# Building a FEM tool from scratch

April 2020, David Braun TUM

The **Finite Element Method** (FEM) is the most common modeling and simulation technique for both 2D and 3D solid bodies. It is a **numerical method** for solving complex problems within mathematical physics and engineering.

Real world problems are usually too complex to be solved analytically. FEM provides an **approximate solution to partial differential equations** by subdividing large systems into smaller, simpler parts that are called finite elements.

Ultimately, the finite element method formulation of a boundary value problem will result in a system of algebraic equations, that can be solved by simple system of linear equations.
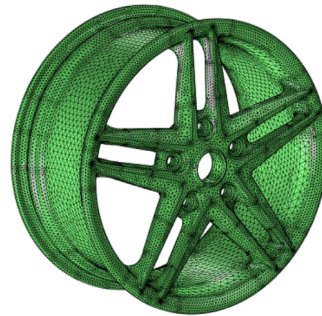
Now, without further ado, let us build a simple FEM application from scratch using MATLAB!

What should our tool do?

Although some of us are true **TUM rocket scientists**, we want to keep it simple. Hence we will consider an example, many of us are familiar with:
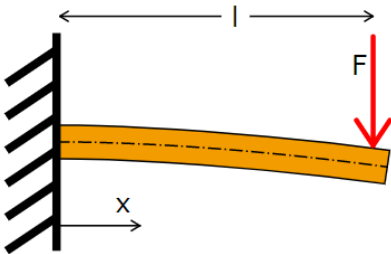
Lets start by analyzing a **clamped beam** in two dimensions. We are interested in the deformation, strain and stress distribution which results after applying a point force F at the tip.

How do we tackle this problem using FEM?

Before continouing onwards, I would like to clearify that an elaborate discussion of FEM would clearly exceed the scope of this project. Interested students are referred to the lecture of Prof. Wall (http://www.lnm.mw.tum.de), where a detailed introduction to the finite element method is provided.

First of all we have to model our beam. For typical problems in structural mechanics, the continous geometry is far to complex to be solved analytically! That's why we break down the geometry into many little subdomains called **Finite Elements**. With the use of simple shape functions we will describe the displacement of each of our finite elements. At this point we will be considering each element individually. We will be able to compute an elementwise stiffness matrix k and an elementwise loading vector s0 for each of our elements by evaluating elementwise defined integrals. Thereafter, all element quantities are assembled to a global stiffness matrix **K** and a global load vector **F**. Visually speaking, two neighboring elements both contribute to the same nodal displacement value. At this point we are able to directly enforce boundary conditions. In our case, we will be accounting for the support of the beam, as well as, applying the point load.

Finally, a discrete gloabl system of linear equations results: **KD=F**

With the help of MATLAB, we will solve for the vector **D=K\F.** Now we have the solution **D** at hand, which consits of node values for the displacement field. Typically, we are also interested in the resulting stresses.

To calculate the approximated stresses we will perform a post-processing step. As always, results are only worth much if they allow for evaluation. That is, we want to visualize both, the resulting deformation and the resulting stress distribution.

1. (Continous) 2D-geometry

We start of by creating a simple 2D rectangular geometry that models our beam.
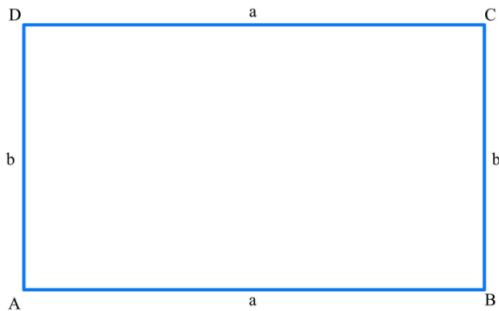
Remark that we are **not yet** concerned about any boundary conditions or support structures.

```matlab
% We want to clear the workspace & the command window
clear,clc
% besides that we also close all figures.
close all

% Throughout this project, we will need a lot
% of differen functionalites. To keep our project
% modular and help debugging we will be using
% multiple utility functions.

% This livescript, main.mlx, will serve as the
% entry point to our FEM tool. From here we will
% call several functions which are all located in
% a subfolder called "utility".

% Let's add the (empty) utility folder to the MATLAB
```

```matlab
% search path.
addpath('utility');

% Lets create a rectangular geometry:

%       ----a----
%       |       |
%       |       |
%       |       |
%       b       b
%       |       |
%       |       |
%       |       |
%       ----a----
%       XXXXXXXXX    <- beam clamped (we will deal with that later down the road)

a = 1;
b = 3;

% Let's visualize our beam in a MATLAB plot!

% In order to keep our main clean, we outsource this
% task to a newly created function, called visualizegeometry.
```

Editor - C:\Users\David\Documents\MATLAB\FEM\FEM_from_scratch\utility\visualizegeometry.m
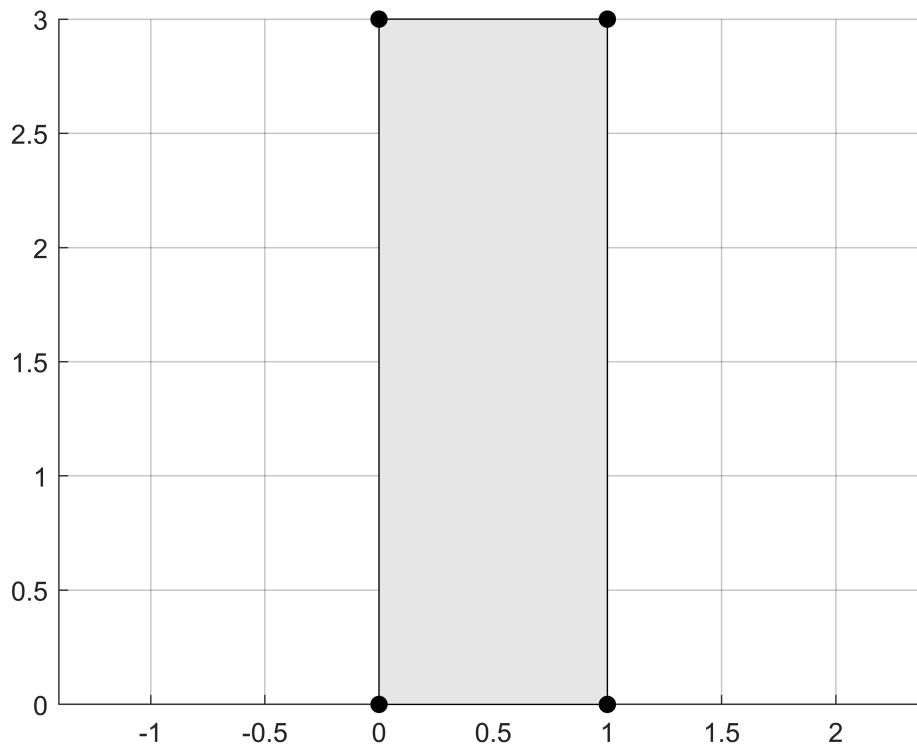
main.mlx ✕ visualizegeometry.m ✕ +

```matlab
1   function [] = visualizegeometry(a,b)
2   % This function visualizes the geometry
3
4   figure('Name','original geometry','NumberTitle','off');
5   % x,y are the coordinates of the 4 nodes
6   x = [0,a,a,0];
7   y = [0,0,b,b];
8   patch(x,y,[0.9 0.9 0.9],'EdgeColor','k','Marker','o','MarkerFaceColor','k');
9   axis equal
10  grid on
11  end
```

```matlab
visualizegeometry(a,b)
```

```
% Note how we call the helper function visualizegeometry.m
% located in the utility folder using a,b as input.
```

The starting point for the FE concept is given by the **weak form** of the governing differential equations of linear-elastic continuum mechanics.

$$\int_{\Omega} \delta \boldsymbol{u}^{\mathrm{T}} \boldsymbol{L}^{\mathrm{T}} \boldsymbol{C} \boldsymbol{L} \boldsymbol{u} \, \mathrm{d}\Omega = \int_{\Omega} \delta \boldsymbol{u}^{\mathrm{T}} \hat{\boldsymbol{b}} \, \mathrm{d}\Omega + \int_{\Gamma} \delta \boldsymbol{u}^{\mathrm{T}} \hat{\boldsymbol{t}} \, \mathrm{d}\Gamma$$

Again, interested students are referred to Prof. Wall's lecture "FEM" for a detailed derivation of above weak form. For the time beeing, let's consider this system of partially differentiable equations as given.

Given complex geometries Ω (that is basically any geometry of interest), solving for an analytical solution $\boldsymbol{u}$ ($\boldsymbol{u}$ = displacement field) is very hard. Instead we try to find an approximate solution $\boldsymbol{u}_h$ by transforming the continous problem into a discrete problem.

2. Discretization

We start of by partitioning our global geometry Ω into *numele* subdomains Ω^e, called finite elements. Typically these elements are cubical or tetrahedral in 3D or rectangular or triangular in 2D. Throughout the following, we will be using 2D-bilinear rectangular elements. The starting- and endpoint, or more general the edges of an element, are denoted as nodes.

Below, I have visually demonstrated the concept of geometry discretization or "meshing" using 2D-bilinear rectangular elements on our beloved beam.
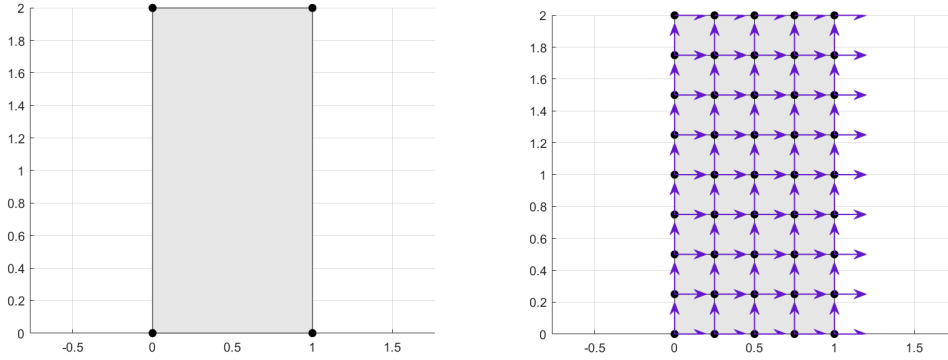
The position (and deformation) of each individual element can be described using the nodal position $\overline{\mathbf{x}}^{(e)}$, given in global coordinates x,y. In order to have the possibility to define functions in a local coordinate system in the same way for each element, we introduce a local parameter space $\xi \in [-1, 1]$ on each element.
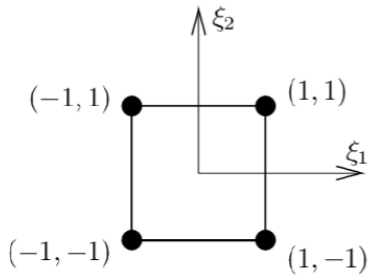

Let us define the node at the bottom left (-1,-1) as node 1. All the remaining nodes are labeled anti-clockwise.

Now, we define four 2D-bilinear shape functions $N^i(\xi_1, \xi_2)$, each taking on the value 1 at the corresponding node i and taking on the value 0 at all other nodes.

$$N^1 = \tfrac{1}{4}(1 - \xi_1)(1 - \xi_2) \quad N^2 = \tfrac{1}{4}(1 + \xi_1)(1 - \xi_2)$$
$$N^3 = \tfrac{1}{4}(1 + \xi_1)(1 + \xi_2) \quad N^4 = \tfrac{1}{4}(1 - \xi_1)(1 + \xi_2)$$

Let us take some time to get familiar with this concept. Consider the shape function N^1. According to its definition, it is taking the value 1 at the node 1 (for justification, try inserting (-1,-1) into N^1) and 0 at all other nodes. This is visualized below:
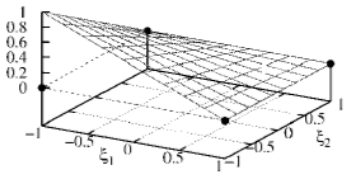
The introduction of shape functions allows us to approximate the geometry $x^{(e)}$ by

$x^{(e)}{}_h(\xi_1,\xi_2) = N(\xi_1,\xi_2)\,\bar{x}^{(e)}$. Here, the vector $\bar{x}^{(e)}$ holds the spatial coordinates $\bar{x}_i^{1,2}$ of each node i in x and y direction.

Similarily, we approximate the elementwise displacement field $u^{(e)}$ by the following expression.

$u^{(e)}{}_h(\xi_1,\xi_2) = N(\xi_1,\xi_2)\,d^{(e)}$, or

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}(\xi_1,\xi_2) = \underbrace{\begin{bmatrix} N^1 & 0 & N^2 & 0 & N^3 & 0 & N^4 & 0 \\ 0 & N^1 & 0 & N^2 & 0 & N^3 & 0 & N^4 \end{bmatrix}}_{N(\xi_1,\xi_2)} \underbrace{\begin{bmatrix} u_1^1 \\ u_2^1 \\ u_1^2 \\ u_2^2 \\ u_1^3 \\ u_2^3 \\ u_1^4 \\ u_2^4 \end{bmatrix}}_{d^{(e)}} .$$

Note that, the vector of nodal displacements $d^{(e)}$ is the solution of the FEM problem and yet to be determined!

For the time being, that's enough theoretical input. Let's start to code a simple meshing algorithm.

```
% We start off by specifying the uniform thickness of our structure T.
T = 1;
% Then we set the element size h of the rectangular elements.
h_rel = 1;
h = h_rel*min(a,b);    %  h_rel*min(a,b)

% Let's specify our material properties using a structure array.
mat = struct('name', 'Hooke', ...
             'emod', 10.e6, ...
             'poisson', 0.0);

% mat = struct('name', 'Hooke', ...
%              'emod', 1, ...
%              'poisson', 0.0);

% Let's specify our loads using a structure array.
Fg_vol = (9.81);
load = struct('volume',[0; -Fg_vol],...
              'traction',[0;0]);
% As we are interested in a clamped beam subject to a point force, we do
% not need any traction stresses. Neithertheless, we want to consider
% gravity. (which is modeled as a body or volume force)

% Finally, we set up a structure array holding all necessary inputs.
flag = struct('type', '2D-bilinear',  'thickness', T,  'material', mat,'numele',[],'load',load)
% Note how we can use structure arrays within structure arrays.
% The entry "numele", hence the number of finite elements, is yet to be specifed.


% At this point everything is set for the discretization. Once again we are
% using a newly created function "meshing.m" to do the heavy-lifting.
[Xbar,EDOF,GDOF,flag] = mashing(h,a,b,flag);
```
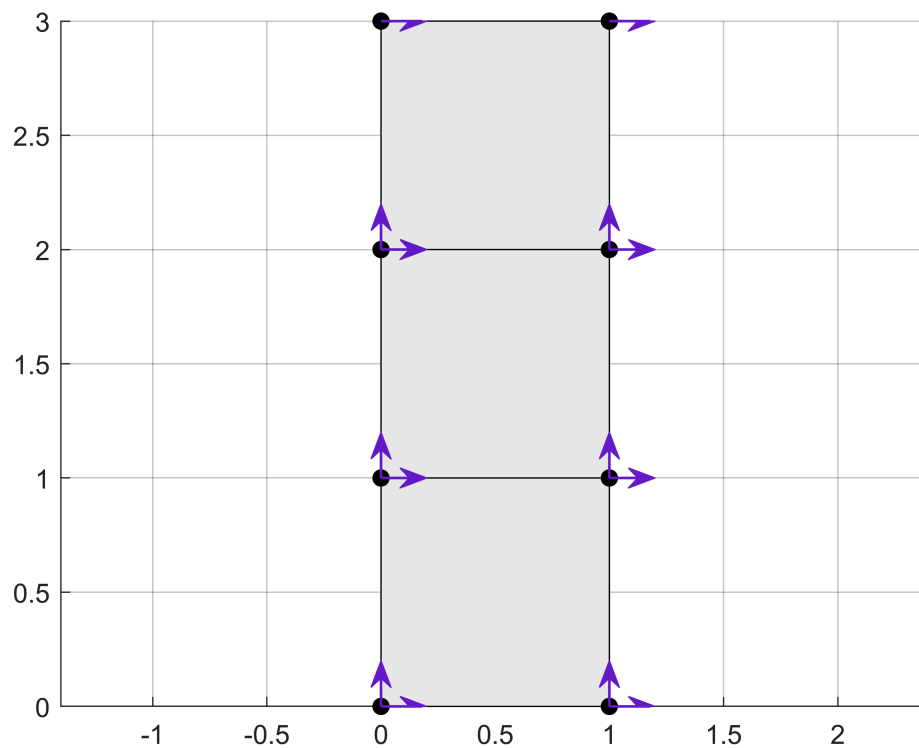
Mashed the geometry into 3 elements.

```
% The function takes the element size h and the geometric parameters a and b
% as input. Besides that, we also give the structured array.

% [Xbar,EDOF,GDOF,flag] are the functions outputs.
Xbar
```

```
Xbar = 8×3
     0     0     0
     0     1     2
     1     1     1
     0     1     2
     1     1     1
     1     2     3
     0     0     0
     1     2     3
```

```
% Xbar holds the absolut position of each node. [#of element-nodes,numele] = size(Xbar)

% NODE NUMBERING:
%      --------a-------
%      | . ...         |
%      | .             |
%      b . ...         b
%      | 4-3 8-7       |
%      | 1-2 5-6 ...   |
%      --------a-------
%      XXXXXXXXXXXXXXXX   <- beam clamped (we will deal with that later down the road)
```

```
EDOF,GDOF
```

```
EDOF = 8×3
        1       9      17
        2      10      18
        3      11      19
        4      12      20
        5      13      21
        6      14      22
        7      15      23
        8      16      24
GDOF = 8×3
        1       7      11
        2       8      12
        3       5       9
        4       6      10
        5       9      13
        6      10      14
        7      11      15
        8      12      16
```

```
% EDOF and GDOF are the element/global degrees of freedom. GDOF, hence the
% global DOF of the structure will be essential for the later assembly!

% The structured array "flag" is returned as an output because we will be
% updating it within the function. (We will assign the # of element: "numele")
```

At this point the geometry has been meshed and we are set to proceed with the computation of the elementwise stiffness matrix and load vector.

3. Elementwise computation of stiffness matrix $k^{(e)}$ and load vector s0

After having discretized the global geometry into *numele* finite elements, the following discrete and elementwise formulation of the weak form results. In essence, we replaced the integral over the whole domain Ω with the sum of integrals over all *numele* individual elements Ω^e.

$$\sum_{e=1}^{numele} \left[\delta d^{(e)\mathrm{T}} \left\{ \underbrace{\int_{\Omega^{(e)}} B^{\mathrm{T}} C B \, \mathrm{d}\Omega}_{\text{element stiffness matrix } k^{(e)}} d^{(e)} - \underbrace{\int_{\Omega^{(e)}} N^{\mathrm{T}} \hat{b} \, \mathrm{d}\Omega - \int_{\Gamma_\sigma^{(e)}} N^{\mathrm{T}} \hat{t} \, \mathrm{d}\Gamma}_{\text{element load vector } s^{0(e)}} \right\} \right]$$

Note that we are dealing with a finite set of (*numele*) equations. In order to calculate the element stiffness matrix $k^{(e)}$, we have a closer look at the following integral:

$$k^{(e)} = \int_{\Omega^{(e)}} B^{\mathrm{T}} C B \, \mathrm{d}\Omega$$

Let's take a moment and try to get familiar with above expression.

The **B**-Operator is defined as the matrix product of the linear differential operator **L** and the matrix of shape functions **N.**

$$\boldsymbol{B} = \boldsymbol{LN}$$

Considering our 2 dimensional problem at hand, the differential operator is defined as

$$\boldsymbol{L} = \begin{bmatrix} \frac{\partial}{\partial x_1} & 0 \\ 0 & \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_1} \end{bmatrix}$$

The **C**-matrix prescribes the elastic behaviour of our material. As we are dealing with a plane strain problem, the **C**-matrix is computed as follows

$$\boldsymbol{C} = \frac{E}{1 - \nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}.$$

After having multiplied above matrices accordingly, we get the integrand for the element

stiffness matrix $\boldsymbol{k}^{(e)}$.

Coordinate Transformation

Remember that all the quantities are defined as functions of the parameter space ξ. Consequently, we would like to do the integration in this parameter space. Using the Jacobian determinant $|\boldsymbol{J}|$, we have to transform the integral from (x, y) to (ξ_1, ξ_2).

$$\boldsymbol{k}^{(e)} = T \int_{-1}^{1} \int_{-1}^{1} (\boldsymbol{LN})^{\mathrm{T}} \boldsymbol{CLN} \underbrace{\overbrace{ab}^{|\boldsymbol{J}|} \mathrm{d}\xi_1 \mathrm{d}\xi_2}_{\mathrm{d}x_1 \mathrm{d}x_2}$$

with the Jacobian Matrix $\boldsymbol{J}$

$$\boldsymbol{J} = \frac{\partial(\boldsymbol{x})}{\partial(\boldsymbol{\xi})}.$$

Numerical integration scheme "Gauss-Legendre quadrature"

In praxis we almost always apply efficient numerical integration methods. Here, we will be using Gauss(-Legendre) integration.

For more reference, consult your favorite math lectures or the web: https://en.wikipedia.org/wiki/Gaussian_quadrature

What about the element load vector?

The computation procedure of the element load vector follows the same steps as the element stiffnes matrix. As we are primarily applying point forces, we will skip the theoretical discussion at this point. (You will see later how we apply the point forces directly...)

```
% As before, the actual coding has been done in several (nested) functions.
% Although not explained, the computation of the load vector is fully implemented.

[k_e,s0_e] = numerical_computation(Xbar,flag);
```

4. Assembly to global stiffness matrix *K* and global force vector *F*

In the previous section, we dealt with the evaluation of the element integrals resulting in the element stiffness matrix $k^{(e)}$ and the element load vector **s0**. Now, we deal with the assembly of the element quantities to a global stiffness matrix **K** and a global load vector **F**.

In standard FEM, the primary field (here: displacements) is chosen continuous, meaning that elements which share a node, also share the nodal quantity (the nodal displacement) values at that node. That being said, two neighboring elements both contribute to the same nodal displacement value.

```
[K,F] = assembly(k_e,s0_e,Xbar,GDOF);
% The function provides the following quantities as output:
%   K - global stiffness matrix of our (discretized) structure
%   F - global load vector of our (discretized) structure
```

5. Apply Dirichlet boundary conditions & external point loads

locked/supported global DOF

```
DBC = [1,2,3,4, 9,10,11,12, 17,18]';     % DBC consists of all fixed DOF
ext_force = [77,-10000;...
             79,-10000;...
             75,-10000];
F(ext_force(:,1)) = F(ext_force(:,1)) + ext_force(:,2);

[K_red,F_red] = enforceDBC(K,F,DBC);
```

6. Solve LGS KD=F for D

```
D = solveFEM(K_red,F_red,DBC);
```

7. Post-processing -> compute resulting stress & strain due to deflection u

```
[stresses,strains,dbar_e] = postprocessing(Xbar,D,GDOF,flag);
```

8. Visualize results

```
visualizeresults(Xbar,dbar_e,GDOF,stresses,strains)
```