Efektywne programowanie w Matlabie. Odwracanie macierzy trójprzekątniowych metodą eliminacji Gaussa

Effective programming in Matlab.
Inverting tridiagonal matrices using Gaussian elimination

Paweł Keller[1], Iwona Wróbel[2]

Streszczenie: Celem cyklu artykułów *Efektywne programowanie w Matlabie* jest prezentacja sposobów pisania bardzo wydajnych algorytmów w języku Matlab, rozwiązujących wybrane problemy obliczeniowe. W niniejszym artykule przedstawiamy efektywną implementację metody eliminacji Gaussa zastosowanej do wyznaczania odwrotności macierzy trójprzekątniowych. Zaimplementowane zostały warianty eliminacji zarówno bez, jak i z wyborem elementów głównych. Wysoka efektywność stworzonych funkcji potwierdzona jest wykonanymi testami obliczeniowymi.

Słowa kluczowe: efektywne programowanie, Matlab, macierz odwrotna, macierz trójprzekątniowa, eliminacja Gaussa

Abstract: The series *Effective programming in Matlab* is meant to present very fast implementations of algorithms for solving various computational problems in the Matlab programming language. In this paper, we present a very efficient implementation of the Gaussian elimination algorithm applied to computing the inverse of a tridiagonal matrix. Two variants of the elimination, without and with pivoting, are considered. The high efficiency of the presented solutions is supported by computational examples.

Keywords: effective programming, Matlab, matrix inverse, tridiagonal matrix, Gaussian elimination

1. Introduction

We consider the problem of computing (in Matlab) the inverse of a nonsingular tridiagonal matrix $A \in \mathbb{R}^{n\times n}$, $A(i,j) = 0$ for $|j-i| > 1$. The problem is very interesting for two reasons. A tridiagonal matrix is a matrix of the simplest structure whose inverse is, in general, a full square matrix. In addition, in Matlab there are no build-in subroutines dedicated for computing inverses of matrices of such a type.

At this point we should note, that in many problems, where the inverse of a matrix appears, there is, in fact, no need to actually compute the inverse. Usually the problem can be solved by computing a solution of a corresponding system of linear equations or the corresponding matrix equation. Sometimes, however, the inverse itself needs to be computed.

Obviously, for computing the inverse of a tridiagonal matrix, one can try to use one of the subroutines for computing the inverse of a general square matrix. In Matlab, this can be done in several ways, e.g., `A^(-1)`, `inv(A)`, `A\eye(n)`, where n is the size of a matrix A (the `eye(n)` function creates the identity matrix of size n). The last one of the above ways is the fastest.

In the first experiment we have set $n = 20000$, generated a random tridiagonal matrix $A \in \mathbb{R}^{n\times n}$, and run the following Matlab code:

```
X = A \ eye(n);
```

where by X we denote (in the whole paper) the numerically computed inverse of the matrix A. The result appeared after two minutes (all the experiments presented in this paper were performed on the four core, 3.7GHz computer running 64-bit version of Matlab R2012b). The long computation time is no surprise, as Matlab solved the matrix equation

$$AX = I,$$

[1]Faculty of Computer Science, Wroclaw School of Information Technology, ul. Wejherowska 28, 54-239 Wroclaw, Poland; p.keller@horyzont.eu, p.keller@mini.pw.edu.pl
[2]Faculty of Mathematics and Information Science, Warsaw University of Technology, pl. Politechniki 1, 00-661 Warsaw, Poland; i.wrobel@mini.pw.edu.pl

using Gaussian elimination with pivoting, not being aware of a special form of the matrix A, and therefore used $O(n^3)$ operations to compute the inverse.

In the next section we shall introduce the Matlab data type called *sparse matrix*, which will help us to easily reduce the computation time of the algorithm based on the Matlab build-in functions to only $O(n^2)$ arithmetic operations.

In Section 3, we shall write our own function in the Matlab programming language. We shall use the simple Gaussian elimination algorithm and try to make our solution as efficient as possible.

In Section 4 we shall modify the fastest solution derived in the previous section, to use Gaussian elimination with pivoting, in order to make the function as accurate as the one based on the Matlab build-in operations, and, hopefully, significantly faster.

Some final efficiency tests and conclusions will be presented in Section 5.

2. Sparse matrices

In Linear Algebra, a *sparse matrix* is a matrix in which the number of nonzero elements is very small compared to the total number of elements. Sparse matrices are natively implemented in the Matlab system. To create a sparse matrix we usually use the `sparse` function (see [4] for more information) in one of several overloaded forms[1]. When a sparse matrix is created, Matlab stores only its nonzero elements (and locations of these elements in the matrix), in the columnwise order. When an operation involving a sparse matrix is performed, Matlab ignores the zero elements, unless it would change the result.

Using sparse matrices, we may easily write a short function that computes the inverse of a tridiagonal matrix in only $O(n^2)$ time. The function is presented in Listing 1. If we run this function to compute the inverse of a random tridiagonal matrix $A \in \mathbb{R}^{n\times n}$, where, as earlier, $n = 20000$, then the result will be obtained in only 6.5 seconds.

The function can be made even a little more efficient if we use more sophisticated way to create the sparse version of the input matrix. Using the

`sparse(A)`

command, we make Matlab search the whole matrix A for nonzero elements. We may speed up the process by pointing exactly which elements have to be stored in the sparse version of the matrix A. To this end, we have to tell the `sparse` function the row indices and column indices of all nonzero elements, and the elements themselves, in the columnwise order. The modified function is presented in Listing 2, and it computes the inverse of a tridiagonal 20000×20000 matrix in 6.1 seconds.

Listing 1. *The Matlab function that computes the inverse of a tridiagonal matrix using sparse matrices and "\" operator.*

```
function X = inv3(A)
%inv3    Tridiagonal matrix inverse.
%   inv3(A) is the inverse of the matrix A.
%   Sparse matrix and the left division "\" are used.

n = size(A,2);  % matrix size

% Making A sparse and computing the inverse...
X = sparse(A) \ eye(n);
```

Listing 2. *The Matlab function that computes the inverse of a tridiagonal matrix using explicitly created sparse matrix and the "\" operator. Please note that we have intentionally put the creation of the 3rd row of the matrix M at the beginning, in order to pre-allocate the memory space at the same time.*

```
function X = inv3s(A)
%inv3s    Tridiagonal matrix inverse.
%   inv3s(A) is the inverse of the matrix A.
%   "Hand made" made sparse matrix is used.

n = size(A,2);  % matrix size

% Preparing column-wise ordered values...
M(3,:) = [ diag(A,-1); 0 ];
M(1,:) = [ 0; diag(A,1) ];
M(2,:) = diag(A,0);
% Column indices: [1:n; 1:n; 1:n]
q = repmat( 1:n, 3, 1 );
% Row indices: [-1;0;1] * ones(1,n) + q
r = repmat([-1;0;1], 1, n) + q;
% Making the sparse version of the input matrix...
H = sparse(r(2:end-1), q(2:end-1), M(2:end-1), n, n);

% Computing the inverse...
X = H \ eye(n);
```

One can say that the time saved (only 6%) is not worth complicating the function code. However, we have done that for one more reason. In practice, a tridiagonal matrix is never stored as a full square one. Usually the superdiagonal, diagonal and subdiagonal elements are given as three separate vectors or one $3 \times n$ (or $n \times 3$) matrix. Following the way we create the sparse matrix (see Listing 2) we choose to remember the elements of the input matrix A in the matrix $M \in \mathbb{R}^{3\times n}$ in such a way that k'th column contains all nonzero elements of the k'th column of A. More precisely,

$$M(i,k) = A(k-2+i,k), \quad i \in \{1,2,3\}$$

(where we assume that $A(0,1) = A(n,n+1) = 0$), or in a little more verbose way,

$$\begin{aligned} M(1,k) &= A(k-1,k),\\ M(2,k) &= A(k,k),\\ M(3,k) &= A(k+1,k). \end{aligned} \tag{2.1}$$

Using (2.1) we may write our final implementation of the function that computes the inverse of a tridiagonal matrix using sparse matrices and Matlab left division operator. The function is given in Listing 3.

[1] It means that the function `sparse` works differently, depending on the type and the number of input arguments.

Listing 3. *The Matlab function that computes the inverse of a tridiagonal matrix using explicitly created sparse matrix and the "\" operator. The elements of a matrix being inverted are stored in the $3 \times n$ array.*

```
function X = inv3v(M)
%inv3v    Tridiagonal matrix inverse.
%   inv3v(M) is the inverse of the tridiagonal
%   matrix A whose nonzero elements are given
%   in the 3-by-n matrix M as follows:
%   M(i,k) = A(k-2+i,k), i = 1,2,3.
%   Sparse matrix and the left division are used.

n = size(M,2);  % matrix size

% Making the sparse version of the input matrix...
% Row indices: [-1;0;1] * ones(1,n) + [1:n; 1:n; 1:n]
% Column indices: [1:n; 1:n; 1:n]
q = repmat( 1:n, 3, 1 );
r = repmat([-1;0;1], 1, n) + q;
H = sparse( r(2:end-1), q(2:end-1), M(2:end-1), n, n );

% Computing the inverse...
X = H \ eye(n);
```

Storing the input matrix in a memory efficient way neither increases nor decreases the performance time of the corresponding function. The 20000×20000 matrix is still inverted in 6.1 seconds. For a better clarity of the listed Matlab functions, in all the listings, we skip the argument checking part.

3. Simple Gaussian elimination

In this section, we shall write the function for inverting tridiagonal matrices, completely from the scratch, using Gaussian elimination without pivoting. If we are succeeded in creating significantly more efficient solution than the one given in Listing 3, then in the next section we shall extend the function to use Gaussian elimination with pivoting, so that our algorithm never fails.

The simple (without pivoting) Gaussian elimination algorithm is a well known and a very simple way to solve the system of linear equations

$$Ax = b \tag{3.1}$$

(see [1, §1.3], [2, §4.2] for more details). The algorithm becomes even simpler, if A is a tridiagonal matrix. The following Matlab code, under some additional assumptions on the matrix A (that LU factorisation of A exists, see [1, §1.3]), solves the system (3.1):

```
n = size(A,1);

% Elimination phase...
for k = 1:n-1
  A(k+1,k+1) = A(k+1,k+1) - A(k+1,k)/A(k,k) * A(k,k+1);
  b(k+1) = b(k+1) - A(k+1,k)/A(k,k) * b(k);
end

% Back substitution phase...
x(n) = b(n) / A(n,n);
for k = n-1:-1:1
  x(k) = ( b(k) - A(k,k+1)*X(k+1) ) / A(k,k);
end
```

All we need to do now is to adapt the above algorithm to solve the matrix equation

$$AX = I \tag{3.2}$$

instead of the system (3.1). This means, that instead of manipulating on the elements of the vector b, we have to perform analogous operations on the whole rows of the identity matrix I, or, in fact, only on those elements of each row, that actually change.

The function that solves the equation (3.2) in the case of a tridiagonal matrix A, i.e. the function that inverts a tridiagonal matrix, using simple Gaussian elimination, is given in Listing 4. In order to save memory space, as in the function `inv3v` in Listing 3, the elements of a matrix being inverted are stored in the $3 \times n$ array, according to formulae (2.1).

Listing 4. *The Matlab function that computes the inverse of a tridiagonal matrix using simple Gaussian elimination algorithm. The elements of a matrix being inverted are stored in the $3 \times n$ array.*

```
function X = inv3vnp(M)
%inv3vnp    Tridiagonal matrix inverse.
%   inv3vnp(M) is the inverse of the tridiagonal
%   matrix A whose nonzero elements are given
%   in the 3-by-n matrix M as follows:
%   M(i,k) = A(k-2+i,k), i = 1,2,3.
%   Simple Gaussian elimination is used.

n = size(M,2);  % matrix size

E = eye(n);  % identity matrix

% Elimination phase...
for k = 1:n-1
  M(2,k+1) = M(2,k+1) - M(3,k)/M(2,k) * M(1,k+1);
  E(k+1,1:k) = E(k+1,1:k) - M(3,k)/M(2,k) * E(k,1:k);
end

% Back substitution phase...
X(n,:) = E(n,:) / M(2,n);
for k = n-1:-1:1
  X(k,:) = ( E(k,:) - M(1,k+1)*X(k+1,:) ) / M(2,k);
end
```

Sadly, the last function requires 16 seconds to invert a random 20000×20000 matrix, and is much slower than the function from Listing 3, based on the build-in Matlab operations. We may observe that the instruction

```
E(k+1,1:k) = E(k+1,1:k) - M(3,k)/M(2,k) * E(k,1:k);
```

(of the elimination phase) can be simplified to

```
E(k+1,1:k) = -M(3,k)/M(2,k) * E(k,1:k);
```

with no influence on the result. This is because, before the instruction is performed, the vector $E(1 : k, k+1)$ contains only zeroes. With this modification, the computation time drops to 12.7 seconds, but this is still much too slow.

In order to considerably speed up our new function, we have to recall a very important fact related to the way Matlab uses to store matrices in the computer memory: the matrices are stored columnwise

(see [3, §5]). Considering the architecture of the modern CPUs, all computations are much more effective if they are performed on a continuous block of data. The function in Listing 4 works on rows of matrices E and X, and the elements in rows are scattered in the memory. Thus, our first optimisation step is to modify the function to perform vector operations only on the columns of the two matrices, instead of on the rows.

If the matrix X is the inverse of a matrix A, then it has to satisfy two equations, (3.2) and

$$XA = I. \tag{3.3}$$

Using the Gaussian elimination to solve the equation (3.3) we shall obtain the algorithm in which only the column operations are performed. The algorithm can be easily derived form the one in Listing 4, if we observe that the equation (3.3) is equivalent to

$$A^T X^T = (XA)^T = I^T = I.$$

From relations (2.1), we can easily see that transposing the matrix A, i.e. exchanging the elements $A(k+1,k)$ with $A(k,k+1)$, is equivalent to exchanging the elements $M(3,k)$ and $M(1,k+1)$ in the corresponding matrix M. Thus the function inverting a tridiagonal matrix by solving the equation (3.3) may look like the one in Listing 5.

Listing 5. *The Matlab function that computes the inverse of a tridiagonal matrix using simple Gaussian elimination and column operations. The elements of a matrix being inverted are stored in the $3 \times n$ array.*

```
function X = inv3vnpc3(M)
%inv3vnpc3    Tridiagonal matrix inverse.
%   inv3vnpc3(M) is the inverse of the tridiagonal
%   matrix A whose nonzero elements are given
%   in the 3-by-n matrix M as follows:
%   M(i,k) = A(k-2+i,k), i = 1,2,3.
%   Simple Gaussian elimination with column
%   operations is used.

n = size(M,2);  % matrix size

E = eye(n);  % identity matrix

% Elimination phase...
for k = 1:n-1
  M(2,k+1) = M(2,k+1) - M(1,k+1)/M(2,k) * M(3,k);
  E(1:k,k+1) = -M(1,k+1)/M(2,k) * E(1:k,k);
end

% Back substitution phase...
X(:,n) = E(:,n) / M(2,n);
for k = n-1:-1:1
  X(:,k) = ( E(:,k) - M(3,k)*X(:,k+1) ) / M(2,k);
end
```

The efficiency gain is quite noticeable. The new function, which performs column operations, requires only 2.9 seconds to compute the inverse of a tridiagonal matrix $A \in \mathbb{R}^{20000\times 20000}$, i.e. is more than twice as fast as the best solution based on Matlab build-in subroutines. We shall try, however, to make the function even more effective. The next optimisation will not be strictly related to the Matlab language.

Observe that there is no need to use extra memory space for the identity matrix E (in Listing 5). In addition, creation of such a matrix also consumes time. Thus, we shall use only one matrix variable to store the identity matrix and to store the final result. This can be easily done, as shown in Listing 6. Now, the computation time drops to 2 seconds, in the case of a random 20000×20000 tridiagonal matrix.

Listing 6. *The Matlab, memory optimised, function that computes the inverse of a tridiagonal matrix using simple Gaussian elimination and column operations. The elements of a matrix being inverted are stored in the $3 \times n$ array.*

```
function X = inv3vnpc2(M)
%inv3vnpc2    Tridiagonal matrix inverse.
%   inv3vnpc2(M) is the inverse of the tridiagonal
%   matrix A whose nonzero elements are given
%   in the 3-by-n matrix M as follows:
%   M(i,k) = A(k-2+i,k), i = 1,2,3.
%   Simple Gaussian elimination with column
%   operations is used. Memory optimised version.

n = size(M,2);  % matrix size

X = eye(n);  % initialising main variable

% Elimination phase...
for k = 1:n-1
  M(2,k+1) = M(2,k+1) - M(1,k+1)/M(2,k) * M(3,k);
  X(1:k,k+1) = -M(1,k+1)/M(2,k) * X(1:k,k);
end

% Back substitution phase...
X(:,n) = X(:,n) / M(2,n);
for k = n-1:-1:1
  X(:,k) = ( X(:,k) - M(3,k)*X(:,k+1) ) / M(2,k);
end
```

The last optimisation step may be a little surprising. We shall increase the number of arithmetic operations to decrease the computation time. During the elimination phase (see Listing 6), we modify the elements $X(i,k+1)$ only for $1 \le i \le k$. This is because the elements $X(i,k)$ for $i > k$ are all equal zero, and there is no need to consider them. Thanks to this, the cost of the elimination phase is only $n^2/2 + O(n)$ arithmetic operations.

On the other hand, in Matlab, addressing a part of a matrix is, in most cases, done quite inefficiently. We shall illustrate this by the following example. Assume n is the number of rows of the matrix X. Let us consider the two instructions

```
Y = X(1:n,k);
```

and

```
Y = X(:,k);
```

In both cases the result is exactly the same. However, the latter is about 2 times faster. In the second instruction Matlab addresses the k'th column of X directly, and thus it is done very fast. In the first instruction of the above, Matlab makes no optimisations. This means that first, a vector $[1,2,\ldots,n]$ of indices is created, and only then a new vector containing the elements $X(i,k)$ for $i \in [1,2,\ldots,n]$ is formed, which is finally assigned to the variable Y.

For the above reason, it may be more time efficient to address the whole columns of X during the elimination phase of our algorithm, even though it increases the number of arithmetic operations of this phase to $n^2 + O(n)$. The final version of the function that inverts a tridiagonal matrix using Gaussian elimination without pivoting is presented in Listing 7. Indeed, to compute the inverse of a random 20000×20000 tridiagonal matrix we now need only 1.6 seconds.

Note (see Listing 7), that we have to additionally restore the diagonal element (equal 1) which is overwritten in the case the whole columns of X are assigned the new values.

Listing 7. *The final version of the Matlab function computing the inverse of a tridiagonal matrix using simple Gaussian elimination and column operations. The elements of a matrix being inverted are stored in the $3 \times n$ array.*

```
function X = inv3vnpc(M)
%inv3vnpc2    Tridiagonal matrix inverse.
%   inv3vnpc(M) is the inverse of the tridiagonal
%   matrix A whose nonzero elements are given
%   in the 3-by-n matrix M as follows:
%   M(i,k) = A(k-2+i,k), i = 1,2,3.
%   Simple Gaussian elimination with column
%   operations is used. Fully optimised version.

n = size(M,2);  % matrix size

X = eye(n);  % initialising main variable

% Elimination phase...
for k = 1:n-1
  M(2,k+1) = M(2,k+1) - M(1,k+1)/M(2,k) * M(3,k);
  X(:,k+1) = -M(1,k+1)/M(2,k) * X(:,k);
  X(k+1,k+1) = 1;
end

% Back substitution phase...
X(:,n) = X(:,n) / M(2,n);
for k = n-1:-1:1
  X(:,k) = ( X(:,k) - M(3,k)*X(:,k+1) ) / M(2,k);
end
```

4. Gaussian elimination with pivoting

The simple Gaussian elimination not always can be performed. The algorithm will fail if any of the elements $A(k,k)$, i.e. $M(2,k)$ in our implementation $(1 \leq k < n)$, become zero. Also, if this element is small, but different from zero, large roundoff errors may appear, making the computed matrix inverse very inaccurate (see, e.g., [1, §1.3] for more details).

There are classes of matrices for which we can use the simple Gaussian elimination with no risk of failure or instability, e.g., for the class of diagonally dominant matrices. A tridiagonal matrix $A \in \mathbb{R}^{n\times n}$ is called *diagonally dominant*, if

$$|A(k,k)| > |A(k-1,k)| + |A(k+1,k)| \quad (1 \leq k \leq n)$$

or

$$|A(k,k)| > |A(k,k-1)| + |A(k,k+1)| \quad (1 \leq k \leq n).$$

In such a case the function `inv3vnpc` from Listing 7 is very fast and very accurate.

In the case of a general tridiagonal matrix, some pivoting technique has to be applied to the Gaussian elimination algorithm, in order to obtain a stable method for inverting tridiagonal matrices. We shall not discuss the pivoting scheme in details. It is clearly explained, e.g., in [2, §4.3]. As, in our algorithm, we solve the matrix equation (3.3), during the elimination phase, we shall look for the largest (in modulus) element in consecutive rows, and exchange the corresponding columns, if necessary. The function that inverts a tridiagonal matrix using Gaussian elimination with pivoting is given in Listing 8. Note that the function requires that $n \geq 2$.

Listing 8. *The Matlab function that computes the inverse of a tridiagonal matrix using Gaussian elimination algorithm with pivoting and column operations. The elements of a matrix being inverted are stored in the $3 \times n$ array.*

```
function X = inv3vppc2(M)
%inv3vnpc2    Tridiagonal matrix inverse.
%   inv3vnpc(M) is the inverse of the tridiagonal
%   matrix A whose nonzero elements are given
%   in the 3-by-n matrix M as follows:
%   M(i,k) = A(k-2+i,k), i = 1,2,3.
%   Gaussian elimination with pivoting
%   and column operations are used.

n = size(M,2);  % matrix size

M(4,n) = 0;  % additional diagonal ("sub-sub-diagonal")

X = eye(n);  % initialising main variable

% Elimination phase...
for k = 1:n-1
  % Pivoting...
  if abs(M(1,k+1)) > abs(M(1,k))
     T = M(2:4,k);
     M(2:4,k) = M(1:3,k+1);
     M(1:3,k+1) = T;
     X(:,[k,k+1]) = X(:,[k+1,k]);
  end
  % Elimination...
  M([2,3],k+1) = M([2,3],k+1)...
              - M(1,k+1)/M(2,k) * M([3,4],k);
  X(:,k+1) = X(:,k+1) - M(1,k+1)/M(2,k) * X(:,k);
end

% Back substitution phase...
X(:,n) = X(:,n) / M(2,n);
X(:,n-1) = ( X(:,n-1) - M(3,n-1)*X(:,n) ) / M(2,n-1);
for k = n-2:-1:1
  X(:,k) = ( X(:,k) - X(:,[k+1,k+2])*M([3,4],k) )...
         / M(2,k);
end
```

The pivoting increased the computation time, in the case of a random 20000×20000 matrix, to 3.7 seconds. This is mostly because we wrote the function in Listing 8 according to the (Matlab) book. As it was shown at the end of Section 3, sometimes less vectorised code may be more efficient, if we can save some background matrix creation operations.

In the function from Listing 8, a $2 \times n$ matrix is created $2n-3$ times (when we refer to $X(:,[k+1,k])$ in the pivoting phase, and to $X(:,[k+1,k+2])$ in the back substitution phase). Therefore, in the next, and our last function (see Listing 9), we have tried to avoid all possible to avoid background matrix creation operations. To this end, we have changed several

vectorised instructions to the equivalent set of scalar ones. The resulting function has significantly longer code, but is over 35% faster than the previous one. Our test matrix is inverted in 2.4 seconds.

Listing 9. *The final version of the Matlab function that computes the inverse of a tridiagonal matrix using Gaussian elimination algorithm with pivoting and column operations. The elements of a matrix being inverted are stored in the $3 \times n$ array.*

```
function X = inv3vppc(M)
%inv3vnpc2    Tridiagonal matrix inverse.
%    inv3vnpc(M) is the inverse of the tridiagonal
%    matrix A whose nonzero elements are given
%    in the 3-by-n matrix M as follows:
%    M(i,k) = A(k-2+i,k), i = 1,2,3.
%    Gaussian elimination with pivoting and column
%    operations are used. Matlab-optimised version.

n = size(M,2);   % matrix size

M(4,n) = 0;   % additional diagonal ("sub-sub-diagonal")

X = eye(n);   % initialising main variable

% Elimination phase...
for k = 1:n-1
  % Pivoting...
  if abs(M(1,k+1)) > abs(M(2,k))
     T1 = M(2,k);
     T2 = M(3,k);
     M(2,k)   = M(1,k+1);
     M(3,k)   = M(2,k+1);
     M(4,k)   = M(3,k+1);
     M(1,k+1) = T1;
     M(2,k+1) = T2;
     M(3,k+1) = 0;
     T = X(:,k);
     X(:,k)   = X(:,k+1);
     X(:,k+1) = T;
  end
  % Elimination...
  M(2,k+1) = M(2,k+1) - M(1,k+1)/M(2,k) * M(3,k);
  M(3,k+1) = M(3,k+1) - M(1,k+1)/M(2,k) * M(4,k);
  X(:,k+1) = X(:,k+1) - M(1,k+1)/M(2,k) * X(:,k);
end

% Back substitution phase...
X(:,n)   = X(:,n) / M(2,n);
X(:,n-1) = ( X(:,n-1) - M(3,n-1)*X(:,n) ) / M(2,n-1);
for k = n-2:-1:1
  X(:,k) = ( X(:,k) - X(:,k+1)*M(3,k)...
                    - X(:,k+2)*M(4,k) ) / M(2,k);
end
```

5. Final experiments and conclusions

In this section, we will compare the functions `inv3v` (Listing 3) and `inv3vppc` (Listing 9) with respect to the accuracy. We shall also make a more thorough time comparison test of the two above functions and the simple `A\eye(n)` Matlab instruction.

As the function `inv3v` solves the equation (3.2) to compute the inverse, while the function `inv3vppc` solves the (3.3) one, to make the accuracy test fair, for each generated random matrix $A \in \mathbb{R}^{100\times100}$ we also invert its transposition. We measure the error

$$\mathcal{E}(A) = \frac{\max\{||AX - I||_2, ||XA - I||_2\}}{\mathrm{cond}_2(A)}, \qquad (5.1)$$

where $||\cdot||_2$ is the second matrix norm, and $\mathrm{cond}_2(\cdot)$ is the corresponding condition number. We define the error in the above way, because a good inverse should satisfy both, (3.2) and (3.3), equations.

In Table 1 we present the results obtained for 1500000 random tridiagonal matrices of size 100. As we can see, the functions are equally accurate, which is no surprise, as both are based on the same method, Gaussian elimination with pivoting.

Table 1. Comparison of the $\mathcal{E}(A)$ error (5.1) of the functions `inv3v` and `inv3vppc`, for 1500000 random tridiagonal matrices $A \in \mathbb{R}^{100\times100}$.

function	average error	maximum error
`inv3v`	$1.7 \cdot 10^{-16}$	$2.7 \cdot 10^{-13}$
`inv3vppc`	$1.7 \cdot 10^{-16}$	$1.5 \cdot 10^{-13}$

In the final test, we compare the efficiency of the functions `inv3v`, `inv3vppc`, and the Matlab `A\eye(n)` instruction (see Section 1), for random tridiagonal matrices of different sizes, from 32 to 16384. The results are presented in Figure 1.

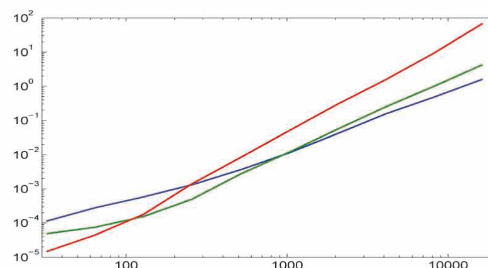

Figure 1. Dependency of the computation time of the Matlab `A\I` instruction (red), the function `inv3v` (green) and the function `inv3vppc` (blue), on the matrix size (displayed in the logarithmic scale).

Matlab is an interpretive language, and therefore a user function always has a huge efficiency disadvantage compared to the build-in one. Thus, for small matrices, the solutions based on the Matlab "\" operator are faster. However, thanks to the optimizations we have made, in the case of large matrices, our function wins, being almost 3 times faster, if a matrix size is greater than 20000. Were the efforts made worth the result? We leave the answer to the reader.

References

[1] G. Dahlquist and Å. Björck, *Numerical Methods in Scientific Computing: Volume 1*, SIAM, 2008.

[2] D. Kincaid and W. Cheney, *Numerical Analysis*, Brooks/Cole, 1993.

[3] *Getting Started with MATLAB*, Matlab documentation, MathWorks, 2007.

[4] `http://www.mathworks.com/help/matlab/`.