Numerical Solution of Skew-Symmetric Linear Systems

by

Tracy Lau

B.Sc., The University of British Columbia, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

December 2009

Abstract

We are concerned with iterative solvers for large and sparse skew-symmetric linear systems. First we discuss algorithms for computing incomplete factorizations as a source of preconditioners. This leads to a new Crout variant of Gaussian elimination for skew-symmetric matrices. Details on how to implement the algorithms efficiently are provided. A few numerical results are presented for these preconditioners. We also examine a specialized preconditioned minimum residual solver. An explicit derivation is given, detailing the effects of skew-symmetry on the algorithm.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I am deeply indebted to my supervisor, Chen Greif, for tirelessly guiding and encouraging me on the journey that led to this thesis, sharing his knowledge and enthusiasm, and for supporting me in all my endeavours. Jim Varah has also assisted in my work and kindly volunteered to be a reader of this thesis, providing many helpful comments.

Scientific Computing Lab members Dan Li, Ewout van den Berg, Hui Huang, and Shidong Shan have been of much help and shown me great kindness throughout my graduate studies. I am especially grateful to Ewout for consistently going out of his way to offer his expertise and excellent advice from the very first time I stepped into the lab for my undergraduate research. I thank all my labmates for fielding my numerous questions, sharing their wisdom and experience, and for their good company and humour.

Chapter 1

Introduction

A real skew-symmetric matrix S is one that satisfies $S = -S^T$. Such matrices are found in many applications, perhaps not explicitly, but more likely as an implicit part of the problem. Any general square matrix can be split into a symmetric part and a skew-symmetric part,

$$A = \frac{A + A^T}{2} + \frac{A - A^T}{2},$$

and any non-symmetric linear system has a nonzero skew component. In some problems, the skew component may dominate the system, for example in convection-dominated convection-diffusion problems.

Here are a few basic properties of a skew-symmetric matrix A:

- Its diagonal is zero since $a_{ij} = -a_{ji} \rightarrow a_{ii} = 0 \quad \forall i$.
- For any x, $(x^T A x)^T = -x^T A x = 0$.
- If the dimension n of A is odd, then A is singular.
- A^{-1} is also skew-symmetric if it exists.
- All eigenvalues of A are either 0 or occur in pure imaginary complex conjugate pairs.

Skew-symmetric matrices can be factorized as $PAP^T = LDL^T$, where P is a permutation matrix, L is block lower triangular, and D is block diagonal with 1×1 and 2×2 blocks. This is similar to the LDL^T factorization for symmetric indefinite matrices [6, §11.1].

In this thesis, we are concerned with solving large and sparse skew-symmetric systems using iterative solvers. Thus preconditioners are the focus of the first section. With the goal of using incomplete factorizations as preconditioners, we begin in Chapter 2 by discussing general factorizations of skew-symmetric matrices. The Bunch decomposition [1] is stable with appropriate monitoring of element growth, and it can be adapted to produce incomplete factorizations. By using skew-symmetry, it halves work

and storage when compared to general LU factorizations. For an efficient algorithm for sparse matrices, however, we turn to the Crout variants of Gaussian elimination, and see how exactly to incorporate pivoting such that skew-symmetry is preserved. A few numerical experiments illustrate the performance of preconditioners generated by this procedure.

In Chapter 3, we discuss the topic of iterative solvers themselves. The various properties of skew-symmetric matrices can lead to specific variants of existing solvers. We focus on a minimum residual algorithm starting with a specialized Lanczos procedure and examining how matrix structure affects MINRES [10]. Finally, we examine the preconditioned skew-MINRES algorithm.

Thesis contributions

In the section on factorizations, we derive a new algorithm to compute incomplete factorizations of skew-symmetric matrices, very much in the spirit of [8, Algorithm 3.3] for symmetric matrices. Our algorithm is based on an incomplete Crout variant of Gaussian elimination and incorporates the partial pivoting strategy of Bunch [1].

Implementation details are given for the two main factorization algorithms we discuss. In particular we consider the issues involved with working only on half of the matrix. Also stemming from our work in MATLAB, prototype code was developed and numerical experiments were run to test our preconditioners.

Our work on MINRES unpacks the details of the derivation of the method given in [5]. We present explicit algorithms for both the unpreconditioned and preconditioned schemes.

Chapter 2

Factorizations of skew-symmetric matrices

We are interested in using iterative solvers on skew-symmetric systems, and so it is natural to explore the options for preconditioning. In particular, we look at incomplete $\mathrm{LDL}^{\mathrm{T}}$ factorizations based on the well-known ILU(0) and ILUT factorizations [11].

We start by briefly looking at two basic variants of Gaussian elimination since the order in which the elimination is executed affects the type of dropping scheme that can be implemented. The Bunch $\mathrm{LDL}^{\mathrm{T}}$ decomposition is then discussed along with some considerations for working with skew-symmetric matrices. We then review incomplete factorizations and mention how they can be generated with the Bunch decomposition. However, Crout factorizations may be better suited to generating incomplete factorizations for sparse matrices, so we derive a skew-symmetric version. Finally, a few numerical experiments examine the performance of the preconditioners that are generated by this new variant.

2.1 Gaussian elimination variants

Gaussian elimination for a general matrix A is typically presented in its KIJ form (Algorithm 2.1), so-named due to the order of the loop indices. At

Algorithm 2.1 KIJ Gaussian elimination for general matrices

```
1: for k = 1 to n − 1 do
2:    for i = k + 1 to n do
3:       a_ik = a_ik/a_kk
4:       for j = k + 1 to n do
5:          a_ij = a_ij − a_ik a_kj
6:       end for
7:    end for
8: end for
```

step k, every row below row k is modified as Schur complement updates for the $A_{k+1:n,k+1:n}$ submatrix are computed. This is fine for a dense matrix, but it is inefficient when working with matrices stored in sparse mode. In this case, the IKJ variant of Gaussian elimination (Algorithm 2.2) is more efficient, modifying only row i in step i. It accesses rows above row i, which have previously been modified, but not those below. This is referred to as a "delayed-update" variant, since all Schur complement updates for a particular element a_{ij} are computed within step i [8].

Algorithm 2.2 IKJ Gaussian elimination for general matrices

```
1: for i = 2 to n do
2:    for k = 1 to i − 1 do
3:       a_ik = a_ik/a_kk
4:       for j = k + 1 to n do
5:          a_ij = a_ij − a_ik a_kj
6:       end for
7:    end for
8: end for
```

The KIJ and IKJ algorithms produce the same complete factorization of a matrix if it has an LU decomposition [11, Proposition 10.3]. Both of these variants can be adapted in various ways to decomposing skew-symmetric matrices. Beyond modifying them to compute the block $\mathrm{LDL^T}$ factorization, the idea is to use skew-symmetry wherever possible to make gains over the plain Gaussian elimination process.

2.2 Bunch's $\mathrm{LDL^T}$ decomposition

A method for decomposing a skew-symmetric matrix A into $\mathrm{LDL^T}$ is detailed in [1]. It takes advantage of the structure of A to halve both work and storage, and stability can be increased with pivoting. It generates the factorization $PAP^T = LDL^T$ where P is a permutation matrix, L is block unit lower triangular containing the multipliers, and D is block diagonal with 1×1 and 2×2 blocks. Because the blocks are also skew, they are either 0 or of the form $\left[\begin{smallmatrix} 0 & -\alpha \\ \alpha & 0 \end{smallmatrix}\right]$. In the case of the latter, they are nonsingular in this decomposition.

We describe here a simplified version of the decomposition, and consider throughout only nonsingular matrices. This implies n is even, and that D will only have 2×2 blocks on its diagonal. When performing operations only on half the matrix, we will use the lower triangular part.

The decomposition starts by partitioning A as

$$\left[\begin{array}{c|c} S & -C^T \\ \hline C & B \end{array}\right]$$

where S is 2×2, C is $(n-2)\times 2$, and B is $(n-2)\times(n-2)$. Note that S and B are also skew-symmetric. If $S = \left[\begin{smallmatrix} 0 & -a_{21} \\ a_{21} & 0 \end{smallmatrix}\right]$ is nonsingular, i.e., $a_{21} \neq 0$, then the first step in the factorization is

$$A = \left[\begin{array}{c|c} S & -C^T \\ \hline C & B \end{array}\right] = \left[\begin{array}{c|c} I & 0 \\ \hline CS^{-1} & I \end{array}\right] \left[\begin{array}{c|c} S & 0 \\ \hline 0 & B + CS^{-1}C^T \end{array}\right] \left[\begin{array}{c|c} I & -S^{-1}C^T \\ \hline 0 & I \end{array}\right].$$

The Schur complement $B+CS^{-1}C^T$ is again skew-symmetric, so to continue the factorization, repeat this first step on the $(n-2)\times(n-2)$ submatrix. Because the submatrix in each step is skew-symmetric, the factorization can be carried out using only the lower triangular part of the matrix. At each step, A can be overwritten with the multipliers in CS^{-1} as well as the lower triangular parts of S and $B + CS^{-1}C^T$.

If S is singular, i.e. $a_{21} = 0$, there must be some i, $2 \leq i \leq n$, such that $a_{i1} \neq 0$ since A is nonsingular. Interchange the ith and second row and column to move a_{i1} into the $(2,1)$ position. This gives

$$A = P_1^T \left[\begin{array}{c|c} S & -C^T \\ \hline C & B \end{array}\right] P_1$$

where S is now nonsingular and $P_1 = P_1^T$ is obtained by interchanging the ith and second column of the identity matrix.

A high-level summary of the decomposition is given in Algorithm 2.3. The factors are obtained as L, which is the collection of CS^{-1} from each step, and D, the collection of matrices S from each step. Both L and D can be stored in half of A, and pivoting information can be stored in one vector.

Algorithm 2.3 Bunch LDL$^{\mathrm{T}}$ for skew-symmetric matrices

1: **for** $k = 1, 3, \ldots, n-1$ **do**
2: find pivot a_{pq} and permute A
3: set $S = A_{k:k+1,k:k+1}$, $C = A_{k+2:n,k:k+1}$, $B = A_{k+2:n,k+2:n}$
4: $A_{k+2:n,k:k+1} = CS^{-1}$
5: $A_{k+2:n,k+2:n} = B + CS^{-1}C^T$
6: **end for**

With regard to using only half of A, there is a small implementation issue that relates to the exact amount of storage required in total. When

computing the multiplier CS^{-1} and the Schur complement $B + CS^{-1}C^T$, it would make sense to avoid computing CS^{-1} twice. It is then natural to overwrite C with CS^{-1}, but C^T is still needed in the computation of the Schur complement. There are two straightforward ways to resolve this. If memory is not an issue, temporarily store either C or CS^{-1}, which only requires two vectors in either case, such that both are available for the Schur complement computation that follows. For the other solution, use the fact that $C^T = ((CS^{-1})S)^T$, hence C^T may be recovered with S and CS^{-1}. Since S is skew-symmetric, this amounts to multiplying two vectors with scalars and swapping their positions; in practice this only involves indexing into the proper elements if the Schur complement is computed element-wise.

Pivoting and symmetric permutations

The pivoting scheme used above only ensures that the factorization may be completed, but does nothing to control element growth in factors. In [1], Bunch proposes a partial piovting scheme that finds $\max_{2\leq p\leq n,\ 1\leq q\leq 2}\{|a_{pq}|\}$. It then moves that element into the block pivot $A_{1:2,1:2}$ by interchanging the first and qth row and column, followed by interchanging the pth and second row and column.

Of course, interchanging rows and columns should also exploit skew-symmetry and be performed only on the lower triangular part of the matrix. Let us examine a small 6×6 example to see the effects of skew-symmetry on how elements are moved around by symmetric permutations.

Example 2.1. Let A be as shown below. The largest element in magnitude in the first two columns is $a_{51} = 12$. It is already in the first column, so we need only interchange the fifth and second row and column. Let $\tilde{A}$ denote this permuted matrix.

$$A = \begin{bmatrix} 0 & -1 & -5 & -11 & -12 & -3 \\ 1 & 0 & -2 & -7 & -8 & -4 \\ 5 & 2 & 0 & -13 & -15 & -6 \\ 11 & 7 & 13 & 0 & -10 & -14 \\ 12 & 8 & 15 & 10 & 0 & -9 \\ 3 & 4 & 6 & 14 & 9 & 0 \end{bmatrix} \quad \tilde{A} = \begin{bmatrix} 0 & -12 & -5 & -11 & -1 & -3 \\ 12 & 0 & 15 & 10 & 8 & -9 \\ 5 & -15 & 0 & -13 & 2 & -6 \\ 11 & -10 & 13 & 0 & 7 & -14 \\ 1 & -8 & -2 & -7 & 0 & -4 \\ 3 & 9 & 6 & 14 & 4 & 0 \end{bmatrix}$$

We show the entire matrix only for reference; the dark shading indicates the elements that cannot be accessed. The light shading highlights positions in the lower triangular part of the matrix where elements are modified. Starting in the $(5, 2)$ position, $\tilde{a}_{52} = -a_{52}$. Elements in $\tilde{A}$ to the right of

this position take their values from elements above this position in A and vice versa. For example, we set $\tilde{a}_{53} = -a_{32}$, instead of a_{23}, which is off limits. That takes care of the elements that change sign. To the left of the second column, rows two and five are swapped giving $\tilde{a}_{21} = 12$ and $\tilde{a}_{51} = 1$. Likewise, below row five, columns two and five are swapped; $\tilde{a}_{62} = 9$ and $\tilde{a}_{65} = 4$ in this example.

In general, suppose rows and columns p and q are to be interchanged where $p < q$. Then

- $\tilde{a}_{qp} = -a_{qp}$,
- $\tilde{a}_{tp} = -a_{qt}$ and $\tilde{a}_{qt} = -a_{tp}$ for $t = p + 1 \dots q - 1$,
- $\tilde{a}_{pt} = a_{qt}$ and $\tilde{a}_{qt} = a_{pt}$ for $t = 1 \dots p - 1$, and
- $\tilde{a}_{tp} = a_{tq}$ and $\tilde{a}_{tq} = a_{pt}$ for $t = q + 1 \dots n$.

While the Bunch partial pivoting scheme can limit element growth, it does not guarantee that the elements in L are smaller than one in magnitude. Indeed, even in Example 2.1, $|\tilde{a}_{32}| = 15 > 12 = |\tilde{a}_{21}|$. Thus element growth would have to be monitored to guarantee stability, similar to Gaussian elimination with partial pivoting.

The rook pivoting strategy chooses a pivot that is maximal in both its original row and column; see [6, §9.1] for its use in general matrices. It is considered as an alternate pivoting scheme for this decomposition in [5]. We only mention here that the adaptation of rook pivoting to using only half of A is straightforward. Where either $A_{:,p}$ or $A_{p,:}$ was searched in the full matrix for its maximal entry in magnitude, now search instead $A_{p,1:p-1}$ and $A_{p:n,p}$.

2.3 Incomplete factorizations

Incomplete factorizations of the coefficient matrix of a linear system are a rich source of preconditioners [11, §10.3]. There are numerous ways to generate these, but here we focus on adapting two of the basic ones, namely ILU(0) and ILUT.

In ILU(0), the sparsity pattern of the factors matches that of the original matrix [11, §10.3.2]. In the skew-symmetric case, since operations are performed on 2×2 blocks, the analogous ILDL$^{\mathrm{T}}$(0) produces factors that match the block sparsity pattern of A. This is fairly straightforward to incorporate into the Bunch decomposition. Compute the multiplier in line 4 of Algorithm 2.3 only if the original 2×2 block in A has nonzero norm. Then

only compute the update for a block in line 5 if the corresponding multiplier from CS^{-1} has nonzero norm.

For general matrices, ILUT is a more accurate factorization than ILU(0) [11, §10.4]. It uses two dropping rules to determine which elements to replace with zero. The first rule limits the amount of work performed by ignoring elements with a small norm: if the norm of an element is below a threshold relative to the norm of its row, then set that element to zero to skip further computations with it. The second dropping rule limits the amount of memory used to store L: in each row, allow at most a fixed number of nonzero blocks, keeping those that are largest in magnitude and setting the rest to zero. The diagonal element is never dropped.

For the skew-symmetric version of ILUT, which we call ILDL$^{\mathrm{T}}$T, we again treat 2×2 blocks rather than individual elements. If ILDL$^{\mathrm{T}}$T were computed using the Bunch decomposition, then column-based dropping should be used instead of the row-based dropping described above. Again, any dropping would be performed after line 4 in Algorithm 2.3. However, the Bunch decomposition is general and not specialized for sparse matrices. When working with such matrices in a compressed storage format, it is not particularly efficient due to the pattern in which matrix entries are accessed. We now turn to Crout factorization, which is able to handle computing ILDL$^{\mathrm{T}}$T efficiently.

2.4 Crout factorization for skew-symmetric matrices

It is possible to adapt ILUT, derived from IKJ Gaussian elimination, to skew-symmetric matrices by creating a 2×2 block version that also halves both work and storage. However, preserving symmetry such that these savings are possible is impractical once pivoting is introduced. The Compressed Sparse Row storage scheme [11, §3.4], which is the data structure typically used for sparse matrices, does not easily lend itself to symmetric pivoting in IKJ Gaussian elimination. Since pivoting is often necessary, we follow [8], which discusses sparse symmetric matrices, in using a Crout variant of Gaussian elimination. As a side note, Crout factorization allows for efficient implementation of more robust dropping strategies [9], but we will not discuss them here. The full Crout LU factorization for general matrices from [11, Algorithm 10.8] is shown here in Algorithm 2.4.

The Crout form of Gaussian elimination for general matrices allows for both symmetric pivoting and the inclusion of dropping rules to produce

Algorithm 2.4 Crout LU factorization for general matrices

1: **for** $k = 1 : n$ **do**
2: **for** $i = 1 : k-1$ and if $a_{ki} \neq 0$ **do**
3: $a_{k,k:n} = a_{k,k:n} - a_{ki}a_{i,k:n}$
4: **end for**
5: **for** $i = 1 : k-1$ and if $a_{ik} \neq 0$ **do**
6: $a_{k+1:n,k} = a_{k+1:n,k} - a_{ik}a_{k+1:n,i}$
7: **end for**
8: **for** $i = k+1 : n$ **do**
9: $a_{ik} = a_{ik}/a_{kk}$
10: **end for**
11: **end for**

incomplete factorizations. Additionally, it is easily adapted to produce a block factorization. It is similar to IKJ Gaussian elimination in that it also involves a delayed update. At each step k, the kth row of U and the kth column of L are computed. Note that if A were symmetric, lines 3 and 6 would produce essentially the same numbers. The $A_{k+1:n,k+1:n}$ submatrix is not accessed.

We now derive an incomplete skew-symmetric version of Crout factorization by making modifications to Algorithm 2.4. The new version is summarized in Algorithm 2.5. This algorithm operates on 2×2 blocks and produces an $\mathrm{LDL}^{\mathrm{T}}$ factorization. As with the Bunch factorization, work and storage can be halved, and so lines 2–4 of Algorithm 2.4 are redundant for skew-symmetric matrices.

Algorithm 2.5 Incomplete skew-symmetric Crout factorization, $\mathrm{ILDL}^{\mathrm{T}}\mathrm{C}$

1: **for** $k = 1, 3, \ldots, n-1$ **do**
2: **for** $i = 1, 3, \ldots, k-2$ and $\|A_{k:k+1,i:i+1}\| \neq 0$ **do**
3: $A_{k:n,k:k+1} = A_{k:n,k:k+1} - A_{k:n,i:i+1}A_{i:i+1,i:i+1}A^{T}_{k:k+1,i:i+1}$
4: **end for**
5: apply dropping rules to $A_{k+1:n,k:k+1}$
6: **for** $i = k+2, k+4, \ldots, n-1$ **do**
7: $A_{i:k+1,k:k+1} = A_{i:i+1,k:k+1}A^{-1}_{k:k+1,k:k+1}$
8: **end for**
9: **end for**

In line 7, there is no explicit inversion of $A_{k:k+1,k:k+1}$ since

$$A_{i:i+1,k:k+1}A^{-1}_{k:k+1,k:k+1} = A_{i:i+1,k:k+1}\begin{bmatrix} 0 & -a_{k+1,k} \\ a_{k+1,k} & 0 \end{bmatrix}^{-1} = \frac{1}{a_{k+1,k}}A_{i:i+1,k:k+1}\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix},$$

which can be trivially computed element-wise. To keep with using only half of the matrix, line 3 can also be computed element-wise so that only $a_{i+1,i}$ is required from $A_{i:i+1,i:i+1}$.

Again there is the issue of overwriting A with the multipliers in L in line 3. If the full matrix were being operated on, the line would read $A_{k:n,k:k+1} = A_{k:n,k:k+1} - A_{k:n,i:i+1}A_{i:i+1,k:k+1}$. Since $A_{i:i+1,k:k+1}$ is in the upper half of A and not available, we want to use $-A^T_{k:k+1,i:i+1}$, but the required value for a particular i would have been overwritten in line 7 during the ith step of the factorization. So $A_{i:i+1,k:k+1}$ can be recovered with $A_{k:k+1,i:i+1}A_{i:i+1,i:i+1}$. Transposing and negating gives line 3 as shown.

Incorporating thresholding to obtain an incomplete LDL$^{\mathrm{T}}$ factorization only requires the addition of line 5. Either method presented in §2.3 can be used here, and the algorithm is certainly not restricted to using only these dropping schemes.

Pivoting

Bunch's partial pivoting strategy employed in the LDL$^{\mathrm{T}}$ decomposition is somewhat more involved when used in a delayed update algorithm. In Crout, once pivoting is incorporated, it is no longer true that elements in $A_{k+2:n,k+2:n}$ are not accessed at step k. As we shall see though, it will be only one column that is needed from this submatrix. Algorithm 2.6 summarizes the result of incorporating Bunch pivoting into ILDL$^{\mathrm{T}}$C.

When choosing a pivot in the usual KIJ Gaussian elimination schemes, the set of elements from which the pivot is chosen has already been modified from the original matrix in previous steps with Schur complement updates. So in Algorithm 2.5, a pivot may only be chosen after the updates in lines 2–4. In Algorithm 2.6, instead of writing these updates directly into A, we store them in a temporary block vector w. This is due to the swapping of elements to and from the submatrix $A_{k+2:n,k+2:n}$ when performing row and column interchanges.

Algorithm 2.6 ILDL$^{\mathrm{T}}$C-BP

for $k = 1, 3, \ldots, n-1$ **do**
  $w_{1:k-1,1:2} = 0$, $w_{k:n,1:2} = A_{k:n,k:k+1}$
  for $i = 1, 3, \ldots, k-2$ and $\|A_{k:k+1,i:i+1}\| \neq 0$ **do**
    $w_{k:n,1:2} = w_{k:n,1:2} - A_{k:n,i:i+1} A_{i:i+1,i:i+1} A^T_{k:k+1,i:i+1}$
  end for
  find $\max |w_{pq'}|$ s.t. $p > q'$
  set $q = q' + k - 1$
  if $p > k+1$ **then**
    $z_{1:p-1} = -A^T_{p,1:p-1}$, $z_{p:n} = A_{p:n,p}$
    for $i = 1, 3, \ldots, k-2$ **do**
      $z_{k:n} = z_{k:n} - A_{k:n,i:i+1} A_{i:i+1,i:i+1} A^T_{p,i:i+1}$
    end for
  end if
  update A with w and z
  interchange rows and columns k and q
  interchange rows and columns $k+1$ and p
  apply dropping rules to $A_{k+2:n,k:k+1}$
  for $i = k+2, k+4, \ldots, n-1$ **do**
    $A_{i:i+1,k:k+1} = A_{i:i+1,k:k+1} A^{-1}_{k:k+1,k:k+1}$
  end for
end for

For simplicity, we want to satisfy the invariant that $A_{k+2:n,k+2:n}$ is untouched after each step of the elimination. Holding the updates in a temporary vector allows us to write into A only the updates for elements that remain, after permutations are performed, in $A_{k:n,k:k+1}$. Any elements that are brought into $A_{k+2:n,k:k+1}$ due to row and column interchanges must also be updated first. Such elements lie in one column and we use the temporary vector z for this purpose. We illustrate this update process with the following example.

Example 2.2. We ignore dropping rules for a moment and step through one step of both Bunch (Algorithm 2.3) and Crout (Algorithm 2.6) factorizations, paying attention to how elements are updated in the context of pivoting. In making this comparison, we see an instance of how delayed-update algorithms differ from the usual KIJ-based algorithms.

Let us take A to be

$$A = \left[\begin{array}{cc|cccccc} 0 & -10 & -1 & -4 & -2 & -4 & -9 & -3 \\ 10 & 0 & -2 & -5 & -3 & -10 & -1 & -2 \\ \hline 1 & 2 & 0 & -0.7 & -4.9 & -11.2 & -10.3 & -2.6 \\ 4 & 5 & 0.7 & 0 & -2.2 & -9 & -3.9 & -3.3 \\ 2 & 3 & 4.9 & 2.2 & 0 & -13.8 & -12.5 & -5.5 \\ 4 & 10 & 11.2 & 9 & 13.8 & 0 & -1.4 & -11.8 \\ 9 & 1 & 10.3 & 3.9 & 12.5 & 1.4 & 0 & -10.5 \\ 3 & 2 & 2.6 & 3.3 & 5.5 & 11.8 & 10.5 & 0 \end{array}\right].$$

We will now focus only on the lower triangular part of A. Let $B^{(1)}$ and $C^{(1)}$ denote what A looks like after the first step of Bunch and Crout, respectively. (The shading will be discussed shortly.) Note that the lower right submatrix of $C^{(1)}$ is unmodified from A.

$$B^{(1)} = \left[\begin{array}{cc|cc|cccc} 0 & & & & & & & \\ 10 & 0 & & & & & & \\ \hline -0.2 & 0.1 & 0 & & & & & \\ -0.5 & 0.4 & 1 & 0 & & & & \\ -0.3 & 0.2 & 5 & 2 & 0 & & & \\ -1 & 0.4 & 11 & 7 & 13 & 0 & & \\ -0.1 & 0.9 & 12 & 8 & 15 & 10 & 0 & \\ -0.2 & 0.3 & 3 & 4 & 6 & 14 & 9 & 0 \end{array}\right]$$

$$C^{(1)} = \left[\begin{array}{cc|ccccccc} 0 & & & & & & & \\ 10 & 0 & & & & & & \\ \hline -0.2 & 0.1 & 0 & & & & & \\ -0.5 & 0.4 & 0.7 & 0 & & & & \\ -0.3 & 0.2 & 4.9 & 2.2 & 0 & & & \\ -1 & 0.4 & 11.2 & 9 & 13.8 & 0 & & \\ -0.1 & 0.9 & 10.3 & 3.9 & 12.5 & 1.4 & 0 & \\ -0.2 & 0.3 & 2.6 & 3.3 & 5.5 & 11.8 & 10.5 & 0 \end{array}\right]$$

In the second step, $k = 3$, we see from $B^{(1)}$ that the next pivot element is in the $(7, 3)$ position. Contrast this with $C^{(1)}_{7,3}$, which is not the largest element in magnitude in the third and fourth columns. The Schur complement update needs to be computed first for these two columns, and so w is loaded with $C^{(1)}_{3:8,3:4}$. After the update, w is the same as the third and fourth columns of $B^{(1)}$. Its elements below the diagonal are shaded.

With the pivot element in the $(p, q) = (7, 3)$ position, only rows and columns 4 and 7 need to be interchanged to move it into the block pivot.

Performing this on $B^{(1)}$,

$$PB^{(1)}P^T = \left[\begin{array}{cc|c|cccc} 0 & & & & & & \\ 10 & 0 & & & & & \\ \hline -0.2 & 0.1 & 0 & & & & \\ -0.1 & 0.9 & 12 & 0 & & & \\ -0.3 & 0.2 & 5 & -15 & 0 & & \\ -1 & 0.4 & 11 & -10 & 13 & 0 & \\ -0.5 & 0.4 & 1 & -8 & -2 & -7 & 0 \\ -0.2 & 0.3 & 3 & 9 & 6 & 14 & 4 & 0 \end{array}\right].$$

There are now unshaded elements in the third and fourth columns, and some shaded elements have moved to the right. This tells us which elements of w need to be written into A in the Crout factorization, namely, those corresponding to the shaded elements that are still within the third and fourth columns. Elements of w corresponding to those that are now to the right of the fourth column should not be updated.

Since $p > k+1$ in our example, this also shows which elements to the right of the fourth column in $C^{(1)}$ need to be loaded into z and updated, namely column seven. Due to the restriction to the lower-half of the matrix, z is also loaded from the seventh row, as stated in line 9 of Algorithm 2.6. The shaded elements in $C^{(1)}$ are those that need to be updated before the matrix is permuted. It is no surprise that the shading pattern here is the same as the one in $PB^{(1)}P^T$ which highlights elements originating from the third and fourth columns; it follows from how symmetric permutations are performed. The updated and permuted matrix in the Crout factorization is

$$\left[\begin{array}{cc|cc|cccc} 0 & & & & & & & \\ 10 & 0 & & & & & & \\ \hline -0.2 & 0.1 & 0 & & & & & \\ -0.1 & 0.9 & 12 & 0 & & & & \\ -0.3 & 0.2 & 5 & -15 & 0 & & & \\ -1 & 0.4 & 11 & -10 & 13.8 & 0 & & \\ -0.5 & 0.4 & 1 & -8 & -2.2 & -9 & 0 & \\ -0.2 & 0.3 & 3 & 9 & 5.5 & 11.8 & 3.3 & 0 \end{array}\right],$$

which agrees with $PB^{(1)}P^T$ in the first four columns. From here, the multipliers are computed in both factorizations, the Bunch factorization also computes Schur complements, and the second step is complete.

We summarize the update procedure for line 14 of Algorithm 2.6 with the following:

```
if q == k
  A(k+1:n,k) = w(k+1:n,1);
  if p == k+1
    A(k+2:n,k+1) = w(k+2:n,2);
  else
    A(p,k+1:p-1) = -z(k+1:p-1);
    A(p+1:n,p) = z(p+1:n);
  end
else % q==k+1
  A(k+1,k) = w(k+1,1);
  A(k+2:p-1,k+1) = w(k+2:p-1,2);
  A(p+1:n,k+1) = w(p+1:n,2);
  if p > k+1
    A(p,k:p-1) = -z(k:p-1);
    if p < n
      A(p+1:n,p) = z(p+1:n);
    end
  end
end
```

The idea is to update elements so that the pivot can be found, and then take permutations into account and write only the updates into A for elements that end up in columns k and $k+1$. There are a few equivalence classes of possible pivot locations within these two columns, and the exact update details, which are trivial and somewhat tedious, are in the code shown above.

We note that rook pivoting may again be used instead of Bunch partial pivoting here, but it is slightly more involved. Recall from §2.2 that rook pivoting on half the matrix involves looking at $A_{p,k:p-1}$ and $A_{p:n,p}$ in each step. In Algorithm 2.6, after finding the maximal element in magnitude of w, z is exactly the next row/column searched in rook. At each subsequent step of rook, another vector serving a role similar to z will be needed to temporarily store updates before finding its maximal element. This is certainly more work, but as noted in [5], the number of rook steps needed to find the pivot is typically small.

2.5 Numerical results

We compare in MATLAB the performance of preconditioners generated from the incomplete factorizations ILDL$^{\mathrm{T}}$ and ILU. We use our ILDL$^{\mathrm{T}}$C-BP routine to generate ILDL$^{\mathrm{T}}$ factors. For the general ILU factors, we use

MATLAB's built-in `luinc` routine. For the test problem, we use the skew-symmetric part of the matrix for the 3D convection-diffusion equation, discretized using the centered finite difference scheme on the unit cube with Dirichlet boundary conditions. The mesh Reynolds numbers are 0.48, 0.5, and 0.52. The grid size is 24, so A has dimension $n = 13,824$. The number of nonzeros of A is 79,488, and the number of nonzeros in its full LU factorization is 7,220,435.

Our results are presented in Table 2.1. The right-hand vector b is set

preconditioner	parameters	nnz	itn	err	res
`luinc`	1e-1	4,409,643	21	1.50	7e-7
`luinc`	1e-2	6,832,291	31	2e-5	8e-7
`ildlc-bp`	1e-2, 50	411,779	9	1.22	3e-7
`ildlc-bp`	1e-3, 50	489,190	9	1.14	1e-7

Table 2.1: Convergence results of preconditioned GMRES for test problem.

to be Ax_e where x_e is the normalized all-ones vector. Preconditioners are generated with a few different parameters for each of the methods. For `luinc` the parameter is the drop tolerance. For `ildlc-bp`, the first parameter shown is the drop tolerance and the second is the maximum number of nonzero blocks allowed per row. The number of nonzeros (nnz) reported is that of $L+U$ for `luinc` and $L+D$ for `ildlc-bp` since we need not store L^T. We use MATLAB's `gmres` with tolerance 10^{-6} and restart of 30 iterations. The remaining columns of the table show the number of iterations required for convergence and the 2-norms of the error and residual.

A plot of the relative residual norms at each step of GMRES is given in Figure 2.1 with no preconditioner and with the second and fourth preconditioners from the table. Unpreconditioned GMRES does not converge within 500 outer iterations.

With the ILDL$^{\mathrm{T}}$ preconditioners, GMRES converges to within the desired tolerance faster than with either of the ILU preconditioners. For all the preconditioned systems, the desired tolerance is achieved for the norm of the residual; however, most of them are not so well conditioned and we see that the error is relatively large except in the case of `luinc` with tolerance 1e-2. Unfortunately, the storage cost for achieving such accuracy is prohibitively large; the number of nonzeros in this `luinc` preconditioner is nearly that of the full factorization. When trying to obtain a sparser factor by using a higher drop tolerance with `luinc`, the error is similar to that for ILDL$^{\mathrm{T}}$ preconditioners. For the latter, however, the factors are much

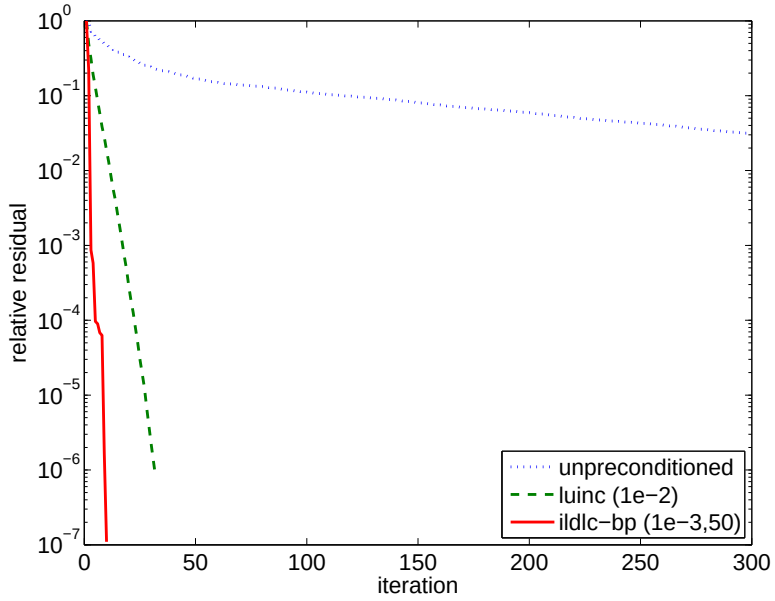

Figure 2.1: Residual vector norms for test problem.

sparser, requiring an order of magnitude less storage than either of the ILU preconditioners. This gain is beyond that coming from not storing L^T.

There is potential for further gains in using ILDL$^{\mathrm{T}}$ over ILU. Gaining insight into the parameter choice may be useful. Let us note that the factors generated were often quite ill-conditioned. It seems from some numerical experiments that this is due to large isolated eigenvalues in the preconditioned matrix. Seeking ways to improve conditioning remains an item for future work.

Chapter 3

Minimum residual iterations for skew-symmetric systems

For skew-symmetric systems, it is natural to turn to short recurrence solvers such as the ones used in the symmetric case, for example, conjugate gradient and MINRES. These save on storage that would be required when using general non-symmetric solvers such as GMRES. Both CG and MINRES are slightly different when adapted to skew-symmetric systems. Greif and Varah give a detailed account of skew-CG in [5] and summarize the derivation for skew-MINRES, including preconditioning for both. See also [7] for an unpreconditioned version of MINRES for shifted skew-symmetric systems.

In this chapter, we elaborate on skew-MINRES. We first discuss the unpreconditioned algorithm, starting with the Lanczos procedure and developing skew-MINRES, before adding preconditioning. Skew-symmetry introduces a number of zeros in predictable places, all due to the zero diagonal in the matrix. This slightly simplifies the iteration compared to that for standard symmetric systems.

3.1 Skew-Lanczos

The Arnoldi iteration computes the Hessenberg reduction of a general matrix $A = V_k H_k V_k^T$ with H_k $k \times k$ upper Hessenberg and V_k $n \times k$ orthonormal. For symmetric A, it is referred to as the Lanczos iteration and H_k is tridiagonal [2, §6.6.1]. Similarly, if A were skew-symmetric then H_k, which we shall now label T_k, is also skew-symmetric:

$$T_k^T = (V_k^T A V_k)^T = V_k^T A^T V_k = -V_k^T A V_k = -T_k.$$

This gives rise to an analogous skew-Lanczos iteration [5].

The iteration starts with the $n \times n$, skew-symmetric matrix A and some initial vector v_0. We take the first vector in V_k to be $v_1 = v_0/\|v_0\|_2$. At the kth step of the iteration, we have a sequence of vectors v_i such that

$$AV_k = V_{k+1} T_{k+1,k}. \tag{3.1}$$

The columns of V_k form an orthonormal basis of $\mathcal{K}^k(A; r_0)$. Due to skew-symmetry, $T_{k+1,k}$ is even sparser than that in the symmetric case:

$$T_{k+1,k} = \begin{pmatrix} 0 & \alpha_1 & & & & & \\ -\alpha_1 & 0 & \alpha_2 & & & & \\ & -\alpha_2 & 0 & \alpha_3 & & & \\ & & -\alpha_3 & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & \ddots & 0 & \alpha_{k-1} \\ & & & & & -\alpha_{k-1} & 0 \\ & & & & & & -\alpha_k \end{pmatrix}.$$

The kth column of (3.1) satisfies

$$Av_k = \alpha_{k-1}v_{k-1} - \alpha_k v_{k+1}.$$

Rearranging this gives the two-term recurrence,

$$v_{k+1} = (\alpha_{k-1}v_{k-1} - Av_k)/\alpha_k,$$

instead of the three-term recurrence found in the symmetric case. The α_i are chosen to normalize the v_i. If ever α_i is effectively 0, the algorithm breaks down and another vector cannot be generated. This implies $Av_i \in \mathcal{K}^i(A; r_0)$.

Algorithm 3.1 summarizes the skew-Lanczos iteration. The results of the algorithm are the set of v_i that make up V_{k+1} and the α_i that define $T_{k+1,k}$. Note that A need not be available explicitly; it suffices to have a routine that returns matrix-vector products with A.

Algorithm 3.1 Skew-Lanczos

```
1: set v_1 = b/‖b‖_2, α_0 = 0, v_0 = 0
2: for i = 1 to k do
3:    z_i = A v_i − α_{i−1} v_{i−1}
4:    α_i = ‖z_i‖_2
5:    if α_i = 0 then
6:       quit
7:    end if
8:    v_{i+1} = −z_i/α_i
9: end for
```

3.2 QR factorization of $T_{k+1,k}$

The skew-MINRES procedure that follows uses the QR factorization of $T_{k+1,k}$,

$$T_{k+1,k} = Q_{k+1}\begin{pmatrix} R_k \\ 0 \end{pmatrix} = \hat{Q}_{k+1}R_k,$$

where Q_{k+1} is $(k+1)\times(k+1)$ and R_k is $k\times k$. It is convenient to define $\hat{Q}_{k+1}$, which consists of the first k columns of Q_{k+1}. The relatively simple structure of $T_{k+1,k}$ in turn gives a relatively simple R_k, and we can derive explicit formulas for each of its nonzero elements. The orthonormal matrix Q is not needed explicitly for skew-MINRES, but it is a product of Givens rotation matrices.

The first step of skew-Lanczos produces $T_{2,1} = \left(\begin{smallmatrix} 0 \\ -\alpha_1 \end{smallmatrix}\right)$. Applying a Givens rotation,

$$G_1^T T_{21} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ -\alpha_1 \end{pmatrix} = \begin{pmatrix} \alpha_1 \\ 0 \end{pmatrix} = \begin{pmatrix} R_1 \\ 0 \end{pmatrix},$$

so $R_1 = (\alpha_1)$. Taking $Q_2 = G_1$, the first factorization is complete.

Each G_i denotes a particular rotation, but its dimension will vary depending on context, padding it with the identity matrix as necessary. In the kth iteration, G_1^T will first be applied to $T_{k+1,k}$, followed by G_2^T and so on up to G_k^T. They are all of dimension $(k+1)\times(k+1)$, and Q_{k+1} is defined as $Q_{k+1} = G_1 G_2 ... G_k$.

The second step of skew-Lanczos produces

$$T_{3,2} = \begin{pmatrix} 0 & \alpha_1 \\ -\alpha_1 & 0 \\ 0 & -\alpha_2 \end{pmatrix}.$$

Applying the first rotation matrix,

$$G_1^T T_{3,2} = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 & \alpha_1 \\ -\alpha_1 & 0 \\ 0 & -\alpha_2 \end{pmatrix} = \begin{pmatrix} \alpha_1 & 0 \\ 0 & \alpha_1 \\ 0 & -\alpha_2 \end{pmatrix}.$$

To zero out $-\alpha_2$, use the rotation $\left(\begin{smallmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{smallmatrix}\right)^T$, where c_2 and s_2 are $\alpha_1/r_{2,2}$ and $\alpha_2/r_{2,2}$, respectively. This replaces the α_1 in the second column with $r_{2,2} = \sqrt{\alpha_1^2 + \alpha_2^2}$. In full,

$$G_2^T(G_1^T T_{3,2}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & c_2 & -s_2 \\ 0 & s_2 & c_2 \end{pmatrix}\begin{pmatrix} \alpha_1 & 0 \\ 0 & \alpha_1 \\ 0 & -\alpha_2 \end{pmatrix} = \begin{pmatrix} \alpha_1 & 0 \\ 0 & \sqrt{\alpha_1^2 + \alpha_2^2} \\ 0 & 0 \end{pmatrix}.$$

To explicitly see the pattern forming, we continue for a few more iterations, generating R_k from $T_{k+1,k}$. For $k = 3$,

$$G_3^T(G_2^T G_1^T T_{4,3}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha_1 & 0 & -\alpha_2 \\ 0 & \sqrt{\alpha_1^2+\alpha_2^2} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -\alpha_3 \end{pmatrix}$$

$$= \begin{pmatrix} \alpha_1 & 0 & -\alpha_2 \\ 0 & \sqrt{\alpha_1^2+\alpha_2^2} & 0 \\ 0 & 0 & \alpha_3 \\ 0 & 0 & 0 \end{pmatrix}, \quad \text{with} \begin{cases} r_{13} = -\alpha_2 \\ r_{33} = \alpha_3 \\ c_3 = 0 \\ s_3 = 1 \end{cases}.$$

For $k = 4$,

$$G_4^T(Q_4^T T_{5,4}) = \begin{pmatrix} \alpha_1 & 0 & -\alpha_2 & 0 \\ 0 & \sqrt{\alpha_1^2+\alpha_2^2} & 0 & -\alpha_3 s_2 \\ 0 & 0 & \alpha_3 & 0 \\ 0 & 0 & 0 & \sqrt{(\alpha_3 c_2)^2+\alpha_4^2} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

with $c_4 = \alpha_3 c_2 / r_{44}$, and $s_4 = \alpha_4 / r_{44}$.

For $k = 5$, we again have $c_5 = 0$ and $s_5 = 1$.

For $k = 6$,

$$G_6^T(Q_6^T T_{7,6}) = \begin{pmatrix} \alpha_1 & 0 & -\alpha_2 & 0 & 0 & 0 \\ 0 & \sqrt{\alpha_1^2+\alpha_2^2} & 0 & -\alpha_3 s_2 & 0 & 0 \\ 0 & 0 & \alpha_3 & 0 & -\alpha_4 & 0 \\ 0 & 0 & 0 & \sqrt{(\alpha_3 c_2)^2+\alpha_4^2} & 0 & -\alpha_5 s_4 \\ 0 & 0 & 0 & 0 & \alpha_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sqrt{(\alpha_5 c_4)^2+\alpha_6^2} \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

with $c_6 = \alpha_5 c_4 / r_{66}$ and $s_6 = \alpha_6 / r_{66}$.

In summary, the QR factorization of $T_{k+1,k}$ can be stated explicitly. First define $c_0 = 1$, and the c_i and s_i are

$$c_i = \begin{cases} 0 & i \text{ odd} \\ \frac{\alpha_{i-1} c_{i-2}}{r_{ii}} & i \text{ even} \end{cases}, \qquad s_i = \begin{cases} 1 & i \text{ odd} \\ \frac{\alpha_i}{r_{ii}} & i \text{ even} \end{cases}.$$

$Q_{k+1} = G_1 G_2 \cdots G_k$ and each G_i is a Givens rotation matrix, which is essentially the identity matrix with the substitution

$$G_i(i:i+1, i:i+1) = \begin{pmatrix} c_i & s_i \\ -s_i & c_i \end{pmatrix}.$$

There are only two nonzero diagonals in R_k given by

$$r_{i,i} = \begin{cases} \alpha_i & i \text{ odd} \\ \sqrt{(\alpha_{i-1}c_{i-2})^2 + \alpha_i^2} & i \text{ even} \end{cases}, \qquad r_{i,i+2} = \begin{cases} -\alpha_{i+1} & i \text{ odd} \\ -\alpha_{i+1}s_i & i \text{ even} \end{cases}.$$

3.3 Skew-MINRES

Having developed skew-Lanczos and examined the QR factorization of the resulting skew-symmetric tridiagonal matrix, we have the tools to adapt MINRES into skew-MINRES to solve $Ax = b$ where A is skew-symmetric.

At each iteration, we seek x_k such that $\|b - Ax_k\|_2$ is minimized over the set $x_0 + \mathcal{K}^k(A; r_0)$. Thus $x_k = x_0 + V_k y_k$ for some y_k, with V_k generated by skew-Lanczos using $v_0 = r_0$.

Throughout, we can use the initial guess $x_0 = 0$ without loss of generality, for suppose we had $x_0 \neq 0$. Then define $b' = b - Ax_0 = r_0$, implying $\mathcal{K}^k(A; r_0) = \mathcal{K}^k(A; b')$, and rewrite the original problem as

$$\begin{aligned} \min_{x_k \in x_0 + \mathcal{K}^k(A;r_0)} \|b - Ax_k\|_2 &= \min_{(x_k - x_0) \in \mathcal{K}^k(A;r_0)} \|b - Ax_0 - A(x_k - x_0)\|_2 \\ &= \min_{x'_k \in \mathcal{K}^k(A;b')} \|b' - Ax'_k\|_2, \end{aligned}$$

with $x'_k := x_k - x_0$. This new problem is equivalent to the original one with the new iterates being related to the original x_k, and in particular, $x'_0 = x_0 - x_0 = 0$. Hence from here on we will consider $\mathcal{K}^k(A; r_0)$ to be $\mathcal{K}^k(A; b)$, and

$$x_k = V_k y_k. \tag{3.2}$$

Making the usual substitutions,

$$\begin{aligned} \min_{x_k} \|b - Ax_k\|_2 &= \min_{y_k} \|b - AV_k y_k\|_2 \\ &= \min_{y_k} \|b - V_{k+1} T_{k+1,k} y_k\|_2. \end{aligned}$$

Since V_{k+1} is orthonormal and $v_1 = b/\|b\|_2$, the problem becomes

$$\min_{y_k} \|\rho e_1 - T_{k+1,k} y_k\|_2, \tag{3.3}$$

where $\rho = \|b\|_2$ and e_1 is the first standard basis vector of size $k+1$.

Define $W_k = V_k R_k^{-1}$ and $z_k = \hat{Q}_{k+1}^T \rho e_1$. Combining (3.2), (3.3), and the QR factorization of $T_{k+1,k}$, the kth approximation to the solution of the system can be written as

$$x_k = (V_k R_k^{-1})(\hat{Q}_{k+1}^T \rho e_1) = W_k z_k.$$

Now we step through skew-MINRES iterations to derive expressions for W_k and z_k, beginning with the latter. Define $\hat{I}_k$ to be the first k rows of the $k+1$ identity matrix. Rewrite $z_k = \hat{Q}_{k+1}^T \rho e_1 = \hat{I}_k Q_{k+1}^T \rho e_1 = \hat{I}_k G_k^T (Q_k^T \rho e_1)$. Note $z_{k-1} = \hat{I}_{k-1}(Q_k^T \rho e_1)$. For $k = 1$,

$$z_1 = \hat{I}_1 Q_2^T \rho e_1 = \hat{I}_1 \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \rho \\ 0 \end{pmatrix} = \hat{I}_1 \begin{pmatrix} 0 \\ \rho \end{pmatrix} = 0,$$

so $x_1 = 0$ also. For $k = 2$ to 6,

$$z_2 = \hat{I}_2 \begin{pmatrix} 1 & 0 & 0 \\ 0 & c_2 & -s_2 \\ 0 & s_2 & c_2 \end{pmatrix} \begin{pmatrix} 0 \\ \rho \\ 0 \end{pmatrix} = \hat{I}_2 \begin{pmatrix} 0 \\ \rho c_2 \\ \rho s_2 \end{pmatrix} = \begin{pmatrix} 0 \\ \rho c_2 \end{pmatrix},$$

$$z_3 = \hat{I}_3 \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ \rho c_2 \\ \rho s_2 \\ 0 \end{pmatrix} = \hat{I}_3 \begin{pmatrix} 0 \\ \rho c_2 \\ 0 \\ \rho s_2 \end{pmatrix} = \begin{pmatrix} 0 \\ \rho c_2 \\ 0 \end{pmatrix},$$

$$z_4 = \hat{I}_4 \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & c_4 & -s_4 \\ 0 & 0 & 0 & s_4 & c_4 \end{pmatrix} \begin{pmatrix} 0 \\ \rho c_2 \\ 0 \\ \rho s_2 \\ 0 \end{pmatrix} = \hat{I}_4 \begin{pmatrix} 0 \\ \rho c_2 \\ 0 \\ \rho c_4 s_2 \\ \rho s_2 s_4 \end{pmatrix} = \begin{pmatrix} 0 \\ \rho c_2 \\ 0 \\ \rho c_4 s_2 \end{pmatrix},$$

$z_5 = \hat{I}_5 (0 \ \rho c_2 \ 0 \ \rho c_4 s_2 \ 0 \ \rho s_2 s_4)^T = (0 \ \rho c_2 \ 0 \ \rho c_4 s_2 \ 0)^T$, and

$z_6 = \hat{I}_6 (0 \ \rho c_2 \ 0 \ \rho c_4 s_2 \ 0 \ \rho c_6 s_2 s_4 \ \rho s_2 s_4 s_6)^T = (0 \ \rho c_2 \ 0 \ \rho c_4 s_2 \ 0 \ \rho c_6 s_2 s_4)^T$.

In general, $z_1 = 0$ and

$$z_k = \begin{pmatrix} z_{k-1} \\ \zeta_k \end{pmatrix},$$

where

$$\zeta_k = \begin{cases} 0 & k \text{ odd} \\ \rho c_k \prod_{i=2,4,\ldots,k-2} s_i & k \text{ even} \end{cases}.$$

As for W_k, only the even-indexed columns are of interest since $x_k = W_k z_k$ and all odd-indexed elements of z_k are zero. Indeed, skew-MINRES will only produce an approximation x_k for k even.

For $k = 2$, $W_2 := V_2 R_2^{-1} = \begin{pmatrix} \frac{v_1}{r_{1,1}} & \frac{v_2}{r_{2,2}} \end{pmatrix}$ and $w_2 = \frac{v_2}{r_{2,2}}$. For $k = 4, 6, \ldots$, recalling the structure of R_k gives $v_k = w_{k-2} r_{k-2,k} + w_k r_{k,k}$. Rearranging this,

$$w_k = \frac{v_k - w_{k-2} r_{k-2,k}}{r_{k,k}}.$$

In the end, each even iteration of skew-MINRES produces

$$x_k = w_2 \zeta_2 + w_4 \zeta_4 + \ldots + w_k \zeta_k.$$

Algorithm 3.2 shows the method in detail, incorporating skew-Lanczos. As in GMRES with a nonsingular A, here if $\alpha_i = 0$ for some i, then residual is also 0 and the solution has been found [11, Proposition 6.10].

Algorithm 3.2 Skew-MINRES

$p = \|b\|_2$, $\alpha_0 = 0$, $v_0 = w = 0$, $v_1 = b/p$, $c = 1$, $s = 0$, $x_0 = 0$
for $k = 1, 2, \ldots,$ until convergence **do**
  $z_k = A v_k - \alpha_{k-1} v_{k-1}$
  $\alpha_k = \|z_k\|_2$
  $v_{k+1} = -z_k / \alpha_k$
  if k even **then**
    $r = \sqrt{(\alpha_{k-1} c)^2 + \alpha_k^2}$
    $w = (v_k + \alpha_{k-1} s w)/r$
    $c = \alpha_{k-1} c / r$
    $\zeta = cp$
    $x_k = x_{k-2} + \zeta w$
    $s = \alpha_k / r$
    $p = ps$
  end if
end for

Typically, the norm of the residual is used to determine convergence and, as usual, it can be computed without calculating $b - Ax$ explicitly. Note that y_k solves $\min_{y_k} \|\rho e_1 - T_{k+1,k} y_k\|$ by satisfying $T_k y_k = \rho e_1$ [11, Proposition

6.9]. Thus the residual vector itself is

$$\begin{aligned} r_k &= b - Ax_k \\ &= \rho v_1 - AV_k y_k \\ &= \rho v_1 - V_{k+1} T_{k+1,k} y_k \\ &= \rho V_k e_1 - (V_k T_k + \alpha_k v_{k+1} e_k^T) y_k \\ &= V_k(\rho e_1 - T_k y_k) - \alpha_k v_{k+1} e_k^T y_k \\ &= -\alpha_k e_k^T y_k v_{k+1}, \end{aligned}$$

where $e_k^T y_k$ is the kth element of y_k. Its norm is given by

$$\begin{aligned} \|r_k\|_2 &= |\alpha_k e_k^T y_k| \\ &= |\alpha_k e_k^T (V_k^T W_k z_k)| \\ &= |\alpha_k e_k^T (R_k^{-1} z_k)| \\ &= |\alpha_k \zeta_k / r_{kk}| \\ &= |\zeta_k|, \end{aligned}$$

exactly as in the general case.

3.4 Preconditioned skew-MINRES

We can precondition skew-symmetric systems in a manner very similar to that in the symmetric case [4, Chapter 8] to obtain a preconditioned skew-MINRES iteration, or skew-PMINRES. To preserve skew-symmetry, a symmetric positive definite preconditioner

$$M = LL^T$$

is used in a symmetric preconditioning scheme, giving the system

$$L^{-1} A L^{-T} \hat{x} = L^{-1} b, \qquad x = L^{-T} \hat{x}. \tag{3.4}$$

Instead of searching over $\mathcal{K}^k(A; b)$ for iterates, skew-PMINRES searches over $\mathcal{K}^k(L^{-1}AL^{-T}; L^{-1}b)$, so we start by developing preconditioned skew-Lanczos. If regular skew-Lanczos (Algorithm 3.1) were run with the matrix $L^{-1}AL^{-T}$ and initial vector $\hat{v}_1 = L^{-1}b/\|L^{-1}b\|_2$, then each iteration would produce

$$\begin{aligned} \hat{z}_k &= L^{-1} A L^{-T} \hat{v}_k - \alpha_{k-1} \hat{v}_{k-1} \\ \alpha_k &= \sqrt{\hat{z}_k^T \hat{z}_k} \\ \hat{v}_{k+1} &= -\hat{z}_k / \alpha_k. \end{aligned}$$

Using this directly in a preconditioned MINRES algorithm would require having L explicitly, yet M may not always be available in factored form. Additionally, it would only compute iterates $\hat{x}_k$, and an additional solve with L^T would be required to obtain x_k. We make modifications to arrive at skew-PMINRES which addresses both of these issues. It requires only one solve with M at each iteration.

First, to combine the L and L^T solves in preconditioned skew-Lanczos, define

$$\begin{aligned} v_k &= L\hat{v}_k \\ z_k &= L\hat{z}_k \\ u_k &= L^{-T}L^{-1}v_k = M^{-1}v_k. \end{aligned}$$

Note that the v_k are no longer orthonormal; it is the $\hat{v}_k$ that are orthonormal and form a basis for $\mathcal{K}^k(L^{-1}AL^{-T}; L^{-1}b)$. Now each iteration generates

$$\begin{aligned} z_k &= LL^{-1}AL^{-T}L^{-1}v_k - \alpha_{k-1}LL^{-1}v_k \\ &= Au_k - \alpha_{k-1}v_k \\ \alpha_k &= \sqrt{(L^{-1}z_k)^T(L^{-1}z_k)} = \sqrt{z_k^T M^{-1} z_k} \\ v_{k+1} &= -z_k/\alpha_k \\ u_{k+1} &= M^{-1}v_{k+1}. \end{aligned}$$

Define $\tilde{u}_k = M^{-1}z_k$ to remove the unnecessary preconditioner solve for α_k. Then

$$\begin{aligned} \alpha_k &= \sqrt{z_k^T \tilde{u}_k} \\ u_{k+1} &= -\tilde{u}_k/\alpha_k. \end{aligned}$$

This gives Algorithm 3.3, the preconditioned skew-Lanczos routine that generates the V_k and $T_{k+1,k}$ used in skew-PMINRES.

There is only one further change to make in skew-MINRES to obtain the skew-PMINRES algorithm; one that makes the algorithm explicitly compute iterates x_k approximating the solution to $Ax = b$ rather than the $\hat{x}_k$ of (3.4).

Consider line 8 in Algorithm 3.2, $w_k = (v_k + \alpha_{k-1}s_{k-2}w_{k-1})/r_k$. The x_k generated in line 11 are simply linear combinations of the w_k, which in turn are linear combinations of the v_k. However, these are the v_k from preconditioned skew-Lanczos. To obtain $\hat{x}_k$, we need $\hat{v}_k = L^{-1}v_k$. If the solve with L^T is introduced here, then line 11 computes the x_k desired. Conveniently,

Algorithm 3.3 Preconditioned skew-Lanczos

1: $a_0 = 0$, $v_0 = 0$, $v_1 = b/\sqrt{b^T M^{-1} b}$, $u_1 = M^{-1} v_1$
2: **for** $i = 1, 2, \ldots, k$ **do**
3: $\quad z_i = A u_i - \alpha_{i-1} v_{i-1}$
4: $\quad \tilde{u}_i = M^{-1} z_i$
5: $\quad \alpha_i = \sqrt{z_i^T \tilde{u}_i}$
6: $\quad v_{i+1} = -z_i / \alpha_i$
7: $\quad u_{i+1} = -\tilde{u}_i / \alpha_i$
8: **end for**

the u_k from preconditioned skew-Lanczos are exactly $L^{-T} L^{-1} v_k$, and line 8 of Algorithm 3.2 becomes

$$w_k = (u_k + \alpha_{k-1} s_{k-2} w_{k-1}) / r_k.$$

With this we have skew-PMINRES in Algorithm 3.4. The inputs are A, b, and either M or L. It works on the preconditioned system of (3.4), and it returns approximations x_k to the solution of $Ax = b$.

Algorithm 3.4 Skew-PMINRES

1: $p = \sqrt{b^T M^{-1} b}$, $\alpha_0 = s = 0$, $c = 1$, $v_0 = w = x_0 = 0$, $v_1 = b/p$, $u_1 = M^{-1} v_1$
2: **for** $k = 1, 2, \ldots,$ until convergence **do**
3: $\quad z_k = A u_k - \alpha_{k-1} v_{k-1}$
4: $\quad \tilde{u}_k = M^{-1} z_k$
5: $\quad \alpha_k = \sqrt{z_k^T \tilde{u}_k}$
6: $\quad v_{k+1} = -z_k / \alpha_k$
7: $\quad u_{k+1} = -\tilde{u}_k / \alpha_k$
8: $\quad$ **if** k even **then**
9: $\quad\quad r = \sqrt{(\alpha_{k-1} c)^2 + \alpha_k^2}$
10: $\quad\quad w = (u_k + \alpha_{k-1} s w)/r$
11: $\quad\quad c = \alpha_{k-1} c / r$
12: $\quad\quad \zeta = cp$
13: $\quad\quad x_k = x_{k-2} + \zeta w$
14: $\quad\quad s = \alpha_k / r$
15: $\quad\quad p = ps$
16: $\quad$ **end if**
17: **end for**

For computing the residual easily in the preconditioned algorithm, the derivation is similar to that for plain skew-MINRES. The problem now solved is that given in (3.4), yet the residual of interest is still $b - Ax_k$. Instead of (3.1), we now have $(L^{-1}AL^{-T})(L^{-1}V_k) = (L^{-1}V_{k+1})T_{k+1,k}$, or

$$AM^{-1}V_k = V_{k+1}T_{k+1,k},$$

since it is the $L^{-1}v_k$ that are orthonormal here. Additionally, it is $\hat{x}_k$ that can be written as $L^{-1}V_k y_k$, for some y_k. By the definition of $\hat{x}_k$,

$$x_k = L^{-T}L^{-1}V_k y_k = M^{-1}V_k y_k.$$

Putting these together,

$$\begin{aligned} r_k &= b - Ax_k \\ &= \rho v_1 - AM^{-1}V_k y_k \\ &= \rho v_1 - V_{k+1}T_{k+1,k}y_k. \end{aligned}$$

The remainder of the derivation is exactly as shown previously in the unpreconditioned case, so again $\|r_k\|_2 = |\zeta_k|$.

Chapter 4

Conclusions and future work

We discussed the Bunch LDL^T decomposition for skew-symmetric matrices and derived a Crout-based factorization for computing incomplete LDL^T factorizations with pivoting. Implementation details have been given on how to practically save on computational work and storage by performing operations on only the lower triangular half of the matrix. This may be useful in writing more efficient implementations using compressed matrix storage schemes.

Numerical results show that a skew-symmetric preconditioner can be an improvement over the general incomplete LU factorization. For preconditioners that gave roughly the same error and residual in GMRES, the skew preconditioner was much sparser than the general one and required fewer iterations for convergence. Further work here may investigate the effects of the parameters of ILDL^{T}C-BP on the performance of preconditioned GMRES. Another item is to explore improving the conditioning of the factors, since those produced by our factorization were not as robust as expected.

We have also presented the details of a skew preconditioned MINRES iteration, derived in [5], starting by adapting Lanczos to skew-symmetric systems and seeing that the structure results in skew-MINRES producing approximations every other iteration. While preconditioning was incorporated, the procedure relies on having a symmetric positive definite preconditioner. Given the gains observed when using a skew-symmetric preconditioner in GMRES, future work includes investigating whether it is possible to derive a form of MINRES that can utilize skew-symmetric preconditioners. It is not clear to us how to formulate such a MINRES procedure.

Finally, we return to the point mentioned originally that skew-symmetric systems arise from non-symmetric systems. Another potential area of work lies in incorporating the above into general non-symmetric solvers when dealing with systems that have a dominant skew part.

Bibliography

[1] J. R. Bunch. A note on the stable decomposition of skew-symmetric matrices. *Math. Comp.*, 38(158):475–479, 1982.

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

[3] I. S. Duff. The design and use of a sparse direct solver for skew symmetric matrices. *J. Comput. Appl. Math.*, 226(1):50–54, 2009.

[4] A. Greenbaum. *Iterative methods for solving linear systems*. SIAM, 1997.

[5] C. Greif and J. M. Varah. Iterative solution of skew-symmetric linear systems. *SIAM J. Math. Anal.*, 31(2):584–601, 2009.

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

[7] R. Idema and C. Vuik. A minimal residual method for shifted skew-symmetric systems. Technical Report 07-09, Delft University of Technology, 2007.

[8] N. Li and Y. Saad. Crout versions of the ILU factorization with pivoting for sparse symmetric matrices. *Electron. Trans. Numer. Anal.*, 20:75–85, 2005.

[9] N. Li, Y. Saad, and E. Chow. Crout versions of ILU for general sparse matrices. *SIAM J. Sci. Comput.*, 25(2):716–728, 2003.

[10] C. C. Page and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12(4):617–629, 1975.

[11] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelpha, PA, 2nd edition, 2003.

[12] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, 1997.