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 best 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.

This document is for v0.1.4 of the package. We used both the AppleAccelerate [10] framework and openBLAS for the examples and the timings may be different from the older versions. All the computations in this paper were done on an Apple Mac Mini with an M2 Pro processor and 8 performance cores. The examples were done with several different versions of **MultiPrecisionArrays.jl** , Julia, and the OS. Hence the examples are indicative of the performance of IR, but the reader's results may differ in timings and iteration counts. The qualitative results are the same.

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

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

1. Introduction. The Julia [2] package **MultiPrecisionArrays.jl** [16, 17] 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 and convergence properties.

We focus on two views of IR. One is as 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.

The other is as a way to address very ill-conditioned problems, which was the purpose of the original paper on the topic [20]. Here one stores and factors $\mathbf{A}$ in one precision but evaluates the residual in a higher precision.

In most of this paper we consider IR using two precisions. We use the naming convention of [1,3,4,6] for these precisions and the others we will consider later. We start with the case where we store the data $\mathbf{A}$ and $\mathbf{b}$ in the higher or *working* precision and compute the factorization in the lower or *factorization* precision. Following standard Julia type notation, we will let TW and TF be the working and factorization precision types. So, for example

```
x = zeros(TW,N)
```

is a high precision vector of length N.

In a typical use case, TW will be double and TF 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 [7] version of the algorithm for the LU factorization.

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 W and F as sub and superscripts for the working and factorization precisions. When we are discussing specific use cases out sub and superscripts will the d, s, and h for double, single, and half precision.

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

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

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

$$\mathbf{A}_F = I_W^F(\mathbf{A})$$

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

The current version of **MultiPrecisionArrays.jl** v0.1.4 requires that the type of $\mathbf{A}$ be `AbstractArray{TW,2}` where TW is single or double. We will make this more general in a later version, but we will always require that the Julia function *lu*! accept the type of $\mathbf{A}$. In particular, this means that **MultiPrecisionArrays.jl** will not accept sparse arrays. I'd like to fix this, but have no idea how to do it.

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.

2.1. Classic Example: Double-Single Precision. While **MultiPrecisionArrays.jl** was designed for research, it is useful in applications in the classic case where the working precision is double and the factorization precision 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**.

The most simple way to use this package is to combine the construction of the MPArray with the factorization of the low precision copy of **A**. One does this with the **mplu** command.

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** package to get timings.

We use **lu!** from Julia because then neither factorization will reallocate the space for the high precision matrix. The excess cost for the allocation of the low precision matrix in **mplu** will show in the timings as will the reduced cost for the factorization.

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))
1.42840e-01

julia> @belapsed mplu($A)
8.60945e-02
```

At this point we have timed **lu!** and **mplu**. The single precision factorization is a bit more than half the cost of the double precision one.

It is no surprise that the factorization in single precision took roughly half as long as the one in double. In the double-single precision case, iterative refinement is a great example of a time/storage tradeoff. You have to store a low precision copy of , so the storage burden increases by 50% and the factorization time is cut in half. The advantages of IR increase as the dimension increases. IR is less impressive for smaller problems and can even be slower

```
julia> N=30; A=I + Gmat(N);

julia> @belapsed mplu($A)
4.19643e-06

julia> @belapsed lu!(AC) setup=(AC=copy($A))
3.70825e-06
```

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**.

2.3. Harvesting Iteration Statistics: Part I. You can get some iteration statistics by using the **reporting** keyword argument to the solvers. The easiest way to do this is with the backslash command. When you use this option you get a data structure with the solution and the residual history.

```
julia> using MultiPrecisionArrays

julia> using MultiPrecisionArrays.Examples

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

julia> MPF=mplu(A);

julia> # Use \ with reporting=true

julia> mpout=\(MPF, b; reporting=true);

julia> norm(b-A*mpout.sol, Inf)
1.33227e-15

julia> # Now look at the residual history

julia> mpout.rhist
5-element Vector{Float64}:
 9.99878e-01
 1.21892e-04
 5.25805e-11
 2.56462e-14
 1.33227e-15
```

As you can see, IR does well for this problem. The package uses an initial iterate of $\mathbf{x} = 0$ and so the initial residual is simply $\mathbf{r} = \mathbf{b}$ and the first entry in the residual history is $\|\mathbf{b}\|_\infty$. The iteration terminates successfully after four matrix-vector products.

You may wonder why the residual after the first iteration was so much larger than single precision roundoff. The reason is that the default when the low precision is single is to downcast the residual to single before the solve (onthefly=false).

One can enable interprecision transfers on the fly and see the difference.

```
julia> MPF2=mplu(A; onthefly=true);

julia> mpout2=\(MPF2, b; reporting=true);
```

```
julia> mpout2.rhist
5-element Vector{Float64}:
 9.99878e-01
 6.17721e-07
 3.84581e-13
 7.99361e-15
 8.88178e-16
```

So the second iteration is much better, but the iteration terminated after four iterations in both cases.

If we repeat the experiment using half precision as the low precision the solutions are equally good, but the iteration is slower.

```
julia> MPF2=mplu(A; TF=Float16);

julia> # The TF keyword argument lets you make half the low precision.

julia> mpout2 = \(MPF2, b; reporting=true);

julia> norm(A*mpout2.sol - b,Inf)
6.66134e-16

julia> mpout2.rhist
9-element Vector{Float64}:
 9.99878e-01
 4.58739e-03
 1.86362e-05
 7.36240e-08
 2.89855e-10
 1.14420e-12
 3.44169e-14
 3.10862e-15
 6.66134e-16
```

2.4. Options and data structures for mplu. Here is the source for `mplu`.

```
"""
function mplu(A::AbstractArray{TW,2}; TF=nothing, TR=nothing,
       residterm=residtermdefault, onthefly=nothing) where TW <: Real
#
# If the high precision matrix is single, the low precision must be half
# unless you're planning on using a high-precision residual where TR > TW
# and also factoring in the working precision, so TW == TF.
#
#
(TR == nothing) && (TR = TW)
TFdef = Float32
(TW == Float32) && (TFdef = Float16)
(TF == nothing) && (TF = TFdef)
#
# Unless you tell me otherwise, onthefly is true if low precision is half
# and false if low precision is single.
#
(onthefly == nothing ) && (onthefly = (TF==Float16))
#
# IF TF = TW then something funny is happening with the residual precision.
#
(TF == TW) && (onthefly=true)
#
# Build the multiprecision array MPA
#
MPA=MPArray(A; TF=TF, TR=TR, onthefly=onthefly)
#
# Factor the low precision copy to get the factorization object MPF
#
MPF=mplu!(MPA; residterm=residterm)
return MPF
end
"""
```

The function `mplu` has four keyword arguments. The easy one to understand is `TF` which is the precision of the factorization. Julia has support for single (`Float32`) and half (`Float16`) precisions. If you set `TF=Float16` then low precision will be half. Don't do that unless you know what you're doing. Using half precision is a fast way to get incorrect results. Look at § 3 for a bit more bad news.

`TR` is the residual precision. The `MPArray` structure preallocates storage for the residual and a local copy of the solution in precision `TR`. For most applications (but see § 6) `TR = TW`.

The final keyword arguments are `onthefly` and `residterm`. The `onthefly` keyword controls how the triangular solvers from the factorization work. When you solve

$$\mathbf{LUd = r}$$

the LU factors are in low precision and the residual r is in high precision. If you let Julia and LAPACK figure out what to do, then the solves will be done in high precision and the entries in the LU factors will be converted to high precision with each binary operation. The output d will be in high precision. This is called interprecision transfer on-the-fly and `onthefly = true` will tell the solvers to do it that way. You have N^2 interprecision transfers with each solve and, as we will see, that can have a non-trivial cost.

When low precision is Float32 and `TR = TW = Float64` then the default is `onthefly = false`. I am rethinking this in view of the experiments in [15] and make `onthefly = true` the default pretty soon.

When you set `onthefly = false` the solve converts r to low precision, does the solve entirely in low precision, and then promotes d to high precision. This is **in-place** interprecision transfer. You need to be careful to avoid overflow and, more importantly, underflow when you do that and we scale r to be a unit vector before conversion to low precision and reverse the scaling when we promote d. We take care of this for you.

The `residterm` keyword controls the termination options. See § 5.1 for the details on that. I will be making changes to the way **MultiPrecisionArrays.jl** controls termination frequently.

`mplu` calls the constructor for the multiprecision array and then factors the low precision matrix. In some cases, such as nonlinear solvers, you will want to separate the constructor and the factorization. I will provide code for this in a future release. I do not export the constructor and if you mess with it, remember that `mplu!` overwrites the low precision copy of A with the factors. The factorization object is different from the multiprecision array, even though they share storage. Be careful with this.

2.5. Memory Allocations: I. The memory footprint of a multiprecision array is dominated by the high precision array and the low precision copy. The allocations of

```
AF1=lu(A)
```

and

```
AF2=mplu(A)
```

are very different. Typically `lu` makes a high precision copy of **A** and factors that with `lu!`. `mplu` on the other hand, uses **A** as the high precision matrix in the multiprecision array structure and the makes a low precision copy to send to `lu!`. Hence `mplu` has half the allocation burden of `lu`.

That is, of course misleading. The most memory-efficient way to apply `lu` is to overwrite **A** with the factorization using

```
AF1=lu!(A).
```

The analog of this approach with a multiprecision array would be to first build an `MPArray` structure with a call to the constructor

```
MPA = MPArray(A)
```

which makes **A** the high precision matrix and also makes a low precision copy. This is the stage where the extra memory is allocated for the the low precision copy. One follows that with the factorization of the low precision matrix to construct the factorization object.

```
MPF = mplu!(MPA).
```

The function `mplu` simply applies `MPArray` and follows that with `mplu!`.

I do not export the constructor `MPArray`. You should use `mplu` and not import the constructor.

Once you have used `mplu` to make a multiprecision factorization, you can reuse that storage for a different matrix as long as the size and the precision are the same. For example, suppose

```
MPF = mplu(A)
```

is a multiprecision factorization of **A**. If you want to factor **B** and reuse the memory, then

```
MPF = mplu!(MPF,B)
```

will do the job.

3. Half Precision. There are two half precision (16 bit) formats. Julia has native support for IEEE 16 bit floats (Float16). A second format (BFloat16) has a larger exponent field and a smaller significand (mantissa), thereby trading precision for range. In fact, the exponent field in BFloat is the same size (8 bits) as that for single precision (Float32). The significand, however, is only 8 bits. Which is less than that for Float16 (11 bits) and single (24 bits). The size of the significand means that you can get in real trouble with half precision in either format.

At this point Julia has no native support for BFloat16. Progress is being made and we expect to support BFloat16 in the future.

Doing the factorization in half precision (Float16 or BFloat16) 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** [5]) support half precision, so all the clever stuff in there is missing. We provide a half precision LU factorization for Float16 /**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 [5] and Julia and Apple support Float16 in hardware. Apple also has hardware support for BFloat16. However, for now, at least for desktop computing, half precision is for research in iterative refinement, not applications.

Here's a table 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 $\mathbf{I} - 800.0\mathbf{G}$.

TABLE 3.1
Half precision is slow: LU timings

N	Double	Single	Half	Ratio
1024	3.96e-03	2.80e-03	2.48e-02	6.26e+00
2048	2.23e-02	1.43e-02	6.91e-02	3.09e+00
4096	1.54e-01	8.51e-02	3.08e-01	1.99e+00
8192	1.14e+00	6.02e-01	4.40e+00	3.87e+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. In this example we use the keyword argument "'onthefly"' to toggle between MPS and LPS.

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

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

julia> MPF=mplu(A; 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> MPF2=mplu(A; onthefly=true);

julia> z=MPF2\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 before converging to the correct result.

4. Using the Low Precision Factorization as a Preconditioner. In this section we present some options if IR fails to converge. This is very unlikely if high precision is double and low precision is single. If low precision is half, the methods in this section might save you.

The idea is simple. Even if

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

has norm larger than one, it could still be the case that

$$\hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}\mathbf{A}$$

is well conditioned and that

$$\mathbf{P} = \hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1} \tag{4.1}$$

could be a useful preconditioner for a Krylov method.

4.1. Direct Preconditioning. The obvious way to use $\mathbf{P}$ is simply to precondition the equation $\mathbf{Ax} = \mathbf{p}$. In this case we prefer right preconditioning where we solve

$$\mathbf{APz} = \mathbf{b}$$

and then set $\mathbf{x} = \mathbf{Px}$. This is different from all IR methods we discuss in this paper and one may lose some accuracy by avoiding the IR loop.

4.2. Krylov-IR. Krylov-IR methods solve the correction equation with a preconditioned Krylov iteration using the low precision solve as the preconditioner. Currently **MultiPrecisionArrays.jl** supports GMRES [18] and BiCGSTAB [19].

4.3. GMRES-IR. GMRES-IR [3,4] solves the correction equation with a preconditioned GMRES [18] 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 $\mathbf{P}$ as the preconditioner. The preconditioned equation is

$$\mathbf{PAd} = \mathbf{Pr}.$$

The reason for using left preconditioning is that one is not interested in a small residual for the correction equation, but in capturing $\mathbf{d}$ as well as possible. The IR loop is the part of the solve that seeks a small residual norm.

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. The Julia function **mpglu** constructs the data structure and factors the low precision copy of the matrix. The output, like that of **mplu** is a factorization object that you can use with backslash.

Here is a well conditioned example. Both IR and GMRES-IR perform well, with GMRES-IR taking significantly more time. In these examples high precision is single and low precision is half.

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

julia> MPF=mplu(A); MPF2=mpglu(A);

julia> z=MPF\b; y=MPF2\b; println(norm(z-x,Inf),"  ",norm(y-x,Inf))
5.9604645e-7  4.7683716e-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;
  40.028 ms (183 allocations: 90.55 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> MPF=mplu(A); MPF2=mpglu(A);

julia> z=MPF\b; y=MPF2\b; println(norm(z-x,Inf),"  ",norm(y-x,Inf))
0.2875508  0.0044728518

julia> println(norm(b-A*z,Inf)/norm(b,Inf),"  ",norm(b-A*y,Inf)/norm(b,Inf))
0.0012593127  1.4025759e-5
```

So, the relative error and relative residual norms for GMRES-IR are much smaller than for IR.

4.4. Memory Allocations: II. Much of the discussion from § 2.5 remains valid for the `MGPArray` structure and the associated factorization structure `MPGEFact`. The only difference that matters is that `MGPArray` contains the Krylov basis and a few other vectors that GMRES needs, so the allocation burden is a little worse.

The allocation burden is less for BiCGSTAB-IR and the `MPBArray` structure.

That aside, `mpglu!` and `mpblu!` work the same way that `mplu!` does when factoring or updating a `MGPArray`.

The functions `mpglu` and `mpblu` combine the allocations and the factorization.

4.5. Harvesting Iteration Statistics: Part 2. The output for GMRES-IR contains the residual history and a vector with the number of Krylov iterations for each IR step. The next example illustrates that.

```
julia> MPGF=mpglu(A);

julia> moutg=\(MPGF, b; reporting=true);

julia> norm(A*moutg.sol-b, Inf)
1.44329e-15

julia> moutg.rhist
3-element Vector{Float64}:
 9.99878e-01
```

```
 7.48290e-14
 1.44329e-15

julia> moutg.khist
2-element Vector{Int64}:
 4
 4
```

While only two IR iterations are needed for convergence, the Krylov history shows that each of those IR iterations needed four GMRES iterations. Each of those GMRES iterations requires a matrix-vector product and a low-precision on-the-fly linear solve. So GMRES-IR is more costly and, as pointed out in [3,4] is most useful with IR does not converge on its own.

We will demonstrate this with one last example. In this example high precision is single and low precision is half. As you will see, this example is very ill-conditioned.

```
julia> N=8102; AD = I - 799.0*Gmat(N); A=Float32.(AD); x=ones(Float32,N); b=A*x;

julia> cond(A, Inf)
2.34824e+05

julia> MPFH=mplu(A);

julia> mpouth=\(MPFH, b; reporting=true);

julia> # The iteration fails.

julia> mpouth.rhist
4-element Vector{Float64}:
 9.88752e+01
 9.49071e+00
 2.37554e+00
 4.80087e+00

julia> # Try again with GMRES-IR and mpglu

julia> MPGH=mpglu(A);

julia> mpoutg=\(MPGH, b; reporting=true);

julia> mpoutg.rhist
4-element Vector{Float32}:
 9.88752e+01
 1.86920e-03
 1.29700e-03
 2.46429e-03

julia> mpoutg.khist
3-element Vector{Int64}:
 10
 10
 10
```

So GMRES-IR does much better. Note that we are taking ten GMRES iterations for each IR step. Ten is the default. To increase this set the keyword argument **basissize**.

5. Details. In this section we discuss a few details that are important for understanding IR, but less important for simply using **MultiPrecisionArrays.jl** .

5.1. Terminating the while loop. There are many parameters in the termination criteria and the defaults in **MultiPrecisionArrays.jl** are reasonable for most applications. These defaults differ slightly from the recommendations in [8] and we explain that later in this section.

One can terminate the iteration when the relative residual is small *i. e.* when

$$\|\mathbf{r}\| < C_r u_r \|\mathbf{b}\| \tag{5.1}$$

or when the normwise backward error is small, *i. e.*

$$\|\mathbf{r}\| < C_e u_r (\|\mathbf{b}\| + \|\mathbf{A}\|\|\mathbf{x}\|). \tag{5.2}$$

You make the choice between (5.1) and (5.2) when you compute the multiprecision factorization.

The prefactors C_r and C_e are algorithmic parameters $C_r = 20.0$ is the default in **MultiPrecisionArrays.jl** . The recommendation in [8] is that $C_e = 1.0$, which is the default in **MultiPrecisionArrays.jl** . The reason $C_r > C_e$ in **MultiPrecisionArrays.jl** is that putting $\|\mathbf{A}\|$ in the termination criterion scales the termination criterion well and making C_r large helps compensate for the lack of such scaling in (5.2). We freely admit that this choice is a hack.

5.1.1. Residual or Backward Error. Using the normwise backward error is more costly because the computation of $\|\mathbf{A}\|$ is N^2 work and is more expensive that a few IR iterations.

In **MultiPrecisionArrays.jl** we do this when we compute $\|\mathbf{A}\|$ during the multiprecision factorization. After we copy $\mathbf{A}$ to the factorization precision, we can take the norm of the low-precision copy before we factor it. Specifically, we compute $\|\mathbf{A}\|_1$ in the factorization precision if `TF = Float32` or `TF = Float64`. If `TF` is half precision, which is still slow in our desktop environment, we compute $\|\mathbf{A}\|$ in the working precision `TW`. We control this via the `residterm` `kwarg` for the factorizations.

The user can change from using (5.1) to (5.2) with the keword parameter `residterm` in `mplu`. The default value is `true`. Setting it to `false` enables termination on small normwise backward error. For example

```
MPF = mplu(A; residterm=false)
```

tells the solver to use (5.2).

5.1.2. Protection Against Stagnation. The problem with either of (5.1) or (5.2) 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

$$\|\mathbf{r}_{new}\| \geq R_{max}\|\mathbf{r}_{old}\| \tag{5.3}$$

even if the small residual condition is not satisfied. The recommendation in [8] is $R_{max} = .5$ and that is the default in **MultiPrecisionArrays.jl** .

The paper [8] also recommends that on put a limit `litmax = 5` on the number of IR iterations. We use a much higher (essentially infinite) value of `litmax = 1000` in **MultiPrecisionArrays.jl** .

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 (iteration 0) 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 test for a meaningful residual reduction. Iterations after iteration 0 cost more because they do a meaningful residual computation.

5.1.3. Changing the parameters. The parameters and defaults are

- $C_r = 20.0$
- $C_e = 1.0$
- $R_{max} = .5$
- $litmax = 1000$

We store the parameters in a `mutable structure` in the main file for **MultiPrecisionArrays.jl** .

```
mutable struct TERM
       Cr::Real
       Ce::Real
       Rmax::Real
       litmax::Int
end
```

We create two `TERM` structures, both initialized to be the defaults. One `term_Parms` is examined by the solves in **MultiPrecisionArrays.jl** and the other `term_parms_default` is used as permanent storage.

To change the parameters use the `update_parms` function. This will always begin with the defaults.

```
function update_parms(t::TERM = term_parms; Cr = 20.0, Ce = 1.0,
         Rmax = 0.5, litmax=1000)
    #
    # The parameters live in a mutable struct term_parms in
    # MultiPrecisionArrays.jl and are initialized to the default values.
    #
    t.Cr = Cr
    t.Ce = Ce
    t.Rmax = Rmax
    t.litmax = litmax
    return t
end
```

To return to the default parameters, simply call `update_parms` with no `kwargs`.

```
julia> using MultiPrecisionArrays
julia> # Look at the default parameters
julia> term_parms
MultiPrecisionArrays.TERM(2.00000e+01, 1.00000e+00, 5.00000e-01, 1000)
julia> # Change C_r to 400
julia> update_parms(; Cr=400.0)
MultiPrecisionArrays.TERM(4.00000e+02, 1.00000e+00, 5.00000e-01, 1000)
julia> # Now go back
julia> update_parms(;)
MultiPrecisionArrays.TERM(2.00000e+01, 1.00000e+00, 5.00000e-01, 1000)
```

Keep in mind that the function always begins with the defaults, so if you want to experiment with a sequence of changes, you need to take care. The parameters are stored in global storage in the module. The reason for this is that I did not want to clutter the API for the solvers with all these parameters.

5.2. Is $O(N^2)$ really negligible. In this section `TR = TW = Float64` and `TF = TS = Float32`, which means that the iterprecision transfers in the triangular solvers are done in-place. We terminate on small residuals.

The premise behind IR is that reducing the $O(N^3)$ cost of the factorization will make the solve faster because everything else is $O(N^2)$ work. It's worth looking into this.

We will use the old-fashioned definition of a FLOP as an add, a multiply and a bit of address computation. So we have N^2 flops for any of (1) matrix-vector multiply $\mathbf{Ax}$, (2) the two triangular solves with the LU factors $(\mathbf{LU})^{-1}\mathbf{b}$, and (3) computation of the ℓ^1 or ℓ^∞ matrix operator norms $\|\mathbf{A}\|_{1,\infty}$.

A linear solve with an LU factorization and the standard triangular solve has a cost of $(N^3/3) + N^2$ TR-FLOPS. The factorization for IR has a cost of $N^3/3$ TF-FLOPS or $N^3/6$ TR-FLOPS.

A single IR iteration costs a matrix-vector product in precision TR and a triangular solve in precision TF for a total of $3N^2/2$ TR-FLOPS. Hence a linear solve with IR that needs n_I iterations costs

$$\frac{N^3}{6} + 3n_I N^2/2$$

TR-FLOPS if one terminates on small residuals and an extra N^2 TR-FLOPS if one computes the norm of $\mathbf{A}$ in precision TR.

IR will clearly be better for large values of N. How large is that? Table 5.1 and Table 5.2 compare the time for factorization nd solve using *lu*! and *ldiv* (cols 2-4) with the equivalent multiprecision commands *mplu* and \. The final column is the time for computing $\|\mathbf{A}\|_1$, which would be needed to terminate on small normwise backward errors.

The operator is `A = I - Gmat(N)`. We tabulate

- LU: time for `AF=lu!(A)`
- TS: time for `ldiv!(AF,b)`
- TOTL = LU+TS
- MPLU: time for `MPF=mplu(A)`
- MPS: time for `MPF\b`
- TOT: MPLU+MPS
- OPNORM: Cost for $\|A\|_1$, which one needs to terminate on small normwise backward error.

The message from the tables is that the triangular solves are more costly than operation counts might indicate. One reason for this is that the LU factorization exploits multi-core computing better than a triangular solve. It is also interesting to see how the choice of BLAS affects the relative costs of the factorization and the solve. Also notice how the computation of $\|\mathbf{A}\|$ is more costly than the two triangular solves for `ldiv!(AF,b)`.

For both cases, multiprecision arrays perform better when $N \geq 2048$ with the difference becoming larger as N increases.

TABLE 5.1
$\alpha = 1$, *openBLAS*

N	LU	TS	TOTL	MPLU	MPS	TOT	OPNORM
512	9.8e-04	6.4e-05	1.0e-03	7.8e-04	2.9e-04	1.1e-03	1.7e-04
1024	3.7e-03	2.6e-04	4.0e-03	3.1e-03	8.3e-04	4.0e-03	7.8e-04
2048	2.1e-02	1.5e-03	2.3e-02	1.4e-02	5.4e-03	1.9e-02	3.4e-03
4096	1.5e-01	5.4e-03	1.5e-01	8.8e-02	2.0e-02	1.1e-01	1.4e-02
8192	1.1e+00	2.1e-02	1.1e+00	6.0e-01	7.3e-02	6.7e-01	5.8e-02

TABLE 5.2
$\alpha = 1$, *AppleAcclerateBLAS*

N	LU	TS	TOTL	MPLU	MPS	TOT	OPNORM
512	7.9e-04	1.1e-04	9.0e-04	4.7e-04	2.9e-04	7.6e-04	1.7e-04
1024	3.4e-03	4.9e-04	3.9e-03	2.4e-03	1.2e-03	3.6e-03	7.8e-04
2048	1.9e-02	2.4e-03	2.1e-02	1.1e-02	1.1e-02	2.2e-02	3.4e-03
4096	1.4e-01	1.2e-02	1.5e-01	6.1e-02	5.6e-02	1.2e-01	1.4e-02
8192	1.2e+00	5.5e-02	1.3e+00	5.8e-01	2.2e-01	8.0e-01	5.7e-02

5.3. 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 TH is a higher precision than TL $a, b \in \mathcal{F}_H$ and $c \in \mathcal{F}_L$ that

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

For now, $TH = TW$ and $TL = TF$, but we need to think of the general case in § 6. 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 the $O(N^2)$ interprecision transfer 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. In the terminology of [1], the MPS approach sets the solver precision TS to the precision of the residual computation TR, which is TW for now.

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

$$(\mathbf{LU})\mathbf{d}_F = I_W^F(\mathbf{r}/\|\mathbf{r}\|) \tag{5.4}$$

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

$$\mathbf{d} = \|\mathbf{r}\| I_F^W(\mathbf{d}_F). \tag{5.5}$$

We will refer to this approach as low precision solves (LPS) when we report computational results in § A. In practice, if low precision is single, the quality of the results is as good as one would get with MPS and the solve phase is somewhat faster. Making the connection to [1] again, the LPS approach sets $TS = TF$.

We express this choice in **MultiPrecisionArrays.jl** with the `onthefly` keyword argument which tells the solvers to do the triangular solves on the fly (`true`) or not (`false`). The default is `true` unless TW is double and TF is single.

5.4. Convergence Theory.

5.4.1. Estimates for $\|\mathbf{M}_{IR}\|$. We will estimate the norm of $\mathbf{M}_{IR}$ to see how the factorization precision affects the convergence. First write

$$\mathbf{M}_{IR} = \mathbf{I} - \hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}\mathbf{A} = \hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}(\hat{\mathbf{L}}\hat{\mathbf{U}} - \mathbf{A}). \tag{5.6}$$

We split $\Delta\mathbf{A} = (\hat{\mathbf{U}}\hat{\mathbf{L}} - \mathbf{A})$ to separate the rounding error from the backward error in the low precision factorization

$$\Delta\mathbf{A} = (\hat{\mathbf{U}}\hat{\mathbf{L}} - I_W^L\mathbf{A}) + (I_W^L\mathbf{A} - \mathbf{A}).$$

The last term can be estimated easily

$$\|I_W^F\mathbf{A} - \mathbf{A}\| \le u_F\|\mathbf{A}\|. \tag{5.7}$$

To estimate the first term we look at the component-wise backward error [7]. If $3Nu_F < 1$ then

$$|\hat{\mathbf{L}}\hat{\mathbf{U}} - I_W^L\mathbf{A}| \le \gamma_{3N}(u_F)|\hat{\mathbf{L}}||\hat{\mathbf{U}}|. \tag{5.8}$$

In (5.8) $|\mathbf{B}|$ is the matrix with entries the absolute values of those in $\mathbf{B}$ and

$$\gamma_k(u) = \frac{ku}{1-ku}.$$

We can combine (5.7) and (5.8) to get

$$\begin{aligned}\|\mathbf{M}_{IR}\| &\le u_F\|\mathbf{A}\| + \gamma_k(u_F)\|\hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}\|\|\hat{\mathbf{L}}\|\|\hat{\mathbf{U}}\| \\ &= u_F\|\mathbf{A}\| + \gamma_k(u_F)\kappa(\hat{\mathbf{L}})\kappa(\hat{\mathbf{U}}).\end{aligned} \tag{5.9}$$

The standard estimate in textbooks for $\|\hat{\mathbf{L}}\|\|\hat{\mathbf{U}}\|$ uses very pessimistic (and unrealistic) worst case bounds on the right side of (5.8). In cases where the conditioning of the factors is harmless, the estimate in (5.9) suggests that IR should converge well if low precision is single.

We will use the probabilistic bounds from [9] to explore this in more detail. Roughly speaking, with high probability for desktop sized $N \leq 10^{10}$ problems we obtain

$$|\hat{\mathbf{L}}\hat{\mathbf{U}} - I_W^L \mathbf{A}| \leq (13u_F\sqrt{N} + O(u_F^2))\|\hat{\mathbf{L}}\|\|\hat{\mathbf{U}}\|. \tag{5.10}$$

If we neglect the $O(u_F^2)$ term in (5.10), our estimate for $\mathbf{M}$ becomes

$$\begin{aligned} \|\mathbf{M}\| &\leq u_F(\|\mathbf{A}\| + \|\hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}\|13\sqrt{N}\|\hat{\mathbf{L}}\|\|\hat{\mathbf{U}}\|) \\ &\leq u_l(\|\mathbf{A}\| + 13\sqrt{N}\kappa(\hat{\mathbf{L}})\kappa(\hat{\mathbf{U}})). \end{aligned} \tag{5.11}$$

So, $\|\mathbf{M}\| < 1$ if

$$\|\mathbf{A}\| + 13\sqrt{N}\kappa(\hat{\mathbf{L}})\kappa(\hat{\mathbf{U}}) < u_F^{-1}.$$

For example if we assume that $\|\mathbf{A}\| = O(1)$, low precision is single ($u_F = u_s \approx 1.2 \times 10^{-7}$), and we make a fairly pessimistic assumption about the conditioning of the low precision factors,

$$\kappa(\hat{\mathbf{L}})\kappa(\hat{\mathbf{U}}) \leq \sqrt{N},$$

then $\|\mathbf{M}\| < 1$ if

$$N < u_s^{-1}/14 \approx 6 \times 10^5 \tag{5.12}$$

which is the case for most desktop sized problems. However, if low precision is half, then (5.12) becomes with $u_F = u_h \approx 9.8 \times 10^{-3}$

$$N < u_F^{-1}/14 \approx 73. \tag{5.13}$$

This is an indication that there are serious risks in using half precision if the conditioning of the low precision factors increases with N, which could be the case if the $\mathbf{A}$ is a discretization of a boundary value problem.

5.4.2. Limiting Behavior of IR. In exact arithmetic one would get a reduction in the error with each iteration of a factor of $\rho(\mathbf{M}_{IR}) \leq \|\mathbf{M}_{IR}\|$. However, when one accounts for the errors in the residual computation, we will see how and when the iteration can stagnate. Our analysis will be a simplified version of the one from [4] and we will neglect many of the details.

In this section we will consider dense matrices with solves with MPS, so the solves with the low precision factors are done in high precision. Hence in exact arithmetic

$$\mathbf{x}_+ = \mathbf{x}_c + \mathbf{M}_{IR}\mathbf{x}_c + \hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}\mathbf{b}.$$

So the residual update is

$$\begin{aligned} \mathbf{r}_+ &= \mathbf{b} - \mathbf{A}\mathbf{x}_c \\ &= \mathbf{r}_c - \mathbf{A}\hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}\mathbf{r}_c \equiv \mathbf{M}_{RES}\mathbf{r}_c, \end{aligned}$$

where

$$\mathbf{M}_{RES} = \mathbf{I} - \mathbf{A}\hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}. = \hat{\mathbf{U}}^{-1}\hat{\mathbf{L}}^{-1}(\hat{\mathbf{L}}\hat{\mathbf{U}} - \mathbf{A}).$$

The analysis in the previous section implies that

$$\|\mathbf{M}_{RES}\| \leq \alpha,$$

where

$$\alpha = u_F(\|\mathbf{A}\| + 13\sqrt{N}\kappa(\hat{\mathbf{L}})\kappa(\hat{\mathbf{U}})). \tag{5.14}$$

One could also use $\rho(\mathbf{M}_{IR})$ for the convergence rate, but we think (5.14) is more illuminating.

As is standard, when one computes a residual $\mathbf{r}$ the computed value $\hat{\mathbf{r}}$ has an error [7]

$$\hat{\mathbf{r}} = \mathbf{r} + \delta_{\mathbf{r}}$$

where

$$\|\delta_{\mathbf{r}}\| \leq \gamma_N(u_W)(\|\mathbf{A}\|\|\mathbf{x}\| + \|\mathbf{b}\|).$$

We will do the analysis in terms of reduction in the residual norm. We will then use that to estimate the limiting behavior of the error norm. We will assume that $\alpha < 1$ and that the IR iteration is bounded

$$\|\mathbf{x}\| \leq C\|\mathbf{x}^*\|$$

Hence

$$\|\delta_{\mathbf{r}}\| \leq \xi \equiv \gamma_N(u_W)(C\|\mathbf{A}\|\|\mathbf{x}^*\| + \|\mathbf{b}\|). \tag{5.15}$$

We will analyze the progress of IR hile only considering the errors in the the residual computation. So we compute

$$\hat{\mathbf{r}}_+ = \mathbf{M}_{RES}\hat{\mathbf{r}}_c + \delta_{\mathbf{r}_c}$$

implying that

$$\|\mathbf{r}_+\| \leq \alpha\|\mathbf{r}_c\| + (1+\alpha)\|\delta_{\mathbf{r}_c}\| \leq \alpha\|\mathbf{r}_c\| + (1+\alpha)\xi.$$

Hence, for any $n \geq 0$

$$\|\mathbf{r}_{n+1}\| \leq \alpha\|\mathbf{r}_n\| + \frac{1+\alpha}{1-\alpha}\xi.$$

So, the iteration will stagnate when

$$\|\mathbf{r}\| \approx \frac{1+\alpha}{1-\alpha}\xi. \tag{5.16}$$

When we terminate the iteration when $\|\mathbf{r}\|/\|\mathbf{b}\|$ is small we are ignoring the $\|\mathbf{A}\|\|\mathbf{x}^*\|$ term in ξ, which one reason we must take watch for stagnation in our solver.

6. Evaluating the residual in extended precision. This idea comes from [20] and I am using the notation from [1,6]. The idea is to evaluate the residual in a precision TR higher than the working precision. If you do this, you should store both the solution and the residual in precision TR and to interprecision transfers on the fly. In that case you are really solving a promoted problem [15]

$$(I_W^R A)x = I_W^R b$$

and, by driving the residual to a small value can mitigate ill-conditioning to some degree. **MultiPrecisionArrays.jl** allows you to do this with the multiprecision factorization you get from `mplu`. I may add this feature to the Krylov-IR solvers later.

The classic example is to let TR and TF be single precision and TR be double. The storage penalty is that you must store two copies of A, one for the residual computation and the other for the factorization. Here is an example with a badly conditioned matrix. You must tell `mplu` to factor in the working precision and then use the `kwargs` in the solver to set TR.

```
julia> using MultiPrecisionArrays

julia> using MultiPrecisionArrays.Examples

julia> N=4096; alpha=799.0; AD=I - alpha*Gmat(N);

# conditioning is bad

julia> cond(AD,Inf)
2.35899e+05
```

```
# Set up the single precision computation

julia> A = Float32.(AD); xe=ones(Float32,N); b=A*xe;

# Make sure TF is what it needs to be for this example

julia> AF = mplu(A; TF=Float32);

# Use the multiprecision array to solve the problem, set TR.
julia> mrout = \(AF, b; reporting=true, TR=Float64);

# look at the residual history

julia> mrout.rhist
5-element Vector{Float64}:
 9.88750e+01
 1.67735e-05
 9.23976e-10
 9.37916e-13
 9.09495e-13

# Compare results with LU and the exact(?) solution

julia> xr=Float32.(mrout.sol); xs = A\b;

julia> [norm(b - A*xr, Inf) norm(b - A*xs, Inf)]
1x2 Matrix{Float32}:
 1.29700e-04  1.41144e-03

# So the residual is better. What about the difference from xe?

julia> [norm(xr - xe, Inf) norm(xs - xe, Inf)]
1x2 Matrix{Float32}:
 8.47816e-04  8.82089e-04

# Nothing exciting here. You have to wonder what this all means.
# Finally, how did we do with the promoted problem?

julia> AP=Float64.(A);

julia> norm(b - AP*mrout.sol, Inf)
7.10543e-13

# Which is what I said it was above.
```

So, is the solution to the promoted problem better than the exact solution I used to build the problem?

Now I'll try this with TF=half (Float16), the default when TW = single (Float32). All that one has to do is replace

```
AF = mplu(A; TF=Float32);
```

with

```
AF = mplu(A);
```

The iteration fails to converge because AF is not accurate enough.o

```
julia> AF = mplu(A; TR=Float64);

julia> mout16=\(AF, b; reporting=true);

julia> mout16.rhist
5-element Vector{Float64}:
 9.88750e+01
 3.92614e+00
 3.34301e-01
 2.01975e-01
 2.24576e-01
```

The advantages of evaluating the residual in extended precision grow when A is extremely ill-conditioned. Of course, in this case the factorization in the working precision could be so inaccurate that IR will fail to converge. One approach to respond to this, as you might expect, is to use the factorization as a preconditioner and not a solver [1].

6.1. IR-Krylov. MultiPrecisionArrays.jl supports IR-GMRES and IR-BiCGSTAB for TR > TW. You get this to work just like in `mplu` by using the keyword argument `TR`. We will continue with the example in this section and do that.

```
julia> GF = mpglu(A; TR=Float64);

julia> moutG = \(GF, b; reporting=true);

julia> moutG.rhist
6-element Vector{Float64}:
 9.88750e+01
 2.23211e-04
 9.61252e-09
 1.26477e-12
 8.10019e-13
 8.66862e-13

# You need several iterations because the default is 10 Krylov vectors
# And we got the solution to the promoted problem...

julia> xp = Float64.(A)\b;

julia> norm(xp-moutG.sol, Inf)
6.52844e-12

# IR-BiCGSTAB should take fewer iterations because there's no storage
# issue. But remember that BiCGSTAB has a higher cost per linear iteration.

julia> BF = mpblu(A; TR=Float64);

julia> moutB = \(BF, b; reporting=true);

julia> moutB.rhist
4-element Vector{Float64}:
 9.88750e+01
 2.16858e-11
 8.38440e-13
 8.81073e-13

julia> norm(xp - moutB.sol, Inf)
1.36534e-11
```

Appendix A. Interprecision Transfers: Part II.

In [11, 13, 14] we advocated LPS interprecision with (5.4) rather than MPS. In this section we will look into that more deeply and this discussion will reflect our recent experience [15]. 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 (5.4) and (5.5). 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.

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

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

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 the 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 [13], 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 [13] and we use that in our nonlinear solver package [12]. 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 is different since the factorization in half precision is so expensive.

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

So, while LPS still may make sense for small problems if high precision is double and low precision is single, the case for using it is weaker if one uses non-allocating matrix-vector multiplies and solves. We do that in **MultiPrecisionArrays.jl**.

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 [4]. 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] P. AMESTOY, A. BUTTARI, N. J. HIGHAM, J.-Y. L'EXCELLENT, T. MARY, AND B. VIEUBLÉ, *Five-precision gmres-based iterative refinement*, SIAM Journal on Matrix Analysis and Applications, 45 (2024), pp. 529–552, https://doi.org/10.1137/23M1549079.

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

[3] 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.

[4] 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.

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

[6] J. DEMMEL, Y. HIDA, W. KAHAN, X. S. LI, S. MUKHERJEE, AND E. J. RIEDY, *Error bounds from extra-precise iterative refinement*, ACM Trans. Math. Soft., (2006), pp. 325–351.

[7] 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.

[8] N. J. HIGHAM, *Iterative refinement for linear systems and LAPACK*, IMA J. Numerical Anal., 17 (1997), pp. 495–509.

[9] N. J. HIGHAM AND T. MARY, *A new approach to probabilistic rounding error analysis*, SIAM J. Sci. Comput., 1 (2019), pp. A2815–A2835.

[10] *Appleaccelerate.jl*. https://github.com/JuliaLinearAlgebra/AppleAccelerate.jl, 2023. Julia Package.

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

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

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

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

[15] C. T. KELLEY, *Interprecision transfers in iterative refinement*, 2024, https://arxiv.org/abs/2407.00827. submitted for publication.

[16] C. T. KELLEY, *MultiPrecisionArrays.jl*, 2024, https://doi.org/10.5281/zenodo.13851500, https://github.com/ctkelley/MultiPrecisionArrays.jl. Julia Package.

[17] C. T. KELLEY, *MultiPrecisionArrays.jl: A Julia package for iterative refinement*, Journal of Open Source Software, 9 (2024), https://doi.org/10.21105/joss.06698.

[18] 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.

[19] H. A. van der Vorst, *Bi-CGSTAB: A fast and smoothly converging variant to Bi-CG for the solution of nonsymmetric systems*, SIAM J. Sci. Stat. Comp., 13 (1992), pp. 631–644.

[20] J. H. Wilkinson, *Progress report on the automatic computing engine*, Tech. Report MA/17/1024, Mathematics Division, Department of Scientific and Industrial Research, National Physical Laboratory, Teddington, UK, 1948, http://www.alanturing.net/turing_archive/archive/l/l10/l10.php.