Implementing Mixed-Precision Iterative Refinement in STRUMPACK

Michael Neuder with support of Dr. Pieter Ghysels

AM 205 Final Project

1 Introduction

For my final project, I wanted to contribute to an open source numerical analysis library. On the recommendation of Dr. Rycroft, I explored the Scalable Solvers Group at Lawrence Berkeley National Lab and found the `STRUMPACK` project. I reached out to Dr. Pieter Ghysels, who is the lead developer, and he was gracious enough to help me start contributing and suggested this project for me to work on. I included the important pieces of code I wrote for this project in the `code/` directory of the submission directory. This is because the files I edited in the live repo are changing as the library is under active development, so attaching the code here is the best way to demonstrate what I wrote, even though it isn't directly runnable. See the installation instructions if you want to run the library locally.

My project was implementing a mixed-precision routine to factor and solve the matrix equation $Ax = b$. In this mixed-precision routine, the factorization is done in single precision, `float` in `C++`, while the calculation of the residual and correction terms are done in, `double` in `C++`. This can lead to a significant performance improvement over doing the entire algorithm in double precision, while still providing an answer accurate to double precision. This improvement comes from the fact that single precision operations can be executed more efficiently on the hardware. Since the factorization is the bottleneck of the algorithm (the only $O(n^3)$ step), doing the factorization in single precision allows for a dramatic speedup. I found that the mixed-precision solver out performs the original solver on both CPU and GPU hardware.

Outline: The remainder of Section 1 outlines the prerequisite content. Section 2 describes the implementation of the mixed-precision solver. Section 3 presents the test equation used to benchmark as well as the performance results on both consumer grade and high performance computing CPUs and GPUs. Section 4 presents an initial analysis of how the mixed-precision solver performs on ill-conditioned matrices. Section 5 concludes and presents avenues for future work.

1.1 STRUMPACK

The STRUctured Matrix PACKage (STRUMPACK) is a C++ software library providing linear algebra routines. See Ghysels et al. [2017], Ghysels et al. [2016], and Rouet et al. [2016] for context on the different functionality of the package. I implemented the StrunmpackSparseSolverMixedPrecision class (see StrumpackSparseSolverMixedPrecision.hpp in the code/ directory attached with this writeup). Additionally, I added matrix casting support to the DenseMatrix and CSRMatrix classes (see DenseMatrix.hpp & CSRMatrix.hpp respectively). Finally, I wrote two tests to evaluate the performance of the mixed-precision solver. First, testPoisson3dMixedPrecision.cpp benchmarks the performance of the mixed-precision solver versus the original solver. Second, testHilbertMixedPrecision.cpp comparse the two solvers perfromance on example ill-conditioned matrices.

1.2 Iterative refinement

Iterative refinement is a simple algorithm to improve the accuracy of an approximate solution $\hat{x}_n$ to a linear system $Ax = b$ [Stewart, 1973] [Moler, 1967]. The process is broken down into the following three steps:

1. Compute the residual: $r = b - A\hat{x}_n$.

2. Solve for the correction term: $Ac = r$.

3. Modify the approximate solution: $\hat{x}_{n+1} = \hat{x}_n + c$.

This process is repeated until the $r < \epsilon$, where ϵ is the tolerance specified by the implementer. See Higham [1997] for a thorough analysis of the error associated with iterative refinement.

1.3 Mixed-precision

Buttari et al. [2007] present the idea of using a mixed-precision approach of the iterative refinement. Algorithm 1 below is presented as Algorithm 2 in Buttari et al. [2007], and it is useful to include here in order to understand exactly how the solver works. I denote operations that use single precision as subscript f and double as subscript d.

Algorithm 1: Mixed-precision iterative refinement

```
1  $A_f$ = cast_down($A_d$, float);
2  $b_f$ = cast_down($b_d$, float);
3  $L_f$, $U_f$, $P_f$ = lu_factor($A_f$);
4  $x_f$ = solve($L_f$, $U_f$, $P_f$, $b_f$);
5  $x_d$ = cast_up($x_f$, double);
6  $r_d = b_d - A_d x_d$;
7  while $r_d$ > tolerance do
8      $r_f$ = cast_down($r_d$, float);
9      $c_f$ = solve($L_f$, $U_f$, $P_f$, $r_f$);
10     $c_d$ = cast_up($c_f$, double);
11     $x_d$ += $c_d$;
12     $r_d = b_d - A_d x_d$;
13 end
```

Let's unpack this a bit. There are two types of casting taking place, referred to above as `cast_up` and `cast_down`, which map `float`→`double` and `double`→`float` respectively. On line 3, the LU factorization (with partial pivoting) takes place. Again, this is the bottleneck of the algorithm with a time complexity of $O(n^3)$, and is done in single precision. On lines 4-6, the initial guess solution of x is calculated and cast to double in order to solve for the initial value of the residual, r_d. The while loop continues until this residual is below a specified tolerance. In my implementation, I set this tolerance to 1×10^{-15} because this is roughly the precision we can expect for a `double` solution. The body of the while loop executes the calculation of the correction term and the modification of our solution x, which corresponds to Steps 2 & 3 in Section 1.2. In total, we can see that only the calculation of the residual and the update to the intermediate solution are done in double precision. See Appendix A Buttari et al. [2007] which uses results from Higham [2002] to perform an error analyses of Algorithm 1.

2 Implementation

With this prerequisite information, I present my implementation of mixed-precision iterative refinement for `STRUMPACK`. The implementation is best understood in two parts: 1. Matrix casting (Section 2.1) 2. Mixed-Precision Solver (Section 2.2).

2.1 Matrix casting

`STRUMPACK` implements many different matrix representations. I needed to add casting support for the `CSRMatrx` and `DenseMatrix` classes.

2.1.1 `CSRMatrix`

The `CSRMatrix`, or Compressed Sparse Row Matrix, class is used to store the matrix A. This representations takes advantage of the fact that A is sparse by only storing the values of the non-zero elements of the matrix. The representation is composed of three parts, which can be found in the `CompressedSparseMatrix.hpp` file:

1. A vector of row indices. $\text{std::vector<integer\_t> ptr\_;}$
2. A vector of column indices. $\text{std::vector<integer\_t> ind\_;}$
3. A vector of values. $\text{std::vector<scalar\_t> val\_;}$

Before describing each of these it is worth noting that this class is templated on two types: `integer_t` and `scalar_t`. The `integer_t` dictates what type of integer to use for the vectors of 1 & 2 above, which end up just being indices of the matrix. For the remainder of this work, we will let `integer_t` = `int`, because the matrices we test on do not need extended integer representations. The `scalar_t` is the type of the values that are elements of the matrix, and is what the casting code needs to switch (e.g., we need to map CSRMatrix<float,int> → CSRMatrix<double,int>). Now consider 1,2, & 3 from above:

1. The i^{th} element of this vector represents the starting point of the i^{th} row in the column indices vector, ind_.
2. The j^{th} element of this vector represents the column of the j^{th} value in the values vector, val_.
3. This vector holds the values of the matrix.

This is best understood through an example, which I will copy from the `STRUMPACK` documentation. Consider the sparse matrix A:

$$A = \begin{bmatrix} 8.2 & 0.1 & & & 3.1 \\ 0.0 & & -4.8 & & \\ 6.2 & 1.1 & & 2.6 & \\ & & -1.0 & & \\ & & & 99.9 & 4.0 \end{bmatrix},$$

where the empty elements are zero. Then the vectors 1,2, & 3 would be:

1. $\text{std::vector<integer\_t> ptr\_} = [0,\ 3,\ 5,\ 8,\ 9,\ 11]$
2. $\text{std::vector<integer\_t> ind\_} = [0,\ 1,\ 4,\ 0,\ 2,\ 0,\ 1,\ 3,\ 2,\ 3,\ 4]$
3. $\text{std::vector<integer\_t> val\_} = [8.2,\ 0.1,\ 3.1,\ 0.0,\ -4.8,\ 6.2,\ 1.1,\ 2.6,\ -1.0,\ 99.9,\ 4.0]$.

So vector 3 contains all of the values from A. Vector 2 contains the indices of each value in it's respective row. Vector 1 contains the starting point of each row in Vector 2, as well as the end index of the last row.

With this understanding of how matrices are stored into `CSRMatrix` objects, it is clear that in order to change the `scalar_t` of the matrix, we only need to change the type of Vector 3, which holds the actual values of the matrix. This cast can be done implicitly in the construction of a `std::vector`, which allows the final version of the casting code to be quite simple. See attached `CSRMatrix.cpp`.

2.1.2 `DenseMatrix`

The `DenseMatrix` class is used to represent all the vectors in the iterative refinement algorithm (x, c, r, b). This class is a much simpler representation. The values of the matrix are again templated on `scalar_t`, and are stored in a raw pointer called `data_`, see `DenseMatrix.hpp`. Thus the casting of the matrix just involves iterating over all the elements of the matrix and casting the values to the new type `cast_t`. See attached `DenseMatrix.cpp`.

2.2 Mixed-Precision Solver

With the matrix casting code in place, I was able to implement the Mixed-precision iterative refinement solver. The `StrumpackSparseSolverMixedPrecision` class implements the solver (see attached `StrumpackSparseSolverMixedPrecision.hpp`). The mixed-precision solver is templated on a `factor_t` as well as `refine_t` (in the typical case `factor_t=float` and `refine_t=double` as described in Algorithm 1). This class is templated to be compatible with real numbers (`float, double`) as well as complex numbers (`std::complex<float>, std::complex<double>`). I think it is easiest to understand how the solver works by first walking through a small example of the usage.

Listing 1: Simplified Mixed-precision solver usage

```
#include "StrumpackSparseSolverMixedPrecision.hpp"

int main() {
    // Construct mixed-precision solver and set tolerance.
    SparseSolverMixedPrecision<float,double,int> spss_mixed;
    spss_mixed.options().set_abs_tol(1e-15);

    // Specify dimension of A, which will be (NxN).
    const int N = 25

    // Make use of auxiliary function to generate A.
    CSRMatrix<double,int> A = GenerateMatrix(N);

    // Constructing b and x. Notice that b is const.
15      const DenseMatrix<double> b = GenerateRandomVector(N);
16      DenseMatrix<double> x = GenerateZeroVector(N);
17
18      spss_mixed.set_matrix(A);
19      spss_mixed.factor();
20      spss_mixed.solve(b, x);
21  }
```

Now let's take a closer look at what is going on.

1. `Line 5-6:` The solver is created with the template arguments `float, double, int`. This means that the factorization will take place with `float` but the residual will be calculated with `double`. The third template parameter of `int` can be ignored (it is just what type to use to index the matrices). Line 6 sets the tolerance to near the limit of a double precision number.

2. `Line 8-12:` First we specify the size of our matrix to be 25, and then we construct A using an auxiliary function (omitted from code sample for brevity). Notice that the values of the matrix have type `double`.

3. `Line 14-16:` Now we construct x and b which are both dense matrices with type `double`. Notice that we initial b to some random vector (could alternatively be a solution specified by the user), and also that it is `const`, meaning our algorithm will not modify any of the values. We initalize x to zeros because its state doesn't matter as it will be overwritten by the solver.

4. `Line 18-20:` First, we set the matrix of the solver. Then we factor A and solve for x.

This is the interface and example usage. Now let's go a bit deeper to understand how this works. One of the most important features of the mixed-precision solver is that it owns an private instance of the original `StrumpackSparseSolver` that is templated on `float`. When the `.set_matrix` interface is called on the mixed solver, two copies of the matrix are saved. The first remains in `double` and is owned directly by the mixed solver. The second is a version cast to `float` and used to initialize the `StrumpackSparseSolver`. Next, `.factor()` is called, which simply calls the exact same interface on the `StrumpackSparseSolver`. In this way, the factorization is done in single precision. Hopefully this has all been pretty straightforward. The last piece of the puzzle is understanding what happens when we call `.solve(x,b)` on the mixed solver.

2.3 Solve

This is where the bulk of Algorithm 1 takes place. At this point, these are the components that we have:

1. A factorized version of A, that is type `float`.

2. A non-factorized version of A, that is type `double`.
3. The vectors b, x which are both type `double`.

Now we enter the while loop of Algorithm 1. At each iteration, we solve for the correction term, cast the correction to `double` and adjust our solution x before checking if the residual is below the tolerance. The key to understanding here is that the solve for the correction term is done on the `StrumpackSparseSolver`, which again is templated on `float`. In code:

Listing 2: Simplified Mixed-precision solve

```
 1 auto solve_func = [&](DenseMatrix<double>& r) {
 2     DenseMatrix<float> new_x(r.rows(), r.cols());
 3     DenseMatrix<float> cast_b = cast_matrix<double,float>(r);
 4     // Solve for correction term.
 5     solver_.solve(cast_b, new_x);
 6     // Set r equal to correction.
 7     r = cast_matrix<float,double>(new_x);
 8 };
 9 iterative::IterativeRefinement<double,int>(
10     A_d, solve_func, x_d, b_d);
```

First, look at the call to `iterative::IterativeRefinement` on line 7. This function handles the calculation of the residual as well as the updates to x, which is why it is passed A_d, x_d, b_d. The second argument to the iterative refinement call is a `std::function` that takes the residual and sets overwrites the residual value with the correction term. This is what the lambda function on lines 1-8 is doing. Lastly, notice that the `solve` on line 5 is called on the `float` solver, but the residual and correction terms are still in `double`, which is why the cast calls at lines 3 and 7 are necessary. See attached `StrumpackSparseSolverMixedPrecision.hpp` for the extra details of the solve.

3 Results

Now we can examine the results of benchmarking this code on different hardware. We analyze the performance by comparing it to the performance of the original `StrumpackSparseSolver` that is templated on `double`.

3.1 Poisson Equation

We use the 3-dimensional Poisson Equation to evaluate the performance of the mixed-precision solver:

$$\begin{aligned} -\nabla^2 u &= f(x, y, z) \qquad (1) \\ &= -\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right). \end{aligned}$$

We use the finite difference approximation of the second derivative for each variable:

$$\begin{aligned}\frac{\partial^2 u}{\partial x^2} &\approx \frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{\Delta x^2} \\ \frac{\partial^2 u}{\partial y^2} &\approx \frac{u_{i,j+1,k} - 2u_{i,j,k} + u_{i,j-1,k}}{\Delta y^2} \\ \frac{\partial^2 u}{\partial z^2} &\approx \frac{u_{i,j,k+1} - 2u_{i,j,k} + u_{i,j,k-1}}{\Delta z^2}.\end{aligned} \tag{2}$$

Let $\Delta_x = \Delta_y = \Delta_z = h$. Then we have the full discretization:

$$-\nabla^2 u \approx \frac{u_{i+1,j,k} + u_{i,j+1,k} + u_{i,j,k+1} - 6u_{i,j,k} + u_{i-1,j,k} + u_{i,j-1,k} + u_{i,j,k-1}}{h^2}. \tag{3}$$

Let n be the number of grid points used for each coordinate. Figure 1 shows the sparsity pattern of this differentiation matrix for $n = 4$. This is a good test equation because it is structured, sparse, square matrix and the dimension grows as n^3, so it is easy to get very large matrices.

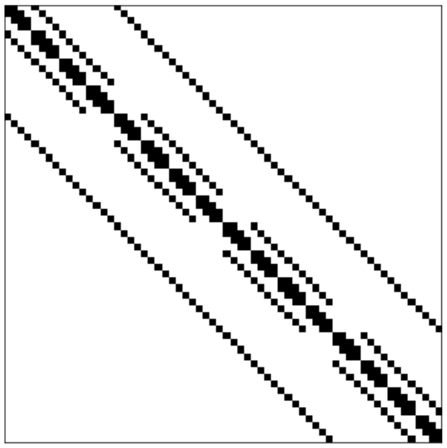

Figure 1: Sparsity pattern of the 3d Poisson Equation differentiation matrix for $n = 4$.

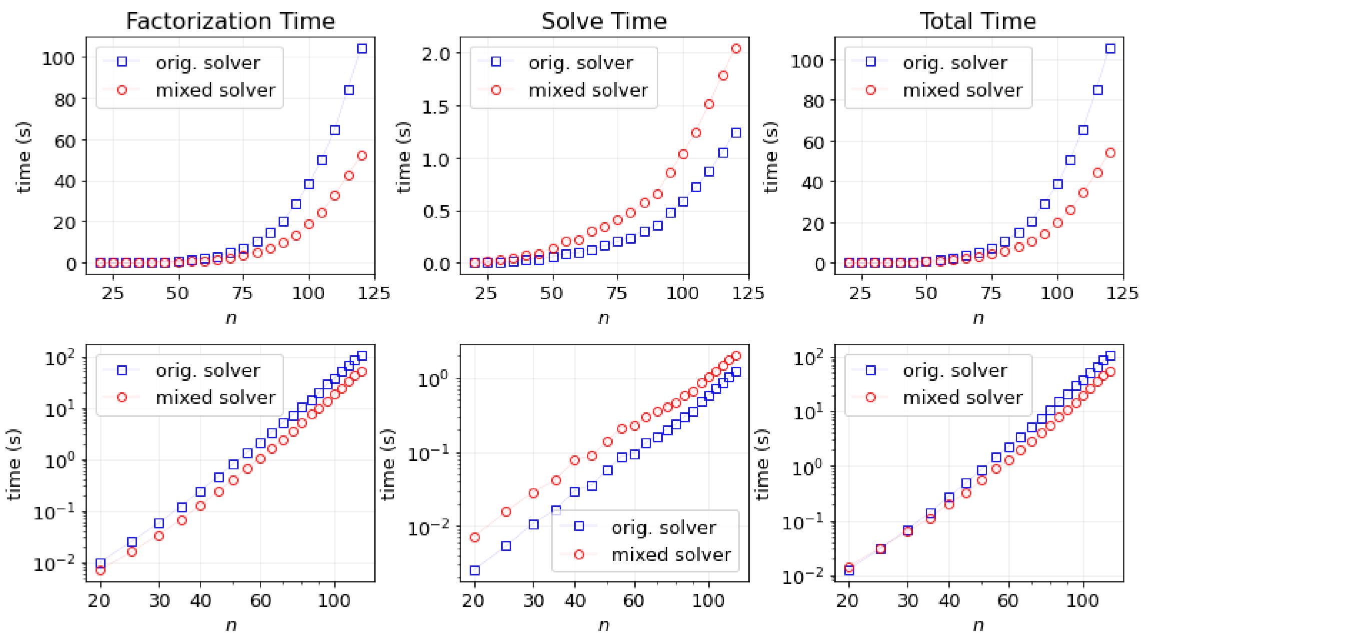

Figure 2: Results for the Ryzen 9 3950X CPU. The top row shows results on linear axes, while the second row is log-log.

3.2 Consumer-grade hardware

First we can examine the performance of the mixed-precision solver on consumer-grade CPUs and GPUs.

3.2.1 AMD Ryzen 9 3950X CPU

Figure 2 Shows results for the Ryzen 9 3950X CPU, which is a high-end desktop processor. The top row of plots show the data on linear axes, while the bottom row shows the same data on a log-log scale. The three columns correspond are factorization time, solve time, and total time. In each plot the blue squares represent the original, non-mixed, solver, while the red circles represent the mixed-precision solver.

We see that the factorization is much faster on the mixed solver, because it is being done in `float` instead of `double`. This is where we get most of our performance improvement. In the solve, the mixed solver is takes longer. This also matches our expectation because the iterative refinement will require several iterations to reach the tolerance of 10^{-15}, because it is doing the inner solve in single precision while updating the residual and the correction term in double precision. The total time ends up being much faster for the mixed-precision solver. Again this is what we expect because the dominant term of the complexity of the algorithm is the LU factorization. Through the use of SIMD[1] instructions we can expect

[1]single instruction multiple data

that floating point operations in single precision will be twice as fast as double precision. We can check this by taking the raio of the total time for each solver. This ratio for the last six data points, $n = 95, 100, 105, 110, 115, 120$, is,

$$\begin{bmatrix}2.00303985 & 1.95618525 & 1.9660357 & 1.89945729 & 1.93212333 & 1.94289973\end{bmatrix}.$$

Thus we can conclude that we are almost getting a full 2x speedup by using the mixed-precision solver.

3.2.2 Nvidia GeForce RTX 2060 GPU

The performance improvement of the mixed-precision solver is even more evident when using consumer grade GPUs.

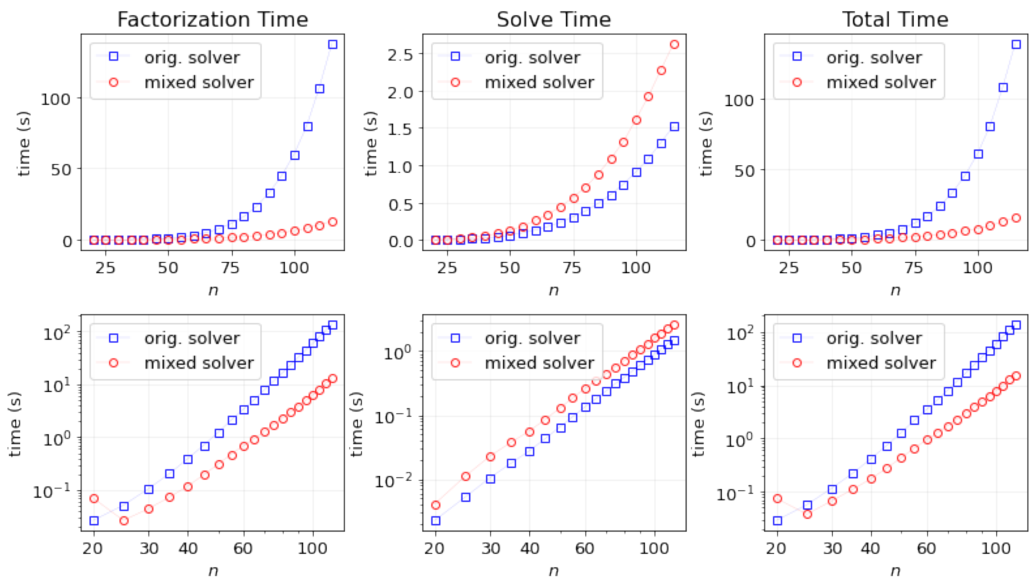

Figure 3: Results for the Nvidia GeForce RTX 2060 GPU.

In this case, the hardware can execute single precision operations about 8x faster than double precision. Again looking at the ratio of the last 6 data points we see very promising results:

$$\begin{bmatrix}6.41441035 & 6.90530708 & 7.28564254 & 7.74923386 & 8.06006752 & 8.5145143\end{bmatrix}.$$

So we are almost maxing out the performance improvement that we hope for.

3.3 High Performance Computing Hardware

We also were able to get performance results for better hardware. Dr. Ghysels has access to the Summit supercomputer at Oak Ridge, which has some of the best hardware available.

Since HPC hardware is optimized for double precision operations, we find our results are reduced. First, we ran the benchmark on a IBM Power9 CPU. I will exclude the full image results, but again the last five ratios of the original over mixed-precision total time are:

$$\begin{bmatrix}1.40328584 & 1.48871594 & 1.55996121 & 1.55520713 & 1.62510489 & 1.60323937\end{bmatrix}.$$

So in this case we are getting closer to a 1.5x improvement, which is still decent, but not as conclusive as our 2x on the AMD CPU. The last piece of hardware we ran the benchmark on was a Nvidia Tesla v100 GPU, which is really one of the top pieces of hardware available. The improvements were:

$$\begin{bmatrix}1.82562197 & 1.6966471 & 1.77399679 & 1.78512899 & 1.80887783 & 1.87944244\end{bmatrix}.$$

These seem to be nearing 2x, which is about the best we can hope for on this specific hardware. So overall, the user will see much larger performance improvements on consumer hardware as compared to HPC-grade.

One last visualization that helps emphasize why the mixed precision solver out performs the original is looking at the percentage of the time spent doing each component of the algorithm, which is shown in Figure 4.

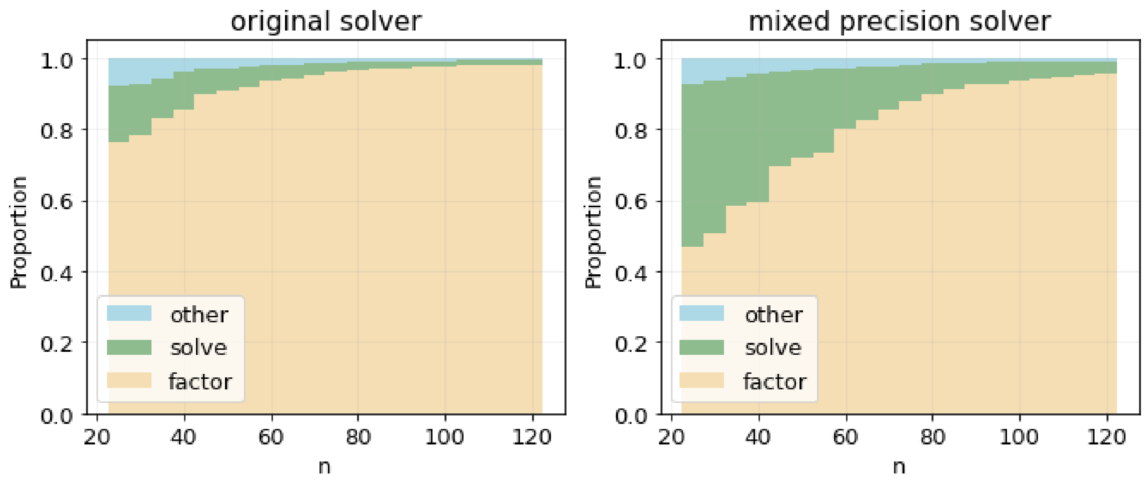

Figure 4: Proportion of time spent on each component for the AMD CPU as a function of n.

We can see that as n grows, the factorization begins to dominate the runtime for each of the solvers. This makes sense because it has the highest computational complexity. We see that for the mixed-precision solver, a bigger chunk of the time is spent on the solve, which also makes sense because more iterative refinement loops are required.

4 Ill-conditioned matrices

One last analysis that I wanted to explore per the recommendation of Dr. Rycroft was how the mixed precision solver worked on ill-conditioned matrices. Because the factorization takes place in single precision, we would expect that the mixed precision solver would perform worse than a fully double precision solver. To test this, I compared how the original vs mixed-precision solvers performed on the Hilbert Matrix. The Hilbert Matrix is a $n \times n$ square matrix where each element is defined as,

$$H(i,j) = \frac{1}{i+j+1} \text{ for } i = 0, 1, ..., n-1 \text{ and } j = 0, 1, ..., n-1.$$

So the 5×5 Hilbert Matrix is,

$$H = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 & 1/5 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 \\ 1/4 & 1/5 & 1/6 & 1/7 & 1/8 \\ 1/5 & 1/6 & 1/7 & 1/8 & 1/9 \end{bmatrix}.$$

This matrix is ill-conditioned. Figure 5 plots the condition number and minimum Eigen value for the Hilbert matrix of size $n \times n$.

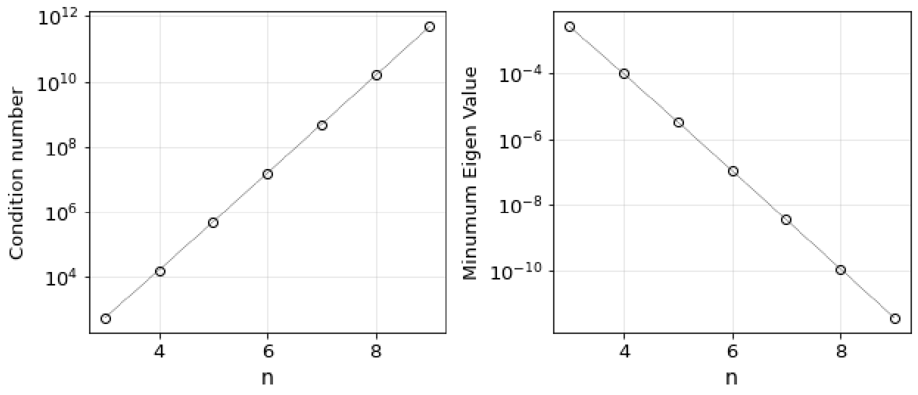

Figure 5: Condition number and minimum Eigen values of the Hilbert matrix as a function of n.

When running the mixed-precision solver on this matrix, I found that it did not find a solution past $n = 7$. This value makes sense because at that point the minimum Eigen value is falling below the accuracy of a single precision float (around 10^{-8}). The full double precision solver had no trouble finding the results all the way up to $n = 13$, at which point the minimum Eigen values are below 10^{-16} and the results will stop making sense. So this demonstrates that in the case of ill-conditioned matrices, it is a much better option to use the fully `double` solver as opposed to the mixed-precision solver.

5 Conclusion

There are many extensions to this work. The simplest next step is to add support for mixed-precision solvers to distributed matrices. STRUMPACK supports distributed memory matrices through the open MPI interface and it would be cool if those matrices could also be solved with mixed-precision. This involves writing the matrix casting code for distributed matrices, as well as making a new class of solver which uses these matrices. It would also be worthwhile to explore more thoroughly how the mixed-precision solver operates in the situation of ill-conditioned matrices. I examined only the Hilbert matrix, which is only a single example. It would be good to verify these result on more examples. There are also other iterative algorithms that may converge more rapidly than standard iterative refinement (Generalized residual method and other Krylov methods for example), and it would be good to support mixed-precision for those algorithms.

This was an excellent project and I am very grateful to Dr. Ghysels for taking the time to meet with me and help come up with an appropriate project. There is still plenty of work to do and I plan on continuing to contributing. Thanks to Dr. Rycroft for helpful discussions as well as the entire AM 205 teaching staff for an excellent course and semester!

References

Pieter Ghysels, Xiaoye Sherry Li, Christopher Gorman, and François-Henry Rouet. A robust parallel preconditioner for indefinite systems using hierarchical matrices and randomized sampling. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 897–906. IEEE, 2017.

Pieter Ghysels, Xiaoye S Li, François-Henry Rouet, Samuel Williams, and Artem Napov. An efficient multicore implementation of a novel hss-structured multifrontal solver using randomized sampling. *SIAM Journal on Scientific Computing*, 38(5):S358–S384, 2016.

François-Henry Rouet, Xiaoye S Li, Pieter Ghysels, and Artem Napov. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Transactions on Mathematical Software (TOMS)*, 42(4):1–35, 2016.

Gilbert W Stewart. *Introduction to matrix computations.* Elsevier, 1973.

Cleve B Moler. Iterative refinement in floating point. *Journal of the ACM (JACM)*, 14(2): 316–321, 1967.

Nicholas J Higham. Iterative refinement for linear systems and lapack. *IMA Journal of Numerical Analysis*, 17(4):495–509, 1997.

Alfredo Buttari, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Jakub Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *The International Journal of High Performance Computing Applications*, 21(4): 457–466, 2007.

Nicholas J Higham. *Accuracy and stability of numerical algorithms.* SIAM, 2002.