

MultiPrecisionArrays.jl: A Julia package for iterative refinement

C. T. Kelley 

North Carolina State University, Raleigh NC, USA

DOI: [10.xxxxxx/draft](https://doi.org/10.xxxxxx/draft)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Open Journals](#) 

Reviewers:

- [@openjournals](#)

Submitted: 01 January 1970

Published: unpublished

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#))

Summary

[MultiPrecisionArrays.jl](#) Kelley (2023b), Kelley (2023a) provides data structures and solvers for several variations of iterative refinement (IR). IR can speed up an LU matrix factorization by factoring a low precision copy and using the low precision factorization in a residual correction loop. The additional storage cost is the low precision copy, so IR is at time vs storage tradeoff. IR is an old algorithm and a good account of the classical theory is in Higham (1996).

Statement of need

Who cares?

Algorithm

This package will make solving dense systems of linear equations faster by using the LU factorization and IR. It is limited to LU for now. A very generic description of this for solving a linear system $Ax = b$ is

IR(A, b)

- $x = 0$
- $r = b$
- Factor $A = LU$ in a lower precision
- While $\|r\|$ is too large
 - $d = (LU)^{-1}r$
 - $x = x + d$
 - $r = b - Ax$
- end
- end

In Julia, a code to do this would solve the linear system $Ax = b$ in double precision by using a factorization in a lower precision, say single, within a residual correction iteration. This means that one would need to allocate storage for a copy of A in the lower precision and factor that copy. Then one has to determine what the line $d = (LU)^{-1}r$ means. Do you cast r into the lower precision before the solve or not? **MultiPrecisionArrays.jl** provides data structures and solvers to manage this. The **MPArray** structure lets you preallocate A , the low precision copy, and the residual r . The factorizations factor the low-precision copy and the solvers use that factorization and the original high-precision matrix to run the while loop in the algorithm.

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

Installation

The standard way to install a package is to type `import Pkg; Pkg.add("MultiPrecisionArrays")` at the Julia prompt. One can run the unit tests with `Pkg.test("MultiPrecisionArrays")`. After installation, type using `MultiPrecisionArrays` when you want to use the functions in the package.

Example

Here is a simple example to show how iterative refinement works. We will follow that with some benchmarking on the cost of factorizations. The functions we use are **MPLu!** to create the structure and **mplu!** to factor the low precision copy. In this example high precision is `Float64` and low precision is `Float32`. The matrix is the sum of the identity and a constant multiple of the 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

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

The code for this is in the `/src/Examples` directory. The file is **Gmat.jl**. You need to do

```
using MultiPrecisionArrays
using MultiPrecisionArrays.Examples
to get to it.
```

The example below compares the cost of a double precision factorization to a `MPLu!` factorization. The `MPLu!` structure has a high precision (TH) and a low precision (TL) matrix. The structure we will start with is

```
struct MPLu!{TH<:AbstractFloat, TL<:AbstractFloat}
    AH::Array{TH, 2}
    AL::Array{TL, 2}
    residual::Vector{TH}
    onthefly::Bool
end
```

The structure also stores the residual. The `onthefly` Boolean tells the solver how to do the interprecision transfers. The easy way to get started is to use the `mplu` command directly on the matrix. That will build the `MPLu!`, follow that with the factorization of `AL`, and put in all in a structure that you can use with `\`.

Now we will see how the results look. In this example we compare the result with iterative refinement with `A\b`, which is LAPACK's LU. As you can see the results are equally good. Note that the factorization object `MPF` is the output of `mplu`. This is analogous to `AF=lu(A)` in LAPACK.

```

71 julia> using MultiPrecisionArrays
72
73 julia> using MultiPrecisionArrays.Examples
74
75 julia> using BenchmarkTools
76
77 julia> N=4096;
78
79 julia> G=Gmat(N);
80
81 julia> A = I - G;
82
83 julia> MPF=mpfu(A); AF=lu(A);
84
85 julia> z=MPF\b; w=AF\b;
86
87 julia> ze=norm(z-x,Inf); zr=norm(b-A*z,Inf)/norm(b,Inf);
88
89 julia> we=norm(w-x,Inf); wr=norm(b-A*w,Inf)/norm(b,Inf);
90
91 julia> println("Errors: $ze, $we. Residuals: $zr, $wr")
92 Errors: 8.88178e-16, 7.41629e-14. Residuals: 1.33243e-15, 7.40609e-14
93
94 So the results are equally good.
95
96 The compute time for mpfu should be roughly half that of lu.
97
98 julia> @belapsed mpfu($A)
99 8.55328e-02
100
101 julia> @belapsed lu($A)
102 1.49645e-01
103
104 It is no surprise that the factorization in single precision took roughly half as long as the one
105 in double. In the double-single precision case, iterative refinement is a great example of a
106 time/storage tradeoff. You have to store a low precision copy of  $A$ , so the storage burden
107 increases by 50% and the factorization time is cut in half.

```

Test citations

```

104
105 Kelley (2022b) is a book. Kelley (2022a) is a paper.
106 Kelley (2023b) is the newest paper
107 The package lives here (kelley2023b?)

```

References

```

108
109 Higham, N. J. (1996). Accuracy and stability of numerical algorithms (p. xxviii+688). Society
110 for Industrial; Applied Mathematics. ISBN: 0-89871-355-2
111
112 Kelley, C. T. (2022a). Newton's method in mixed precision. SIAM Review, 64, 191–211.
113 https://doi.org/10.1137/20M1342902
114
115 Kelley, C. T. (2022b). Solving Nonlinear Equations with Iterative Methods: Solvers and
116 Examples in Julia. SIAM. ISBN: 978-1-611977-26-4

```

- 115 Kelley, C. T. (2023a). *MultiPrecisionArrays.jl*. <https://github.com/ctkelley/MultiPrecisionAr->
116 rays.jl. <https://doi.org/10.5281/zenodo.7521427>
- 117 Kelley, C. T. (2023b). *Newton's method in three precisions*. <https://arxiv.org/abs/2307.16051>

DRAFT