USING MULTIPRECISONARRAYS.JL: ITERATIVE REFINEMENT IN JULIA

C. T. KELLEY*

Abstract. MultiPrecisionArrays.jl is a Julia package. This package provides data structures and solvers for several variants of iterative refinement. It will become much more useful when half precision (aka Float16) is fully supported in LAPACK/BLAS. For now, its only general-purpose application is classical iterative refinement with double precision equations and single precision factorizations.

It is useful as it stands for people doing research in iterative refinement. We provide a half precision LU factorization that, while far from optimal, is much better than the default in Julia.

Key words. Interative Refinement, Mixed-Precision Arithmetic, Interprecision Transfers, Julia

AMS subject classifications. 65F05, 65F10, 45B05, 45G10,

1. Introduction. The Julia [1] package **MultiPrecisionArrays.jl** provides data structures and algorithms for several variations of iterative refinement (IR). In this introductory section we look at the classic version of iterative refinement and discuss its implementation.

IR is a perfect example of a storage/time tradeoff. To solve a linear system $\mathbf{Ax} = \mathbf{b}$ in R^N with IR, one incurs the storage penalty of making a low precision copy of $\mathbf{A}$ and reaps the benefit of only having to factor the low precision copy.

In most of this paper we consider IR using two precisions, which we will call high and low. In a typical use case, high will be double and low will be single. We will make precision and inter precision transfers explicit in our algorithmic descriptions.

The first three sections of this paper use **MultiPrecisionArrays.jl** to generate tables which compare the algorithmic options, but do not talk about using Julia.

Algorithm 1 is the textbook version [5] version of the algorithm for the LU factorization.

IR$(\mathbf{A}, \mathbf{b})$
  $\mathbf{x} = 0$
  $\mathbf{r} = \mathbf{b}$
  Factor $\mathbf{A} = \mathbf{LU}$ in low precision
  while $\|\mathbf{r}\|$ too large **do**
    $\mathbf{d} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{r}$
    $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{d}$
    $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$
  end while

One must be clear on the meanings of "factor in low precision" and $\mathbf{d} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{r}$ to implement the algorithm. As we indicated above, the only way to factor $\mathbf{A}$ in low precision is to make a copy and factor that copy. We must introduce some notation for that. We let $\mathcal{F}_p$ be the set of floating point numbers in precision p, u_p the unit roundoff in that precision, and fl_p the rounding operator. Similarly we let $\mathcal{F}_p^N$, $\mathcal{F}_p^{N\times N}$ denote the vectors and matrices in precision p. We let I_p^q denote the copying operator from precision p to precision q. When we are not explicitly specifying the precisions, we will use H and L as sub and superscripts for high and low precision. When we are discussing specific use cases out sub and superscripts will the d, s, and h for double, single, and half precision.

So, factoring a high-precision matrix $\mathbf{A} \in \mathcal{F}_H^{N\times N}$ in low precision L means copy $\mathbf{A}$ into low precision and obtain

$$\mathbf{A}_L = I_H^L(\mathbf{A})$$

and then factor $\mathbf{A}_L = \mathbf{LU}$.

*North Carolina State University, Department of Mathematics, Box 8205, Raleigh, NC 27695-8205, USA (Tim_Kelley@ncsu.edu). This work was partially supported by Department of Energy grant DE-NA003967.

1.1. Terminating the while loop. We terminate the loop when

$$\|\mathbf{r}\| < \tau\|\mathbf{b}\| \tag{1.1}$$

where we use $\tau = 10 * eps(TH)$. Here eps(TH) is high precision machine epsilon. The problem with this criterion is that IR can stagnate, especially for ill-conditioned problems, before the termination criterion is attained. We detect stagnation by looking for a unacceptable decrease (or increase) in the residual norm. So we will terminate the iteration if

$$\|\mathbf{r}_{new}\| \geq .9\|\mathbf{r}_{old}\| \tag{1.2}$$

even if (1.1) is not satisfied.

In this paper we count iterations as residual computations. This means that the minimum number of iterations will be two. Since we begin with $\mathbf{x} = 0$ and $\mathbf{r} = \mathbf{b}$, the first iteration computes $\mathbf{d} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{b}$ and then $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{d}$, so the first iteration is the output of a low precision solve. We will need at most one more iteration to get a meaningful residual reduction.

1.2. Interprecision Transfers: Part I. The meaning of $\mathbf{d} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{r}$ is more subtle. The problem is that the factors $\mathbf{U}$ and $\mathbf{L}$ are store in low precision and $\mathbf{r}$ is a high precision vector. LAPACK will convert $\mathbf{L}$ and $\mathbf{U}$ to the higher precision "on the fly" with each mixed precision binary operation at a cost of $O(N^2)$ interprecision transfers. The best way to understand this is to recall that if $a, b \in \mathcal{F}_H$ and $c \in \mathcal{F}_L$ that

$$fl_H(a * c + b) = fl_H(a + I_L^H(c) + b).$$

As we will see this interprecision transfer can have a meaningful cost even though the factorization will dominate with $O(N^3)$ work.

One can eliminate the cost by copying $\mathbf{r}$ into low precision, doing the triangular solves in low precision, and then mapping the result into high precision. The two approaches are not the same. To see this we $\mathbf{x}_c$ denote the current iterate and $\mathbf{x}_+$ the new iterate.

If one does the solves on the fly then the IR iteration

$$\begin{aligned}
\mathbf{x}_+ &= \mathbf{x}_c + \mathbf{d} = \mathbf{x}_c + \hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}\mathbf{r} \\
&= \mathbf{x}_c + \hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}_c) \\
&= (\mathbf{I} - \hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}\mathbf{A})\mathbf{x}_c + \hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}\mathbf{b}
\end{aligned}$$

is a linear stationary iterative method. Hence on the fly IR will converge if the spectral radius of the iteration matrix

$$\mathbf{M}_{IR} = \mathbf{I} - \hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}\mathbf{A}$$

is less than one. We will refer to the on the fly approach as mixed precision solves (MPS) when we report computational results in § A.

On the other hand, if one does the triangular solves in low precision, one must first take care to scale $\mathbf{r}$ to avoid underflow, so one solves

$$(\mathbf{LU})\mathbf{d}_L = I_H^L(\mathbf{r}/\|\mathbf{r}\|) \tag{1.3}$$

in low precision and then promotes $\mathbf{d}$ to high precision and reverses the scaling to obtain

$$\mathbf{d} = \|\mathbf{r}\| I_L^H(\mathbf{d}_L). \tag{1.4}$$

We will refer to this approach as low precision solves (LPS) when we report computational results in § A.

2. Integral Equations Example. The submodule **MultiPrecisionArrays.Examples** has an example which we will use repeatedly. The function **Gmat(N)** returns the N point trapezoid rule discretization of the Greens operator for $-d^2/dx^2$ on $[0,1]$

$$Gu(x) = \int_0^1 g(x,y)u(y)\,dy$$

where

$$g(x,y) = \begin{cases} y(1-x); & x > y \\ x(1-y); & x \le y \end{cases}$$

The eigenvalues of G are $1/(n^2\pi^2)$ for $n = 1, 2, \ldots$.

The code for this is in the /**src**/**Examples** directory. The file is **Gmat.jl**.

In the examples we will use **Gmat** to build a matrix $\mathbf{A} = I - \alpha\mathbf{G}$. In the examples we will use $\alpha = 1.0$, a very well conditioned case, and $\alpha = 800.0$. This latter case is very near singularity.

The GitHub repository for **MultiPrecisionArrays.jl** has a directory for the Julia functions we use to make the tables and plots in this paper. That directory **Codes_For_Docs** is not a subdirectory of /**src** because it is not part of the solvers and we do not do unit testing on the files in that directory.

2.1. Classic Example: Double-Single Precision. While **MultiPrecisionArrays.jl** was designed for research, it is useful in applications in the classic case where high precision is double and low is single. This case avoids the (very interesting) problems with half precision.

Here is a Julia code that implements IR in this case. We will use this as motivation for the data structures in **MultiPrecisionArrays.jl** .

```
"""
IR(A,b)
Simple minded iterative refinement
Solve Ax=b
"""
function IR(A, b)
    x = zeros(length(b))
    r = copy(b)
    tol = 10.0 * eps(Float64)
    #
    # Allocate a single precision copy of A and factor in place
    #
    A32 = Float32.(A)
    AF = lu!(A32)
    #
    # Give IR at most ten iterations, which it should not need
    # in this case
    #
    itcount = 0
    rnorm=norm(r)
    rnormold = 2.0*rnorm
    while (rnorm > tol * norm(b)) && (rnorm < .9 * rnormold)
        #
        # Store r and d = AF\r in the same place.
        #
        ldiv!(AF, r)
        x .+= r
        r .= b - A * x
        rnorm=norm(r)
        itcount += 1
    end
    return x
end
```

2.2. Running MultiprecisionArrays: I. The function **IR** allocates memory for the low precision matrix and the residual with each call. **MultiPrecisionArrays.jl** addresses that with the **MPArray** data structure which allocates for the low precision copy of **A** and the residual **r**.

We then use the function **mplu!** to factor the low precision copy and then **mpgeslir** executes the IR loop. We have overloaded the backslash for the linear solve \ with a call to **mpgeslir**.

We do this with the **MPArray** structure.

```
struct MPArray{TH<:AbstractFloat,TL<:AbstractFloat}
    AH::Array{TH,2}
    AL::Array{TL,2}
    residual::Vector{TH}
end
```

The constructor for a double precision matrix **AH** will use a single precision **AL**.

```
function MPArray(AH::Array{Float64,2}; TL = Float32, onthefly=false)
    AL = TL.(AH)
    (m,n)=size(AH); res=ones(eltype(AH),n)
    onthefly ?  MPA = MPEArray(AH, AL, res) : MPA = MPArray(AH, AL, res)
end
```

The cost of the constructor is the allocation of a low precision copy of the matrix.

All IR needs after you construct the MPArray is a factorization of the low precision matrix. The **mplu!** function does that and returns a factorization object.

As an example we will solve the integral equation with both double precision LU and an MPArray and compare execution time and the quality of the results. We will use the function **@belapsed** from the **BenchmarkTools.jl** to get timings. The problem setup is pretty simple

```
julia> using MultiPrecisionArrays

julia> using BenchmarkTools

julia> using MultiPrecisionArrays.Examples

julia> N=4096; G=Gmat(N); A=I - G; x=ones(N); b=A*x;

julia> @belapsed lu!(AC) setup=(AC=copy($A))
1.43148e-01
```

At this point we have timed **lu!**. The next step is to construct an MPArray and factor the low precision matrix. We use the constructor **MPArray** to store **A**, the low precision copy, and the residual. Then apply the function **mplu!** to factor the low precision copy in place.

```
julia> MPA=MPArray(A);

julia> @belapsed mplu!(MPAC) setup=(MPAC=deepcopy($MPA))
8.02158e-02
```

So the single precision factorization is roughly half the cost of the double precision one.

Now for the solves. Both **lu!** and **mplu!** produce a Julia factorization object and \ works with both. You have to be a bit careful because MPA and A share storage. So I will use **lu** instead of **lu!** when factoring **A**.

```
julia> AF=lu(A); xf = AF\b;

julia> MPAF=mplu!(MPA); xmp=MPAF\b;

julia> luError=norm(xf-x,Inf); MPError=norm(xmp-x,Inf);

julia> println(luError, "  ", MPError)
7.41629e-14  8.88178e-16
```

So the relative errors are equally good. Now look at the residuals.

```
julia> luRes=norm(A*xf-b,Inf)/norm(b,Inf); MPRes=norm(A*xmp-b,Inf)/norm(b,Inf);

julia> println(luRes,"  ",MPRes)
7.40609e-14  1.33243e-15
```

So, for this well-conditioned problem, IR reduces the factorization cost by a factor of two and produces results as good as LU on the double precision matrix. Even so, we should not forget the storage cost of the single precision copy of **A**.

3. Half Precision and GMRES-IR. Using half precision (`Float16`) will not speed up the solver, in fact it will make the solver slower. The reason for this is that LAPACK and the BLAS do not (**YET** [4]) support half precision, so all the clever stuff in there is missing. We provide a half precision LU factorization /**src**/**Factorizations**/**hlu!.jl** that is better than nothing. It's a hack of Julia's `generic_lu!` with threading and a couple compiler directives. Even so, it's 2 – 5 times **slower** than a double precision LU. Half precision support is coming [4] and Julia and Apple support it in hardware. For now, at least for desktop computing, half precision is for research in iterative refinement, not applications.

Here's a table (created with /**Code_For_Docs**/**HalfTime.jl**) that illustrates the point. In the table we compare timings for LAPACK's LU to the LU we compute with `hlu!.jl`. The matrix is $\mathbf{I} - \mathbf{G}$.

TABLE 3.1
Half precision is slow: LU timings

N	Double	Single	Half	Ratio
1024	4.02e-03	3.24e-03	5.24e-03	1.31e+00
2048	2.27e-02	1.41e-02	3.72e-02	1.64e+00
4096	1.56e-01	8.52e-02	2.55e-01	1.63e+00
8192	1.15e+00	6.03e-01	4.36e+00	3.77e+00

The columns of the table are the dimension of the problem, timings for double, single, and half precision, and the ratio of the half precision timings to double. The timings came from Julia 1.10-beta2 running on an Apple M2 Pro with 8 performance cores.

Half precision is also difficult to use properly. The low precision can make iterative refinement fail because the half precision factorization can have a large error. Here is an example to illustrate this point. The matrix here is modestly ill-conditioned and you can see that in the error from a direct solve in double precision.

```
julia> A=I - 800.0*G;

julia> x=ones(N);

julia> b=A*x;

julia> xd=A\b;

julia> norm(b-A*xd,Inf)
6.96332e-13

julia> norm(xd-x,Inf)
2.30371e-12
```

Now, if we downcast things to half precision, nothing good happens.

```
julia> AH=Float16.(A);

julia> AHF=hlu!(AH);

julia> z=AHF\b;

julia> norm(b-A*z,Inf)
6.25650e-01
```

```
julia> norm(z-xd,Inf)
2.34975e-01
```

So you get very poor, but unsurprising, results. While MultiPrecisionArrays.jl supports half precision and I use it all the time, it is not something you would use in your own work without looking at the literature and making certain you are prepared for strange results. Getting good results consistently from half precision is an active research area.

So, it should not be a surprise that IR also struggles with half precision. We will illustrate this with one simple example. In this example high precision will be single and low will be half. Using **MPArray** with a single precision matrix will automatically make the low precision matrix half precision.

```
julia> N=4096; G=800.0*Gmat(N); A=I - Float32.(G);

julia> x=ones(Float32,N); b=A*x;

julia> MPA=MPArray(A); MPF=mplu!(MPA; onthefly=false);

julia> y=MPF\b;

julia> norm(b - A*y,Inf)
1.05272e+02
```

So, IR completely failed for this example. We will show how to extract the details of the iteration in a later section.

It is also worthwhile to see if doing the triangular solves on-the-fly (MPS) helps.

```
julia> MPB=MPArray(A; onthefly=true); MPBF=mplu!(MPB);

julia> z=MPBF\b;

julia> norm(b-A*z,Inf)
1.28174e-03
```

So, MPS is better in the half precision case. Moreover, it is also less costly thanks to the limited support for half precision computing. For that reason, MPS is the default when high precision is single.

However, on-the-fly solves are not enough to get good results and IR still terminates too soon.

3.1. GMRES-IR. GMRES-IR [2,3] solves the correction equation with a preconditioned GMRES [11] iteration. One way to think of this is that the solve in the IR loop is an approximate solver for the correction equation

$$\mathbf{Ad} = \mathbf{r}$$

where one replaces $\mathbf{A}$ with the low precision factors $\mathbf{LU}$. In GMRES-IR one solves the correction equation with a left-preconditioned GMRES iteration using $\mathbf{U}^{-1}\mathbf{L}^{-1}$ as the preconditioner. The preconditioned equation is

$$\mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{Ad} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{r}.$$

GMRES-IR will not be as efficient as IR because each iteration is itself an GMRES iteration and application of the preconditioned matrix-vector product has the same cost (solve + high precision matrix vector product) as a single IR iteration. However, if low precision is half, this approach can recover the residual norm one would get from a successful IR iteration.

There is also a storage problem. One should allocate storage for the Krylov basis vectors and other vectors that GMRES needs internally. We do that in the factorization phase. So the structure **MPGEFact** has the factorization of the low precision matrix, the residual, the Krylov basis and some other vectors needed in the solve.

Here is a well conditioned example. Both IR and GMRES-IR perform well, with GMRES-IR taking significantly more time. Note that I cannot use the same multiprecision array for both factorizations because

the data for the low precision factorization would be overwritten. So I use the **deepcopy** command from Julia **before** factoring either multiprecision array.

```
julia> using MultiPrecisionArrays

julia> using MultiPrecisionArrays.Examples

julia> using BenchmarkTools

julia> N=4069; AD= I - Gmat(N); A=Float32.(AD); x=ones(Float32,N); b=A*x;

julia> MPA=MPArray(A); MPA2=deepcopy(MPA);

julia> MPF=mplu!(MPA); MPF2=mpglu!(MPA2);

julia> # build two MPArrays and factor them for IR or GMRES-IR

julia> z=MPF\b; y=MPF2\b; println(norm(z-x,Inf),"  ",norm(y-x,Inf))
5.9604645e-7  2.9802322e-7

julia> # and the relative residuals look good, too

julia> println(norm(b-A*z,Inf)/norm(b,Inf),"  ",norm(b-A*y,Inf)/norm(b,Inf))
4.768957e-7  3.5767178e-7

julia> @btime $MPF\$b;
  13.582 ms (4 allocations: 24.33 KiB)

julia> @btime $MPF2\$b;
  52.020 ms (223 allocations: 92.84 KiB)
```

If you dig into the iteration statistics (more on that later) you will see that the GMRES-IR iteration took almost exactly four times as many solves and residual computations as the simple IR solve.

We will repeat this experiment on the ill-conditioned example. In this example, as we saw earlier, IR fails to converge.

```
julia> N=4069; AD= I - 800.0*Gmat(N); A=Float32.(AD); x=ones(Float32,N); b=A*x;

julia> MPA=MPArray(A); MPA2=deepcopy(MPA);

julia> MPF=mplu!(MPA); MPF2=mpglu!(MPA2);

julia> z=MPF\b; y=MPF2\b; println(norm(z-x,Inf),"  ",norm(y-x,Inf))
0.2875508  0.004160166

julia> println(norm(b-A*z,Inf)/norm(b,Inf),"  ",norm(b-A*y,Inf)/norm(b,Inf))
0.0012593127  7.937655e-6
```

So, the relative error and relative residual norm for GMRES-IR is much smaller than that for IR.

4. Literature. The important references are [2,3,5–7,9,10].

5. Conclusions. Appendix A. Interprecision Transfers: Part II.

In [7,9,10] we advocated LPS interprecision with (1.3) rather than MPS. In this paper we will look into that more deeply. We will begin that investigation by comparing the cost of triangular solves with the two approaches to interprecision transfer to the cost of a single precision LU factorization. Since the triangular solvers are $O(N^2)$ work and the factorization is $O(N^3)$ work, the approach to interprecision transfer will matter less as the dimension of the problem increases.

The test problem was $\mathbf{Ax} = \mathbf{b}$ where the right side is $\mathbf{A}$ applied to the vector with 1 in each component. In this way we can compute error norms exactly.

A.1. Double-Single IR. In Table A.1 we report timings from Julia's **BenchmarkTools** package for double precision matrix vector multiply (MV64), single precision LU factorization (LU32) and three approaches for using the factors to solve a linear system. HPS is the time for a fully double precision triangular solved and MPS and LPS are the mixed precision solve and the fully low precision solve using

(1.3) and (1.4). IR will use a high precision matrix vector multiply to compute the residual and a solve to compute the correction for each iteration. The low precision factorization is done only once.

TABLE A.1
Timings for matrix-vector products and triangular solves vs factorizations: $\alpha = 800$

N	MV64	LU32	HPS	MPS	LPS	LU32/MPS
512	4.2e-05	1.2e-03	5.0e-05	1.0e-04	2.8e-05	1.2e+01
1024	8.2e-05	3.2e-03	1.9e-04	4.3e-04	1.0e-04	7.3e+00
2048	6.0e-04	1.4e-02	8.9e-04	2.9e-03	4.0e-04	4.8e+00
4096	1.9e-03	8.4e-02	4.8e-03	1.4e-02	2.2e-03	5.8e+00
8192	6.8e-03	5.8e-01	1.9e-02	5.8e-02	9.8e-03	1.0e+01

The last column of the table is the ratio of timings for the low precision factorization and the mixed precision solve. Keeping in mind that at least two solves will be needed in IR, the table shows that MPS can be a significant fraction of the cost of the solve for smaller problems and that LPS is at least 4 times less costly. This is a compelling case for using LPS in case considered in this section, where high precision is double and low precision is single, provided the performance of IR is equally good.

If one is solving $\mathbf{Ax} = \mathbf{b}$ for multiple right hand sides, as one would do for nonlinear equations in many cases [9], then LPS is significantly faster for small and moderately large problems. For example, for $N = 4096$ the cost of MPS is roughly 15% of the low precision LU factorization, so if one does more than 6 solves with the same factorization, the solve cost would be more than the factorization cost. LPS is five times faster and we saw this effect while preparing [9] and we use that in our nonlinear solver package [8]. The situation for IR is similar, but one must consider the cost of the high precision matrix-vector multiply, which is about the same as LPS.

We make LPS the default for IR if high precision is double and low precision is single. This decision is good for desktop computing. If low precision is half, then the LPS vs MPS decision needs more scrutiny and we will explain why the default is MPS later.

Finally we mention a subtle issue. We made Table A.1 with the standard commands for matrix-vector multiply ($\mathbf{A} * \mathbf{x}$), factoriztion **lu**, and used \ for the solve. Julia also offers non-allocating versions of these functions. In Table A.2 we show how usine those commands changes the results. We used **mul!** for matrix-vector multiply, **lu!** for the factorization, and **ldiv!** for the solve.

TABLE A.2
Timings for non-allocating matrix-vector products and triangular solves vs factorizations: $\alpha = 800$

N	MV64	LU32	HPS	MPS	LPS	LU32/MPS
512	3.6e-05	9.1e-04	5.0e-05	4.8e-05	2.8e-05	1.9e+01
1024	9.0e-05	2.7e-03	1.9e-04	1.8e-04	1.0e-04	1.5e+01
2048	6.2e-04	1.3e-02	8.9e-04	7.3e-04	3.9e-04	1.8e+01
4096	2.2e-03	8.0e-02	4.8e-03	3.3e-03	2.3e-03	2.4e+01
8192	6.5e-03	5.7e-01	2.1e-02	1.5e-02	1.0e-02	3.9e+01

A.2. Accuracy of MPS vs LPS. Since MPS does the triangular solves in high precision, one should expect that the results will be more accurate and that the improved accuracy might enable the IR loop to terminate earlier [3]. We should be able to see that by timing the IR loop after computing the factorization. One should also verify that the residual norms are equally good.

We will conclude this section with two final tables for the results of IR. We compare the well conditioned case ($\alpha = 1$) and the ill-conditioned case ($\alpha = 800$) for a few values of N. We will look at residual and error norms for both approaches to interprecision transfer. The conclusion is that if high precision is double and low is single, the two approaches give equally good results.

The columns of the tables are the dimensions, the ℓ^∞ relative error norms for both LP and MP interprecision transfers (ELP and EMP) and the corresponding relative residual norms (RLP and RMP).

The results for $\alpha = 1$ took 5 IR iterations for all cases. As expected the LPS iteration was faster than

MPS. However, for the ill-conditioned $\alpha = 800$ case, MPS took one fewer iteration (5 vs 6) than EPS for all but the smallest problem. Even so, the overall solve times were essentially the same.

TABLE A.3
Error and Residual norms: $\alpha = 1$

N	ELP	EMP	RLP	RMP	TLP	TMP
512	4.4e-16	5.6e-16	3.9e-16	3.9e-16	3.1e-04	3.9e-04
1024	6.7e-16	4.4e-16	3.9e-16	3.9e-16	1.1e-03	1.5e-03
2048	5.6e-16	4.4e-16	3.9e-16	3.9e-16	5.4e-03	6.2e-03
4096	1.1e-15	1.1e-15	7.9e-16	7.9e-16	1.9e-02	2.5e-02
8192	8.9e-16	6.7e-16	7.9e-16	5.9e-16	6.9e-02	9.3e-02

TABLE A.4
Error and Residual norms: $\alpha = 800$

N	ELP	EMP	RLP	RMP	TLP	TMP
512	6.3e-13	6.2e-13	2.1e-15	1.8e-15	3.0e-04	3.8e-04
1024	9.6e-13	1.1e-12	3.4e-15	4.8e-15	1.4e-03	1.5e-03
2048	1.0e-12	1.2e-12	5.1e-15	4.5e-15	6.5e-03	7.1e-03
4096	2.1e-12	2.1e-12	6.6e-15	7.5e-15	2.6e-02	2.4e-02
8192	3.3e-12	3.2e-12	9.0e-15	1.0e-14	9.1e-02	8.7e-02

REFERENCES

[1] J. BEZANSON, A. EDELMAN, S. KARPINSKI, AND V. B. SHAH, *Julia: A fresh approach to numerical computing*, SIAM Review, 59 (2017), pp. 65–98.

[2] E. CARSON AND N. J. HIGHAM, *A new analysis of iterative refinement and its application of accurate solution of ill-conditioned sparse linear systems*, SIAM Journal on Scientific Computing, 39 (2017), pp. A2834–A2856, https://doi.org/10.1137/17M112291.

[3] E. CARSON AND N. J. HIGHAM, *Accelerating the solution of linear systems by iterative refinement in three precisions*, SIAM Journal on Scientific Computing, 40 (2018), pp. A817–A847, https://doi.org/10.1137/17M1140819.

[4] J. DEMMEL, M. GATES, G. HENRY, X. LI, J. RIEDY, AND P. TANG, *A proposal for a next-generation BLAS*, 2017. preprint.

[5] N. J. HIGHAM, *Accuracy and Stability of Numerical Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996, http://www.ma.man.ac.uk/~higham/asna.html.

[6] N. J. HIGHAM, S. PRANESH, AND M. ZOUNON, *Squeezing a matrix into half precision, with an application to solving linear systems*, SIAM J. Sci. Comp., 41 (2019), pp. A2536–A2551.

[7] C. T. KELLEY, *Newton's method in mixed precision*, SIAM Review, 64 (2022), pp. 191–211, https://doi.org/10.1137/20M1342902.

[8] C. T. KELLEY, *SIAMFANLEquations.jl*, 2022, https://doi.org/10.5281/zenodo.4284807, https://github.com/ctkelley/SIAMFANLEquations.jl. Julia Package.

[9] C. T. KELLEY, *Solving Nonlinear Equations with Iterative Methods: Solvers and Examples in Julia*, no. 20 in Fundamentals of Algorithms, SIAM, Philadelphia, 2022.

[10] C. T. KELLEY, *Newton's method in three precisions*, 2023, https://arxiv.org/abs/2307.16051.

[11] Y. SAAD AND M. SCHULTZ, *GMRES a generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comp., 7 (1986), pp. 856–869.