University of Maryland College Park

Institute for Advanced Computer Studies TR–92–109
Department of Computer Science TR–2973

On the Solution of Block Hessenberg Systems*

G. W. Stewart†

October, 1992
Revised December, 1992

ABSTRACT

This paper describes a divide-and-conquer strategy for solving block Hessenberg systems. For dense matrices the method is a little more efficient than Gaussian elimination; however, because it works almost entirely with the original blocks, it is be much more efficient for sparse matrices or matrices whose blocks can be generated on the fly. For Toeplitz matrices, the algorithm can be combined with the fast Fourier transform to give a new superfast algorithm.

*This report is available by anonymous ftp from `thales.cs.umd.edu` in the directory `pub/reports`.

†Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. This work was supported in part by the National Science Foundation under grant CCR 9115568.

On the Solution of Block Hessenberg Systems

G. W. Stewart

ABSTRACT

This paper describes a divide-and-conquer strategy for solving block Hessenberg systems. For dense matrices the method is a little more efficient than Gaussian elimination; however, because it works almost entirely with the original blocks, it is be much more efficient for sparse matrices or matrices whose blocks can be generated on the fly. For Toeplitz matrices, the algorithm can be combined with the fast Fourier transform to give a new superfast algorithm.

1. Introduction

This paper was motivated by the attempt to find the steady-state of Markov chains of types $M/G/1$ and $G/M/1$ (see [8] for definitions and further details). The matrix of transition probabilities of a chain of type $M/G/1$ has the block upper Hessenberg form

$$M = \begin{pmatrix} B_0 & B_1 & B_2 & B_3 & B_4 & \cdots \\ C_0 & A_1 & A_2 & A_3 & A_4 & \cdots \\ 0 & A_0 & A_1 & A_2 & A_3 & \cdots \\ 0 & 0 & A_0 & A_1 & A_2 & \cdots \\ 0 & 0 & 0 & A_0 & A_1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \end{pmatrix}. \tag{1.1}$$

The transition matrices of chains of type $G/M/1$ are block lower Hessenberg matrices, formed in analogy with (1.1). Note that after the first row and column of (1.1) are deleted, the matrix becomes block Toeplitz.

We shall return to these chains at the end of this paper. For now we are going to consider the more general problem of solving the block upper Hessenberg system $AX = B$, which we write in partitioned form as

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1,n-1} & A_{1n} \\ A_{21} & A_{22} & A_{23} & \cdots & A_{2,n-1} & A_{2n} \\ 0 & A_{32} & A_{33} & \cdots & A_{3,n-1} & A_{3n} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & A_{n-1,n-1} & A_{n-1,n} \\ 0 & 0 & 0 & \cdots & A_{n,n-1} & A_{n,n} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_{n-1} \\ X_n \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ B_3 \\ \vdots \\ B_{n-1} \\ B_n \end{pmatrix}. \qquad (1.2)$$

The diagonal blocks A_{ii} are assumed to be square of order p_i.

The system (1.2) can be solved stably by Gaussian elimination with partial pivoting. When all the blocks are dense and of order p, the elimination step takes about $\frac{3n^2p^3}{4}$ flops,[1] and the back substitution about $\frac{n^2p^2}{2}$. Unfortunately, Gaussian elimination, for all its simplicity, has its drawbacks. For sparse matrices, the rows of A may fill in, in which case the operation count will approach the count for dense matrices, and the memory required will be almost the $\frac{p^2n^2}{2}$ locations needed to store the dense matrix. Moreover, since Gaussian elimination alters the all the rows except the first, it cannot take advantage of problems in which the blocks of A can be cheaply generated on the fly.

The algorithm propose here is based on a divide-and-conquer strategy, reminiscent of certain recent algorithms for the symmetric tridiagonal eigenvalue problem [3, 4, 9]. Essentially the problem is divided by deleting a subdiagonal block of A, yielding a simpler block triangular system. This system is solved recursively by further subdividing the diagonal blocks. An updating formula is then used to patch up the solutions of the subproblems at each level of recursion. For dense matrices, the method has an operation count comparable to Gaussian elimination. However, because it does not modify the blocks of the original matrix, there is no fill-in and the algorithm can be applied to matrices in which the blocks are generated at need. Moreover, if A is block Toeplitz the calculations can be abbreviated; in fact for a scalar Hessenberg matrix, the accelerated algorithm is fast.

In this paper we will describe and analyze our algorithms at a fairly high level. Implementation details and numerical experiments will appear in another paper. In the next section we will describe the basic algorithm. In §3 we will treat the space and time complexity of the algorithm for the case where n is a power of two and all the blocks are of of the same size. In §4 we will consider variations of the

[1] This formula assumes that both n and p are reasonably greater than one. It does *not* reduce to the usual count for Gaussian elimination when $p = 1$.

algorithm for block Toeplitz matrices. Finally, in the last section, we will return to Markov chains and the problem that motivated this research.

2. The Algorithm

The algorithm is a recursive descent procedure based on a generalization of the Sherman-Morrison-Woodbury formula for the inverse of a modified matrix (e.g., see [5, §5.1]). We begin by selecting an index i ($1 \leq i < n$). Let E consist of the columns of the identity matrix corresponding to the rows of A spanned by $A_{i+1,i}$, and let F consist of the columns of the identity matrix corresponding to the columns of A spanned by $A_{i+1,i}$. If we write $A_{\text{sw}} = A_{i+1,i}$ (sw stands for south west), then

$$\hat{A} = \begin{pmatrix} A_{\text{nw}} & A_{\text{ne}} \\ 0 & A_{\text{se}} \end{pmatrix} \equiv A - EA_{\text{sw}}F^{\text{T}} \tag{2.1}$$

is block triangular, and its diagonal blocks, A_{nw} and A_{se} are themselves block Hessenberg. (To distinguish the blocks in (2.1) from those in (1.2), we call the latter *atomic* blocks.)

The heart of the algorithm is a relation between the inverse of A and $\hat{A}$. Specifically, suppose that $\hat{A}$ is nonsingular and that A_{sw} has the full rank factorization

$$A_{\text{sw}} = URV^{\text{T}},$$

where R is nonsingular and U and V have orthonormal columns. Let

$$S = RV^{\text{T}}F^{\text{T}}\hat{A}^{-1}EU,$$

and assume that $I + S$ is nonsingular. Then A is nonsingular. Moreover, if $\hat{R}$ satisfies

$$(I + S)\hat{R} = R, \tag{2.2}$$

then

$$A^{-1} = \hat{A}^{-1} - (\hat{A}^{-1}EU\hat{R})(V^{\text{T}}F^{\text{T}})\hat{A}^{-1}. \tag{2.3}$$

The proof is by direct verification: multiply the right hand side of (2.3) by $A = \hat{A} + EA_{\text{sw}}F^{\text{T}}$ and simplify.

Now let

$$P = A^{-1}EU\hat{R}.$$

Then it follows from (2.3) that

$$X = A^{-1}B = \hat{A}^{-1}B - PF^{\text{T}}\hat{A}^{-1}B = \hat{X} - PF^{\text{T}}\hat{X}. \tag{2.4}$$

Thus if we can solve the systems

$$\hat{A}\hat{X} = B \quad \text{and} \quad \hat{A}G = EU$$

and determine $\hat{R}$ satisfying (2.2), so that we can compute the patch matrix $P = GR$, then (2.4) gives a solution to the original problem. By way of nomenclature, we will call (2.1) a *tearing* of A because it tears A_{sw} out of A, and we will call P the *patch matrix* for the tearing because it patches up the solution $\hat{X}$.[2]

Because $\hat{A}$ is block triangular, we can reduce the problem of solving $\hat{A}X = B$ to one of solving two smaller block Hessenberg systems. Specifically, partition

$$\hat{X} = \begin{pmatrix} \hat{X}_{\mathrm{n}} \\ \hat{X}_{\mathrm{s}} \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} B_{\mathrm{n}} \\ B_{\mathrm{s}} \end{pmatrix} \tag{2.5}$$

conformally with (2.1). Then the system $\hat{A}\hat{X} = B$ is equivalent to the system

$$\begin{pmatrix} A_{\mathrm{nw}} & A_{\mathrm{ne}} \\ 0 & A_{\mathrm{se}} \end{pmatrix} \begin{pmatrix} \hat{X}_{\mathrm{n}} \\ \hat{X}_{\mathrm{s}} \end{pmatrix} = \begin{pmatrix} B_{\mathrm{n}} \\ B_{\mathrm{s}} \end{pmatrix}, \tag{2.6}$$

which can be solved in two steps:

$$\begin{array}{ll} 1. & \text{Solve} \quad A_{\mathrm{se}}\hat{X}_{\mathrm{s}} = B_{\mathrm{s}}, \\ 2. & \text{Solve} \quad A_{\mathrm{nw}}\hat{X}_{\mathrm{n}} = B_{\mathrm{n}} - A_{\mathrm{ne}}\hat{X}_{\mathrm{s}}. \end{array} \tag{2.7}$$

Since the blocks A_{se} and A_{nw} are strictly smaller than A, the above two systems can be solved recursively by the same strategy. This will result if four block Hessenberg systems, which can again be solved by tearing. At the bottom of this recursive process we arrive at systems involving only the diagonal blocks A_{ii}, which can be solved by standard techniques.

The following recursive program describes this algorithms. We assume that a tearing order has been prescribed, so that the matrices A_{ne}, B_{n}, etc. are defined at all levels of the recursion. As is usual, the solution X overwrites the right hand side B.

[2]The term "tearing" is due to Kron [6, 7]. However, his papers are couched obscurely in terms of electrical networks, and it is not at all clear that his method of tearing is the same as the one given here and elsewhere. For further references see [10, 11].

bhsolve(A, B)
  if (not at the bottom) then
    bhsolve$(A_{\mathrm{se}}, B_{\mathrm{s}})$
    $B_{\mathrm{n}} = B_{\mathrm{n}} - A_{\mathrm{ne}}B_{\mathrm{s}}$
    bhsolve$(A_{\mathrm{nw}}, B_{\mathrm{n}})$
    $B = B - PF^{\mathrm{T}}B$
  else
    $B = A^{-1}B$
  end if
end

The mathematical notation used here conceals some potential economies. For example, since F is formed from columns of the identity matrix, the calculation of $F^{\mathrm{T}}B$ amounts to extracting the corresponding rows of B. Again, the computation $A^{-1}B$ would in general be done by using an appropriate decomposition of the diagonal block A. Finally, any special structure of A, like sparseness, can be used to economize the computation of $A_{\mathrm{ne}}B_{\mathrm{s}}$. This last point is especially important, since the multiplication accounts for most of the work in the algorithm.

The function *bhsolve* is analogous to the solution phase of a method based on Gaussian elimination, since once the the patch matrices P are in place, it can be repeated for many different systems. What corresponds to Gaussian elimination itself is the initial calculation of the patch matrices. Now to calculate P at any level of the recursion we must do three things: solve the system $\hat{A}G = EU$, compute $\hat{R}$, and finally compute $P = G\hat{R}$. Since the last two steps are algorithmically trivial, we will combine them into a separate function, *patchend*, and say no more about them. The first step can in principal be accomplished by *bhsolve*; however, there are two special considerations.

1. If we use *bhsolve* at one level of the recursion, we must take care that patch matrices have been already been generated at the lower levels. This can be accomplished by two recursive calls to the patch generator, one to generate the patches for A_{nw} and the other to generate the patches for A_{se}.

2. The first step of the solution of $\hat{A}G = E$ involves the solution of the system $\hat{A}_{\mathrm{sw}}G_{\mathrm{s}} = E_{\mathrm{s}}$. However, only the first atomic block of E_{s} is nonzero. Hence, we will use a special solve routine *topsolve* to take computational advantage of this fact.

We are now in a position to write down the function that generates the patch matrices.

```
patchgen(A)
    if (not at the bottom) then
        patchgen(A_nw)
        patchgen(A_se)
        P = EU
        topsolve(A_se, P_s)
        P_n = -A_ne P_s
        bhsolve(A_nw, P_n)
        patchend(A, P)
    else
        Compute an appropriate decomposition of A
    end if
end
```

The decomposition computed at the bottom is for the use of *bhsolve* when it must compute $A^{-1}B$.

The function *topsolve* is like *bhsolve*, except that it recognizes that many of the subproblems solved by the latter have zero solutions.

```
topsolve(A, B)
    if (not at the bottom) then
        B_s = 0
        topsolve(A_nw, B_n)
        B = B - P F^T B
    else
        B = A^{-1} B
    end if
end
```

This completes the description of the algorithm.

3. Complexity

In this section we will consider the space and time complexity of the algorithm described in the last section. For simplicity we will restrict ourselves to the case where all the blocks are of order p and there are $n = 2^k$ blocks. At each level of the recursion the matrix is assumed to be split at the center, so that the work is balanced each level. Although it is possible to set up and analyze recurrences to get our expressions, it is easier to write down special cases and generalize — so-called engineering induction.

We will be begin with the storage required by the algorithm. Here we exclude the storage for the original matrix and concentrate on the extra storage required for the patch matrices. The following table lists the amount of storage required at each level of the recursion for $n = 2^8$. The general form of entries is sufficiently well illustrated by the first five levels, and we omit the lower levels.

level	order of block	number of matrices	storage per matrix	total storage
1	$2^8 p$	2^0	$2^8 p^2$	$2^8 p^2$
2	$2^7 p$	2^1	$2^7 p^2$	$2^8 p^2$
3	$2^6 p$	2^2	$2^6 p^2$	$2^8 p^2$
4	$2^5 p$	2^3	$2^5 p^2$	$2^8 p^2$
5	$2^4 p$	2^3	$2^4 p^2$	$2^8 p^2$
.	.	.	.	.
			Grand total $\cong$	$8 \cdot 2^8 p^2$

(3.1)

Recognizing that $8 = \log_2 2^8$, we see that

$$\text{Storage for patch matrices} \cong p^2 n \log_2 n. \tag{3.2}$$

We next consider the operation count for the function *bhsolve* when it is applied to a right hand side with one column. The key observation here is that most of the work is done in forming the product $A_{\mathrm{ne}} B_{\mathrm{s}}$. If the blocks of A are dense, then the number of operations will be the square of the order of the block. Hence we have the following table.

level	order of blocks	number of blocks	total operations
1	$\frac{1}{2} 2^8 p$	2^0	$\frac{1}{2} 2^{15} p^2$
2	$\frac{1}{2} 2^7 p$	2^1	$\frac{1}{2} 2^{14} p^2$
3	$\frac{1}{2} 2^6 p$	2^2	$\frac{1}{2} 2^{13} p^2$
4	$\frac{1}{2} 2^5 p$	2^3	$\frac{1}{2} 2^{12} p^2$
5	$\frac{1}{2} 2^4 p$	2^4	$\frac{1}{2} 2^{11} p^2$
.	.	.	.
		Grand total $\cong$	$\frac{1}{2} 2^{16} p^2$

(3.3)

Thus we see that

$$\text{Operations for } \mathit{bhsolve} \cong \frac{1}{2}n^2p^2, \tag{3.4}$$

which is the operation count for the back substitution step in Gaussian elimination.

Turning now to *patchgen*, we observe that the bulk of the work is concentrated in calls to *bhsolve* and the calculation of $A_{\mathrm{ne}}P_{\mathrm{s}}$ (the calls to *topsolve* account for a negligible amount of the work). The following table summarizes the situation at each level.

level	order of system	operations per system	number of systems	total operations
1	2^7p	$\frac{3}{2}2^{14}p^2$	2^0p	$\frac{3}{4}2^{15}p^3$
2	2^6p	$\frac{3}{2}2^{12}p^2$	2^1p	$\frac{3}{4}2^{14}p^3$
3	2^5p	$\frac{3}{2}2^{10}p^2$	2^2p	$\frac{3}{4}2^{13}p^3$
4	2^4p	$\frac{3}{2}2^{8}p^2$	2^3p	$\frac{3}{4}2^{12}p^3$
5	2^3p	$\frac{3}{2}2^{6}p^2$	2^4p	$\frac{3}{4}2^{11}p^3$
.	.	.	.	.
			Grand total $\cong$	$\frac{3}{4}2^{16}p^3$

(3.5)

Thus

$$\text{Operations for } \mathit{patchgen} \cong \frac{3}{4}n^2p^3. \tag{3.6}$$

This is the same as the count for Gaussian elimination that was given in the introduction.

These results show that for a well balanced tearing the descent algorithm is as good as Gaussian elimination. The price we pay is a modest increase in storage and unknown stability; however, in some cases there are additional benefits that may make this price well worth paying.

In the first place, our counts always have a factor of p^2, reflecting the cost of multiplying an atomic block by a vector. If the atomic blocks are sparse, then this cost is reduced, and the reduction is inherited in full measure by the descent algorithm. Moreover, unlike Gaussian elimination, the algorithm does not alter the blocks — i.e., there is no fill in — so that the storage requirement remains constant. Similar considerations apply when the atomic blocks can be generated on the fly.

Finally, when the matrix A is block Toeplitz, the algorithm has additional economies, to which we now turn.

4. Toeplitz Matrices

In this section we shall assume that A is block Toeplitz; that is, it has the form

$$A = \begin{pmatrix} A_1 & A_2 & A_3 & A_4 & \cdots \\ A_0 & A_1 & A_2 & A_3 & \cdots \\ 0 & A_0 & A_1 & A_2 & \cdots \\ 0 & 0 & A_0 & A_1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \end{pmatrix}$$

Our object is to determine if we can take effective advantage of the Toeplitz structure in our algorithm. As in the last section, we will assume that $n = 2^k$.

The first thing to note is that *at any level in the recursion, the patch matrices are all the same.* Consequently we can expect some gains in storage and generation time, since we only need to form and store on patch matrix at each level.

The storage can be analyzed by placing ones in the column labeled "number of matrices" in (3.1). Repeating the argument there we find that the storage for the patch matrices is approximately $2np^2$, a modest reduction in storage over (3.2)

The operation count for *bhsolve* remains unchanged. The operation count for *patchgen* can be computed by placing replacing the powers of two in the column labeled "number of system" in (3.5) by ones. This causes the sum of the elements of the last column to become

$$\frac{3}{8}p^3(4^8 + 4^7 + 4^6 + \cdots).$$

From the approximation

$$1 + 4 + 16 + \cdots 4^k \cong \frac{4}{3}4^k,$$

we see that the operation count for *patchgen* becomes $\frac{1}{2}n^2p^3$ — an insignificant improvement over (3.6).

The reason for the disappointing results is that the operation counts are being driven by the matrix multiplication $A_{\mathrm{ne}}x_{\mathrm{s}}$ in (2.7). Since the amount of work is proportional to the square of the order of A_{ne}, a disproportionate amount of work is being done near top of the recursion. What goes on below does not make much difference.

However, we have not taken into account the fact that A_{ne} is itself block Toeplitz; e.g., for $n = 8$ we must compute the produce

$$\begin{pmatrix} A_5 & A_6 & A_7 & A_8 \\ A_4 & A_5 & A_6 & A_7 \\ A_3 & A_4 & A_5 & A_6 \\ A_2 & A_3 & A_4 & A_5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}. \tag{4.1}$$

Now there are fast algorithms for multiplying Toeplitz matrices by a vector. Unfortunately, they do not generalize to block-Toeplitz matrices. However, we will now show how the product (4.1) can be converted into a block product in which the blocks are Toeplitz.

Let the number of atomic block in the matrix A_{ne} be m, and consider the permutation

$$\begin{array}{rrcr} 1 \leftarrow 1 & 2 \leftarrow p+1 & \cdots & m-1 \leftarrow (m-1)p+1 \\ m \leftarrow 2 & m+1 \leftarrow p+2 & \cdots & 2m-1 \leftarrow (2m-1)p+2 \\ 2m \leftarrow 3 & 2m+1 \leftarrow p+3 & \cdots & 3m-1 \leftarrow (3m-1)p+3 \\ \vdots & \vdots & & \vdots \\ (p-1)m \leftarrow p & (p-q)m+1 \leftarrow 2p & \cdots & mp \leftarrow mp \end{array}$$

This permutation reorders the vector in (4.1) so that the first components of the vectors $x_1, \ldots, x_4$ appear first, then the second components of $x_1, \ldots, x_4$, and then the third, and so on. If P denotes the matrix of this permutation, we can rewrite the product $y = A_{\mathrm{ne}}x$ in the form $\bar{y} \equiv P^{\mathrm{T}}y = P^{\mathrm{T}}A_{\mathrm{nw}}PP^{\mathrm{T}}x \equiv C\bar{x}$, or

$$\begin{pmatrix} \bar{y}_1 \\ \bar{y}_2 \\ \vdots \\ \bar{y}_p \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1p} \\ C_{21} & C_{22} & \cdots & C_{2p} \\ \vdots & \vdots & & \vdots \\ C_{p1} & C_{p2} & \cdots & C_{pp} \end{pmatrix} \begin{pmatrix} \bar{x}_1 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_p \end{pmatrix}.$$

Moreover, it is easy to see that each block C_{ij} is an $m \times m$ Toeplitz matrix.

Thus the problem of computing the product $A_{\mathrm{ne}}x$ has been reduced to computing and summing the products $C_{ij}x_j$. Now the product of a Toelplitz matrix of order m and a vector can be computed in $O(m \log_2 m)$ [1, 2]. For definiteness let us say that it requires $Km \log_2 m$ operations. Hence we can compute the product $A_{\mathrm{ne}}x_{\mathrm{s}}$ with $Kp^2 m \log_2 m$ operations.

We now have only to insert this bound in the table (3.3) to update the operation count for *bhsolve*. The result is

level	order of blocks	number of blocks	total operations
1	$\frac{1}{2}2^8 p$	2^0	$\frac{1}{2}2^8 \cdot 7 \cdot Kp^2$
2	$\frac{1}{2}2^7 p$	2^1	$\frac{1}{2}2^8 \cdot 6 \cdot Kp^2$
3	$\frac{1}{2}2^6 p$	2^2	$\frac{1}{2}2^8 \cdot 5 \cdot Kp^2$
4	$\frac{1}{2}2^5 p$	2^3	$\frac{1}{2}2^8 \cdot 4 \cdot Kp^2$
5	$\frac{1}{2}2^4 p$	2^4	$\frac{1}{2}2^8 \cdot 3 \cdot Kp^2$
.	.	.	.
		Grand total $\cong$	$\frac{1}{4}2^8 \cdot 8^2 \cdot Kp^2$

The factor 8^2 comes from the fact that

$$(k-1)+(k-2)+(k-3)+\cdots+1 \cong \frac{k^2}{2}.$$

Thus revised count is

$$\text{Operations for } \textit{bhsolve} \cong \frac{K}{4} p^2 n \log_2^2 n, \tag{4.2}$$

Comparing (3.4) with (4.2), we see that the factor of n^2 has been reduced to $n \log_2^2 n$ — a substantial savings.

Finally, regarding *patchgen*, we must solve one system with p columns at each level. Hence from (4.2), we find that the operations to generate the patch matrices is

$$\begin{aligned}\frac{K}{4} p^3 \sum_{i=1}^{k} i^2 2^i &\leq \frac{K}{4} p^3 k^2 2^k \left[1 + \frac{(k-1)^2}{k^2}\left(\frac{1}{2}+\frac{1}{4}+\frac{1}{4}+\cdots\right)\right] \\ &\cong \frac{K}{4} p^3 k^2 2^k \cdot 2.\end{aligned}$$

Thus the revised count is

$$\text{Operations for } \textit{patchgen} \lesssim \frac{K}{4} p^3 n \log_2^2 n, \tag{4.3}$$

again a substantial improvement over (3.6).

It remains to add that the counts (4.2) and (4.3) remain valid when $p = 1$. In this case our algorithm becomes an $O(n \log_2^2 n)$ algorithm for solving Toeplitz Hessenberg systems.

5. Markov Chains

We now return to the problem that motivated our investigations — the computation of the left eigenvector corresponding to 1 of the Markov chain (1.1). This is equivalent to finding a left null vector of the matrix $I - M$. Since in general, the chain will be infinite, we assume that it has been truncated to form a finite system. We will also assume that the resulting chain is irreducible, so that the eigenvector is unique, up to a scaling factor.

The system we have to solve is not a block Toeplitz system; moreover, it is homogeneous. However, we can turn it into an inhomogeneous block Toeplitz system by the following device. Partition

$$I - M = \begin{pmatrix} I - B_0 & V^{\mathrm{T}} \\ U & T \end{pmatrix}.$$

Then the first block in the solution of the homogeneous equation will be a null vector of the Schur complement of T in $I - M$; i.e., of

$$S = (I - B_0) - V^{\mathrm{T}} T^{-1} U.$$

Now T is block Toeplitz and block upper Hessenberg. Consequently, we can apply our algorithm to solve the system

$$TW = U, \tag{5.1}$$

after which we can compute $S = (I - B_0) - V^{\mathrm{T}} W$. Note that since only the first block of U is nonzero, we can use *topsolve* to solve (5.1), with some savings in operations.

Once a null vector y_1^{T} of S has been computed, we may compute the remaining components y_2^{T} of y^{T} by solving the system

$$y_2^{\mathrm{T}} T = -y_1^{\mathrm{T}} V.$$

Here we are solving a transpose system, and we must use analogues of the algorithms proposed in the preceding sections.

An attractive feature of this method is that if having computed a solution we decide that the infinite system was prematurely truncated, we can double the size of T and still use previously computed quantities to update the system.

References

[1] G. S. Ammar and W. B. Gragg. The generalized Schur algorithm for the superfast solution of Toeplitz systems. In J. Gilewicz, M. Pindor, and W. Siemaszko, editors, *Rational Approximation and Its Applications in Mathematics and Physics. Proceedings, Lańut, 1985*, pages 315–330, New York, 1985. Springer.

[2] G. S. Ammar and W. B. Gragg. The implementation and use of the generalized Schur algorithm. In C. I. Byrnes and A. Linquist, editors, *Computational and Combinatorial Methods in Systems Theory*, pages 265–279. North Holland, Amsterdam, 1986.

[3] J. J. M. Cuppen. A divide and conquer method for the symmetric eigenproblem. *Numerische Mathematik*, 36:177–195, 1981.

[4] J. J. Dongarra and D. C. Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM Journal on Scientific and Statistical Computing*, 8:s139–s154, 1987.

[5] A. S. Householder. *The Theory of Matrices in Numerical Analysis*. Dover Publishing, New York, 1964. Originally published by Ginn Blaisdell.

[6] G. Kron. *Tensor Analysis of Networks*. John Wiley, New York, 1939.

[7] G. Kron. A set of principles to interconnect solutions of physical systems. *Journal of Applied Physics*, 24:965–980, 1953.

[8] M. Neuts. *Matrix-Geometric Solutions in Stochastic Models*. Johns Hopkins University Press, Baltimore, 1981.

[9] D. C. Sorensen and P. T. P. Tang. On the orthogonality of eigenvectors computed by divide and conquer techniques. *SIAM Journal on Numerical Analysis*, 28:1752–1775, 1990.

[10] D. V. Steward. On an approach to techniques for the analysis of the structure of large systems of equations. *SIAM Review*, 4:321–342, 1962.

[11] D. V. Steward. Partitioning and tearing systems of equations. *SIAM Journal on Numerical Analysis*, 2:345–365, 1965.