

Introduction to MATLAB vectorization

Avoiding for-loops

The **for**-loops in MATLAB are useful for repeating a set of commands several times. However, there are many situations, especially when dealing with operations involving matrices, where the use of a **for**-loop can be avoided. This has a special interest in cases with large matrices (huge number of elements/degrees of freedom), where nested **for**-loops (i.e. **for**-loops inside other **for**-loops) can dramatically slow down the computation. In this regard, MATLAB has the built-in functionality of performing certain operations involving vectors and matrices with a kind of *internal parallelization* that allows to significantly speed-up element-wise operations.

A very simple example of this can be seen by performing the element-wise product of two vectors, i.e. the product of each component of one vector with the same component in the other. This operation can be carried out with a **for**-loop:

```
for i = 1:length(a)
    c(i) = a(i)*b(i);
end
```

but the efficiency can be increased by simply doing

```
c = times(a,b);
```

or

```
c = a.*b;
```

For a small vector size, the difference between both methods is not appreciable, but as the size is increased, the speed-up obtained with the **times** function or the **.*** operator becomes substantial.

For more details on which element-wise operations can be internally parallelized with MATLAB refer to the [bsxfun documentation](#).

Fast assembly operation

Typically, in structural problems, the assembly process of the stiffness/mass matrix or force vector is the most computationally demanding operation and can significantly slow down the code execution if nested **for**-loops are used.

Assuming each element stiffness matrix is stored in **Ke(:, :, e)**, each element force vector in **fe(:, e)**, the total number of degrees of freedom **ndof** is known, as well as the degrees of freedom connectivities matrix **Tdof**, the typical assembly algorithm using nested **for**-loops is:

```
K = zeros(ndof,ndof);
f = zeros(ndof,1);
for e = 1:size(Tdof,1)
    for i = 1:size(Tdof,2)
        I = Tdof(e,i);
```

```

    f(I) = f(I) + fe(i,e);
  for j = 1:size(Tdof,2)
    J = Tdof(e,2);
    K(I,J) = K(I,J) + Ke(i,j,e);
  end
end
end

```

In this case, to avoid the nested **for**-loops, one can use the **repmat**, **repelem** and **sparse** functions to perform the assembly as:

```

K = sparse(ndof,ndof);
f = zeros(ndof,1);
for e = 1:size(Tdof,1)
  i = Tdof(e,:)';
  f(i) = f(i) + fe(:,e);
  I = repmat(i,size(Tdof,2),1);
  J = repelem(i,size(Tdof,2),1);
  K = K + sparse(I,J,Ke(:, :, e),ndof,ndof);
end

```

Note that the **sparse** function has been used instead of the **zeros** function to initialize the stiffness matrix. This is to create a matrix that only allocates memory for the non-zero terms. This is necessary to save computer memory space when dealing with large matrices in which most of its components are null (such as the stiffness matrix). Additionally, having **K** stored as a sparse matrix makes it possible to efficiently carry out the matrix inversion operation (through **K\b**) with a lot less of computational effort.

The function **sparse** allows to build a sparse matrix indicating the indices (rows and columns) of the non-null components. To use this to our advantage, one needs to have a previous knowledge on how MATLAB understands a matrix as a one-dimensional vector. As an example, the matrix:

```

A = [a11, a12, a13;
     a21, a22, a23;
     a31, a32, a33];

```

Can be turned into a column-vector in MATLAB using by:

```

a = A(:);

```

where the elements will be reorganized in the following fashion:

```

a = [a11; a21; a31; a12; a22; a32; a13; a23; a33];

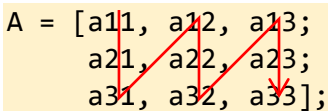
```

i.e. MATLAB sweeps through the matrix elements column-by-column:

```

A = [a11, a12, a13;
     a21, a22, a23;
     a31, a32, a33];

```



Now, in order to understand how to build a matrix with the **sparse** function, following the example above, consider:

```
A = sparse(i,j,a,3,3);
```

where **i**, **j** are the indices (rows and columns, respectively) of the position that each term in vector **a** will occupy in the matrix **A**. The last two arguments of the function are the expected size of the matrix **A**, so that MATLAB can fill the remaining terms with zeros. In this specific example the rows and columns list would be:

```
i = [1; 2; 3; 1; 2; 3; 1; 2; 3];  
j = [1; 1; 1; 2; 2; 2; 3; 3; 3];
```

Back to our case, the objective of the assembly is to add each term of an element stiffness matrix into the corresponding degrees of freedom (row and column) of the global stiffness matrix. Since we already know the list of terms to add (i.e. the equivalent column-vector of the element stiffness matrix **Ke(:, :, e)**), we just need to find the lists of rows and columns with the information of which degrees of freedom correspond to the element (given by **Tdof(e, :)**). To do so, the functions **repmat** and **repelem** can be useful to complete these lists, since

```
I = repmat(i,3,1);
```

creates a vector repeating the whole vector **i** 3 times:

```
I = [i; i; i];
```

While

```
J = repelem(i,3,1);
```

creates a vector repeating the components in vector **i** 3 times:

```
J = [i(1); i(1); i(1); i(2); i(2); i(2); ... i(N); i(N); i(N)];
```

You can refer to the MATLAB documentation for the functions [repmat](#), [repelem](#) and [sparse](#) for detailed information on how to properly operate with them.

Note: The assembly process can be further optimized by avoiding the elements' **for**-loop with a deeper knowledge on MATLAB's higher-order matrix reordering and proper use of the functions **repmat**, **repelem** and **sparse**.