An Adaptive Solver for Systems of Linear Equations

Conrad Sanderson [†‡] and Ryan Curtin [◇]

[†] Data61 / CSIRO, Australia
[‡] Griffith University, Australia
[◇] RelationalAI, USA

***Abstract*—Computational implementations for solving systems of linear equations often rely on a one-size-fits-all approach based on LU decomposition of dense matrices stored in column-major format. Such solvers are typically implemented with the aid of the xGESV set of functions available in the low-level LAPACK software, with the aim of reducing development time by taking advantage of well-tested routines. However, this straightforward approach does not take into account various matrix properties which can be exploited to reduce the computational effort and/or to increase numerical stability. Furthermore, direct use of LAPACK functions can be error-prone for non-expert users and results in source code that has little resemblance to originating mathematical expressions. We describe an adaptive solver that we have implemented inside recent versions of the high-level Armadillo C++ library for linear algebra. The solver automatically detects several common properties of a given system (banded, triangular, symmetric positive definite), followed by solving the system via mapping to a set of suitable LAPACK functions best matched to each property. The solver also detects poorly conditioned systems and automatically seeks a solution via singular value decomposition as a fallback. We show that the adaptive solver leads to notable speedups, while also freeing the user from using direct calls to cumbersome LAPACK functions.**

***Index Terms*—adaptive systems, numerical linear algebra, mapping problem, system of linear equations, computational implementation.**

I. Introduction

Solving systems of linear equations is a fundamental computational task in numerous fields [10], [12], including signal processing and machine learning. The general form for a system of linear equations is expressed as:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \cdots & + & a_{mn}x_n & = & b_m \end{array} \tag{1}$$

which can be compactly represented in matrix form as:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{2}$$

where matrix $\boldsymbol{A}$ with size $m \times n$ contains the coefficients of the system, vector $\boldsymbol{x}$ with size $n \times 1$ represents the unknown variables to be found, and vector $\boldsymbol{b}$ with size $m \times 1$ contains known constants.

For the common case of square-sized $\boldsymbol{A}$, a naive approach to find $\boldsymbol{x}$ is via matrix inverse, i.e. $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$. However, in practice computing the inverse is actually not necessary and can result in numerical instability, leading to inaccurate solutions [13]. An effective and general approach is to solve the system with the help of lower-upper (LU) decomposition [10], [12]. This can be typically implemented through the `xGESV` set of functions[1] available in the ubiquitous and de-facto industry standard LAPACK software [1]. Several optimised versions of LAPACK are available, such as MKL [14] for the `x86-64` architecture used by the widely employed Intel and AMD processors.

The task of converting an arbitrary linear algebra expression into an efficient sequence of optimally matched LAPACK function calls, either manually or automatically, is known as the *linear algebra mapping problem* [19]. When the conversion is done manually, the process can be laborious and error-prone; it typically requires thorough understanding of several areas: high-performance computing, numerical linear algebra, and the intricacies of LAPACK. Furthermore, source code that directly uses LAPACK functions has several downsides: it **(i)** has little resemblance to the originating mathematical expression, **(ii)** requires manual memory management, **(iii)** requires keeping track of many extra variables. In turn, these downsides reduce the readability of the source code by non-expert users, while increasing the maintenance burden and risk of bugs [16], [23].

To address the above issues and hence to increase productivity, linear algebra expressions are often implemented with the aid of high-level frameworks [25], such as Matlab, Octave [15] and Armadillo [20]. While these frameworks facilitate simplified user code that focuses on high-level algorithm logic, the automatic mapping of given mathematical expressions into LAPACK functions can be suboptimal [2], [4].

Higher level frameworks default to storing dense matrices in straightforward column-major format (where all the elements in each column are stored consecutively in memory), without taking into account any special structure or properties of the matrix. This choice is made by framework maintainers to reduce the internal code complexity of the framework, and to avoid burdening users with the choice of storage types; users may not have the inclination nor expertise to understand the advantages and disadvantages of various storage formats.

Within the above context, the dense matrix LU-based solver is an effective and often-used one-size-fits-all approach for

[1] For each LAPACK function, we substitute the first letter of the function with 'x' to indicate a set of functions which differ only in the applicable element type (eg. single- or double-precision element type). For example, the set {`SGESV`, `DGESV`, `CGESV`, `ZGESV`} is represented as `xGESV`.

* **Published in:** International Conf. Signal Processing and Communication Systems, 2020.
DOI: 10.1109/ICSPCS50536.2020.9309998

solving systems of linear equations. However, this straightforward approach does not take into account various matrix properties which can be exploited to reduce the computational effort and/or to increase numerical stability. For example, when matrix $\boldsymbol{A}$ is symmetric positive definite, it is more computationally efficient to use Cholesky decomposition instead of LU decomposition [12].

In this paper we describe an adaptive solver that we have devised and implemented inside recent versions of the open-source Armadillo C++ linear algebra library. The library provides a high-level *domain specific language* [17], [22] embedded within the host C++ language, allowing mathematical operations with matrices to be expressed in a concise and easy-to-read manner similar to Matlab / Octave. This facilitates prototyping directly in C++ and aids the conversion of research code into production environments.

The adaptive solver is able to automatically detect several common properties of matrices, while being robust to inherent limitations in the precision of floating point representations [11], [18], and then map them to a large set of suitable LAPACK functions. We show that this leads to considerable speedups, thereby freeing the user from worrying about storage formats and using cumbersome manual calls to LAPACK functions in order to obtain good computational efficiency.

We continue the paper as follows. Section II describes several common matrix properties which can be exploited, summarises the algorithms used for detecting such properties, and lists the suitable sets of LAPACK functions tailored for each property. Section III provides an empirical evaluation showing the speedups attained by the adaptive solver. Section IV provides a brief discussion on computational complexity and runtime considerations. The salient points and avenues for further exploitation are summarised in Section V.

II. Automatic Detection and Mapping

The adaptive solver detects special matrix structures in the following order: **(i)** banded, **(ii)** upper or lower triangular, **(iii)** symmetric positive definite (sympd). Examples of the structures are shown in Fig. 1. A specialised solver is employed as soon as one of the structures is detected. The detection algorithms and associated LAPACK functions for solving the systems are described in Sections II-A through to II-C. If no special structure is detected, an extended form of an LU-based solver is used, described in Section II-D. A flowchart summarising the adaptive solver is given in Fig. 2.

$$\begin{bmatrix} 1 & 9 & 0 & 0 & 0 \\ 6 & 2 & 8 & 0 & 0 \\ 0 & 7 & 3 & 7 & 0 \\ 0 & 0 & 8 & 4 & 6 \\ 0 & 0 & 0 & 9 & 5 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 6 & 0 & 0 & 0 \\ 3 & 7 & 1 & 0 & 0 \\ 4 & 8 & 2 & 4 & 0 \\ 5 & 9 & 3 & 5 & 6 \end{bmatrix} \quad \begin{bmatrix} 9 & 1 & 2 & 3 & 4 \\ 1 & 8 & 1 & 2 & 3 \\ 2 & 1 & 7 & 1 & 2 \\ 3 & 2 & 1 & 6 & 1 \\ 4 & 3 & 2 & 1 & 5 \end{bmatrix}$$

(a) (b) (c)

Figure 1: Examples of matrix structures: (a) banded, (b) lower triangular, (c) symmetric positive definite.

Each of the solvers estimates the reciprocal condition number [3] of the given system, which is used to determine the quality of the solution. If any of the solvers fail to find a solution (including the solver for general matrices), or if the system is determined to be poorly conditioned, an approximate solution is attempted using a solver based on singular value decomposition (SVD), described in Section II-E.

The code for the detection and mapping is implemented as part of the `solve()` function in Armadillo. As of Armadillo version 9.900, the `solve()` function is comprised of about 3000 lines of code, not counting LAPACK code. Example usage of the `solve()` function is shown in Fig. 3.

A. Banded Matrices

Banded matrices contain elements on the main diagonal and typically on several more diagonals, above and/or below the main diagonal. All other elements are zero. As such, computational effort can be considerably reduced by exploiting the sparseness of the matrix. Fig. 1(a) shows an example banded matrix.

To efficiently detect a banded matrix structure within a column-major dense matrix, each column is examined rather than each possible diagonal. To determine the number of super-diagonals (diagonals above the main diagonal), the elements in each column are traversed from the top of the column to the location of the main diagonal within that column, followed by noting the difference in the locations of the first non-zero element and the main diagonal. The maximum of all noted differences is taken as the number of super-diagonals. To determine the number of sub-diagonals (diagonals below the main diagonal), the elements in each column are traversed from the location of the main diagonal within that column to the bottom of the column, followed by noting the difference in the locations of the main diagonal and the last non-zero element. The maximum of all noted differences is taken as the number of sub-diagonals.

As soon as the number of detected diagonals indicates that more than 25% of the elements within the matrix are non-zero, all further processing is stopped and the matrix is deemed to be non-banded. The threshold of 25% was determined empirically as a good trade-off between the lower computational requirements of solving a banded system, and the amount of extra processing required for both the detection and further wrangling of data in the banded structure.

If a banded matrix structure is detected, data from the diagonals must be first converted into a relatively compact storage format, where each diagonal is stored as a vector in a separate dense matrix [1]. Data in the compact storage format is then used by the LAPACK function `xGBTRF` to compute the LU decomposition of the banded matrix, followed by the `xGBTRS` function to solve the system based on the LU decomposition, and finally the `xGBCON` function to estimate the reciprocal condition number from the LU decomposition.

Diagonal matrices are treated as banded matrices, where the number of diagonals above and below the main diagonal is zero. While it is certainly possible to have specialised handling

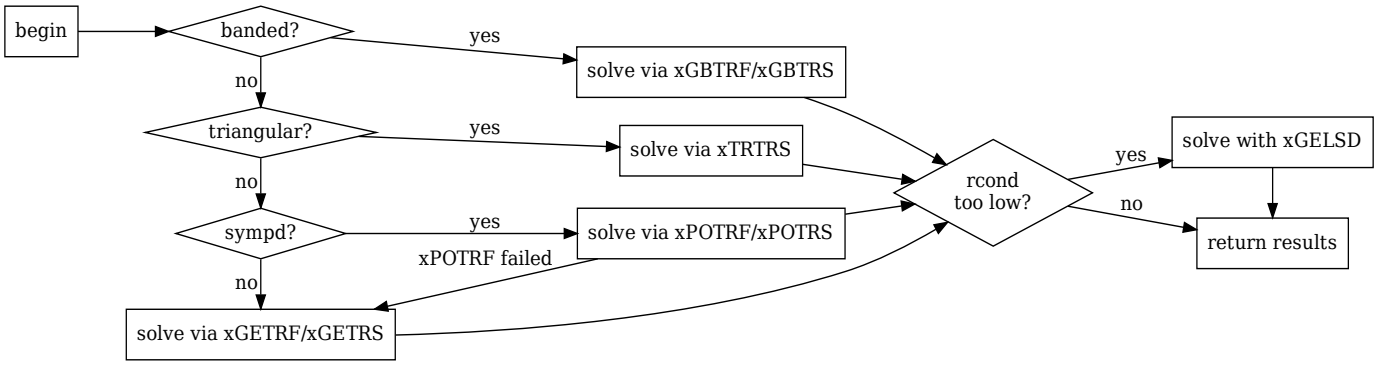

Figure 2: Flowchart for the adaptive solver. Depending on properties of the input matrix $\boldsymbol{A}$, more efficient LAPACK solvers are used when possible. In all cases, if the reciprocal condition number (rcond) of $\boldsymbol{A}$ is too low, an approximate SVD-based solver is used as a fallback.

for diagonal matrices, in practice such matrices are seldom encountered when solving systems of linear equations.

B. Triangular Matrices

For triangular matrices, the LU decomposition can be avoided and the solution can be obtained via either back-substitution (for upper triangular matrices) or forward-substitution (for lower triangular matrices) [12]. As such, considerable reductions in computational effort can be attained. An example of a lower triangular matrix is shown in Fig. 1(b).

```
#include <armadillo>

using namespace arma;

int main()
  {
  // generate matrix with random values in [-0.5,+0.5] interval
  mat R = randu(100, 100) - 0.5;

  // generate symmetric matrix A via A = R'R
  mat A = R.t() * R;

  // ensure values on the the main diagonal are dominant
  A.diag() += 1.0;

  // generate random column vector
  vec B(100, fill::randu);

  // solve for X in the random sympd system AX = B
  vec X = solve(A,B);

  X.print("solution:");

  return 0;
  }
```

Figure 3: An example C++ program for solving a random sympd system. The `solve()` function as well as the `mat` and `vec` classes (for matrices and vectors) are provided by the Armadillo library [20]. Several optional arguments have been omitted for brevity. The `solve()` function is internally comprised of about 3000 lines of code, not counting LAPACK code. Documentation for all available classes and functions can be viewed at http://arma.sourceforge.net/docs.html.

The process of forward-substitution can be summarised as first finding the value for x_1 in Eqn. (1), where a_{12} through to a_{1n} are known to be zero, resulting in $x_1 = b_1/a_{11}$. The value for x_1 is then used to find x_2, where a_{23} through to a_{2n} are zero. This iterative process can be compactly expressed as:

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{ij} \cdot x_j}{a_{ii}} \tag{3}$$

The process of back-substitution follows a similar manner, starting from the bottom of the matrix instead of the top (i.e. x_n is solved first instead of x_1).

The detection of triangular matrices is done in a straightforward manner. If all elements above the main diagonal are zero, a lower triangular matrix is present. Conversely, if all elements below the main diagonal are zero, an upper triangular matrix is present. An attempt to solve the system is made by using the `xTRTRS` function, and the reciprocal condition number is computed via the `xTRCON` function.

C. Symmetric Positive Definite Matrices

A matrix $\boldsymbol{M}$ is considered to be symmetric positive definite (sympd) if $\boldsymbol{v}^\top \boldsymbol{M} \boldsymbol{v} > 0$ for every non-zero column vector $\boldsymbol{v}$, and $\boldsymbol{M}$ is symmetric (i.e.. its lower triangular part is equivalent to a transposed version of its upper triangular part, excluding the main diagonal). Fig. 1(c) shows an example sympd matrix. For solving systems of linear equations, the sympd property can be exploited by using Cholesky decomposition [12] instead of LU decomposition in order to reduce computational effort.

To determine whether a given matrix is sympd, the traditional approach is to check whether all its eigenvalues are real and positive [5]. However, this raises two complications. First, the eigen-decomposition of a matrix is computationally expensive, which would defeat the aim of saving computational effort. Second, since the matrix is stored in simple column-major format, all matrix elements must be checked to ensure that symmetry is actually present. This in turn raises a further complication, as a simple check for equality between elements does not take into account minor differences that may be present due to the accumulation of rounding errors

stemming from limitations in the precision of floating point representations [11], [18].

To address the above issues, we have devised a fast two-pass algorithm which aims to ensure the presence of several necessary conditions for a sympd matrix [5], [12]: **(i)** all diagonal elements are greater than zero, **(ii)** the element with largest modulus is on the main diagonal, **(iii)** the matrix is diagonally dominant, and **(iv)** the matrix is symmetric while tolerating minor variations in a robust manner.

We note that while the conditions checked by the algorithm are necessary, they are not sufficient to absolutely guarantee the presence of a sympd matrix. However, for practical purposes, in our experience (mainly in the machine learning area) the conditions appear adequate for determining sympd presence.

The algorithm is shown in Fig. 4, which returns either *true* or *false* to indicate that a matrix is likely to be sympd. As soon as any condition is not satisfied, the algorithm aborts further processing and returns *false*. Lines 7 to 8 check whether all diagonal elements are greater than zero. Lines 12 to 14 check the presence of symmetry by determining whether the difference between an element and its corresponding transposed element is below a threshold; the difference is compared against the threshold in both absolute and relative terms, for robustness against precision variations between low-magnitude and high-magnitude floating point representations [11], [18]. We have empirically chosen the threshold as $100 \cdot \epsilon$, where ϵ is the *machine epsilon*[2]. Line 15 checks whether the element with largest modulus is on the main diagonal, while line 16 performs a rudimentary check for diagonal dominance.

If the algorithm determines that a sympd matrix is present, an attempt to solve the system is made with a combination of the following LAPACK functions: `xPOTRF`, `xPOTRS`, and `xPOCON`. The `xPOTRF` function computes the Cholesky decomposition, `xPOTRS` solves the system based on the Cholesky decomposition, and finally `xPOCON` computes the reciprocal condition number from the Cholesky decomposition.

If the `xPOTRF` function fails (i.e. the Cholesky decomposition was not found), the failure is taken to indicate that the matrix is not actually sympd. In that case, the system is solved by the generic solver described in Section II-D.

D. General Matrices

For general matrices, where no special structure has been detected, a standard LU-based solver is used. Here matrix $\boldsymbol{A}$ from Eqn. (2) is expressed as $\boldsymbol{A} = \boldsymbol{LU}$, where $\boldsymbol{L}$ is a unit lower triangular matrix, and $\boldsymbol{U}$ is an upper triangular matrix. Rewriting Eqn. (2) yields:

$$\boldsymbol{L}\,(\boldsymbol{U}\boldsymbol{x}) = \boldsymbol{b} \tag{4}$$

Solving for $\boldsymbol{x}$ is then accomplished in two steps. In the first step, Eqn. (4) is rewritten as $\boldsymbol{LY} = \boldsymbol{b}$, and a solution for $\boldsymbol{Y}$ is found. Since $\boldsymbol{L}$ is lower triangular and $\boldsymbol{b}$ is known, $\boldsymbol{Y}$ is easily found via forward-substitution (as shown in Section II-B). In the second step, $\boldsymbol{Y}$ is expressed as $\boldsymbol{Y} = \boldsymbol{U}\boldsymbol{x}$. Since $\boldsymbol{U}$ is lower triangular and $\boldsymbol{Y}$ is known, $\boldsymbol{x}$ is found via back-substitution, thereby providing the desired solution to Eqn. (2).

Rather than simply using the `xGESV` family of functions from LAPACK that do not provide an estimate of the reciprocal condition number, we use a combination of `xGETRF`, `xGETRS` and `xGECON`. The `xGETRF` function computes the LU decomposition, `xGETRS` solves a system of linear equations based on the LU decomposition, and finally `xGECON` computes the reciprocal condition number from the LU decomposition.

E. Fallback for Poorly Conditioned Systems

If the above solvers fail or if the estimated reciprocal condition number indicates a poorly conditioned system, an approximate solution is attempted using a solver based on SVD. A poorly conditioned system is detected when the reciprocal condition number is below a threshold; following LAPACK's example, we have set this threshold to 0.5ϵ, where ϵ is the machine epsilon as defined in Section II-C. The use of the SVD-based fallback solver can be disabled through an optional argument to the `solve()` function.

The SVD-based solver uses the `xGELSD` set of functions, which find a minimum-norm solution to a linear least squares problem [6], i.e. $\boldsymbol{x}$ is found via minimisation of $||\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}||_2$. In brief, the solution to the system in Eqn. (2) is reformulated as $\boldsymbol{x} = (\boldsymbol{A}^\top \boldsymbol{A})^{-1}\boldsymbol{A}^\top \boldsymbol{b}$, where $(\boldsymbol{A}^\top \boldsymbol{A})^{-1}\boldsymbol{A}^\top$ is known as the Moore-Penrose pseudo-inverse, which can be obtained via the SVD of $\boldsymbol{A}$ [12]. This reformulation is equivalent to the least squares solution.

```
1  proc likely_sympd
2    input: A (square matrix in column-major format)
3    input: N (number of rows in A)
4    output: boolean (true or false)
5    tol ← 100 · ϵ   (where ϵ is machine epsilon)
6    max_diag ← 0
7    forall j ∈ [0, N):
8      if A_{j,j} ≤ 0 then return false
9      if A_{j,j} > max_diag then max_diag ← A_{j,j}
10   forall j ∈ [0, N − 1):
11     forall i ∈ [j + 1, N):
12       delta ← |A_{i,j} − A_{j,i}|
13       if delta > tol and delta > tol · max(|A_{i,j}|, |A_{j,i}|)
14         then return false
15       if |A_{i,j}| ≥ max_diag then return false
16       if (|A_{i,j}| + |A_{j,i}|) ≥ (A_{i,i} + A_{j,j}) then return false
17   return true
```

Figure 4: An algorithm for detecting whether a given matrix A is likely to be symmetric positive definite (sympd). The matrix is assumed to have column-major storage, with its elements accessed via $A_{i,j}$, where i indicates the row and j indicates the column. Indexing starts at zero, following C++ convention.

[2]Machine epsilon (ϵ) is defined as the difference between 1 and the next representable floating point value [11], [18]. For double-precision floating point numbers on `x86-64` machines, $\epsilon \approx 2.22045 \times 10^{-16}$.

matrix size	standard	adaptive	reduction
100×100	1.48×10^{-4}	4.76×10^{-5}	67.89%
250×250	1.00×10^{-3}	1.66×10^{-4}	83.45%
500×500	5.41×10^{-3}	5.94×10^{-4}	89.02%
1000×1000	3.27×10^{-2}	2.89×10^{-3}	91.18%

Table I: Comparison of time taken (in seconds) to solve random banded systems using a standard dense solver (which ignores the banded property) against the adaptive solver (which takes into account the banded property). Average wall-clock time across 1000 runs is reported.

matrix size	standard	adaptive	reduction
100×100	1.40×10^{-4}	3.82×10^{-5}	72.78%
250×250	1.05×10^{-3}	2.78×10^{-4}	73.59%
500×500	5.48×10^{-3}	1.12×10^{-3}	79.61%
1000×1000	3.26×10^{-2}	5.19×10^{-3}	84.09%

Table II: Comparison of time taken (in seconds) to solve random triangular systems using a standard dense solver (which ignores the triangular property) against the adaptive solver (which takes into account the triangular property). Average wall-clock time across 1000 runs is reported.

matrix size	standard	adaptive	reduction
100×100	1.44×10^{-4}	1.19×10^{-4}	17.35%
250×250	1.03×10^{-3}	7.66×10^{-4}	25.62%
500×500	5.63×10^{-3}	4.27×10^{-3}	24.02%
1000×1000	3.31×10^{-2}	2.40×10^{-2}	27.54%

Table III: Comparison of time taken (in seconds) to solve random symmetric positive definite (sympd) systems using a standard dense solver (which ignores the sympd property) against the adaptive solver (which takes into account the sympd property). Average wall-clock time across 1000 runs is reported.

matrix size	standard	adaptive	overhead
100×100	1.486×10^{-4}	1.511×10^{-4}	1.627%
250×250	1.220×10^{-3}	1.223×10^{-3}	0.243%
500×500	6.056×10^{-3}	6.063×10^{-3}	0.114%
1000×1000	3.598×10^{-2}	3.605×10^{-2}	0.187%

Table IV: Comparison of time taken (in seconds) to solve random dense systems (without any special structure) using a standard dense solver against the adaptive solver (which attempts to detect special structures). Average wall-clock time across 1000 runs is reported.

III. Empirical Evaluation

In this section we demonstrate the speedups resulting from automatically detecting the matrix properties described in Section II and selecting the most appropriate set of LAPACK functions for solving systems of linear equations. The evaluations were done on a machine with an Intel Core i5-5200U CPU running at 2.2 GHz. Compilation was done with the GCC 10.1 C++ compiler with the following configuration options: `-O3 -march=native`. We used the open-source OpenBLAS 0.3.9 [26] package which provides optimised implementations of LAPACK functions.

We performed evaluations on the following set of sizes for matrix $\boldsymbol{A}$, ranging from small to large: {100×100, 250×250, 500×500, 1000×1000}. Three matrix types were used: **(i)** banded (with 5 diagonals), **(ii)** lower triangular, **(iii)** symmetric positive definite (sympd). For each matrix size and type, 1000 unique random systems were solved, with each random system solved using the standard LU-based solver (as per Section II-D) and the adaptive solver. For each solver, the average wall-clock time (in seconds) across the 1000 runs is reported. The results are presented in Tables I, II and III.

The results indicate that for all matrix types, the adaptive solver reduces the wall-clock time. On average, the reduction is most notable for banded systems, closely followed by triangular systems. While the reductions for sympd systems are less pronounced, they still show useful speedups. In general, the larger the matrix size, the larger the degree in reduction of wall-clock time.

To gauge the overhead of all the detection algorithms, we compare the time taken to solve random dense systems (without any special structure) using the standard LU-based solver against the adaptive solver. The results shown in Table IV indicate that the overhead is negligible. This is due to each detection algorithm stopping as soon as it determines that a special structure is not present.

IV. Runtime Considerations

In the previous section we showed the proposed algorithm to be empirically efficient on randomly generated linear systems. Indeed, this is partly due to the fact that the checks that we have proposed are asymptotically more efficient than the factorisations we employ. To see this, observe that the banded structure check described in Section II-A can be performed in a single pass over the matrix. For a square matrix of size $n \times n$, this is $O(n^2)$ time. Similarly, the triangular structure check described in Section II-B can also be performed in a single pass, yielding $O(n^2)$ time. Finally, the **likely_sympd** algorithm in Fig. 4, as written, can be performed as an $O(n)$ diagonal pass on $\boldsymbol{A}$ followed by a single $O(n^2)$ pass. Overall, this yields a total quadratic runtime for all of the proposed checks.

In the situation where the given matrix is triangular, we can perform forward- or back-substitution in $O(n^2)$ time, yielding an asymptotic improvement over the standard LU-based solver, which takes $O(n^3)$ time [12]. Cholesky decomposition and singular value decomposition both also take $O(n^3)$ time to compute [12]. However, in practice the Cholesky decomposition tends to be faster (as only one triangular part of the symmetric matrix needs to be processed), explaining the rest of the empirical advantage that was observed in our experiments.

V. Conclusion

We have described an adaptive solver for systems of linear equations that is able to automatically detect several common properties of a given system (banded, triangular, and symmetric positive definite), followed by solving the system via mapping to a set of suitable LAPACK functions best matched to each property. Furthermore, the solver can detect poorly conditioned systems and automatically obtain an approximate solution via singular value decomposition as a fallback. The automatic handling facilitates simplified user code that focuses on high-level algorithm logic, freeing the user from worrying about various storage formats and using cumbersome manual calls to LAPACK functions in order to obtain good computational efficiency.

The solver is present inside recent versions of the high-level Armadillo C++ library for linear algebra [20], [21], which allows matrix operations to be expressed in an easy-to-read manner similar to Matlab/Octave. The solver is comprised of about 3000 lines of code, not counting LAPACK code; it is able to handle matrices with single- and double-precision floating point elements, in both real and complex forms. The source code for the adaptive solver is provided under the permissive Apache 2.0 license [24], allowing unencumbered use in commercial products; it can be obtained from `http://arma.sourceforge.net`.

The adaptive solver has been successfully used to increase efficiency in various open-source toolkits, such as the *mlpack* library (`https://mlpack.org`) for machine learning [8], and the *ensmallen* library (`https://ensmallen.org`) for numerical optimisation [9]. Areas for further improvements include specialised handling of plain symmetric matrices (symmetric but not positive definite), and tri-diagonal matrices which are a common subtype of banded matrices [7].

Acknowledgements

We would like to thank our colleagues at Data61/CSIRO (Frank De Hoog, Mark Staples) and Griffith University (M.A. Hakim Newton, Majid Namazi) for discussions leading to the improvements of this paper.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1999.

[2] H. Barthels, C. Psarras, and P. Bientinesi. The sad truth about linear algebra software. XXI Householder Symposium on Numerical Linear Algebra, 2020.

[3] D. A. Belsley, E. Kuh, and R. E. Welsch. *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*. Wiley-Interscience, 1980.

[4] D. Berényi, A. Leitereg, and G. Lehel. Towards scalable pattern-based optimization for dense linear algebra. *Concurrency and Computation: Practice and Experience*, 30(22), 2018.

[5] R. Bhatia. *Positive Definite Matrices*. Princeton University Press, 2015.

[6] S. Boyd and L. Vandenberghe. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. Cambridge University Press, 2018.

[7] E. W. Cheney and D. R. Kincaid. *Numerical Mathematics and Computing*. Cengage Learning, 7th edition, 2012.

[8] R. R. Curtin, M. Edel, M. Lozhnikov, Y. Mentekidis, S. Ghaisas, and S. Zhang. mlpack 3: a fast, flexible machine learning library. *Journal of Open Source Software*, 3(26):726, 2018.

[9] R. R. Curtin, M. Edel, R. G. Prabhu, S. Basak, Z. Lou, and C. Sanderson. The ensmallen library for flexible numerical optimization. *Journal of Machine Learning Research*, 22(166), 2021.

[10] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[11] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[12] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 4th edition, 2013.

[13] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002.

[14] Intel. Math Kernel Library (MKL), 2020. https://software.intel.com/mkl.

[15] S. Linge and H. P. Langtangen. *Programming for Computations - MATLAB/Octave*. Springer, 2016.

[16] R. Malhotra and A. Chug. Software maintainability: Systematic literature review and current trends. *International Journal of Software Engineering and Knowledge Engineering*, 26(8):1221–1253, 2016.

[17] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[18] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2nd edition, 2018.

[19] C. Psarras, H. Barthels, and P. Bientinesi. The linear algebra mapping problem. *arXiv:1911.09421*, 2019.

[20] C. Sanderson and R. Curtin. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, 1:26, 2016.

[21] C. Sanderson and R. Curtin. Practical sparse matrices in C++ with hybrid storage and template-based expression optimisation. *Mathematical and Computational Applications*, 24(3), 2019.

[22] M. Scherr and S. Chiba. Almost first-class language embedding: taming staged embedded DSLs. In *ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 21–30, 2015.

[23] H. M. Sneed. A cost model for software maintenance & evolution. In *IEEE International Conference on Software Maintenance*, 2004.

[24] A. St. Laurent. *Understanding Open Source and Free Software Licensing*. O'Reilly Media, 2008.

[25] P. Viviani, M. Drocco, and M. Aldinucci. Scaling dense linear algebra on multicore and beyond: a survey. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2018.

[26] Z. Xianyi, W. Qian, and W. Saar. OpenBLAS, 2020. http://www.openblas.net.