A Fast and Robust Mixed-Precision Solver for the Solution of Sparse Symmetric Linear Systems

J. D. HOGG and J. A. SCOTT
Rutherford Appleton Laboratory

On many current and emerging computing architectures, single-precision calculations are at least twice as fast as double-precision calculations. In addition, the use of single precision may reduce pressure on memory bandwidth. The penalty for using single precision for the solution of linear systems is a potential loss of accuracy in the computed solutions. For sparse linear systems, the use of mixed precision in which double-precision iterative methods are preconditioned by a single-precision factorization can enable the recovery of high-precision solutions more quickly and use less memory than a sparse direct solver run using double-precision arithmetic.

In this article, we consider the use of single precision within direct solvers for sparse symmetric linear systems, exploiting both the reduction in memory requirements and the performance gains. We develop a practical algorithm to apply a mixed-precision approach and suggest parameters and techniques to minimize the number of solves required by the iterative recovery process. These experiments provide the basis for our new code `HSL_MA79`—a fast, robust, mixed-precision sparse symmetric solver that is included in the mathematical software library HSL.

Numerical results for a wide range of problems from practical applications are presented.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Error analysis; linear systems (direct and iterative methods); sparse, structured and very large systems (direct and iterative methods)*; G.4 [**Mathematical Software**]: *Efficiency*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Gaussian elimination, sparse symmetric linear systems, iterative refinement, FGMRES, multifrontal method, mixed precision, Fortran 95

ACM Reference Format:

Hogg, J. D. and Scott, J. A. 2010. A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems. ACM Trans. Math. Softw. 37, 2, Article 17 (April 2010), 24 pages.
DOI = 10.1145/1731022.1731027 http://doi.acm.org/10.1145/1731022.1731027

Work supported by EPSRC grants EP/F006535/1 and EP/E053551/1.
Authors' addresses: Computational Science and Engineering Department, Rutherford Appleton Laboratory, Chilton, OX11 0QX, U.K.; email: {jonathan.hogg, jennifer.scott}@stfc.ac.uk.

DOI 10.1145/1731022.1731027 http://doi.acm.org/10.1145/1731022.1731027

1. INTRODUCTION

A common task in scientific software packages is solving linear systems

$$A\boldsymbol{x} = \boldsymbol{b}, \tag{1}$$

where A is a large sparse symmetric matrix and $\boldsymbol{b}$ is the known right-hand side. There are two common approaches to the solution of such systems:

—*Direct methods*. These are generally variants of Gaussian elimination, involving a factorization $PAP^T \rightarrow LDL^T$ of the system matrix A, where L is a unit lower triangular matrix, D is block diagonal matrix (with 1×1 and 2×2 blocks), and P is a permutation matrix. The solution process is completed by performing forward and then backward substitutions (i.e., by first solving a lower triangular system and then an upper triangular system). Direct methods are popular because, when properly implemented, they are generally robust and achieve a high level of accuracy, making them suitable for use as general-purpose black-box solvers for a wide range of problems. The main limitation of direct methods is that the memory required normally increases rapidly with problem size.

—*Iterative methods*. These involve some iterative scheme and are often based on using Krylov subspaces of A. In general, their performance is dependent upon the availability of an appropriate preconditioner. For large-scale problems, a carefully chosen and tuned preconditioned iterative method will often run significantly faster than a direct solver and will require far less memory; indeed, for very large problems, an iterative method is often the only available choice. Unfortunately, for many of the "tough" systems that arise from practical applications, the difficulties involved in finding and computing a good preconditioner can make iterative methods infeasible.

In this article, we are concerned with using a direct method to obtain an approximate solution to (1) and then applying an iterative method to refine the solution to improve its accuracy. In other words, we use the direct factorization as a preconditioner for an iterative solver. All our results are obtained using multifrontal solvers, but much of our work will apply equally to left- and right-looking supernodal solvers. Modern direct solvers for sparse matrices make extensive use of Level 3 BLAS kernels [Dongarra et al. 1990] in their aim to achieve close to dense performance on modern cache-based architectures (see, e.g., Gould et al. [2007] for an overview of currently available sparse symmetric direct solvers). But direct solvers also perform sparse scatter operations (these involve taking the entries of a dense vector and putting, or scattering, them into a sparse vector). While the speed of the BLAS operations is limited largely by latency and the flop rate, the scatter operations are limited more by memory bandwidth and latency. With the recent emergence of multicore processors and with future chips likely to have ever larger numbers of cores, the data transfer rate and memory latency are expected to become an ever tighter constraint [Graham et al. 2004].

Computing and storing the matrix factor L in single precision (by which we mean working with 32-bit floating-point numbers) rather than in double precision not only offers significant reductions in data movement but gives storage savings. In the dense case, L is the same size as the original matrix so that, assuming the original matrix A is retained (it can be overwritten by L if it is no longer needed), holding L in single precision reduces the total memory by 25%. For a sparse direct solver, the savings in its memory requirements are closer to 50% since the storage is dominated by that required for L which, while sparse, generally contains many more entries than A. As a result, using single precision allows the sparse solver to be used to solve larger problems (or those with denser factors). Single precision is potentially particularly advantageous for an out-of-core sparse direct solver (i.e., one that stores the factors and possibly some of the work arrays in files) because the amount of disk access is also approximately halved.

In addition to the advantages of memory savings and reduced data movement, single-precision arithmetic is currently more highly optimized (and hence faster) than double-precision computation on a number of architectures, such as commodity Intel chips, Cell processors and general-purpose computing on graphics cards (GPGPU). Buttari et al. [2007a] reported differences as great as a factor of 10 in speed. Thus it is highly advantageous to carry out as much computation as possible on these chips using single-precision arithmetic.

In general, direct solvers use double-precision arithmetic, although some packages (including those from the HSL mathematical software library [HSL 2007]) also offer a single-precision version. The accuracy required when solving system (1) is application dependent. If the required accuracy is less than about 10^{-5} (which may be all that is appropriate if, e.g., the problem data is not known to high accuracy), single-precision may often be used. However, users frequently request greater accuracy or, if the problem is very ill conditioned, higher precision may be necessary to obtain a solution that is accurate to at least a modest number of significant figures. In such cases, the double-precision version of the solver has traditionally been used. Motivated by the advantages of using single-precision on modern architectures, recent studies [Arioli and Duff 2008; Buttari et al. 2007b, 2008] have shown that it may be possible to use a matrix factorization computed using the single-precision version of a direct solver as a preconditioner for a simple iterative method that is used to regain higher precision.

Our aim is to design and develop a library-quality mixed-precision sparse solver for the solution of symmetric (possibly indefinite) linear systems and to demonstrate its performance on problems from practical applications. Our mixed-precision solver, which is included within HSL, is called `HSL_MA79`. `HSL_MA79` is a serial code that is built upon the HSL multifrontal solvers `MA57` [Duff 2004] and `HSL_MA77` [Reid and Scott 2008, 2009b]. `HSL_MA77` is an out-of-core solver that is specifically designed for solving large problems. By employing it within `HSL_MA79`, `HSL_MA79` can also be used as an out-of-core solver, enabling it to solve much larger problems than would otherwise be possible. Working out-of-core can add a significant overhead to the time required to factorize and then solve a linear system. In our numerical experiments, we consider whether the savings achieved by performing the factorization in single precision and storing the single-precision factor data in files can offset the time required by the additional solves (each of which involves reading the factor data from disk) that are needed by the iterative procedure used to recover double precision accuracy. In addition to providing users with a mixed-precision solver that is efficient (in terms of both memory requirements and computation times), portable and easy-to-use, our main contribution in this article is to explore how to combine the direct solvers with iterative refinement and with FGMRES [Saad 1994] and, in particular, how to choose and tune the parameters involved to ensure HSL_MA79 is robust and suitable for use both as a black-box linear solver and as a tool for experimenting with mixed-precision solution techniques.

This article is organized as follows. We end this introduction by describing our test environment. Section 2 describes the algorithms used in our mixed-precision solver and, using numerical experiments, establishes default values for the parameters involved, extending both the work of Buttari et al. [2008] and the preliminary MATLAB experimental results of Arioli and Duff [2008]. Section 3 describes our Fortran 95 implementation of the mixed-precision solver HSL_MA79. Numerical results for HSL_MA79 are given in Section 4 and our conclusions are presented in Section 5. The availability of the codes discussed in this article is covered in Section 6.

1.1 Test Environment

All reported experiments in this article are performed on a Dell Precision T5400 with two Intel E5420 quad core processors running at 2.5 GHz backed by 8 GB of RAM. In all our tests, we use the Goto BLAS [Goto and van de Geijn 2008] and the gfortran-4.3 compiler with -O1 optimization. Although our test machine has multiple cores, we used only sequential BLAS, as preliminary experiments with the threaded BLAS seemed to show negligible performance increases (in some cases they gave decreases) in a sparse solver context. All timings were elapsed times in seconds. We worked with two test sets; all but three of the problems were drawn from the University of Florida Sparse Matrix Collection [Davis 2007] and all are symmetric with either real or integer valued entries.

—*Test Set 1*. Small to medium matrices with $n \geq 1000$ and at most 10^7 entries in the upper (or lower) triangular part. This set comprises 330 problems.

—*Test Set 2*. Medium to large matrices with $n \geq 10000$. This set comprises 232 problems.

We note that 170 problems belong to both sets. The problems were held as two test sets because it was more practical to perform a lot of tests on the smaller test problems. Furthermore, MA57 was not able to solve the largest problems in Test Set 2 (because of insufficient memory); for these, the out-of-core facilities offered by HSL_MA77 were needed. In all our experiments, we used threshold partial pivoting with the threshold parameter set to $u = 0.01$ (thus all the test problems were treated as indefinite, even though some are known to be positive

Table I. Summary of IEEE 754-2008 Standard, Comparing Single and Double Precision

	Single	Double
Sign + exponent + mantissa	$32 = 1 + 8 + 23$	$64 = 1 + 11 + 52$
Epsilon ($\min \epsilon : 1 + \epsilon \neq 1$)	$2^{-24} \approx 6.0 \times 10^{-8}$	$2^{-53} \approx 1.1 \times 10^{-16}$
Underflow threshold	$2^{-126} \approx 1.2 \times 10^{-38}$	$2^{-1022} \approx 2.3 \times 10^{-308}$
Overflow threshold	$(2 - 2^{-23}) \times 2^{127}$ $\approx 3.4 \times 10^{38}$	$(2 - 2^{-52}) \times 2^{1023}$ $\approx 1.8 \times 10^{308}$

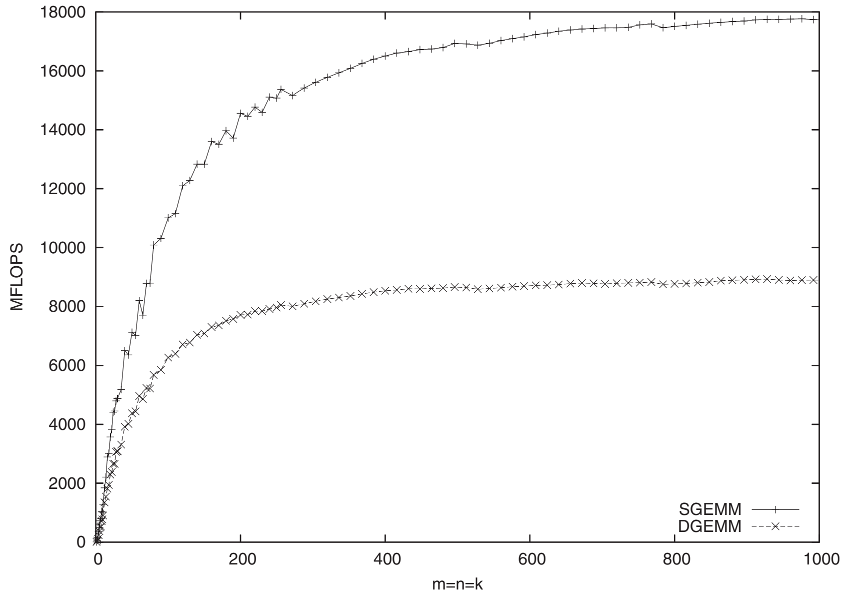

Fig. 1. The performance of single- (SGEMM) and double- (DGEMM) precision matrix-matrix multiplication for a range of sizes of square matrices. These experiments used the Goto BLAS.

definite). Furthermore, we scaled the test problems using the HSL package MC77 (the ∞-norm scaling was used) [Hogg and Scott 2008; Ruiz 2001].[1]

We end this section by summarizing in Table I the IEEE floating-point standard [IEEE 2008]. Of particular interest are the epsilon values, which effectively limit the precision in direct solvers.

2. ALGORITHM

As we have already observed, single-precision operations can be performed much faster than double-precision, speeding up core operations (this was explained in detail in [Buttari et al. 2007b]). This is illustrated in Figure 1. Here we show the performance of the Level 3 BLAS kernel for matrix-matrix multiplication in single precision (SGEMM) and in double-precision (DGEMM) for

[1]In our tests, the direct solvers were unable to solve problems GHS_indef/boyd1 and GHS_ indef/aug3d with this choice of scaling so, in these instances, we scaled using MC64 [Duff 2004; Duff and Koster 2001].

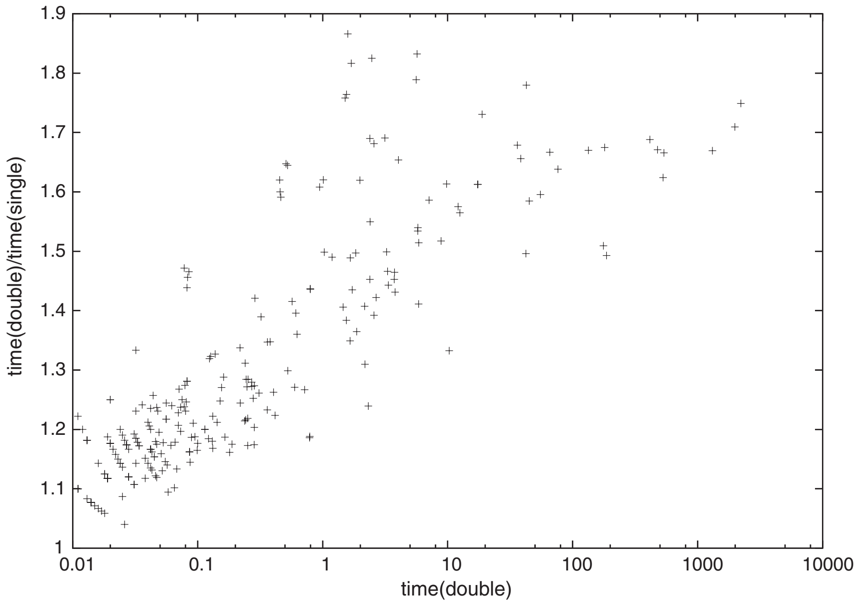

Fig. 2. The ratios of the times required by the factorization phase of the sparse direct solver MA57 when run in single and double precision (the problems are a subset of Test Set 1).

square matrices of order up to 1000. Since GEMM is used extensively within modern sparse direct solvers (including MA57 and HSL_MA77), this demonstrates the potential advantage of performing the factorization using single precision. Figure 2 shows how this translates into a performance gain for the factorization phase of MA57 (here the test set comprises the subset of Test Set 1 problems that take at least 0.01 s to factorize in double precision on our test machine). While we do not see gains of quite the factor of 2 that was achieved for SGEMM, we do see worthwhile improvements on the larger problems, that is, those taking longer than about 1 s to factorize. Here _GEMM operations dominate the factorization time, whereas, on the smaller problems, integer operations as well as book-keeping are more dominant and for such problems there may be little reward in terms of computation time in pursuing a mixed-precision approach.

In our early runs of MA57, floating-point underflows caused single precision to underperform double. This was because of the way modern Intel CPUs handle such events. We were able to reduce this problem by setting a processor flag to flush denormals to zero, avoiding a cascade of slow operations, although there is still a potential speed drop if a significant number occur (indeed, a similar situation can occur in double-precision but is less common because of the much larger exponent range). There was also one test matrix (not shown in Figure 2) for which the ratio of the single-precision to the double-precision factorization time was well in excess of 2. This was an indefinite problem and although in both cases the pivot sequence was the same, during the factorization this sequence was modified to maintain numerical stability more in the double- than in the single-precision case, resulting in a higher flop count and dense factor.

Algorithm 1. Mixed-precision solver.

```
Input: Requested accuracy $\gamma$
Set $prec = single$
do
  Factorize $PAP^T$ as $LDL^T$ using precision $prec$
  Solve $A\boldsymbol{x} = \boldsymbol{b}$ and compute $\beta$.
  if $\beta \le \gamma$ then exit
  Perform iterative refinement (Algorithm 2)
  if $\beta \le \gamma$ then exit
  Perform FGMRES (Algorithm 3)
  if $\beta \le \gamma$ then exit
  if $prec = single$ then
    Set $prec = double$
  else
    Set error flag and exit
  endif
end do
```

Our aim was to perform a single-precision factorization and then, if necessary, use double-precision postprocessing to recover a solution to the desired precision. For maximum efficiency, we wanted to try the cheapest algorithm first and, only if this failed, did we want to resort to applying more computationally expensive alternatives. In the worst case, we would fall back to performing a double-precision factorization.

Setting $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$, we define the norm of the scaled residual (the backward error) to be

$$\beta = \frac{\|\boldsymbol{r}\|_\infty}{\|A\|_\infty \|\boldsymbol{x}\|_\infty + \|\boldsymbol{b}\|_\infty}. \qquad (2)$$

We require that the computed solution $\boldsymbol{x}$ satisfies $\beta \le \gamma$, where γ is a parameter chosen by the user. If $\beta \le \gamma$ is satisfied, we say that the *requested accuracy* has been achieved. This provides a stopping criteria in our algorithms. We note that, in the case where we scale A prior to the factorization, the unscaled matrix is retained for any iterative method used to refine the solution and (2) is used with the unscaled A for checking the accuracy (see also Hogg and Scott [2008]).

Algorithm 1 summarizes the basic mixed-precision approach. The factorization is performed in single precision then, if the scaled residual β exceeds γ, mixed-precision iterative refinement (Algorithm 2) is performed. If the requested accuracy has not yet been achieved, mixed-precision FGMRES (Algorithm 3) is used and, finally, if β is still too large, a switch is made to double precision and the computation restarted. In the next two sections, we discuss the iterative refinement and FGMRES steps.

2.1 Iterative Refinement

Iterative refinement is a simple first-order method used to improve a computed solution of (1); it is outlined in Algorithm 2. Here β_k is the norm of the scaled residual (2) on the kth iteration and `i_maxitr` is the maximum number of iterations. The system $A\boldsymbol{x} = \boldsymbol{b}$ and the correction equation $A\boldsymbol{y}_{k+1} = \boldsymbol{r}_k$ are solved

Algorithm 2. Mixed-precision iterative refinement.

```
Input: Single-precision factorization of $A$, double-precision right-hand size $\boldsymbol{b}$,
       requested accuracy $\gamma$, minimum reduction $\delta$ and i_maxitr
Solve $A\boldsymbol{x}_1 = \boldsymbol{b}$ (using single precision)
Compute $\boldsymbol{r}_1 = \boldsymbol{b} - A\boldsymbol{x}_1$ and $\beta_1$ (using double precision)
Set $k = 1$
do while ($\beta_k > \gamma$ and $k <$ i_maxitr)
  Solve $A\boldsymbol{y}_{k+1} = \boldsymbol{r}_k$ (using single precision)
  Set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{y}_{k+1}$ (using double precision)
  Compute $\boldsymbol{r}_{k+1} = \boldsymbol{b} - A\boldsymbol{x}_{k+1}$ and $\beta_{k+1}$ (using double precision)
  if $\beta_{k+1} > \delta\beta_k$ or $\|\boldsymbol{r}_{k+1}\|_\infty \geq 2\|\boldsymbol{r}_k\|_\infty$ then Set error flag and exit (stagnation)
  Set $k = k + 1$
end do
$\boldsymbol{x} = \boldsymbol{x}_k$
```

Algorithm 3. Mixed-precision FGMRES right preconditioned by a direct solver with adaptive restarting. Norms here are 2-norms.

```
Input: Single-precision factorization of $A$, double-precision right-hand size $\boldsymbol{b}$,
       $\gamma, \delta$, f_maxitr, restart, max_restart
Solve $A\boldsymbol{x} = \boldsymbol{b}$
Compute $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$ and $\beta$
Initialise $j = 0; \beta_{\text{old}} = \beta; \boldsymbol{x}_{\text{old}} = \boldsymbol{x}$
do while ($\beta > \gamma$ and $j <$ f_maxitr)
  $\beta_{\text{old}} = \beta$
  Initialise $\boldsymbol{v}_1 = \boldsymbol{r}/\|\boldsymbol{r}\|$, $\boldsymbol{y}_0 = \boldsymbol{0}$, $k = 0$
  do while($\big\| \|\boldsymbol{r}\|\boldsymbol{e}_1 - H_k\boldsymbol{y}_k \big\| \geq \gamma(\|A\|\|\boldsymbol{x}\| + \|\boldsymbol{b}\|)$ and $k <$ restart)
    $k = k + 1$ (Increment restart counter)
    $j = j + 1$ (Increment iteration counter)
    Solve $A\boldsymbol{z}_k = \boldsymbol{v}_k$ and compute $\boldsymbol{w} = A\boldsymbol{z}_k$
    Orthogonalize $\boldsymbol{w}$ against $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ to obtain a new $\boldsymbol{w}$. Set $\boldsymbol{v}_{k+1} = \boldsymbol{w}/\|\boldsymbol{w}\|$
    Form $H_k$, a trapezoidal basis for the Krylov subspace spanned by $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$
      (Full details of this step may be found in Saad [2003])
    $\boldsymbol{y}_k = \arg\min_{\boldsymbol{y}} \big\| \|\boldsymbol{r}\|\boldsymbol{e}_1 - H_k\boldsymbol{y}_k \big\|$
      (Minimize residual over the Krylov subspace)
  end do
  Set $Z_k = [\boldsymbol{z}_1 \cdots \boldsymbol{z}_k]$
  Compute $\boldsymbol{x} = \boldsymbol{x} + Z_k\boldsymbol{y}_k$, $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$ and compute new $\beta$
  if $\beta \geq \delta\beta_{\text{old}}$
    restart = $2 \times$ restart
    if restart > max_restart then Set error flag and exit
  end if
  if $\beta > \beta_{\text{old}}$
    $\boldsymbol{x} = \boldsymbol{x}_{\text{old}}$
  else
    $\boldsymbol{x}_{\text{old}} = \boldsymbol{x}; \beta_{\text{old}} = \beta$
  end if
end do
```

using the computed single-precision factors of A. There are a number of ways this could be done depending on what data type conversions we perform. In Algorithm 2, "solve $A\boldsymbol{x} = \boldsymbol{b}$ using single precision" means that we transform

Table II. Number of Problems in Test Set 1 for Which Iterative Refinement Achieved Requested Accuracy ($\gamma = 5 \times 10^{-15}$) Using a Range of Values of the Improvement Parameter δ (`i_maxitr`= 10) (Results for our default setting are in bold.)

δ	0.001	0.01	0.05	0.07	0.08	0.1	0.2	**0.3**	0.4	0.5	∞
Converged	194	227	252	256	258	259	264	**265**	265	265	268
Failed	136	103	78	74	72	71	66	**65**	65	65	62

the input right-hand side vector $\boldsymbol{b}$ from double to single precision, use the single-precision factorization of A to solve for $\boldsymbol{x}$ in single precision, and then transform $\boldsymbol{x}$ from single to double precision. This allows us to use an unmodified single-precision version of the solution phase of our direct solver. The transformations between single- and double-precision vectors are done by taking copies that are transitory and exist only for the duration of the solve; all vectors are otherwise kept in double precision.

Skeel [1980] proved that, to reduce the scaled residual to a given precision, it is sufficient to compute the residual and the correction in that precision. However, since we wished to obtain residuals with double-precision accuracy using factors computed in single precision, we performed the forward and back substitutions (which we refer to as the *solves* throughout the rest of this article) in single precision and compute the residuals and corrected solution $\boldsymbol{x}_{k+1}$ in double precision. This mixed-precision version of iterative refinement was also used by Buttari et al. [2007b] and Buttari et al. [2008], while Demmel et al. [2009] used a similar scheme for overdetermined least-squares problems, further developing bounds on the forward error.

Iterative refinement generally decreases the residual significantly for a number of iterations before stagnating (i.e., reaching a point after which little further accuracy is achieved), although for some problems (including the test problems `HB/bcsstm27`, `Cylshell/s3rmq4m1`, and `GHS_psdef/s3dkq4m2` that were considered in Arioli and Duff [2008]), a large number of iterations are needed before any substantial reduction in the residual is achieved. To detect stagnation (and thus avoid performing unnecessary solves), we employed a minimum improvement parameter δ. A large δ allows the iterative refinement to continue until the maximum number of iterations has been performed. This increases the likelihood of convergence at the expense of carrying out additional iterations for problems that have stagnated before reaching the requested accuracy γ. The number of additional iterations can be reduced or eliminated by choosing a small δ. The condition $\|\boldsymbol{r}_{k+1}\|_\infty \geq 2\|\boldsymbol{r}_k\|_\infty$ detects numerical issues where $\boldsymbol{x}_{k+1}$ explodes but the residual β_{k+1} remains small due to scaling by $\boldsymbol{x}_{k+1}$. For different values of δ, Table II reports the number of problems belonging to Test Set 1 that achieved the requested accuracy when factorized using `MA57` in single precision and then corrected using mixed-precision iterative refinement. We see that values in the range $[0.05, 0.5]$ have a similar success rate of just under 80%. We chose as our default $\delta = 0.3$ as this provides a good compromise between the number of problems that converged (259) and minimizes the wasted iterations on the remainder—62% of the problems that failed with this δ used the same number of solves as for $\delta = 0.001$, and only 4 of the 65 required more than two additional iterations before stagnation was recognized.

We remark that the package MA57 includes an option (which we did not use) to perform iterative refinement (using the same precision as the factorization) and, by default, it uses $\delta = 0.5$ (see also Demmel et al. [2006] and Higham [1997]).

Our implementation of iterative refinement also offers an option to terminate once a chosen number i_maxitr of iterations has been performed. An upper limit on the maximum number of iterations can be established by considering the following example. Assume the initial scaled residual is $\beta = 10^{-7}$ and the default improvement parameter $\delta = 0.3$ is used. If stagnation has not occurred, after 13 iterations the scaled residual must be less than 1.6×10^{-14} and, after 15 iterations, less than 4.8×10^{-15}. Based on our experiments, we set the default value to i_maxitr $= 10$ (note that for most of our test examples we found that either the requested accuracy was achieved or stagnation was recognized before this limit was reached).

2.2 Preconditioned FGMRES

For our examination of FGMRES, we used the 62 problems from Test Set 1 that failed to achieve the requested accuracy using iterative refinement with any δ. We call this the *Reduced Test Set 1*.

In Algorithm 1, FGMRES [Saad 1994] refers to a right-preconditioned variant of FGMRES. Arioli et al. [2007] have shown that, in cases where iterative refinement fails, FGMRES may succeed and is more robust than either iterative refinement or GMRES. Arioli and Duff [2008] proved that a mixed-precision computation, where the matrix factorization is computed in single-precision and the FGMRES iteration in double-precision, gives a backward stable algorithm. We note that their result is given for the restart counter k in Algorithm 3 sufficiently large and so is not of practical use in our case because we will only use a small number of iterations.

Our variant of FGMRES, shown as Algorithm 3, is essentially that given in Arioli and Duff [2008] but additionally uses an adaptive restart parameter. Here f_maxitr is the maximum number of iterations and $\boldsymbol{e}_1$ denotes the first column of the identity matrix. Algorithm 3 uses double precision throughout except for the solution of the systems involving A; for these systems, we have the ability to perform the forward and backward substitutions in either single or double precision, as we detail below. Our adaptive restarting strategy relies on a similar concept to the minimum improvement parameter in iterative refinement—we expect to reduce the backward error in the outer iterations and, if the reduction is too small, we increase the restart parameter (up to a specified maximum max_restart). If there is no reduction in the backward error β (although in exact arithmetic the convergence of FGMRES is nonincreasing, in finite-precision arithmetic the backward error may increase), we restore the solution from the previous outer iteration before restarting. In our experiments, we compared adaptive restarting with using a fixed restart parameter. We found that a small initial value for the adaptive restart parameter (typically less than or equal to 4) reduced the number of iterations required to obtain the requested accuracy and enabled us to solve some problems that failed to converge with a fixed restart;

Table III. Comparison of Performance of FGMRES Using Single- and Double-Precision Solves Following Single-Precision Factorization for a Range of Restart Parameters on Reduced Test Set 1 ($\delta = 0.3$, `f_maxitr=128`, `max_restart=128`)

`restart` =	1	2	4	8	16
Problems failed for both single- and double-precision solves	23	23	23	23	23
Problems solved using double- but not single-precision solve	5	5	5	5	5
Average difference in number of solves (see (3))	14.9	15.6	16.2	12.9	12.9
Average ratio of number of solves (see (4))	2.8	2.5	3.1	2.1	2.1

the strategy had little effect for larger initial values. All FGMRES results in this article are hence based on this adaptive restarting algorithm.

An alternative to the "solve using single precision" described in Section 2.1 is to convert the computed factors from single to double precision and then use the double-precision version of the solution phase of our direct solver. The additional memory needed for this can be limited by writing a special-purpose single-to-double solve routine that takes the double-precision right-hand side $\boldsymbol{b}$ together with the single-precision factorization of A and returns the computed solution $\boldsymbol{x}$ in double precision. To avoid holding the factorization in both single and double precision, the routine converts the single-precision entries of L to double precision only as they are required and discards them afterwards. For our multifrontal solvers, the additional double-precision storage needed is at most that required for the largest frontal matrix. We refer to this technique as *solve using double precision*.

Performing the solves using single-precision has obvious speed advantages per iteration, but, if solving using double-precision results in a lower iteration count, it may be faster overall. Our experiments using a range of values for the adaptive restart parameter showed that, in all cases, using double precision reduces the total number of solves required. Table III attempts to capture to what extent the double-precision approach is better. Shown here are

— the number of problems that fail to converge using either single- or double-precision solves;
— the number of problems that converge only using double-precision solves;
— the arithmetic mean of the number of extra solves needed when solving using single precision, that is,

$$\frac{1}{|\mathcal{P}|}\sum_{i\in\mathcal{P}}\left(\text{Solves}_i(\textit{single}) - \text{Solves}_i(\textit{double})\right), \qquad (3)$$

where $\text{Solves}_i(\textit{double})$ ($\text{Solves}_i(\textit{single})$) is the number of solves used for problem i when using double- (single-) precision solves, and $\mathcal{P}$ is the set of problems on which both single- and double-precision solves converge to the requested accuracy.
— The geometric mean of the ratio of the number of solves using single precision to the number using double precision, that is,

$$\left(\prod_{i\in\mathcal{P}}\frac{\text{Solves}_i(\textit{single})}{\text{Solves}_i(\textit{double})}\right)^{\frac{1}{|\mathcal{P}|}}. \qquad (4)$$

A possible explanation of the good performance of the solves using double precision is that the norms used in FGMRES are 2-norms rather than the ∞-norms used elsewhere in this article, and are prone to larger error accumulation as a result, which higher precision may help to counter.

We comment that a constant number of failing problems regardless of the restart value is what we expect from our adaptive restarting procedure.

The reduction in the number of solves when using double precision has to be set against the increased time per solve compared to single precision. On our test computer, the time for a solve in double precision is approximately twice that in single precision; hence if the ratio of the number of solves in single precision to those in double is more than 2 (as is the case in the last line of Table III) then double precision is faster, while also allowing us to solve more problems. As a result, in the remainder of this article we use double-precision solves for FGMRES. Note that iterative refinement will still use single-precision solves as similar experiments showed there to be no significant benefit in using double precision. In particular, although more iterations were carried out, iterative refinement still eventually stagnates on the problems belonging to the Reduced Test Set 1.

In Table IV, we report the number of solves performed within FGMRES for a range of values for the adaptive restart parameter on the 39 problems in the Reduced Test Set 1 for which FGMRES was successful. The results indicate that `restart` = 4, 8, or 16 was generally the best choice; `restart` = 4 was taken to be the default. We observe that for some problems a higher `restart` parameter resulted in a larger number of iterations. This was because the termination conditions were only tested when the algorithm was restarted; higher values of `restart` tested for termination less frequently.

3. IMPLEMENTING THE MIXED-PRECISION STRATEGY

In this section, we discuss the design and development of our new mixed-precision solver `HSL_MA79`. The code is written in Fortran 95 and, at its heart, uses the HSL direct solvers `MA57` [Duff 2004] and `HSL_MA77` [Reid and Scott 2008, 2009b]. We start by giving a brief overview of these solvers, highlighting some of their key features that are important for `HSL_MA79`.

3.1 `MA57`

`MA57` is designed to solve sparse symmetric linear systems (1); the system matrix may be either positive definite or indefinite. The multifrontal method is used [Duff and Reid 1983]. A detailed discussion of the design of `MA57` and its performance is given by Duff [2004]. Relevant work on the pivoting and scaling strategies available within `MA57` is given by Duff and Pralet [2005, 2007].

In common with other HSL solvers, `MA57` is available in both double- and single-precision versions. It offers a range of options, including solving for multiple right-hand sides, computing partial solutions, error analysis, a matrix modification facility, and a stop and restart facility. Although the default settings for the control parameters should work well in general, there are several

Table IV. Number of Solves Performed within FGMRES for Reduced Test Set 1 Using a Range of Values for `restart` Following Unsuccessful Iterative Refinement. (For each problem, the best result is in bold. Results are shown for $\delta = 0.3$, `f_maxitr`= 64, and `max_restart`=64. The 23 problems that failed for all values of `restart` are not shown.)

Problem	`restart`				
	1	2	4	8	16
`Boeing/bcsstk35`	49	**48**	64	**48**	**48**
`Boeing/bcsstk38`	**3**	4	4	8	8
`Boeing/crystk01`	15	14	12	**8**	**8**
`Boeing/crystk02`	**3**	6	4	8	8
`Boeing/crystk03`	**3**	6	4	8	8
`Boeing/msc01050`	7	6	**4**	8	8
`Cunningham/qa8fk`	**3**	6	4	8	8
`Cylshell/s3rmq4m1`	11	10	**8**	**8**	**8**
`Cylshell/s3rmt3m1`	17	12	**4**	8	8
`DNVS/ship_001`	**48**	**48**	**48**	64	64
`GHS_indef/cont-201`	25	24	20	**16**	**16**
`GHS_indef/cvxqp3`	23	**22**	32	24	24
`GHS_indef/ncvxbqp1`	11	**8**	**8**	**8**	**8**
`GHS_indef/ncvxqp5`	10	10	**8**	**8**	**8**
`GHS_indef/sparsine`	14	14	**12**	16	16
`GHS_indef/stokes128`	**1**	2	4	8	8
`GHS_psdef/oilpan`	11	10	**8**	**8**	**8**
`GHS_psdef/s3dkq4m2`	40	16	16	**8**	**8**
`GHS_psdef/s3dkt3m2`	17	**16**	**16**	**16**	**16**
`GHS_psdef/vanbody`	31	30	28	**24**	**24**
`Gset/G33`	**2**	**2**	4	8	8
`HB/bcsstm27`	13	12	**8**	**8**	**8**
`Koutsovasilis/F2`	**3**	6	4	8	8
`ND/nd3k`	**2**	**2**	4	8	8
`Oberwolfach/gyro`	10	**8**	**8**	**8**	**8**
`Oberwolfach/gyro_k`	10	**8**	**8**	**8**	**8**
`Oberwolfach/t2dah`	7	6	**4**	8	8
`Oberwolfach/t2dah_a`	7	6	**4**	8	8
`Oberwolfach/t2dal`	7	6	**4**	8	8
`Oberwolfach/t2dal_a`	7	6	**4**	8	8
`Oberwolfach/t2dal_bci`	7	6	**4**	8	8
`Oberwolfach/t3dh`	**3**	6	4	8	8
`Oberwolfach/t3dh_a`	**3**	6	4	8	8
`Oberwolfach/t3dl`	**3**	6	4	8	8
`Oberwolfach/t3dl_a`	**3**	6	4	8	8
`Pajek/Reuters911`	15	12	12	**8**	**8**
`Schenk_IBMNA/c-56`	31	30	28	**16**	**16**
`Schenk_IBMNA/c-62`	31	30	28	**24**	**24**
`Simon/olafu`	16	12	16	**8**	**8**

parameters available to enable the user to tune the code for his or her problem class or computer architecture.

Like many modern symmetric direct solvers, MA57 has three distinct phases: an analyse phase that works only with the sparsity pattern to set up data structures for the factorization, the numerical factorization phase that uses these data structures to compute the matrix factor, and a solve phase that may be called any number of times after the factorization is complete to solve repeatedly for different right-hand sides.

The efficiency of a direct method, in terms of both the storage needed and the work performed, is dependent on the order in which the elimination operations are performed, that is, the order in which the pivots are selected. For symmetric matrices that are positive definite, the pivotal sequence chosen using the sparsity pattern of A alone can be used during the factorization without modification. For symmetric indefinite problems, a tentative pivot sequence is chosen based upon the sparsity pattern (treating zeros on the diagonal as entries) and this is modified if necessary (possibly to include 2×2 pivots) during the factorization to maintain numerical stability. MA57 offers the user a number of ordering options, including variants of the minimum degree algorithm [Amestoy et al. 1996, 2004] and multilevel nested dissection through an interface to the well-known MeTiS package [Karypis and Kumar 1999]. Based on the study by Duff and Scott [2005], by default, MA57 chooses between MeTiS and approximate minimum degree (avoiding problems with dense rows [Amestoy et al. 2007; Dollar and Scott 2009]).

3.2 HSL_MA77

HSL_MA77 [Reid and Scott 2008, 2009b] is also a multifrontal solver designed to solve positive definite and indefinite sparse symmetric systems. It too has separate analyse, factorize, and solve phases. A fundamental difference between MA57 and HSL_MA77 is that the latter is an out-of-core solver, that is, it is designed to allow the matrix data, the computed factors, and some of the intermediate work arrays to be held in files. The advantage of this is that it enables much larger problems to be solved.

Storing data in files potentially adds a significant overhead to the time required to factor and then solve the linear system. To minimize this overhead, Reid and Scott [2009a] have written a set of Fortran subroutines that manage a virtual memory system so that actual input/output occurs only when really necessary. These routines are available within HSL as the package HSL_OF01 and are used by HSL_MA77 to efficiently handle all input and output. Results reported by Reid and Scott [2009b] illustrated the effectiveness of HSL_OF01 in minimizing the out-of-core overhead; they also presented results for problems that were too large for MA57 to solve on their test machine. On problems that MA57 is able to solve, the performance of HSL_MA77 is favorable: on some problems it is faster than MA57, while on others the converse is true. The main exception is the solve time. The factor L has to be read from file once for the forward substitution and then again (in reverse order) for the back substitution. This is independent of the number of right-hand sides. Thus, for a single right-hand side (or small number of right-hand sides), the solve phase of HSL_MA77 can be significantly more expensive than the corresponding phase of MA57, although for a large number of right-hand sides there is a smaller relative difference.

As with MA57, HSL_MA77 offers a range of options. These include allowing the files to be replaced by arrays (so that, if there is sufficient space, the data is all stored in main memory). The user can specify the initial sizes of these arrays and an overall limit on their total size. If an array is found to be too small, the code will continue using a combination of files and arrays. Another important option allows the user to specify whether he or she is running the code on a 32-bit or 64-bit architecture. On a 64-bit machine, it is possible to run problems with much larger frontal matrices (on a 32-bit machine, the maximum front size is limited to approximately 16,000).

We remark that HSL_MA77 requires the user to input the pivot order. This can be computed by calling MeTiS or the HSL package MC47 that implements an approximate minimum degree algorithm; alternatively, the analyse phase of MA57 may be used to select the pivot sequence. Each of these alternatives computes a pivot order of 1×1 pivots; 2×2 pivots may be selected during the factorization. In some cases, it may be advantageous to specify a tentative pivot sequence that includes 2×2 pivots; HSL_MA77 allows the user to do this. A pivot order that contains 2×2 pivots may be found using the package HSL_MC68, and this remains an area of active research.

3.3 Design of HSL_MA79

HSL_MA79 has been designed to provide a robust and efficient implementation of Algorithm 1 for both positive-definite and indefinite systems, using existing HSL packages as its main building blocks. In particular, it uses the direct solvers MA57 and HSL_MA77 and the implementation of FGMRES offered by MI15, together with the scaling packages MC30, MC64, and MC77. HSL_MA79 includes a range of options but it is not our intention to incorporate all possibilities available within the direct solvers MA57 and HSL_MA77. Instead, we have designed a general-purpose package that is straightforward to use and, by restricting the number of parameters that have to be set, we do not require the user to have a detailed knowledge and understanding of all the different components of the algorithms used.

The following procedures are available to the user.

(1) MA79_factor_solve accepts the matrix A, the right-hand sides $\boldsymbol{b}$, and the requested accuracy. Based on the matrix, it selects whether to use MA57 or HSL_MA77; by default, single precision is selected as the initial precision. The code then implements Algorithm 1. The matrix factorization is retained for further solves.

(2) MA79_refactor_solve uses the information returned from a previous call to MA79_factor_solve to reduce the time required to factorize and solve $A\boldsymbol{x} = \boldsymbol{b}$. The sparsity pattern of A must be unchanged since the call to MA79_factor_solve; only the numerical values of the entries of A and $\boldsymbol{b}$ may have changed. By default, the precision for the factorization is chosen based on that used by MA79_factor_solve. The matrix factorization is retained for further solves.

(3) MA79_solve uses the computed factors generated by MA79_factor_solve (or MA79_refactor_solve) to solve further systems $A\boldsymbol{x} = \boldsymbol{b}$. Multiple calls to MA79_solve may follow a call to MA79_factor_solve (or MA79_refactor_solve).

(4) `MA79_finalize` should be called after all other calls are complete for a problem. It deallocates the components of the derived data types and discards the matrix factors.

We now discuss the first three of the preceding procedures in more detail (the finalize routine needs no further explanation).

3.3.1 `MA79_factor_solve`. On the call to `MA79_factor_solve`, the user must supply the entries in the lower triangular part of A in compressed sparse column (CSC) format and may optionally supply a pivot order. If no pivot order is supplied, one is computed using the analyse phase of `MA57`. Statistics on the forthcoming factorization (such as the maximum frontsize, the number of flops, and the number of entries in the factor) are also computed. These are exact for the factorization phase of `MA57` if the problem is positive definite; otherwise, they are lower bounds for `MA57` and estimates of lower bounds for `HSL_MA77`. By default, the statistics are used to choose the direct solver. `MA57` is selected unless one or more of the following holds.

(1) It is not possible to allocate the arrays required by `MA57`. (We allow the user to specify a maximum amount of memory, and, if the predicted memory usage for `MA57` exceeds this, we use `HSL_MA77`.)
(2) The matrix is positive definite with a maximum frontsize greater than 1500.
(3) The matrix is not positive definite and the user-supplied pivot sequence includes 2×2 pivots.
(4) The user has chosen `HSL_MA77`.

The choice in list item (2) was made on the basis of numerical experimentation (see Reid and Scott [2009b]). The main motivation for selecting `MA57` as the default solver is that `HSL_MA77` is primarily designed as an out-of-core solver and this incurs an overhead (which may be significant if the problem is not very large). Furthermore, the process of refactorizing in double precision is also more expensive for `HSL_MA77` because it is necessary to reload the matrix data and repeat its analysis phase (this can be avoided for `MA57`).

At the start of the factorization with `MA57`, `HSL_MA79` allocates the required arrays based on the analyse statistics If larger work arrays are later needed because of delayed pivots, `HSL_MA79` attempts to use the stop and restart facility offered by `MA57` but, if there is insufficient memory to allocate sufficiently large arrays, `HSL_MA79` switches to `HSL_MA77`. This may add a significant extra cost as `MA77_analyse` must be called and the factorization completely restarted.

By default, `HSL_MA79` works in mixed precision following Algorithm 1 and its development through Section 2. The user may, however, choose to perform the whole computation in double precision. In this case, `HSL_MA79` provides a convenient interface to `MA57` and `HSL_MA77` (albeit without the full flexibility and options offered by each of these packages individually). This facilitates comparisons between mixed and double precision. Working in double precision throughout may be advisable for very ill-conditioned systems or for very large problems for which repeated calls to the solve routine `MA79_solve` are expected.

HSL_MA79 includes a number of scaling options, provided by the HSL packages MC30 (Curtis and Reid's [1972] method minimizing the sum of logarithms of the entries), MC64 (symmetrized scaling based on maximum matching by Duff and Koster [Duff and Koster 2001; Duff and Pralet 2005]), and MC77 using the 1- or ∞-norms (iterative process of simultaneous norm equilibration [Ruiz 2001]). The default is the ∞-norm equilibration scaling from MC77 because recent tests [Hogg and Scott 2008] on a large number of problems from practical applications have shown that, in general, this provides a good fast scaling.

HSL_MA79 offers complete control of the parameters in Algorithms 2 and 3, in addition to the ability to disable any particular method of recovering precision in Algorithm 1 (e.g., the user may specify that the use of iterative refinement is to be skipped). It also supports tuning of the major parameters affecting the performance of the factorization phase, such as the block size used by the dense linear algebra kernels that lie at the heart of the multifrontal algorithm.

By default, the requested accuracy is achieved when β given by (2) is less than a user-prescribed value γ. However, HSL_MA79 also allows the user to specify, using an optional subroutine argument on the call to MA79_factor_solve, an alternative measure of accuracy. If present, it must compute a function $\beta = f(A, \boldsymbol{x}, \boldsymbol{b}, \boldsymbol{r})$ that is compared to γ when testing for termination. For example, it may be used to test for the requested accuracy in the 2-norm or using a component-wise approach.

An important feature of MA79_factor_solve is that it returns detailed information on the solution process. This includes which solver was used and the precision, together with details of the matrix factorization (the number of entries in the factor, the maximum frontsize, the number of 2×2 pivots chosen, the numbers of negative and zero pivots) and information on the refinement (the number of steps of iterative refinement, the number of FGMRES iterations performed, and the total number of calls to the solution phase of MA57 or HSL_MA77). In addition, the full information type or array from the factorization code (MA57B or MA77_factor) is returned to the user.

We note that the user can pass any number of right-hand sides b to MA79_factor_solve. In particular, the user can set the number of right-hand sides to zero. In this case, the routine will only perform the matrix factorization in the requested (or default) precision.

3.3.2 MA79_refactor_solve. We envisage that a user may want to factorize and solve a series of problems with the same sparsity pattern as the original matrix A but different numerical values. In this case, HSL_MA79 takes advantage both of the pivot sequence used within MA79_factor_solve and of the experience gained on the initial factorization.

On a call to MA79_refactor_solve, the user must input the values of the entries in both the lower and upper triangular parts of the new matrix in CSC format, with the entries in each column in order of increasing row index. This format (which is the format the original matrix is returned to the user in on exit from MA79_factor_solve) is required so that HSL_MA79 can avoid, before the factorization begins, taking and manipulating additional copies of the matrix (for large problems, this avoidance is important). Having the matrix in this form also has the side benefit of allowing a more efficient matrix-vector product.

3.3.3 MA79_solve. After a call to MA79_factor_solve, MA79_solve may be called to solve for additional right-hand sides. If MA57 has been used (or HSL_MA77 was run in-core), the cost of each additional solve is generally small but, if HSL_MA77 was run with the factors held on disk, the solve time can be significant (see Reid and Scott [2009b]). If the solve is performed at the same time as the factorization, the entries of L are used to perform the forward substitution as they are generated, cutting the amount of data that must be read (and hence the time) for the solve approximately in half.

3.4 Errors and Warnings

HSL_MA79 issues two levels of errors: fatal errors that cause the computation to terminate immediately and warnings that are intended to alert the user to what could be a problem but which will not prevent the computation from continuing. In either case, a flag is set (with a negative value for an error and a positive value for a warning) and a message is optionally printed (the user controls the level of diagnostic printing). Examples of fatal errors include a user-supplied pivot order that is not a permutation and calls to routines within the HSL_MA79 package that are out of sequence.

A warning is issued if the user-supplied matrix data contains out-of-range indices (these are ignored) or duplicated indices (the corresponding matrix entries are summed). A warning is also issued if the matrix is found to be singular or if MA57 was requested but HSL_MA77 is used because of insufficient main memory. Note that more than one warning may be issued. At the end of the computation, a warning is given if the requested accuracy was not obtained after all allowable fall-back options were attempted. In particular, if the factorization has been performed in single precision, the requested accuracy may not be achieved on a call to MA79_solve without resorting to a double-precision factorization. In this case, because this cannot occur within MA79_solve, the user should call MA79_refactor_solve, explicitly specifying the factorization is to be performed in double precision.

Full details of errors and warnings and of the levels of diagnostic printing are given in the user documentation for HSL_MA79.

4. NUMERICAL RESULTS

In this section, we present results obtained using Version 1.1.0 of HSL_MA79. This uses Version 3.2.0 of MA57 and Version 4.0.0 of HSL_MA77. Our experiments were performed on the machine and test examples described in Section 1. The requested accuracy was $\beta < 5 \times 10^{-15}$ and, unless stated otherwise, we used the default settings for all the HSL_MA79 parameters (in particular, the parameters chosen in Section 2 are used to control the solution recovery).

Figures 3 and 4 compare the performance of mixed-precision and double precision for Test Sets 1 and 2, respectively, with MA57 selected as the direct solver within HSL_MA79 for Test Set 1 and HSL_MA77 for Test Set 2. If the requested

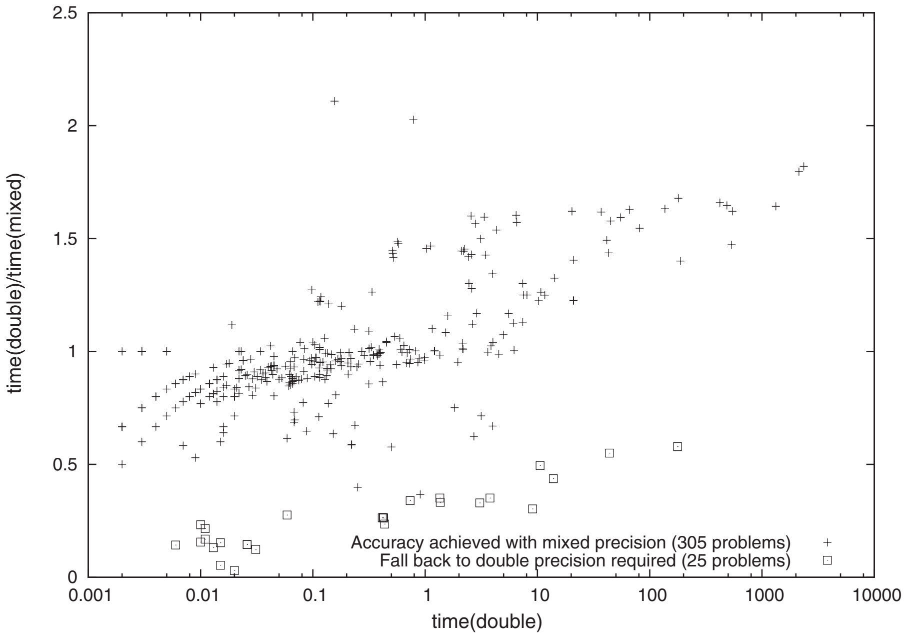

Fig. 3. Ratio of times to solve Equation (1) with HSL_MA79 in mixed-precision and double-precision modes on Test Set 1 using MA57 as the solver, with accuracy $\gamma = 5 \times 10^{-15}$.

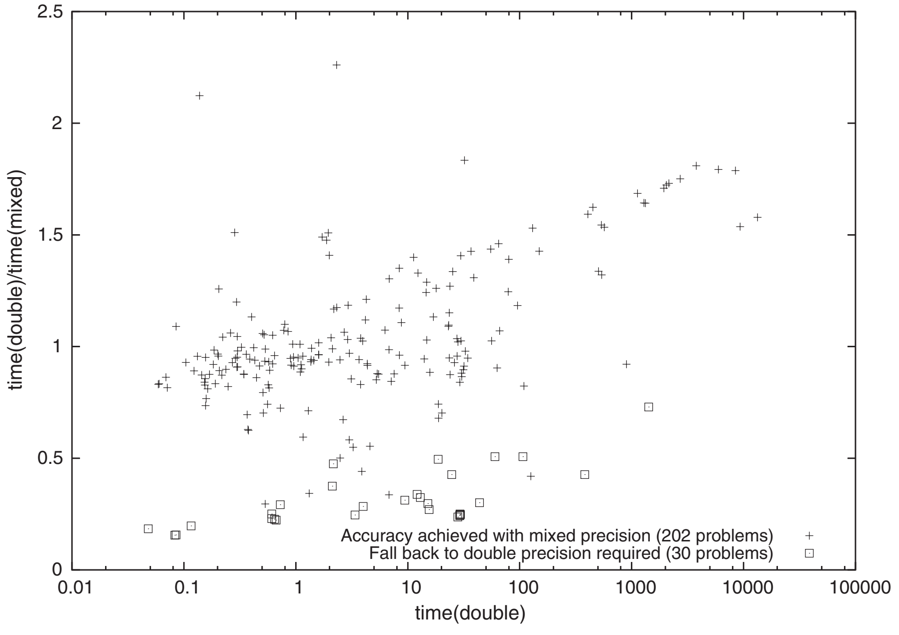

Fig. 4. Ratio of times to solve Equation (1) with HSL_MA79 in mixed-precision and double-precision modes on Test Set 2 using HSL_MA77 as the solver, with accuracy $\gamma = 5 \times 10^{-15}$.

Table V. Number of Problems That Exited at Each Stage of Algorithm 1 Implemented as HSL_MA79

	Test Set 1 MA57		Test Set 2 HSL_MA77	
After iterative refinement	265	81%	157	68%
After FGMRES	40	12%	45	19%
After fall-back to double-precision	24	7%	28	12%
Failed	1	<1%	2	1%

accuracy was only achieved by falling back on a double-precision factorization, the mixed-precision time included the double-precision factorization time. From Figure 3, we see that, if the time taken by HSL_MA79 in double precision is less than about 1 s, there is generally little or no advantage in terms of computational time to using mixed precision (in fact, for a number of problems, running in double precision is almost twice as fast as using mixed precision). However, for the larger problems within Test Set 1, mixed precision outperformed double precision by more 50%. For the problems in Test Set 2 with the out-of-core solver HSL_MA77, on our test machine mixed precision is only recommended if the double-precision time is greater than about 10 s. For problems that run more rapidly than this, the savings from the single-precision factorization are not large enough to offset the cost of the additional solves (which, in this case, involve reading data from disk). Of course, if the user is prepared to accept a less accurate solution (i.e., the requested accuracy γ is chosen to be greater than 5×10^{-15}), this will affect the balance between the mixed-precision time (which will decrease as fewer refinement steps will be needed) and the double-precision time (which, in many instances, will be unchanged).

In Table V, we report the number and percentage of problems in each test set for which the requested accuracy was achieved after iterative refinement and after iterative refinement followed by FGMRES. We also report the number of problems that had to fall back to a double-precision factorization to achieve the requested accuracy and the number that failed to achieve this even in double precision. There were just two such problems: GHS_indef/boyd1 and GHS_indef/blockqp1 (these had final scaled residuals of 6.2×10^{-14} and 2.9×10^{-14}, respectively).

As expected, when mixed precision failed to reach the requested accuracy, HSL_MA79 spent longer establishing this fact than if double precision was used originally. Thus it is essential for a potential user to experiment to see whether the mixed-precision approach will be advantageous for his or her application and computing environment.

It is of interest to consider not only the total time taken to solve the system (1), but also the times for each phase of the solution process in mixed precision and in double precision. Table VI reports timings for the various phases for a subset of problems of different sizes from Test Set 2. The problems are ordered by the total time required to solve (1) using double precision. The time for the analyze phase (which here includes the time to scale the matrix) is independent of the precision.

Table VI. Times to Solve Equation (1) for Subset of Problems from Test Set 2 with `HSL_MA79` Using `HSL_MA77` as Solver. (m denotes mixed precision and d denotes double precision. The numbers in parentheses are iteration counts. - indicates iterative refinement (or FGMRES) was not required.)

	Total		Analyze	Factorize		Iterative refinement		FGMRES		β	
Problem	m	d		m	d	m	d	m	d	m	d
`HB/bcsstk17`	0.30	0.25	0.10	0.11	0.14	0.06 (5)	-	-	-	1.17e-15	1.24e-16
`Boeing/crystm03`	1.08	1.09	0.64	0.33	0.44	0.09 (3)	-	-	-	3.78e-16	4.80e-16
`Boeing/bcsstm39`	3.86	3.97	3.66	0.11	0.30	0.09 (2)	-	-	-	2.31e-15	1.28e-16
`Rothberg/cfd1`	6.22	8.41	2.98	2.48	5.43	0.69 (3)	-	-	-	1.41e-16	2.32e-16
`Rothberg/cfd2`	11.8	14.6	4.97	5.27	9.62	1.39 (3)	-	-	-	1.69e-16	2.74e-16
`INPRO/msdoor`	28.7	20.2	5.39	9.64	11.8	2.70 (3)	-	10.1 (8)	-	2.96e-16	2.64e-16
`ND/nd6k`	29.7	38.8	5.33	20.3	33.1	3.77 (8)	-	-	-	3.67e-15	1.68e-15
`GHS_psdef/apache2`	61.8	66.1	19.8	29.1	46.4	12.5 (6)	-	-	-	1.79e-16	1.43e-15
`Koutsovasilis/F1`	63.5	79.1	16.9	36.1	60.1	9.10 (4)	-	-	-	1.75e-15	2.16e-16
`Lin/Lin`	57.4	79.9	10.8	39.3	66.4	7.26 (5)	2.69 (1)	-	-	3.20e-16	2.03e-16
`ND/nd12k`	104	149	14.5	78.0	134	11.5 (8)	-	-	-	1.00e-15	2.07e-15
`PARSEC/Ga3As3H12`	348	537	21.6	302	511	24.30 (8)	5.86 (1)	-	-	3.27e-15	3.37e-16
`ND/nd24k`	372	570	36.5	297	532	36.5 (9)	-	-	-	1.02e-15	2.94e-15
`GHS_indef/sparsine`	409	540	18.4	314	517	5.28 (2)	4.92 (1)	70.5 (16)	-	4.20e-16	3.51e-16
`PARSEC/Si34H36`	783	1287	38.7	706	1236	37.7 (6)	12.1 (1)	-	-	4.91e-16	2.71e-16
`GHS_psdef/audikw_1`	810	1329	65.7	620	1262	120 (7)	-	-	-	4.67e-15	5.84e-17
`PARSEC/Ga10As10H30`	1187	2046	51.6	1082	1976	53.3 (6)	18.7 (1)	-	-	2.96e-16	1.96e-16
`PARSEC/Si87H76`	6058	9310	153	4703	8815	1202 (7)	344 (1)	-	-	6.64e-16	1.95e-16

5. CONCLUSIONS AND FUTURE DIRECTIONS

In this article, we have explored a mixed-precision strategy that is capable of outperforming a traditional double-precision approach for solving large sparse symmetric linear systems. Building on the recent work of Arioli and Duff [2008] and Buttari et al. [2008], we have designed and developed a practical and robust sparse mixed-precision solver; the new package HSL_MA79 is available within the HSL Library. Numerical experiments on a large number of problems have shown that, in about 90% of our test cases, it was possible to use a mixed-precision approach to get accuracy of 5×10^{-15}; in the remaining cases, it was necessary to resort to computing a double-precision factorization (or to accept a less accurate solution). HSL_MA79 is designed to allow an automatic fall back to double precision and is tuned to minimize the work performed before this happens. However, although we have demonstrated robustness, our experience is that, in terms of computational time, the advantage of using mixed precision is limited to large problems (how large will depend on the direct solver used within HSL_MA79, on the computing platform, and also on the requested accuracy); the user is advised that experimentation with his or her problems will be necessary to decide whether or not to use mixed precision.

Future work on HSL_MA79 will focus on more efficiently recovering double-precision accuracy in the case of multiple right-hand sides; this will lead to the replacement of the MI15 implementation of FGMRES with a specially modified variant of FGMRES, which may require a different adaptive restarting strategy. Our current implementation of HSL_MA79 requires that the original matrix A reside in memory throughout the factorization. This is not a requirement for the out-of-core solver HSL_MA77 and we would like to remove it from HSL_MA79 by offering an interface that allows A to be supplied by the user in a file. We would additionally like to collect more large problems to test and refine our code on.

Throughout this article, we have considered factorizing A in single precision combined with recovery using double precision. However, this does not have to be the case. Provided the condition number of the matrix is less than the reciprocal of the requested accuracy γ, the theory [Arioli and Duff 2008] supports recovery to arbitrary precision. In this case, the refinement must be carried out in extended precision. It is also possible to perform a factorization in double precision and then recover to higher precision. This will be the subject of a separate study.

6. CODE AVAILABILITY

All the codes discussed in this article have been developed for inclusion in the mathematical software library HSL. All use of HSL requires a license. Individual HSL packages (together with their dependencies and accompanying documentation) are available without charge to individual academic users for their personal (noncommercial) research and for teaching; licenses for other uses involve a fee. Details of the packages and how to obtain a license plus conditions of use are available online.[2]

[2] www.cse.stfc.ac.uk/nag/hsl.

ACKNOWLEDGMENTS

We are grateful to our colleagues John Reid, Iain Duff, and Mario Arioli for useful discussions relating to this work. We would also like to thank three anonymous referees for their constructive and helpful comments.

REFERENCES

Amestoy, P., Davis, T., and Duff, I. 1996. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl. 17*, 886–905.

Amestoy, P., Davis, T., and Duff, I. 2004. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30*, 3, 381–388.

Amestoy, P., Dollar, H., Reid, J., and Scott, J. 2007. An approximate minimum degree algorithm for matrices with dense rows. Tech. rep. RAL-TR-2007-020. Rutherford Appleton Laboratory, Chilton, U.K.

Arioli, M. and Duff, I. 2008. FGMRES to obtain backward stability in mixed precision. Tech. rep. RAL-TR-2008-006. Rutherford Appleton Laboratory, Chilton, U.K.

Arioli, M., Duff, I., Gratton, S., and Pralet, S. 2007. A note on GMRES preconditioned by a perturbed LDL^T decomposition with static pivoting. *SIAM J. Sci. Comput. 29*, 5, 2024–2044.

Buttari, A., Dongarra, J., and Kurzak, J. 2007a. Limitations of the playstation 3 for high performance cluster computing. Tech. rep. CS-07-594. University of Tennessee Computer Science Department, Knokville, TN.

Buttari, A., Dongarra, J., Kurzak, J., Luszczek, P., and Tomov, S. 2008. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Softw. 34*, 4.

Buttari, A., Dongarra, J., Langou, J., Langou, J., Luszczek, P., and Kurzak, J. 2007b. Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. J. High Perform. Comput. Appl. 21*, 4, 457–466.

Curtis, A. and Reid, J. 1972. On the automatic scaling of matrices for Gaussian elimination. *IMA J. Appl. Math. 10*, 1, 118–124.

Davis, T. 2007. The University of Florida sparse matrix collection. Technical rep. University of Florida, Gaisesville, FL. `http://www.cise.ufl.edu/~davis/techreports/matrices.pdf`.

Demmel, J., Hida, Y., Kahan, W., Li, X. S., Mukherjeek, S., and Riedy, E. J. 2006. Error bounds from extra precise iterative refinement. *ACM Trans. Math. Softw. 32*, 2, 325–351.

Demmel, J., Hida, Y., Li, X. S., and Riedy, E. J. 2009. Extra-precise iterative refinement for overdetermined least squares problems. *ACM Trans. Math. Softw. 35*, 4. (Also LAPACK Working Note 188.)

Dollar, H. and Scott, J. 2009. A note on fast approximate minimum degree orderings for matrices with some dense rows. *Numer. Lin. Alg. Appl. 17*, 1, 43–45.

Dongarra, J., Croz, J. D., Hammarling, S., and Duff, I. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 1, 1–17.

Duff, I. 2004. MA57—a new code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Math. Softw. 30*, 118–154.

Duff, I. and Koster, J. 2001. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl. 22*, 4, 973–996.

Duff, I. and Pralet, S. 2005. Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM J. Matrix Anal. Appl. 27*, 313–340.

Duff, I. and Pralet, S. 2007. Towards a stable static pivoting strategy for the sequential and parallel solution of sparse symmetric indefinite systems. *SIAM J. Matrix Anal. Appl. 29*, 1007–1024.

Duff, I. and Reid, J. 1983. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw. 9*, 302–325.

Duff, I. and Scott, J. 2005. Towards an automatic ordering for a symmetric sparse direct solver. Tech. rep. RAL-TR-2006-001. Rutherford Appleton Laboratory, Chilton, U.K.

Goto, K. and van de Geijn, R. 2008. High performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw. 35*, 1.

GOULD, N., SCOTT, J., AND HU, Y. 2007. A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Trans. Math. Softw. 33*, 2, 10:1–10:32.

GRAHAM, S. L., SNIR, M., AND PATTERSON, C. A. 2004. *Getting Up to Speed: The Future of Supercomputing*. National Academies Press, Washington, DC.

HIGHAM, N. 1997. Iterative refinement for linear systems and LAPACK. *IMA J. Numer. Anal. 17*, 495–509.

HOGG, J. D. AND SCOTT, J. A. 2008. The effects of scalings on the performance of a sparse symmetric indefinite solver. Tech. rep. RAL-TR-2008-007. Rutherford Appleton Laboratory, Chilton, U.K.

HSL. 2007. A collection of Fortran codes for large-scale scientific computation. `http://www.cse.stfc.ac.uk/nag/hsl`.

IEEE. 2008. IEEE standard for floating-point arithmetic. IEEE Standard 754-2008. IEEE Standards Committee, Piscataway, NJ, 1–58.

KARYPIS, G. AND KUMAR, V. 1999. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 359–392.

REID, J. AND SCOTT, J. 2008. An efficient out-of-core sparse symmetric indefinite direct solver. Tech. rep. RAL-TR-2008-024. Rutherford Appleton Laboratory, Chilton, U.K.

REID, J. AND SCOTT, J. 2009a. Algorithm 891: A Fortran virtual memory system. *ACM Trans. Math. Softw. 36*, 1.

REID, J. AND SCOTT, J. 2009b. An out-of-core sparse Cholesky solver. *ACM Trans. Math. Softw. 36*, 2.

RUIZ, D. 2001. A scaling algorithm to equilibrate both rows and columns norms in matrices. Tech. rep. RAL-TR-2001-034. Rutherford Appleton Laboratory, Chilton, U.K.

SAAD, Y. 1994. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Statist. Comput. 14*, 461–469.

SAAD, Y. 2003. *Iterative Methods for Sparse Linear Systems*, 2nd ed. SIAM, Philadelphia, PA, 273–275.

SKEEL, R. 1980. Iterative refinement implies numerical stability for Gaussian elimination. *Math. Computat. 35*, 817–832.

Received September 2008; revised March 2009, July 2009; accepted October 2009