Fast Matlab-based Finite Element Solver for the Problem of Bending of Mindlin Plates

Alexander P. Suvorov, Nelly N. Rogacheva

Moscow State Civil Engineering University, Moscow

Abstract: In this paper we describe efficient Matlab-based implementation of the finite element solver for the problem of bending of Mindlin plates. This solver is tested on fine meshes consisting of a large number of linear rectangular finite elements. The performance of this solver is so good that even for fine meshes considered the CPU time is small enough.

Keywords: simply-supported plate, Mindlin plate theory, sparse solver, Matlab, finite element method

Introduction

Classical theory of thick plates and plates of moderate thickness is presented in references [1–4]. Various extensions of the classical theory can be found in references [5–10]. Recall that in the theory of thick plates (Mindlin plate theory) the transverse normal can rotate in such a way that after rotation it does not remain perpendicular to the mid-surface of the plate and thus the shear deformations are not zero. In contrast, in the theory of thin plates (Kirchhoff plate theory) the vertical element of the plate always remains perpendicular to the mid-surface of the plate and therefore the shear deformations are zero. For better accuracy of the solution for moderately thick and thick plates it is recommended to use Mindlin plate theory.

In [11] we analyzed bending moments in simply-supported plate and we showed that for this type of plate two kinds of boundary conditions are possible in case one uses the Mindlin plate theory. These are SS-A and SS-B types of boundary conditions and they are different in how the angles of rotations ϕ_x, ϕ_y are prescribed in the planes of edges of the plate. In particular, for SS-A plate zero rotations are prescribed but for SS-B plate zero rotations are not prescribed but the twisting moments in the planes of edges of the plate are equal to zero.

The problem of bending of Mindlin plate is solved using the finite element method and for that the plate is subdivided into linear rectangular finite elements. The square plate is considered and the number of elements along the horizontal and vertical axis is taken the same. The plate is subjected to the action of uniformly distributed vertical load (pressure). To obtain the solution of the present problem, the Matlab code by Ferreira [2] was taken as a basis but this code turned out to be slow enough for fine meshes. The objective of this paper is to propose some improvements in this code so that it remains efficient and fast even for fine meshes.

Problem Description and Results for Shear Force Distribution

As in [11] we consider a simply-supported plate that is acted upon by the uniform vertical pressure q. The plate of square form lies in the $x-y$ plane and its side length is equal to a. The thickness of the plate is h (Fig. 1).

We use the following notation. Let u, v and w denote the displacements of the mid-surface of the plate along the x, y and z axes respectively. Therefore, w is the transverse displacement or deflection. Angles of rotations of the normal to the mid-surface around the y and x axes are denoted by ϕ_x and ϕ_y. We also denote the bending moment around the axes y and x by M_x and M_y respectively, and the twisting moment is denoted by M_{xy}. In addition, Q_x signifies the shear force acting in the cross-section $x = const$, and Q_y is the shear force acting in the cross-section $y = const$.

A simply-supported plate can be defined as a plate in which the transverse displacement w is zero and also the corresponding bending moments at the edges of the plate are equal to zero, i.e.,

$$\begin{aligned} &w=0, \quad x=0,a, \quad y=0,a \\ &M_x=0, \quad x=0,a \\ &M_y=0, \quad y=0,a \end{aligned} \qquad (1)$$

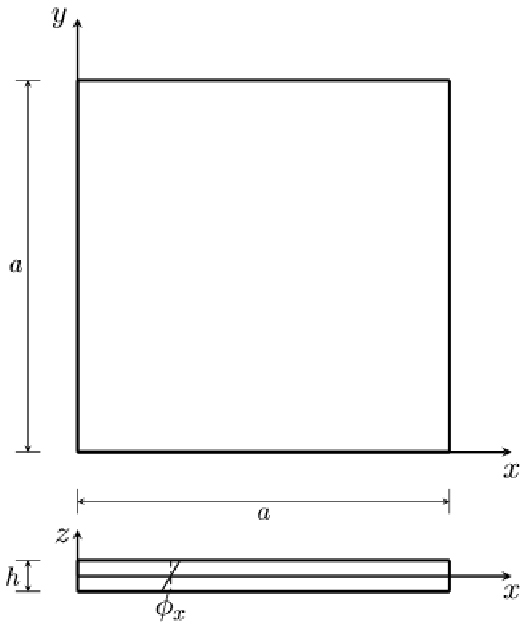

Fig. 1. – Geometry of square plate

As in [11] two types of simply-supported boundary conditions are considered. For the first type of boundary conditions, denoted here by SS-A, the zero angles of rotations are prescribed on the edges of the plate:

$$\begin{aligned} &\phi_y = 0, \quad x = 0, a, \\ &\phi_x = 0, \quad y = 0, a. \end{aligned} \qquad (2)$$

For the second type of boundary conditions, denoted by SS-B, the zero twisting moment is prescribed on the edges of the plate:

$$M_{xy} = 0, \quad x = 0, a, \quad y = 0, a \qquad (3)$$

Boundary conditions of SS-A type are much more common, while the analysis of the plate with SS-B boundary conditions is presented rarely and in [11] we showed some important differences in the solutions for the problem of bending of plates with SS-A and SS-B types of boundary conditions. Note that in the terminology of Wang et al. [1] the plates with SS-A и SS-B types of boundary conditions are denoted by SS and SS* respectively.

The solution of this problem is obtained with the help of finite element method and is based on the theory of plates of moderate thickness or Mindlin plate theory. The square plate is discretized into linear rectangular elements (4-node elements) and the number of elements along the x and y axes is the same (for example, 40 by 40 elements). As was already mentioned, the finite element program written in Matlab and presented in the book by Ferreira [2] was taken as a basis for obtaining the solution. To be able to use this program for fine meshes, say 200 by 200 elements, some important modifications must be introduced into the program and these improvements will be described below.

Before describing these improvements of the code let us illustrate the importance of solving the present problem on fine meshes. As an example consider a square simply-supported plate with a side of length $a=1$. The magnitude of the transverse pressure is $q=1$. The Young's modulus is $E=10920$ and Poisson's ratio is $\nu=0.3$. The thickness of the plate is $h=0.1$, and therefore, this plate is a plate of moderate thickness with $h/a=1/10$. This data was used in the book by Ferreira [2]. In Figs. 2–3 we show how the shear force values for this plate change with the increase in the number of elements used in discretization of the domain of the plate.

The shear force Q_y at the center of the side of the plate $y=0$ evaluated for finite elements meshes with various number of elements along the x or y axis is shown in Fig. 2. The shear force Q_y is shown for the plates with boundary conditions of SS-A and SS-B types. We see that the convergence of the shear force Q_y for the SS-B plate is slower than for the SS-A plate. Indeed, a reasonably good accuracy in the value of the shear force can be achieved with 100 elements for the SS-A plate but for the SS-B plate we need at least 200–250 elements along each side of the plate. Having the mesh with 100 elements we can get only a rough estimate of the shear force Q_y for the SS-B plate. Fig. 2 also shows that by using the meshes with small number elements, say 20 by 20 elements, it is not possible to achieve a good accuracy in distribution of shear force even for the plate with boundary conditions of SS-A type.

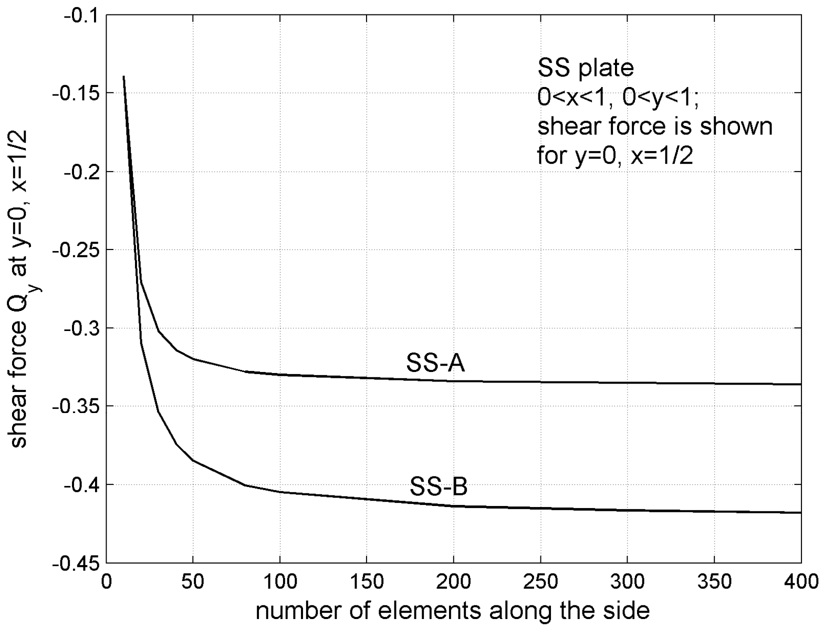

Fig. 2. – Dependence of the shear force Q_y at the center of the side $y=0$ of the square plate, subjected to uniform transverse pressure of magnitude 1, on the number of elements

Fig. 3 shows the shear force Q_y at the corner of the plate $y=0$, $x=0$. Due to the symmetry $Q_y = Q_x$ at the corner. For the SS-A plate the shear force is equal to zero, but for the SS-B plate this force is not zero. We see here that situation with the convergence of the shear force results is slightly better compared to the shear force at the center of the side. Indeed, with 150–200 elements along each side of the plate we can obtain a reasonably good approximation to the value of the shear force at the corner.

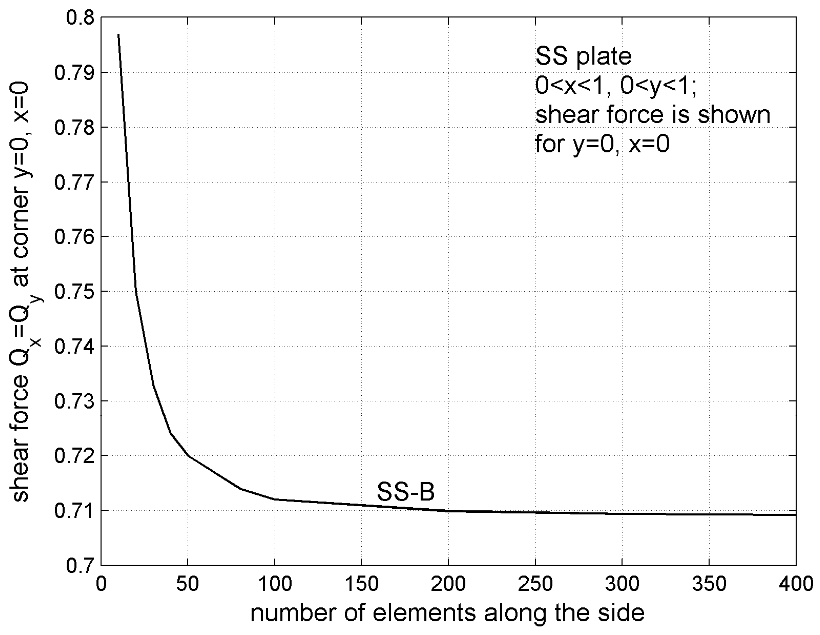

Fig. 3. – Dependence of the shear force $Q_y = Q_x$ at the corner $y = 0$, $x = 0$ of the square plate, subjected to uniform transverse pressure of magnitude 1, on the number of elements

All these results suggest that having the mesh with large number of elements is strongly desired in the analysis of the simply-supported plate, especially in case of the plate with SS-B type of boundary conditions.

Implementation of Efficient FE solver in Matlab

For solving the problem described above the Matlab code presented in the book by Ferreira [2] is used. In this code the entire stiffness matrix with all zero terms is stored in computer memory and then the Gaussian elimination procedure is used to solve the resulting system of linear equations. If one uses the mesh with $N_{elx} = N_{ely}$ elements along each side of the square plate and each element is a rectangle with 4 nodes, then the total number of nodes in the mesh is $N_n = (N_{elx} + 1)^2$. Taking into account the fact that each node has 3 degrees of freedom (one for transverse displacement and two for rotations), the total number degrees of freedom for this problem is $N_{dof} = 3(N_{elx} + 1)^2$. Thus, the global stiffness matrix has $N_t = 9(N_{elx} + 1)^4$ terms (numbers) which must be stored in computer memory. If, for example, $N_{elx} = 100$, then $N_t \approx 900000000$ or 900 mln. terms and if $N_{elx} = 50$, then $N_t \approx 56250000$ or 56 mln. terms. It appears that the memory resources of the average home-based workstation with 8 GB of RAM may be insufficient to store all this data. In fact, we have established that the maximum number of elements in 1D array that the aforementioned computer may store is about 200 mln. elements given that 8 bytes is required for storing each element of the array.

Table 1 gives the amount of CPU time that the computer spends on solution of the present problem for different sizes of the mesh. Original code of Ferreira [2] was used for solving the problem. It was not possible to solve the present problem on the mesh of 80x80 elements because the memory limits were exceeded.

Table 1

CPU time required for solving the problem with the help of original code

Mesh size	Time [sec]
20x20 elements	0.19
30x30 elements	0.94
50x50 elements	13.7
70x70 elements	95.8
80x80 elements	out of memory

To be able to run problems with large number of elements, we should take advantage of Matlab capabilities of working with sparse matrices. The most straightforward implementation is to declare the global stiffness matrix as sparse

```
K=sparse(GDof,GDof);
```

where the variable `Gdof` is the total number of degrees of freedom in the mesh. By doing so, only nonzero terms of the global stiffness matrix are stored in computer memory and the problem with memory shortage will be solved. However, after running the modified code, we observe that the computation speed is low for problems on fine meshes. Surprisingly, most of the CPU time is now spent not on the solution of the system of equations (with the help of Matlab operator “backslash”) but on assembling the global stiffness matrix. Assembling the global stiffness matrix in the code by Ferreira [2] is divided into two stages: first the bending part of the stiffness matrix for all elements is assembled, and after that the shear part of the stiffness matrix for all elements is assembled:

```
loop over all elements
  K_b = create bending part of stiffness matrix
  K(elementDof,elementDof)=...
  K(elementDof,elementDof)+K_b;
end loop
loop over all elements
  K_s = create shear part of stiffness matrix
  K(elementDof,elementDof)=...
  K(elementDof,elementDof)+K_s;
end loop
```

Here `elementDof` is an array with row and column numbers that correspond to location within the global stiffness matrix into which the local stiffness matrix of the element must be placed. In other words, `elementDof` stores all degrees of freedom of the element using global numbering.

It should be noted that the bending part of the stiffness matrix `K_b` has 64 nonzero terms while the shear part of the stiffness matrix `K_s` has 104 nonzero terms out of total 124 terms (since there are 12 degrees of freedom for each element). Therefore, `K_s` is less sparse and its assembly takes noticeably more time that the assembly of `K_b`. It is also important to note that assembly of the stiffness matrices for the first, say, 1000 elements takes considerably less time than the assembly of the matrices for the last 1000 elements in the loop. This is true because the structure of the global stiffness matrix gets more and more complex as we sequentially go through all elements in the loop and add more and more nonzero terms to the global stiffness matrix.

In view of the reasons mentioned above, we can somewhat improve the code performance (in terms of speed) by executing first the loop where the shear part of the stiffness matrix is assembled, and then placing the loop in which the bending part of the stiffness matrix is assembled, i.e. by reversing the original order:

```
loop over all elements
  K_s = create shear part of stiffness matrix
  K(elementDof,elementDof)=...
  K(elementDof,elementDof)+K_s;
end loop
loop over all elements
  K_b = create bending part of stiffness matrix
  K(elementDof,elementDof)=...
  K(elementDof,elementDof)+K_b;
end loop
```

This new code will be faster since here we first assemble the non-sparse local stiffness matrix `K_s` into a relatively empty global stiffness matrix and after that the sparse matrix `K_b` is placed into the relatively populated global stiffness matrix. Thus, we avoid the situation in which the non-sparse matrix is placed into the populated global stiffness matrix, which we had in the first code.

It is also worth mentioning that combining two loops over all elements into one loop has led to a slight slow-down of the code:

```
loop over all elements
  K_s = create shear part of stiffness matrix
  K(elementDof,elementDof)=...
  K(elementDof,elementDof)+K_s;
  K_b = create bending part of stiffness matrix
  K(elementDof,elementDof)=...
  K(elementDof,elementDof)+K_b;
end loop
```

Resulting CPU times (after this improvement) are shown in Table 2 for different meshes (again we remark that almost all CPU time is spent on the assembly of the global stiffness matrix).

Table 2

CPU time required for solving the problem
with the help of the code after the first improvement

Mesh size	Time [sec]
50x50 elements	7.9
70x70 elements	29
100x100 elements	117
150x150 elements	707

This is of course an improvement of the code performance compared to the original version but it is still not satisfying as we need a further speed-up.

A considerable improvement of the code can be achieved if we create the sparse matrix by using triplets `i`, `j`, `v`:

```
K = sparse(i,j,v);
```

where `i` is an array that contains the row numbers of all nonzero elements of the matrix `K`, `j` is an array that contains the column numbers of the nonzero elements of the matrix in the same order, and `v` is an array that stores the nonzero elements of the matrix in the same order.

At the beginning, the arrays similar to `i` and `j` can be formed locally, for each element, using the array `elementDof`, which stores all degrees of freedom of the element. Below we describe how it can be done in Matlab.

Using the function `meshgrid` we can obtain two-dimensional arrays in which degrees of freedom for the element are repeated row-wise or column-wise

```
[X,Y]=meshgrid(elementDof,elementDof);
```

Note that here in the array `X` the degrees of freedom are changing along the row, while in the array `Y` the degrees of freedom are varying along the column.

After that we transform the two-dimensional arrays `X` and `Y` into one-dimensional arrays `X1` and `Y1` by using the function `reshape`

```
X1=reshape(X,1,12^2);
Y1=reshape(Y,1,12^2);
```

and here we use the fact that there are 12 degrees of freedom for each element. The function `reshape` forms 1-D array column-wise (first, it copies elements from the first column of a two-dimensional array, then from the second column, etc.). Thus, the array `X1` will contain `dof1` (degree of freedom number 1) repeated 12 times, then `dof2` repeated 12 times, etc. On the hand, the array `Y1` will contain the sequence `dof1`, `dof2`, …, `dof12` repeated 12 times.

Next, for each finite element we transform the element stiffness matrix into 1D array `V1`, also by using the function `reshape`

```
V1=reshape(K_s+K_b,1,12^2);
```

where `K_s` and `K_b` are the shear and bending parts of the element stiffness matrix.

Obviously, we need to create similar arrays `X1`, `Y1`, `V1` for each element. Then, for each element we save its indices `X1`, `Y1` inside the global arrays `II` and `JJ` in proper places and save the components of the stiffness matrix of each element inside the global array `VV`:

```
II((e-1)*12*12+1:e*12*12)=X1;
JJ((e-1)*12*12+1:e*12*12)=Y1;
VV((e-1)*12*12+1:e*12*12)=V1;
```

where `e` is the element number. Obviously, the global arrays `II`, `JJ` will have repeated terms since most degrees of freedoms in the mesh are shared by several elements. But that is not a problem for assembling the global stiffness matrix since the Matlab function `sparse` will accumulate values of `VV` (add them up) if the indices inside the arrays `II` and `JJ` are repeated. Thus, we can conveniently use the command

```
K=sparse(II,JJ,VV);
```

which will create the desired sparse global stiffness matrix `K`.

In the code that we used we did not actually form the array `V1` but placed the components of the stiffness matrix of the element directly inside the array `VV`

```
loop over all elements
  K_s = create shear part of stiffness matrix
  VV((e-1)*12*12+1:e*12*12)=...
  VV((e-1)*12*12+1:e*12*12)+reshape(K_s,1,12^2);
end loop
loop over all elements
  K_b = create bending part of stiffness matrix
  VV((e-1)*12*12+1:e*12*12)=...
  VV((e-1)*12*12+1:e*12*12)+reshape(K_b,1,12^2);
end loop
```

In Table 3 we present the resulting CPU times required for the solution of the problem by using the efficient assembly of the sparse stiffness matrix described above. Note that these times do not include the time spent on mesh generation, which also grows as the mesh gets large.

Table 3

CPU time required for solving the problem
with the help of the code after the second improvement

Mesh size	Time [sec]
100x100 elements	3.3
200x200 elements	14.3
300x300 elements	33.7
400x400 elements	63.3

We can see now that the time spent on the assembly of the global stiffness matrix and solution of the system of equations is now considerably smaller and even for very fine meshes the solution can be obtained within a reasonable amount of time.

Conclusions

In this article we described an efficient implementation of the finite element solver for the problem of bending of simply-supported plate. This implementation is realized in Matlab software and it is based on the code written by Ferreira [2].

It was shown that for better accuracy of the solution of the present problem, and in particular, shear force distribution, it is strongly advised to use fine meshes, at least when simple linear 4-node finite elements are used in the mesh. But the finite element code proposed by Ferreira [2] could hardly be used for the fine meshes due to low speed of execution and limitations on computer memory. That was a motivation for searching the ways of code improvement.

Major improvement of the code performance was achieved by using the Matlab function `sparse` and its capability to create the sparse matrix from one-dimensional arrays with accumulation of values over the repeated indices. The latter capability is very convenient for assembling the global stiffness matrix during finite element computations.

References

1. Wang C.M., Reddy J.N., Lee K.H. Shear deformable plates and shells: relationships with classical solutions. Elsevier Science Ltd., 2000. 296 p.
2. Ferreira A.J.M. Matlab Codes for Finite Element Analysis: Solids and Structures, Solid Mechanics and Its Applications, 157. Springer Science+Business Media B.V., 2009. 235 p.

3. Khennane A. Introduction to finite element analysis using MATLAB and Abaqus. Taylor & Francis Group, LLC, Boca Raton, 2013. 487 p.

4. Mindlin R.D. Journal of Applied Mechanics, 18, 1951. pp. 31–38.

5. Ma H.H., Gao X.-L., Reddy J.N. Acta Mechanica, 220, 2011. pp. 217–235.

6. Rajapakse R.N.K.D., Selvadurai A.P.S. International Journal for Numerical Methods in Engineering, 23, 1986. pp. 1229–1244.

7. Soldatos K.P. A refined laminated plate and shell theory with applications. Journal of Sound and Vibration, 144(1), 1991. pp. 109–129.

8. Kabir H.R.H. A double Fourier series approach to the solution of a moderately thick simply supported plate with antisymmetric angle-ply laminations. Computers & Structures, 43(4), 1992. pp. 769–774.

9. Kabir H.R.H. Analysis of a simply supported plate with symmetric angle-ply laminations. Computers and Structures, 51, 1994. pp. 299–307.

10. Chaudhuri R.A. On boundary-discontinuous double Fourier series solution to a system of completely coupled PDEs. International Journal of Engineering Science, 27(9), 1989. pp. 1005–1022.

11. Suvorov A.P. Inženernyj vestnik Dona (Rus), 2019, №5. URL: ivdon.ru/ru/magazine/archive/N5y2019/5892.