University of Cincinnati, Finite Element Methods

Finite Element Analysis on Bike Components

written by:
Natalie Reed, Piero Paialunga

04/15/2022

Contents

1 Introduction

Finite Element Method (FEM) plays a fundamental role in the modelling and simulation of engineering systems [2]. In fact, when building complex engineering systems, numerous steps of modelling and simulation techniques must be carefully conducted, and FEM provides a very useful set of strategy that can be used in those stages [1]. In this work, the components of a bike will be studied using FEM techniques. Multiple loads will be applied to the bike and its response will be analyzed using the Finite Element Method. In chapter 2 of this work, a brief theoretical background of the FEM method will be given, highlighting all the equations that will be implemented in the following part of the work. In chapter 3, all the codes that have been written and developed will be carefully explained, all the assumptions that have been made will be discussed and the structure of the numerical implementation of the project will be given. In chapter 4, our work will be compared with a hand-solved problem. In this step of validation, the differences between the output produced by our model (implemented following the rule of the previous chapter) and the solution found using pen and paper will be compared. The FEM will be able to give a ready-to-use structure of the problem, given the stiffness and mass matrices. It will be sufficient to apply the loads to the nodes to see the effect on the whole structure we generated. The result of these kind of studies (both dynamic and static) will be given in chapter 4. Finally, the summary of the proposed work and a description of the roles of each member of the group will be given in chapter 5.

2 Theoretical Background

In this chapter, the theoretical background of the Finite Element Method that will be used in this work will be given. The first part of the chapter will explain the Domain discretization of the problem, that will convert the continuous problem of the bike into the discrete one using the nodes. Then, the static problem of a load applied to one or more of the node(s) will be considered, obtaining the equation that needs to be solved to find the relative displacements of the nodes. Finally, a more realistic scenario will be applied, considering the dynamic equation and relative dynamic displacement of the nodes.

2.1 FEM for (Spatial) Frames

The first problem, given a system that needs to be analyzed using FEM, is to discretize the system itself. It means that the solid body needs to be dividend into N_e elements. In particular, each element is formed by connecting together a certain number of nodes. This creates the so called connectivity of the element. All the elements together should be able to recreate the entire domain of the body that is analyzed. In our specific case, the structure is uniformly "sampled" in the sense that we have a sufficiently uniform number of nodes in each part of the studied domain. In this project, each element is modeled to be a frame. Formally, a frame carries the property of the truss element and the beam element. In fact, it is capable of carrying both axial and transverse forces, as well as moments. Our study case is three dimensional. For this reason, spatial frames have been considered.

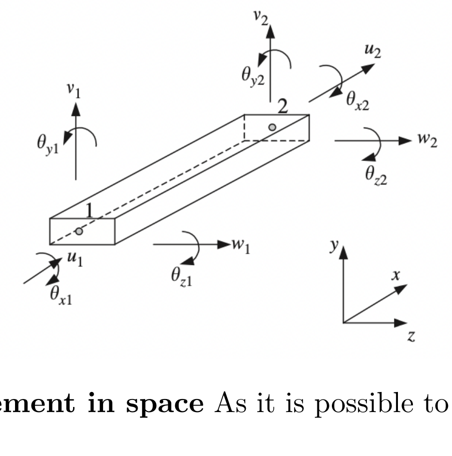

Figure 2.1: **Frame element in space** As it is possible to see, this structure has 12 degrees of freedom.

As it is possible to see in Figure 2.1 we have three translational displacements in the x,y and z directions and three rotations with respect to the x, y and z axes. For this reason, the element displacement vector for a frame element in space which contains two nodes can be written as:

$$\mathbf{d_e} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \\ d_9 \\ d_{10} \\ d_{11} \\ d_{12} \end{bmatrix} = \begin{bmatrix} u_1 \\ v_2 \\ w_1 \\ \theta_{x1} \\ \theta_{y1} \\ \theta_{z1} \\ u_2 \\ v_2 \\ w_2 \\ \theta_{x2} \\ \theta_{y2} \\ \theta_{z2} \end{bmatrix} \tag{2.1}$$

The stiffness matrix and the mass matrix are thus 12×12 matrices. The expressions are reported in Figure 2.2 and Figure 2.3.

$$\mathbf{k}_e = \begin{array}{cccccccccccc}
u_1 & v_1 & w_1 & \theta_{x1} & \theta_{y1} & \theta_{z1} & u_2 & v_2 & w_2 & \theta_{x2} & \theta_{y2} & \theta_{z2} \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\frac{AE}{2a} & 0 & 0 & 0 & 0 & 0 & \frac{-AE}{2a} & 0 & 0 & 0 & 0 & 0 \\
 & \frac{3EI_z}{2a^3} & 0 & 0 & 0 & \frac{3EI_z}{2a^2} & 0 & \frac{-3EI_z}{2a^3} & 0 & 0 & 0 & \frac{3EI_z}{2a^2} \\
 & & \frac{3EI_y}{2a^3} & 0 & \frac{-3EI_y}{2a^2} & 0 & 0 & 0 & \frac{-3EI_y}{2a^3} & 0 & \frac{-3EI_y}{2a^2} & 0 \\
 & & & \frac{GJ}{2a} & 0 & 0 & 0 & 0 & 0 & \frac{-GJ}{2a} & 0 & 0 \\
 & & & & \frac{2EI_y}{a} & 0 & 0 & 0 & \frac{3EI_y}{2a^2} & 0 & \frac{EI_y}{a} & 0 \\
 & & & & & \frac{2EI_z}{a} & 0 & \frac{-3EI_z}{2a^2} & 0 & 0 & 0 & \frac{EI_z}{a} \\
 & & & & & & \frac{AE}{2a} & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & \frac{3EI_z}{2a^3} & 0 & 0 & 0 & \frac{-3EI_z}{2a^2} \\
 & & sy. & & & & & & \frac{3EI_y}{2a^3} & 0 & \frac{3EI_y}{2a^2} & 0 \\
 & & & & & & & & & \frac{GJ}{2a} & 0 & 0 \\
 & & & & & & & & & & \frac{2EI_y}{a} & 0 \\
 & & & & & & & & & & & \frac{2EI_z}{a}
\end{array}$$

Figure 2.2: **Stiffness matrix for a 3d frame element**.

$$
\mathbf{m}_e = \frac{\rho A a}{105}
\begin{bmatrix}
70 & 0 & 0 & 0 & 0 & 0 & 35 & 0 & 0 & 0 & 0 & 0 \\
 & 78 & 0 & 0 & 0 & 22a & 0 & 27 & 0 & 0 & 0 & -13a \\
 & & 78 & 0 & -22a & 0 & 0 & 0 & 27 & 0 & 13a & 0 \\
 & & & 70r_x^2 & 0 & 0 & 0 & 0 & 0 & -35r_x^2 & 0 & 0 \\
 & & & & 8a^2 & 0 & 0 & 0 & -13a & 0 & -6a^2 & 0 \\
 & & & & & 8a^2 & 0 & 13a & 0 & 0 & 0 & -6a^2 \\
 & & & & & & 70 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & 78 & 0 & 0 & 0 & -22a \\
 & & & & & & & & 78 & 0 & 22a & 0 \\
 & & & sy. & & & & & & 70r_x^2 & 0 & 0 \\
 & & & & & & & & & & 8a^2 & 0 \\
 & & & & & & & & & & & 8a^2
\end{bmatrix}
$$

Figure 2.3: **Mass matrix for a 3d frame element**.

Each of these matrices has to be coordinate transformed into the global coordinate system. In order to do that, a **T** matrix has to be defined. This matrix is a diagonal 12×12 matrix, that can be summarized in the following expressions:

$$
\mathbf{T} = \begin{bmatrix} \mathbf{T_3} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{T_3} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{T_3} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{T_3} \end{bmatrix} \tag{2.2}
$$

Where:

$$
\mathbf{0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{2.3}
$$

And:

$$
\mathbf{T_3} = \begin{bmatrix} l_x & m_x & n_x \\ l_y & m_y & n_y \\ l_z & m_z & n_z \end{bmatrix} \tag{2.4}
$$

And:

$$
\begin{aligned}
l_x &= \cos(x, X), & m_x &= \cos(x, Y), & n_x &= \cos(x, Z) \\
l_y &= \cos(y, X), & m_y &= \cos(y, Y), & n_y &= \cos(y, Z) \\
l_z &= \cos(z, X), & m_z &= \cos(z, Y), & n_z &= \cos(z, Z)
\end{aligned} \tag{2.5}
$$

In the above expressions x, y and z are local coordinates and X, Y, Z are the global coordinates. Given the transformation matrix, the elemental mass and stiffness matrices in the global coordinate system can be found using the following equations.

$$
\mathbf{K_e} = \mathbf{T^T k_e T} \tag{2.6}
$$

And:

$$\mathbf{M_e} = \mathbf{T^T m_e T} \tag{2.7}$$

The transformed elemental stiffness and mass matrices will be assembled together into global stiffness and mass matrices as will be shown in chapter 3. Assembly of these global matrices defines the relationships between adjacent elements which share nodes. With 6 degrees of freedom for each node, these global matrices are $6N \times 6N$ where N = number of nodes. This is true both for the $\mathbf{K_e}$ matrix and the $\mathbf{M_e}$ one, which are the assembled version of the small $\mathbf{k_e}$ and $\mathbf{m_e}$ matrices. The $\mathbf{D_e}$ vector will be the global vector of displacement, and its dimension will be $6N \times 1$.

2.2 Static Analysis

In this work, a static analysis has been considered on the bike structure. Given the stiffness matrix $\mathbf{K_e}$ and the displacement vector $\mathbf{D_e}$, we will end up having:

$$\mathbf{K_e D_e} = \mathbf{F_e} \tag{2.8}$$

As the $\mathbf{K_e}$ matrix is defined by the properties of the elements (see Figure 2.2), we considered it to be known. Then, different loads can be applied to different nodes, thus defining the $\mathbf{F_e}$ vector has well. Inverting equation 2.8 we are thus able to find the displacement vector $\mathbf{D_e}$. When the number of nodes N is large ($\approx 10^3$) and the dimension of the stiffness matrix is large as well ($\approx 10^4 \times 10^4$), this equation is usually solved using an iterative method.

2.3 Dynamic Analysis

The main difference between the static and dynamic analysis is that, instead of solving a set of linear equations, it is now necessary to solve the following differential equation:

$$\mathbf{M_e \ddot{D}_e} + \mathbf{K_e D_e} = \mathbf{F_e} \tag{2.9}$$

Building a solver for this set of differential equations is considerably more complex than the previous problem and it is best to be done using low level computational languages like C++ or Fortran. As it goes beyond the scope of this project, the dynamic analysis has been set up and prepared to be use, except for the final differential equation solver which is left to be implemented.

3 Numerical Implementation

In this chapter, the numerical implementation of the proposed work is explained in detail. Starting from the creation of the node structure and ending in the computation of the displacement and force vectors, all the numerical details of our work will be discussed. The codes are displayed in chapter 6 of this work and documented in the GitHub page of the project as well. All the information about the bike design and its mechanical properties has been extracted from a bike model that we found in a Free CAD database. In this work, we assumed that the bike is made of titanium with material properties pulled from an online source (Material Properties).

3.1 FEM Bike Design

The bike is made up of 8 macro-elements. Each macro-element is defined by two connected nodes. Other than the two connected nodes, each macro-element has been divided into $N = 20$ smaller elements. The 8 macro-elements have different mechanical properties, while each one of the smaller elements has the same properties as the macro-element to which they belong. The two connected nodes that build the macro element are reported in Table 1.

Table 1: Eight macro-element nodes of the bike.

Element	p1	x1	y1	z1	p2	x2	y2	z2
1	1	-0.1266	0.4565	0	2	0.3893	0.5501	0
2	2	0.3893	0.5501	0	3	0.4102	0.4876	0
3	3	0.4102	0.4876	0	4	0	0	0
4	4	0	0	0	1	-0.1266	0.4565	0
5	1	-0.1266	0.4565	0	5	-0.4244	0.0687	0.075
6	1	-0.1266	0.4565	0	6	-0.4244	0.0687	-0.075
7	4	0	0	0	5	-0.4244	0.0687	0.075
8	4	0	0	0	6	-0.4244	0.0687	-0.075

The mechanical properties of each of the macro-element are reported in Table 2, while all the macro-elements have the following constants:

- **Young's Modulus** $E = 1.20 \times 10^{11} N/m^2$
- **Poisson Ratio** $\nu = 0.361$
- **Shear Modulus** $G = 4.41 \times 10^{10} N/m^2$
- **Density** $\rho = 4510 kg/m^3$

Table 2: Properties of the 8 macro-elements.

Ri	Ro	A	Ix	Iy	Iz	J	t
13.47	14.18	61.64	5891.79	5891.79	11783.57	11783.57	0.71
17.00	19.00	226.19	36756.63	36756.63	73513.27	73513.27	2.00
17.86	18.59	83.09	13806.83	13806.83	27613.67	27613.67	0.73
13.60	14.60	88.59	8817.65	8817.65	17635.31	17635.31	1.00
6.51	7.23	31.01	733.52	733.52	1467.03	1467.03	0.72
6.51	7.23	31.01	733.52	733.52	1467.03	1467.03	0.72
8.02	8.74	37.78	1330.08	1330.08	2660.16	2660.16	0.72
8.02	8.74	37.78	1330.08	1330.08	2660.16	2660.16	0.72

Where the names in the table have the following meaning:

- $t[mm]$ is the thickness of the element
- R_i and R_o are the inner and outer radius $[mm]$
- I_x, I_y and I_z are the second moment of inertia $[m^4]$
- A is the cross-sectional area $[m^2]$
- J is the polar moment of inertia $[m^4]$

Each macro-element has been "filled" with N smaller elements from node 1 to node 2. This has been done using the `frameForm` function reported in chapter 6.1. This function is parametric meaning the number of elements in each macro-element can be readily changed. The result of this process is shown in Figure 3.1. As it is possible to see, there are 8 macro-elements, but each macro-element is made with couples of nodes which construct smaller (micro) elements. We will just refer them as "element" from now on.

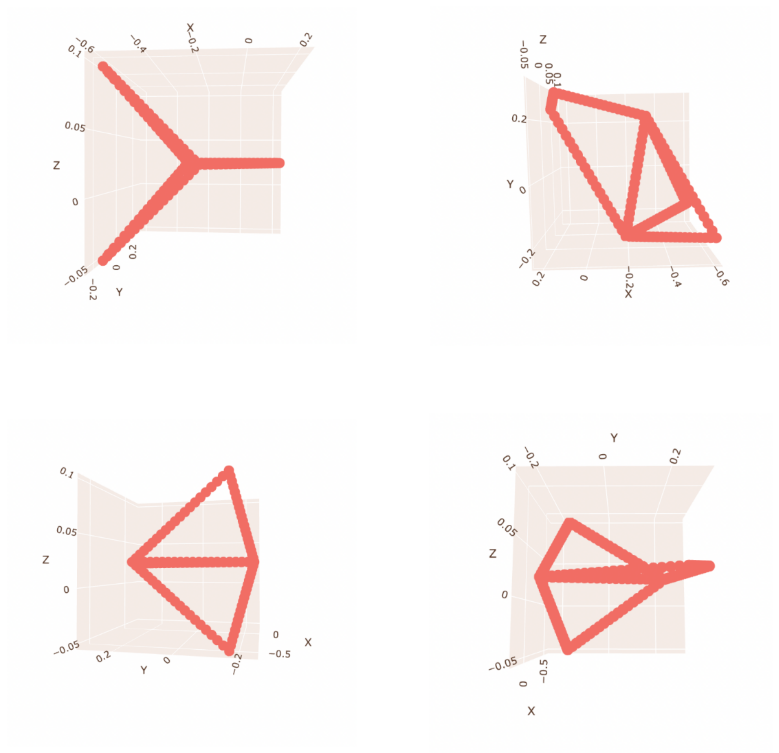

Figure 3.1: **Four different angles of the structure made using the nodes**. As it is possible to see, there are 8 macro-elements, but each macro-element is made with couples of nodes which construct smaller (micro) elements. This image has been generated using the Plotly express Python Library.

3.2 Mass, Stiffness, Transformation Matrix

The second output of this function is a dataframe where all the mechanical properties of all the elements are listed. Again, changing the information of this structure, we can analyze different kind of materials and get different results. Using all the information that is listed inside this dataframe and the `massMat` and `kMat` functions shown in chapter 6.2, the stiffness matrix and mass matrix have been created for each element. After this creation, each matrix has been transformed using the transformation matrix as shown in equations 2.6 and 2.7. After these matrix transformations, all elemental mass and k matrices share a common coordinate system and can therefore be assembled using the `glob` function. As it is possible to see from the expression in Figure 2.2 and Figure 2.3 the matrices are supposed to have multiple zero entries. This can be seen in the heatmap of the two matrices of Figure 3.2 as well.

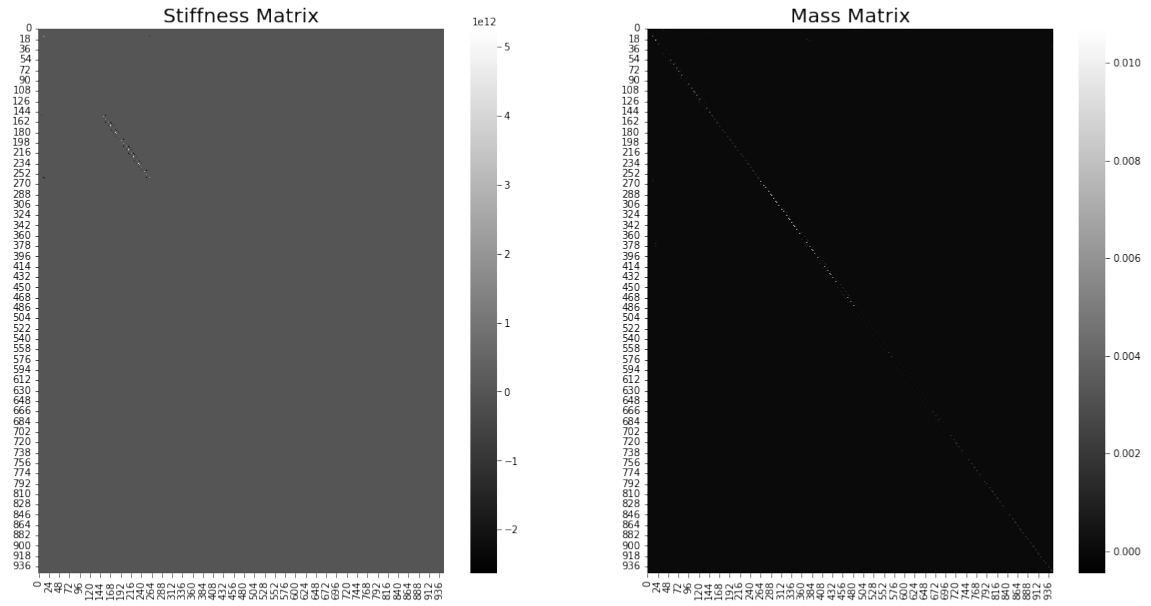

Figure 3.2: **Mass matrix's heatmap (left) and K matrix's heatmap (right).** Plots of the mass matrix (right) and stiffness matrix (left) using heatmaps.

The 12×12 transformation matrix heatmap is shown in Figure 3.3 as well.

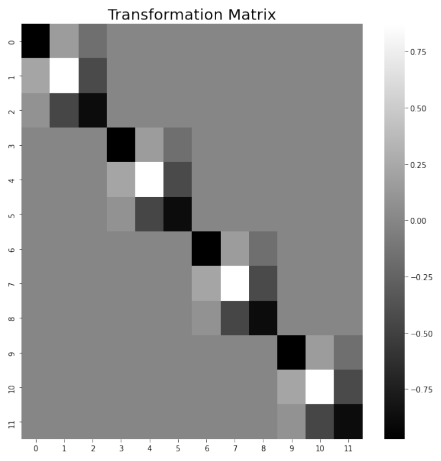

Figure 3.3: **Transformation matrix's heatmap.**

3.3 Boundary Conditions

In our case, a set of three nodes, that are the front wheel and the split in the back related to the back wheel are considered to be fixed in all three directions. The three nodes are highlighted in Figure 3.4. The input of

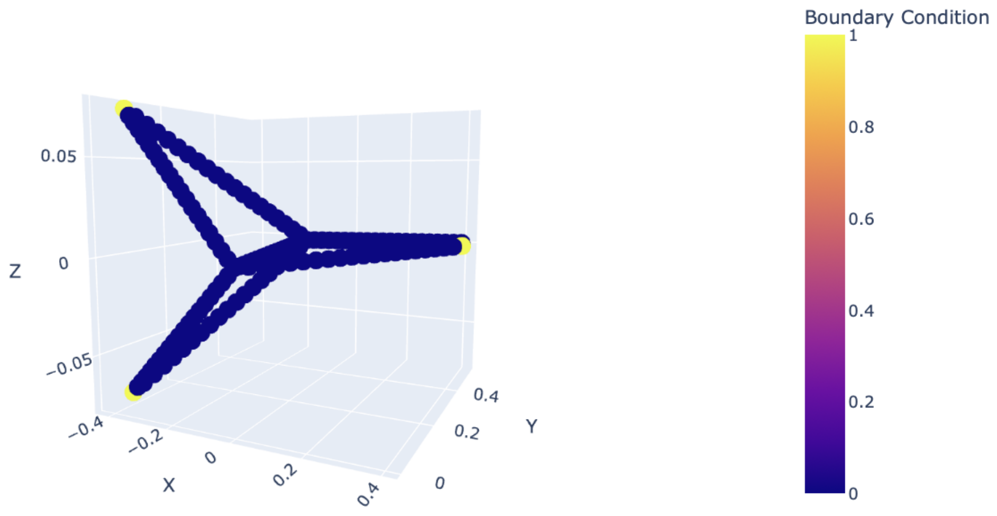

Figure 3.4: **Boundary condition summary.** Three nodes (highlighted as yellow nodes) are considered to be fixed (no displacement).

boundary conditions such as these has been made using tunable parameters. This means that the `.csv` file can be modified and different (or more) nodes can be considered to be constrained. A function, named `findclose` will find the node that is closest to the point that has been considered. For example, if in the `.csv` file, the point x=0, y=0 and z=0 is set to have displacement zero in the x and y direction, `findclose` will find the node that is closest to x=0, y=0 and z=0 and set the conditions on that node. All these operations are done in the `appBound` function.

3.4 Force Vector

The force vector has been again read from a `.csv` file. In this case, again, we apply the `findclose` function to find the closest node to the points we are considering. The `.csv` file is processed by the so called `forceVector` function. Again, this setup is parametric. This means that multiple force scenarios (virtually infinite) can be considered. Forces can be applied to multiple nodes at once, with different orientations. Our study case is made by two different static problems and a dynamic one. The first one simulates a person sitting on the bike seat. In the node where the force is applied, the direction of the force is only y. In the second case, the person is still sitting on the bike seat, but in another node (in the front part of the bike) we consider another force with application in both the x and y direction. This simulates the starting position of a biker, which applies some pressure in the x direction as well in the front of the bike. The summary of the two cases is reported in Figure 3.5

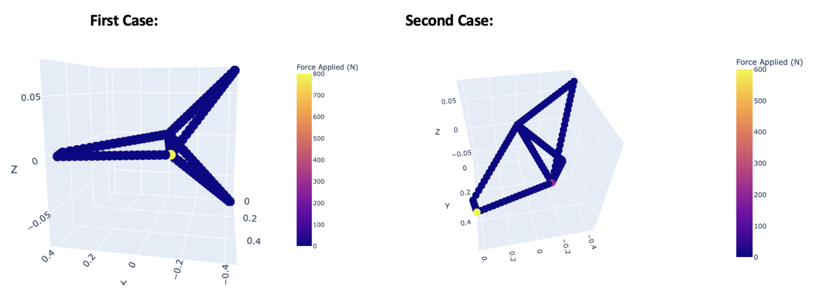

Figure 3.5: **First and Second Case forces shown.** In the first case, only a y direction forces is shown in a specific node. On the other hand, in the second case another force is added in a front node, on the x direction.

In the first case an ≈ 80 kg person ($800N$) is supposed to sit on the bike seat. In the second case, the same force is divided into 200 N in the previous node and 600 N in the front node. This simulates the fact that the rider is pushing on the handle bars down and forward away from their body with most of their weight. The summary of these two forces scenarios are considered in table 3 and 4. X,Y, and Z are the node location, F(N) is the force in Netwon, while xV,yV and zV represent the direction of the force.

Table 3: Force applied in the first case.

X	Y	Z	F(N)	xV	yV	zV
-0.1266	0.4565	0	800	0.000000	-1.000000	0.0

Table 4: Force applied in the second case.

X	Y	Z	F(N)	xV	yV	zV
-0.1266	0.4565	0	200	0.000000	-1.000000	0.0
0.3893	0.5501	0	600	-0.984808	0.173648	0.0

3.5 Numerical Solver

Given the stiffness matrix, the boundaries condition and the force input vector, we are able to numerically solve the system of equations reported in equation 2.8. Similarly to what we have done in the homeworks so far, the following process has been established:

1. Apply the boundary conditions by eliminating the rows and columns corresponding to the appropriate nodes and degrees of freedom.
2. Compute the nodal displacements by solving the linear system with the reduced stiffness matrix and force vector.
3. Add back 0 values to the nodal displacement vector in the rows which were removed during application of the boundary conditions.
4. We used the full stiffness matrix and the modified nodal displacement vector to calculate the force vector.

This method has been repeated for the two force vectors we have studied. Nonetheless, as it has been already stated, it is sufficient to change the number of nodes and change the `.csv` file to analyze different scenarios and obtain different results. In general, given k different force vectors, the numerical solver we implemented will generate k different `.csv` file, whose name will be `staticRes`$+i+$`.csv` where $1 \leq i \leq k$. Each file will be made by N=number of nodes rows. Each row will represent a node and will be accompanied by 16 columns of results:

- Index of the node (column 1)
- x, y and z location of the nodes (columns 2 thru 4)
- Nodal Displacement for the 6 degrees of freedom (columns 5 thru 10)

- Nodal forces in the x, y and z directions (columns 11 thru 13)
- Nodal Moments about the x, y and z directions (columns 14 thru 16)

This is considered to be the real output of the `BikeMain.py` file.

4 Results

4.1 Verification

It is important to note that a verification of our FEM implementation was performed prior to the calculation of the Bike problem. The same procedure as described above was followed to set up the problem given in Homework 5. The output of our FEM program for this problem is shown below in 5 and was found to match the hand calculated results from Homework 5. The structure studied in Homework 5 is reported in Figure 4.1.

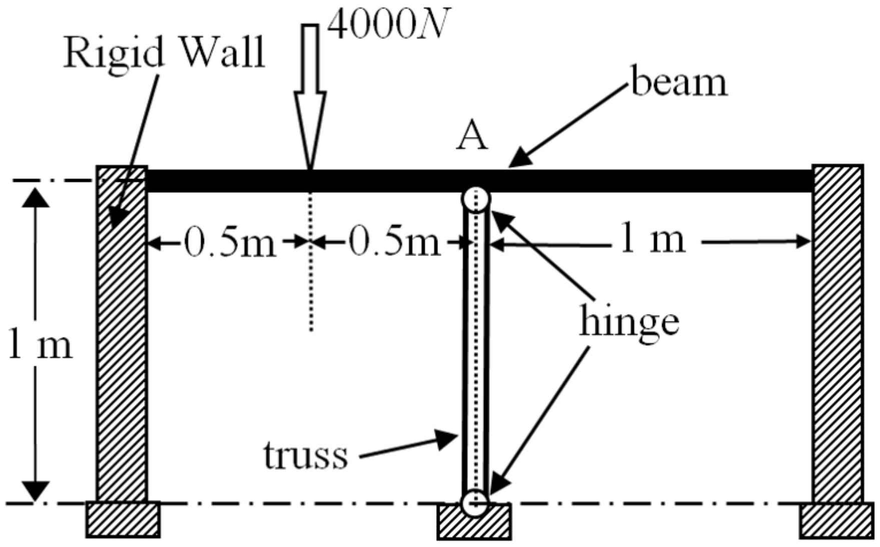

Figure 4.1: **Homework 5 structure**

Table 5: Displacement for the 5 different nodes of Homework 5.

	X	Y	Z	DOF 1 Displacement	DOF 2 Displacement	DOF 3 Displacement	DOF 4 Displacement	DOF 5 Displacement	DOF 6 Displacement
0	-1	1	0	0	0	0	0	2863.543372	869.2716858
1	0	1	0	0	-2.56E-05	3.93E-05	0	2.27E-13	-1.14E-13
2	1	1	0	0	0	0	0	113.5433715	-119.2716858
3	0	0	0	0	0	0	0	1022.913257	0
4	-0.5	1	0	0	-3.08E-05	-4.82E-05	0	-4000	0

4.2 Displacements

An example of the displacement vector for 6 different nodes in the first scenario (one force only) is reported in table 6.

Table 6: Displacement of 6 random input nodes of the bike structure.

	X	Y	Z	DOF 1 Displacement	DOF 2 Displacement	DOF 3 Displacement	DOF 4 Displacement	DOF 5 Displacement	DOF 6 Displacement
128	-0.190980	0.030915	0.03375	0.000007	-2.676025e-05	1.647138e-06	0.000023	0.000015	0.000149
27	0.392435	0.540725	0.00000	0.000002	5.289828e-07	-4.134485e-07	0.000009	0.000008	0.000047
120	-0.021220	0.003435	0.00375	0.000012	-4.634244e-05	1.806530e-06	-0.000006	-0.000006	0.000032
125	-0.127320	0.020610	0.02250	0.000008	-3.637562e-05	2.201029e-06	0.000017	0.000010	0.000127
4	-0.424400	0.068700	0.07500	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000

In figure 4.2, the nodes are plotted with different colors with respect to the displacement in the three directions for the one force scenario (right figure 3.5).

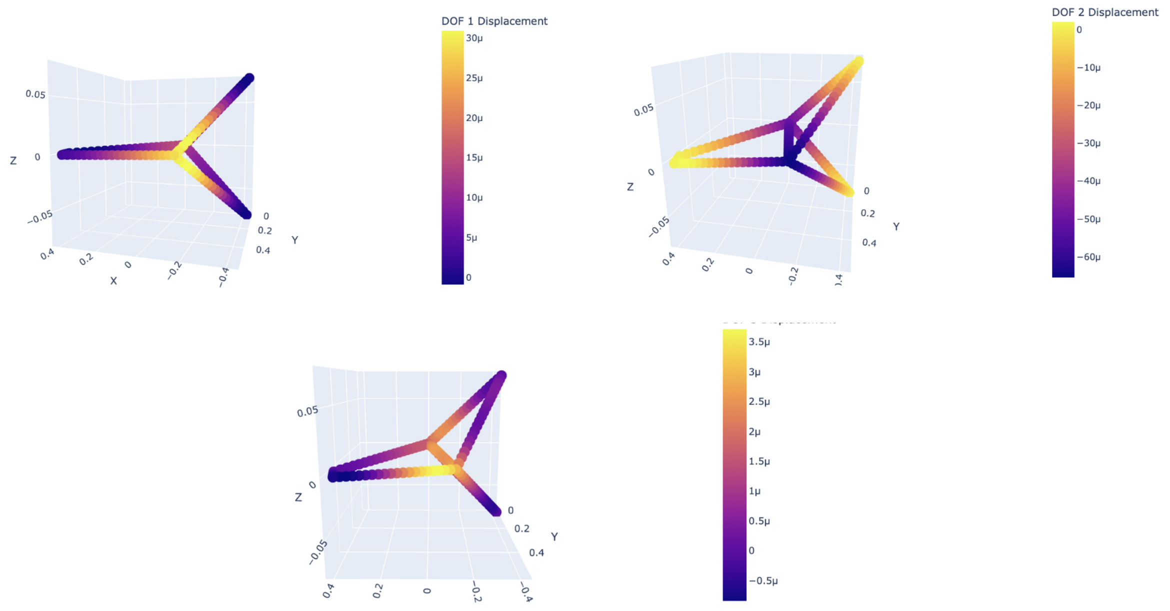

Figure 4.2: **Displacement in the first case force.**

As it is possible to see, the displacement is negative only for the y axis, while it is positive only for the x axis. Nonetheless, the front of the bike stands still in both the y direction (null displacement). Even in the x case the displacement slowly fades to 0. The z direction shows that the front of the bike is moving in the negative direction, the zone where the force is applied is moving towards the positive direction and the extreme parts are moving in the negative directions as well. In all the three displacement, the same

"fading" behaviour from the middle towards the extreme parts of the bike can be seen. Both the effect of the first (y direction, back) and the second (x direction, front) can be seen.
In figure 4.3, the nodes are plotted with different colors with respect to the displacement in the three directions for the two forces (right figure 3.5 are applied). As it is possible to see, the displacements are completely different

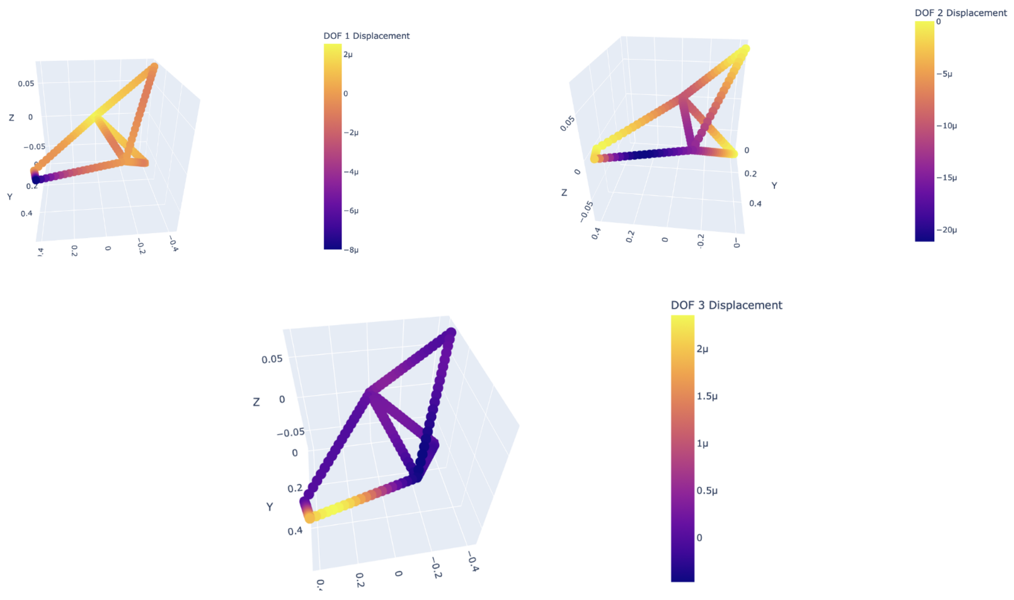

Figure 4.3: **Displacement in the first case force.**

in this scenario. The most predictable one is the x axis displacement, which shows a negative displacement on the front node. To balance this direction, almost the other nodes have positive displacement. In particular, the displacement becomes larger when Y and X decreases. The displacement in the y direction fades to 0 in the extreme parts, while it is negative in one "arm" of the bike and in the central element. The displacement is almost everywhere null in the z direction except for the zone close to the "front" force, where the displacement is slightly positive. It is possible to notice that in both the first and second case force, the displacement in the boundary conditions is identically 0, which is what we expect from our theoretical background.

5 Conclusions

In this project, a bike body has been studied using a Finite Element Method (FEM) simulation.
This work has been made by Natalie Reed an Piero Paialunga, which have worked together in the same office and room for the entire project. This means that all the steps of the project, from the code developing to the report writing, have been made together in the same room.
Starting, from scratch and having a bike model as a guideline, we created the bike structure given a certain number of node $N = 158$. From this simple structure, and having the model of the mechanical properties of the bike, we created the stiffness and mass matrix. Moreover, the transformation matrix (to change the coordinates of the stiffness and mass matrix) has been created as well. Given that we consider the two wheels of the bike to be steady, we applied boundary conditions there, considering 0 displacement on the three directions. Two different static scenario have been considered. The first one is a person sitting on the bike, the second one is a person sitting on the same point of the bike but pushing the bike forward on the x directions. Given these two forces, and applying a numerical solver, we found and displayed the displacement on the entire bike and the resulting internal forces. The dynamic problem has been started as well, but as it required a non trivial numerical implementation given the necessity of building a differential equation solver, it has not been fully implemented.
The whole code has been built from scratch using Python. The entire structure of the code is parametric. This means that virtually infinite scenarios can be considered by just changing the number of nodes, the mechanical properties, the boundary conditions and the force vectors in the correspondent input `.csv` files. The code has been proven to be correct by being applied in a controlled situation where we knew the exact displacement results. In our study case, the results have furnished insightful considerations about the displacement on the entire bike structure. An interesting development of this project would be to compare this result with the ones of an FEM software like ABACUS or to implement a system of differential equation solver to solve the dynamic scenario that has been here prepared to be solved. The codes are displayed in chapter 6 of this work and reported in the GitHub page.

6 Codes

6.1 Node Creation Function

```python
def frameForm(propFile):
    baseDF = pd.read_csv(propFile, header=0, skiprows=[1])
    print(baseDF)
    nPairs = len(baseDF)
    nodeDF = pd.DataFrame(columns=['X', 'Y', 'Z'])
    elemDF = pd.DataFrame(columns=['Node 1', 'Node 2', 'A', 'E', 'G', 'rho', 'a'
    eInd = 0
    print(baseDF.columns)
    nInd = int(baseDF[['p1', 'p2']].to_numpy().max())
    for ind in range(nPairs):
        ind1 = baseDF['p1'].iloc[ind] - 1
        ind2 = baseDF['p2'].iloc[ind] - 1
        eNum = baseDF['numElem'].iloc[ind]
        temp1 = baseDF[['x1', 'y1', 'z1']].iloc[ind].to_numpy()
        temp2 = baseDF[['x2', 'y2', 'z2']].iloc[ind].to_numpy()
        unitV = temp2 - temp1
        distV = np.sqrt(np.dot(unitV, unitV))
        unitV = unitV / distV
        dD = distV / eNum
        conLst = baseDF[['A', 'E', 'G', 'rho', 'Ix', 'Iy', 'Iz', 'J']].iloc[ind]
        conLst.insert(4, dD/2)
        conLst += list(unitV)
        nodeDF.loc[ind1] = temp1
        nodeDF.loc[ind2] = temp2
        for node in range(eNum):
            if node == 0:
                elemDF.loc[eInd] = [ind1, nInd]+conLst
                eInd += 1
            else:
                if node == eNum - 1:
                    elemDF.loc[eInd] = [nInd, ind2]+conLst
                    eInd += 1
                else:
                    elemDF.loc[eInd] = [nInd, nInd + 1]+conLst
                    eInd += 1
                nodeDF.loc[nInd] = temp1 + dD * unitV * node
                nInd += 1
    return nodeDF.sort_index(), elemDF.sort_index()
```

6.2 Mass and Stiffness Matrix Functions

```
def massMat(rho, A, a, J):
    rV = J/A
    a2 = a**2
    mass = np.zeros((12, 12), dtype=float)
    mass[np.array([0, 6]), np.array([0, 6])] += 70
    mass[np.array([1, 2, 7, 8]), np.array([1, 2, 7, 8])] += 78
    mass[np.array([3, 9]), np.array([3, 9])] += 70*rV
    mass[np.array([4, 5, 10, 11]), np.array([4, 5, 10, 11])] += 8*a2
    mass[np.array([0]), np.array([6])] += 35
    mass[np.array([1, 8, 2, 7]), np.array([5, 10, 4, 11])] += 22*a
    mass[np.array([1, 2]), np.array([7, 8])] += 27
    mass[np.array([2, 5, 1, 4]), np.array([10, 7, 11, 8])] += 13*a
    mass[np.array([3]), np.array([9])] += -35*rV
    mass[np.array([4, 5]), np.array([10, 11])] += -6*a2
    mass[np.array([1, 4, 2, 7]), np.array([11, 8, 4, 11])] *= -1
    mass *= rho*A*a/105
    mass = np.tril(mass.T) + np.triu(mass, 1)
    return mass

def kMat(A, E, G, a, Iy, Iz, J):
    k = np.zeros((12, 12), dtype=float)
    # Set numerators
    k[np.array([0, 6, 0]), np.array([0, 6, 6])] += A*E
    k[np.array([1, 7, 5, 1, 7, 1, 1]), np.array([1, 7, 7, 5, 11, 7, 11])] += 3*E
    k[np.array([2, 8, 2, 8, 4, 2, 2]), np.array([2, 8, 4, 10, 8, 8, 10])] += 3*E
    k[np.array([3, 9, 3]), np.array([3, 9, 9])] += G*J
    k[np.array([4, 10, 4]), np.array([4, 10, 10])] += 2*E*Iy
    k[np.array([5, 11, 5]), np.array([5, 11, 11])] += 2*E*Iz
    # Set denominators
    k[np.array([0, 3, 6, 9, 0, 3, 4, 5]), np.array([0, 3, 6, 9, 6, 9, 10, 11])]
    k[np.array([2, 5, 8, 1, 4, 7, 2, 1]), np.array([4, 7, 10, 5, 8, 11, 10, 11])
    k[np.array([1, 2, 7, 8, 1, 2]), np.array([1, 2, 7, 8, 7, 8])] *= 1/(2*a**3)
    k[np.array([4, 5, 10, 11]), np.array([4, 5, 10, 11])] *= 1/a
    # Set negative values
    k[np.array([2, 5, 7, 0, 1, 2, 3, 2]), np.array([4, 7, 11, 6, 7, 8, 9, 10])]
    k = np.tril(k.T) + np.triu(k, 1)
    return k
```

6.3 Assembly and Transformation Function

```
def glob(nodeDF, elemDF, globC):
    nNum = len(nodeDF)
    massG = np.zeros((nNum*6, nNum*6), dtype=float)
    kG = np.zeros((nNum*6, nNum*6), dtype=float)
    for index, row in elemDF.iterrows():
        # Calculate mass and stiffness matrices
        tempM = massMat(row['rho'], row['A'], row['a'], row['J'])
        tempK = kMat(row['A'], row['E'], row['G'], row['a'], row['Iy'], row['Iz'
        # Coordinate transformation
        tMat = transM(row[['xV', 'yV', 'zV']].to_numpy(), globC)
        tMatT = np.transpose(tMat)
        tempM = np.matmul(np.matmul(tMatT, tempM), tMat)
        tempK = np.matmul(np.matmul(tMatT, tempK), tMat)
        node1 = int(row['Node 1'])
        node2 = int(row['Node 2'])
        #Mass Matrix Assembly
        massG[6*node1:6*(node1+1), 6*node1:6*(node1+1)] += tempM[:6, :6]
        massG[6*node2:6*(node2+1), 6*node2:6*(node2+1)] += tempM[6:, 6:]
        massG[6*node1:6*(node1+1), 6*node2:6*(node2+1)] += tempM[:6, 6:]
        massG[6*node2:6*(node2+1), 6*node1:6*(node1+1)] += tempM[6:, :6]
        #Stiffness Matrix Assembly
        kG[6 * node1:6 * (node1 + 1), 6 * node1:6 * (node1 + 1)] += tempK[:6, :6
        kG[6 * node2:6 * (node2 + 1), 6 * node2:6 * (node2 + 1)] += tempK[6:, 6:
        kG[6 * node1:6 * (node1 + 1), 6 * node2:6 * (node2 + 1)] += tempK[:6, 6:
        kG[6 * node2:6 * (node2 + 1), 6 * node1:6 * (node1 + 1)] += tempK[6:, :6
    return massG, kG
```

6.4 Boundary Condition Function

```
def findClose(pt, nodeDF):
    dist = nodeDF.apply(lambda row: np.abs(np.linalg.norm(row[['X', 'Y', 'Z']].t
    ind = dist.idxmin()
    return ind, nodeDF.iloc[ind]
def appBound(boundFile, nodeDF):
    boundDF = pd.read_csv(boundFile, header=0, skiprows=[1])
    indLst = []
    for index, row in boundDF.iterrows():
        # Locate closest point
        ind, _ = findClose(row[['X', 'Y', 'Z']].to_numpy(), nodeDF)
```

```python
        # Define fixed degrees of freedom
        lInd = np.where(row[['DOF 1', 'DOF 2', 'DOF 3', 'DOF 4', 'DOF 5', 'DOF 6
        if len(lInd) > 0:
            indLst.extend(6*ind+lInd)
    indLst.sort()
    return indLst
```

6.5 Force Vector Function

```python
def forceVect(forceFile, nodeDF):
    forceDF = pd.read_csv(forceFile, header=0, skiprows=[1])
    staticFV = np.zeros((6*len(nodeDF),), dtype=float)
    dynamicFunc = {}
    for index, row in forceDF.iterrows():
        # Calculated unit direction vector
        temp = row[['xV', 'yV', 'zV']].to_numpy()
        dV = np.sqrt(np.dot(temp, temp))
        temp = temp/dV
        # Locate closest point
        ind, _ = findClose(row[['X', 'Y', 'Z']].to_numpy(), nodeDF)
        if row['freq'] > 0:
            for dirI in range(3):
                dynamicFunc[6*ind+dirI] = lamDef(row['mag'], temp[dirI])
        else:
            staticFV[6*ind+[0, 1, 2]] = row['mag']*temp
    return staticFV, dynamicFunc
```

6.6 Numerical Solver (BikeMain.py)

```python
globC = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])

nodeDF, elemDF = frameForm(frameFile)
nodeDF.to_csv("nodes.csv")
elemDF.to_csv("elements.csv")
mM, kM = glob(nodeDF, elemDF, globC)

boundInd = appBound(boundFile, nodeDF)
print(boundInd)

solI = 1
for file in forceFile:
    staticFV, dynamicFunc = forceVect(file, nodeDF)
    if bool(dynamicFunc):
        print("Dynamic Case")
    else:
        print("Static Case")
        nDis, nFor = staticSolve(kM, staticFV, boundInd)
        nd_df = pd.DataFrame(data=np.array_split(nDis, len(nodeDF)), columns=["D
        nf_df = pd.DataFrame(data=np.array_split(nFor, len(nodeDF)), columns=["D
        strT = "staticRes"+str(solI)+".csv"
        pd.concat([nodeDF, nd_df, nf_df], axis=1).to_csv(strT)
        solI += 1
```

References

[1] Gui-Rong Liu and Siu Sin Quek. *The finite element method: a practical course.* Butterworth-Heinemann, 2013.

[2] Gilbert Strang and George J Fix. "An analysis of the finite element method(Book- An analysis of the finite element method.)" In: *Englewood Cliffs, N. J., Prentice-Hall, Inc., 1973. 318 p* (1973).