching and driving down the dual objective. In either case, the new edge ij satisfies the relaxed CS condition, so i is the most profitable item for j. The algorithm never unmatches vertices from $\mathcal{R}$, so entering a new vertex i into X is forward progress.

The most profitable i may be matched to some j' already. Removing ij' and adding ij does not change the size of the partial matching. The new edge may not even increase the primal objective $\operatorname{Tr} B^T X$ or decrease the dual objective. The new edge does progress towards a solution, however.

Consider the implicit dual objective (6.8.4a):

$$\underset{p_r}{\text{minimize}}\, 1_r^T p_r + \sum_{j \in \mathcal{C}} \max_{k \in \mathcal{R}} \left(B(k,j) - p_r(k) \right)$$

The bidding process chose i to have the largest profit. Hence, if a bid from j increases $p_r(i)$ by δ, it also decreases the profit $\max_k B(k,j) - p_r(k)$ by δ. The change in contributions to the dual objective from i and j cancel.

It is very important that $\mu > 0$. When j and j' share two common neighbors i and i' with all corresponding entries in B equal, $\mu = 0$ could lead to an infinite loop.

If adding the edge ij required removing ij', then i had achieved $\max_k B(k,j') - p_r(k) = B(i,j') - p_r(i)$ for j'. If increasing $p_r(i)$ decreases this maximum, then we have made progress towards minimizing the dual. If increasing $p_r(i)$ does *not* decrease $\max_k B(k,j') - p_r(k)$, then there is some other node $i' \in \mathcal{R}$ which also achieved that maximum and is just as profitable for j' as i' was.

```
def match_unmatched (B, mu, unmatched, X_r, price_r,
                     soluble_bound):
    """Find a matching X on the bipartite graph represented
    by B. On unexceptional termination, all columns listed in
    unmatched are matched in X. The matching maximizes
    $\Tr B^T X$ to within an additive factor of mu * (B.ncols()-1).
    """
    for j in unmatched:
        bid = find_bid (B, j, price_r)
        # Make (i,j) satisfy the relaxed CS condition,
        # ensuring progress.
        bid.price += mu
        if bid.price > soluble_bound and finite(bid.price):
            raise Exception("Insoluble problem")

        old_match_j = record_bid (j, bid, X_r, price_r)
        # add_vertex(-1) is a no-op
        unmatched.add_vertex (old_match_j)

def auction (B, mu = -1):
    """Run an auction to produce a maximum weight, complete matching
    of B. Returns (X_r, price_r). If X_r[i] = j, then ij is in the
    matching. Optional parameter mu determines the quality of the
    matching; the matching is within an additive factor of
    mu*(B.ncols()-1) of the maximum weight matching.

    Raises "Insoluble problem" if there is no complete matching on B.
    """
    assert B.ncols() == B.nrows()
    n = B.ncols()
    if mu <= 0:
        # This mu ensures that we find _the_ optimal for
        # an integer-valued B
        mu = 1.0/n

    #Place all n columns into the unmatched list
    unmatched = UnmatchedList(n)
    # Initialize to zero prices and an empty matching
    price_r = zeros((n))
    X_r = -ones((n))
    soluble_bound = (n-1) * (mu + B.entry_range())

    match_unmatched (B, mu, unmatched, X_r, price_,
                     soluble_bound)
    return (X_r, price_r)
```

Listing 7.1: The basic auction algorithm

```
def find_bid (B, j, price):
    "Find a match for column j in B"
    bc = Collector()
    # A collector tracks the top two values seen,
    # along with the index and entry of the best
    # value.
    for (i, ent) in B.col(j):
        v = ent - price[i]
        bc.collect (v, i, ent)
    # bc.ent is the bid edge's entry in B, bc.v2nd
    # the second-best value, and bc.i the bid vertex i.
    return Bid( price = bc.ent - bc.v2nd, i = bc.i )
```

Listing 7.2: Finding a bid

```
def record_bid (j, bid, price, matching):
    "Update the matching and prices with the bid"
    assert bid.price > price[bid.i]
    old_match = matching[bid.i]
    matching[bid.i] = j
    price[bid.i] = bid.price
    return old_match
```

Listing 7.3: Recording a bid

Removing ij' from the matching makes j' eligible to bid for this i'. When i' is unmatched, then we have a short augmenting path which increases the size of the matching. A matched i' implies that (j, i, j', i') is the head of an alternating path. If G admits a complete matching, then this is also the head of an augmenting path. The bidding process applies (j, i, j'), which by Theorem 6.3.2 leaves a matching which also contains an augmenting path starting from j'. Essentially, the bid applies the head of an augmenting path before finding the entire path.

Augmenting paths may collide, in which case the more profitable wins. Rather than formalizing what happens when paths collide, we turn to the dual variables bounds from Section 6.12 to show termination. Termination is discussed in Section 7.4.

7.3 Infinite prices and required edges

Some edges in a graph may be present in every complete matching. If a vertex is adjacent to only one edge, then that edge must be in all complete matchings. Call the vertices adjacent to such a required edge *stubborn*. This requirement can cascade. If all but one of the vertices adjacent to a vertex j are stubborn, then the edge leading to that one edge is required.

Examining how stubborn vertices interact with the auction algorithm is useful not only because they arise in practice, but also because we may want to find a maximizing matching which includes a particular edge. Section 6.6 pointed out that we cannot set $B(i, j) = \infty$ to force ij into a matching, but we can set an infinite price $p_r(i)$ to keep any other vertices from

matching to i.

An infinite price also occurs on stubborn vertices in $\mathcal{R}$. Say i is the only vertex adjacent to j. In this case, the second best value is $-\infty$, so the new price is $B(i,j) - -\infty = \infty$. The infinite prices can cascade through a graph.

Mechanically calculating the resulting profit $\pi_c(j) = B(i,j) - p_r(i) = -\infty$ appears to yield an undefined dual objective $\sum_i p_r(i) + \sum_j \pi_c(j)$. However, $p_r(i) = \infty$ just encodes the information that ij cannot be removed from the matching. Because the edge is in the matching, we know that $p_r(i) + \pi_c(j) = B(i,j)$ no matter what $p_r(i)$ and $\pi_c(j)$ are. Hence, their contribution to the dual objective is always $B(i,j)$, and the dual remains well-defined.

The infinite price is naturally larger than the solubility bound. The finiteness test in line 13 of match_unmatched() in Listing 7.1 prevents the special price from artificially triggering the insolubility condition. If *all* the vertices adjacent to some unmatched j have infinite prices, then the problem is indeed insoluble. The strict inequalities in the bid finding procedure catch this case, as $-\infty$ value will never change the best i. This will be important in merging bids for the parallel algorithm, as well.

7.4 Optimality and termination

The auction algorithm terminates from one of two conditions: Either the algorithm found a complete matching X with associated prices p_r, or a price rose beyond a bound. In the first case, X satisfies the relaxed CS condition 6.11.3 on page 132 with its dual variables. So by Theorem 6.11.2, the matching is within an additive factor of $\mu(n-1)$ of the optimum LAP value.

Now if the algorithm has terminated because a price in p_r has increased beyond the bound of Theorem 6.12.1, then we know the matching X has no alternating paths. We terminate because of a bounds violation only when X is incomplete, so we have a matching in G with no augmenting paths. Hence G has no complete matching.

We have established the following result:

Theorem 7.4.1: The auction algorithm 7.1 on page 137 applied to a bipartite graph G always terminates. If it terminates with a complete matching X, then $\operatorname{Tr} B^T X \leq \operatorname{Tr} B^T X_* \leq \mu(n-1) + \operatorname{Tr} B^T X$, where X_* achieves the optimum for the linear assignment problem 6.7.1 on page 126. Otherwise, G has no complete matching.

Theorem 7.4.1 essentially is equivalent to the result of Bertsekas [12] but with a factor of $n-1$ rather than n.

7.5 μ-Scaling

The prices in Listing 7.1 need not start at zero. If the prices start closer to their optimum values, we would hope that the algorithm would minimize the dual, and hence terminate,

more quickly. And the larger μ, the more perfect matchings satisfy the relaxed CS condition, and the more quickly the algorithm terminates. If there is no perfect matching, then a larger μ increases prices towards the insolubility bound more quickly as well.

This suggests solving a sequence of problems with each problem's μ_k approaching our final μ. Going back to the barrier formulation in Section 6.11, this is equivalent to following the central path towards a solution, a key ingredient in interior-point algorithms. Choosing the sequence of μ_k is problem-dependent, but we choose a simple, general method. After each μ_k matching, we compute the primal and dual values, then lower μ_k to match the *achieved* gap. We then scale to a target $\mu_{k+1} = \mu/4$. Reducing the current μ_k to the gap actually achieved skips many μ steps for "easy" problems.

We need to exercise some care if we use $p_r(i) = \infty$ to denote an only-choice vertex. Clearing the match of i without clearing the price $p_r(i)$ could produce an insoluble situation. So whenever $p_r(i) = \infty$, the match of i should not be cleared or placed into the unmatched list.

Using the terminal μ as a tunable approximation factor as in Optimization Problem 6.11.4 can accelerate performance significantly even while returning a matching very near maximum weight, see Section 8.3.

Definition 7.5.1: We define the *relative gap* as

$$\text{relgap} \equiv \frac{1_r^T p_r + 1_c^T \pi_c - \operatorname{Tr} B^T X}{\operatorname{Tr} B^T X}.$$

By Theorem 6.11.2, the relative gap relgap $\leq n\mu$ at solution, and $\operatorname{Tr} B^T X$ is within an additive factor of $n\mu$ of the maximum weight matching.

We can select a terminal μ with an allowable approximation factor in mind.

7.6 Algorithmic complexity

Any individual price requires $O(\bar{B}/\mu)$ increments by μ to surpass the solubility threshold, and so no i receives more than $O(\bar{B}/\mu)$ bids. Also, once all n vertices in $\mathcal{R}$ receives a bid, the algorithm will terminate. This places an upper bound of $O(n\bar{B}/\mu)$ on the total number of bids placed. If d is the largest degree of a vertex in G, the most edges adjacent to any vertex, then each bid requires $O(d)$ work. This gives a basic complexity without μ-scaling of

$$O(nd\bar{B}/\mu).$$

We know that the algorithm may fail to terminate when $\mu = 0$, and the complexity diverges to ∞ appropriately. For 'dense' graphs, where $\mathcal{E} = \mathcal{R} \times \mathcal{C}$, the bound is $O(n^2\bar{B}/\mu)$, matching published results[12].

Note that the analysis does not impose requirements on the bidding order. There is a complex analysis by Bertsekas and Eckstein [14] which relates the auction algorithm with

a relaxed flow problem and imposes a particular ordering on the bids. The complexity of auction algorithms is a corollary from a much longer exposition, so we simply provide the results. For details, see [14]. This analysis finds a theoretical complexity of

$$O\left(n\tau \log(n\bar{B})\right) \tag{7.6.1}$$

for the μ-scaling auction, where $\tau = |\mathcal{E}|$. This author does not know of a proof for (7.6.1) that does not rely on a flow problem. Difficulties arise in relating solutions for different μ_k.

For instances of combinatorial optimization problems, the worst-case algorithmic complexity is often a very loose upper bound. The best cases are much faster, but the variability between different formats even of the same input make modeling performance difficult (Chapter 8).

7.7 Forward and reverse auctions

Consider the bipartite graph $G^T = \{\mathcal{C}, \mathcal{R}; \mathcal{E}^T\}$, where $ji \in \mathcal{E}^T \Leftrightarrow ij \in \mathcal{E}$. We can apply the auction algorithm just as well to G^T as to G. This is generally called a *reverse auction*. More interestingly, we can swap graphs *during* the auction algorithm for a combined *forward-reverse auction*.

Combined auctions do not have a single, monotonically increasing price vector p_r, so under what conditions will a combined auction terminate? Requiring an edge to be added to the matching before switching suffices. Auctions never decrease the number of edges in a matching. If no edge is ever added, then we stick with one direction and its dual variable. The bidding process in that direction will eventually drive that dual variable above the solubility bound calculated when entering that direction.

An implementation can choose between always maintaining the forward and reverse variables or converting the variables before switching. An algorithm to convert the variables is given in Listing 7.4.

Both directions maintain the relaxed CS condition (6.11.3), so a matching produced from either direction is μ-optimal. Some studies have indicated that the combined auction is less sensitive to the range of B and does not need μ-scaling for speed.

Rather than apply a full forward-reverse matching, we run a single price-shrinking pass on each μ phase. This occasionally helps shorten the long tail when chasing the final matching edge.

7.8 Blocked auctions

The algorithm in 7.1 dispatched bids one at a time. Consider the other extreme, where every unmatched column bids simultaneously. This variation of match_unmatched is shown in Listing 7.5. In the literature, this is referred to as a *Jacobi* auction, and single-bid variant as

```
def forw_to_rev (X_r, price_r, X_c = None, profit_c = None):
    """Convert the forward matching X_r and dual
    variable price_r on rows to the reverse matching
    X_c and dual profit_c on columns. Returns
    arrays X_c and profit_c, which may be input as
    optional arguments.
    This could be implemented in-place at a
    significant cost to the complexity constant.
    """
    n = len(X_r)
    if X_c is None: X_c = -ones(n)
    if profit_c is None: profit_c = array(n)
    # Find the profits.
    for j in range(n):
        profit = -infinity
        for (i, ent) in A.col(j):
            profit = max(profit, ent - price_r[i])
        profit_c[j] = profit
    # Invert the matching.
    for i in range(n):
        j = X_r[i]
        if j >= 0:
            X_c[j] = i
    return (X_c, profit_c)
```

Listing 7.4: Converting between forward and reverse auctions

a *Gauss-Seidel* auction [12]. Sequential simultaneous bidding is inefficient, but it provides a step towards a parallel auction.

The simultaneous bids need resolved before being committed to X and p_r. Lines 22–24 check that a new bid increases the price before committing it. In the case of a tie, the matching column with lesser index wins the bid. The tie breaking rule is arbitrary, but it must be consistent. This particular tie breaking rule is independent of the bid order, a very useful fact for debugging implementations. The single-bid auction found a bid which always passed this test, and so the price check is not included in Listing 7.1.

Before progressing to parallel auctions, we investigate a *blocked auction*. The blocked auction makes a parallel algorithm obvious but allows reasoning in a sequential environment. We partition the vertices of $\mathcal{C}$ and call each partition a block. Each block $\mathcal{C}^{(k)}$ is responsible for matching its vertices on the subgraph induced by removing all $j \notin \mathcal{C}^{(k)}$ from G. Call this restricted graph $G^{(k)}$. We run a full auction on each block, producing a complete matching for each $G^{(k)}$. These complete matchings are not necessarily optimal for the non-square problem.

If the matchings $X^{(k)}$ on each block are completely disjoint, then the union of these matchings is a perfect matching on the original graph G. Does this mean the matching $X = \cup_k X^{(k)}$ solves the LAP? If each block begins with the same prices, then yes, it does. Any i matched in any of the blocks remains matched throughout the auction algorithm. So if the matchings $X^{(k)}$ are each complete and distinct, then they have never bid for the same i, and

```
def match_unmatched_simultaneously (B, mu, unmatched,
                                    matching, price,
                                    soluble_bound):
    """Find a matching X on the bipartite graph represented
    by B. On unexceptional termination, all columns listed in
    unmatched are matched in X. The matching maximizes
    $\Tr B^T X$ to within an additive factor of mu * (B.ncols()-1).
    """
    bid = []
    for j in unmatched:
        bid.append( find_bid (B, j, price) )
    for k in range(len(unmatched)):
        bid[k].price += mu
        if bid[k].price > soluble_bound and finite(bid.price):
            raise Exception("Insoluble problem")
        j = unmatched[k]
        old_match = matching[bid[k].i]
        # no longer have bid[k].price > price[bid[k].i]
        if bid[k].price > price[bid[k].i] \
           or (bid[k].price == price[bid[k].i] \
               and j < old_match):
            matching[bid[k].i] = j
            price[bid[k].i] = bid[k].price
            unmatched.add_vertex (old_match_j)
```

Listing 7.5: An auction with simultaneous bids

```
def merge_vars (X1_r, price1_r, X2_r, price2_r, unmatchedlists):
    """Merge the primal and dual variables. Break ties by the bid's
    column index. The earlier columns win. Because the tie breaking
    rule is independent of the order of arguments, merge_vars is both
    associative and commutative.
    Returns the winning bids. Unmatched columns are returned to the
    unmatched lists.
    Note that every i matched in either X1 or X2 remains matched in
    _one_ of X1 and X2.
    """
    n = len(X_r)
    for i in range(n):
        if price2_r[i] > price1_r[i] or \
           (price2_r[i] == price1_r[i] and X2_r[i] > X1_r[i]):
            pricewin_r = price2_r
            (Xlose_r, pricelose_r) = (X1_r, price1_r)
        else:
            pricewin_r = price1_r
            (Xlose_r, pricelose_r) = (X2_r, price2_r)

        pricelose_r[i] = pricewin_r[i]

        old_match = Xlose_r[i]
        Xlose_r[i] = -1
        if old_match >= 0:
            unmatchedlists.add_vertex(old_match)
```

Listing 7.6: Merging block variables

the price changes also do not overlap. So the union matching X satisfies the CS conditions and solves the LAP.

We can also consider each matching $X^{(k)}$ with its corresponding dual $p_r^{(k)}$ as a sequence of bids. If $ij \in X^{(k)}$, then j bids for i with price $p_r^{(k)}(i)$. If no i appears in more than one $X^{(k)}$, then each bid wins, showing again that the union X solves the LAP. But if some i does appear multiple times, then one bid must lose.

The losing bid is placed back into an unmatched list, just as for the single-bid auction, and the blocked auction is repeated until no unmatched columns remain. Listing 7.6 implements this merging by scanning the entire matching. A more efficient version need scan only the changes, tracked through a common scatter-gather mechanism.

Note that Listing 7.6 does not specify in which block an unmatched column old_match is placed. With the merging operator, we can define a blocked auction algorithm.

The sequential all-to-all reduction reduce_vars() can be specified as in Listing 7.8. The tie-breaking tests result in an idempotent operator. Also, because the tie breaking rule does not depend on the argument order, the merging operation is commutative. Merging blocks k and k' will produce the same result regardless of the order they are presented to Listing 7.6.

Once all the unmatched lists are empty, there must be no conflicting bids. Listing 7.9 constructs the final answer, including relevant tests as assertions.

Theorem 7.8.1: Listing 7.7 computes a solution to the LAP when a perfect matching exists

```
def blocked_auction (B, mu = -1):
    """Run an auction to produce a maximum weight, complete
    matching of B. Returns (matching, price). If
    X_r[i] = j, then ij is in the matching. Optional
    parameter mu determines the quality of the matching; the
    matching is within an additive factor of mu*(B.ncols()-1)
    of the maximum weight matching.

    Raises "Insoluble problem" if there is no complete matching on B.
    """
    assert B.ncols() == B.nrows()
    n = B.ncols()
    if mu <= 0:
        # This mu ensures that we find _the_ optimal for
        # an integer-valued B
        mu = 1.0/n

    # Place all n columns into unmatched lists, blocked
    # according to the implementation
    unmatchedlists = UnmatchedLists(n)
    n_blocks = len(unmatchedlists)

    # Initialize to zero prices and an empty matching
    price_rs = []; X_rs = []
    for k in range(n_blocks):
        price_rs.append( zeros(n) )
        X_rs.append ( -ones(n) )
    soluble_bound = (n-1) * (mu + B.entry_range())

    X_r = None
    price_r = None
    while not all_empty (unmatchedlists):
        for k in range(n_blocks):
            match_unmatched (B, mu, unmatchedlists[k],
                             X_rs[k], price_rs[k],
                             soluble_bound)
        (X_r, price_r) = union_merge (X_rs, price_rs)

        # Perform an all-to-all reduction across the
        # blocks. The reduction operator is merge_vars,
        # an associative and commutative operation.
        reduce_vars (X_rs, price_rs, unmatchedlists)
    return final_merge(X_rs, price_rs)
```

Listing 7.7: A blocked auction algorithm

```
def reduce_vars (X_rs, price_rs, unmatchedlists):
    """Sequentially perform an all−to−all merge between the
    matchings and dual variables.f
    """
    n_blocks = len(X_rs)
    n = len(X_rs[0])
    for k in range(1,n_blocks):
        merge_vars (X_rs[0], price_rs[0], X_rs[k], price_rs[k],
                    unmatchedlists)
    for k in range(1,n_blocks):
        merge_vars (X_rs[0], price_rs[0], X_rs[k], price_rs[k],
                    unmatchedlists)
```

Listing 7.8: Sequential reduction of auction variables

```
def union_merge (X_rs, price_rs, X_r = None, price_r = None):
    """Produce the union matching and price. Conflicting
    bids yield an AssertionError, as does an incomplete
    matching.
    """
    n_blocks = len(X_rs)
    n = len(X_rs[0])

    # Explicitly copy the first vars.
    X_r = array(X_rs[0])
    price_r = array(price_rs[0])

    for k in range(1,n_blocks):
        for i in range(n):
            if X_rs[k][i] >= 0:
                assert X_r[i] == −1
                X_r[i] = X_rs[k][i]
                price_r[i] = price_rs[k][i]
    assert −1 not in X_r

    return (X_r, price_r)
```

Listing 7.9: The blocked auction's final merge

and signals a problem insoluble otherwise.

Proof. Treat the result of each auction on a block as a sequence of bids. The merge operation in Listing 7.6 merges two sequences of bids by playing them against each other. The tie-breaking rule is equivalent to placing bids in increasing j-order, and a bid must increase a price to be accepted.

So running auctions on blocks and merging their results is equivalent to producing a partial matching through a sequence of price-increasing bids. The partial matching satisfies the CS condition (6.10.1). If no columns are left unmatched, then we have a perfect matching that satisfies CS and hence a solution of the LAP. All successful bids increase prices, so if there is no perfect matching, a price will increase beyond the bound of Theorem 6.12.1 and the algorithm will signal that the problem is insoluble. □

7.9 Parallel auctions

In the blocked auction of Listing 7.7 on page 145, the blocks do not interact until their variables are merged. Hence each block auction can run in parallel. The overall parallel algorithm is as follows:

1. Repeat:
   (a) Run a blocked local auction.
   (b) Collect all changed prices as global bids.
   (c) Collectively synchronize global bids.
   (d) If another processor out-bids the local processor for a row, place the column back on the local unmatched list. (Ties are broken arbitrarily by preferring the lower-ranked processor.)
2. Until there are no global bids outstanding.

Unlike most previous parallel algorithms, a parallel auction works with a distributed matrix B. Each block auction accesses only the columns within its unmatched list. If columns are placed back on the same list as they are unmatched, then we distribute B across processors by column according to the initial unmatched list partitioning. No processor needs the entire matrix.

Splitting B by columns allows a good amount of freedom in distributing B. Distributing B by explicitly stored entries appears even more attractive. However, the auction algorithm does need the best and second-best value from the *entire* column of B. Finding a bid in a subset of the column could pick the wrong target, starting along an alternating path which is *not* augmenting. Because having multiple processors finding a single bid involves frequent

communication within an inner loop, we will not explore that option even though two-dimensional, hypergraph partitioning[24] holds potential for minimizing that communication.

By not collaborating on individual bids, all communication is encapsulated in the merge operation. Essentially, each processor poses its changed prices as global bids. Those bids are sent to other processors that share the same rows, either individually or through a collective reduction or gather. For processor counts small enough such that each processor holds a significant portion of the matrix (a few thousand columns), packing outgoing bids into a buffer and applying a simple `MPI_Alltoallv` all-to-all collective operation[75] suffices to achieve memory scalability. In our current implementation, however, we use $O(N)$ data per node and call `MPI_Allgatherv` to collect all prices and current matches from every process. For the parallel system sizes of interest, this performs better than other MPI-provided or hand-written collectives.

Optimizing for pure message passing across a massive number of nodes is feasible but of decreasing interest. Current parallel systems are built with a large amount of memory per node shared among multiple processors. The number of pathways to the shared memory is a more important constraint than the amount of memory used. Also, current interconnection networks are moving towards supporting more fine-grained global access. Future work will investigate using PGAS languages like UPC[26] or X10[94] rather than pushing MPI into unnatural corners.

Chapter 8

Auction performance

8.1 Summary

Ultimately, the performance of auction algorithm implementations is highly unpredictable and not directly correlated to dimension or number of matrix entries. The augmenting path implementation MC64[47] shows similar sequential variability. Parallel performance can drop off a cliff when the algorithm forms an almost completely wrong partial matching and rebuilds the matching one parallel phase at a time.

Section 8.2 discusses basic sequential performance and demonstrates high variability depending on the input format for the same data. Section 8.3 loosens the approximation factor for the matching's weight and sees drastically improved sequential performance for little weight loss. Section 8.5 discusses parallel performance with all its warts.

All our auction results use a C implementation of the algorithms presented earlier. Duff and Koster [47]'s MC64 is written in Fortran but is driven by the same C driver as our implementations. The implementations are available from the author[1].

8.2 Sequential performance

We test the performance of our sequential auction implementation on computing a permutation that makes the matrix's diagonal's product as large in magnitude as possible. The input matrix A is transformed to a benefit matrix B by $B(i,j) = \log_2 |A(i,j)|$. Zero entries in A become $-\infty$ edges in B and are never chosen.

All of the presented times report the least of three randomly scheduled runs. The performance was timed on a 2.66 GHz Intel Xeon X5550 after compilation with gcc and gfortran 4.4[2]. The matrices in Table 8.1 were chosen while collaborating with Bora Uçar while

[1] Currently at `http://lovesgoodfood.com/jason/cgit/index.cgi/matchpres/` .

[2] `http://gcc.gnu.org`

Group	Name	Dimension	# entries
Bai	af23560	23560	460598
FEMLAB	poisson3Db	85623	2374949
FIDAP	ex11	16614	1096948
GHS_indef	cont-300	180895	988195
GHS_indef	ncvxqp5	62500	424966
Hamm	scircuit	170998	958936
Hollinger	g7jac200	59310	717620
Hollinger	g7jac200sc	59310	717620
Mallya	lhr14	14270	305750
Mallya	lhr14c	14270	307858
Schenk_IBMSDS	3D_51448_3D	51448	537038
Schenk_IBMSDS	ibm_matrix_2	51448	537038
Schenk_IBMSDS	matrix_9	103430	1205518
Schenk_ISEI	barrier2-4	113076	2129496
Vavasis	av41092	41092	1683902
Zhao	Zhao2	33861	166453

Table 8.1: Test matrices for sequential auction and MC64 performance comparisons. Chosen to match Riedy [87], work in consultation with Bora Uçar. "Group" refers to the directory within the UF Sparse Matrix Collection, and "Name" is the base file name.

visiting CERFACS[87]. They were chosen to illustrate problems with each of the outstanding methods.

Table 8.2 shows overall performance on our matrix testbed running with algorithm defaults and comparing to MC64. Both MC64 and the auction algorithm are provided the input in column-major (compressed sparse column) format. MC64 works with the input and its transpose, while the auction algorithm works only with the input. Both of these implementations run relatively quickly, but neither dominates the other in performance.

Table 8.3 shows the first strange variability. Simply transposing the input matrix can change the running time by an *order of magnitude*. Such large variability makes predicting the running time difficult and befuddles obvious performance modeling attempts.

Table 8.4 shows less variability when rounding the input benefit matrix from $\log_2 |A(i,j)|$ to $\lfloor \log_2 |A(i,j)| \rfloor$. This affects the matching itself little. Reducing the benefit matrix to smaller integers may reduce memory pressure and balance CPU functional unit usage.

8.3 Approximations for faster matching

One benefit of the auction algorithm over other currently implemented algorithms is in tuning the quality of the output matching. We can return a less-than-maximum weight matching

Group	Name	Auction time (s)	MC64 time (s)	MC64/Auction
Bai	af23560	0.025	0.017	0.68
FEMLAB	poisson3Db	0.014	0.040	2.74
FIDAP	ex11	0.060	0.015	0.26
GHS_indef	cont-300	0.007	0.019	2.89
GHS_indef	ncvxqp5	0.338	0.794	2.35
Hamm	scircuit	0.048	0.024	0.50
Hollinger	g7jac200	0.355	0.817	2.30
Hollinger	g7jac200sc	0.304	0.678	2.23
Mallya	lhr14	0.044	0.026	0.60
Mallya	lhr14c	0.089	0.054	0.61
Schenk_IBMSDS	3D_51448_3D	0.031	0.010	0.33
Schenk_IBMSDS	ibm_matrix_2	0.031	0.008	0.27
Schenk_IBMSDS	matrix_9	0.074	0.024	0.33
Schenk_ISEI	barrier2-4	0.291	0.044	0.15
Vavasis	av41092	5.462	3.595	0.66
Zhao	Zhao2	1.041	3.237	3.11

Table 8.2: Performance comparison between a μ-scaling auction implementation and an optimized augmenting path implementation (mc64). Both are applied in a column-major fashion to maximize the product of the matching given floating-point data. They achieve the same total matching weight up to round-off. Performance is extremely variable, but both are quite fast. Performance ratios over 2 and under 0.5 are outside run-to-run variability.

Group	Name	Col-major (s)	Row-major (s)	Row/Col
Bai	af23560	0.025	0.028	1.13
FEMLAB	poisson3Db	0.014	0.016	1.11
FIDAP	ex11	0.060	0.060	1.00
GHS_indef	cont-300	0.007	0.006	0.84
GHS_indef	ncvxqp5	0.338	0.318	0.94
Hamm	scircuit	0.048	0.047	0.99
Hollinger	g7jac200	0.355	0.339	0.95
Hollinger	g7jac200sc	0.304	0.232	0.77
Mallya	lhr14	0.044	0.065	1.47
Mallya	lhr14c	0.089	0.075	0.85
Schenk_IBMSDS	3D_51448_3D	0.031	0.282	**9.22**
Schenk_IBMSDS	ibm_matrix_2	0.031	0.275	**9.02**
Schenk_IBMSDS	matrix_9	0.074	0.613	**8.29**
Schenk_ISEI	barrier2-4	0.291	0.193	0.66
Vavasis	av41092	5.462	4.083	0.75
Zhao	Zhao2	1.041	0.609	0.58

Table 8.3: Performance comparison between applying μ-scaling auction implementation in a column-major fashion to applying the same implementation in a row-major fashion (transposed). In some cases, performance differences are dramatic ands unpredictable. Performance ratios over 2 and under 0.5 are outside run-to-run variability.

Group	Name	Float (s)	Int (s)	Int/Float
Bai	af23560	0.025	0.040	1.61
FEMLAB	poisson3Db	0.015	0.016	1.08
FIDAP	ex11	0.060	0.029	0.49
GHS_indef	cont-300	0.007	0.006	0.91
GHS_indef	ncvxqp5	0.338	0.425	1.26
Hamm	scircuit	0.048	0.016	0.34
Hollinger	g7jac200	0.355	1.004	2.83
Hollinger	g7jac200sc	0.304	0.987	3.25
Mallya	lhr14	0.044	0.050	1.12
Mallya	lhr14c	0.089	0.137	1.55
Schenk_IBMSDS	3D_51448_3D	0.031	0.020	0.66
Schenk_IBMSDS	ibm_matrix_2	0.031	0.020	0.66
Schenk_IBMSDS	matrix_9	0.074	0.066	0.89
Schenk_ISEI	barrier2-4	0.291	0.261	0.91
Vavasis	av41092	5.462	5.401	0.99
Zhao	Zhao2	1.041	2.269	2.18

Table 8.4: Performance comparison between applying μ-scaling auction implementation in a column-major fashion given a floating-point benefit matrix or after rounding the matrix to integers. Again, performance varies although rarely as dramatically. Performance ratios over 2 and under 0.5 are outside run-to-run variability.

Name		Terminal μ value 0	5.96e-08	2.44e-04	5.00e-01
af23560	Primal	1342850	1342850	1342850	1342670
	Time(s)	0.14	0.05	0.03	0
	ratio		0.37	0.21	**0.02**
poisson3Db	Primal	2483070	2483070	2483070	2483070
	Time(s)	0.02	0.02	0.02	0.02
	ratio		1.01	1.04	1.07
ex11	Primal	963560	963560	963560	962959
	Time(s)	0.11	0.05	0.02	0.01
	ratio		0.44	0.22	**0.07**
cont-300	Primal	902982	902982	902982	902982
	Time(s)	0.01	0.01	0.01	0.01
	ratio		1.02	1.04	**0.91**
ncvxqp5	Primal	1119520	1119520	1119520	1116780
	Time(s)	1.27	0.53	0.15	0.02
	ratio		0.42	0.12	**0.01**
scircuit	Primal	8720900	8720900	8720900	8720900
	Time(s)	0.12	0.02	0.01	0.01
	ratio		0.14	0.07	**0.07**
g7jac200	Primal	3533980	3533980	3533980	3533340
	Time(s)	2.98	1.07	0.28	0.18
	ratio		0.36	0.09	**0.06**
g7jac200sc	Primal	3518610	3518610	3518610	3518080
	Time(s)	1.32	0.56	0.73	0.12
	ratio		0.42	0.55	**0.09**

Table 8.5: Approximating the maximum weight of a maximal matching affects that weight only slightly for integer problems but can provide dramatic performance improvements. (Continued in Table 8.6.)

by increasing the relative gap defined in Definition 7.5.1 between the primal and dual. This pays of handsomely in sequential performance.

Tables 8.5 and 8.6 give the final matching weight and time required on our test matrices. The inputs here are rounded to integers, and the implementation differs from Section 8.2's implementation by not reducing prices between μ-stages as mentioned in Section 7.7. This seemingly minor change causes av40192's running time to explode from around 5 seconds to over 25 seconds. By permitting a relative gap between the primal (matching weight) and dual of 0.5, however, the time is reduced to slightly over a tenth of a second. Speed-ups in the range of 10× to 100× at the largest μ value of 0.5 appear common; see the bold ratios in Tables 8.5 and 8.6.

Name		Terminal μ value 0	5.96e-08	2.44e-04	5.00e-01
lhr14	Primal	10698500	10698500	10698500	10698400
	Time(s)	0.09	0.05	0.03	0.02
	ratio		0.59	0.39	**0.19**
lhr14c	Primal	14223100	14223100	14223100	14215400
	Time(s)	0.35	0.15	0.06	0.03
	ratio		0.43	0.16	**0.09**
3D_51448_3D	Primal	3910020	3910020	3910020	3908000
	Time(s)	0.08	0.03	0.01	0.003
	ratio		0.39	0.18	**0.06**
ibm_matrix_2	Primal	3961440	3961440	3961440	3959460
	Time(s)	0.08	0.03	0.01	0
	ratio		0.40	0.17	**0.06**
matrix_9	Primal	8377720	8377720	8377720	8368370
	Time(s)	0.17	0.07	0.03	0.01
	ratio		0.39	0.18	**0.05**
barrier2-4	Primal	105044000	105044000	105044000	104597000
	Time(s)	0.54	0.23	0.10	0.04
	ratio		0.42	0.18	**0.08**
av41092	Primal	3156210	3156210	3156210	3155920
	Time(s)	**24.51**	8.09	2.48	**0.11**
	ratio		0.33	0.10	**0.00**
Zhao2	Primal	333891	333891	333891	333487
	Time(s)	7.69	2.37	3.65	0.02
	ratio		0.31	0.47	**0.00**

Table 8.6: (Continued from Table 8.5.) Approximating the maximum weight of a maximal matching affects that weight only slightly for integer problems but can provide dramatic performance improvements. Note that this implementation is slightly different than the one used in Table 8.4, but the slight difference caused a dramatic increase in time required for av41092. Approximating the best matching erased the difference.

8.4 Approximate matchings for static pivoting

Returning to the test bed of Chapter 5, we find no appreciable difference between solution results using the approximation levels in Section 8.3. Systems that fail to converge because of element growth fare no better, and systems that do converge fare no worse. This is somewhat surprising and deserves more validation. As in Tables 8.5 and 8.6, the matching quality differs only in small elements. If those elements are too tiny and are perturbed, they are perturbed in both cases. If not too tiny, then choosing one over the other seems not to make a difference.

8.5 Performance of distributed auctions

The run-time performance of our memory scalable auction algorithm is highly variable. Some matrices and configurations of processors and auction options find great speed-ups, while others find large slow-downs. We have not yet found a good indicator to performance that requires less time than running the auction algorithm.

Our performance results[3] are from a small cluster of dual quad-core X5550 Nehalem processors running at 2.66GHz. The nodes have 24 GiB of memory and are connected by a Mellanox ConnectX DDR InfiniBand with an MPI latency of around 1 μs. We use a subset of the matrices in Section 8.2; we only select matrices with at least 2 400 rows. At 24 processors, there are at least 100 rows allocated to each processor. We use the fastest of two runs for each matrix. The sequential implementation provides the baseline rather than running the distributed implementation on a single processor.

Figure 8.1 shows a widely variable speed-up on our matrix suite across multiple processors. Occasional best cases reach speed-ups of over 1 000×, but the median speed-up begins at 3× with two processors and drops below 1× by 24 processors.

Reporting fair speed-ups for combinatorial algorithms is difficult. The algorithms take different paths to the solution depending on the distribution of the problem. Sequentially running a blocked auction (Section 7.8) with the same distribution may eliminate the "superlinear" speed-ups of over 1 000× on two processors. Problems with trivial solutions still will achieve near-linear speed-ups. The first pass that forms the initial matching is almost perfectly parallel, and bids do not conflict between processors.

Figure 8.2 scales the speed-up more appropriately for our target use in distributed, sparse *LU* factorization. If the input is distributed, a sequential algorithm first must gather all the data onto a single node. Figure 8.3 compares the speed-up of the distributed algorithm to the slow-down from gathering the matrix data.

The frequent communication for non-trivial problems may lead to network latency dominating performance. To demonstrate that the algorithm's performance is not dominated by latency, Figure 8.4 compares different arrangements of processors across nodes. Processes

[3]Source code currently at `http://lovesgoodfood.com/jason/cgit/index.cgi/matchpres/` .

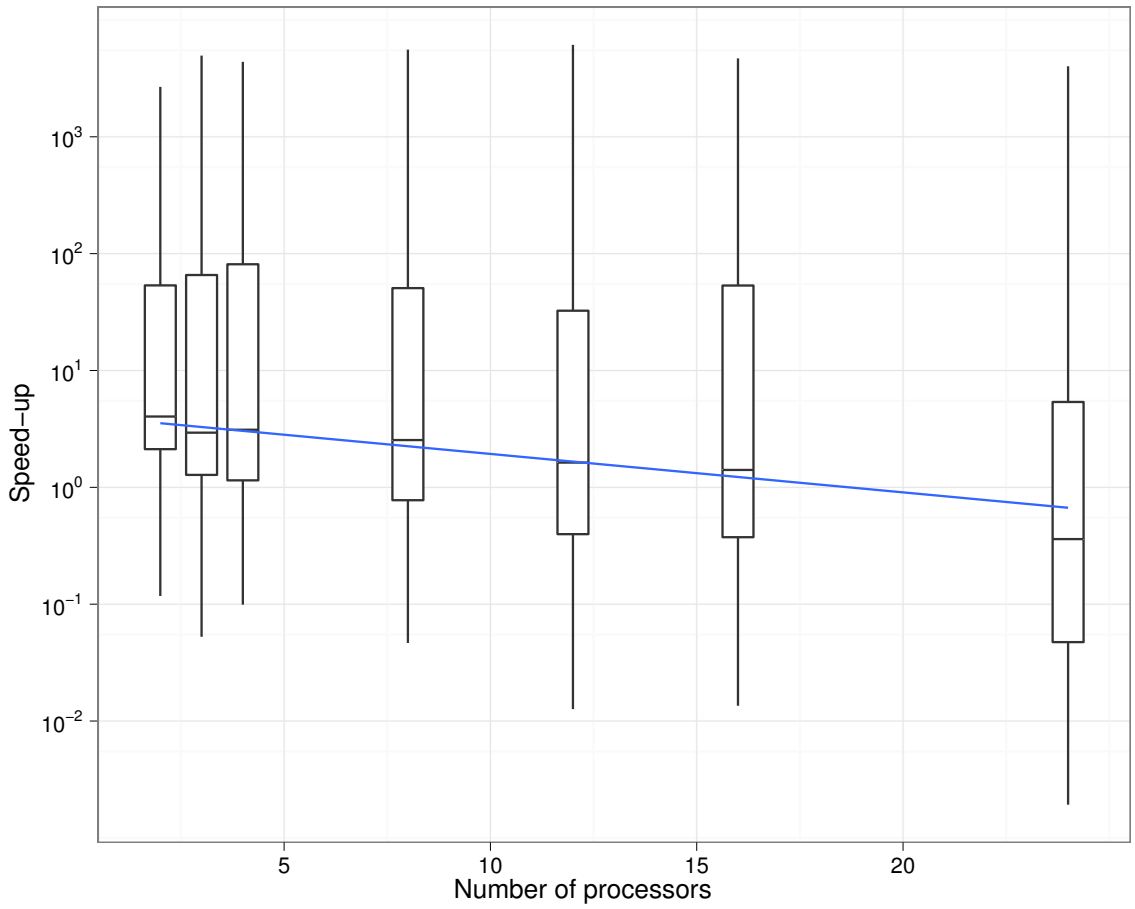


Figure 8.1: This speed-up of a distributed auction over a sequential auction shows a general lack of performance scalability and wide variability in performance. We take the best time of two runs for each sample. Each box covers the 25% to 75% of results, and the bar within the box denotes the median (50%) of results. The whiskers (lines) grow out to sample that is at most twice as far from the median as the box extends. No dots beyond that upward segment means that no sample is more than twice the 50%→75% distance (downward, 50%→25%). The blue diagonal line is a median regression line; 50% of the data points are at worst above the line, and 50% are at best on or below the line.

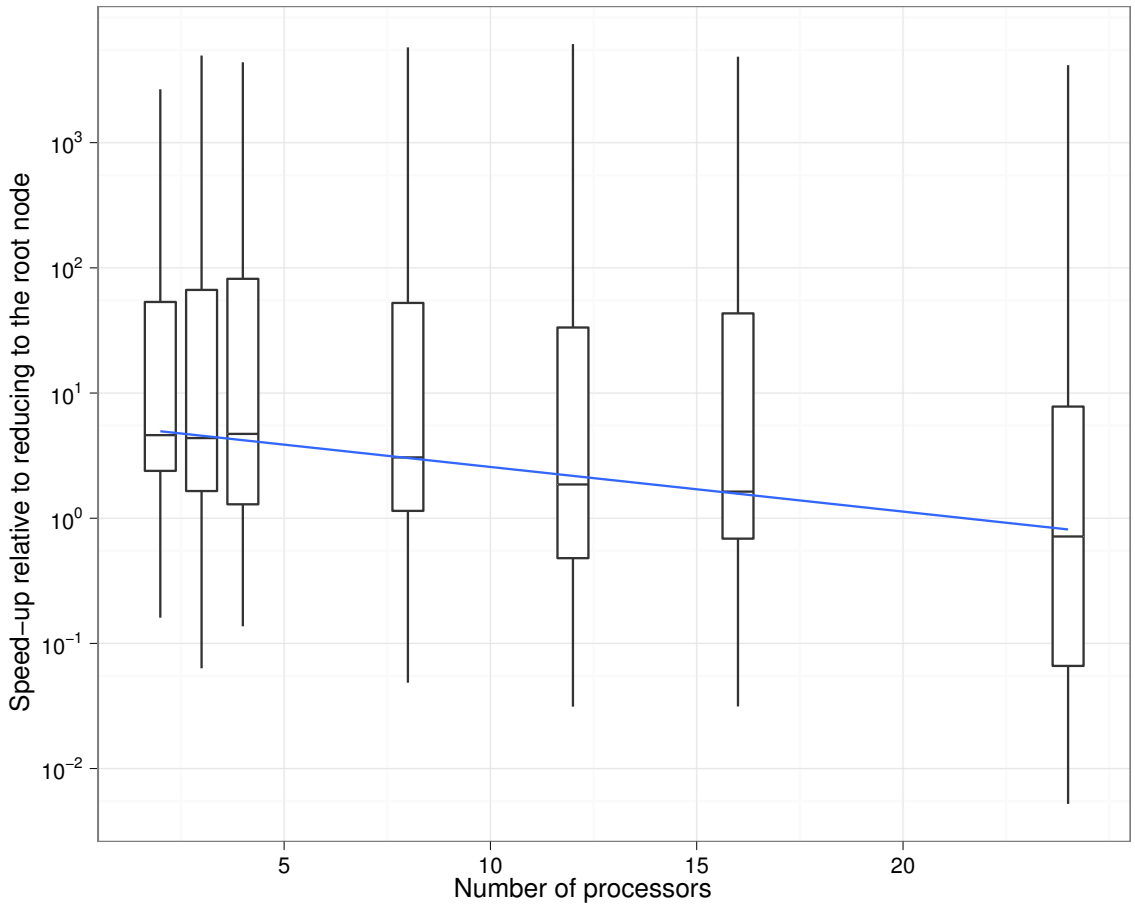


Figure 8.2: This speed-up of a distributed auction over collecting the graph at a single node, running a sequential weight matching algorithm, and distributing the results show slightly better scalability but still wide variability in performance. Comparing performance assuming distributed input data is more fair for our target application, distributed-memory sparse LU factorization. We take the best time of two runs for each sample. The box covers the 25% to 75% of results, and the bar within the box denotes the median (50%) of results. The whiskers (lines) grow out to sample that is at most twice as far from the median as the box extends. No dots beyond that upward segment means that no sample is more than twice the 50%→75% distance (downward, 50%→25%). The blue diagonal line is a median regression line; 50% of the data points are at worst above the line, and 50% are at best on or below the line.

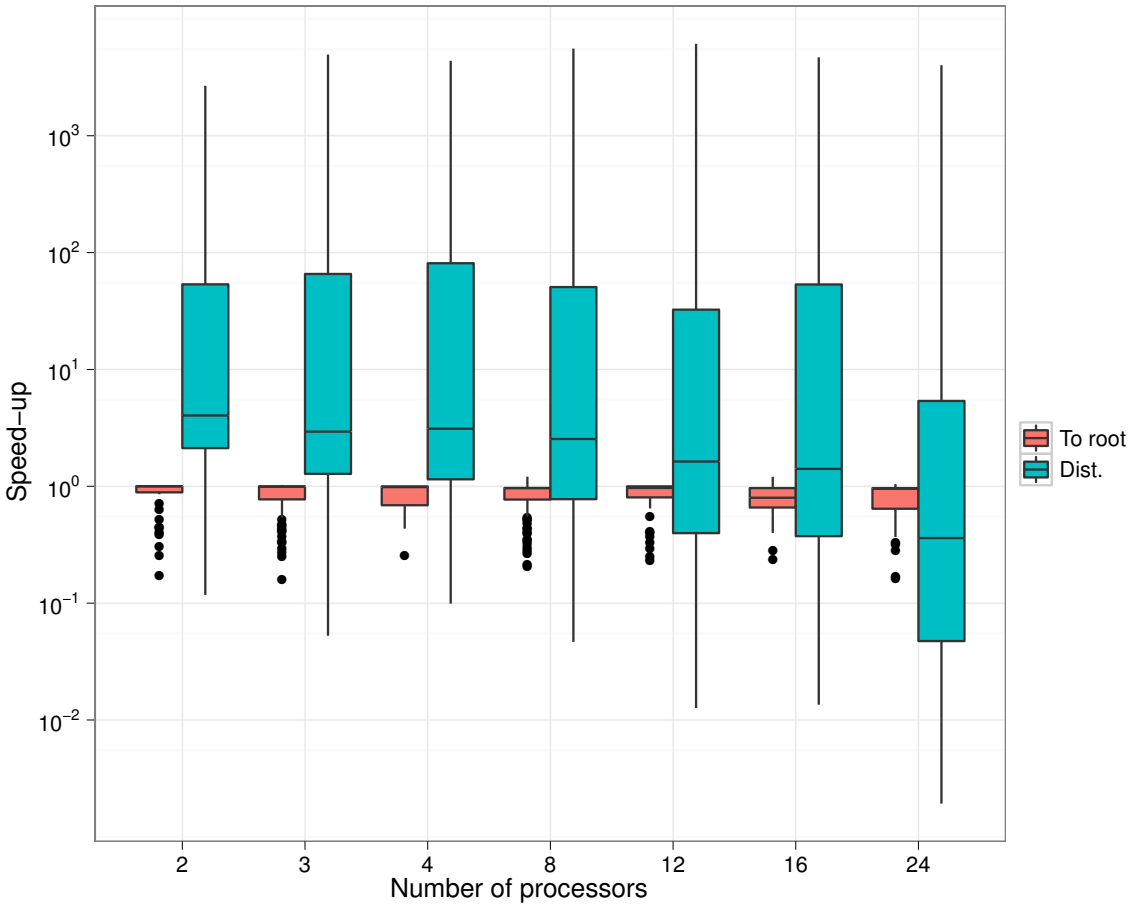


Figure 8.3: Comparison of speed-ups of a distributed auction over collecting the graph at a single node, running a sequential weight matching algorithm, and distributing the results. The overheads in collecting the graph almost always reduces performance. Exceptions are within the level of run-time noise. Comparing performance assuming distributed input data is more fair for our target application, distributed-memory sparse LU factorization. We take the best time of two runs for each sample. Each box covers the 25% to 75% of results for the algorithm in question (gathering to a root or remaining distributed), and the bar within the box denotes the median (50%) of results. The whiskers (lines) grow out to sample that is at most twice as far from the median as the box extends. No dots beyond the upward segment means that no sample is more than twice the 50%→75% distance (downward, 50%→25%). The blue diagonal line is a median regression line; 50% of the data points are at worst above the line, and 50% are at best on or below the line.

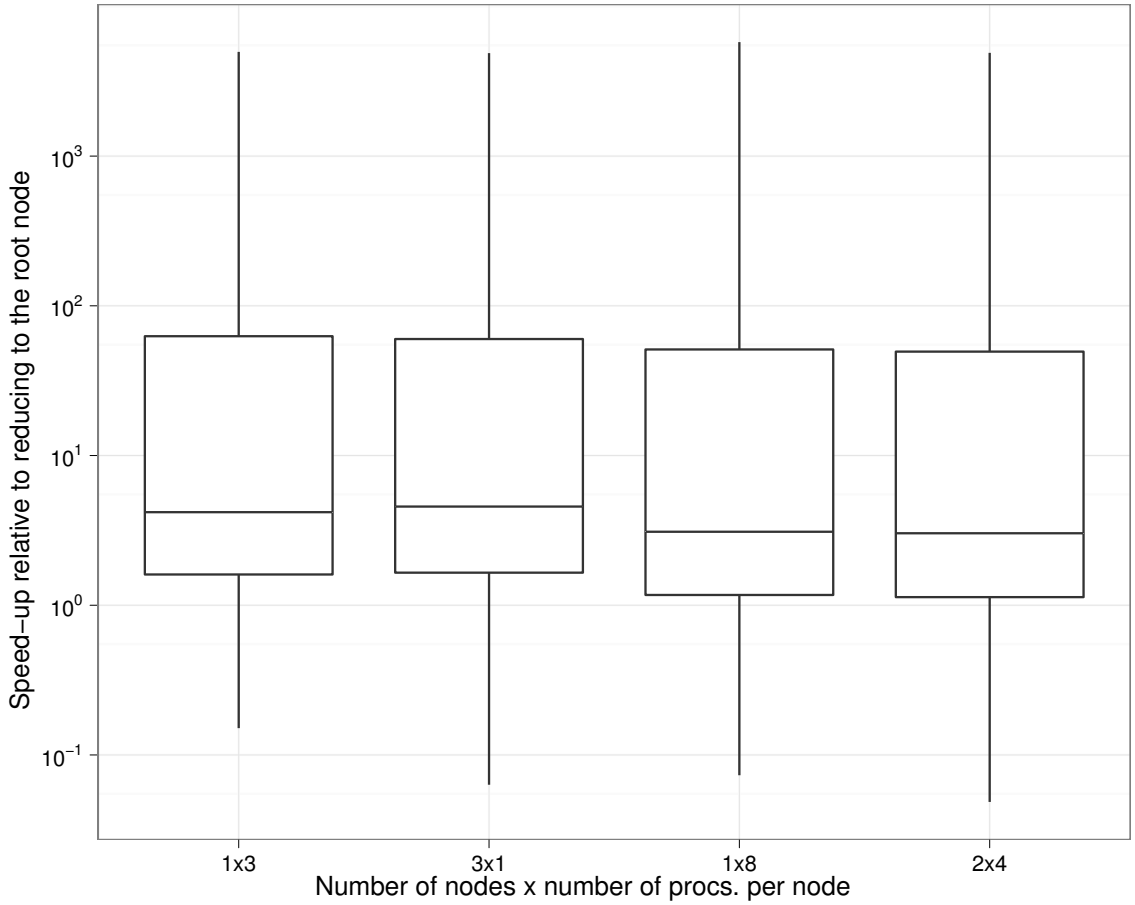


Figure 8.4: The issue is not simply latency. This plot shows "equivalent" processor allocations spread across separate nodes connected by Infiniband. If performance were directly related to the interconnect's latency, the (1x3, 3x1) and (1x8, 2x4) results would be appreciably different. Each of the pairs involves the same number of processing cores. The 3x1 combination involves more memory controllers than the 1x3 combination as well as cross-node communication yet still shows the same performance. With auctions, the performance depends on the number of phases and not on the network's alacrity at resolving the phase.

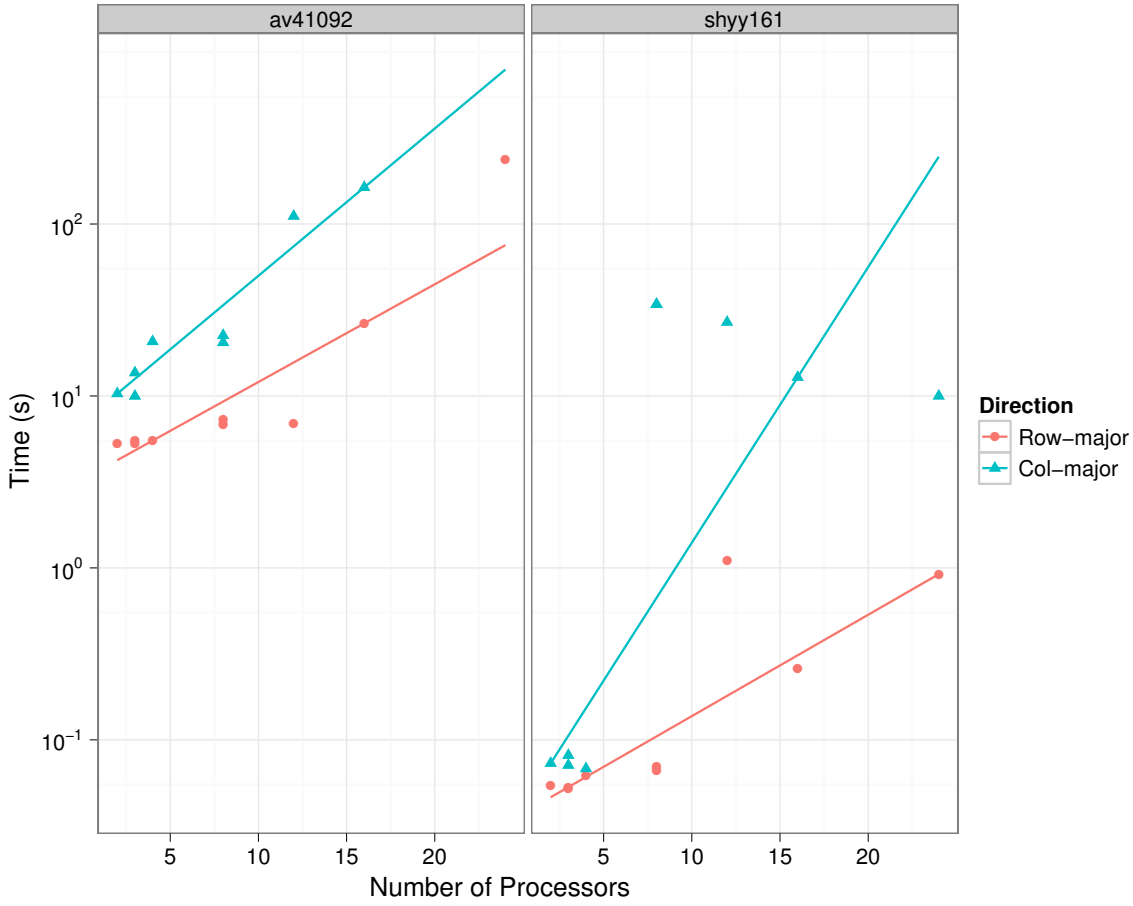


Figure 8.5: Example of two matrices, av41092 (41 092 rows, 1 683 902 entries) and shyy161 (76 480 rows, 329 762 entries) with utterly different performance profiles. The av41092 matrix does *not* scale with respect to performance for our algorithm regardless of the access order, but shyy161 appears "easy" to the row-major order and "hard" to the column-major order. The lines are median regressions ensuring that half the data points are above the line and half below.

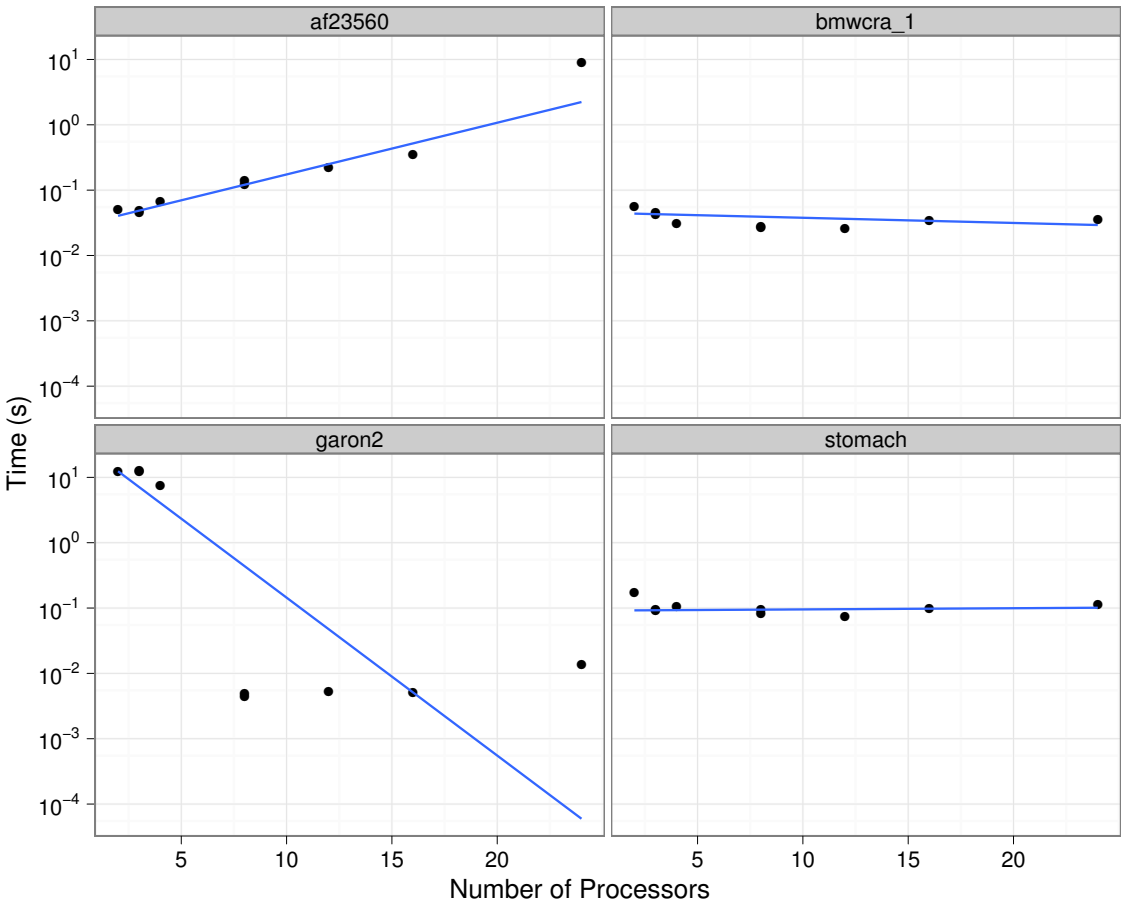


Figure 8.6: The results of Figure 8.5 show matrices that both do not scale, but that is not the only performance mode of our algorithm. Here there are four matrices (af23560, bwmcra_1, garon2, stomach) with three different performance profiles. The matrix af23560 is so "simple" that additional processors simply add overhead. Matrix garon2 shows a sudden drop by happening upon the optimal solution immediately. The other two, bwmcra_1 and stomach, show a slight speed-up from the initial pass followed by a flat performance profile. Ultimately the wide variety of matrices show a wide and unpredictable variety of performance profiles. The lines show a median regression; half the data points are above the line and half below.

within a node communicate with one-tenth the inter-node latency. If latency dominated the performance, the different arrangements in Figure 8.4 would show very different speed-ups. However, there is almost no difference between the arrangements.

Figure 8.5 shows that the performance variability between working with the matrix or its transpose ("row-major") seen sequentially in Section 8.2 complicates parallel performance as well. Figure 8.6 shows how differently a few select matrices perform with increasing processor count. Modeling and predicting performance reasonably remains outside our reach.

Some instances of slow performance appear related to the structure of the matrix's Dulmage-Mendelsohn decomposition[84, 50]. Every edge within a Dulmage-Mendelsohn decomposition's square blocks can be permuted to the diagonal. When the edge weights within that block are similar, many permutations may be tried before finding the optimum matching. The distributed algorithm runs nearly sequentially, and almost every change requires a global communication.

Section 8.4's approximate matching does reduce some of these slow-downs. However, the baseline sequential running time drops even more and makes the speed-up appear worse.

Part III

Summary and Future Directions

Chapter 9

Summary and future directions

9.1 Summary

Essentially, iterative refinement with only a limited amount of extra precision works wonderfully. The solutions not only are accurate but also are dependably labeled as accurate or potentially inaccurate. Our tests with artificial dense systems (Chapter 4) and sparse systems based on practical matrices (Chapter 5) found no failures to identify an erroneous solution. We do not rely on condition estimation to identify success, improving [37] substantially for sparse and distributed-memory use. We are evaluating Chapter 3's algorithm as a replacement for LAPACK's refinement in x`GESVXX`, and future work will compare numerical and timing performance of an LAPACK-style implementation of our new algorithm with LAPACK's.

With extra precision in refinement and an improved, column-relative perturbation scheme, static pivoting achieves dependable solutions. The solutions are accurate even in forward componentwise error unless the element growth is too large (Section 5.4) . Static pivoting appears relatively insensitive to approximating the maximum weight matching (Section 8.4), permitting high-performance sequential pivot selection.

Unfortunately, the parallel performance of our distributed auction algorithm is difficult to characterize. Some combinations of matrices and processor counts achieve abysmal performance. Larger approximation factors assist performance somewhat, but explaining the performance to end users will become a support nightmare.

9.2 Future directions for iterative refinement

Iterative refinement wraps around many different factorization and solution techniques. The normwise stable and asymptotically optimal algorithms for many linear algebra problems[36, 38] can use our refinement algorithm to obtain componentwise accurate results. Iterative refinement is applicable to more than just solving square systems $Ax = b$. Demmel et al. [39] successfully applies the algorithm from Demmel et al. [37] to over-determined least squares

problems. Our algorithm likewise will apply, although choosing exactly which backward errors are important and which quantities need monitored requires more thought.

Because we forgo condition estimation, our refinement algorithm should apply to iterative methods. We only need *some* way to solve $Ady_i = r_i$ at each step. Wrapping our iterative refinement algorithm around GMRES(k) and experimenting with telescoping the backward error within $Ady_i = r_i$ could prove effective and efficient.

The analysis of refinement as Newton's method by Tisseur and Higham (included in Higham [59]) should permit bootstrapping small backward error in other circumstances. So long as the residual can be computed to at least twice working precision, our refinement algorithm should support implicitly defined linear operators.

9.3 Future directions for weighted bipartite matching

The future for distributed weighted bipartite matching algorithms that maintain partial matchings seems bleak. Being restricted to the corners of the matching polytope severely limits the computation's structure. Specialized interior-point methods are most promising. The internal solves can be handled quickly and in little memory using graph-preconditioned iterative methods. However, rounding to a final answer is similar to rebuilding an entire matching[10]. More effective ways to convert fractional matchings or flows to an integral matching remains an interesting open problem.

As a practical alternative, running multiple different algorithms simultaneously on partitions within a larger parallel process and terminating all but the fastest may be useful. For use in static pivoting, the numerical phase requires far more memory and processing, so there may be room to replicate the symbolic work. Engineering such a system is a challenge in the current MPI model.

Bibliography

[1] American Media, Inc. Weekly world news, Possibly defunct. http://weeklyworldnews.com/.

[2] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*. American National Standards Institute, 1999. http://webstore.ansi.org/ansidocstore/product.asp?sku=ANSI%2FISO%2FIEC+9899%2D1999.

[3] Patrick Amestoy, Iain Duff, Jean-Yves L'Excellent, and Xiaoye Li. Analysis and comparison of two general sparse solvers for distributed memory computers. Technical Report LBNL–45992, Lawrence Berkeley National Laboratory, July 2000. http://www.nersc.gov/~xiaoye/TR_slu_mumps.ps.gz.

[4] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. MUMPS: A general purpose distributed memory sparse solver. *Lecture Notes in Computer Science*, 1947, 2001. http://citeseer.nj.nec.com/amestoy00mumps.html.

[5] Patrick R. Amestoy, Iain S. Duff, Daniel Ruiz, and Bora Uçar. Towards parallel bipartite matching algorithms. Presentation at Scheduling for large-scale systems, Knoxville, Tennessee, USA, May 2009.

[6] Edward Anderson, Zhaojun Bai, Christian H. Bischof, L. S. Blackford, James W. Demmel, Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, Anne Greenbaum, A. McKenney, and Danny C. Sorensen. *LAPACK Users' guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, third edition, 1999. ISBN 0-89871-447-8. http://www.netlib.org/lapack/lug/.

[7] Richard J. Anderson and Joo C. Setubal. On the parallel implementation of goldberg's maximum flow algorithm. In *Proceedings of the fourth annual ACM symposium on Parallel Algorithms and Architectures*, pages 168–177, San Diego, California, United States, 1992. ACM Press. ISBN 0-89791-483-X. doi: 10.1145/140901.140919.

[8] Ami Arbel. *Exploring Interior-Point Linear Programming*. Foundations of Computing. The MIT Press, 1993. ISBN 0-262-51073-1.

[9] Mario Arioli, James W. Demmel, and Iain S. Duff. Solving sparse linear systems with sparse backward error. *SIAM Journal on Matrix Analysis and Applications*, 10:165–190, April 1989. doi: 10.1137/0610013.

[10] Sanjeev Arora, Alan Frieze, and Haim Kaplan. A new rounding procedure for the assignment problem with applications to dense graph arrangement problems. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, page 21. IEEE Computer Society, 1996. doi: 10.1109/SFCS.1996.548460.

[11] Nader Bagherzadeh and Kent Hawk. Parallel implementation of the auction algorithm on the intel hypercube. In Viktor K. Prasanna and Larry H. Canter, editors, *IPPS*, pages 443–447. IEEE Computer Society, 1992. ISBN 0-8186-2672-0.

[12] Dimitri Bertsekas. Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications*, 1:7–66, 1992. `http://citeseer.nj.nec.com/bertsekas92auction.html`.

[13] Dimitri Bertsekas. *Network Optimization: Continuous and Discrete Models*. Athena Scientific, 1998. ISBN 1-886529-02-7. `http://www.athenasc.com/netbook.html`.

[14] Dimitri Bertsekas and J. Eckstein. Dual coordinate step methods for linear network flow problems. *Mathematical Programming Series B*, 42:203–243, 1988.

[15] Dimitri P. Bertsekas and David A. Castaon. Parallel synchronous and asynchronous implementations of the auction algorithm. *Parallel Computing*, 17:707–732, September 1991. doi: 10.1016/S0167-8191(05)80062-6.

[16] Garret Birkhoff. Tres observaciones sobre el algebra lineal. *Revista. Serie A: Matematica y fisica teorica*, 5:147–151, 1946. `http://citeseer.nj.nec.com/context/236572/0`.

[17] Åke Björck. Iterative refinement and reliable computing. In M.G. Cox and S.J. Hammarling, editors, *Reliable Numerical Computation*, pages 249–266. Oxford University Press, 1990. ISBN 978-0198535645.

[18] H.J. Bowdler, R.S. Martin, G. Peters, and J.H. Wilkinson. Handbook series linear algebra: Solution of real and complex systems of linear equations. *Numerische Mathematik*, 8:217–234, 1966.

[19] Stephen P. Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, UK, 2004. ISBN 0521833787. `http://www.stanford.edu/~boyd/cvxbook.html`.

[20] Rainer Burkard and Erand Çela. Linear assignment problems and extensions. In Panos M. Pardalos and Ding-Zhu Du, editors, *Handbook of Combinatorial Optimization*

- *Supplement Volume A*, pages 75–149. Kluwer Academic Publishers, October 1999. ISBN 0-7923-5924-0. ftp://ftp.tu-graz.ac.at/pub/papers/math/sfb127.ps.gz.

[21] Rainer E. Burkard. Selected topics on assignment problems. *Discrete Applied Mathematics*, 123:257–302, 2002. doi: 10.1016/S0166-218X(01)00343-2.

[22] Libor Bus and Pavel Tvrdík. Distributed memory auction algorithms for the linear assignment problem. In Selim G. Akl and Teofilo F. Gonzalez, editors, *International Conference on Parallel and Distributed Computing Systems, PDCS 2002*, pages 137–142. IASTED/ACTA Press, November 2002. ISBN 0-88986-366-0.

[23] Orhan Camoglu, Tamer Kahveci, and Ambuj K. Singh. Towards index-based similarity search for protein structure databases. In *CSB*, 2003. http://www.cs.ucsb.edu/~tamer/papers/csb2003.ps.

[24] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R.T. Heaphy, and L.A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007. doi: 10.1109/IPDPS.2007.370258.

[25] Sheung Hun Cheng and Nicholas J. Higham. Implementation for LAPACK of a block algorithm for matrix 1-norm estimation, 2001. http://citeseer.ist.psu.edu/cheng01implementation.html.

[26] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005. http://www.gwu.edu/~upc/publications/LBNL-59208.pdf.

[27] Stefania Corsaro and Marina Marino. Interval linear systems: the state of the art. *Computational Statistics*, 21:365–384, 2006. ISSN 0943-4062. doi: 10.1007/s00180-006-0268-5.

[28] Timothy A. Davis. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30:196–199, June 2004. ISSN 0098-3500. doi: 10.1145/992200.992206. http://doi.acm.org/10.1145/992200.992206.

[29] Timothy A. Davis. The University of Florida sparse matrix collection. (submitted to ACM Transactions on Mathematical Software), October 2010. http://www.cise.ufl.edu/research/sparse/matrices. See also NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 1997.

[30] Timothy A. Davis, John R. Gilbert, Stefan I. Larimore, and Esmond G. Ng. A column approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software.*, 30:353–376, 2004. doi: 10.1145/1024074.1024079.

[31] J. W. Demmel, B. Diament, and G. Malajovich. On the complexity of computing error bounds. *Foundations of Computational Mathematics*, 1(1):101–125, September 2001. doi: 10.1007/s10208001004.

[32] James W. Demmel. Underflow and the reliability of numerical software. *SIAM Journal on Scientific and Statistical Computing*, 5:887–919, December 1984. doi: 10.1137/0905062.

[33] James W. Demmel. The geometry of ill-conditioning. *Journal of Complexity*, 3(2): 201–229, 1987. ISSN 0885-064X. doi: 10.1016/0885-064X(87)90027-6.

[34] James W. Demmel. *Applied numerical linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia, 1997. ISBN 978-0-89871-389-3.

[35] James W. Demmel, Yozo Hida, W. Kahan, Xiaoye S. Li, Sonil Mukherjee, and E. Jason Riedy. Error bounds from extra-precise iterative refinement. LAPACK Working Note 165, Netlib, 2005. `http://www.netlib.org/lapack/lawnspdf/lawn165.pdf`. Also issued as UCB//CSD-05-1414, UT-CS-05-547, and LBNL-56965; expanded from TOMS version.

[36] James W. Demmel, Ioana Dumitriu, and Olga Holtz. Fast linear algebra is stable. *Numerische Mathematik*, 108(1):59–91, November 2006. doi: 10.1007/s00211-007-0114-x.

[37] James W. Demmel, Yozo Hida, W. Kahan, Xiaoye S. Li, Sonil Mukherjee, and E. Jason Riedy. Error bounds from extra-precise iterative refinement. *ACM Transactions on Mathematical Software*, 32:325–351, 2006. ISSN 0098-3500. doi: 10.1145/1141885.1141894.

[38] James W. Demmel, Ioana Dumitriu, Olga Holtz, and Robert Kleinberg. Fast matrix multiplication is stable. *Numerische Mathematik*, 106:199–224, April 2007. doi: 10.1007/s00211-007-0061-6.

[39] James W. Demmel, Yozo Hida, Xiaoye S. Li, and E. Jason Riedy. Extra-precise iterative refinement for overdetermined least squares problems. LAPACK Working Note 188, Netlib, May 2007. `http://www.netlib.org/lapack/lawnspdf/lawn188.pdf`. Also issued as UCB/EECS-2007-77.

[40] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag, Heidelberg, 3rd edition, July 2005. ISBN 3-540-26182-6. `http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/`.

[41] J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, 1979. http://books.google.com/books?id=AmSm1n3Vw0cC.

[42] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988. doi: 10.1145/42288.42291.

[43] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16: 1–17, 1990. doi: 10.1145/77626.79170.

[44] Tobin A. Driscoll and Kara L. Maki. Searching for rare growth factors using multicanonical monte carlo methods. *SIAM Review*, 49(4):673–692, 2007. doi: 10.1137/050637662.

[45] Paul F. Dubois. *Numerical Python*, November 2010. http://numpy.scipy.org/.

[46] Iain S. Duff. Algorithm 575: Permutations for a zero-free diagonal. *ACM Transactions on Mathematical Software*, 7(3):387–390, 1981. doi: 10.1145/355958.355968.

[47] Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001. http://citeseer.nj.nec.com/duff99algorithms.html.

[48] Iain S. Duff and Stéphane Pralet. Towards a stable static pivoting strategy for the sequential and parallel solution of sparse symmetric indefinite systems. Technical Report RAL-TR-2005-007, Numerical Analysis Group, Science and Technology Facilities Council, 2005. http://www.numerical.rl.ac.uk/reports/reports.shtml#2005.

[49] Iain S. Duff, Albert M. Erisman, and John K. Reid. *Direct methods for sparse matrices*. Oxford University Press, Inc, 1986. ISBN 0-19-853408-6.

[50] A. L. Dulmage and N. S. Mendelsohn. Matrices associated with the Hitchcock problem. *Journal of the ACM (JACM)*, 9(4):409–418, 1962. ISSN 0004-5411. doi: 10.1145/321138.321139.

[51] John W. Eaton. *GNU Octave Manual*. Network Theory Limited, 2002. ISBN 0-9541617-2-6. http://www.gnu.org/software/octave/doc/interpreter/.

[52] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, April 1972. ISSN 0004-5411. doi: 10.1145/321694.321699.

[53] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, 1981. http://www.librarything.com/work/1538622.

[54] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins University Press, Baltimore, 3rd ed. edition, 1996. ISBN 978-0-8018-5413-2.

[55] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 155–162, June 2001. ISBN 0-7695-1150-3. doi: 10.1109/ARITH.2001.930115.

[56] Desmond J. Higham and Nicholas J. Higham. Backward error and condition of structured linear systems. *SIAM Journal on Matrix Analysis and Applications*, 13:162–175, 1992. doi: 10.1137/0613014.

[57] Nicholas J. Higham. FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation. *ACM Transactions on Mathematical Software*, 14:381–396, 1988. doi: 10.1145/50063.214386.

[58] Nicholas J. Higham. Iterative refinement for linear systems and LAPACK. *IMA Journal of Numerical Analysis*, 17:495–509, October 1997. doi: 10.1093/imanum/17.4.495.

[59] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM: Society for Industrial and Applied Mathematics, 2nd edition, August 2002. ISBN 0-89871-521-0.

[60] Nicholas J. Higham and Sheung Hun Cheng. Modifying the inertia of matrices arising in optimization. *Linear Algebra and its Applications*, 275-276:261 — 279, 1998. ISSN 0024-3795. doi: 10.1016/S0024-3795(97)10015-5. Proceedings of the Sixth Conference of the International Linear Algebra Society.

[61] Nicholas J. Higham and Desmond J. Higham. Large growth factors in gaussian elimination with pivoting. *SIAM Journal on Matrix Analysis and Applications*, 10(2): 155–164, 1989. doi: 10.1137/0610012.

[62] Nicholas J. Higham and Françoise Tisseur. A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra. *SIAM Journal on Matrix Analysis and Applications*, 21:1185–1201, 2000. doi: 10.1137/S0895479899356080.

[63] J. D. Hogg and J. A. Scott. A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 37:17:1–17:24, April 2010. ISSN 0098-3500. doi: 10.1145/1731022.1731027. `http://doi.acm.org/10.1145/1731022.1731027`.

[64] IEEE Standards Committee 754r. *IEEE Standard for binary floating-point arithmetic, ANSI/IEEE Standard 754r-2008*. Institute of Electrical and Electronics Engineers, New York, 2008. `http://grouper.ieee.org/groups/754/`.

[65] Amazon Incorporated. Elastic compute cloud. `http://aws.amazon.com/ec2/`, 2010. (pricing as of December, 2010).

[66] Andrzej Kiełbasiński. Iterative refinement for linear systems in variable-precision arithmetic. *BIT*, 21:97–103, 1981. doi: 10.1007/BF01934074.

[67] Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955. doi: 10.1002/nav.3800020109.

[68] Johannes Langguth, Md. Mostofa Ali Patwary, and Fredrik Manne. Parallel algorithms for bipartite matching problems on distributed memory computers. (in submission), October 2010.

[69] Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006. doi: 10.1145/1188455.1188573.

[70] Eugene Lawler. *Combinatorial Optimization; Networks and Matroids*. Dover, 1976. `http://www.librarything.com/work/1041237`.

[71] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29:110–140, 2003. doi: 10.1145/779359.779361.

[72] Joseph W. H. Liu. On threshold pivoting in the multifrontal method for sparse indefinite systems. *ACM Trans. Math. Softw.*, 13(3):250–261, 1987. ISSN 0098-3500. doi: 10.1145/29380.31331.

[73] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. management. *Management*, 3:255–269, 1957. doi: 10.1287/mnsc.3.3.255.

[74] The MathWorks™. MATLAB® *7 Programming Fundamentals*. Natick, MA, 2008. `http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/matlab_prog.pdf`.

[75] Message-Passing Interface Forum. *MPI-2.0: Extensions to the Message-Passing Interface*. MPI Forum, June 1997. `http://www.mpi-forum.org/docs/docs.html`.

[76] Cleve B. Moler. Iterative refinement in floating point. *Journal of the Association for Computing Machinery*, 14(2):316–321, 1967. `http://doi.acm.org/10.1145/321386.321394`.

[77] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, March 1957. doi: 10.1137/0105003.

[78] Hong Diep Nguyen and Nathalie Revol. Doubling the precision for the residual and the solution in interval iterative refinement for linear system solving and certifying. In *NSV-3: Third International Workshop on Numerical Software Verification*, July 2010.

[79] UML 1.4. *Unified Modelling Language Specification, version 1.4*. Object Modeling Group, September 2001. `http://www.omg.org/cgi-bin/doc?formal/01-09-67`.

[80] W. Oettli and W. Prager. Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides. *Numerische Mathematik*, 6:405–409, December 1964. doi: 10.1007/BF01386090.

[81] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26:1955–1988, 2005. doi: 10.1137/030601818.

[82] Markus Olschowka and Arnold Neumaier. A new pivoting strategy for gaussian elimination. *Linear Algebra and its Applications*, 240:131–151, June 1996. doi: 10.1016/0024-3795(94)00192-8.

[83] Ali Pinar, Edmond Chow, and Alex Pothen. Combinatorial algorithms for computing column space bases that have sparse inverses. *Electronic Transactions on Numerical Analysis; Special Volume on Saddle Point Problems: Numerical Solution and Applications*, 22, 2006. `http://etna.mcs.kent.edu/vol.22.2006/index.html`.

[84] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16:303–324, 1990. doi: 10.1145/98267.98287.

[85] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2010. `http://www.R-project.org/`. ISBN 3-900051-07-0.

[86] E. Jason Riedy. Making static pivoting dependable, 2006. `http://purl.oclc.org/NET/jason-riedy/resume/material/bascd2006-poster.pdf`. Seventh Bay Area Scientific Computing Day.

[87] E. Jason Riedy. Auctions for distributed (and possibly parallel) matchings. Visit to CERFACS courtesy of the Franco-Berkeley Fund, December 2008. `http://purl.oclc.org/NET/jason-riedy/resume/material/cerfacs08.pdf`.

[88] J. L. Rigal and J. Gaches. On the compatibility of a given solution with the data of a linear system. *Journal of the ACM*, 14:543–548, 1967. doi: 10.1145/321406.321416.

[89] J. Rohn. Enclosing solutions of linear interval equations is NP-hard. *Computing*, 53: 365–368, 1994. ISSN 0010-485X. doi: 10.1007/BF02307386.

[90] Daniel Ruiz. A scaling algorithm to equilibrate both row and column norms in matrices. Technical report, Rutherford Appleton Laboratory, September 2001. `ftp://ftp.numerical.rl.ac.uk/pub/reports/drRAL2001034.pdf`.

[91] Siegfried M. Rump. A class of arbitrarily ill conditioned floating-point matrices. *SIAM Journal on Matrix Analysis and Applications*, 12(4):645–653, October 1991. doi: 10.1137/0612049.

[92] S.M. Rump. Solving algebraic problems with high accuracy. In U.W. Kulisch and W.L. Miranker, editors, *A New Approach to Scientific Computation*, pages 51–120. Academic Press, 1983.

[93] S.M. Rump. Verified computation of the solution of large sparse linear systems. *Zeitschrift für Angewandte Mathematik und Mechanik (ZAMM)*, 75:S439–S442, 1995.

[94] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. Report on the programming language X10; version 2.1. Technical report, IBM Research, October 2010.

[95] Robert B. Schnabel and Elizabeth Eskow. A revised modified cholesky factorization algorithm. *SIAM Journal on Optimization*, 9(4):1135–1148, 1999. ISSN 1052-6234. doi: 10.1137/S105262349833266X.

[96] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, New York, 1973. ISBN 0-89871-355-2. xiii+441 pp.

[97] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13: 354–356, 1969. ISSN 0029-599X. doi: 10.1007/BF02165411.

[98] Sivan Toledo and Anatoli Uchitel. A supernodal out-of-core sparse gaussian-elimination method. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics, 7th International Conference (7th PPAM'07)*, volume 4967 of *Lecture Notes in Computer Science (LNCS)*, pages 728–737, Gdansk, Poland, September 2007, Revised Selected Paper 2008. Springer-Verlag (New York).

[99] Guido van Rossum and Fred L. Drake. *The Python Reference Manual*, 2.3 edition, July 2003. `http://www.python.org/doc/2.3/`.

[100] John von Neumann. A certain zero-sum two-person game equivalent to the optimal assignment problem. In Harold W. Kuhn, editor, *Contributions to the Theory of*

Games, volume 2, pages 5–12. Princeton University Press, Princeton, New Jersey, 1953. `http://citeseer.nj.nec.com/context/1145728/0`.

[101] Joel M. Wein and Stavros A. Zenios. Massively parallel auction algorithms for the assignment problem. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, pages 90–99. IEEE Computer Society, 1990. doi: 10.1109/FMPC.1990.89444.

[102] Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009. ISBN 978-0-387-98140-6. `http://had.co.nz/ggplot2/book`.

[103] James Hardy Wilkinson. *Rounding Errors in Algebraic Processes*. Notes on Applied Science No. 32, Her Majesty's Stationery Office, London, 1963. ISBN 0-486-67999-3. Also published by Prentice-Hall, Englewood Cliffs, NJ, USA. Reprinted by Dover, New York, 1994.

[104] Hossam A. Zaki. A comparison of two algorithms for the assignment problem. *Computational Optimization and Applications*, 4:23–45, 1995. doi: 10.1007/BF01299157.