University of Maryland College Park

Institute for Advanced Computer Studies TR–94–70
Department of Computer Science TR–3295

Implementing an Algorithm for Solving Block Hessenberg Systems*

G. W. Stewart†

December, 1993

ABSTRACT

This paper describes the implementation of a recursive descent method for solving block Hessenberg systems. Although the algorithm is conceptually simple, its implementation in C (a natural choice of language given the recursive nature of the algorithm and its data) is nontrivial. Particularly important is the balance between ease of use, computational efficiency, and flexibility.

*This report and the programs it documents are 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.

Implementing an Algorithm for Solving Block Hessenberg Systems

G. W. Stewart

ABSTRACT

This paper describes the implementation of a recursive descent method for solving block Hessenberg systems. Although the algorithm is conceptually simple, its implementation in C (a natural choice of language given the recursive nature of the algorithm and its data) is nontrivial. Particularly important is the balance between ease of use, computational efficiency, and flexibility.

1. Introduction

Block Hessenberg arise in a number of applications, including queueing theory [2, 3] and repairable systems [1]. This paper describes a collection of programs (BHESS) for solving a system of order N having the form

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1,n-1} & A_{1n} \\ A_{21} & A_{22} & A_{23} & \cdots & A_{2,n-1} & A_{2n} \\ 0 & A_{32} & A_{33} & \cdots & A_{3,n-1} & A_{3n} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & A_{n-1,n-1} & A_{n-1,n} \\ 0 & 0 & 0 & \cdots & A_{n,n-1} & A_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix}, \tag{1.1}$$

where the diagonal blocks A_{ii} are of order m_i. Programs are also provided to solve the system $x^{\mathrm{T}}A = b^{\mathrm{T}}$, so that the package can also handle block lower Hessenberg systems.

The system is solved by recursively dividing it into smaller block Hessenberg systems. The mathematical underpinings of the algorithm as well as its complexity have been treated elsewhere by the author [4]. If the matrix A is dense, then the method costs the same as Gaussian elimination. However, if A is sparse, the method can result in considerable savings in both time and memory. A rounding error analysis [5] shows that the method is stable for diagonally dominant matrices, which covers the applications mentioned above.

The purpose of this paper is to describe an implementation in C of the algorithm. The details of the implementation have been driven by a number of considerations, not all of them consonant. Here are the main ones.

1. Ease of use. The basic algorithm, though conceptually simple, is not one that lends itself to a black-box implementation. The user must know something of how the algorithm works and the data structures it uses. To ease the burden, BHESS provides an initialization routine, some useful utility programs, and sample programs.

2. Storage management. The matrix A can become very large (in some application it is a section of an infinite matrix). To conserve storage, options have been provided to reuse storage that would otherwise go unused. The structure containing the matrix can be reinitialized and deallocated.

3. Efficiency. The most significant computation performed in the course of the algorithm is the multiplication of certain submatrices of A by a vector. The implementation segregates this operation in a function, which the user may code to take advantage of any special properties of the matrix.

4. Extensibility. In some queueing theory applications, larger part of A has a block Toeplitz structure. Although BHESS does not deal directly with such matrices, the data structures and programs are compatible with future extensions to this case.

5. Matrix issues. The treatment of matrices in C is an open problem. I have used the implementation of this package as an opportunity to come to grips with some of the issues. Although they are not addressed explicitly in this paper, the expert will be able to see the nature of my solutions from the programs themselves. A detailed exposition will be left for a later work.

The paper is organized as follows. In the next section we give a brief description of the algorithm. Section 3 gives an overview of the package. The next four sections are devoted to describing the specific functions that compose BHESS in greater detail. In Section 8 the package is used in an actual application, a Markov model of a reservoir. Three appendices include a tutorial example, brief descriptions of the support routines used by the package, and an annotated index of the functions that compose BHESS.

2. The Algorithm

As mentioned above, the algorithm solves a block Hessenberg system by recursive descent; that is, it reduces the problem to that of solving two smaller block Hessenberg problems.[1] The basic idea is to tear out one of the subdiagonal blocks, solve the resulting block triangular system, and patch up the solution so that it solves the original system.

Specifically, let t satisfy $1 \leq t < n$ (t is called the *tear index*). Partition A in the

[1] It would be a mistake to say the algorithm divides and conquers, since for dense matrices it does no better than Gaussian elimination. Rather say it is a divide-and-survive algorithm.

form

$$
A = \left(\begin{array}{cccccc|cccccc}
A_{11} & A_{12} & \cdots & A_{1,t-1} & A_{1t} & A_{1,t+1} & A_{1,t+2} & \cdots & A_{1,n-1} & A_{1n} \\
A_{21} & A_{22} & \cdots & A_{2,t-1} & A_{2t} & A_{2,t+1} & A_{2,t+2} & \cdots & A_{2,n-1} & A_{2n} \\
0 & A_{32} & \cdots & A_{3,t-1} & A_{3t} & A_{3,t+1} & A_{3,t+2} & \cdots & A_{3,n-1} & A_{3n} \\
\vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\
0 & 0 & \cdots & A_{t-1,t-1} & A_{t-1,t} & A_{t-1,t+1} & A_{t-1,t+2} & \cdots & A_{t-1,n-1} & A_{t-1,n} \\
0 & 0 & \cdots & A_{t,t-1} & A_{t,t} & A_{t,t+1} & A_{t,t+2} & \cdots & A_{t,n-1} & A_{tn} \\
\hline
0 & 0 & \cdots & 0 & A_{t+1,t} & A_{t+1,t+1} & A_{t+1,t+2} & \cdots & A_{t+1,n-1} & A_{t+1,n} \\
0 & 0 & \cdots & 0 & 0 & A_{t+2,t+1} & A_{t+2,t+2} & \cdots & A_{t+2,n-1} & A_{t+2,n} \\
0 & 0 & \cdots & 0 & 0 & 0 & A_{t+3,t+2} & \cdots & A_{t+3,n-1} & A_{t+3,n} \\
\vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\
0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & A_{n-1,n-1} & A_{n-1,n} \\
0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & A_{n,n-1} & A_{nn}
\end{array}\right).
$$

Equivalently,

$$
A = \begin{pmatrix} A_{\mathrm{nw}} & A_{\mathrm{ne}} \\ A_{\mathrm{sw}} & A_{\mathrm{se}} \end{pmatrix}, \tag{2.1}
$$

where the subscripts refer to points of the compass. Then the system (1.1) can be written in the form

$$
\begin{pmatrix} A_{\mathrm{nw}} & A_{\mathrm{ne}} \\ A_{\mathrm{sw}} & A_{\mathrm{se}} \end{pmatrix} \begin{pmatrix} x_{\mathrm{n}} \\ x_{\mathrm{s}} \end{pmatrix} = \begin{pmatrix} b_{\mathrm{n}} \\ b_{\mathrm{s}} \end{pmatrix}, \tag{2.2}
$$

where we have partitioned x and b conformally with (2.1).

Now consider the system

$$
\begin{pmatrix} A_{\mathrm{nw}} & A_{\mathrm{ne}} \\ 0 & A_{\mathrm{se}} \end{pmatrix} \begin{pmatrix} \hat{x}_{\mathrm{n}} \\ \hat{x}_{\mathrm{s}} \end{pmatrix} = \begin{pmatrix} b_{\mathrm{n}} \\ b_{\mathrm{s}} \end{pmatrix}, \tag{2.3}
$$

in which the matrix A_{sw} has been torn from the matrix A. Since A_{nw} and A_{sw} are block Hessenberg matrices, the torn system can be solved recursively by the following algorithm.

$$
\begin{array}{ll} 1. & \text{Solve } A_{\mathrm{se}}\hat{x}_{\mathrm{s}} = b_{\mathrm{s}} \\ 2. & \text{Solve } A_{\mathrm{nw}}\hat{x}_{\mathrm{n}} = b_{\mathrm{n}} - A_{\mathrm{ne}}\hat{x}_{\mathrm{s}} \end{array} \tag{2.4}
$$

To recover a solution of the original system (1.1) from the solution of (2.3), let r_{sw} be the rank of the matrix A_{sw}. Then A_{sw} has a full-rank decomposition

$$
A_{\mathrm{sw}} = QR \tag{2.5}
$$

where Q is $m_{t+1} \times r_{\mathrm{sw}}$ and R is $r_{\mathrm{sw}} \times m_t$. Partition $\hat{x}$ conformally with (1.1):

$$
\hat{x}^{\mathrm{T}} = (\hat{x}_1^{\mathrm{T}}, \ldots, \hat{x}_t^{\mathrm{T}}, \hat{x}_{t+1}^{\mathrm{T}}, \ldots, \hat{x}_n^{\mathrm{T}}).
$$

Then it can be shown that there is an $N \times r_{\mathrm{sw}}$ matrix V and an $r_{\mathrm{sw}} \times r_{\mathrm{sw}}$ matrix S such that

$$x = \hat{x} - VS^{-1}R\hat{x}_t.$$

The matrix V is called the *right patch matrix* for the torn system (2.3). (The word "right" comes from the fact that the patch matrix is used in solving $Ax = b$, in which x appears to the right of A.) The matrix S is the *central patch matrix.* There is an analogous left patch matrix U for the transposed system $x^{\mathrm{T}}A = b^{\mathrm{T}}$. (The central patch matrix is the same for both systems.) We will describe these patch matrices more fully in Section 6, where their generation is treated.

3. Organization

The above description of the algorithm suggests how BHESS is organized. The matrix A is defined by a recursive structure, called a `bhtree` (block Hessenberg tree). This structure contains, among other items, pointers to the matrices A_{ne}, A_{sw}, as well as pointers to the `bhtree`s A_{nw} and A_{sw}. At the bottom, there is only a pointer to the diagonal block A_{ii}.

Since a `bhtree` is a complicated structure, BHESS provides a function, `make_bhtree`, to initialize it. However, the user must feed matrices to this routine, which is done by a user supplied function of type `USERGET`.

Once the `bhtree` has been initialized, but before a system can be solved, the patch matrices must be generated. This is done by the function `patchgen`.

Finally, BHESS provides the functions `bhrsolve` and `bhlsolve` to solve $Ax = b$ and $x^{\mathrm{T}}A = b^{\mathrm{T}}$. (The "`r`" in `bhrsolve` refers to the fact that x appears to the right of A in the equation $Ax = b$. Analogously, the "`l`" in `bhlsolve` stands for left.) The solutions overwrite the right-hand sides.

It turns out that the bulk of the work in the algorithm consist in forming products of the form $A_{\mathrm{ne}}\hat{x}_{\mathrm{s}}$ [cf. (2.4.2)]. This is the point where the user can speed up the algorithm by taking advantage of any structure in the matrix A_{nw} (e.g., sparseness). For this reason, BHESS does not attempt to perform the multiplication itself. Instead, the user furnishes a function of type `ANECOMP`, to perform the multiplication. Both `patchgen` and the solve routines use this function.

The following table gives a summary of the structure of BHESS and what the user must provide.

```
struct bhtree{
   int n;                    /* The number of blocks in the matrix A. */
   int *dbx;                 /* dbx[i] is the starting index of the
                                i-th diagonal block.  By convention
                                dbx[n+1] is one plus the order of A. */
   int first;                /* The index of the first diagonal block. */
   int last;                 /* The index of the last diagonal block. */
   int tear;                 /* A[tear+1][tear] = A_sw is the tear block. */
   double eps;               /* Criterion for determining rank of A_sw. */
   void *a_ne;               /* Pointer to A_ne. */
   struct dmat   *a_sw;      /* Pointer to A_sw  */
   struct dhqrdc *aswfr;     /* Full rank decomposition of A_sw. */
   struct bhtree *a_nw;      /* Pointer to the structure containing A_nw. */
   struct bhtree *a_se;      /* Pointer to the structure containing A_se. */
   struct dmat   *a_ii;      /* Pointer to the diagonal block. */
   struct dhqrdc *aiiqr;     /* QR decomposition of the diagonal block. */
   struct dmat *v;           /* The right patch matrix. */
   struct dmat *u;           /* The left patch matrix. */
   struct dhqrdc *s;         /* The central patch matrix. */
   double *aux;              /* Auxiliary storage */
};
```

Figure 4.1: The Structure `bhtree`

	BHESS	USER
1.	Initialization	Initialization
	`make_bhtree`	`USERGET`
2.	Patch generation	Multiplication by A_{ne}
	`patchgen`	`ANECOMP`
3.	Solution	Multiplication by A_{ne}
	`bhrsolve`	`ANECOMP`
	`bhlsolve`	

The next four sections are devoted to explicating this table.

4. Representation

As we mentioned, the matrix of the system to be solved is contained in a recursive structure called a `bhtree`. The definition of the structure is given in Figure 4.1.

The first two fields in this structure specify the global structure of the matrix. The field `n` contains the number of blocks. The ith element of the array `dbx` (for diagonal block index) is the index of the $(1,1)$-element of A_{ii} in A. By convention, `dbx[n+1]` is the order of A plus one.[2] Thus,

$$\text{the order of } A_{ii} \text{ is } \texttt{dbx[i+1]-dbx[i]}. \qquad (4.1)$$

Since a `bhtree` is a recursive structure, we need local information to tell us where we are in the tree. The fields `first` and `last` contain the indices of the first and last diagonal blocks in the current subtree. The field `tear` contains the tear index. BHESS sets

```
tear = (first+last)/2;
```

however, this assignment can be overridden as described below. When `first` and `last` are equal, we are at a leaf of the tree, and the structure contains only the diagonal block $A_{\texttt{first},\texttt{first}}$.

For example, suppose `n` and `dbx` are initialized as follows:

```
int n=8, dbx[] = {0,1,2,6,9,10,11,13,15,16};
```

Then the matrix A and its recursive partition are as follows.

$$\begin{array}{c|cccc|cccc|cccccc}
X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
\hline
X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
\hline
 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
 & & & & & X & X & X & X & X & X & X & X & X & X \\
\hline
 & & & & & & & & X & X & X & X & X & X & X \\
 & & & & & & & & & X & X & X & X & X & X \\
 & & & & & & & & & X & X & X & X & X & X \\
 & & & & & & & & & & X & X & X & X & X \\
 & & & & & & & & & & X & X & X & X & X \\
 & & & & & & & & & & & & X & X & X
\end{array} \qquad (4.2)$$

[2] One difficulty with matrix computations in C is that the initial index of a C array is zero, whereas the initial index of a vector is conventionally one. BHESS follows the usual vector-matrix conventions, sometimes sacrificing a little storage to do so.

It is instructive to follow the partitioning down to the diagonal blocks by computing successive tear indices. It is also instructive (and useful) to show that

$$A_{\mathrm{ne}} \text{ has } \begin{cases} \texttt{a->dbx[a->tear+1] - a->dbx[a->first]} \text{ rows and} \\ \texttt{a->dbx[a->last+1] - a->dbx[a->tear+1]} \text{ columns,} \end{cases} \tag{4.3}$$

while

$$A_{\mathrm{sw}} \text{ has } \begin{cases} \texttt{a->dbx[a->tear+2] - a->dbx[a->tear+1]} \text{ rows and} \\ \texttt{a->dbx[a->tear+1] - a->dbx[a->tear]} \text{ columns.} \end{cases} \tag{4.4}$$

Note that by an abuse of notation, A_{sw} here refers to the nonzero part of the southwest corner of the partition; i.e., $A_{\mathtt{tear},\mathtt{tear}+1}$.

5. Initialization and Deallocation

BHESS provides a function to initialize a `bhtree`. Its calling sequence is given in Figure 5.1. The first four arguments to the function specify the structure of the tree. The argument `getbh` is a pointer to a function used to help in the initialization (more on this later).

The last three parameters have to do with storage management. They may all be zero and the program will work. What follows is a brief explanation of their functions.

BHESS needs an array of auxiliary storage whose size is the order of the largest diagonal block. If `aux` is `NULL` (i.e., zero) BHESS will allocate this storage. Otherwise it will use the storage pointed to by `aux`.

BHESS must compute the full rank factorization (2.5) to patch together solutions. If `gobble` is true (i.e., nonzero), BHESS uses the memory allocated for A_{sw} to hold one of the factors Q or R. In addition, the memory for A_{ii} is preempted for other use.

In some parameter studies, block Hessenberg matrices of the same structure are solved repeatedly. In such cases it does not make sense to reallocate the storage for the submatrices of the tree afresh. Instead, one can pass a pointer `a` to the old `bhtree`, and `make_bhtree` will refill it.

The submatrices that compose the block Hessenberg matrix must be provided by the user. To get them, `make_bhtree` invokes a function whose calling sequence is given in the type definition in Figure 5.2. The initializations performed by the function my be divided into the obligatory and the optional. The function *must* be prepared to initialize the following fields of the `bh_tree` pointed to by `a`:

1. `a_ne`
2. `a_sw`
3. `a_ii`

```
struct bhtree *make_bhtree(int n, int *dbx, int first, int last,
            double *aux, USERGET *getbh, int gobble, struct bhtree *a)
/*
   On invocation
      n      is the number of diagonal blocks in the matrix.
      dbx    is an array of dimension at least n+2.  dbx[i]
             is the index of the (1,1)-element of the i-th
             diagonal block.  dbx[n+1] is one plus the order
             of the matrix.
      first  = 1 on the first invocation.  Subsequently it is the
             number of the first block in the subtree being initialized.
      last   = n on the first invocation.  Subsequently it is the
             number of the last block in the subtree being initialized.
      getbh  is a pointer to a user supplied program to help
             initialize the tree.  See above for the calling sequence.
      aux    is a pointer to working storage.  If it is NULL,
             storage is allocated.
      gobble If gobble is true the storage for A_sw and A_ii is
             used to hold their QR decomposition.
      a      If a is NULL, the tree is built up ab initio.
             Otherwise the bhtree is placed in the bhtree
             pointed to by a.  The old and the new trees
             *must* have exactly the same structure.

   On return, make_bhtree returns a pointer to an iniitialized
   bhtree.  A null pointer indicates failure.
*/
```

Figure 5.1: The function `make_bhtree`

4. `eps`

Let's consider each in turn.

1. `a_ne`. If `first` is not equal to `last`, the function must initialize `a_ne` which points to a structure containing A_{ne}. Since the user is responsible for manipulating A_{ne}, the pointer has been given type `void`, so that the user can use any structure deemed suitable.

2. `a_sw`. If `first` is not equal to `last`, the function must initialized `a_sw`, which points to a structure containing A_{sw}. The structure is a `dmat` (double precision matrix), which is used by BHESS to represent matrices in C. The sequence

```
typedef void (*USERGET)(struct bhtree *a);

   USERGET initializes the pointers a_ne, a_sw, a_ii, and the rank
   criterion eps.  It may also initialize tear, aswfr, and aiiqr.
```

Figure 5.2: The User Initializaton Function

```
struct dmat *asw;
asw = make_dmat(nr, nc);
```

produces a pointer to a `dmat` with `nr` rows and `nc` columns. The expression

```
asw->val[i][j]
```

references the (`i`, `j`)-element of the `dmat`. The `dmat` and all the storage allocated for it can be made to go away by invoking

```
kill_dmat(asw);
```

If an old `bhtree` is being reinitialized, the pointer `a_sw` will be nonnull and point to a `dmat` of the proper dimension. This is a general rule.

> If a pointer in a `bhtree` is nonnull, it points to a structure of the appropriate type and size which may be used in initialization.

3. `a_ii`. If `first` is equal to `last`, the function must initialize the pointer `a_ii` to $A_{\mathtt{first},\mathtt{first}}$, which is also a `dmat`. The above comments on `make_dmat` and reinitialization apply here.

4. `eps`. Recall that the patch matrix V has r_{sw} columns, where r_{sw} is the rank of A_{sw}. The field `eps` is used to determine the rank. It should be thought of as the size of an element that can be set to zero in A_{sw} without undue harm. There is a trade-off here. A larger value of `eps` may result in a lower rank and a savings in time. But the results will be less accurate. Rank reduction is most useful when, except for rounding error, A_{sw} is deficient in rank, in which case one can set `eps` equal to, say, one hundred times the product of the rounding unit and the size of a typical element in A_{sw}. A value of zero turns off rank checking.

The following may be initialized at the user's option.

1. `tear`
2. `aswfr`
3. `aiiqr`

```
void kill_bhtree(struct bhtree *a, ANECOMP anecomp)
/*
   kill_bhtree frees the storage allocated by make_bhtree.

   On invocation
      a       is a pointer to a bhtree created by make_bhtree.
      anecomp is a pointer to a routine to manipulate A_ne.

   On return the storage for the bhtree is deallocated.
*/
```

Figure 5.3: The Function `kill_bhtree`

1. `tear`. The default tear index is `(first+last)/2`. If the sizes of all the diagonal blocks are approximately the same, this strategy will produce a computationally balanced tree. However, if the blocks vary widely in size, a different strategy may be in order. This is the place to implement it.

2. `aswfr`. This field points to the structure that contains the full rank factorization (2.5) of A_{sw}. Ordinarily BHESS will compute this factorization automatically. However, in some simple cases it is possible to read off the factorization directly from the matrix. For example, a full rank factorization of the matrix

$$A_{\mathrm{sw}} = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

is

$$QR \equiv \begin{pmatrix} 1 \\ 0 \end{pmatrix} (1\ 2\ 0).$$

In such cases you can save time by providing the factor directly to BHESS.

The structure containing the full rank factorization is named `dhqrdc`. It can be created by the statement [cf. (4.4)]

```
a->aswfr = make_dhqrdc(a->dbx[a->tear+2] - a->dbx[a->tear+1],
                       a->dbx[a->tear+1] - a->dbx[a->tear], NULL);
```

The field `a->aswfr->r` points to a `dmat` that must initialized to R. Similar comments apply to the field `a->aswfr->q` which corresponds to Q. In addition, the field `a->aswfr->effrnk` must be initialized to the rank of A_{sw} (one in the above example),

```
void patchgen(struct bhtree *a, int job, ANECOMP *anem)
/*
   patchgen generates right patch matrices for bhlsolve and
   left patch matrices for bhrsolve.

   On invocation
      a        is a pointer to the bhtree containing the matrix A.
      job      If job<2, generate only the right patch matrices.
               If job=2, generate only the left patch matrices.
               If job>2, generate both the left and right patch matrices.
      anecomp  is a pointer to a routine to manipulate A_ne.

   On return the the structure contains the patch matrices.
*/
```

Figure 5.4: The function `patchgen`

and the field `a->aswfr->status` must be set to one to tell `makebhtree` that a factorization is present.[3] As always if `a->aswfr` is nonnull, it already points to a `dhqrdc` waiting to be initialized.

3. `aiiqr`. Ordinarily BHESS uses the program `dhqr_dc` (see §B.1) to compute QR decompositions of the diagonal blocks, which is used to solve systems involving them. Here the user may initialize `a->aiiqr` to the output of `dhqr_dc`. The flag `a->aiiqr->status` must be set to one. The only likely case where this option will save time is when all the matrices A_{ii} are the same.

If the number of blocks is large or the dimensions of the individual blocks are large, a `bhtree` will consume a great deal of storage. BHESS provides a function, `kill_bhtree`, to free that storage when the job is done. Its calling sequence is given in Figure 5.3.

6. Patch generation

Once the matrix has been initialized, patch matrices must be generated at all levels of the tree. This is the most expensive part of the algorithm, and it can be likened to the decomposition phase of a solution of linear systems by Gaussian elimination. If fact, if the matrix A is dense and the subdiagonal blocks are of full rank, the operation count

[3] In some cases (see §8) the matrices A_{sw} are all the same, so that storage can be saved by causing `a->aswfr` to always point to the same structure. However, each invocation of `USERGET` must set the status flag to one, since `makebhtree` sets it to zero.

```
typedef void (*ANECOMP)(
        struct bhtree *a,     // The bhtree
        double *b,            // A vector
        int bstr,             // A stride for b
        double *c,            // Another vector
        int cstr,             // A stride for c
        int job               // A parameter
);

If job == 1, the function calculates  b = b - A_ne*c.

If job == 2, the function calculates  b' = b' - c'*A_ne.

If job == 3, the function calculates   b = b + A_ne*c.

If job == 4, the function calculates  b' = b' + c'*A_ne.

If job == -1, the function cleans up a->ane.
```

Figure 6.1: The Specification ANECOMP

is the same as for Gaussian elimination.

To describe the patch matrices, we must introduce some additional notation. As in Section 2, let $A_{t+1,t} = QR$ be a full rank factorization, and let

$$E = \begin{array}{c} \\ \\ \\ t+1 \\ \\ \\ \end{array} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ Q \\ \vdots \\ 0 \end{pmatrix} \qquad \text{and} \qquad F = \begin{array}{c} \begin{array}{cccccc} & & t & & & \end{array} \\ \begin{pmatrix} 0 & \cdots & R & 0 & \cdots & 0 \end{pmatrix}. \end{array}$$

Here the matrices Q and R occupy positions corresponding to blocks $t+1$ and t respectively. Then the left, central, and right patch matrices are

1. $U = F\hat{A}^{-1}$,
2. $S = I + UA^{-1}V$,
3. $V = \hat{A}^{-1}E$.

The key observation that drives the algorithm is that the definition of the patch matrices involves only the matrix $\hat{A}$, not A itself. Consequently, if patch matrices for A_{nw} and A_{se} have already been generated, we can generate the patch matrices for A by solving the systems

$$U\hat{A} = F \qquad \text{and} \qquad \hat{A}V = E.$$

Thus the entire ensemble of patch matrices can be generated by starting at the bottom of the `bhtree` and working upwards.

There two ways to save work in generating the patch matrices. First, in solving, say, the system

$$\hat{A}_{\mathrm{se}}U_{\mathrm{s}} = E_{\mathrm{s}}$$

[the first step in the algorithm (2.4)] we can take advantage of the fact that all but the first block of E_{s} is zero. This is done by the function `topsolve` (the corresponding program for the left patch matrix is `rightsolve`). Second, the central patch matrix can be computed in the form $I + FV$, and since most of the blocks of F are zero, the multiplication is almost trivial.

If we wish to solve systems of the form $Ax = b$, we must generate the right patch matrices; for systems of the form $x^{\mathrm{T}}A = b^{\mathrm{T}}$, we need the left patch matrices. The function `patchgen` does the job. Its calling sequence is given in Figure 5.4.

The function `patchgen` requires that the user furnish a function to manipulate A_{ne}. On request, it performs the following computations. First, it computes $b_{\mathrm{n}} - A_{\mathrm{ne}}b_{\mathrm{s}}$ as required in (2.4). Second, it computes $b_{\mathrm{e}}^{\mathrm{T}} - b_{\mathrm{w}}^{\mathrm{T}}A_{\mathrm{nw}}$, where $b^{\mathrm{T}} = (b_{\mathrm{w}}^{\mathrm{T}} \;\; b_{\mathrm{e}}^{\mathrm{T}})$, which is required to solve $x^{\mathrm{T}}A = b^{\mathrm{T}}$. Third and fourth, the program may compute $b_{\mathrm{n}} + A_{\mathrm{ne}}b_{\mathrm{s}}$ or $b_{\mathrm{e}}^{\mathrm{T}} + b_{\mathrm{w}}^{\mathrm{T}}A_{\mathrm{nw}}$, which are useful in generating test cases. Finally, the function may be invoked to free storage allocated to A_{ne}.

The calling sequence for the program that does all this is given in Figure 6.1. The parameter `job`, tells the function what to do. The pointer `a->a_ne` points to the current A_{ne}. The pointer itself is of type `void`, and presumably the user will recast it to a pointer of an appropriate type; e.g., a pointer to `dmat`. The argument `bstr` is the distance between components of b. Thus the ith component of b is `*(b+bstr*(i-1))`. Similarly the ith component of c is `*(c+cstr*(i-1))`.

7. Solution of Block Hessenberg Systems

Once the patch matrices are in place, the routines `bhrsolve` and `bhlsolve` can be called as often as needed. The former (the name stands for block Hessenberg *right* solver) solves $Ax = b$, while the latter (the *left* solver) solves $x^{\mathrm{T}}A = b^{\mathrm{T}}$. Both overwrite b with the solution. Both require a function to multiply by A_{ne}. The calling sequence for `bhrsolve` is given in Figure 7.1, and that `bhlsolve` in Figure 7.2.

```
void bhrsolve(struct bhtree *a, double *b, int bstr, ANECOMP *anecomp)
/*
   bhrsolve (block Hessenberg right solve) solves the block Hessenberg
   system Ax = b, overwriting b with the solution.

   On invocation
      a         is a pointer to the bhtree containing the matrix A.
      b         is a pointer to the right hand side of the equation.
      bstr      is the stride of b.
      anecomp   is a pointer to a routine to manipulate A_ne.

   On return
      b      contains the solution of the equation.
*/
```

Figure 7.1: The Function `bhrsolve`

```
void bhlsolve(struct bhtree *a, double *b, int bstr, ANECOMP *anecomp)
/*
   bhlsolve (block Hessenberg left solve) solves the block Hessenberg
   system x'A = b', overwriting b with the solution.

   On invocation
      a         is a pointer to the bhtree containing the matrix A.
      b         is a pointer to the right hand side of the equation.
      bstr      is the stride of b.
      anecomp   is a pointer to a routine to manipulate A_ne.

   On return
      b      contains the solution of the equation.
*/
```

Figure 7.2: The Function `bhlsolve`

8. A Markov Model of a Reservoir

One of the principal applications of BHESS is to the analysis of Markov chains, and particularly to queueing models. In this section we present a Markov model of a reservoir. The process illustrates some of the economies that can be obtained in specialized applications.

8.1. Background

The problem treated here is called the Odoom–Lloyd dam model. The following sketch is intended to get the reader quickly to the matrix problem that must be solved. For more technical details see the exposition in [3].

In this model things happen at discrete points in time called epochs. At the kth epoch, Mother Nature puts r_k units of water into a reservoir behind a dam, and at the same time the dam keeper releases one unit of water. Thus if d_k denotes the amount in the reservoir just before the kth epoch, then

$$d_{k+1} = \max\{0, d_k + r_k - 1\}.$$

The amount of water coming into the reservoir is determined by a Markov chain with states 1, 2, ..., m. The ith state represents the input of $i-1$ units of water. The transition matrix for the chain will be denoted by W. Thus w_{ij} is the probability that after an input of $i-1$ units occurs at one epoch the next epoch will see an input of $j-1$ units.

The pairs (d_n, r_n) at each epoch also form a Markov process, but one with an infinite number of states, since we have put no upper bound on the d_i. If we order the states as follows

$$(0,0),(0,1),\ldots,(0,m-1),(1,0),(1,1),\ldots,(1,m-1),\ldots,$$

then the transition matrix for the problem can be written as follows. Partition

$$W = \begin{pmatrix} w_1^{\mathrm{T}} \\ w_2^{\mathrm{T}} \\ \vdots \\ w_m^{\mathrm{T}} \end{pmatrix},$$

by rows, and let

$$A_i = \mathbf{e}_{i+1} w_{i+1}^{\mathrm{T}}, \qquad i = 0, 1, \ldots m-1,$$

where $\mathbf{e}_i$ is the ith unit vector. Then the matrix has the form

$$\begin{pmatrix} A_0 + A_1 & A_2 & A_3 & A_4 & \cdots & A_{m-1} & 0 & 0 & \cdots \\ A_0 & A_1 & A_2 & A_3 & \cdots & A_{m-2} & A_{m-1} & 0 & \cdots \\ 0 & A_0 & A_1 & A_2 & \cdots & A_{m-3} & A_{m-2} & A_{m-1} & \cdots \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \end{pmatrix}$$

We will be interested in computing the nonnegative matrix G, whose (i,j)-element represents the probability encountering state (k,j) starting from state $(k+1,i)$. Note that unless the row sums of G are one, there is a finite probability that the reservoir could rise to a level k and never sink below it. Since any real reservoir has finite capacity, this is a very undesirable state of affairs. In this case the process is called nonrecurrent.

We can compute an approximation to G as follows. Let Q_n be the matrix

$$I - \begin{pmatrix} A_0 & A_1 & A_2 & A_3 & \cdots & A_{m-2} & A_{m-1} & 0 & \cdots \\ 0 & A_0 & A_1 & A_2 & \cdots & A_{m-3} & A_{m-2} & A_{m-1} & \cdots \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \end{pmatrix}$$

truncated so that it is of (block) order n. Let the right patch matrix V be generated for this matrix, and let V_t be the block corresponding to the tear index t. Then

$$G \cong G_n \equiv -V_t R,$$

where R is the R-factor from the full rank factorization $A_0 = QR$ (see §6). Thus an approximation to G is available from the patch generation step.

8.2. Some observations

Before turning to code to compute G, let's make some observations on the computational aspects of the problem.

1. The matrix Q_n is very sparse — only one row per block. Moreover, the rows are copies of the rows of W. The routine that computes the product of A_{ne} and a vector should take advantage of these facts.

2. The matrix A_{sw} is always
$$-A_0 = -\mathbf{e}_1 w_1^{\mathrm{T}}, \tag{8.1}$$
which is of rank one. Thus V will be a single column. Moreover, $-\mathbf{e}_1 w_1$ is a full-rank factorization of A_{sw}, so that there is no need for BHESS to compute it.

3. The diagonal blocks are all $I - A_1$. Thus we can compute one QR decomposition and have it serve all the leaves of the `bhtree`.

4. The matrix Q_n is block Toeplitz. This means that if n is a power of two, the patch matrices at each level of the `bhtree` are identical, and we need only compute one per level. However, that would require a complete restructuring of BHESS. Whether the savings are worth the trouble is problematical. See the operation counts in [4].

One last point. For debugging purposes, it is a good idea to have a problem whose answer can be recognized. It can be shown that if the process is recurrent, then $G = (w_1 \; w_1 \; \ldots \; w_1)^{\mathrm{T}}$. To get a problem with a known transition to nonrecurrence, we take

$$W = \frac{\mathbf{e}\mathbf{e}^{\mathrm{T}}D}{\mathbf{e}^{\mathrm{T}}D\mathbf{e}},$$

where $D = \mathrm{diag}(1, \alpha, \alpha^2 \ldots, \alpha^{m-1})$ for some $\alpha > 0$. It can be shown that the point where the process becomes nonrecurrent is the positive root of the equation

$$(m-2)\alpha^{m-1} + (m-3)\alpha^{m-2} + \cdots + \alpha^2 - 1 = 0.$$

For $m = 5$ the transition point is $\alpha = 0.5677\ldots$

8.3. The program

We begin by making some global definitions and declarations.

```
#include <math.h>

#include "dmat.h"
#include "dclams.h"
#include "bh.h"
#include "dhqrdc.h"

#define NULL 0

#define MMAX 11
#define NMAX 513

int m, n, dbx[NMAX];
double w[MMAX][MMAX], g[MMAX][MMAX];
struct dmat *aii;
struct dhqrdc *aiiqr, *aswfr;

USERGET daminit;
ANECOMP dammult;
```

The headers are respectively for the C math library, an implementation of matrices in C, a set of C linear algebra modules, and a QR decomposition.

The declarations suggest our strategy for dealing with $A_{\rm sw}$ and A_{ii}. Specifically, we will compute factorization of these matrices in common storage. Then the initialization routine has only to assign pointers to them when it is called by make_bhtree.

We begin with some initializations.

```
main()
{
   int i, j, k;
   double sum, alpha;
   struct bhtree *a;

   printf("Enter m, n, alpha: ");
   scanf("%d %d %lf", &m, &n, &alpha);
```

Next we generate W and use the utility program dprtarry to print it.

```
   sum = 0;
   for (i=1; i<=m; i++)
      sum = sum + pow(alpha, (double) i-1);

   for (i=1; i<=m; i++)
      for (j=1; j<=m; j++)
         w[i][j] = pow(alpha, (double) j-1)/sum;

   dprtary("W =", m, m, &w[1][1], MMAX);
```

Now we compute a QR decomposition of $A_{ii} = I - A_1$.

```
   aii = make_dmat(m, m);
   for (j=1; j<=m; j++){
     aii->val[2][j] = -w[2][j];
     aii->val[j][j] = aii->val[j][j] + 1;
   }
   aiiqr = dhqr_dc(aii, -1, 1, 0);
```

The program dhqr_dc generates the QR factorization.

The creation of a full rank factorization of $A_{\rm sw} = -A_0$ is similar, except now we can initialize Q and R ourselves [see (8.1)].

```
   aswfr = make_dhqrdc(m, m, 0);
   aswfr->effrnk = 1;
   for (i=1; i<=m; i++){
      aswfr->q->val[i][1] = 0.;
      aswfr->r->val[1][i] = -w[1][i];
   }
   aswfr->q->val[1][1] = 1.;
```

Next we create a bhtree, complete with patches.

```
    dbx[1] = 1;
      for (i=1; i<=n; i++)
         dbx[i+1] = dbx[i] + m;

      a = make_bhtree(n, dbx, 1, n, daminit, NULL, 0, NULL);

      patchgen(a, 1, dammult);
```

The main program ends by computing and printing G.

```
      for (i=1; i<=m; i++){
         for (j=1; j<=m; j++){
            g[i][j] = 0.;
            for (k=1; k<=a->aswfr->effrnk; k++){
               g[i][j] = g[i][j] + a->v->val[i+a->dbx[a->tear+1]-1][k]*
                           a->aswfr->r->val[k][j];
            }
         }
      }
      dprtary("-G =", m, m, &g[1][1], MMAX);
   }
```

Because of the work we have already done, the user program to help initialize the `bhtree` is particularly simple.

```
   void daminit(struct bhtree *a)
   {
      if (a->first != a->last){
         a->aswfr = aswfr;
         a->aswfr->status = 1;
      }
      else{
         a->aiiqr = aiiqr;
         a->aiiqr->status = 1;
      }
   }
```

The status flags for both `aswfr` and `aiiqr` are set to one to tell **mkbhtree** that the factorizations are already in place. Note that we do not bother to initialize `a_ne`, since it will be reconstituted from global data.

The program to multiply by A_{ne} is the trickiest part of the code. It begins by computing the number of rows and columns in A_{ne} and setting the code up for addition or subtraction.

```
void dammult(struct bhtree *a, double *b, int bstr,
                                double *c, int cstr, int job)
{
   int i, ii, j, jj, k, nr, nc;
   double pmone;

   nr = a->dbx[a->tear+1]-a->dbx[a->first];
   nc = a->dbx[a->last+1]-a->dbx[a->tear+1];

   if (job==1 || job==2)
      pmone  = -1.;
   else if (job==3 || job==4)
     pmone = 1.;
```

Now comes the actual multiplication.

```
   if (job==1 || job==3){
      for (i=3; i<=m; i++){
         if ((ii=nr-(i-2)*m+i)<1)
            break;
         for (j=i; j<=m; j++){
            if ((jj=(j-i)*m+1)>nc)
               break;
            for (k=1; k<=m; k++){
               *(b+(ii-1)*bstr) = *(b+(ii-1)*bstr) -
                        *(c+(jj-1)*cstr)*w[j][k]*pmone;
               jj++;
            }
            ii++;
         }
      }
   }
```

As a practical matter, the only way to understand this code is to work through it. Here are some hints. The integers `i` and `j` range over the rows of W that are in A_{ne} (hence they are never less than three). The indices `ii` and `jj` represent the corresponding positions in the arrays `c` and `b`. The `break`s prevent boundary overflow when A_{ne} is small.

The code for transpose multiplication is similar and is not reproduce here.

Here is a typical run for the recurrent case

```
Enter m, n, alpha: 5 100 .5
W =
  5.1613e-01   2.5806e-01   1.2903e-01   6.4516e-02   3.2258e-02
  5.1613e-01   2.5806e-01   1.2903e-01   6.4516e-02   3.2258e-02
  5.1613e-01   2.5806e-01   1.2903e-01   6.4516e-02   3.2258e-02
  5.1613e-01   2.5806e-01   1.2903e-01   6.4516e-02   3.2258e-02
  5.1613e-01   2.5806e-01   1.2903e-01   6.4516e-02   3.2258e-02
-G =
 -5.1613e-01  -2.5806e-01  -1.2903e-01  -6.4516e-02  -3.2258e-02
 -5.1613e-01  -2.5806e-01  -1.2903e-01  -6.4516e-02  -3.2258e-02
 -5.1613e-01  -2.5806e-01  -1.2903e-01  -6.4516e-02  -3.2258e-02
 -5.1613e-01  -2.5806e-01  -1.2903e-01  -6.4516e-02  -3.2258e-02
 -5.1613e-01  -2.5806e-01  -1.2903e-01  -6.4516e-02  -3.2258e-02
```

And here is a run for the nonrecurrent case.

```
Enter m, n, alpha: 5 100 .6
W =
  4.3373e-01   2.6024e-01   1.5614e-01   9.3685e-02   5.6211e-02
  4.3373e-01   2.6024e-01   1.5614e-01   9.3685e-02   5.6211e-02
  4.3373e-01   2.6024e-01   1.5614e-01   9.3685e-02   5.6211e-02
  4.3373e-01   2.6024e-01   1.5614e-01   9.3685e-02   5.6211e-02
  4.3373e-01   2.6024e-01   1.5614e-01   9.3685e-02   5.6211e-02
-G =
 -4.3373e-01  -2.6024e-01  -1.5614e-01  -9.3685e-02  -5.6211e-02
 -3.8770e-01  -2.3262e-01  -1.3957e-01  -8.3744e-02  -5.0246e-02
 -3.4655e-01  -2.0793e-01  -1.2476e-01  -7.4855e-02  -4.4913e-02
 -3.0975e-01  -1.8585e-01  -1.1151e-01  -6.6906e-02  -4.0143e-02
 -2.7684e-01  -1.6610e-01  -9.9662e-02  -5.9797e-02  -3.5878e-02
```

The solution times amounted to a hiccup on a Sparc IPC. When $m = 10$ and $n = 500$, so that the system is of order 5,000, the processing time was about seven seconds.

References

[1] J. C. S. Lui and R. R. Muntz. Evaluating bounds on staeady state availability of repairable systems from Markov models. In W. J. Stewart, editor, *Numerical Solution of Markov Chains*, pages 435–454, Amsterdam, 1989. North Holland.

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

[3] M. F. Neuts. *Structured Stochastic Matrices of M/G/1 Type and Their Applications.* Marcel Dekker, New York, 1989.

[4] G. W. Stewart. On the solution of block hessenberg systems. Technical Report CS-TR-2973, Department of Computer Science, University of Maryland, College Park, 1992. To appear in *Numerical Linear Algebra and Applications.*.

[5] U. von Matt and G. W. Stewart. Rounding errors in solving block hessenberg systems. Institute for Advanced Computer Studies, Universiyt of Maryland, College Park, MD 20742, 1994.

Appendix A. Tutorial Example

In this appendix We present a simple program that illustrates the how to set up an application of BHESS. It is hoped that it will be useful to people using the package for the first time.

A.1. Global stuff

We start by including some files.

```
#include <math.h>

#include "dmat.h"
#include "dclams.h"
#include "bh.h"
```

The file `math.h` is from the standard C library. The remaining files are as follows.

`dmat.h`	header for general purpose matrix routines
`dclams.h`	header for C linear algebra modules
`bh.h`	header for BHESS

Now for some definitions and global declarations.

```
#define gabs(a) (a>=0 ? a : -(a))
#define gmax(a,b)  (a>=b ? a : b)
#define gmin(a,b)  (a<=b ? a : b)
#define NULL 0

double b[100][3], c[3][100], x[100][3];

USERGET getit;
ANECOMP anecomp;
```

The arrays `b` and `c` will contain the right-hand sides for `bhrsolve` and `bhlsolve` respectively. The array `x` contains the solution and is used to generate `b` and `c`. Also declared are the functions used to initialize the block Hessenberg matrix and multiply by A_{ne}.

A.2. The main program

The main program begins with declarations that specify the structure of the matrix.

```
main()
{
   int n=8, dbx[] = {0,1,2,6,9,10,11,13,15,16};
   int i;
   struct bhtree *a;
```

This structure is just the one pictured in (4.2).

The next thing to do is to create the bhtree.

```
   a = make_bhtree(n, dbx, 1, n, getit, NULL, 0, NULL);
```

Since this is a top level call, the parameter `first` is `1` and the parameter `last` is `n`. The `NULL` parameter is for working storage, which `make_bhtree` generates for itself and passes down the subtree. The next to last `0` says that BHESS will not gobble the storage allocated for A_{sw}. The last `NULL` says that this is a new structure.

It is now time to generate the right hand sides. The procedure is to first generate a solution and then use BHESS utility routines to multiply it by the matrix. We will be solving four systems — two with `bhrsolve` and two with `bhlsolve`. In both cases one solution will be the vector consisting of all ones and the second will be the vector whose components are 1, 2, 3, ... We begin by generating these solution vectors.

```
   for (i=1; i<=dbx[n+1]-1; i++){
      x[i][1] = 1;
      x[i][2] = i;
   }
```

To generate the right hand side we use two BHESS utility routines that calculate Ax and $x^{\mathrm{T}}A$. Their calling sequences are given in Figure A.1. (Note that these routines will not work if the `bhtree` has been initialized with the gobble option.) They are used as follows.

```
   dbh_xeay(a, &b[1][1], 3, &x[1][1], 3, anem);
   dbh_xeay(a, &b[1][2], 3, &x[1][2], 3, anem);
   dbh_xeyta(a, &c[1][1], 1, &x[1][1], 3, anem);
   dbh_xeyta(a, &c[2][1], 1, &x[1][2], 3, anem);
```

These calling sequences also illustrate the use of strides. The components of x are stored columnwise in the array `x`. Since C stores the elements of an array by rows and because `x` has three columns, the stride is three. Similar considerations apply to the array `b`. On the other hand the components of the left hand-side c^{T} are stored by rows in the array `c`. Since consecutive elements of a row of an array are next to one another in storage, the stride is one.

The following statement tests to see whether we can reinitialize the structure successfully. Such a redundant reinitialization would never occur in practice.

```
void dbh_xeay(struct bhtree *a, double *x, int xstr,
                         double *y, int ystr, ANECOMP *anecomp)
/*
   dbh_xeay computes the product x of a and y.  The matrices
   A_sw and A_ii must not have been gobbled by make_bhtree.
*/

void dbh_xeyta(struct bhtree *a, double *x, int xstr,
                         double *y, int ystr, ANECOMP *anecomp)
/*
   dbh_xeytp computes the product x of y' and a.  The matrices
   A_sw and A_ii must not have been gobbled by make_bhtree.
*/
```

Figure A.1: The Functions `dbh_xeay` and `dbh_xeyta`

```
a = make_bhtree(n, dbx, 1, n, getit, NULL, 0, a);
```

Now we generate the patches and print out the bhtree using the utility function `printbh`.

```
patchgen(a, 3, anem);
printbh(a);
```

The function `printbh` prints the tree in inorder; that is, it prints the information in the northwest subtree, then its own information, and finally the information in the southeast subtree. The subtrees in turn are printed in inorder. Figure A.2 gives an initial segment of the output.

Finally we solve the systems and print the results using the utility function `dprtary`.

```
bhrsolve(a, &b[1][1], 3, anem);
bhrsolve(a, &b[1][2], 3, anem);
bhlsolve(a, &c[1][1], 1, anem);
bhlsolve(a, &c[2][1], 1, anem);
dprtary("B", dbx[n+1]-1, 2, &b[1][1], 3);
dprtary("C", 2, dbx[n+1]-1, &c[1][1], 100);
```

This piece of code also illustrates the difference between the stride of a vector and the stride of an array. The stride of a vector is the difference between two consecutive elements. The stride of an array is the difference between two consecutive elements in a column, or equivalently, the second dimension in the declaration of the array. Since

```
first=1 last=1 tear=1
dbx[1,...,2] = 1 2
A_ii
  3.0000e+00
A_ii: Q
  1.4142e+00
A_ii: R
 -3.0000e+00

first=1 last=2 tear=1
dbx[1,...,3] = 1 2 6
A_ne
  1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00
A_sw
  2.0000e+00
  3.0000e+00
  4.0000e+00
  5.0000e+00
A_sw: Q
 -2.7217e-01
 -4.0825e-01
 -5.4433e-01
 -6.8041e-01
A_sw: R
 -7.3485e+00
Patches
U
 -2.4495e+00   2.7217e-01   2.7217e-01   2.7217e-01   2.7217e-01
V
  7.0561e-02
 -1.2096e-02
 -3.9313e-02
 -6.6529e-02
 -9.3746e-02
```

Figure A.2: Printout of Part of a `bhtree` `dprtary` is printing an array, rather than a vector, the stride for `b` is 3 and the stride for `c` is 100.

The test case concludes by deallocating the `bhtree`.

```
    kill_bhtree(a, anec);
}
```

Note that if you have been playing games with the pointers in the `bhtree`, `kill_bhtree` is likely to die in an unpleasant manner.

A.3. The function `getit`

The function `getit` is called by `make_bhtree` to initialize the matrices in the `bhtree`. It begins with some declarations and useful abbreviations.

```
void getit(struct bhtree *a)
{
int i, j, nr, nc, first, last, tear, *dbx;
struct dmat *ane, *asw, *aii;

   first = a->first;
   last = a->last;
   tear = a->tear;
   dbx = a->dbx;
```

We now ask a question.

```
   if (first != last){
```

If the first and last blocks of the current subtrees are not the same, then we must initialize $A_{\rm ne}$, $A_{\rm sw}$, and the rank criterion `eps`.

For this example we set the elements of the ith row of $A_{\rm ne}$ equal to i, so that it has the form illustrated below:

$$A_{\rm ne} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{pmatrix}.$$

The following code generates $A_{\rm ne}$.

```
      nr = dbx[tear+1]-dbx[first];
      nc = dbx[last+1]-dbx[tear+1];
      if (!(ane=a->a_ne))
         ane = make_dmat(nr, nc);
      for (i=1; i<=nr; i++)
         for (j=1; j<=nc; j++){
            ane->val[i][j] = i;
      }
      a->a_ne = (void *) ane;
```

The number of rows and columns of A_{ne} is computed as in (4.3). In this example we have chosen to represent A_{ne} as a `dmat`. In practice we would probably use a structure that takes advantage of special properties of A_{ne}. Note the cast to type `void` in the initialization of `a->a_ne`. Note also that we first make sure `a->a_ne` is `NULL` before calling `make_dmat`. If it were not `NULL`, we would be reinitializing an existing structure and could use the pointer as is.

The (i, j)-element of A_{sw} is set to $i + j$. This insures that A_{sw} has rank of at most two. The code is analogous to the code for A_{ne}.

```
        nr = dbx[tear+2]-dbx[tear+1];
        nc = dbx[tear+1]-dbx[tear];
        if (!(asw=a->a_sw))
           asw = make_dmat(nr, nc);
        for (i=1; i<=nr; i++){
           for (j=1; j<=nc; j++){
              asw->val[i][j] = i+j;
           }
        }
        a->a_sw = asw;
     }
```

If the first and last block are the same, we are at the bottom of the tree and must initialize A_{ii}. For this example we make A_{ii} a diagonally dominant matrix of the form illustrated below for a matrix of order 4:

$$A_{ii} = \begin{pmatrix} 6 & 1 & 1 & 1 \\ 1 & 6 & 1 & 1 \\ 1 & 1 & 6 & 1 \\ 1 & 1 & 1 & 6 \end{pmatrix}. \tag{A.1}$$

Here is the code.

```
     else{
        nr = dbx[first+1]-dbx[first];
        nc = nr;
        if (!(aii=a->a_ii))
           aii = make_dmat(nr, nc);
        for (i=1; i<=nr; i++)
           for (j=1; j<=nc; j++){
              aii->val[i][j] = (nr+1)*(i/j)*(j/i)+1;
           }
        a->a_ii = aii;
     }
```

The initialization routine concludes by setting the criterion for determining rank.

```
     a->eps = 1.e-10;
  }
```

A.4. Manipulating A_{ne}

The following routine for manipulating A_{ne} can be used whenever it is represented by a `dmat`.

```
void anec(struct bhtree *a, double *b, int bstr,
                            double *c, int cstr, int job)
{
   int i, j;
   struct dmat *ane;
   double pmone;

   ane = (struct dmat *) a->a_ne;
```

If `job` is -1, the program must deallocate A_{ne}. In this case we may use the function `kill_dmat`.

```
   if (job == -1){
      kill_dmat(ane);
      return;
   }
```

Otherwise, we must compute a product of `b` with A_{ne} and combine it with `c`. First the functions sets the variable `pmone` (for plus or minus one), according as the product is to be added or subtracted.

```
   if (job==1 || job ==2)
      pmone = 1.;
   else if (job==3 || job==4)
      pmone = -1.;
```

Then the function computes the products.

```
   if (job==1 || job==3)

     /* Compute b = b +/- Ac. */

     for (i=1; i<=ane->nr; i++)
         *(b+(i-1)*bstr) = *(b+(i-1)*bstr) -
             pmone*d_xty(ane->nc, &ane->val[i][1], 1, c, cstr);

   if (job==2 || job==4)

     /* Compute b' = b' +/- c'A. */

      for (j=1; j<=ane->nc; j++)
         *(b+bstr*(j-1)) = *(b+bstr*(j-1)) -
             pmone*d_xty(ane->nr, c, cstr, &ane->val[1][j], ane->str);
}
```

Note the use of the CLAM (C Linear Algebra Module)

```
d_xty(int n, double *x, int xstr, double *y, int ystr)
```

which computes the inner product of the vectors pointed to by `x` and `y`.

Appendix B. Support

BHESS does not exist in a vacuum; it requires the support of other programs. We have already seen programs for generating the structures `dmat` and `dhqrdc`, and we have used the CLAM `d_xty` to compute inner products. In addition, BHESS requires programs to solve equations involving the diagonal blocks and to decompose A_{sw} to generate the patch matrices. In this section we list these programs along with brief descriptions of their functions.

B.1. The QR decomposition (dhqrdc.h, dhqrdc.c)

`dhqr_dc`	computes the QR decomposition of a matrix. The orthogonal (Q) part of the decomposition is represented by a product of Householder transformations. At the users request, column pivoting and rank checking will be performed.
`dhqr_xeaix`	takes as input the unpivoted Householder QR decomposition of a square matrix A and a vector x. It overwrites x with $A^{-1}x$.
`dhqr_xextai`	takes as input the unpivoted Householder QR decomposition of a square matrix A and a vector x. It overwrites x with $x^{\mathrm{T}}A^{-1}$.
`struct dhqrdc`	is a structure containing a Householder QR decomposition.
`make_dhqrdc`	is a constructor for the structure `dhqrdc`.
`kill_dhqrdc`	is a destructor for the structure `dhqrdc`.

B.2. Matrices (dmat.h, dmat.c)

`struct dmat`	A structure containing a double precision matrix.
`make_dmat`	A constructor for the structure `dmat`.
`kill_dmat`	A destructor for the structure `dmat`.
`resize_dmat`	changes the dimensions of a `dmat`.

B.3. C linear algebra modules (dclams.h, dclams.c)

The file `dclams.c` contains programs to perform basic functions with vectors and matrices. In what follows A is a matrix, x and y are vectors, and s is a scalar.

d_xes	$x_i = s$.
d_xeey	interchanges x and y.
d_xey	$x = y$.
d_xexpsy	$x = x + sy$.
d_xty	returns $x^{\mathrm{T}}y$.
d_xhy	returns $x^{\mathrm{H}}y$.
d_xesx	$x = sx$.
d_xeyta	$x = y^{\mathrm{T}}A$.
dge_xeyha	$x = y^{\mathrm{H}}A$.
dge_aeapsxyt	$A = A + sxy^{\mathrm{T}}$.
dge_xeay	$x = Ay$.
dge_xexpsay	$x = x + sAy$.
dge_xexpsyta	$x = x + sy^{\mathrm{T}}A$.
dge_aes	$A_{ij} = s$.
d_nrm2	returns $\sqrt{\sum x_i^2}$.
d_housegen	generates a Householder transformation.
dprtary	pretty prints an array.

Appendix C. BHESS, the functions

The following is a list of the functions and structures that compose BHESS.

ANECOMP	The typdef of a function provided by the user to manipulate A_{ne}. Figure 6.1.
bhlsolve	solves the block Hessenberg system $x^{\mathrm{T}}A = b^{\mathrm{T}}$. Figure 7.2.
bhrsolve	solves the block Hessenberg system $Ax = b$. Figure 7.1.
struct bhtree	A recursive data structure used to specify a block Hessenberg matrix and the points at which it is torn. Figure 4.1.
dbh_xeay	computes Ax. Figure A.1.
dbh_xeyta	computes $x^{\mathrm{T}}A$. Figure A.1.
kill_bhtree	A function to deallocate a `bhtree`. Figure 5.3.
make_bhtree	A function to initialize a `bhtree`. Figure 5.1.
patchgen	A function to generate patch matrices. Figure 5.4.

`rightsolve`	A function used in the generation of the left patch matrix. Page 6.
`topsolve`	A function used in the generation of the right patch matrix. Page 6.
`USERGET`	A function provided by the user to help initialize a `bhtree`. Figure 5.2.