OpenPulse - Open source code for numerical modelling of low-frequency acoustically induced vibration in gas pipeline systems

Theory Reference C: Fast assembly procedure
V1.0

Jacson G. Vargas, Lucas V. Q. Kulakauskas, André Fernandes,
Olavo M. Silva, José L. Souza, Diego M. Tuozzo and Ana P. Rocha
jgvargas@mopt.com.br

21 May 2020

1 Introduction

The assembly is a procedure in the Finite Element Method that allows obtaining the global matrices based on elementary matrices and elements' connectivity matrix. The global matrices are used in the construction of mathematical models of interest regardless of the type of analysis to be solved. The assembly process requires that all relevant model information such as mesh data, material and cross-section properties have been previously defined. Specifically, in *OpenPulse*, it is necessary that the mesh had already been processed and that all attributes of *Node* and *Element* classes have already been assigned.

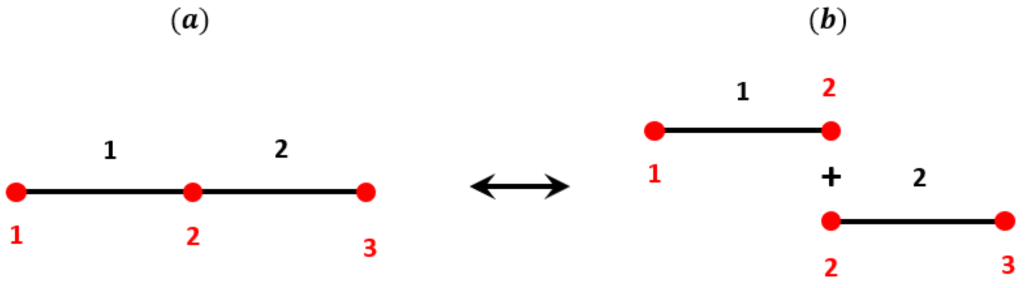


Figure 1: Element connectivity.

The beam element of *OpenPulse* (based on the Timoshenko theory) has two nodes, each node with six degrees of freedom (DOFs) totaling twelve DOFs per element. Therefore, elementary matrices have a 12x12 dimension. For example, consider a “free" model (without boundary conditions) composed of two elements and three nodes, as shown in 1a. The process of assembling the element matrices would result in global matrices with dimensions 18x18, as illustrated in Fig. 2. As can be seen, at the center of the global matrix two quadrants of elementary matrices overlap. This overlap, pictured in 1b, is the result of coincidence of DOFs (u and θ) of the second node of element 1 in relation to the first node of element 2. The coincidence ensures the continuity C_0 of the internal nodes and, therefore, the “coupling” of the DOFs at elements union.

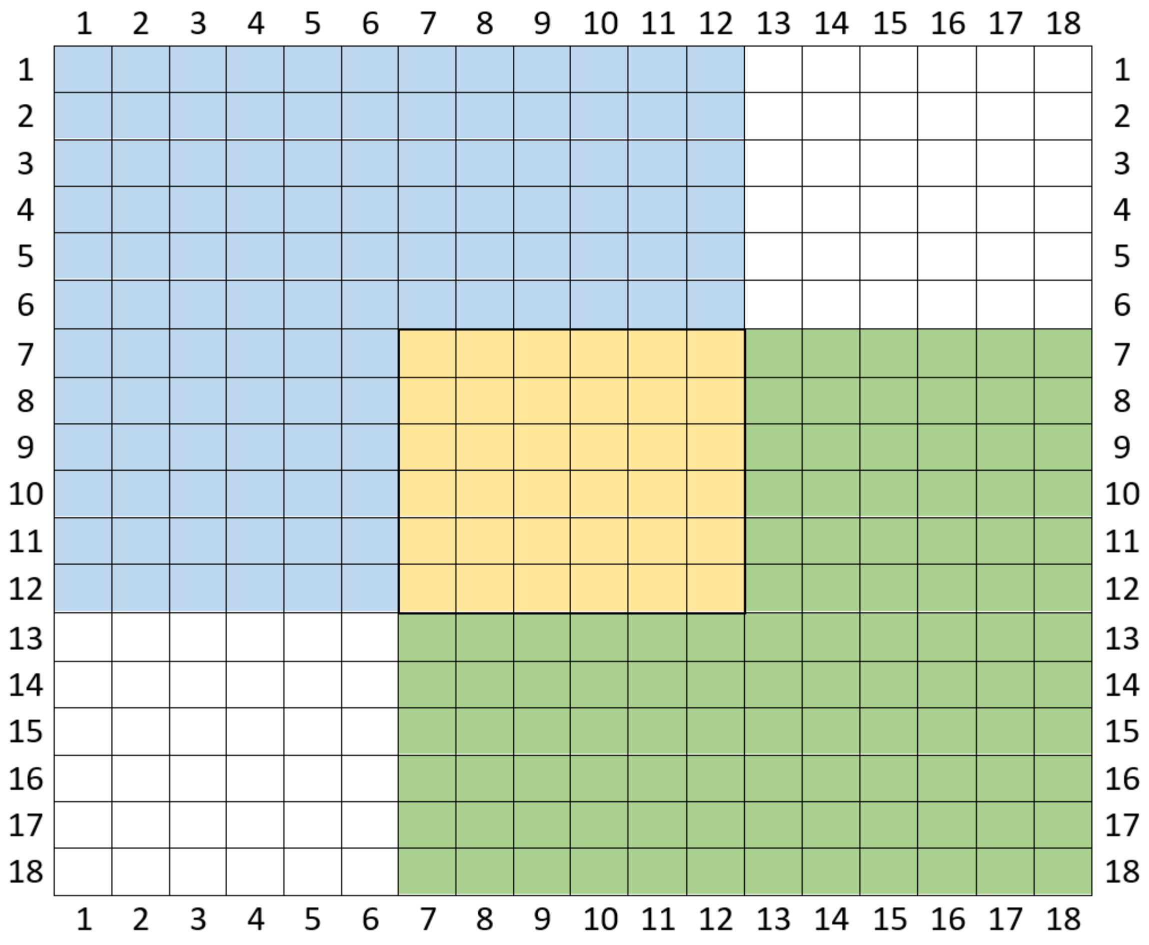


Figure 2: Example of global matrix resulting from the assembly of two beam elements.

Based on what was introduced, the assembly process consists of adding the elementary matrices in their respective spaces within the global matrices, taking account of the element’s connectivity and the global indices of the element DOFs. The DOFs coupling between elements arises when in the adding of elementary matrices the same indices are accessed. Representation through a color scheme of the coupling is shown in Fig. 2, precisely at the lines/columns 7 to 12 of matrix.

In general, finite element models can be easily constituted of hundreds of thousands or even millions of nodes and elements. In the assembly process as many elementary matrices as the number of elements will be added. Thus, the number of operations can be considerably high. The assembly algorithms must be efficient, otherwise, the time spent on assembly and the amount of memory need to store information can be too high, which is undesirable.

Depending on the volume data involved in the finite model construction, it is common that the amount of memory required to store the matrices information reaches tens of gigabytes. For example, a dense square matrix with 10^5 rows composed of number in 32bits float point notation requires 37.25GB of memory to be stored. Fortunately, the global matrices are sparse since that have a high number of null elements. For this reason, it is possible to use specific algorithms for processing sparse matrix. These algorithms considerably reduce the amount of memory required to store matrices data, since only non-null values and its respective indices are stored. Additionally, such algorithms reduce the number of operations and, consequently, the processing time since operations that are known to result in null values are not performed. Also, the data processing through sparse matrix algorithms made it possible the solution of models with millions of DOFs in desktop computers.

2 The *OpenPulse* assembly code

The *OpenPulse* assembly code was developed using functions for building sparse matrices from *SciPy* library and based on vectorization combined with memory pre-allocation. The assembly of global matrices is performed through the *csc_matrix* function whose inputs are the values of matrix elements and their respective indices for rows and columns. The application of this function is relatively convenient for the assembly process since it automatically adds two elements with the same indices. Furthermore, it allows to perform efficiently matrix-vector products, simple arithmetic operations and slicing of columns of global matrices. As applied in the *OpenPulse*, the command line used in the assembly of matrix **K**, for example, through the *csc_matrix* function, is summarized to:

$$K = csc\_matrix((dataK, (i, j)), shape = (total\_dofs, total\_dofs), dtype = float)$$

where $dataK$ are the values of matrix elements written in the form of a vector[1]; i e j the respective indices of rows and columns of matrix elements; *shape* defines the matrix dimensions; and the data type is defined by *dtype* variable.

The *OpenPulse* assembly of global matrices is performed inside $get\_global\_matrices()$ method of $Assembly()$ class. This method contains a single loop that sweep all elements of model to calculate the elementary matrices. The values obtained are stacked forming a three-dimensional data structure. The 3D data arrangement, schematized in Fig. 3, is convenient for the assembly itself and for the post-processing analyses, such as, for instance, in the calculation of stresses at elements.

The calculation of global indices of the rows and columns is performed by means of basic operations and with aid of functions from the *NumPy* library. The $get\_gloabal\_indexes()$ method of $Mesh()$ class returns, in the form of a vector, the calculated values for indices i e j. The calculation of indices is performed after the mesh processing, therefore, it occurs outside the loop, reducing the number of operations and, consequently, the processing time. After gathering all the necessary information, the assembly is carried out using the command line already mentioned.

Further information on assembly codes and global matrix indices calculation method are found in Appendix A.

3 Performance of assembly algorithms

The vectorization is a technique that allows improving the code performance of some repetitive structures. This technique reduces the processing time once repetitive operations, like loops for

[1]Let **M** be any matrix (rank 2 or 3). It's possible to represent **M** in the form of a vector, concatenating row by row using the command $M.reshape(-1)$.

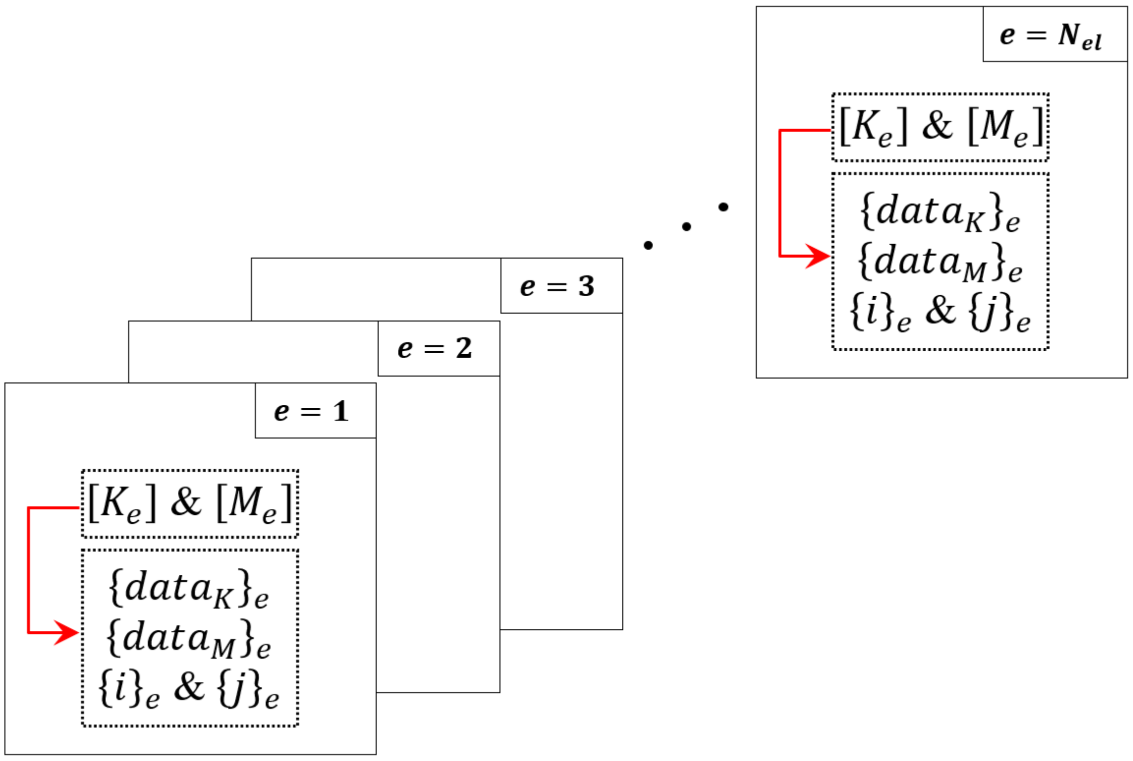
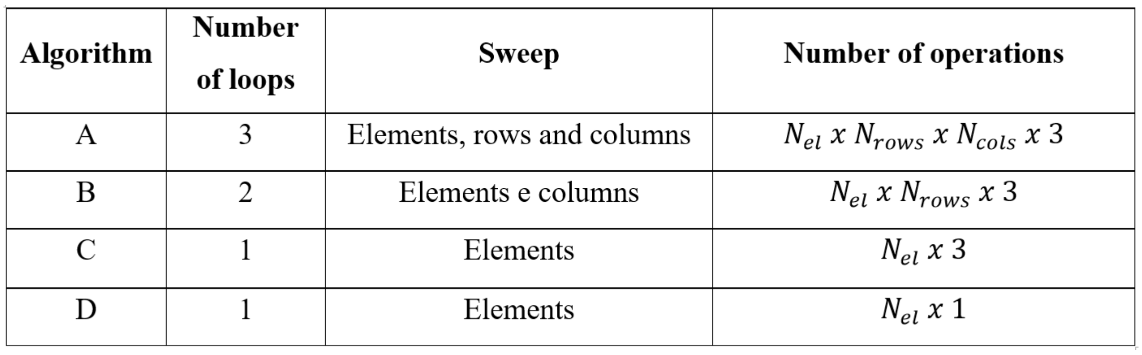


Figure 3: 3D data arrangement.



Algorithm	Number of loops	Sweep	Number of operations
A	3	Elements, rows and columns	$N_{el}\ x\ N_{rows}\ x\ N_{cols}\ x\ 3$
B	2	Elements e columns	$N_{el}\ x\ N_{rows}\ x\ 3$
C	1	Elements	$N_{el}\ x\ 3$
D	1	Elements	$N_{el}\ x\ 1$

Figure 4: Characteristics of algorithms A, B, C and D.

example, are substituted by basic operations between vectors and matrices. These operations are evaluated in one single run and normally take less time to be processed.

An assembly algorithm can be written in different ways, using 3 loops, 2 loops, or even, 1 loop. In general, the efficiency of the algorithm increases as the number of loops decreases. Significant reductions in processing time are obtained through the appropriate application of vectorization technique concomitant to the memory allocation. It is recommended to avoid concatenating operations, after all, operations that involve memory re-allocation tend to have high computational costs.

In order to evaluate the performance of assembly algorithms, four generic algorithms A, B, C and D were implemented for beam elements, like *OpenPulse* ones. The algorithms are found in Appendix B and have the characteristics summarized in Fig. 4, where N_{el} is the number of elements, N_{rows} is the number of rows and N_{cols} is the number of columns.

The algorithm A contains three loops to scan the rows and columns of all elementary matrices. The algorithm B has two loops whereas the algorithms C and D have only one loop to perform the same task. In the algorithms C and D, the number of operations performed differs depending on the structures that calculate the indices of rows and columns. According to lines 30 and 31 from algorithm C, the indices i and j for each element node pair are evaluated and stored in the vectors: $rows$ and $cols$. In algorithm D, the calculation of indices is carried out before the loop that sweeps the elements of the model (see lines 19 to 22). Thus, the number of operations of algorithm C is three times major than algorithm D.

For comparison, it has been performed a benchmark of the processing time of four algorithms, considering a model composed of 10^6+1 nodes and 10^6 elements. To ensure homogeneity in the tests, it was decided to generate matrices 12x12 of "ones" in the floating-point notation of 32bits for each element. It should be noted that the objective of benchmark test is to evaluate the performance of assembly algorithms and not to solve the finite element model itself. The average processing times obtained[2], considering ten consecutive analyses in the average calculation, for the algorithms A, B, C and D are 101.9s, 47.5s 18.23s and 6.6s, respectively. The assembly implemented in *OpenPulse* is based on algorithm D , since it presents a smaller number of operations, better performance and allows to store in a simple way the elementary matrices for the post-processing steps.

At this point, it is worth noting that the amount of memory used in storing global matrix information is considerably less. The same dense matrix of free model considered in the tests is composed of $(10^6 + 1)^2$ values. The assembly of sparse matrix requires $12^2 \times 10^6$ values in float-point notation and $2 \times 12^2 \times 10^6$ integers values to form the respective pair of indices i and j. Directly, the amount of data needed to assembly sparse matrix is 5950 times smaller compared to the dense matrix. The total memory used to storage the sparse matrix input data is 3.22GB, while dense matrix would require 3.72TB.

[2]Desktop configuration - Processor: Intel Core i7 6700K; RAM memory: 32GB HyperX CL15 2400MHz; Storage: SSD NVMe Corsair MP510 960Gb; Graphic card: NVIDIA Quadro M2000.

APPENDIX A

Method to obtain the global matrices

```
def get_global_matrices(self):

    total_dof = DOF_PER_NODE_STRUCTURAL * len(self.mesh.nodes)
    number_elements = len(self.mesh.elements)
    rows, cols = self.mesh.get_global_indexes()

    mat_Ke = np.zeros((number_elements, DOF_PER_ELEMENT, DOF_PER_ELEMENT), dtype=float)
    mat_Me = np.zeros((number_elements, DOF_PER_ELEMENT, DOF_PER_ELEMENT), dtype=float)

    for index, element in enumerate(self.mesh.elements.values()):
        mat_Ke[index,:,:], mat_Me[index,:,:] = element.matrices_gcs()

    full_K = csr_matrix((mat_Ke.flatten(), (rows, cols)), shape=[total_dof, total_dof])
    full_M = csr_matrix((mat_Me.flatten(), (rows, cols)), shape=[total_dof, total_dof])

    prescribed_indexes = self.get_prescribed_indexes()
    unprescribed_indexes = self.get_unprescribed_indexes()
    |
    K = full_K[unprescribed_indexes, :][:, unprescribed_indexes]
    M = full_M[unprescribed_indexes, :][:, unprescribed_indexes]
    Kr = full_K[:, prescribed_indexes]
    Mr = full_M[:, prescribed_indexes]

    return K, M, Kr, Mr
```

Method to calculate the global indices of the rows and columns for the assembly process

```
def get_global_indexes(self):

    rows, cols = len(self.elements), DOF_PER_NODE_STRUCTURAL*NODES_PER_ELEMENT
    cols_nodes = self.connectivity_matrix[:,1:].astype(int)
    cols_dofs = cols_nodes.reshape(-1,1)*DOF_PER_NODE_STRUCTURAL + np.arange(6, dtype=int)
    cols_dofs = cols_dofs.reshape(rows, cols)
    J = np.tile(cols_dofs, cols)
    I = cols_dofs.reshape(-1,1)@np.ones((1,cols), dtype=int)

    return I.flatten(), J.flatten()
```

APPENDIX B

Algorithm A

```
from time import time
import numpy as np
from scipy.sparse import csc_matrix, csr_matrix

DOF_PER_NODE = 6
NODE_PER_ELEMENT = 2
DOF_PER_ELEMENT = DOF_PER_NODE*NODE_PER_ELEMENT

number_nodes = int(1e6+1)
nodes = np.arange(number_nodes)
ind = np.arange(number_nodes-1)+1

connectivity = np.array([ind,ind-1,ind]).T
number_elements = connectivity.shape[0]

entries = number_elements*(DOF_PER_ELEMENT**2)
total_dofs = DOF_PER_NODE*number_nodes
local_dof = np.arange(6)

rows = np.zeros(entries)
cols = np.zeros(entries)
data_K = np.zeros(entries)
ind = 0

t0 = time()

global_dof = np.zeros(DOF_PER_ELEMENT, dtype=int)
for start_node, end_node in connectivity[:, 1:]: # sweep elements
    Ke = np.ones((DOF_PER_ELEMENT, DOF_PER_ELEMENT), dtype=float)
    global_dof[:DOF_PER_NODE] = start_node*DOF_PER_NODE + local_dof
    global_dof[DOF_PER_NODE:] = end_node*DOF_PER_NODE + local_dof
    for i, row in enumerate(global_dof): # sweep rows
        for j, col in enumerate(global_dof): # sweep columns
            rows[ind] = row
            cols[ind] = col
            data_K[ind] = Ke[i, j]
            ind += 1
K = csc_matrix((data_k, (rows, cols)), shape=(total_dofs, total_dofs), dtype=float)

dt = time() - t0
```

Algorithm B

```
from time import time
import numpy as np
from scipy.sparse import csc_matrix, csr_matrix

DOF_PER_NODE = 6
NODE_PER_ELEMENT = 2
DOF_PER_ELEMENT = DOF_PER_NODE*NODE_PER_ELEMENT

number_nodes = int(1e6+1)
nodes = np.arange(number_nodes)
ind = np.arange(number_nodes-1)+1

connectivity = np.array([ind,ind-1,ind]).T
number_elements = connectivity.shape[0]

entries = number_elements*(DOF_PER_ELEMENT**2)
total_dofs = DOF_PER_NODE*number_nodes
local_dof = np.arange(6)

rows = np.zeros(entries)
cols = np.zeros(entries)
data_K = np.zeros(entries)
ind = 0

t0 = time()

start, end = 0, 0
aux = np.ones(DOF_PER_ELEMENT)
global_dof = np.zeros(DOF_PER_ELEMENT, dtype=int)
for start_node, end_node in connectivity[:, 1:]: # sweep elements
    Ke = np.ones((DOF_PER_ELEMENT, DOF_PER_ELEMENT), dtype=float)
    global_dof[:DOF_PER_NODE] = start_node*DOF_PER_NODE + local_dof
    global_dof[DOF_PER_NODE:] = end_node*DOF_PER_NODE + local_dof
        for i, row in enumerate(global_dof): # sweep rows
            start = end
            end = start + DOF_PER_ELEMENT
            rows[start:end] = row*aux
            cols[start:end] = global_dof
            data_K[start:end] = Ke[i, :]
K = csc_matrix((data_K, (rows, cols)), shape=(total_dofs, total_dofs), dtype=float)

dt = time() - t0
```

Algorithm C

```
from time import time
import numpy as np
from scipy.sparse import csc_matrix, csr_matrix

DOF_PER_NODE = 6
NODE_PER_ELEMENT = 2
DOF_PER_ELEMENT = DOF_PER_NODE*NODE_PER_ELEMENT

number_nodes = int(1e6+1)
nodes = np.arange(number_nodes)
ind = np.arange(number_nodes-1)+1

connectivity = np.array([ind,ind-1,ind]).T
number_elements = connectivity.shape[0]

entries = number_elements*(DOF_PER_ELEMENT**2)
total_dofs = DOF_PER_NODE*number_nodes
local_dof = np.arange(6)

rows = np.zeros(entries)
cols = np.zeros(entries)
data_K = np.zeros(entries)

t0 = time()

start, end = 0, 0
aux = np.ones((1,DOF_PER_ELEMENT))
global_dof = np.zeros(DOF_PER_ELEMENT, dtype=int)
for start_node, end_node in connectivity[:, 1:]: # sweep elements
  Ke = np.ones((DOF_PER_ELEMENT, DOF_PER_ELEMENT), dtype=float)
  global_dof[:DOF_PER_NODE] = start_node*DOF_PER_NODE + local_dof
  global_dof[DOF_PER_NODE:] = end_node*DOF_PER_NODE + local_dof
  mat_rows = global_dof.reshape(-1,1)@aux
  mat_cols = mat_rows.T
  start = end
  end = start + (DOF_PER_ELEMENT**2)
  rows[start:end] = mat_rows.flatten()
  cols[start:end] = mat_cols.flatten()
  data_K[start:end] = Ke.flatten()
K = csc_matrix((data_k, (rows, cols)), shape=(total_dofs, total_dofs), dtype=float)

dt = time() - t0
```

Algorithm D

```
from time import time
import numpy as np
from scipy.sparse import csc_matrix, csr_matrix

DOF_NODE = 6
NODE_ELEMENT = 2
DOF_ELEMENT = DOF_NODE*NODE_ELEMENT

number_nodes = int(1e6+1)
nodes = np.arange(number_nodes)
ind = np.arange(number_nodes-1)+1

connectivity = np.array([ind,ind-1,ind]).T
number_elements = connectivity.shape[0]

entries = number_elements*(DOF_ELEMENT**2)
total_dofs = DOF_NODE*number_nodes
local_dof = np.arange(6)

rows = np.zeros(entries)
cols = np.zeros(entries)
data_k = np.zeros(entries)

t0 = time()

cols_dofs = connectivity[:,1:].reshape(-1,1)*DOF_NODE + np.arange(6, dtype=int)
cols_dofs = cols_dofs.reshape(number_elements, DOF_ELEMENT)
cols = np.tile(cols_dofs, DOF_ELEMENT).reshape(-1)
rows = (cols_dofs.reshape(-1,1)@np.ones((1,cols), dtype=int)).reshape(-1)
data_k = np.zeros((number_elements,DOF_ELEMENT,DOF_ELEMENT), dtype=float)

for index, (start_node, end_node) in enumerate(connectivity[:, 1:]): # sweep elements
    data_K[index, :, :] = np.ones((DOF_ELEMENT, DOF_ELEMENT), dtype=float)
data_K = data_K.reshape(-1)
K = csc_matrix((data_K, (rows, cols)), shape=(total_dofs, total_dofs), dtype=float)

dt = time() - t0
```