Avoiding breakdown in incomplete factorizations in low precision arithmetic

Avoiding Breakdown in Incomplete Factorizations in Low Precision Arithmetic

JENNIFER SCOTT, STFC Rutherford Appleton Laboratory, Oxfordshire, UK and University of Reading, Reading, UK
MIROSLAV TŮMA, Charles University, Prague, Czech Republic

The emergence of low precision floating-point arithmetic in computer hardware has led to a resurgence of interest in the use of mixed precision numerical linear algebra. For linear systems of equations, there has been renewed enthusiasm for mixed precision variants of iterative refinement. We consider the iterative solution of large sparse systems using incomplete factorization preconditioners. The focus is on the robust computation of such preconditioners in half precision arithmetic and employing them to solve symmetric positive definite systems to higher precision accuracy; however, the proposed ideas can be applied more generally. Even for well-conditioned problems, incomplete factorizations can break down when small entries occur on the diagonal during the factorization. When using half precision arithmetic, overflows are an additional possible source of breakdown. We examine how breakdowns can be avoided and implement our strategies within new half precision Fortran sparse incomplete Cholesky factorization software. Results are reported for a range of problems from practical applications. These demonstrate that, even for highly ill-conditioned problems, half precision preconditioners can potentially replace double precision preconditioners, although unsurprisingly this may be at the cost of additional iterations of a Krylov solver.

CCS Concepts: • **Mathematics of computing** → **Solvers**; **Mathematical software performance**; **Computations on matrices**;

Additional Key Words and Phrases: Sparse matrices, sparse linear systems, preconditioning, incomplete factorizations, half precision arithmetic, mixed precision arithmetic, iterative refinement

ACM Reference format:
Jennifer Scott and Miroslav Tůma. 2024. Avoiding Breakdown in Incomplete Factorizations in Low Precision Arithmetic. *ACM Trans. Math. Softw.* 50, 2, Article 9 (June 2024), 25 pages.
https://doi.org/10.1145/3651155

1 Introduction

We are interested in solving sparse linear systems $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is nonsingular and $x, b \in \mathbb{R}^n$. The majority of algorithms for solving such systems fall into two main categories: direct methods and iterative methods. Direct methods transform A using a finite sequence of elementary transformations into a product of simpler sparse matrices in such a way that solving linear systems of equations with the factor matrices is comparatively easy and relatively inexpensive. For example,

Jennifer Scott and Miroslav Tůma contributed equally to this research.

Authors' Contact Information: Jennifer Scott (Corresponding author), STFC Rutherford Appleton Laboratory, Harwell Campus, Didcot, Oxfordshire, UK and University of Reading, Reading, UK; e-mail: jennifer.scott@stfc.ac.uk; Miroslav Tůma, Department of Numerical Mathematics, Faculty of Mathematics and Physics, Charles University, Sokolovská, Prague, Czech Republic; e-mail: mirektuma@karlin.mff.cuni.cz.

ACM 0098-3500/2024/6-ART9
https://doi.org/10.1145/3651155

for a general nonsymmetric matrix, $A = PLUQ$, where L is a lower triangular matrix, U is an upper triangular matrix, and P and Q are permutation matrices chosen to preserve sparsity in the factors and ensure the factorization is stable. Direct methods, when properly implemented, are robust, frequently very fast, and can be confidently used as black-box solvers for computing solutions with predictable accuracy. However, they require significant expertise to implement efficiently (particularly in parallel). They can also need large amounts of memory (which increases nonlinearly with the size and density of A) and the matrix factors normally contain many more nonzero entries than A; these extra entries are termed the fill-in and much effort goes into trying to minimize the amount of fill-in.

By contrast, iterative methods compute a sequence of approximations $x^{(0)}, x^{(1)}, x^{(2)}, \ldots$ that (hopefully) converge to the solution in an acceptable number of iterations. The number of iterations (and whether or not convergence occurs at all) depends on $x^{(0)}$, A and b as well as the required accuracy in x. Basic implementations of iterative solvers are relatively straightforward as they only use the sparse matrix A indirectly, through matrix–vector products and, most importantly, their memory demands are limited to a (small) number of vectors of length n, making them attractive for very large problems as well as for problems where A is not available explicitly. However, preconditioning is usually essential to enhance the convergence of the iterative method. Preconditioning seeks to transform the system into one that is more tractable and from which the required solution of the original system can easily be recovered. Determining and computing effective preconditioners is highly problem dependent and generally very challenging. Algebraic preconditioners that are built using an incomplete factorization of A in which entries that would be part of a complete factorization are dropped are frequently used, especially when the underlying physics of the problem is difficult to exploit. Such preconditioners can be employed within more sophisticated methods; for example, to precondition subdomain solves in domain decomposition schemes or as smoothers in multigrid methods.

The performance differences for computing and communicating in different precision formats have led to a long history of efforts to enhance numerical algorithms by combining precision formats. The goals of mixed-precision algorithms include accelerating the computational time by including the use of lower-precision formats while obtaining high-precision accuracy of the output and, by reducing the memory requirements, extending the size of problems that can be solved as well as potentially lowering energy usage. Numerical linear algebra software, and linear system solvers in particular, traditionally uses double precision (64-bit) arithmetic, although some packages (including the BLAS and LAPACK routines and some sparse solvers, such as those in the HSL mathematical software library [27]) have always also offered single precision (32-bit) versions. In the mid-2000s, the speed difference between single and double precision arithmetic on what were then state-of-the-art architectures, notably Sony/Toshiba/IBM Cell processors (see, e.g., [10, 32, 33]), led to studies into the feasibility of factorizing a matrix in single precision and then using the factors as a preconditioner for a simple iterative method to regain higher precision accuracy [5, 11, 12]. Hogg and Scott [26] extended this to develop a mixed precision sparse symmetric indefinite solver. More recently, the potential for employing single precision arithmetic in solving sparse systems to double precision accuracy using multiple cores has been considered by Zounon et al. [49]. For such systems, the integer data used to hold the sparse structures of the matrix and its factors are independent of the precision, and memory savings come only from the real factor data. This offers the potential to increase the size of problem that can be tackled and, in the incomplete factorization case, to allow more entries to be retained in the factors, which can result in a higher quality preconditioner, leading to savings in the total solution time.

In the past few years, the emergence of lower precision arithmetic in hardware has led to further interest in mixed precision algorithms. A key difference compared to earlier work is the use of half precision (16-bit) arithmetic. A comprehensive state-of-the-art survey of work on mixed precision numerical linear algebra routines (including an extensive bibliography) is given in [22] (see also [1]). In particular, there have been important ideas and theory on mixed precision iterative refinement methods that employ the matrix factors computed in low precision as a preconditioner to recover higher precision accuracy [2, 14, 15]. The number of entries in the factors of a sparse matrix A is typically much greater than in A and so, unlike in the dense case, the overhead of keeping a high precision copy of A for computing the residual in the refinement process is small. Amestoy et al. [3] investigate the potential of mixed precision iterative refinement to enhance methods for sparse systems based on a particular class of approximate sparse factorizations. They employ the well-known parallel sparse direct solver MUMPS [4], which is able to exploit block low-rank factorizations and static pivoting to compute approximate factors. The reported results in [3] are restricted to combining single and double precision arithmetic because, in common with all other currently available sparse direct solvers, MUMPS does not support the use of half precision arithmetic. Developing an efficient half precision solver is not straightforward, requiring 16-bit versions of the dense linear algebra routines that provide the building blocks behind direct solvers. Such routines are becoming available (e.g., dense matrix–matrix multiplication in half precision is now supported by NVIDIA's cuBLAS library), making the future development of efficient software potentially more feasible.

Higham and Pranesh [24] focus on symmetric positive definite (SPD) linear systems. They compute a Cholesky factorization using low precision arithmetic and employ the factors as preconditioners in GMRES- and conjugate gradient (CG)-based iterative refinement. While they are interested in the sparse case, their MATLAB experiments (which simulate low precision using their chop function [23]) store the sparse test examples as dense matrices and their Cholesky factorizations are computed using dense routines. The reported theoretical and numerical results demonstrate the potential for low precision (complete) factors to be used to obtain high precision accuracy. Most recently, Carson and Khan [16] have considered using sparse approximate inverse (SPAI) preconditioners that are based on Frobenius norm minimization [20]. They again propose computing the preconditioner using low precision and then employing GMRES-based iterative refinement and report MATLAB experiments using the chop function. Mixed precision has also been investigated for multigrid methods (see, e.g., the error analysis of McCormick et al. [39], who observed that different levels in the grid hierarchy should use different precisions).

Our emphasis is on low precision incomplete factorization preconditioners, combined with Krylov subspace-based iterative refinement. Although our ideas can be used for general sparse linear systems, we focus on the sparse SPD case. We use half precision arithmetic to construct **incomplete Cholesky (IC)** factorizations that are then employed as preconditioners to recover double precision accuracy. Our primary objective is to show that for a range of problems (some of which are highly ill-conditioned) it is possible to successfully obtain and use low precision incomplete factors. We consider the potential sources of overflow during the incomplete factorization and look at how to safely detect and prevent overflow. We follow a number of others working on the development of numerical linear algebra algorithms in mixed precision in performing experiments that aim to explore the feasibility of the ideas by simulating half precision (see, e.g., [13, 16, 23]). We want the option to experiment with sparse problems that may be too large for MATLAB and have chosen to develop our software in Fortran. Half precision and double precision versions are tested on systems coming from practical applications.

This paper makes the following contributions:

(a) It considers the practicalities of computing classical incomplete factorizations using low precision arithmetic, in particular, the safe prediction of possible overflows.

Table 1. Parameters for bfloat16, fp16, fp32, and fp64 Arithmetic: The Number of Bits in the Significand and Exponent, Unit Roundoff u, Smallest Positive (Subnormal) Number x^s_{min}, Smallest Normalized Positive Number x_{min}, and Largest Finite Number x_{max}, All Given to Three Significant Figures

	Signif.	Exp.	u	x^s_{min}	x_{min}	x_{max}
bfloat16	8	8	3.91×10^{-3}	[a]	1.18×10^{-38}	3.39×10^{38}
fp16	11	5	4.88×10^{-4}	5.96×10^{-8}	6.10×10^{-5}	6.55×10^{4}
fp32	24	8	5.96×10^{-8}	1.40×10^{-45}	1.18×10^{-38}	3.40×10^{38}
fp64	53	11	1.11×10^{-16}	4.94×10^{-324}	2.22×10^{-308}	1.80×10^{308}

[a]In Intel's bfloat16 specification, subnormal numbers are not supported.

(b) It successfully employs global modifications combined with a simple prescaling to prevent breakdowns during the factorization.
(c) It develops level-based IC factorization software in half precision (written in Fortran).
(d) It demonstrates the potential for robust half precision incomplete factorization preconditioners to be used to solve ill-conditioned SPD systems.

The rest of the paper is organized as follows. In Section 2, we briefly recall incomplete factorizations of sparse matrices and consider the challenges that IC factorizations can face, particularly when low precision arithmetic is employed. In Section 3, we summarize basic mixed precision iterative refinement algorithms before presenting numerical results for a range of problems coming from practical applications in Section 4. Concluding remarks and possible future directions are given in Section 5.

Terminology. We use the term high precision for precision formats that provide high accuracy at the cost of a larger memory volume (in terms of bits) and low precision to refer to precision formats that comprise fewer bits (smaller memory volume) and provide low(er) accuracy. Unless stated otherwise, we mean IEEE double precision (64-bit) when using the term high precision (denoted by fp64) and the 1985 IEEE standard 754 half precision (16-bit) when using the term low precision (denoted by fp16). bfloat16 is another form of half-precision arithmetic that was introduced by Google in its tensor processing units and formalized by Intel; we do not use it in this paper. Table 1 summarizes the parameters for different precision arithmetic. We use u_{16}, u_{32}, and u_{64} to denote the unit roundoffs in fp16, fp32, and fp64 arithmetic, respectively.

2 Incomplete Factorizations

In this section, we briefly recall incomplete factorizations of sparse matrices and then discuss how breakdown can happen, particularly when using low precision.

2.1 A Brief Introduction to Incomplete Factorizations

For a general sparse matrix A, the incomplete factorizations that we are interested in are of the form $A \approx LU$, where L and U are sparse lower and upper triangular matrices, respectively (for simplicity of notation, the permutations P and Q that are for preserving sparsity in the factors are omitted). The computed incomplete factors are used to define the preconditioner; the preconditioned linear system is $U^{-1}L^{-1}A = U^{-1}L^{-1}b$. If A is an SPD matrix then $A \approx LL^T$. There are three main classes of incomplete factorization preconditioners. Firstly, threshold-based $ILU(\tau)$ methods in which the locations of permissible fill-in in the factors are determined in conjunction with the numerical factorization of A; entries of the computed factors that are smaller than a prescribed

ALGORITHM 1. **Basic right-looking IC factorization**
Input: *SPD matrix A and a target sparsity pattern $\mathcal{S}\{L\}$*
Output: *Incomplete Cholesky factorization $A \approx LL^T$.*

```
1: Initialize $l_{ij} = a_{ij}$ for all $(i, j) \in \mathcal{S}\{L\}$
2: for $k = 1 : n$ do                                              ▷ Start of k-th major step
3:     $l_{kk} \leftarrow (l_{kk})^{1/2}$                                   ▷ Diagonal entry is the pivot
4:     for $i = k + 1 : n$ such that $(i, j) \in \mathcal{S}\{L\}$ do
5:         $l_{ik} \leftarrow l_{ik}/l_{kk}$            ▷ Scale pivot column k of the incomplete factor by the pivot
6:     end for                                                    ▷ Column k of L has been computed
7:     for $j = k + 1 : n$ such that $(j, k) \in \mathcal{S}\{L\}$ do
8:         for $i = j : n$ such that $(i, j) \in \mathcal{S}\{L\}$ do
9:             $l_{ij} \leftarrow l_{ij} - l_{ik}l_{jk}$                         ▷ Update operation on column j
10:        end for
11:    end for
12: end for
```

threshold are dropped, for example, [41]. Secondly, memory-based $ILU(m)$ methods in which the amount of memory available for the incomplete factorization is prescribed and only the largest entries are retained at each stage of the factorization, for example, [29]. Thirdly, structure-based $ILU(\ell)$ methods in which an initial symbolic phase uses a hierarchy of sparsity structures and determines the location of permissible entries using only the sparsity pattern of A, for example, [48]. The memory requirements for the incomplete factors are then determined before the numerical factorization is performed. The simplest such approach (for which the symbolic phase is trivial) is an $ILU(0)$ factorization (or $IC(0)$ in the SPD case) that limits entries in the incomplete factors to positions corresponding to entries in A (no fill-in is permitted). $ILU(0)$ preconditioners are frequently used for comparison purposes when assessing the performance of other approaches.

The different approaches have been developed, modified, and refined over many years. Variants have been proposed that combine the ideas and/or employ them in conjunction with discarding entries in A (sparsification) before the factorization commences. For an introduction to ILU-based preconditioners and details of possible variants, we recommend [17] and Chapter 10 of [45], while [46] provides a brief historical overview and further references.

Algorithm 1 outlines a basic (right-looking) IC factorization of an SPD matrix $A = \{a_{ij}\}$. Here all computations are performed in the same precision as A. The algorithm assumes a target sparsity pattern $\mathcal{S}\{L\}$ for $L = \{l_{ij}\}$ is provided, where

$$\mathcal{S}\{L\} = \{(i, j) \mid l_{ij} \neq 0,\ 1 \leq j \leq i \leq n\}.$$

It is assumed that $\mathcal{S}\{L\}$ includes the diagonal entries. Modifications can be made to incorporate threshold dropping strategies and to determine $\mathcal{S}\{L\}$ as the method proceeds. At each major step k, outer product updates are applied to the part of the matrix that has not yet been factored (Lines 7–11).

2.2 Breakdown during Incomplete Factorizations

For arbitrary choices of the sparsity pattern $\mathcal{S}\{L\}$, the IC factorization exists if A is an M-matrix or an H-matrix with positive diagonal entries [38, 40]. But for a general SPD matrix, there is no such guarantee and an incomplete factorization algorithm can (and frequently does) break down.

ALGORITHM 2. **Safe test for a safe update operation (detecting B3 breakdown)**
Input: *Scalars a, b, c such that $|a|, |b|, |c| \leq x_{max}$.*
Output: *Either $v = a - bc$ or $flag = -3$ (unsafe to perform update).*

```
1:  $flag = 0$
2:  **if** $|b| \leq 1$ or $|c| \leq 1$ **then**          ▷ $x_{max}/|c|$ cannot be tested if $|c| < 1$
3:      $w = bc$
4:  **else if** $|b| \leq x_{max}/|c|$ **then**
5:      $w = bc$
6:  **else**
7:      $flag = -3$ and **return**          ▷ Unsafe to compute bc
8:  **end if**
9:  **if** $a \geq 0$ **then**
10:     **if** $w \geq 0$ or $(x_{max} - a \geq -w)$ **then**
11:         $v = a - w$
12:     **else**
13:         $flag = -3$ and **return**          ▷ Unsafe to perform subtraction
14:     **end if**
15: **else**
16:     **if** $w < 0$ or $(x_{max} + a \geq w)$ **then**
17:         $v = a - w$
18:     **else**
19:         $flag = -3$ and **return**          ▷ Unsafe to perform subtraction
20:     **end if**
21: **end if**
```

There are three places where breakdown can occur. We refer to these as problems B1, B2, and B3 as follows:

— B1: The computed diagonal entry l_{kk} (which is termed the pivot at step k) may be unacceptably small or negative.
— B2: The scaling $l_{ik} \leftarrow l_{ik}/l_{kk}$ may overflow.
— B3: The update $l_{ij} \leftarrow l_{ij} - l_{ik}l_{jk}$ may overflow.

It is crucial for a robust implementation to detect the possibility of overflow before it occurs (otherwise, the code will simply crash). Tests for potential breakdown must themselves only use operations that cannot overflow. We say that an operation is *safe* in the precision being used if it cannot overflow; otherwise, it is *unsafe*.

Safe detection of problem B1 is straightforward as we simply need to check at Line 3 of Algorithm 1 that the pivot satisfies $l_{kk} \geq \tau > x_{min}$, where the threshold parameter τ depends on the precision used. Based on our experience with practical problems, in our reported results in Section 4, we use $\tau = 10^{-5}$ for half precision factorizations and, for double precision factorizations, $\tau = 10^{-20}$ (note that preconditioner quality is not sensitive to the precise choice of τ).

Problem B2 can happen during the column scaling at Line 5 of Algorithm 1. Let a be the entry of largest absolute value in the current pivot column k and let d denote the current pivot l_{kk}

ALGORITHM 3. **Basic IC factorization with safe checks for breakdown**
Input: *SPD matrix A, a target sparsity pattern $\mathcal{S}\{L\}$ and parameter $\tau > 0$ for testing small entries.*
Output: *Either $flag < 0$ (breakdown occurred) or $A \approx LL^T$ with L lower triangular*

```
1: Initialize $l_{ij} = a_{ij}$ for all $(i, j) \in \mathcal{S}\{L\}$
2: Set $flag = 0$
3: for $k = 1 : n$ do                                              ▷ Start of $k$-th major step
4:     if $l_{kk} < \tau$ then
5:         Set $flag = -1$ and return                                        ▷ B1 breakdown
6:     end if
7:     $l_{kk} \leftarrow (l_{kk})^{1/2}$
8:     $a = \max_{i=k+1:n}\{|l_{ik}| : (i, k) \in \mathcal{S}\{L\}\}$
9:     if $l_{kk} \geq 1$ or $l_{kk} \geq a/x_{max}$ then  ▷ Note that if $l_{kk} \geq 1$ then $a$ does not need to be computed
10:        for $i = k + 1 : n$ such that $(i, k) \in \mathcal{S}\{L\}$ do
11:            $l_{ik} \leftarrow l_{ik}/l_{kk}$                                 ▷ Perform safe scaling
12:        end for                                           ▷ Column $k$ of $L$ has been computed
13:    else
14:        Set $flag = -2$ and return                                        ▷ B2 breakdown
15:    end if
16:    for $j = k + 1 : n$ such that $(j, k) \in \mathcal{S}\{L\}$ do
17:        Use Algorithm 2 to test for safe update
18:        if $flag = -3$ then
19:            return                                                        ▷ B3 breakdown
20:        end if
21:        for $i = j : n$ such that $(i, j) \in \mathcal{S}\{L\}$ do
22:            $l_{ij} \leftarrow l_{ij} - l_{ik}l_{jk}$                 ▷ Perform safe update operation
23:        end for
24:    end for                                               ▷ Column $j$ of $L$ has been updated
25: end for
```

(set at Line 3). If $a \leq x_{max}$ and $1 \leq d \leq x_{max}$ then $d \geq a/x_{max}$ and it is safe to compute a/d (and thus safe to scale column k).

Problem B3 can occur during the update operations at Line 9 of Algorithm 1. Algorithm 2 can be used to perform safe updates. Here the scalars and the computation are in the working precision.

Algorithm 3 presents the basic IC factorization algorithm with the inclusion of safe checks for possible breakdown. Here the matrix A and the computation are in the working precision. In practice, when using single and double precision the tests in Lines 8–11 and 16–20 are omitted, but B1 can occur when using any arithmetic. Note that the safe test for B3 breakdown does not have to be applied to individual scalar entries but can be applied to each column $j \geq k + 1$.

Observe that the occurrence of underflows when using fp16 arithmetic does not prevent the computation of the incomplete factors, although underflows could lead to a loss of information that affects the preconditioner quality. However, provided the problem has been well scaled, the dropping strategy used within the incomplete factorization normally has more influence on the computed factors than underflows do. A subnormal floating-point number is a nonzero number with absolute value less than that of the smallest normalized number. Floating-point operations on subnormals can be very slow because they often require extra clock cycles, which introduces a high overhead. If an off-diagonal factor entry is subnormal, it can again be replaced by zero without significantly affecting the preconditioner quality.

2.3 Prescaling of the System Matrix

Sparse direct solvers typically prescale A and factorize $\widehat{A} = S_1^{-1} A S_2^{-1}$, where S_1 and S_2 are diagonal scaling matrices ($S_1 = S_2 = S$ in the symmetric case). Because no single choice of scaling always results in the best performance (in terms of the time, factor sizes, memory requirements, and data movement), several possibilities (with different associated costs) are typically offered to allow a user to experiment and select the optimum for their application. For incomplete factorizations, prescaling can reduce the incidence of breakdowns. This is illustrated for SPD matrices in [47], where experiments (in double precision arithmetic) show that the cheap scaling in which the entries in column j of A are normalized by the 2-norm of column j (so that the absolute values of the entries of the scaled matrix are all less than 1) is generally a good choice. However, scaling alone cannot guarantee to prevent breakdown. If breakdown does happen then modifications need to be made to the scaled matrix that is being factorized, either before or during the factorization; this is discussed in the next subsection.

In their work on matrix factorizations in low precision, Higham et al. [25] prescale A using the working precision and then round to fp16 once all the absolute values of all the entries of the matrix are at most x_{max} (see also [22, 24]). They employ an equilibration approach [31] and then multiply each entry of the scaled matrix by $\mu = \theta x_{max}/\gamma$, where γ is the entry of largest absolute value in the scaled matrix $\widehat{A}$ and $\theta \in (0, 1)$ is a parameter chosen with the objective of limiting the possibility of overflow during the factorization. Our experiments have found, in the incomplete factorization case, that it is sufficient to use an inexpensive l_2-norm scaling and we set $\mu = 1$.

2.4 Global Modifications to Prevent Breakdown

Local diagonal modifications were first described in the 1970s by Kershaw [30]. The idea is to simply modify an individual diagonal entry of A during the factorization if it is found to be too small (or negative) to use as a pivot, that is, at the Lines 4–6 of Algorithm 3, instead of terminating the factorization, the pivot is perturbed by some positive quantity so that it is at least τ. While such local modifications are inexpensive to implement within a right-looking factorization algorithm, it is frequently the case that even a small number of modifications can result in a poor preconditioner. Consider, for example, the extreme case that, at the start of the major step k of Algorithm 3 the diagonal entry l_{kk} is equal to zero and the absolute values of all the remaining entries of column k are x_{max}. A local modification that replaces l_{kk} by some chosen value less than 1 prevents B1, but the corresponding column then does not scale (entries overflow) so that B1 is transferred into a B2 problem.

Global strategies are generally more successful in terms of the quality of the resulting preconditioner (see, e.g., [8] and the theoretical and numerical results given in [34, 38, 47]). When an incomplete factorization breaks down, a straightforward strategy is to select a shift $\alpha > 0$, replace the scaled matrix $\widehat{A}$ by $\widehat{A} + \alpha I$ (I is the identity matrix), and restart the factorization. This is outlined in Algorithm 4. The factors of the shifted matrix are used to precondition $\widehat{A}$. If A_D and A_E are, respectively, the diagonal and off-diagonal parts of $\widehat{A}$, then there is always some α for which $(1 + \alpha)A_D + A_E$ is diagonally dominant. Provided the target sparsity pattern of the incomplete factors contains the positions of the diagonal entries, then it can be shown that the incomplete

ALGORITHM 4. **Shifted incomplete IC factorization**

Input: *SPD matrix A, scaling matrix S, and initial shift $\alpha_S \geq 0$, all in precision u*

Output: *Shift $\alpha \geq 0$ and incomplete Cholesky factorization $S^{-1}AS^{-1} + \alpha I \approx LL^T$ in precision $u_l \geq u$*

```
1: $\alpha_0 = 0$
2: $\widehat{A} = S^{-1}AS^{-1}$ in precision $u$
3: if $u_l \neq u$ then $\widehat{A}^{(\ell)} = fl(\widehat{A})$        ▷ Squeeze the scaled matrix into precision $u_l$
4: for $i = 0, 1, 2, \ldots$ do
5:     $\widehat{A}^{(\ell)} + \alpha_i I \approx LL^T$ in precision $u_l$        ▷ Use Algorithm 3
6:     if successful then set $\alpha = \alpha_i$ and return        ▷ This is the case of no breakdown
7:     $\alpha_{i+1} = \max(2\alpha_i,\ \alpha_S)$        ▷ Breakdown detected so increase the shift and restart
8: end for
```

factorization of this shifted matrix does not break down [38]. Diagonal dominance is sufficient for avoiding breakdown, but it is not a necessary condition, and an incomplete factorization may be breakdown free for much smaller values of α. An appropriate α is not usually known *a priori*: too large a value may harm the quality of the incomplete factors as a preconditioner and too small a value will not prevent breakdown, necessitating more than one restart, with a successively larger α. Typically, the shift is doubled after a breakdown, although more sophisticated strategies are sometimes used (see, e.g., [47]).

When using fp16 arithmetic for the factorization ($u_\ell = u_{16}$), Line 3 in Algorithm 4 "squeezes" the scaled matrix $\widehat{A}$ from the working precision u into half precision. The squeezed matrix $\widehat{A}^{(\ell)}$ is factorized using Algorithm 3 with everything in precision u_l. Our experiments on SPD matrices confirm that, provided we prescale A, the number of times we must increase α and restart is generally small but it depends on the IC algorithm and the precision (see the statistics *nmod* and $nofl$ is the tables of results in Section 4).

2.5 Using the Low Precision Factors

Each application of an incomplete LU factorization preconditioner is equivalent to solving a system $LUv = w$ (or $LL^Tv = w$ in the SPD case). This involves a solve with the lower triangular L factor followed by a solve with the upper triangular U (or L^T) factor; these are termed forward and back substitutions, respectively. Algorithm 5 outlines a simple lower triangular solve.

If the factors are computed and stored in fp32 or fp64 arithmetic, then overflows are unlikely to occur in Algorithm 5. However, if half precision arithmetic u_ℓ is used for the factors and the forward and back substitutions are applied in precision u_ℓ, then overflows are much more likely at Lines 3 and 6. We can try and avoid this using simple scaling of the right-hand side so that we solve $LUv = w/\|w\|_\infty$ (or $LL^Tv = w/\|w\|_\infty$) and then set $y = v \times \|w\|_\infty$ (see [15]). Nevertheless, as in problems B2 and B3 above, overflows can still happen. The safe tests for monitoring potential overflow can again be used. If detected, higher precision arithmetic can be used for the triangular solves.

3 LU- and Cholesky Factorization-Based Iterative Refinement

3.1 LU-IR and Krylov-IR Using Low Precision Factors

As already observed, a number of studies in the late 2000s looked at computing the (complete) matrix factors in single precision and then employing them as a preconditioner within an iterative

Algorithm 5. **Forward substitution: lower triangular solve $Ly = w$**
Input: *Lower triangular matrix L with nonzero diagonal entries and right-hand side w.*
Output: *The solution vector y.*

```
1: Initialise $y_j = w_j$, $1 \le j \le n$
2: for $j = 1 : n$ do
3:     if $y_j \ne 0$ then
4:         $y_j \leftarrow y_j / l_{jj}$
5:         for $i = j + 1 : n$ do
6:             if $l_{ij} \ne 0$ then
7:                 $y_i \leftarrow y_i - l_{ij} y_j$
8:             end if
9:         end for
10:    end if
11: end for
```

Algorithm 6. **LU-based iterative refinement using three precisions (LU-IR)**
Input: *Non singular matrix A and vector b in precision u, three precisions satisfying $u_r \le u \le u_\ell$*
Output: *Computed solution of the system $Ax = b$ in precision u*

```
1: Compute the factorization $A = LU$ in precision $u_\ell$
2: Initialize $x_1$ (e.g., by solving $LUx_1 = b$ using substitution in precision $u_\ell$)
3: for i = 1 : itmax or until converged do                ▷ itmax is the maximum iteration count
4:     Compute $r_i = b - Ax_i$ in precision $u_r$; store $r_i$ in precision $u$
5:     Use the computed factors to solve $Ad_i = r_i$ by substitution in precision $u_\ell$; store $d_i$ in precision $u$
6:     Compute $x_{i+1} \leftarrow x_i + d_i$ in precision $u$
7: end for
```

solver to regain double precision accuracy (e.g., [5, 11, 12, 26]). The simplest method is iterative refinement, which seeks to improve the accuracy of a computed solution x by iteratively repeating the following steps until the required accuracy is achieved, the refinement stagnates, or a prescribed limit on the number of iterations is reached.

(1) Compute the residual $r = b - Ax$.
(2) Solve the correction equation $Ad = r$.
(3) Update the computed solution $x \leftarrow x + d$.

A number of variants exist. For a general matrix A, the most common is LU-IR, which computes the LU factors of A in precision u_ℓ and then solves the correction equation by forward and back substitution using the computed LU factors in precision u_ℓ. The computation of the residual is performed in precision u_r and the update is performed in the working precision u, with $u_r \le u \le u_\ell$. This is outlined in Algorithm 6. Here and in Algorithm 7, A and b are held in the working precision u and the computed solution is also in the working precision. A discussion of stopping tests (and many references) may be found in [21] (see also the later papers [7, 18]).

LU-IR can stagnate. In particular, if the LU factorization is performed in fp16 arithmetic, then LU-IR is only guaranteed to reduce the solution error if the condition number $\kappa(A)$ satisfies

ALGORITHM 7. **Krylov-based iterative refinement using precisions (Krylov-IR)**

Input: *Non singular matrix A and vector b in precision u, a Krylov subspace method, and five precisions u_r, u_g, u_p, u, u_ℓ*

Output: *Computed solution of the system $Ax = b$ in precision u*

```
1: Compute the factorization $A = LU$ in precision $u_\ell$
2: Initialize $x_1$ (e.g., by solving $LUx_1 = b$ using substitution in precision $u_\ell$)
3: for $i = 1 : itmax$ or until converged do        ▷ itmax is the maximum iteration count
4:     Compute $r_i = b - Ax_i$ in precision $u_r$; store $r_i$ in precision $u$
5:     Solve $U^{-1}L^{-1}Ad_i = U^{-1}L^{-1}r_i$ using the Krylov method in precision $u_g$, with $U^{-1}L^{-1}A$
         performed in precision $u_p$; store $d_i$ in precision $u$
6:     Compute $x_{i+1} \leftarrow x_i + d_i$ in precision $u$
7: end for
```

$\kappa(A) \ll 2 \times 10^3$. To extend the range of problems that can be tackled, Carson and Higham [14] propose a variant that uses GMRES preconditioned by the LU factors to solve the correction equation. This is outlined in Algorithm 7, with the Krylov subspace method set to GMRES. Carson and Higham use two precisions $u = u_\ell$ and $u_r = u_g = u_p = u^2$; this was later extended to allow up to five precisions [2, 15]. If the LU factorization is performed in fp16 arithmetic and $u_g = u_p = u_{64}$, then the solution error is reduced by GMRES-IR provided $\kappa(A) \ll 3 \times 10^7$. Note that Algorithm 7 requires two convergence tests and stopping criteria: firstly, for the Krylov method on Line 5 (inner iteration) and secondly, for testing the updated solution (outer iteration).

In the SPD case, a natural choice is to choose the Krylov method to be the CG method. However, the supporting rounding error analysis applies only to GMRES, because it relies on the backward stability of GMRES and preconditioned CG is not guaranteed to be backward stable [19]. This is also the case for MINRES. Nevertheless, the MATLAB results presented in [24] suggest that in practice CG-IR generally works as well as GMRES-IR.

We observe that Arioli and Duff [5] earlier proposed a simplified two precision variant of Krylov-IR in which *itmax* was set to 1. They used single and double precision and employed restarted FGMRES [44] as the Krylov solver. This choice was based on their experience that FGMRES is more robust than GMRES [6]. Hogg and Scott [26] subsequently developed a single–double precision solver for large-scale symmetric indefinite linear systems; this Fortran code is available as `HSL_MA79` within the HSL library [27]. It uses a single precision multifrontal method to factorize A (it computes a sparse LDLT factorization) and then mixed precision iterative refinement (that is, LU-IR with $u_r = u = u_{64}$ and $u_\ell = u_{32}$). If iterative refinement stagnates, `HSL_MA79` employs restarted FGMRES to try and obtain double precision accuracy, that is, a switch is automatically made within the code from LU-IR to Krylov-IR with x_1 in Line 2 of Algorithm 7 taken to be the current approximation to the solution computed using LU-IR and $itmax = 1$ (see Algorithms 1–3 of [26]).

3.2 Generalization to Low Precision Incomplete Factors

GMRES-IR can be modified by replacing $U^{-1}L^{-1}$ with a preconditioner M^{-1}. Work on using scalar Jacobi and ILU(0) preconditioners has been reported by Lindquist et al. [35, 36], with tests performed on a GPU-accelerated node combining single and double precision arithmetic. Loe et al. [37] present an experimental evaluation of multi precision strategies for GMRES on GPUs using block Jacobi and polynomial preconditioners. Carson and Khan [16] use an SPAI preconditioner computed in low precisions and present MATLAB results. In addition, Amestoy et al. [3] use the option within

ALGORITHM 8. **Krylov-based iterative refinement with an incomplete factorization preconditioner using five precisions (IC-Krylov-IR)**

Input: *SPD matrix A and vector b in precision u, a Krylov subspace method, and five precisions u_r, u_g, u_p, u and u_ℓ*

Output: *Computed solution of the system $Ax = b$ in precision u*

```
1: Compute an incomplete Cholesky factorization of
   A in precision $u_\ell$                                   ▹ Use Algorithm 4 to compute L
2: Initialize $x_1 = 0$
3: for i = 1 : itmax or until converged do              ▹ itmax is the maximum iteration count
4:     Compute $r_i = b - Ax_i$ in precision $u_r$; store $r_i$ in precision $u$
5:     Solve $Ad_i = r_i$ using the preconditioned Krylov method in precision $u_g$,
           with preconditioning and products with A performed in precision $u_p$; store $d_i$
           in precision u                            ▹ Computed factors used as the preconditioner
6:     Compute $x_{i+1} \leftarrow x_i + d_i$ in precision u
7: end for
```

the MUMPS solver to compute sparse factors in single precision using block low-rank factorizations and static pivoting and then employ them within GMRES-IR to recover double precision accuracy.

Our interest is in using the IC factors of the SPD matrix A as preconditioners within CG-IR and GMRES-IR. Algorithm 8 summarizes the approach, which we call IC-Krylov-IR to emphasize that an IC factorization is used (this is consistent with the notation used in [16]). Note that if $itmax = 1$ then the algorithm simply applies the preconditioned Krylov solver to try and achieve the requested accuracy. Algorithm 6 can be modified in a similar way to obtain what we will call the IC-LU-IR method.

4 Numerical Experiments

In this section, we investigate the potential effectiveness and reliability of half precision IC preconditioners. Our test examples are SPD matrices taken from the SuiteSparse Collection; they are listed in Table 2. In the top part of the table are those we classify as being well-conditioned (those for which our estimate $cond2$ of the 2-norm condition number is less than 10^7) and in the lower part are ill-conditioned examples. We have selected problems coming from a range of application areas and of different sizes and densities. Many are initially poorly scaled and some (including the first three problems in Table 2) contain entries that overflow in fp16 and thus prescaling of A is essential. We use the l_2 norm scaling (computed and applied in double precision arithmetic). We have performed tests using equilibration scaling (implemented using the HSL routine MC77 [42, 43]) and found that the resulting preconditioner is of a similar quality; this is consistent with [47]. The right-hand side vector b is constructed by setting the solution x to be the vector of 1's.

We use Algorithm 8 to explore whether we can recover (close to) double precision accuracy using preconditioners computed in fp16 arithmetic, although we are aware that in practice much less accuracy in the computed solution may be sufficient (indeed, in many practical situations, inaccuracies in the supplied data may mean low precision accuracy in the solution is all that can be justified). We thus use two precisions: $u_\ell = u_{16}$ for the factorization and $u_r = u_g = u_p = u_{64}$. Algorithm 8 terminates when the normwise backward error for the computed solution satisfies

$$res = \frac{\|b - Ax\|_\infty}{\|A\|_\infty \|x\|_\infty + \|b\|_\infty} \leq \delta. \tag{1}$$

Table 2. Statistics for Our Test Examples

Identifier	n	$nnz(A)$	$normA$	$normb$	$cond2$
HB/bcsstk27	1,224	2.87×10^4	2.96×10^7	9.74×10^5	2.41×10^4
Nasa/nasa2146	2,146	3.72×10^4	2.79×10^8	9.05×10^6	1.72×10^3
Cylshell/s1rmq4m1	5,489	1.43×10^5	8.14×10^6	1.73×10^5	1.81×10^6
MathWorks/Kuu	7,102	1.74×10^5	4.73×10^2	5.01	1.58×10^4
Pothen/bodyy6	19,366	7.71×10^4	1.09×10^5	9.81×10^4	9.91×10^4
GHS_psdef/wathen120	36,441	3.01×10^5	1.52×10^3	2.66×10^2	9.58×10^2
GHS_psdef/jnlbrng1	40,000	1.20×10^5	3.29×10^1	2.00×10^{-1}	1.83×10^2
Williams/cant	62,451	2.03×10^6	2.92×10^5	5.05×10^3	8.06×10^3
UTEP/Dubcova2	65,025	5.48×10^5	6.67×10^1	1.18	3.33
Cunningham/qa8fm	66,127	8.63×10^5	4.28×10^{-3}	9.51×10^{-4}	8.00
Mulvey/finan512	74,752	3.36×10^5	3.91×10^2	3.78×10^1	2.51×10^1
GHS_psdef/apache1	80,800	3.11×10^5	8.10×10^5	6.76×10^{-1}	4.18×10^2
Williams/consph	83,334	3.05×10^6	6.61×10^5	7.20×10^3	1.25×10^5
AMD/G2_circuit	150,102	4.38×10^5	2.27×10^4	2.17×10^4	2.02×10^4
Boeing/msc01050	1,050	1.51×10^4	2.58×10^7	1.90×10^6	4.58×10^{15}
HB/bcsstk11	1,473	1.79×10^4	1.21×10^{10}	7.05×10^8	2.21×10^8
HB/bcsstk26	1,922	1.61×10^4	1.68×10^{11}	8.99×10^{10}	1.66×10^8
HB/bcsstk24	3,562	8.17×10^4	5.28×10^{14}	4.21×10^{13}	1.95×10^{11}
HB/bcsstk16	4,884	1.48×10^5	4.12×10^{10}	9.22×10^8	4.94×10^9
Cylshell/s2rmt3m1	5,489	1.13×10^5	9.84×10^5	1.73×10^4	2.50×10^8
Cylshell/s3rmt3m1	5,489	1.13×10^5	1.01×10^5	1.73×10^3	2.48×10^{10}
Boeing/bcsstk38	8,032	1.82×10^5	4.50×10^{11}	4.04×10^{11}	5.52×10^{16}
Boeing/msc10848	10,848	6.20×10^5	4.58×10^{13}	6.19×10^{11}	9.97×10^9
Oberwolfach/t2dah_e	11,445	9.38×10^4	2.20×10^{-5}	1.40×10^{-5}	7.23×10^8
Boeing/ct20stif	52,329	1.38×10^6	8.99×10^{11}	8.87×10^{11}	1.18×10^{12}
DNVS/shipsec8	114,919	3.38×10^6	7.31×10^{12}	4.15×10^{11}	2.40×10^{13}
Um/2cubes_sphere	101,492	8.74×10^5	3.43×10^{10}	3.59×10^{10}	2.59×10^8
GHS_psdef/hood	220,542	5.49×10^6	2.23×10^9	1.51×10^8	5.35×10^7
Um/offshore	259,789	2.25×10^6	1.44×10^{15}	1.16×10^{15}	4.26×10^9

Those in the top half are considered to be well conditioned and those in the lower half to be ill conditioned. $nnz(A)$ denotes the number of entries in the lower triangular part of A. $normA$ and $normb$ are the infinity norms of A and b. $cond2$ is a computed estimate of the condition number of A.

In our experiments, we set $\delta = 10^3 \times u_{64}$. The implementations of CG and GMRES used are `MI21` and `MI24`, respectively, from the HSL software library [27]. Except for the results in Table 4, the CG and the GMRES convergence tolerance is $\delta_{krylov} = u_{64}^{1/4}$ and the limit on the number of iterations for each application of CG and GMRES is 1,000.

The numerical experiments are performed on a Windows 11-Pro-based machine with an Intel(R) Core(TM) i5-10505 CPU processor (3.20 GHz). Our results are for the level-based IC factorization $IC(\ell)$ for a range of values of $\ell \geq 0$. The sparsity pattern of L is computed using the approach of Hysom and Pothen [28]. This computes the pattern of each row of L independently. Our software is written in Fortran and compiled using the NAG compiler (Version 7.1, Build 7118). This is currently the only Fortran compiler that supports the use of fp16. The NAG documentation states that their half precision implementation conforms to the IEEE standard. In addition, using the `-round_hreal` option, all half-precision operations are rounded to half precision, both at compile time and runtime. Section 3 of [23] presents an insightful discussion on rounding every operation or rounding kernels (see also [22]). Because of all the conversions needed, half precision is slower using the NAG compiler than single precision and so timings are not useful.

We refer to the $IC(\ell)$ factorizations computed using half and double precision arithmetic as fp16-$IC(\ell)$ and fp64-$IC(\ell)$, respectively. The key difference between the fp16 and fp64 versions of our $IC(\ell)$ software is that for the former, during the incomplete factorization, we incorporate the safe checks for the scaling and update operations (as discussed in Section 2.2); for the fp64 version, only tests for B1 breakdowns are performed (B2 and B3 breakdowns are not encountered in our double precision experiments). In addition, the fp16 version allows the preconditioner to be applied in either half or double precision arithmetic; the former is for IC-LU-IR and the latter for IC-Krylov-IR. In the IC-Krylov-IR case, preconditioning is performed in double precision. For L computed using fp16, there are two straightforward ways of handling solving systems with L and L^T in double precision. The first makes an explicit copy of L by casting the data into double precision, but this negates the important benefit that half precision offers of reducing memory requirements. The alternative casts the entries on the fly. This is straightforward to incorporate into a serial triangular solve routine and only requires a temporary double precision array of length n.

In the results tables, $nnz(L)$ is the number of entries in the incomplete factor L; $iouter$ denotes the number of iterative refinement steps (that is, the number of times the loop starting at Line 3 in Algorithm 8 is executed) and $totits$ is the total number of CG (or GMRES) iterations performed; $resint$ is Equation (1) with $x = L^{-T}L^{-1}b$ (because L depends on the precision used to compute it, $resint$ is precision dependent), and $resfinal$ is Equation (1) for the final computed solution; $nmod$ and $nofl$ are the numbers of times problems B1 and B3 occur during the incomplete factorization, with the latter for fp16 only. Increasing the global shift and restarting the factorization after the detection of problem B1 avoided problem B2 in all our tests. Problem B3 can occur in column k if after the shift, the diagonal entry l_{kk} is close to the shift α and $|l_{ik}|/x_{max} \geq \alpha$ for some $i > k$. In all the experiments on well-conditioned problems, we found $nofl = 0$ and so this statistic is not included in the corresponding tables of results. As expected, $nmod > 0$ can occur for fp16 and fp64, and for both well-conditioned and ill-conditioned examples.

4.1 Results for IC-LU-IR

IC-LU-IR is attractive because if fp16 arithmetic is used then the application of the preconditioner is performed in half precision arithmetic. Table 3 reports results for IC-LU-IR with the $IC(3)$ preconditioner. The iteration count is limited to 1,000. When using fp16 and fp64 arithmetic, we were unable to achieve the requested accuracy within this limit for some problems. Additionally, for problem Williams/consph, the refinement procedure diverges and the process is stopped when the norm of the residual approaches x_{max} in double precision. For the ill-conditioned problems, results are only given for the ones that were successfully solved; for the other test examples, we failed to achieve convergence.

4.2 Dependence of the Iteration Counts on the CG Tolerance

The results in Table 4 illustrate the dependence of the number of iterative refinement steps and the total iteration count on the convergence tolerance δ_{krylov} used by CG within IC-CG-IR. The preconditioner $IC(3)$ was computed using fp16 arithmetic. For six examples, we test δ_{krylov} ranging from 10^{-10} to 10^{-1}. As expected, the number of outer iterations increases slowly with δ_{krylov}, but the precise choice of δ_{krylov} is not critical. The results confirm the choice of $u_{64}^{1/4}$, which is used for

Table 3. IC-LU-IR Results Using the $IC(3)$ Preconditioner

Preconditioner fp16-$IC(3)$					
Identifier	$resinit$	$resfinal$	$nnz(L)$	$iters$	$nmod$
HB/bcsstk27	8.27×10^{-5}	6.11×10^{-14}	4.88×10^{4}	13	0
Nasa/nasa2146	9.07×10^{-5}	1.89×10^{-13}	7.89×10^{4}	15	0
Cylshell/s1rmq4m1	5.81×10^{-5}	6.77×10^{-9}	3.15×10^{5}	>1,000	0
MathWorks/Kuu	1.74×10^{-4}	2.17×10^{-13}	7.65×10^{5}	194	0
Pothen/bodyy6	7.39×10^{-3}	2.22×10^{-13}	1.76×10^{5}	822	2
GHS_psdef/wathen120	4.34×10^{-4}	1.72×10^{-14}	8.30×10^{5}	5	0
GHS_psdef/jnlbrng1	1.41×10^{-3}	6.48×10^{-14}	2.77×10^{5}	16	0
Williams/cant	3.26×10^{-4}	4.61×10^{-8}	9.95×10^{6}	>1,000	0
UTEP/Dubcova2	1.54×10^{-3}	2.19×10^{-13}	6.22×10^{6}	736	0
Cunningham/qa8fm	3.54×10^{-4}	3.42×10^{-15}	5.14×10^{6}	5	0
Mulvey/finan512	3.50×10^{-4}	2.59×10^{-15}	4.08×10^{6}	5	0
GHS_psdef/apache1	8.52×10^{-5}	1.05×10^{-7}	1.54×10^{6}	>1,000	0
Williams/consph	1.05×10^{-4}	[a]	2.02×10^{7}	[a]	0
AMD/G2_circuit	8.16×10^{-4}	7.71×10^{-8}	1.04×10^{6}	>1,000	0
Oberwolfach/t2dah_e	6.81×10^{-4}	1.42×10^{-14}	3.29×10^{5}	5	0
Um/2cubes_sphere	1.02×10^{-3}	9.10×10^{-16}	8.70×10^{6}	5	0

Preconditioner fp64-$IC(3)$				
Identifier	$resinit$	$resfinal$	$nnz(L)$	$iters$
HB/bcsstk27	3.93×10^{-5}	2.06×10^{-14}	4.88×10^{4}	7
Nasa/nasa2146	4.32×10^{-5}	6.29×10^{-14}	7.89×10^{4}	15
Cylshell/s1rmq4m1	3.55×10^{-5}	4.11×10^{-9}	3.15×10^{5}	>1,000
MathWorks/Kuu	2.50×10^{-4}	1.95×10^{-13}	7.64×10^{5}	110
Pothen/bodyy6	9.31×10^{-3}	2.20×10^{-13}	1.76×10^{5}	>1,000
GHS_psdef/wathen120	1.77×10^{-4}	5.62×10^{-15}	8.30×10^{5}	5
GHS_psdef/jnlbrng1	9.40×10^{-3}	1.14×10^{-13}	2.77×10^{5}	14
Williams/cant	3.19×10^{-4}	2.55×10^{-8}	9.95×10^{6}	>1,000
UTEP/Dubcova2	1.65×10^{-3}	2.17×10^{-13}	6.22×10^{6}	709
Cunningham/qa8fm	3.29×10^{-4}	1.49×10^{-15}	5.14×10^{6}	4
Mulvey/finan512	3.72×10^{-4}	9.22×10^{-15}	4.08×10^{6}	4
GHS_psdef/apache1	8.28×10^{-5}	2.83×10^{-8}	1.54×10^{6}	>1,000
Williams/consph	3.38×10^{-4}	[a]	2.02×10^{7}	[a]
AMD/G2_circuit	8.42×10^{-5}	2.64×10^{-8}	1.04×10^{6}	>1,000
HB/bcsstk24	4.01×10^{-7}	2.21×10^{-13}	2.77×10^{5}	801
HB/bcsstk16	7.01×10^{-4}	1.81×10^{-9}	4.89×10^{6}	>1,000
Oberwolfach/t2dah_e	5.49×10^{-6}	5.95×10^{-14}	3.29×10^{5}	4
Um/2cubes_sphere	2.25×10^{-5}	1.10×10^{-13}	8.70×10^{6}	3

$resint$ and $resfinal$ are the initial and final scaled residuals; $nnz(L)$ is the number of entries in the $IC(3)$ factor; $iters$ is the number of refinement steps; and $nmod$ denotes the number of times problem B1 occurs during the factorization (for fp64-$IC(3)$ it is equal to 0 for all our test cases and is omitted). >1,000 indicates the requested accuracy was not obtained within the iteration limit.

[a]Indicates the refinement procedure breaks down.

Table 4. The Effects of Changing the CG Convergence Tolerance δ_{krylov} Used in IC-CG-IR

	UTEP/Dubcova2									
δ_{krylov}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
$iouter$	2	2	2	2	2	3	3	4	6	8
$totits$	79	73	64	58	49	68	55	57	58	65
	HB/bcsstk26									
δ_{krylov}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
$iouter$	2	2	2	2	2	3	3	4	6	9
$totits$	134	121	107	93	78	103	81	84	97	93
	Cylshell/s2rmt3m1									
δ_{krylov}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
$iouter$	2	2	2	2	2	3	3	4	5	8
$totits$	132	130	125	120	116	94	83	117	85	150
	GHS_psdef/wathen120									
δ_{krylov}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
$iouter$	2	2	2	2	2	2	3	3	6	6
$totits$	9	8	7	6	6	5	6	5	6	6
	GHS_psdef/jnlbrng1									
δ_{krylov}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
$iouter$	2	2	2	2	2	2	3	4	5	7
$totits$	20	18	16	14	12	15	12	12	11	12
	Oberwolfach/t2dah_e									
δ_{krylov}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
$iouter$	2	2	2	2	2	3	3	3	5	5
$totits$	9	8	7	6	5	7	6	5	5	5

The preconditioner is fp16-$IC(3)$. $iouter$ and $totits$ denote the number of outer iterations and the total number of CG iterations, respectively.

all remaining experiments. Similar results are obtained for IC-GMRES-IR, emphasizing that an advantage of the IR approach is that it can significantly reduce the maximum number of Krylov vectors used.

4.3 Results for $IC(0)$

As already remarked, $IC(0)$ is a very simple preconditioner but one that is frequently reported on in publications. Results for IC-CG-IR using an $IC(0)$ preconditioner computed in half and double precision arithmetics are given in Tables 5 and 6 for well-conditioned and ill-conditioned problems, respectively. If the total iteration count ($totits$) for fp16 is within 10% of the count for fp64 (or is less than the fp64 count) then it is highlighted in bold. Note that the number of entries in L is equal to the number of entries in the matrix that is being factorized, that is, $nnz(L) = nnz(\widehat{A}^{(l)})$ for fp16 and $nnz(L) = nnz(A)$ for fp64. The difference between them is the number of entries that underflow and are dropped when the scaled matrix is squeezed into half precision. For the well-conditioned problems, the only problem for which $\widehat{A}^{(l)}$ is significantly sparser than A is Williams/cant but for

Table 5. Results for IC-CG-IR Using an $IC(0)$ Preconditioner: Well-Conditioned Problems

Preconditioner fp16-$IC(0)$						
Identifier	*resinit*	*resfinal*	$nnz(L)$	*iouter*	*totits*	*nmod*
HB/bcsstk27	4.19×10^{-4}	2.02×10^{-14}	2.87×10^{4}	3	**35**	0
Nasa/nasa2146	3.26×10^{-4}	3.82×10^{-15}	3.72×10^{4}	3	**24**	0
Cylshell/s1rmq4m1	1.49×10^{-4}	2.16×10^{-14}	1.15×10^{5}	3	210	0
MathWorks/Kuu	3.27×10^{-3}	1.38×10^{-14}	1.43×10^{5}	3	274	4
Pothen/bodyy6	1.71×10^{-4}	1.58×10^{-16}	7.03×10^{4}	4	178	2
GHS_psdef/wathen120	1.09×10^{-2}	1.09×10^{-13}	3.01×10^{5}	3	**17**	0
GHS_psdef/jnlbrng1	9.98×10^{-3}	1.18×10^{-14}	1.20×10^{5}	3	**40**	0
Williams/cant	4.62×10^{-3}	2.81×10^{-8}	1.46×10^{6}	2	>1,000	9
UTEP/Dubcova2	5.55×10^{-3}	1.86×10^{-13}	4.19×10^{5}	3	**225**	0
Cunningham/qa8fm	3.59×10^{-3}	3.52×10^{-16}	8.63×10^{5}	4	**14**	0
Mulvey/finan512	2.88×10^{-3}	1.25×10^{-14}	3.36×10^{5}	3	**14**	0
GHS_psdef/apache1	3.61×10^{-4}	2.54×10^{-14}	3.11×10^{5}	2	274	0
Williams/consph	8.59×10^{-5}	1.62×10^{-14}	3.05×10^{6}	3	**435**	7
AMD/G2_circuit	7.75×10^{-4}	9.01×10^{-16}	4.38×10^{5}	4	**842**	0
Preconditioner fp64-$IC(0)$						
Identifier	*resinit*	*resfinal*	$nnz(L)$	*iouter*	*totits*	*nmod*
HB/bcsstk27	4.31×10^{-4}	2.42×10^{-14}	2.87×10^{4}	3	35	0
Nasa/nasa2146	3.06×10^{-4}	4.74×10^{-15}	3.72×10^{4}	3	24	0
Cylshell/s1rmq4m1	1.61×10^{-4}	1.57×10^{-14}	1.43×10^{5}	3	176	0
MathWorks/Kuu	9.47×10^{-4}	2.47×10^{-14}	1.74×10^{5}	3	130	0
Pothen/bodyy6	8.68×10^{-5}	1.40×10^{-16}	7.71×10^{4}	4	138	0
GHS_psdef/wathen120	1.09×10^{-2}	1.10×10^{-13}	3.01×10^{5}	3	17	0
GHS_psdef/jnlbrng1	9.96×10^{-3}	1.05×10^{-14}	1.20×10^{5}	3	40	0
Williams/cant	2.63×10^{-3}	4.82×10^{-8}	2.03×10^{6}	2	>1,000	8
UTEP/Dubcova2	5.36×10^{-3}	1.40×10^{-13}	5.48×10^{5}	3	227	0
Cunningham/qa8fm	3.57×10^{-3}	1.23×10^{-16}	8.63×10^{5}	4	14	0
Mulvey/finan512	2.92×10^{-3}	4.18×10^{-14}	3.36×10^{5}	3	14	0
GHS_psdef/apache1	3.63×10^{-4}	1.81×10^{-14}	3.11×10^{5}	2	244	0
Williams/consph	8.26×10^{-5}	2.46×10^{-14}	3.05×10^{6}	3	440	7
AMD/G2_circuit	5.44×10^{-4}	6.55×10^{-16}	4.38×10^{5}	4	855	0

resint and *resfinal* are the initial and final scaled residuals. $nnz(L)$ is the number of entries in the $IC(0)$ factor. *iouter* and *totits* denote the number of outer iterations and the total number of CG iterations, respectively >1,000 indicates CG tolerance not reached on outer iteration *iouter*. *nmod* denotes the number of times problem B1 occurs during the factorization. A count in bold indicates the fp16 result is within 10% of (or is better than) the corresponding fp64 result.

some of the ill-conditioned problems (including Boeing/msc01050 and DNVS/shipsec8), $nnz(L)$ is much smaller for fp16 than for fp64.

From the tables we see that, for well-conditioned problems, using fp16 arithmetic to compute the $IC(0)$ factorization is often as good as using fp64 arithmetic. For problem Williams/cant, on the second outer iteration, the CG method fails to converge within 1,000 iterations for both the fp16 and the fp64 preconditioners. *nofl* is omitted from Tables 5 and 6 because it was 0 for all our test examples. However, for many problems (particularly the ill-conditioned ones), $nmod > 0$

Table 6. Results for IC-CG-IR and IC-GMRES-IR Using an $IC(0)$ Preconditioner: Ill-Conditioned Problems

Preconditioner fp16-$IC(0)$						
Identifier	*resinit*	*resfinal*	$nnz(L)$	*iouter*	*totits*	*nmod*
Boeing/msc01050	1.45×10^{-5}	1.20×10^{-13}	9.49×10^{3}	3 (3)	572 (**333**)	7
HB/bcsstk11	5.37×10^{-4}	1.66×10^{-13}	1.52×10^{4}	3 (3)	914 (644)	5
HB/bcsstk26	6.70×10^{-4}	1.18×10^{-16}	1.33×10^{4}	4 (4)	**488** (**422**)	2
HB/bcsstk24	4.50×10^{-5}	1.41×10^{-12}	7.97×10^{4}	3 (3)	>1,000 (**834**)	3
HB/bcsstk16	3.36×10^{-3}	3.50×10^{-14}	1.27×10^{5}	3 (3)	88 (80)	4
Cylshell/s2rmt3m1	1.41×10^{-4}	9.88×10^{-15}	1.02×10^{5}	3 (3)	347 (**473**)	0
Cylshell/s3rmt3m1	1.13×10^{-5}	1.64×10^{-14}	1.02×10^{5}	3 (3)	>1,000 (>1,000)	3
Boeing/bcsstk38	3.27×10^{-2}	1.58×10^{-10}	1.63×10^{5}	3 (3)	>1,000 (>1,000)	8
Boeing/msc10848	5.46×10^{-6}	1.96×10^{-14}	6.18×10^{5}	3 (3)	>1,000 (**622**)	2
Oberwolfach/t2dah_e	4.87×10^{-2}	5.61×10^{-9}[a]	9.38×10^{4}	1 (4)	**15** (**34**)	0
Boeing/ct20stif	3.82×10^{-3}	1.83×10^{-9}	1.30×10^{6}	3 (3)	>1,000 (>1,000)	7
DNVS/shipsec8	2.31×10^{-3}	2.40×10^{-9}	1.53×10^{6}	3 (3)	>1,000 (>1,000)	8
Um/2cubes_sphere	1.88×10^{-2}	1.77×10^{-14}	8.74×10^{5}	3 (3)	**13** (**11**)	0
GHS_psdef/hood	1.93×10^{-3}	5.01×10^{-17}	5.06×10^{6}	4 (3)	620 (**346**)	2
Um/offshore	1.49×10^{-2}	2.88×10^{-16}	2.25×10^{6}	4 (3)	**602** (**101**)	0
Preconditioner fp64-$IC(0)$						
Identifier	*resinit*	*resfinal*	$nnz(L)$	*iouter*	*totits*	*nmod*
Boeing/msc01050	6.03×10^{-3}	3.11×10^{-14}	1.51×10^{4}	3 (3)	555 (485)	8
HB/bcsstk11	7.49×10^{-4}	7.44×10^{-14}	1.79×10^{4}	3 (3)	902 (475)	4
HB/bcsstk26	1.59×10^{-3}	2.37×10^{-16}	1.61×10^{4}	4 (4)	592 (504)	4
HB/bcsstk24	4.59×10^{-5}	1.11×10^{-13}	8.17×10^{4}	3 (3)	>1,000 (812)	3
HB/bcsstk16	2.76×10^{-3}	2.66×10^{-14}	1.48×10^{5}	3 (3)	68 (66)	0
Cylshell/s2rmt3m1	1.45×10^{-4}	1.10×10^{-14}	1.13×10^{5}	3 (3)	314 (455)	0
Cylshell/s3rmt3m1	1.14×10^{-5}	7.88×10^{-15}	1.13×10^{5}	3 (3)	584 (719)	0
Boeing/bcsstk38	1.21×10^{-2}	1.69×10^{-13}	1.82×10^{5}	3 (3)	>1,000 (>1,000)	7
Boeing/msc10848	1.07×10^{-5}	1.38×10^{-15}	6.20×10^{5}	3 (3)	>1,000 (760)	3
Oberwolfach/t2dah_e	4.86×10^{-2}	1.52×10^{-16}[a]	9.38×10^{4}	1 (4)	15 (34)	0
Boeing/ct20stif	3.65×10^{-3}	3.20×10^{-12}	1.38×10^{6}	3 (3)	>1,000 (>1,000)	7
DNVS/shipsec8	2.75×10^{-4}	6.00×10^{-13}	3.38×10^{6}	3 (3)	>1,000 (>1,000)	4
Um/2cubes_sphere	1.89×10^{-2}	2.55×10^{-14}	8.74×10^{5}	3 (3)	13 (11)	0
GHS_psdef/hood	9.82×10^{-4}	1.73×10^{-13}	5.49×10^{6}	3 (3)	546 (377)	2
Um/offshore	1.51×10^{-2}	1.92×10^{-13}	2.25×10^{6}	4 (3)	590 (100)	0

resint is the initial scaled residual; *resfinal* is the final IC-CG-IR scaled residual. $nnz(L)$ is the number of entries in the $IC(0)$ factor. *iouter* and *totits* denote the number of outer iterations and the total number of CG iterations with the GMRES statistics in parentheses. >1,000 indicates CG (or GMRES) tolerance not reached on outer iteration *iouter*. *nmod* denotes the number of times problem B1 occurs during the factorization. A count in bold indicates the fp16 result is within 10% of (or is better than) the corresponding fp64 result.
[a]Denotes early termination of CG.

for both half precision and double precision and this can lead to a poor quality preconditioner, indicated by high iteration counts, with the limit of 1,000 iterations being exceeded on the third outer iteration for a number of test examples (such as Boeing/bcsstk38 and Boeing/msc10848). Although the requested accuracy is not achieved, there are still significant reductions in the initial

Table 7. Results for IC-GMRES (Without Outer Steps) and IC-GMRES-IR Using the fp16-$IC(0)$ Preconditioner: Ill-Conditioned Problems

Identifier	$resfinal$	$totits$		$maxbasis$
Boeing/msc01050	7.68×10^{-16}	439	(333)	162
HB/bcsstk11	1.28×10^{-14}	602	(644)	510
HB/bcsstk26	3.21×10^{-14}	273	(422)	148
HB/bcsstk24	1.00×10^{-14}	755	(834)	695
HB/bcsstk16	2.05×10^{-15}	61	(80)	29
Cylshell/s2rmt3m1	1.66×10^{-15}	262	(473)	223
Cylshell/s3rmt3m1	2.71×10^{-15}	841	(>1,000)	739
Boeing/bcsstk38	4.01×10^{-13}	>2,000	(>1,000)	>1,000
Boeing/msc10848	4.36×10^{-16}	491	(622)	474
Oberwolfach/t2dah_e	2.35×10^{-16}	54	(34)	11
Boeing/ct20stif	1.71×10^{-12}	>2,000	(>1,000)	>1,000
DNVS/shipsec8	1.52×10^{-12}	>2,000	(>1,000)	>1,000
Um/2cubes_sphere	1.63×10^{-16}	26	(11)	4
GHS_psdef/hood	2.13×10^{-14}	365	(346)	189
Um/offshore	3.32×10^{-13}	>2,000	(101)	92

$resfinal$ is the final scaled residual. $totits$ denotes the number of GMRES iterations, with the counts for IC-GMRES-IR in parentheses. >2,000 indicates convergence is not achieved within 2,000 iterations. >1,000 indicates GMRES tolerance not reached on an outer iteration. $maxbasis$ is the maximum number of GMRES iterations performed on an outer iteration of IC-GMRES-IR.

residual, so the preconditioners may be acceptable if less accuracy is required. Nevertheless, the fp16 performance is often competitive with that of fp64. For problem Oberwolfach/t2dah_e, the CG algorithm terminates before the requested accuracy has been achieved; this is because the curvature encountered within the CG algorithm is found to be too small, triggering an error return.

We have also tested IC-GMRES-IR with the $IC(0)$ preconditioners. For the well-conditioned problems, the iteration counts using GMRES are similar to those for CG. For the ill-conditioned examples, the IC-GMRES-IR counts are given in parentheses in the *iouter* and *totits* columns of Table 6. Our findings are broadly consistent with those reported in [24]. Although IC-GMRES-IR can require fewer iterations than IC-CG-IR, it is important to remember that a GMRES iteration is more expensive than a CG iteration, so simply comparing counts may be misleading.

In Table 7, we give results for $itmax = 1$, that is, preconditioned GMRES is not applied within a refinement loop (we denote this by IC-GMRES). The initial residual, the number of entries in the factor, and the number of occurrences of problem B1 are as in Table 6 and so are not included. We also report the maximum number *maxbasis* of GMRES iterations performed on an outer iteration of IC-GMRES-IR using the same fp16-$IC(0)$ preconditioner. We see that the total iteration count for IC-GMRES-IR can exceed the count for IC-GMRES but, for many of the problems (including Boeing/msc01050, HB/bcsstk26, and Oberwolfach/t2dah_e), the maximum size of the constructed Krylov basis is smaller for IC-GMRES-IR than for IC-GMRES. Another option would be to use restarted GMRES, which would involve selecting an appropriate restart parameter. We do not consider this here.

4.4 Results for $IC(\ell)$

Figure 1 illustrates the influence of the number of levels ℓ in the $IC(\ell)$ preconditioner computed in half precision and double precision. Typically, ℓ is chosen to be small (large values lead to

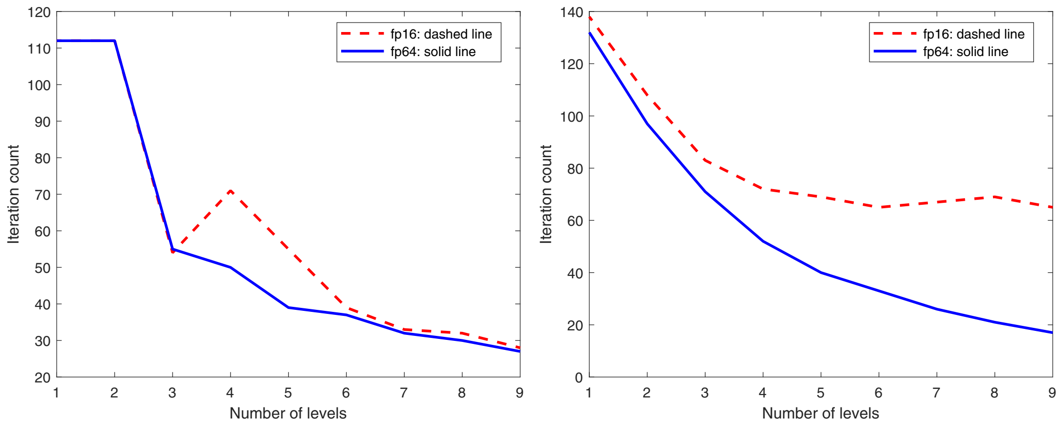

Fig. 1. IC-CG-IR total iteration counts for the fp16+$IC(\ell)$ preconditioner (dashed line) and fp64+$IC(\ell)$ preconditioner (solid line) with l ranging from 1 to 9 for problems UTEP/Dubcova2 (left) and Cylshell/s2rmt3m1 (right).

slow computation times and loss of sparsity in L) although there are cases where larger ℓ may be employed [28]. In general, it is hoped that as ℓ increases, the additional fill-in in L will result in a better preconditioner (but there is no guarantee of this). We observe that for the well-conditioned problem UTEP/Dubcova2, the half precision and double precision preconditioners generally behave in a similar way (with the fp16 preconditioner having a higher iteration count for $\ell = 4$ and 5). As ℓ increases from 1 to 9, the number of entries in the incomplete factor increases from 1.76×10^6 to 3.65×10^7. For the ill-conditioned problem Cylshell/s2rmt3m1, the corresponding increase is from 1.43×10^5 to 5.58×10^5. In this case, the iteration count for fp64+$IC(\ell)$ steadily decreases as ℓ increases but for fp16+$IC(\ell)$ the decrease is much less and it stagnates for $\ell > 4$. In the rest of this section, we use $\ell = 3$.

Tables 8 and 9 present results for IC-CG-IR with the fp16-$IC(3)$ and fp64-$IC(3)$ preconditioners for the well-conditioned and ill-conditioned test sets, respectively. The latter also reports total iteration counts for IC-GMRES-IR (the number of outer iterations *iouter* for IC-GMRES-IR and IC-CG-IR are the same for all the test examples). B3 breakdowns occur for a small number of the ill-conditioned problems. We see that, for our examples, the number of entries in L is (approximately) the same for both fp16 and fp64 arithmetic, indicating that (in contrast to $IC(0)$) the number of subnormal numbers is small. fp16-$IC(3)$ performs as well as fp64-$IC(3)$ on most of the well-conditioned problems. Note that for problem Williams/consph the required accuracy was not achieved using fp64 on the first outer iteration. For the ill-conditioned problems, while the fp16-$IC(3)$ preconditioner combined with IC-CG-IR and IC-GMRES-IR is able to return a computed solution with a small residual, for the problems for which B1 and/or B3 are detected in fp16 arithmetic the iteration counts are significantly greater than for the fp64-$IC(3)$ preconditioner (see, for instance, Boeing/msc01050 and HB/bcsstk11). This is because the use of shifts to prevent breakdowns means the computed factors are for a shifted matrix. However, for a small number of problems the computation of the fp64-$IC(3)$ preconditioner fails because the computed factor has very large entries, which do not overflow in double precision but make it useless as a preconditioner. This indicates that it is not just in half precision arithmetic that it is necessary to monitor the possibility of growth occurring in the factor entries, something that is not currently considered when computing IC factorizations in double (or single) precision arithmetic.

Table 8. Results for IC-CG-IR Using an $IC(3)$ Preconditioner: Well-Conditioned Problems

Preconditioner fp16-$IC(3)$

Identifier	*resinit*	*resfinal*	*nnz(L)*	*iouter*	*totits*	*nmod*
HB/bcsstk27	8.27×10^{-5}	1.23×10^{-15}	4.88×10^{4}	3	11	0
Nasa/nasa2146	9.07×10^{-5}	7.41×10^{-15}	7.89×10^{4}	3	**11**	0
Cylshell/s1rmq4m1	5.81×10^{-5}	1.05×10^{-14}	3.15×10^{5}	3	47	0
MathWorks/Kuu	1.74×10^{-4}	8.21×10^{-15}	7.65×10^{5}	3	**33**	0
Pothen/bodyy6	7.39×10^{-3}	1.40×10^{-16}	1.76×10^{5}	4	108	2
GHS_psdef/wathen120	4.34×10^{-4}	6.69×10^{-16}	8.30×10^{5}	3	**6**	0
GHS_psdef/jnlbrng1	1.41×10^{-3}	5.10×10^{-15}	2.77×10^{5}	3	**12**	0
Williams/cant	3.28×10^{-4}	3.74×10^{-15}	9.95×10^{6}	3	1,103	0
UTEP/Dubcova2	1.54×10^{-3}	1.88×10^{-13}	6.22×10^{6}	3	**54**	0
Cunningham/qa8fm	3.54×10^{-4}	1.04×10^{-16}	5.14×10^{6}	3	**6**	0
Mulvey/finan512	3.50×10^{-4}	4.97×10^{-17}	4.08×10^{6}	3	**6**	0
GHS_psdef/apache1	8.51×10^{-5}	3.65×10^{-14}	1.54×10^{6}	2	**116**	0
Williams/consph	1.04×10^{-4}	1.57×10^{-14}	2.02×10^{7}	3	**160**	0
AMD/G2_circuit	8.15×10^{-4}	3.28×10^{-16}	1.04×10^{6}	4	**282**	0

Preconditioner fp64-$IC(3)$

Identifier	*resinit*	*resfinal*	*nnz(L)*	*iouter*	*totits*
HB/bcsstk27	3.93×10^{-5}	3.13×10^{-16}	4.88×10^{4}	3	8
Nasa/nasa2146	4.32×10^{-5}	8.80×10^{-16}	7.89×10^{4}	3	11
Cylshell/s1rmq4m1	3.55×10^{-5}	1.66×10^{-15}	3.15×10^{5}	3	56
MathWorks/Kuu	2.50×10^{-4}	2.25×10^{-15}	7.65×10^{5}	3	30
Pothen/bodyy6	9.31×10^{-3}	1.40×10^{-16}	1.76×10^{5}	4	63
GHS_psdef/wathen120	1.77×10^{-4}	9.56×10^{-17}	8.30×10^{5}	3	6
GHS_psdef/jnlbrng1	9.40×10^{-4}	3.22×10^{-16}	2.77×10^{5}	3	12
Williams/cant	3.19×10^{-4}	6.09×10^{-15}	9.95×10^{6}	3	927
UTEP/Dubcova2	1.65×10^{-3}	1.84×10^{-13}	6.22×10^{6}	3	55
Cunningham/qa8fm	3.29×10^{-5}	1.24×10^{-16}	5.14×10^{6}	3	5
Mulvey/finan512	3.72×10^{-5}	6.63×10^{-17}	4.08×10^{6}	3	6
GHS_psdef/apache1	8.28×10^{-5}	4.12×10^{-14}	1.54×10^{6}	2	120
Williams/consph	3.39×10^{-4}	6.07×10^{-3}	2.02×10^{7}	1	>1,000
AMD/G2_circuit	8.42×10^{-5}	2.46×10^{-16}	1.04×10^{6}	4	273

resint and *resfinal* are the initial and final scaled residuals. $nnz(L)$ is the number of entries in the $IC(3)$ factor. *iouter* and *totits* denote the number of outer iterations and the total number of CG iterations, respectively. >1,000 indicates CG tolerance not reached on outer iteration *iouter*. *nmod* denotes the number of times problem B1 occurs during the factorization (for fp64-$IC(3)$ it is equal to 0 for all our test cases and is omitted). A count in bold indicates the fp16 result is within 10% of (or is better than) the corresponding fp64 result.

Note that, because the L factor computed using fp16 arithmetic is less accurate than that computed using fp64 arithmetic, the initial residual scaled *resint* given by Equation (1) with $x = L^{-T}L^{-1}$ is typically larger for fp16 than for fp64 arithmetic. However, if double precision accuracy in the computed solution is not required, performing refinement may be unnecessary (or a small number of steps may be sufficient), even for fp16 incomplete factors.

Table 9. Results for IC-CG-IR and IC-GMRES-IR Using an $IC(3)$ Preconditioner: Ill-Conditioned Problems

	Preconditioner fp16-$IC(3)$						
Identifier	$resinit$	$resfinal$	$nnz(L)$	$iouter$	$totits$	$nmod$	$nofl$
Boeing/msc01050	1.61×10^{-3}	6.47×10^{-14}	3.74×10^{4}	3	125 (64)	2	0
HB/bcsstk11	5.84×10^{-4}	1.89×10^{-13}	4.18×10^{4}	3	265 (184)	0	2
HB/bcsstk26	6.42×10^{-5}	1.51×10^{-13}	3.43×10^{4}	3	80 (70)	2	0
HB/bcsstk24	5.91×10^{-7}	1.77×10^{-13}	2.27×10^{5}	3	437 (260)	0	2
HB/bcsstk16	6.16×10^{-4}	6.64×10^{-15}	4.89×10^{5}	3	17 (17)	0	0
Cylshell/s2rmt3m1	9.24×10^{-6}	4.11×10^{-15}	2.60×10^{5}	3	83 (117)	0	0
Cylshell/s3rmt3m1	1.72×10^{-6}	9.01×10^{-15}	2.60×10^{5}	3	**386** (504)	1	1
Boeing/bcsstk38	1.07×10^{-3}	1.43×10^{-15}	5.64×10^{5}	4	1,004 (282)	2	0
Boeing/msc10848	5.97×10^{-7}	1.13×10^{-14}	2.51×10^{6}	3	138 (89)	0	0
Oberwolfach/t2dah_e	6.81×10^{-4}	1.88×10^{-16}	3.29×10^{5}	3	**6** (**6**)	0	0
Boeing/ct20stif	2.78×10^{-5}	1.63×10^{-9}	6.70×10^{6}	3	>1,000 (>1,000)	2	0
DNVS/shipsec8	5.56×10^{-6}	1.26×10^{-16}	1.22×10^{7}	4	2,067 (1,399)	2	0
Um/2cubes_sphere	1.02×10^{-3}	1.63×10^{-16}	8.70×10^{6}	3	**6** (**6**)	0	0
GHS_psdef/hood	6.80×10^{-4}	5.01×10^{-17}	2.78×10^{7}	4	**444** (**406**)	0	4
Um/offshore	1.53×10^{-3}	1.38×10^{-13}	2.08×10^{7}	3	**103** (**40**)	0	4

	Preconditioner fp64-$IC(3)$				
Identifier	$resinit$	$resfinal$	$nnz(L)$	$iouter$	$totits$
Boeing/msc01050	1.41×10^{-4}	5.80×10^{-14}	3.74×10^{4}	3	38 (25)
HB/bcsstk11	1.48×10^{-5}	6.96×10^{-14}	4.18×10^{4}	3	29 (29)
HB/bcsstk26	9.24×10^{-5}	1.68×10^{-13}	3.43×10^{4}	3	60 (62)
HB/bcsstk24	4.01×10^{-7}	1.01×10^{-13}	2.27×10^{5}	3	71 (67)
HB/bcsstk16	7.01×10^{-4}	3.13×10^{-15}	4.89×10^{5}	3	15 (14)
Cylshell/s2rmt3m1	8.61×10^{-6}	8.74×10^{-15}	2.60×10^{5}	3	71 (105)
Cylshell/s3rmt3m1	2.00×10^{-6}	6.12×10^{-5}	2.60×10^{5}	1	>1,000 (>1,000)
Boeing/bcsstk38	4.67×10^{-9}	8.34×10^{-14}	5.64×10^{5}	3	154 115)
Boeing/msc10848	8.73×10^{-10}	2.26×10^{-16}	2.51×10^{6}	3	47 (46)
Oberwolfach/t2dah_e	5.49×10^{-6}	6.54×10^{-16}	3.29×10^{5}	3	5 (5)
Boeing/ct20stif	5.82×10^{-8}	2.02×10^{-11}	6.70×10^{6}	3	>1,000 (>1,000)
DNVS/shipsec8	7.50×10^{-7}	1.11×10^{-16}	1.22×10^{7}	4	313 (181)
Um/2cubes_sphere	2.25×10^{-5}	1.63×10^{-16}	8.70×10^{6}	3	5 (5)
GHS_psdef/hood	[a]	[a]	[a]	[a]	[a] [a]
Um/offshore	[a]	[a]	[a]	[a]	[a] [a]

resint is the initial scaled residual; *resfinal* is the final IC-CG-IR scaled residual. $nnz(L)$ is the number of entries in the $IC(3)$ factor. *iouter* and *totits* denote the number of outer iterations and the total number of CG iterations with the GMRES statistics in parentheses. >1,000 indicates CG (or GMRES) tolerance not reached on outer iteration *iouter*. *nmod* and *nofl* denote the numbers of times problems B1 and $B3$ occur during the factorization (for fp64-$IC(3)$ they are equal to 0 for all our test cases and are omitted). A count in bold indicates the fp16 result is within 10% (or better) of the corresponding fp64 result.

[a]Indicates failure to compute the factorization because of enormous growth in its entries.

5 Concluding Remarks and Future Directions

The focus of this study is the construction and employment of incomplete factorizations using half precision arithmetic to solve large-scale sparse linear systems to double precision accuracy. Our experiments, which simulate half precision arithmetic through the use of the NAG compiler, demonstrate that, when carefully implemented, the use of fp16 level-based incomplete factorization preconditioners may not impact on the overall accuracy of the computed solution, even when a small tolerance is imposed on the requested scaled residual. Unsurprisingly, the number of iterations of the Krylov subspace method that is used in the refinement process can be greater for fp16 factors compared to fp64 factors, but generally this increase is only significant for highly ill-conditioned systems. Our results support the view that, for many real-world problems, it is sufficient to employ half precision arithmetic. Its use may be particularly advantageous if the linear system does not need to be solved to high accuracy. Our study also encourages us to conjecture that by building safe operations into sparse direct solvers, it should be possible to build efficient and robust half precision variants, and because this would lead to substantial memory savings for the matrix factors, it could potentially allow direct solvers to be used (in combination with an appropriate refinement process) to solve much larger problems than is currently possible. This is a future direction that is becoming more feasible to explore as compiler support for fp16 arithmetic improves and becomes available on more platforms and for more languages.

It is of interest to consider other Krylov solvers such as flexible CG and flexible GMRES and to explore other classes of algebraic preconditioners to consider how they can be safely computed using fp16 arithmetic and how effective they are compared to higher precision versions. For SPAI preconditioners, we anticipate that it may be possible to combine the systematic dropping of subnormal quantities with avoiding overflows in local linear solves as we have discussed without significantly affecting the preconditioner quality because such changes may only influence a small number of the computed columns of the SPAI preconditioner (see also [16] for a recent analysis of SPAI preconditioners in mixed precision). For other approximate inverses, such as AINV [9] and AIB [45], the sizes of the diagonal entries follow from maintaining a generalized orthogonalization property and avoiding overflows using local or global modifications may be possible but challenging. We also plan to consider the construction of the HSL_MI28 [47] IC preconditioner using low precision arithmetic. HSL_MI28 uses an extended memory approach for the robust construction of the factors. Once the factors have been computed, the extended memory can be freed, limiting the size of the factors and the work needed in the substitution steps, generally without a significant reduction in the preconditioner quality. A challenge here is safely allowing intermediate quantities of absolute value at least x_{min} to be retained in the factors or in the extended memory.

Acknowledgments

We are very grateful to the anonymous reviewers for their detailed comments that have led to significant improvements to this paper.

References

[1] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tzai, and U. Meier Yang. 2021. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. *Int. J. High Perform. Comput. Appl.* 35, 4 (2021), 344–369. DOI: https://doi.org/10.1177/10943420211003313

[2] P. Amestoy, A. Buttari, N. J. Higham, J.-Y. L'Excellent, T. Mary, and B. Vieuble. 2024. *Five-Precision GMRES-Based Iterative Refinement*. SIAM J. on Matrix Analysis and Applications 45 (2024), 529–552. DOI: https://doi.org/10.1137/23M1549079

[3] P. Amestoy, A. Buttari, N. J. Higham, J.-Y. L'Excellent, T. Mary, and B. Vieuble. 2023. Combining sparse approximate factorizations with mixed precision iterative refinement. *ACM Trans. Math. Software* 49, 1 (2023), 4:1–4:28. DOI: https://doi.org/10.1145/3582493

[4] P. R. Amestoy, I. S. Duff, and J-Y L'Excellent. 2000. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.* 184, 2–4 (2000), 501–520. DOI: https://doi.org/10.1016/S0045-7825(99)00242-X

[5] M. Arioli and I. S Duff. 2009. Using FGMRES to obtain backward stability in mixed precision. *Electron. Trans. Numer. Anal.* 33 (2009), 31–44.

[6] M. Arioli, I. S. Duff, S. Gratton, and S. Pralet. 2007. A note on GMRES preconditioned by a perturbed LDLT decomposition with static pivoting. *SIAM J. Sci. Comput.* 29, 5 (2007), 2024–2044. DOI: https://doi.org/10.1137/060661545

[7] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. 2009. Accelerating scientific computations with mixed precision algorithms. *Comp. Phys. Comm.* 180, 12 (2009), 2526–2533. DOI: https://doi.org/10.1016/j.cpc.2008.11.005

[8] M. Benzi. 2002. Preconditioning techniques for large linear systems: A survey. *J. Comput. Phys.* 182, 2 (2002), 418–477. DOI: https://doi.org/10.1006/jcph.2002.7176

[9] M. Benzi, C. D. Meyer, and M. Tůma. 1996. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Sci. Comput.* 17, 5 (1996), 1135–1149. DOI: https://doi.org/10.1137/S1064827594271421

[10] A. Buttari, J. Dongarra, and J. Kurzak. 2007. *Limitations of the PlayStation 3 for High Performance Cluster Computing*. Technical Report MIMS EPrint: 2007.93. Manchester Institute for Mathematical Sciences, University of Manchester, Manchester, UK.

[11] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. 2008. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Software* 34 (2008), 17:1–17:22. DOI: https://doi.org/10.1145/1377596.1377597

[12] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak. 2007b. Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. J. High Perform. Comput. Appl.* 21, 4 (2007), 457–466. DOI: https://doi.org/10.1177/1094342007084026

[13] E. Carson, N. Higham, and S. Pranesh. 2020. Three-precision GMRES-based iterative refinement for least squares problems. *SIAM J. Sci. Comput.* 42, 6 (2020), A4063–A4083. DOI: https://doi.org/10.1137/20M1316822

[14] E. Carson and N. J. Higham. 2017. A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. *SIAM J. Sci. Comput.* 39, 6 (2017), A2834–A2856. DOI: https://doi.org/10.1137/17M1122918

[15] E. Carson and N. J. Higham. 2018. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM J. Sci. Comput.* 40, 2 (2018), A817–A847. DOI: https://doi.org/10.1137/17M1140819

[16] E. Carson and N. Khan. 2023. Mixed precision iterative refinement with sparse approximate inverse preconditioning. *SIAM J. Sci. Comput.* 45, 3 (2023), C131–C153. DOI: https://doi.org/10.1137/22M1487709

[17] T. F. Chan and H. A. van der Vorst. 1997. Approximate and incomplete factorizations. In *Parallel Numerical Algorithms, ICASE/LaRC Interdisciplinary Series in Science and Engineering,* Vol. 4. D.E. Keyes, A. Sameh, and V. Venkatakrishnan (Eds.), Kluwer Academic Publishers, Dordrecht, 167–202. DOI: https://doi.org/10.1007/978-94-011-5412-3_6

[18] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy. 2006. Error bounds from extra-precise iterative refinement. *ACM Trans. Math. Software* 32, 2 (2006), 325–351. DOI: https://doi.org/10.1145/1141885.1141894

[19] A. Greenbaum. 1997. Estimating the attainable accuracy of recursively computed residual methods. *SIAM J. Matrix Anal. Appl.* 18 (1997), 535–551. DOI: https://doi.org/10.1137/S0895479895284944

[20] M. J. Grote and T. Huckle. 1997. Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput.* 18, 3 (1997), 838–853. DOI: https://doi.org/10.1137/S1064827594276552

[21] N. J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (2nd ed.). SIAM, Philadelphia, PA. DOI: https://doi.org/10.1137/1.9780898718027

[22] N. J. Higham and T. Mary. 2022. Mixed precision algorithms in numerical linear algebra. *Acta Numerica* 31 (2022), 347–414. DOI: https://doi.org/10.1017/S0962492922000022

[23] N. J. Higham and S. Pranesh. 2019. Simulating low precision floating-point arithmetic. *SIAM J. Sci. Comput.* 41, 5 (2019), C585–C602. DOI: https://doi.org/10.1137/19M1251308

[24] N. J. Higham and S. Pranesh. 2021. Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems. *SIAM J. Sci. Comput.* 43, 1 (2021), A258–A277. DOI: https://doi.org/10.1137/19M1298263

[25] N. J. Higham, S. Pranesh, and M. Zounon. 2019. Squeezing a matrix into half precision, with an application to solving linear systems. *SIAM J. Sci. Comput.* 41, 4 (2019), A2536–A2551. DOI: https://doi.org/10.1137/18M1229511

[26] J. D. Hogg and J. A. Scott. 2010. A fast and robust mixed precision solver for sparse symmetric systems. *ACM Trans. Math. Software* 37, 2 (2010), 1–24. DOI: https://doi.org/10.1145/1731022.1731027

[27] HSL 2023. HSL. A Collection of Fortran Codes for Large-Scale Scientific Computation. Retrieved from http://www.hsl.rl.ac.uk

[28] D. Hysom and A. Pothen. 1999. Efficient parallel computation of ILU(k) preconditioners. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC '99).* ACM/IEEE, Portland, OR, 1–29. DOI: https://doi.org/10.1145/331532.331561

[29] M. T. Jones and P. E. Plassmann. 1995. An improved incomplete Cholesky factorization. *ACM Trans. Math. Software* 21, 1 (1995), 5–17. DOI: https://doi.org/10.1145/200979.200981

[30] D. S. Kershaw. 1978. The incomplete Cholesky-conjugate gradient method for the iterative solution of systems of linear equations. *J. Comput. Phys.* 26 (1978), 43–65. DOI: https://doi.org/10.1016/0021-9991(78)90098-0

[31] P. A. Knight, D. Ruiz, and B. Uçar. 2014. A symmetry preserving algorithm for matrix scaling. *SIAM J. Matrix Anal. Appl.* 35, 3 (2014), 931–955. DOI: https://doi.org/10.1137/110825753

[32] J. Kurzak and J. Dongarra. 2007. Implementation of mixed precision in solving systems of linear equations on the CELL processor. *Concurrency Comput. Pract. Exper.* 19, 10 (2007), 1371–1385. DOI: https://doi.org/10.1002/cpe.1164

[33] J. Langou, Jul. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. 2006. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. 50–56. DOI: https://doi.org/10.1109/SC.2006.30

[34] C.-J. Lin and J. J. Moré. 1999. Incomplete Cholesky factorizations with limited memory. *SIAM J. Sci. Comput.* 21, 1 (1999), 24–45. DOI: https://doi.org/10.1137/S1064827597327334

[35] N. Lindquist, P. Luszczek, and J. Dongarra. 2020. Improving the performance of the GMRES method using mixed-precision techniques. In *Proceedings of the Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI: 17th Smoky Mountains Computational Sciences and Engineering Conference (SMC '20), Oak Ridge, TN, USA, August 26–28, 2020, Revised Selected Papers 17*. Springer, 51–66. DOI: https://doi.org/10.1007/978-3-030-63393-6_4

[36] N. Lindquist, P. Luszczek, and J. Dongarra. 2022. Accelerating restarted GMRES with mixed precision arithmetic. *IEEE Trans. Parallel Distrib. Syst.* 33, 4 (2022), 1027–1037. DOI: https://doi.org/10.1109/TPDS.2021.3090757

[37] J. A. Loe, C. A. Glusa, I. Yamazaki, E. G. Boman, and S. Rajamanickam. 2021. Experimental evaluation of multiprecision strategies for GMRES on GPUs. In *Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 469–478. DOI: https://doi.org/10.1109/IPDPSW52791.2021.00078

[38] T. A. Manteuffel. 1980. An incomplete factorization technique for positive definite linear systems. *Math. Comp.* 34 (1980), 473–497. DOI: https://doi.org/10.2307/2006097

[39] S. F. McCormick, J. Benzaken, and R. Tamstorf. 2021. Algebraic error analysis for mixed-precision multigrid solvers. *SIAM J. Sci. Comput.* 43, 5 (2021), S392–S419. DOI: https://doi.org/10.1137/20M1348571

[40] J. A. Meijerink and H. A. van der Vorst. 1977. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.* 31, 137 (1977), 148–162. DOI: https://doi.org/10.2307/2005786

[41] N. Munksgaard. 1980. Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients. *ACM Trans. Math. Software* 6, 2 (1980), 206–219. DOI: https://doi.org/10.1145/355887.355893

[42] D. Ruiz. 2001. *A Scaling Algorithm to Equilibrate Both Rows and Columns Norms in Matrices.* Technical Report RAL-TR-2001-034. Rutherford Appleton Laboratory, Chilton, Oxfordshire, England.

[43] D. Ruiz and B. Uçar. 2011. *A Symmetry Preserving Algorithm for Matrix Scaling*. Technical Report INRIA RR-7552. INRIA, Grenoble, France.

[44] Y. Saad. 1994. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Stat. Comput.* 14 (1994), 461–469. DOI: https://doi.org/10.1137/0914028

[45] Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). SIAM, Philadelphia, PA. DOI: https://doi.org/10.1137/1.9780898718003

[46] J. A. Scott and M. Tůma. 2011. The importance of structure in incomplete factorization preconditioners. *BIT Numer. Math.* 51 (2011), 385–404. DOI: https://doi.org/10.1007/s10543-010-0299-8

[47] J. A. Scott and M. Tůma. 2014. HSL _MI28: An efficient and robust limited-memory incomplete Cholesky factorization code. *ACM Trans. Math. Software* 40, 4 (2014), 1–19. DOI: https://doi.org/10.1145/2617555

[48] J. W. Watts III 1981. A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation. *Soc. Pet. Eng. J.* 21 (1981), 345–353. DOI: https://doi.org/10.2118/8252-PA

[49] M. Zounon, N. J. Higham, C. Lucas, and F. Tisseur. 2022. Performance impact of precision reduction in sparse linear systems solvers. *Peer J Comput. Sci.* 8 (2022), e778 1–22. Retrieved May 13, 2024 from https://peerj.com/articles/cs-778/

Received 1 May 2023; revised 21 February 2024; accepted 24 February 2024