TUDelft

Delft University of Technology

CIEM5000 - Matrix method in Statics

van Woudenberg, Tom; Rocha, Iuri

Publication date
2025

Document Version
Proof

Citation (APA)
van Woudenberg, T., & Rocha, I. (2025). *CIEM5000 - Matrix method in Statics*. Delft University of Technology. https://ciem5000-2025.github.io/book/

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Home

Welcome to the submodule on the Matrix Method for Statics, part of Unit 2 of CIEM5000 Course base Structural Engineering at Delft University of Technology.

This TeachBook contains the material for the course.

Course schedule 📅

See the schedule below for the different weeks. Clicking the links will take you to the relevant content pages.

Week	Monday	Tuesday	Thursday
1		Lecture 1	Workshop 1
2		Lecture 2	Workshop 2, at any moment after workshop 2: Additional assignments
7			Question hour of all parts of CIEM5000
8			Question hour of all parts of CIEM5000
April 18th, 23:59	Hand in report: Graded assignment		
June 20th, 23:59	Hand in report for resit: Graded assignment		

How to use this TeachBook 📘

Contents

> ⓘ **Added in version v2025.0.2:** 2025-02-11 10:42
>
> Added html export of book as zip

This TeachBook combines the course schedule and content. Announcement an grading results are provided via Brightspace.

Interactive features

This TeachBook includes interactive coding features! Click 🚀 –> Live Code on the top right corner of interactive page to start up a python-kernel in your browser! Any interactions you do here are not stored. You can also download those pages as a notebook to apply the content on your own computer. For the practice exercises this book shows a preview of the notebooks and py-files. These pages allow you to test the functionality. However, please fork and clone the assignment to work on it locally from [GitHub](). This allows you to edit multiple files simultaneously and save your work.

Spot a mistake?

If you spot any mistakes, you can click on ⓞ –> 💡, login with a GitHub account and report your issue. It'll be solved soon!

Personalised book

If you'd like to make this TeachBook more personal by adding (private or public) annotations I can recommend the [Hypothesis extension](). This is only for your own use, I won't monitor public post on this platform.

Version

This is the `2024-2025`-version of the TeachBook. Updates during this course are communicated on the relevant pages and in [the changelog](). After each of the workshops, updates will follow containing the solutions to the practice exercises.

Offline book

If you'd like to have an offline version of this book, you can download it by clicking 📘 on the top right corner. Note that some interactive features will not work (for example, the Python code). You can always download separate pages by clicking the ⬇ button on any page.

Contact information 💬

Contents

- Tom van Woudenberg
- Iuri Rocha

This submodule is taught by Tom van Woudenberg and Iuri Rocha. Please contact us if you've any questions, feedback or when you've personal circumstances which we should know.

Tom van Woudenberg

- Room 6.45
- 015-2789739
- T.R.vanWoudenberg@tudelft.nl

Iuri Rocha

- Room 6.40
- 015-2781458
- I.Rocha@tudelft.nl

Lecture 1

> ⚠️ ***Changed in version v2025.0.1:*** before first lecture
>
> Updated lecture slides: moved slides on python packages and updated installation requirements first workshop

During today's lesson we'll get started with the basics of the matrix method: you will be introduced to the Matrix Method for solving combinations of 1D elements in statics. You'll review how we solved single- and multiple-field problems in statics during Q2, and how you might have solved statically indeterminate structures using the displacement method. We'll see how that procedure can be streamlined and optimized for computer code. Following you'll be presented the basic concepts of the Matrix Method: Obtaining element stiffness matrices, local-global transformations, assembly and postprocessing. The lecture will be finalized by arriving at an abstraction of the Matrix Method that makes it readily implementable in computer code with the help of Object Oriented Programming (OOP).

This lecture is given by Tom van Woudenberg.

This book shows the full content of the first lecture. The slides of the lecture have the same content and are available ⬇ here

Recap differential equations for structures

Contents

Let's recap how to solve a structure using differential equations.

> **Learning objective**
>
> We'll investigate the differences between solving structures with differential equations and the matrix method. Furthermore, you'll need differential equations in the matrix method itself to define some default elements.

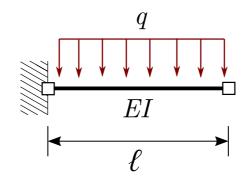Given is a cantilever beam with a distributed load:

Fig. 1 Cantilever beam with distributed load

The differential equation for the Euler-Bernoulli model can be derived leading to:

- Kinematic relations:
  - $\varphi = -\dfrac{\mathrm{d}w}{\mathrm{d}x}$

- $\kappa = \dfrac{\mathrm{d}\varphi}{\mathrm{d}x}$
- Constitutive relation:
  - $M = EI\kappa$
- Equilibrium relations:
  - $\dfrac{\mathrm{d}V}{\mathrm{d}x} = -q$
  - $\dfrac{\mathrm{d}M}{\mathrm{d}x} = V$

These relations can be combined into one fourth order differential equation:

$$EI\frac{\mathrm{d}^4 w}{\mathrm{d}x^4} = q$$

This differential equation can be solved directly to solve structures.

Solving differential equation of one field by hand

This differential equations can be solved by integrating:

- $V(x) = -qx + \bar{C}_1$
- $M(x) = -\dfrac{qx^2}{2} + \bar{C}_1 x + \bar{C}_2$
- $\kappa(x) = -\dfrac{qx^2}{2EI} + \tilde{C}_1 x + \tilde{C}_2$
- $\varphi(x) = -\dfrac{qx^3}{6EI} + \dfrac{\tilde{C}_1 x^2}{2} + \tilde{C}_2 x + \tilde{C}_3$
- $w(x) = \dfrac{qx^4}{24EI} + \dfrac{C_1 x^3}{6} + \dfrac{C_2 x^2}{2} + C_3 x + C_4$

The boundary conditions follow from the clamped side at $x = 0$ and free end at $x = \ell$:

- $w(0) = 0$
- $\varphi(0) = 0$
- $M(\ell) = 0$
- $V(\ell) = 0$

Solving these four equations for the integration constants gives:

- $C_1 = -\dfrac{q\ell}{EI}$

- $C_2 = \dfrac{q\ell^2}{2EI}$

- $C_3 = 0$

- $C_4 = 0$

Substituting these constants, a final solution for w can be found:

$$w(x) = \frac{qx^4}{24EI} - \frac{q\ell x^3}{6EI} + \frac{q\ell^2 x^2}{4EI}$$

Solving differential equation of one field using SymPy

The differential equation can also be solved using SymPy:

```
import sympy as sym
```

```
EI, q, x, L = sym.symbols('EI, q, x, ell')
C_1, C_2, C_3, C_4 = sym.symbols('C_1, C_2, C_3, C_4')
```

```
V = -sym.integrate(q,x) + C_1
M = sym.integrate(V,x) + C_2
kappa = M/EI
phi = sym.integrate(kappa,x) + C_3
w = -sym.integrate(phi,x) + C_4
display(w)
```

$$-\frac{C_1 x^3}{6EI} - \frac{C_2 x^2}{2EI} - C_3 x + C_4 + \frac{qx^4}{24EI}$$

```
eq1 = sym.Eq(w.subs(x,0),0)
eq2 = sym.Eq(phi.subs(x,0),0)
eq3 = sym.Eq(M.subs(x,L),0)
eq4 = sym.Eq(V.subs(x,L),0)
C_sol = sym.solve([eq1, eq2, eq3, eq4 ], [C_1, C_2, C_3, C_4])
for key in C_sol:
    display(sym.Eq(key, C_sol[key]))
```

$$C_1 = \ell q$$

$$C_2 = -\frac{\ell^2 q}{2}$$

$$C_3 = 0$$

$$C_4 = 0$$

```
w.subs(C_sol)
```

$$\frac{\ell^2 q x^2}{4EI} - \frac{\ell q x^3}{6EI} + \frac{q x^4}{24EI}$$

Solving differential equations of two fields

A similar approach can be taken when solving two fields.

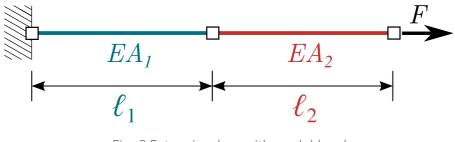Let's investigate the following structure, consisting of two fields.

Fig. 2 Extension bar with nodal load

As this structure is loaded along its axis, the differential equation for extension is used.

For the first field this gives:

- $EA_1 \dfrac{\mathrm{d}^2 u_1}{\mathrm{d}x^2} = 0$

- $N_1 = C_1$

- $u_1(x) = \dfrac{C_1}{EA} x + C_2$

- Boundary conditions: $u_1(0) = 0$

For the second field it gives:

- $EA_2 \dfrac{\mathrm{d}^2 u_2}{\mathrm{d}x^2} = 0$

- $N_2 = C_3$

- $u_2(x) = \dfrac{C_3}{EA}x + C_4$

- Boundary conditions: $N_2(\ell_1 + \ell_2) = F$

The two remaining integration constants can be solved by specifying interface conditions:

- $u_1(\ell_1) = u_2(\ell_1)$

- $N_1 = N_2$

Solving differential equations of more fields

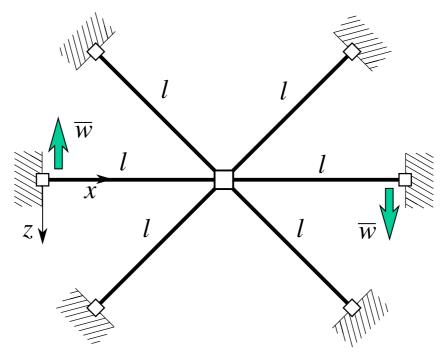The same approach can be taken to tackle problems with more field, like the one below:

Fig. 3 Frame structure with many fields

How many integration constants should be solved for here? How many boundary- and interface conditions would be needed for that? It gets annoying very quickly as each of these conditions need to be defined carefully.

Equivalence with matrix method

While both methods segment the structure in different parts, the matrix method applies a different principle in solving the structure than when directly solving differential equations: instead of solving for integration constants, nodal displacements are solved for. This shows big potential because setting up all the boundary- and interface conditions can be tedious and is problem-specific. The matrix method applies a generic algorithmic approach to combine all unknown nodal displacements

The similarities and differences are shown in the table below.

Table 2 Equivalence solving differential equations and matrix method

Solving differential equations	Matrix method
Segment structure in separate fields	Segment structure in mostly repetitive elements
Define all boundary- and interface conditions	Define relations in generic algorithmic manner
Solve for integration constants $C_1, C_2, \ldots$	Solve nodal displacements $u_1, u_2, \ldots$

Recap displacement method

Contents

In the previous [chapter](#) you've seen how solving for integration constants because a labour-intensive process for more complicated structures. A way of circumventing that is solving for nodal displacements! You might have seen that before when solving statically indeterminate structures using the displacement method!

> **Learning objective**
>
> We'll investigate the equivalence between solving structures with the displacement method and the matrix method.

The displacement method for statically indeterminate structures works by defining a single nodal displacement of the equivalent statically determinate structure which defines the displacement of the full structure. The nodal displacement can be solved for by equilibrium relations of the external forces and the force corresponding to the displacement. Solving this equation allows you to find the full displacement- and force distribution.

Let's look at an statically indeterminate example
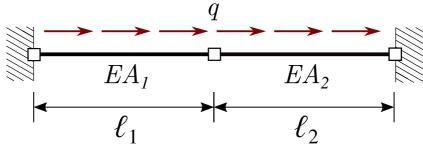

Fig. 4 Statically indeterminate extension bar

A statically determinate equivalent structure is i.e. the same structure with the middle connection replaced by a displacement u_2 and its corresponding reaction force $F^{(1)}$ and

$F^{(2)}$. This leads to two parts. If $F^{(1)} = F^{(2)}$, the structure is equivalent to the statically indeterminate structure.
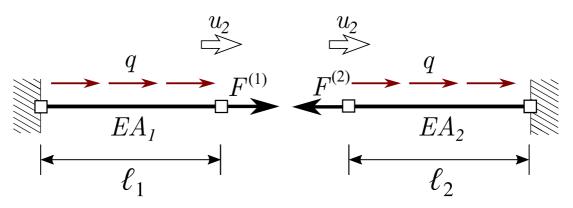
Fig. 5 Equivalent statically determinate extension bar if $F^{(1)} = F^{(2)}$

Displacements of all parts statically equivalent structure

Due to the nodal load, a constant section force is present in both fields. Using the constitutive equations $\Delta \ell = \dfrac{N\ell}{EA}$, the corresponding displacement u_2 can be found. For the load q, the displacement can be calculated using the kinematic, constitutive and kinematic relations. This leads to u_2 as a function of F_1 and F_2:

- $u_2 = \dfrac{\ell_1}{EA_1} F^{(1)} + \dfrac{\ell_1^2 q}{2EA_1}$
- $u_2 = -\dfrac{\ell_2}{EA_1} F^{(2)} + \dfrac{\ell_2^2 q}{2EA_2}$

These relations can be rewritten as F_1 and F_2 in terms of u_2 so that the force equilibrium can be solved for:

- $F^{(1)} = \dfrac{EA_1}{\ell_1} u_2 - \dfrac{\ell_1 q}{2}$
- $F^{(2)} = -\dfrac{EA_2}{\ell_2} u_2 + \dfrac{\ell_2 q}{2}$

Solve for displacements

Using $F^{(1)} = F^{(2)}$ the displacement u_2 can now be solved for:

$$\frac{EA_1}{\ell_1} u_2 - \frac{\ell_1 q}{2} = -\frac{EA_2}{\ell_2} u_2 + \frac{\ell_2 q}{2}$$

$$\left(\frac{EA_1}{\ell_1} + \frac{EA_2}{\ell_2} \right) u_2 = \frac{\ell_1 q}{2} + \frac{\ell_2 q}{2}$$

$$u_2 = \frac{\dfrac{\ell_1 q}{2} + \dfrac{\ell_2 q}{2}}{\dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2}}$$

Equivalence with matrix method

The matrix method applies exactly the same principle as the displacement method; both are solving force equilibrium of nodal forces to find nodal displacements.

However, the displacement method becomes difficult to apply if multiple nodal displacements are taken into account, as nodal forces have effect on multiple nodal displacements. Furthermore, the calculation of the displacements of each part can become tedious because they're problem-dependent and external forces have to be taken into account in the full derivation.

The matrix method addresses these issues by splitting the structure in mostly identical elements for which the force-displacement relations for all potential nodal displacements are evaluated once and can be reused over and over again. The same approach is taken for external forces, of which the resulting relations can be added afterwards. Finally, the calculations are structured in matrices to allow for easy implementation in software.

The similarities and differences are shown in the table below.

Table 3 Equivalence displacement method and matrix method

Displacement method	Matrix method
Convert structure in two statically determinate parts	Convert structure in mostly identical elements
Evaluate one nodal displacements for each parts	Evaluate all free nodal displacements using standard elements
Solve nodal equilibrium where the two statically determinate parts are connected	Solve nodal equilibrium in matrix form $\mathbf{Ku} = \mathbf{f}$

Force-displacement relations single extension element

In the previous [chapter](#) you've seen how you can solve structure using nodal displacements. However, the approach was still very problem-dependent. As proposed, the matrix method solves this by defining a default element which can be solved for a priori.

> **Learning objective**
>
> Let's define the force-displacement relations for a single element loading in extension in terms of nodal displacements.

We'll do that using differential equations. Later you'll see how to get the same result using shape functions.

Let's consider the most simple extension element:
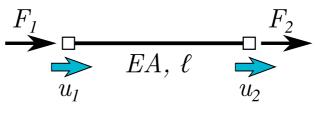
Fig. 6 Single extension element

The same approach is used as in [Recap differential equations for structures](#). However, the boundary conditions now are defined in terms of unknown nodal displacements:

- $u(0) = u_1$
- $u(\ell) = u_2$

This results in:

- $C_1 = \dfrac{u_2 - u_1}{\ell}$
- $C_2 = u_1$

Effectively, we replaced the unknown integration constants by unknown nodal displacements. However, this will prove to be useful because these nodal displacements

have a clear physical meaning and it will allow us to 'glue' elements together as other elements will be connected with the same nodal displacement.

Using our new formulation of unknown, the continuous distributions for the displacement and section force can be evaluated too:

- $u(x) = u_1 \left(1 - \dfrac{x}{\ell} \right) + u_2 \dfrac{x}{\ell}$
- $N = -\dfrac{EA}{\ell} u_1 + \dfrac{EA}{\ell} u_2$

We'll combine elements using force equilibrium, therefore, the force at the ends of the elements are of main interst. The section force N is derived above, which is not the same as the force F_1 and F_2 defined as positive to the right. For the node on the right, the direction of N coincides with the positive direction of F_1. However, on the left-hand-side, the sign flips. Leading to our force-displacement relation:

- $F_1 = -N = \dfrac{EA}{\ell} u_1 - \dfrac{EA}{\ell} u_2$
- $F_2 = N = -\dfrac{EA}{\ell} u_1 + \dfrac{EA}{\ell} u_2$

Combine elements

In the previous [chapter](#) you've seen how to set up the force-displacement relations for a single element. However, to solve for complete structures you'll need to combine multiple elements.

> **Learning objective**
>
> You'll look into how to combine elements to represent a full structure.

Let's reconsider the extension problem with two fields:
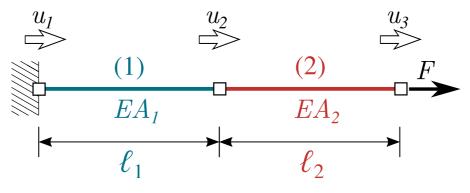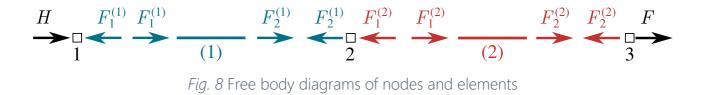
Fig. 7 Extension bar with nodal load

Both elements have the same solution as derived in [Force-displacement relations single extension element](#). To keep track of the different forces, we'll use the subscript F_1 and F_2 for defining the left- and right-end force of an element, the superscript $F^{(1)}$ and $F^{(2)}$ to define the element itself. The nodal displacement are numbered with a subscript u_1, u_2 and u_3.

Now, let's draw free body diagrams of the nodes and the elements itself. The forces acting on the ends of the elements, act in the opposite direction on the nodes. On node 1 and 3 additional forces are present: H for the reaction force in 1 which is assumed in the positive direction and F for the external force:

Fig. 8 Free body diagrams of nodes and elements

Horizontal equilibrium of each of the nodes gives:

- $\sum F_1 = 0 \Rightarrow -\dfrac{EA_1}{\ell_1} u_1 + \dfrac{EA_1}{\ell_1} u_2 + H = 0$

- $\sum F_2 = 0 \Rightarrow \dfrac{EA_1}{\ell_1} u_1 - \dfrac{EA_1}{\ell_1} u_2 - \dfrac{EA_2}{\ell_2} u_2 + \dfrac{EA_2}{\ell_2} u_3 = 0$

- $\sum F_3 = 0 \Rightarrow \dfrac{EA_2}{\ell_2} u_2 - \dfrac{EA_2}{\ell_2} u_3 + F = 0$

The algoritmic approach should become visible now!

These equations could also be regarding in a vector formulation. If $\mathbf{f}^e$ represents a vector with all the forces acting on each of the nodes coming from a single element and $\mathbf{f}_{\text{nodal}}$ represents a vector of all the nodal forces not coming from the elements (the reaction force H and external force F), these equations can be simplified as:

$$-\sum_e \mathbf{f}^e + \mathbf{f}_{\text{nodal}} = \mathbf{0}$$

$$\sum_e \mathbf{f}^e = \mathbf{f}_{\text{nodal}}$$

All that's needed now is to solve our linear set of equations for our unknown nodal displacements. Luckily the amount of equations equals the amount of unknowns, so you should have no problem solving this! Take into account that one displacement is already known: $u_1 = 0$. Without this, the matrix is singular.

However, solving our vector formulation is not trivial. $\mathbf{f}^e$ still contains our unknown nodal displacements hidden inside the vector. Let's split the vector on the next page to reach our final form of the matrix method formulation.

Combine elements using matrix formulation

Contents

In the previous chapter you've seen how to combine multiple elements to arrive to a linear system of equation to solve the nodal displacements. However, the nodal displacements were part of $\mathbf{f}^e$, which doesn't allow us to solve the vector formulation directly.

> **Learning objective**
>
> You'll look how to define local and global matrices and vectors of separately: unknown displacements vector, external force vector and elements stiffness matrices.

Local stiffness matrix, displacement vector and force vector

In Force-displacement relations single extension element you've derived the following two equations relating the nodal forces and displacements:

- $F_1 = \dfrac{EA}{\ell} u_1 - \dfrac{EA}{\ell} u_2$
- $F_2 = -\dfrac{EA}{\ell} u_1 + \dfrac{EA}{\ell} u_2$

These two equations can be rewritten in a matrix formulation:

$$\frac{EA}{\ell} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix}$$

This can represented as follows too:

$$\mathbf{K}^{(e)}\mathbf{u}^{(e)} = \mathbf{f}^{(e)}$$

$\mathbf{K}^{(e)}$ is known as the local stiffness matrix, $\mathbf{u}^{(e)}$ the local displacement vector and $\mathbf{f}^{(e)}$ the local force vector.

Global stiffness matrix, displacement vector and force vector

Now let's see how to combine multiple elements using our new matrix formulation. In Combine elements you've derived the following three equation relating the nodal displacements with external forces:

- $\sum F_1 = 0 \Rightarrow -\dfrac{EA_1}{\ell_1}u_1 + \dfrac{EA_1}{\ell_1}u_2 + H = 0$
- $\sum F_2 = 0 \Rightarrow \dfrac{EA_1}{\ell_1}u_1 - \dfrac{EA_1}{\ell_1}u_2 - \dfrac{EA_2}{\ell_2}u_2 + \dfrac{EA_2}{\ell_2}u_3 = 0$
- $\sum F_3 = 0 \Rightarrow \dfrac{EA_2}{\ell_2}u_2 - \dfrac{EA_2}{\ell_2}u_3 + F = 0$

These equation can be represented in a matrix formulation:

$$\begin{bmatrix} \dfrac{EA_1}{\ell_1} & -\dfrac{EA_1}{\ell_1} & 0 \\ -\dfrac{EA_1}{\ell_1} & \dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2} & -\dfrac{EA_2}{\ell_2} \\ 0 & -\dfrac{EA_2}{\ell_2} & \dfrac{EA_2}{\ell_2} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} H \\ 0 \\ F \end{bmatrix}$$

This can represented as follows too:

$$\mathbf{K}\mathbf{u} = \mathbf{f}$$

$\mathbf{K}$ is known as the global stiffness matrix, $\mathbf{u}$ the global displacement vector and $\mathbf{f}$ the glocal force vector.

As you might see, the local matrices are visible as 'blocks' in the global matrix, while the displacement and force vector are very clean. It seems to be possible to define the global stiffness matrix directly without manually evaluating the force equilibrium at every node! You'll look at that in the next chapter!

Setting up global equations directly

Contents

Now that you've seen the final matrix formulation, let's set up a procedure to find it directly!

> **Learning objective**
>
> You'll look into how to setup the global formulation directly

We'll do that for the same structure as before:
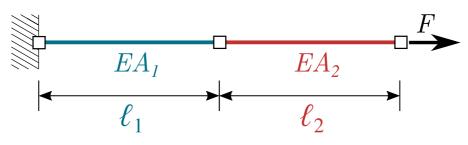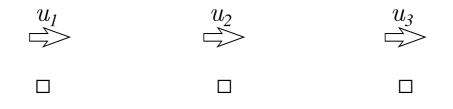
Fig. 9 Extension bar with nodal load

1. Identify degrees of freedom

The nodal displacement are the degrees of freedom. Let's define all of them, even though some seem to be fixed:

$$u_1 \quad\quad\quad u_2 \quad\quad\quad u_3$$

2. Initialize the system with zeros

As the amount of degrees of freedom is known, the size of our global matrices and vectors are defined and they can be initialized with zeros:
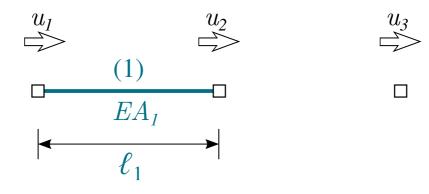
$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

3. Assemble stiffness, element by element

Now let's add the local stiffness matrices element by element. We can use the default stiffness matrix $\dfrac{EA}{\ell} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$ and places its parts in the global stiffness matrix where the index of the displacement (columns) and forces (rows) match.

Element (1)
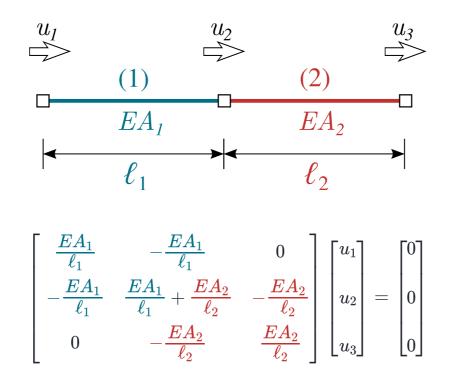
The first element links the first and second nodal displacement with the first and second nodal forces:

$$\begin{bmatrix} \dfrac{EA_1}{\ell_1} & -\dfrac{EA_1}{\ell_1} & 0 \\[2ex] -\dfrac{EA_1}{\ell_1} & \dfrac{EA_1}{\ell_1} & 0 \\[2ex] 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\[1ex] u_2 \\[1ex] u_3 \end{bmatrix} = \begin{bmatrix} 0 \\[1ex] 0 \\[1ex] 0 \end{bmatrix}$$

Element (2)

Now let's add the second element, linking the second and third nodal displacements with the second and third nodal forces:
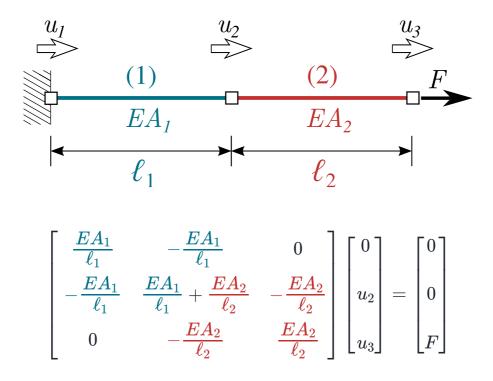
$$\begin{bmatrix} \dfrac{EA_1}{\ell_1} & -\dfrac{EA_1}{\ell_1} & 0 \\[2ex] -\dfrac{EA_1}{\ell_1} & \dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2} & -\dfrac{EA_2}{\ell_2} \\[2ex] 0 & -\dfrac{EA_2}{\ell_2} & \dfrac{EA_2}{\ell_2} \end{bmatrix} \begin{bmatrix} u_1 \\[1ex] u_2 \\[1ex] u_3 \end{bmatrix} = \begin{bmatrix} 0 \\[1ex] 0 \\[1ex] 0 \end{bmatrix}$$

4. Apply external loads

Now, the external loads can be applied. These external loads are called Neumann boundary conditions. These act directly on our nodes, so can be directly added to the global force vector. The sign of the forces to be added aligns with the positive direction of the nodal displacements.

$$
\begin{bmatrix} \dfrac{EA_1}{\ell_1} & -\dfrac{EA_1}{\ell_1} & 0 \\ -\dfrac{EA_1}{\ell_1} & \dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2} & -\dfrac{EA_2}{\ell_2} \\ 0 & -\dfrac{EA_2}{\ell_2} & \dfrac{EA_2}{\ell_2} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ F \end{bmatrix}
$$
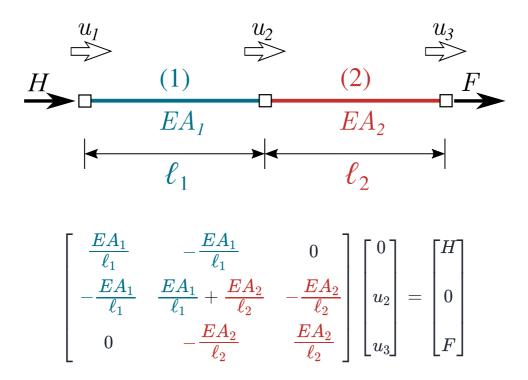
5. Apply prescribed displacements

Now, the external loads can be applied. These external loads are called Neumann boundary conditions. These act directly on our nodes, so can be directly added to the global force vector. The sign of the forces to be added aligns with the positive direction of the nodal displacements.

The Neumann boundary condition causes a prescribed displacement $u_1 = 0$

$$
\begin{bmatrix} \dfrac{EA_1}{\ell_1} & -\dfrac{EA_1}{\ell_1} & 0 \\ -\dfrac{EA_1}{\ell_1} & \dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2} & -\dfrac{EA_2}{\ell_2} \\ 0 & -\dfrac{EA_2}{\ell_2} & \dfrac{EA_2}{\ell_2} \end{bmatrix} \begin{bmatrix} 0 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ F \end{bmatrix}
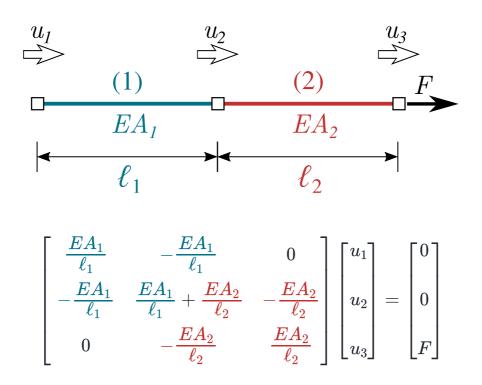$$

However, it also adds a force:

$$\begin{bmatrix} \dfrac{EA_1}{\ell_1} & -\dfrac{EA_1}{\ell_1} & 0 \\[2ex] -\dfrac{EA_1}{\ell_1} & \dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2} & -\dfrac{EA_2}{\ell_2} \\[2ex] 0 & -\dfrac{EA_2}{\ell_2} & \dfrac{EA_2}{\ell_2} \end{bmatrix} \begin{bmatrix} 0 \\[2ex] u_2 \\[2ex] u_3 \end{bmatrix} = \begin{bmatrix} H \\[2ex] 0 \\[2ex] F \end{bmatrix}$$

6. Solve for the unknown nodal displacements

Finally, we can solve for the unknown nodal displacements. For now, we can solve this system by only taking into account the second and third row:

$$\begin{bmatrix} \dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2} & -\dfrac{EA_2}{\ell_2} \\[2ex] -\dfrac{EA_2}{\ell_2} & \dfrac{EA_2}{\ell_2} \end{bmatrix} \begin{bmatrix} u_2 \\[2ex] u_3 \end{bmatrix} = \begin{bmatrix} 0 \\[2ex] F \end{bmatrix}$$

This results in:

- $u_2 = \dfrac{F\ell_1}{EA_1}$

- $u_3 = \dfrac{F\left(EA_1\ell_2 + EA_2\ell_1\right)}{EA_1\,EA_2}$

Later on, we'll introduce another way of solving the system of equations which allows for nonzero Dirichlet boundary conditions.

Implementation in Python

Contents

The matrix method is implemented in an (incomplete) python package which you'll extend in the workshops and graded assignments.

> **Learning objective**
>
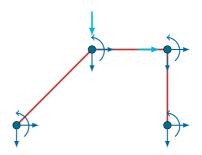> You'll look into the structure of the matrix method python package.

The method is broken down as follows:

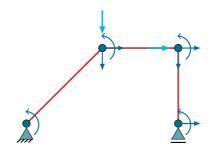- A list of nodes floating in space with loads and DOFs associated to them

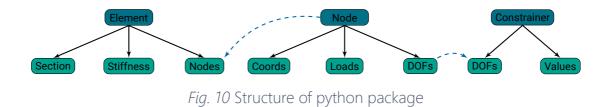- A list of elements defined by linking two nodes together

- A constrainer to apply Dirichlet boundary conditions

With this in mind, the python package is set up as an object-oriented code which each of the items above as a class, containing multiple attributes and functions. The docstring provide an introduction to the classes, attributes and functions. These will be treated later in the workshops

The three classes work together: nodes defined with the `Node` class are an input for the `Element` class. The `Constrainer` takes the nodes defined with the `Node` class and constrains specific degree of freedom to comply with Dirichlet boundary conditions.

Fig. 10 Structure of python package

The global stiffness matrix, nodal displacement vector and nodal force vector are not defined anywhere in each of the classes (only local stiffness matrices are defined in the `Element` class). So, these need to defined separately.

Setting up global equations with python package

The steps provide in the [previous page](#) can be implemented using the matrix method package as follows:

1. Identify degrees of freedom

The `Node` class automatically keeps track of the amount of degrees of freedom based on the amount of instances created with that class. For every node it will count 3 degrees of freedom. So far you've only seen one degree of freedom per node, but soon more will follow!

2. Initialize the system with zeros

As said before, the global stiffness matrix, nodal displacement vector and nodal force vector are not defined anywhere in each of the classes. Therefore, you'll to create empty matrices and vectors yourself in the form of numpy arrays.

3. Assemble stiffness, element by element

The `Element` class can create a local stiffness matrix based on the `Node`-instances which are provided as an input. You'll need some smart indexing to map the local stiffness matrix to the correct location of the global stiffness matrix. Again, this is not part of any of the provided classes.

4. Apply external loads

Loads can be added to the instances of the `Node` class. Together with the indexing of degrees of freedom inside these instances, the loads can be mapped to the correct location in the global nodal force vector.

5. Apply prescribed displacements

The prescribed displacements can be applied using the `Constrain` class. It allows you to fix the degrees of freedom, which you'll need in the next step. It doesn't take care of the reaction force belonging to the prescribed displacement.

6. Solve for the unknown nodal displacements

Again, the `Constrain` class can help you here. As it knows which degrees of freedom are fixed, it can give you a reduced global stiffness matrix and global nodal force vector which you can use yourself to solve for the free global nodal displacements

Local stiffness matrix Euler-Bernoulli element

Contents

- Derivation using SymPy

In [Force-displacement relations single extension element](#) and [Combine elements using matrix formulation](#) you've seen how to derive the local stiffness matrix for a simple extension element. But can you do the same for other elements?

> **Learning objective**
>
> You'll look into deriving the local stiffness matrix for the Euler-Bernoulli element.

Previously, the local stiffness matrix for a simple extension element was found as:

$$\mathbf{K}^{(e)} = \frac{EA}{\ell} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

The same procedure can be followed for other element, like a combined extension and Euler-Bernoulli element:
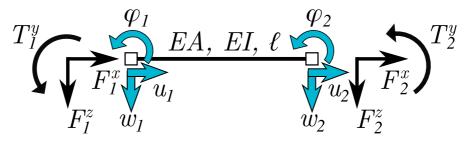

Fig. 11 Combined extension and Euler-Bernoulli element

The amount of degrees of freedom increases, as both ends of the element can translate in two directions and rotate. However, the approach is exactly the same, leading to the following element stiffness matrix:

$$\mathbf{K}^{(e)} = \begin{bmatrix} \dfrac{EA}{\ell} & 0 & 0 & -\dfrac{EA}{\ell} & 0 & 0 \\[2ex] 0 & \dfrac{12EI}{\ell^3} & -\dfrac{6EI}{\ell^2} & 0 & -\dfrac{12EI}{\ell^3} & -\dfrac{6EI}{\ell^2} \\[2ex] 0 & -\dfrac{6EI}{\ell^2} & \dfrac{4EI}{\ell} & 0 & \dfrac{6EI}{\ell^2} & \dfrac{2EI}{\ell} \\[2ex] -\dfrac{EA}{\ell} & 0 & 0 & \dfrac{EA}{\ell} & 0 & 0 \\[2ex] 0 & -\dfrac{12EI}{\ell^3} & \dfrac{6EI}{\ell^2} & 0 & \dfrac{12EI}{\ell^3} & \dfrac{6EI}{\ell^2} \\[2ex] 0 & -\dfrac{6EI}{\ell^2} & \dfrac{2EI}{\ell} & 0 & \dfrac{6EI}{\ell^2} & \dfrac{4EI}{\ell} \end{bmatrix}$$

for

$$\mathbf{u}^{(e)} = \begin{bmatrix} u_1 \\ w_1 \\ \varphi_1 \\ u_2 \\ w_2 \\ \varphi_2 \end{bmatrix}$$

Derivation using SymPy

We can make use of software like SymPy, as we did before in [Solving differential equation of one field using SymPy](#) to do the calculations in this derivation:

```
import sympy as sym
```

```
EI, x, L = sym.symbols('EI, x, L')
w = sym.Function('w')

ODE_bending = sym.Eq(w(x).diff(x, 4) * EI, 0)
display(ODE_bending)
```

$$EI\frac{d^4}{dx^4}w(x) = 0$$

```
w = sym.dsolve(ODE_bending, w(x)).rhs
display(w)
```

$$C_1 + C_2x + C_3x^2 + C_4x^3$$

```
phi = -w.diff(x)
kappa = phi.diff(x)
M = EI * kappa
V = M.diff(x)
```

```
w_1, w_2, phi_1, phi_2 = sym.symbols('w_1, w_2, phi_1, phi_2')

eq1 = sym.Eq(w.subs(x,0),w_1)
eq2 = sym.Eq(w.subs(x,L),w_2)
eq3 = sym.Eq(phi.subs(x,0),phi_1)
eq4 = sym.Eq(phi.subs(x,L),phi_2)

sol = sym.solve([eq1, eq2, eq3, eq4 ], sym.symbols('C1, C2, C3, C4'))
for key in sol:
    display(sym.Eq(key, sol[key]))
```

$$C_1 = w_1$$

$$C_2 = -\phi_1$$

$$C_3 = \frac{2L\phi_1 + L\phi_2 - 3w_1 + 3w_2}{L^2}$$

$$C_4 = \frac{-L\phi_1 - L\phi_2 + 2w_1 - 2w_2}{L^3}$$

```
F_1_z, F_2_z, T_1_y, T_2_y = sym.symbols('F_1_z, F_2_z, T_1_y, T_2_y')

eq5 = sym.Eq(-V.subs(sol).subs(x,0), F_1_z)
eq6 = sym.Eq(V.subs(sol).subs(x,L), F_2_z)
eq7 = sym.Eq(-M.subs(sol).subs(x,0), T_1_y)
eq8 = sym.Eq(M.subs(sol).subs(x,L), T_2_y)
```

```
K_e, f_e = sym.linear_eq_to_matrix([eq5,eq7, eq6, eq8], [w_1, phi_1, w_2, phi_2])
display(K_e)
```

$$
\begin{bmatrix}
\dfrac{12EI}{L^3} & -\dfrac{6EI}{L^2} & -\dfrac{12EI}{L^3} & -\dfrac{6EI}{L^2} \\[2mm]
-\dfrac{6EI}{L^2} & \dfrac{4EI}{L} & \dfrac{6EI}{L^2} & \dfrac{2EI}{L} \\[2mm]
-\dfrac{12EI}{L^3} & \dfrac{6EI}{L^2} & \dfrac{12EI}{L^3} & \dfrac{6EI}{L^2} \\[2mm]
-\dfrac{6EI}{L^2} & \dfrac{2EI}{L} & \dfrac{6EI}{L^2} & \dfrac{4EI}{L}
\end{bmatrix}
$$

To use the stiffness matrix without manually copying it over, you can make use of the `lambdify` which converts a symbolic SymPy object in a python function. This allows you to evaluate it for specific numerical values and continue using it in the numerical framework of the matrix method.

```python
K = sym.lambdify((L, EI), K_e)
print(K.__doc__)
```

```
Created with lambdify. Signature:

func(L, EI)

Expression:

Matrix([[12*EI/L**3, -6*EI/L**2, -12*EI/L**3, -6*EI/L**2], [-6*EI/L**2,...

Source code:

def _lambdifygenerated(L, EI):
    return array([[12*EI/L**3, -6*EI/L**2, -12*EI/L**3, -6*EI/L**2], [-6*EI/L**2, 4*EI/L,

Imported modules:
```

```python
print('Example of K with L=5 and EI=1000:\n',K(5,1000))
```

```
Example of K with L=5 and EI=1000:
 [[  96. -240.  -96. -240.]
 [-240.  800.  240.  400.]
 [ -96.  240.   96.  240.]
 [-240.  400.  240.  800.]]
```
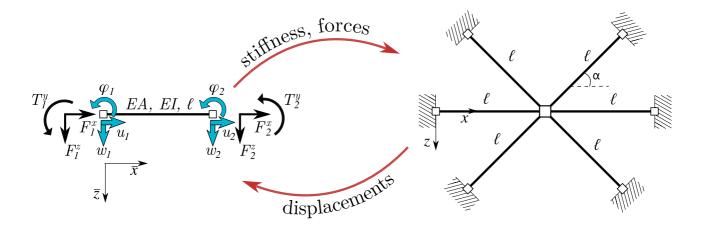
Transformations

Contents

Up until now we didn't care about the orientation of elements. Actually, all elements had exactly the same orientation. But how do deal with elements in a different orientation?

> **Learning objective**
>
> You'll look into how to transform elements from a local to global orientation
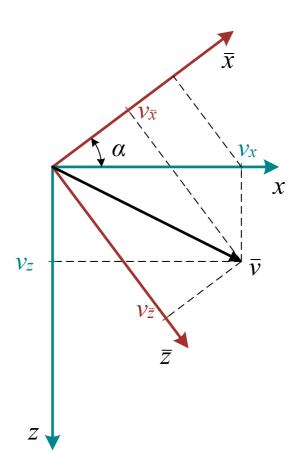
The stiffness matrix is defined in a local coordinate system following an element's orientation. This is useful because it allows us to reuse that same stiffness matrix and again without the need to rederive it. However, during assembly, when combining multiple elements, it would be useful to have them all in the same coordinate system, the global one. After solving for displacements in the global coordinate system, it might be needed to transform back to the local coordinate system to get expression for continuous displacements and section forces.
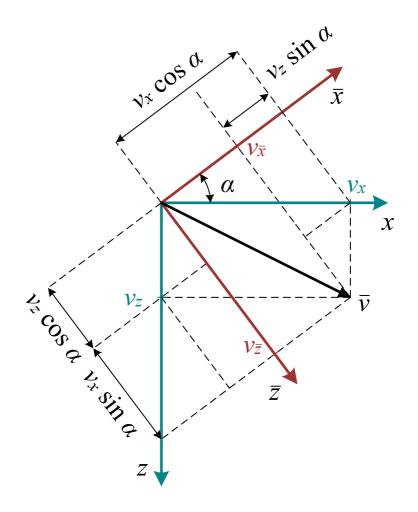
We'll be using a $x - z$-coordinate system as the differential equations are derived using those axes. This is different than in most finite-element-implementations and in most cases not in bar with international standards.

For an arbitrary vector v with two components, the transformation matrix can be derived by comparing the vector's components in the local $(v_{\bar{x}}, v_{\bar{z}})$ and global (v_x, v_z) coordinate system:

The relations between the components can be found using geometry:

Leading to:

$$v_{\bar{x}} = v_x \cos \alpha - v_z \sin \alpha$$
$$v_{\bar{z}} = v_x \sin \alpha + v_z \cos \alpha$$

This can be rewwritten in matrix form as:

$$\begin{bmatrix} v_{\bar{x}} \\ v_{\bar{z}} \end{bmatrix} = \underbrace{\begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix}}_{\mathbf{R}} \begin{bmatrix} v_x \\ v_z \end{bmatrix}$$

And the inverse relation:

$$\begin{bmatrix} v_x \\ v_z \end{bmatrix} = \underbrace{\begin{bmatrix} \cos \alpha & \sin \alpha \\ -\sin \alpha & \cos \alpha \end{bmatrix}}_{\mathbf{R}^{\mathrm{T}}} \begin{bmatrix} v_{\bar{x}} \\ v_{\bar{z}} \end{bmatrix}$$

Transformation for a complete element

To transform a complete element, the displacements of both endpoints have to be transformed, while the rotations are independent of the element orientation:

$$
\begin{bmatrix} \bar{u}_1 \\ \bar{w}_1 \\ \bar{\varphi}_1 \\ \bar{u}_2 \\ \bar{w}_2 \\ \bar{\varphi}_2 \end{bmatrix} = \underbrace{\begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & 0 & 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{T}} \begin{bmatrix} u_1 \\ w_1 \\ \varphi_1 \\ u_2 \\ w_2 \\ \varphi_2 \end{bmatrix}
$$

Resulting in:

- $\bar{\mathbf{u}} = \mathbf{T}\mathbf{u}$
- $\bar{\mathbf{f}} = \mathbf{T}\mathbf{f}$
- $\mathbf{u} = \mathbf{T}^{\mathrm{T}}\bar{\mathbf{u}}$
- $\mathbf{f} = \mathbf{T}^{\mathrm{T}}\bar{\mathbf{f}}$

Transformation for stiffness matrix

Using the known transformation for the first-order tensors $\mathbf{u}$ and $\mathbf{f}$ the transformation matrix for the second-order tensor $\mathbf{K}$ can be derived:

$$
\begin{aligned}
\bar{\mathbf{K}}\bar{\mathbf{u}} &= \bar{\mathbf{f}} \\
\bar{\mathbf{K}}\mathbf{T}\mathbf{u} &= \mathbf{T}\mathbf{f} \\
\mathbf{T}^{\mathrm{T}}\bar{\mathbf{K}}\mathbf{T}\mathbf{u} &= \mathbf{T}^{\mathrm{T}}\mathbf{T}\mathbf{f} \\
\mathbf{K}\mathbf{u} &= \mathbf{f}
\end{aligned}
$$

So $\mathbf{K} = \mathbf{T}^{\mathrm{T}}\bar{\mathbf{K}}\mathbf{T}$

Workshop 1

> ⊘ **_Added in version v2025.1.0:_** After workshop 1
>
> Solutions workshop 1 in downloads

> ❗ **Attention**
>
> This pages shows a preview of the assignment including its solution. Please fork
> and clone the assignment to work on it locally from GitHub

During today's workshop you'll implement and check missing components, and solve a
complicated frame.

Implementation

Contents

> ⚠️ **Attention**
>
> This page shows a preview of the assignment. Please fork and clone the assignment to work on it locally from [GitHub](#)

> ⓘ *Added in version v2025.1.0:* After workshop 1
>
> Solutions workshop 1 in text and downloads

> ⚠️ *Changed in version v2025.0.3:* 2025-02-10 13:33, before workshop 1
>
> Fixed typo in [Exercise 2.6](#)

In this notebook you will implement the matrix method and check it with some sanity checks.

Our matrix method implementation is now completely stored in a local package, consisting of three classes. If you need a refresher on how to code with Classes and Objects, refer to the [section on Object Oriented Programming in the MUDE-book](#), with additionally [programming assignment 1.7](#)

```python
import numpy as np
import matplotlib as plt
import matrixmethod as mm
%config InlineBackend.figure_formats = ['svg']
```

1. The Node class

This class is stored in `./matrixmethod/node.py`

The purpose of this class is to store node information and keep track of the total number of DOFs of the problem. Note the automatic bookkeeping we introduce in `__init__`. This simple but efficient way of keeping track of which DOFs belong to which nodes will make life much easier when we need to assemble matrices from multiple elements. The Node class doesn't need any modification.

> 🧩 **Exercise (Workshop 1 - 1.1)**
>
> To test whether you understand how the class works, create two nodes on coordinates $(0,0)$ and $(3,4)$ and print the string representation of both nodes. The `clear` function is called to restart the node and DOF counters. Make sure this is done whenever you start solving a new problem.
>
> ```
> mm.Node.clear()
>
> node1 #= mm.Node(YOUR CODE HERE
>
> print(node1)
> #YOUR CODE HERE
> ```

> 💡 **Hint** ⌄

> 📝 **Solution to Exercise (Workshop 1 - 1.1)** ⌄

2. The Element class

This class is stored in `./matrixmethod/elements.py`

This class keeps track of each element in the model, including:

- Cross-section properties
- Element orientation (for coordinate system transformations)
- Which Nodes make up each element, and in turn (with help of the Node class) which DOFs belong to each element

Apart from bookkeeping element data, the other main task of this class is to provide the element stiffness matrix in the global coordinate system (for subsequent assembly) and postprocess element-level fields. For now we keep postprocessing for next week and focus only on getting the correct stiffness matrix.

Here the class describes an element combining extension and Euler-Bernoulli bending. A similar (or inherited) class could also be implemented for different element types (*e.g.* shear beam, Timoshenko beam, cable elements, etc). Here we also keep it simple by assuming elements are all arranged in a 2D plane.

However, the implementation is incomplete:

- The transformation matrix is missing in `__init__`, which is given in Transformations. Make sure you take into account that a positive Δz with a positive Δx gives a negative angle α. Make use of `numpy.arctan2` to return the angle between $-\pi$ and π, `numpy.arctan` returns an angle between $-\dfrac{\pi}{2}$ and $-\dfrac{\pi}{2}$, and therefore cannot distinguish between all four quadrants.
- The correct stiffness matrix for this extension-bending element coordinate system is missing in `stiffness`. You can derive the stiffness matrix yourself using pen and paper, SymPy or Maple, or copy the given stiffness matrix from Local stiffness matrix Euler-Bernoulli element.
- We keep the functions which add a distributed load and compute the moments / displacements untouched for this week. Next week we'll implement those as well.

> 🧩 **Exercise (Workshop 1 - 2.1)**
>
> Add the missing pieces to the code in `./matrixmethod/elements.py`, before you perform the checks below. Do you specify your stiffness matrix in the global or local coordinate system?

> 📄 **Solution to Exercise (Workshop 1 - 2.1)**   ⌄

Whenever you make changes to your code in the `./matrixmethod/` folder, you need to reimport those. Instead of restarting the kernel, we use some magic ipython commands. Run the cell below once. Consequently, whenever you save your changes in one of the `.py`-files, it's automatically reloaded.

```
%load_ext autoreload
%autoreload 2
```

🧩 Exercise (Workshop 1 - 2.2)

First, let's check the stiffness matrix for a beam which doesn't require rotation. Create a horizontal element with length 2 and $EI = 4$ and print both the transformation matrix and the stiffness matrix.

Do the matrices match with what you'd expect?

```
mm.Node.clear()
mm.Element.clear()
```

```
#YOUR CODE HERE

elem #= mm.Element(#YOUR CODE HERE

section = {}
section['EI'] #= YOUR CODE HERE
elem.set_section(section)

print(elem.T)
print(elem.stiffness())
```

📖 Solution to [Exercise (Workshop 1 - 2.2)](Exercise)   ⌄

🧩 Exercise (Workshop 1 - 2.3)

Now, create a vertical element with length 2 and $EI = 4$ and print the transformation and stiffness matrix.

Do the matrices match with what you'd expect?

```
#YOUR CODE HERE
```

Solution to [Exercise (Workshop 1 - 2.3)](#)

Exercise (Workshop 1 - 2.4)

Now, create an element rotated in $120°$ with length 2 and print the transformation matrix.

Do the matrices match with what you'd expect?

```
#YOUR CODE HERE
```

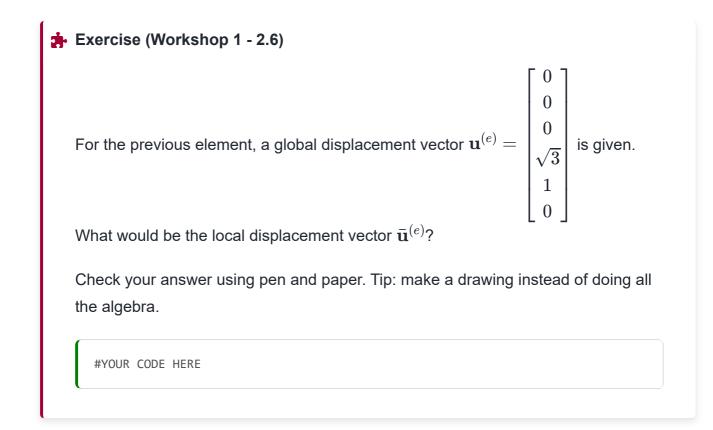## Solution to [Exercise (Workshop 1 - 2.4)](#)

Exercise (Workshop 1 - 2.5)

Now, create an element rotated in $60°$ with length 2 and print the transformation matrix.

Do the matrices match with what you'd expect?

```
#YOUR CODE HERE
```

Solution to [Exercise (Workshop 1 - 2.5)](#)

Exercise (Workshop 1 - 2.6)

For the previous element, a global displacement vector $\mathbf{u}^{(e)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \sqrt{3} \\ 1 \\ 0 \end{bmatrix}$ is given.

What would be the local displacement vector $\bar{\mathbf{u}}^{(e)}$?

Check your answer using pen and paper. Tip: make a drawing instead of doing all the algebra.

```
#YOUR CODE HERE
```

Solution to Exercise (Workshop 1 - 2.6) ⌄

3. The Constrainer class

This class is stored in `./matrixmethod/constrainer.py`

This small class keeps track of which DOFs have prescribed displacements and takes care of applying these constraints to the global $\mathbf{K}$ and $\mathbf{f}$. For now we keep it simple and assume all constraints fix the DOF values to zero. Next week we will deal with non-zero prescribed values.

However, the implementation is incomplete:

- The `constrain` function is incomplete, which should mimic the process of striking rows/columns of constrained DOFs and reduce the size of the system to be solved. Remember that `Constrainer` stores which DOFs are constrained in `self.dofs`, so **all the others** should be free. After gathering the free DOFs in an array, you will need to select the correct blocks of $\mathbf{K}$ and $\mathbf{f}$. For the stiffness matrix you will need the `np.ix_()` helper function (check its documentation here)
- We keep the function which calculates supports reaction untouched for this week. Next week we'll implement that one as well.

Exercise (Workshop 1 - 3.1)

Add the missing pieces to the code, before you perform the check below

Solution to Exercise (Workshop 1 - 3.1) ⌄
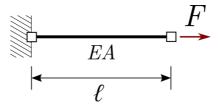
Exercise (Workshop 1 - 3.2)

Take the inclined element of exercise 2.5 and a bending stiffness of 1. What happens if you invert $\mathbf{K}$? Now fix all degrees of freedom of the first node. What happens when you invert your 'constrained' $\mathbf{K}$? Are the dimensions of the 'constrained' $\mathbf{K}$ correct?

```
#YOUR CODE HERE

con = mm.Constrainer()

#YOUR CODE HERE

f = np.zeros (6) #empty load vector
Kff, Fff = con.constrain( K, F )
print(np.shape(np.linalg.inv(Kff)))
```

Solution to Exercise (Workshop 1 - 3.2) ⌄

4. Full implementation extension bar

Having made our implementations, we now check them with two simple examples that serve as sanity checks. The first is a simple bar undergoing extension:

With $EA = 1000$, $F = 100$ and $L = 1$.

Use the code blocks below to set up and solve this problem using the classes above. The steps to follow are outlined below and short explanations/hints are given. Once you have a

solution for the horizontal displacement of the node at the right end of the bar, compare it to the analytical solution you obtained in the first half of the course.

```
mm.Node.clear()
mm.Element.clear()
```

🧩 Exercise (Workshop 1 - 4.1)

Create two nodes here. You can store them on a `list` or simply create them as two separate objects (*e.g.* `node1` and `node2`).

```
#YOUR CODE HERE
```

📋 Solution to Exercise (Workshop 1 - 4.1) ⌄

🧩 Exercise (Workshop 1 - 4.2)

Here we only have a single element, so there is no need to store it in a `list` yet. You are also going to need a `dict` defining the cross-section of the element.

```
#YOUR CODE HERE
```

📋 Solution to Exercise (Workshop 1 - 4.2) ⌄

🧩 Exercise (Workshop 1 - 4.3)

Let's define the boundary conditions. We create an instance of the `Constrainer` class to deal with prescribed displacements. Take a look at its functions and inform if Node 1 is fully fixed.

You also need to pass the load F on to Node 2. Check the member functions of `Node` to infer how that should be done.

```
#YOUR CODE HERE
```

Solution to Exercise (Workshop 1 - 4.3) ⌄

Exercise (Workshop 1 - 4.4)

Now assemble the global stiffness matrix and force vector. Since we only have one element, there is no real assembly to be performed other than getting the stiffness matrix of the single element and storing the load at Node 2 in the correct positions of $\mathbf{f}$.

```
#YOUR CODE HERE
```

Solution to Exercise (Workshop 1 - 4.4) ⌄

Exercise (Workshop 1 - 4.5)

Constrain the problem and solve for nodal displacements.

```
#YOUR CODE HERE
```

Solution to Exercise (Workshop 1 - 4.5) ⌄

Exercise (Workshop 1 - 4.6)

Finally, compare the displacement at the end of the bar with the one coming from the ODE solution. Note that since our element is already suitable for frames combining extension and bending, $\mathbf{u}$ has three entries. Which one is the entry that matters to us here? Did your solutions match? If so, that is a sign your implementation is correct. Can you use the function `full_disp` to obtain a vector of all displacements?

```
#EVENTUALLY YOUR CODE HERE
```

Solution to Exercise (Workshop 1 - 4.6) ⌄

5. Full implementation bending beam

In the first example above we tested our model under extension. But that does not really guarantee it will behave correctly in bending. That is the goal of this second sanity check. Let's solve the following problem:

Choose appropriate values yourself

When setting up and solving your model, note that we are now interested in w displacements, our load is now vertical and the cross-section property driving our deformation is now EI. Good luck!

```
mm.Node.clear()
mm.Element.clear()
```

🧩 Exercise (Workshop 1 - 5.1)

Create nodes

```
#YOUR CODE HERE
```

📋 Solution to [Exercise (Workshop 1 - 5.1)](#) ⌄

🧩 Exercise (Workshop 1 - 5.2)

Create element

```
#YOUR CODE HERE
```

📋 Solution to [Exercise (Workshop 1 - 5.2)](#) ⌄

🧩 Exercise (Workshop 1 - 5.3)

Set boundary conditions

```
#YOUR CODE HERE
```

✅ Solution to Exercise (Workshop 1 - 5.3) ⌄

🧩 Exercise (Workshop 1 - 5.4)

Assemble the system of equations.

```
#YOUR CODE HERE
```

✅ Solution to Exercise (Workshop 1 - 5.4) ⌄

🧩 Exercise (Workshop 1 - 5.5)

Constrain the problem and solve for nodal displacements

```
#YOUR CODE HERE
```

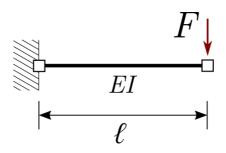## ✅ Solution to Exercise (Workshop 1 - 5.5) ⌄

🧩 Exercise (Workshop 1 - 5.6)

Check with the analytical solution

Did your solutions match? If so, your implementation is correct!

```
#EVENTUALLY YOUR CODE HERE
```

Solution to Exercise (Workshop 1 - 5.5)

Apply

Contents

> ⚠ **Attention**
>
> This page shows a preview of the assignment. Please fork and clone the assignment to work on it locally from GitHub
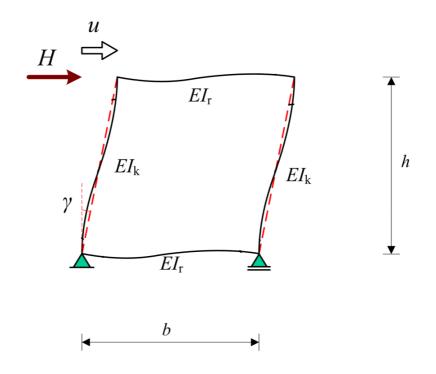
> ⓘ *Added in version v2025.1.0:* After workshop 1
>
> Solutions workshop 1 in text and downloads

In this notebook you will work on a homework assignment involving a Vierendeel frame.

Our matrix method implementation is now completely stored in a local package, consisting of three classes.

Vierendeel frame

With:

- $h = 1$
- $b = 1$
- $EI_r = 10000$
- $EI_k = 1000$
- $EA = 1 \cdot 10^{10}$
- $H = 100$

In the first half of this course last quarter, you have learned that the deformation of Vierendeel frames (an example of which is shown above) can be obtained in a simplified way by assuming the global deformation can be described by a shear beam with equivalent stiffness given by:

$$k = \frac{24}{h \left(\dfrac{h}{EI_k} + \dfrac{b}{EI_r} \right)}$$

🧩 Exercise (Workshop 1 - Apply)

Now that you have the tools to solve the original frame problem using the Matrix Method, your task in this assignment is to investigate the validity of this equivalent shear beam model.

Note that the checks only had a single element. For this model you need to obtain $\mathbf{K}$ and $\mathbf{f}$ of all elements and add them to the correct locations in the global stiffness matrix and force vector. To do that, make use of the `global_dofs` function of the Element class and the `np.ix_` Numpy utility function. (Tip: refer back to what you did in the `constrain` function).

Once you have a solution, use SymPy / Maple / pen and paper to solve a shear beam problem with the equivalent stiffness given above (It is very similar to the simple extension problem above) and compare the horizontal displacement at the point of application of H for the two models.

Investigate how the two models compare for different values of EA, ranging from very small (*e.g.* $1 \cdot 10^{-5}$) to very large (*e.g.* $1 \cdot 10^{10}$). What explains the behavior you observe?

```python
import numpy as np
import matplotlib as plt
import matrixmethod as mm
%config InlineBackend.figure_formats = ['svg']
```

```python
mm.Node.clear()
mm.Element.clear()
```

```python
#YOUR CODE HERE
```

```python
global_k #= np.zeros(YOUR CODE HERE
global_f #= np.zeros(YOUR CODE HERE

for elem in elems:
    elmat = elem#.YOUR CODE HERE
    idofs = elem#.YOUR CODE HERE

    #YOUR CODE HERE

for node in nodes:
    #YOUR CODE HERE
```

```
#YOUR CODE HERE
```

```
#provided in case you want to solve the shear beam problem using SymPy
import sympy as sym
x, k, L, H = sym.symbols('x, k, L, H')
w = sym.Function('w')

ODE_shear = #YOUR CODE HERE
```

Solution to Exercise (Workshop 1 - Apply) ⌄

Lecture 2

> ❗ ***Changed in version v2025.1.1:*** after second leture
>
> Updated lecture slides: fixed typo and removed example

During today's lesson you'll wrap up the discussion on the Matrix Method for statics and implement in code some new content. You'll be given the last theoretical details of the method, including how to consider element loads, non-zero Dirichlet boundary conditions and postprocessing for support reactions and element fields.

This lecture is given by Iuri Rocha.

This book shows the full content of the first lecture. The slides of the lecture have the same content and are available ⬇ here

Element loads

Contents

- Force-displacement relations using differential equations

- Combine elements

- Force-displacement relations using conservation of work

- Example

As the matrix method is a discrete approach, nodal loads were treated with ease. However, what to do with continuous loads or loads which are not applied at the nodes?

> **Learning objective**
>
> You'll look into how to model element loads using differential equations and conservation of work and how these are combined in the matrix formulation.

Force-displacement relations using differential equations

As before, we can derive the force-displacement relations of a single extension element. However, now let's include the loads, for example a continuous load q
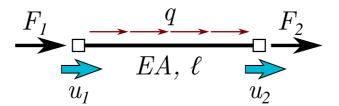

Fig. 12 Single extension element with distributed load q

The same approach is used as in Recap differential equations for structures. This results in:

- $C_1 = \dfrac{q\ell}{2EA} + \dfrac{u_2 - u_1}{\ell}$

- $C_2 = u_1$

The continuous distributions for the displacement and section force can be evaluated too:

- $u(x) = \dfrac{q}{2EA} \left(\ell x - x^2 \right) + u_1 \left(1 - \dfrac{x}{\ell} \right) + u_2 \dfrac{x}{\ell}$

- $N(x) = \dfrac{q}{2} \left(\ell - 2x \right) - \dfrac{EA}{\ell} u_1 + \dfrac{EA}{\ell} u_2$

These are extended results in comparison to [before](#)

Combine elements

As before, we can glue elements together by applying nodal equilibrium:

This leads to:

- $F_1 = -N = \dfrac{EA}{\ell} u_1 - \dfrac{EA}{\ell} u_2 - \dfrac{q\ell}{2}$

- $F_2 = N = -\dfrac{EA}{\ell} u_1 + \dfrac{EA}{\ell} u_2 - \dfrac{q\ell}{2}$

or in matrix notation:

$$
\frac{EA}{\ell} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} - \begin{bmatrix} \dfrac{q\ell}{2} \\ \dfrac{q\ell}{2} \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix}
$$

Effectively, we converted the continuous load to an equivalent nodal load.

This influences the nodal equilibrium (in the global coordinate system) too:

$$
-\sum_e \mathbf{f}^e + \mathbf{f}_{\text{nodal}} = \mathbf{0}
$$

$$
-\sum_e \left(\mathbf{K}^e \mathbf{u}^e - \mathbf{f}_{\text{eq}}^e \right) + \mathbf{f}_{\text{nodal}} = \mathbf{0}
$$

$$
\sum_e \mathbf{f}^e = \mathbf{f}_{\text{nodal}} + \sum_e \mathbf{f}_{\text{eq}}^e
$$

This means that we can calculate all the equivalent nodal loads separately and add them to the nodal loads. Please note that all these forces should be in the global coordinate system:

$\mathbf{f}_{\text{eq}} = \mathbf{T}^{\text{T}} \bar{\mathbf{f}}_{\text{eq}}$

Force-displacement relations using conservation of work

Point load

Let's consider another example in which the application of the differential equations are not trivial: we'll introduce a discontinuous force in the form of a point load halfway the element. We'll compare this with the same element loaded by vertical forces and bending moments at the ends:
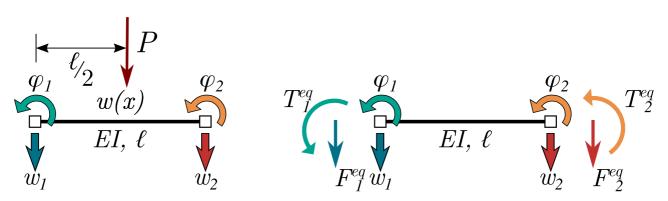

Fig. 13 Single extension element with point load P halfway in comparison to element loaded by vertical forces and bending moments at the ends

The conservation of work states that the work done by the force P should be equal to the force done by the forces F_1^{eq}, F_2^{eq}, T_1^{eq} and T_2^{eq}: $W_P = W_{eq}$

To ease the calculation, we'll split the displacement in four cubic shape functions (cubic is justified because with $q = 0$ a cubic displacement function is found). Each shape function will have a displacement of 1 in direction of each of the four edge forces (w_1, φ_1, w_2 and φ_2):
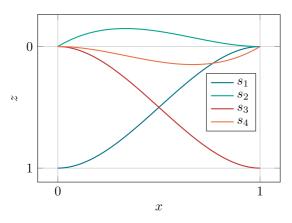

Fig. 14 Four shape functions

The shape functions have the following function:

$$w(x) = \left(\frac{2x^3}{\ell^3} - \frac{3x^2}{\ell^2} + 1 \right) w_1 + \left(-\frac{x^3}{\ell^2} + \frac{2x^2}{\ell} - x \right) \varphi_1 + \left(-\frac{2x^3}{\ell^3} + \frac{3x^2}{\ell^2} \right) w_2 +$$

$$\underbrace{\qquad\qquad\qquad}_{s_1} \qquad \underbrace{\qquad\qquad\qquad}_{s_2} \qquad \underbrace{\qquad\qquad\qquad}_{s_3}$$

Consequently, the work W_{eq} can be splitted in four independent terms easing the calculation, which can be compared to the same terms in W_P.

The work performed by the edge forces equals:

$$W_{eq} = F_1^{eq} w_1 + T_1^{eq} \varphi_1 + F_2^{eq} w_2 + T_2^{eq} \varphi_2$$

The work performed by P (under the same displacement) is:

$$W_P = P\,w \left(\frac{\ell}{2} \right) = P\,s_1 \left(\frac{\ell}{2} \right) w_1 + P\,s_2 \left(\frac{\ell}{2} \right) \varphi_1 + P\,s_3 \left(\frac{\ell}{2} \right) w_2 + P\,s_4 \left(\frac{\ell}{2} \right) \varphi$$

Enforcing $W_F = W_q$ and isolating terms gives:

$$\mathbf{f}_{eq} = \begin{bmatrix} F_1^{eq} \\ T_1^{eq} \\ F_2^{eq} \\ T_2^{eq} \end{bmatrix} = \begin{bmatrix} \dfrac{P}{2} \\ -\dfrac{P\ell}{8} \\ \dfrac{P}{2} \\ \dfrac{P\ell}{8} \end{bmatrix}$$

In the local coordinate system.

Distributed load

The same can be done for a distributed load:
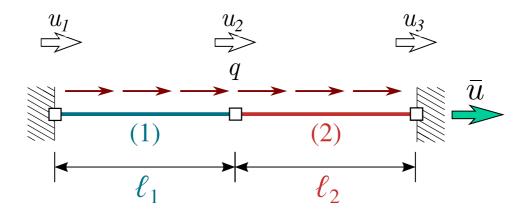
$$W_q = \int_\ell q\,w(x)\mathrm{d}x = \int_\ell q\,s_1(x)\mathrm{d}x\,w_1 + \int_\ell q\,s_2(x)\mathrm{d}x\,\varphi_1 + \int_\ell q\,s_3(x)\mathrm{d}x\,w_2 + \int_\ell q\,s_4$$

Leading to:

$$
\mathbf{f}_{\text{eq}} =
\begin{bmatrix}
F_1^{\text{eq}} \\
T_1^{\text{eq}} \\
F_2^{\text{eq}} \\
T_2^{\text{eq}}
\end{bmatrix}
=
\begin{bmatrix}
\dfrac{q\ell}{2} \\
-\dfrac{q\ell^2}{12} \\
\dfrac{q\ell}{2} \\
\dfrac{q\ell^2}{12}
\end{bmatrix}
$$

Example

Let us use what we have just learned on a simple example:
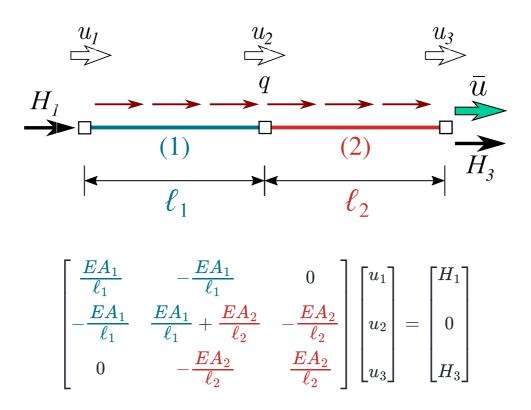
This example has the same two-element bar model as in [Setting up global equations directly](#), so the unconstrained stiffness matrix is unchanged:

$$
\begin{bmatrix}
\dfrac{EA_1}{\ell_1} & -\dfrac{EA_1}{\ell_1} & 0 \\
-\dfrac{EA_1}{\ell_1} & \dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2} & -\dfrac{EA_2}{\ell_2} \\
0 & -\dfrac{EA_2}{\ell_2} & \dfrac{EA_2}{\ell_2}
\end{bmatrix}
$$

Both supports will introduce a support reaction in the form of a Neumann boundary condition:

$$
\begin{bmatrix}
\dfrac{EA_1}{\ell_1} & -\dfrac{EA_1}{\ell_1} & 0 \\[2ex]
-\dfrac{EA_1}{\ell_1} & \dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2} & -\dfrac{EA_2}{\ell_2} \\[2ex]
0 & -\dfrac{EA_2}{\ell_2} & \dfrac{EA_2}{\ell_2}
\end{bmatrix}
\begin{bmatrix} u_1 \\[2ex] u_2 \\[2ex] u_3 \end{bmatrix}
=
\begin{bmatrix} H_1 \\[2ex] 0 \\[2ex] H_3 \end{bmatrix}
$$

The distributed load can be added directly as equivalent load vector $\mathbf{f}_{\text{eq}}$ to the global force vector as the local coordinate system aligns with the global coordinate system. Doing so for both elements leads to:

$$
\begin{bmatrix}
\dfrac{EA_1}{\ell_1} & -\dfrac{EA_1}{\ell_1} & 0 \\[2ex]
-\dfrac{EA_1}{\ell_1} & \dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2} & -\dfrac{EA_2}{\ell_2} \\[2ex]
0 & -\dfrac{EA_2}{\ell_2} & \dfrac{EA_2}{\ell_2}
\end{bmatrix}
\begin{bmatrix} u_1 \\[2ex] u_2 \\[2ex] u_3 \end{bmatrix}
=
\begin{bmatrix} H_1 + \dfrac{q\ell_1}{2} \\[2ex] \dfrac{q\ell_1}{2} + \dfrac{q\ell_2}{2} \\[2ex] H_3 + \dfrac{q\ell_2}{2} \end{bmatrix}
$$

Non-zero Dirichlet boundary conditions

Contents

In [Setting up global equations directly](#) we defined how to handle Dirichlet boundary conditions which enforce a displacement of 0 by striking the corresponding row/column in the final system. However, this approach doesn't work with nonzero displacements.

> **Learning objective**
>
> You'll look into how to model non-zero Dirichlet boundary conditions using static condensation and a size-preserving approach

Static condensation

To apply nonzero constraints we can partition the system:

$$\begin{bmatrix} \mathbf{K}_{ff} & \mathbf{K}_{fc} \\ \mathbf{K}_{cf} & \mathbf{K}_{cc} \end{bmatrix} \begin{bmatrix} \mathbf{u}_f \\ \mathbf{u}_c \end{bmatrix} = \begin{bmatrix} \mathbf{f}_f \\ \mathbf{f}_c \end{bmatrix}$$

With subscript f for the free degrees of freedom and the subscript c for the constraint degrees of freedom.

The unknown free displacements can now be solved for:

$$\mathbf{K}_{ff}\mathbf{u}_f + \mathbf{K}_{fc}\mathbf{u}_c = \mathbf{f}_f$$
$$\mathbf{u}_f = \mathbf{K}_{ff}^{-1}\left(\mathbf{f}_f - \mathbf{K}_{fc}\mathbf{u}_c\right)$$

Furthermore, this allows us to solve for the support reactions, which are, among the other terms from nodal and equivalent loads, part of $\mathbf{f}_c$:

$$\mathbf{f}_c = \mathbf{K}_{cf}\mathbf{u}_f + \mathbf{K}_{cc}\mathbf{u}_c$$

However, this approach can be annoying to code because reordering the system costs computation time and the gains when inverting the stiffness matrix are very limited.

Size-preserving approach

An alternative approach is to modify the equations such that the system is not reordered and the size is preserved. This can be done by replacing the line which includes the non-zero Dirichlet boundary condition with the boundary conditions itself, i.e. for a system of 3 equations:

$$\begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix}$$

The boundary condition $u_2 = \Delta_2$ can be inserted:
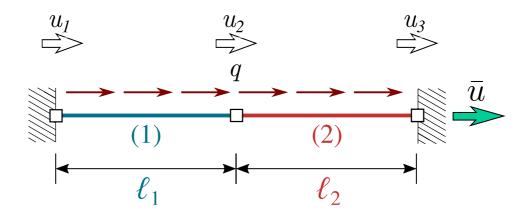
$$\begin{bmatrix} K_{11} & K_{12} & K_{13} \\ 0 & 1 & 0 \\ K_{31} & K_{32} & K_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} f_1 \\ \Delta_2 \\ f_3 \end{bmatrix}$$

Which can be further simplified to:

$$\begin{bmatrix} K_{11} & 0 & K_{13} \\ 0 & 1 & 0 \\ K_{31} & 0 & K_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} f_1 - K_{12}\,\Delta_2 \\ \Delta_2 \\ f_3 - K_{32}\,\Delta_2 \end{bmatrix}$$

Example

Let us use what we have just learned on a simple example:

This example has the same two-element bar model as in [Setting up global equations directly](#) and [Element loads](#), so the stiffness matrix is unchanged, including the support reactions:

$$\begin{bmatrix} \dfrac{EA_1}{\ell_1} & -\dfrac{EA_1}{\ell_1} & 0 \\[2ex] -\dfrac{EA_1}{\ell_1} & \dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2} & -\dfrac{EA_2}{\ell_2} \\[2ex] 0 & -\dfrac{EA_2}{\ell_2} & \dfrac{EA_2}{\ell_2} \end{bmatrix} \begin{bmatrix} u_1 \\[2ex] u_2 \\[2ex] u_3 \end{bmatrix} = \begin{bmatrix} H_1 + \dfrac{q\ell_1}{2} \\[2ex] \dfrac{q\ell_1}{2} + \dfrac{q\ell_2}{2} \\[2ex] H_3 + \dfrac{q\ell_2}{2} \end{bmatrix}$$

Static condensation

Our constrained degrees of freedom are u_1 and u_3, while u_2 is free:

- $\mathbf{u}_{\mathrm{f}} = u_2$

- $\mathbf{u}_{\mathrm{c}} = \begin{bmatrix} u_1 \\ u_3 \end{bmatrix}$

This gives:

- $\mathbf{f}_{\mathrm{f}} = \dfrac{q\ell_1}{2} + \dfrac{q\ell_2}{2}$

- $\mathbf{f}_{\mathrm{c}} = \begin{bmatrix} H_1 + \dfrac{q\ell_1}{2} \\[2ex] H_3 + \dfrac{q\ell_2}{2} \end{bmatrix}$

Partitioning the stiffness matrix leads to:

$$
\begin{bmatrix}
\dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2} & -\dfrac{EA_1}{\ell_1} & -\dfrac{EA_2}{\ell_2} \\
-\dfrac{EA_1}{\ell_1} & \dfrac{EA_1}{\ell_1} & 0 \\
-\dfrac{EA_2}{\ell_2} & 0 & \dfrac{EA_2}{\ell_2}
\end{bmatrix}
\begin{bmatrix} u_2 \\ 0 \\ \bar{u} \end{bmatrix}
=
\begin{bmatrix}
\dfrac{q\ell_1}{2} + \dfrac{q\ell_2}{2} \\
H_1 + \dfrac{q\ell_1}{2} \\
H_3 + \dfrac{q\ell_2}{2}
\end{bmatrix}
$$

with:

- $\mathbf{K}_{\mathrm{ff}} = \dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2}$

- $\mathbf{K}_{\mathrm{fc}} = \begin{bmatrix} -\dfrac{EA_1}{\ell_1} & -\dfrac{EA_2}{\ell_2} \end{bmatrix}$

- $\mathbf{K}_{\mathrm{cf}} = \begin{bmatrix} -\dfrac{EA_1}{\ell_1} \\ -\dfrac{EA_2}{\ell_2} \end{bmatrix}$

- $\mathbf{K}_{\mathrm{cc}} = \begin{bmatrix} \dfrac{EA_1}{\ell_1} & 0 \\ 0 & \dfrac{EA_2}{\ell_2} \end{bmatrix}$

Now, the unknown $\mathbf{u}_\mathrm{f}$ (including only u_2) can be solved for using the equations provided in Static condensation.

Size-preserving approach

Now let's apply the alternative approach. First, let's set the $u_1 = 0$:

$$
\begin{bmatrix}
1 & 0 & 0 \\
0 & \dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2} & -\dfrac{EA_2}{\ell_2} \\
0 & -\dfrac{EA_2}{\ell_2} & \dfrac{EA_2}{\ell_2}
\end{bmatrix}
\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}
=
\begin{bmatrix}
0 \\
\dfrac{q\ell_1}{2} + \dfrac{q\ell_2}{2} \\
H_3 + \dfrac{q\ell_2}{2}
\end{bmatrix}
$$

No terms are added to the force vector as this enforced displacement is 0.

Let's continue with the displacement $u_3 = \bar{u}$:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \dfrac{EA_1}{\ell_1} + \dfrac{EA_2}{\ell_2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ \dfrac{q\ell_1}{2} + \dfrac{q\ell_2}{2} - \dfrac{EA_2}{\ell_2}\bar{u} \\ \bar{u} \end{bmatrix}$$

Now, a term is added to the force vector, as $\mathbf{K}_{23}$ wasn't equal to 0 (and now it is).

This matrix equation can now be solved for $\mathbf{u}$.
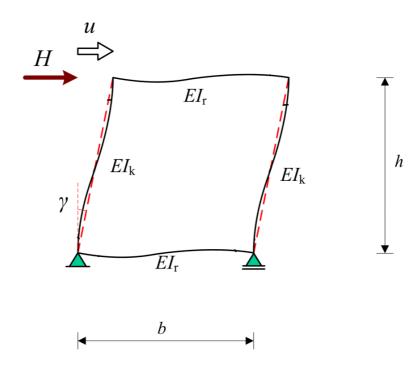
Postprocessing for continuum fields

Up until now, we've only looked at discrete results: nodal displacement and support reactions. However, these results can be used to obtain the continuum field.

> **Learning objective**
>
> You'll look into how to postprocess discrete results to obtain continuum results.

After solving for discrete nodal displacements, we can use the expressions derived in Force-displacement relations single extension element and Element loads to obtain continuous results. Remember than the nodal displacements are in the global coordinate system, which needs to be converted back into the local coordinate systems ($\bar{\mathbf{u}}^e = \mathbf{T}\mathbf{u}^e$) to use the derived expressions.
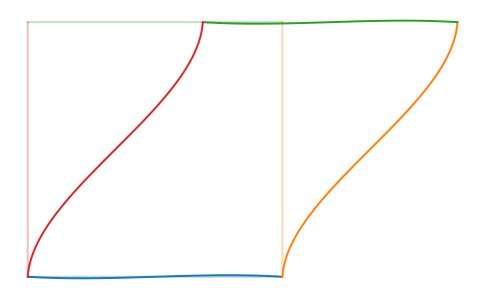
If you want to create a figure which combines the internal forces / displacements of multiple elements, you need the results in the global coordinate system again. In the provided package, this is implemented with a boolean operation `global_c` in the class `elements.py` function `plot_moment_diagram` and `plot_displaced`. For example, the frame treated in Workshop 1 - Apply:

gives the following displaced shape:

Displaced structure

Finite element method vs. Matrix Method

Contents

> This page reuses content from MUDE Teachers and the Student Army from Delft University of Technology [MUDETatSAfDUoTechnology24a]. 66Find out more here.

You've seen the finite element method before, which could be used to solve similar problems. But what are the differences?

> **Learning objective**
>
> We'll investigate the differences and equivalence between solving structures with the finite element method and the matrix method.

Equivalence with matrix method

Although the two method can give the same results, the methods are different.

The matrix method solves the strong from of the differential equation, as derived in Force-displacement relations single extension element. The finite element method solves the weak form by multiplying the strong form by a test function [MUDETatSAfDUoTechnology24b]. In doing so, the choice for the shape function of the test-functions and approximate solution matters. The two methods end up with the same solution if the "approximation" assumed by FEM (linear shape functions for extension, cubic for bending) turn out to be the exact ODE solution.

In terms of global and local coordinate systems, there's an additional difference. Where the matrix method solves the nodal displacements and support reactions globally, using locally derived force-displacement relations. On the contrary, the finite element method solves the weak form globally with shape functions defined globally.

Finally, the matrix method has limitations. It turns out to be impossible to glue element through equilibrium for twodimensional elements. Furthermore, exact solution for the differential equations, required for defining the local stiffness matrix, generally do not exist for twodimensional elements.

Table 4 Equivalence finite element- and matrix method

Finite element method	Matrix method
Solves weak form	Solves the strong form
Solves everything globally	Solves discrete solution globally, with locally derived equations
Generally applicable to differential equations	Only applicable for 1D elements

Example with finite element method

Let's consider the examples from Recap displacement method:
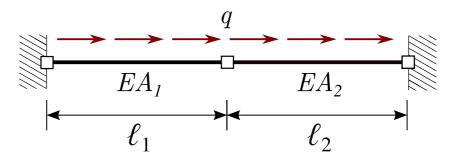
Fig. 15 Statically indeterminate extension bar

We'll apply the finite element method by using the matrix implementation from MUDE [MUDETatSAfDUoTechnology24a]. We use linear shape functions: $N_a(x) = \dfrac{x_b - x}{x_b - x_a} = \dfrac{x_b - x}{\Delta x}$ and $N_b(x) = \dfrac{x - x_a}{x_b - x_a} = \dfrac{x - x_a}{\Delta x}$. For this specific example, this matches the linear normal force distribution, leading to similar results:

▶ Show code cell content

▶ Show code cell content

▶ Show code cell source

```
x, u = simulate(2)
print(u[1])
```

```
0.01125
```

Example with matrix method

Applying the matrix method (including implementations you'll implement during Workshop 2) yields the same result:

▶ Show code cell content

▶ Show code cell source

```
print(u_free[0])
```

```
0.01125
```
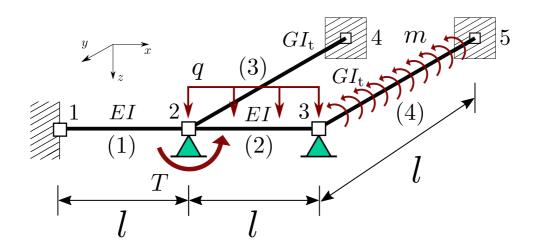
Which is the same result (not true in general).

Example - 3D frame with torsion

Contents

This page shows an example for a threedimensional frame (with onedimensional elements), loaded in torsion. This requires defining a new element, but the approach is identical to what we've seen before.

The following numerical values can be used:

- $EI = 1000$ kNm^2
- $GI_t = 800$ kNm^2
- $\ell = 2$ m
- $T = 4$ kNm

- $q = 6$ kN/m
- $m = 2$ kNm/m

Defining new element

In this problem, all nodes only rotate around the y-axis; there's no translation or rotation in another direction. For element (3) and (4) this introduces torsion in the elements. The model for torsional elements has been treated before in Week 2.2, Unit 2, Lecture 6 of this course [HansWfDUoTechnology24]:
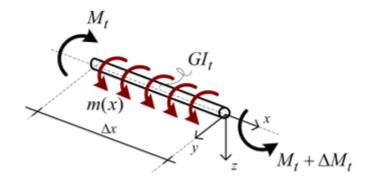
Fig. 16 Torsional element from Hans Welleman from Delft University of Technology [HansWfDUoTechnology24]

he differential equation for the this element can be derived leading to:

- Kinematic relations: $\theta = \dfrac{\mathrm{d}\varphi_x}{\mathrm{d}x}$
- Constitutive relation: $M_t = GI_t\,\theta$
- Equilibrium relations: $\dfrac{\mathrm{d}M_t}{\mathrm{d}x} = -m$

These relations can be combined into one second order differential equation:

$$GI_t\frac{\mathrm{d}^2\varphi_x}{\mathrm{d}x^2} = -m$$

This looks identical (with m for q, GI_t for EA and φ for u) to our results from the extension element, therefore, we can directly write down the stiffness matrix and (equivalent) force vector directly:

$$\frac{GI_t}{\ell}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} \varphi_1 \\ \varphi_2 \end{bmatrix} = \begin{bmatrix} T_1 \\ T_2 \end{bmatrix} + \begin{bmatrix} \dfrac{m\,\ell}{2} \\ \dfrac{m\,\ell}{2} \end{bmatrix}$$

As our torsional elements (3) and (4) are identical, we get:

$$\mathbf{K}^{(3)} = \mathbf{K}^{(4)} = \begin{bmatrix} 400 & -400 \\ -400 & 400 \end{bmatrix}$$

And for element (4) an additional equivalent force vector due to the element load:

$$\mathbf{f}_{\text{eq}}^{(4)} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

Reduce bending element for tractability

Element (1) and (2) will bend in this case. However, no forces act in the local x-direction. This allows us to reduce the element defined in Local stiffness matrix Euler-Bernoulli element:

$$\mathbf{K}_{\text{bending}}^{(e)} = \begin{bmatrix} \dfrac{12EI}{\ell^3} & -\dfrac{6EI}{\ell^2} & -\dfrac{12EI}{\ell^3} & -\dfrac{6EI}{\ell^2} \\ -\dfrac{6EI}{\ell^2} & \dfrac{4EI}{\ell} & \dfrac{6EI}{\ell^2} & \dfrac{2EI}{\ell} \\ -\dfrac{12EI}{\ell^3} & \dfrac{6EI}{\ell^2} & \dfrac{12EI}{\ell^3} & \dfrac{6EI}{\ell^2} \\ -\dfrac{6EI}{\ell^2} & \dfrac{2EI}{\ell} & \dfrac{6EI}{\ell^2} & \dfrac{4EI}{\ell} \end{bmatrix}$$

This gives for our numerical values:

$$\mathbf{K}^{(1)} = \mathbf{K}^{(2)} = \begin{bmatrix} 2000 & 1000 \\ 1000 & 2000 \end{bmatrix}$$

$$\mathbf{f}_{\text{eq}}^{(2)} = \begin{bmatrix} -2 \\ 2 \end{bmatrix}$$

Identify degrees of freedom

As mentioned before, all nodes only rotate around the y-axis; there's no translation or rotation in another direction. Therefore, the degrees of freedom are:

$$\begin{bmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \varphi_4 \\ \varphi_5 \end{bmatrix}$$

With φ being specifically φ_y.

Assemble stiffness, element by element

As all elements use the same orientation for the degrees of freedom, no transformation are required. Note that this requires:

- Defining element (1) from 1 to 2
- Defining element (2) from 2 to 3
- Defining element (3) from 2 to 4
- Defining element (4) from 3 to 5

Or all the other way around. Otherwise, the rotations at node 2 and 3 do not match.

Element (1)

The first element links the first and second nodal displacement with the first and second nodal forces:

$$\mathbf{K} = \begin{bmatrix} 2000 & 1000 & 0 & 0 & 0 \\ 1000 & 2000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Element (2)

The second element links the second and third nodal displacement with the second and third nodal forces:

$$\mathbf{K} = \begin{bmatrix} 2000 & 1000 & 0 & 0 & 0 \\ 1000 & 4000 & 1000 & 0 & 0 \\ 0 & 1000 & 2000 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Element (3)

The third element links the second and fourth nodal displacement with the second and fourth nodal forces:

$$\mathbf{K} = \begin{bmatrix} 2000 & 1000 & 0 & 0 & 0 \\ 1000 & 4400 & 1000 & -400 & 0 \\ 0 & 1000 & 2000 & 0 & 0 \\ 0 & -400 & 0 & 400 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Element (4)

The fourth element links the third and fifth nodal displacement with the third and fifth nodal forces:

$$\mathbf{K} = \begin{bmatrix} 2000 & 1000 & 0 & 0 & 0 \\ 1000 & 4400 & 1000 & -400 & 0 \\ 0 & 1000 & 2400 & 0 & -400 \\ 0 & -400 & 0 & 400 & 0 \\ 0 & 0 & -400 & 0 & 400 \end{bmatrix}$$

Apply external loads

Nodal loads

Now, let's add the external loads, starting with the nodal load at 2:

$$\mathbf{f} = \begin{bmatrix} 0 \\ 4 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Equivalent nodal loads

Let's add the equivalent nodal loads from the continuous elements loads too:

$$\mathbf{f} = \begin{bmatrix} 0 \\ 2 \\ 4 \\ 0 \\ 2 \end{bmatrix}$$

Boundary loads

And finally, let's add the forces from the Neumann boundary conditions:

$$\mathbf{f} = \begin{bmatrix} M_{t,1} \\ 2 \\ 4 \\ M_{t,4} \\ 2 + M_{t,5} \end{bmatrix}$$

Complete system of equations

Now, we found the complete system of equations:

$$\begin{bmatrix} 2000 & 1000 & 0 & 0 & 0 \\ 1000 & 4400 & 1000 & -400 & 0 \\ 0 & 1000 & 2400 & 0 & -400 \\ 0 & -400 & 0 & 400 & 0 \\ 0 & 0 & -400 & 0 & 400 \end{bmatrix} \begin{bmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \varphi_4 \\ \varphi_5 \end{bmatrix} = \begin{bmatrix} M_{t,1} \\ 2 \\ 4 \\ M_{t,4} \\ 2 + M_{t,5} \end{bmatrix}$$

Solving boundary conditions

As we've no non-zero boundary conditions, we can apply the row-striking technique of lecture 1, which leads to:

$$\begin{bmatrix} 4400 & 1000 \\ 1000 & 2400 \end{bmatrix} \begin{bmatrix} \varphi_2 \\ \varphi_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

Solving this system of equations gives:

- $\varphi_2 \approx 1 \cdot 10^{-4}$
- $\varphi_3 \approx 1.6 \cdot 10^{-3}$

The support reactions can be found by inserting these into our original system of equations and solving for the rows containing the support reactions:

$$\begin{bmatrix} 2000 & 1000 & 0 & 0 & 0 \\ 0 & -400 & 0 & 400 & 0 \\ 0 & 0 & -400 & 0 & 400 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \cdot 10^{-4} \\ 1.6 \cdot 10^{-3} \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} M_1 \\ M_{t,4} \\ 2 + M_{t,5} \end{bmatrix}$$

Note that to calculate $M_{t,4}$ the element load should be taken into account! This gives:

- $M_1 \approx 0.1$ kNm
- $M_{t,4} \approx -0.033$ kNm
- $M_{t,5} \approx -2.7$ kNm

Postprocessing moments element (1)

The continuum displacement field of element (1) can be described by the shape function:

$$w\left(x\right) = \left(-\frac{x^3}{\ell^2} + \frac{x^2}{\ell}\right)\varphi_2$$

This gives:

$$\varphi\left(x\right) = -\frac{\mathrm{d}w\left(x\right)}{\mathrm{d}x} = \left(\frac{3x^2}{\ell^2} - \frac{2x}{\ell}\right)\varphi_2$$

$$\kappa\left(x\right) = \frac{\mathrm{d}\varphi\left(x\right)}{\mathrm{d}x} = \left(\frac{6x}{\ell^2} - \frac{2}{\ell}\right)\varphi_2$$

$$M\left(x\right) = EI\kappa = EI\left(\frac{6x}{\ell^2} - \frac{2}{\ell}\right)\varphi_2$$

This is a linear distribution, with values at $x = 0$ and $x = \ell$:

- $M\left(0\right) \approx -0.1$ kNm
- $M\left(2\right) \approx 0.2 kNm$

The value at $x = 0$ has indeed the same absolute value as the support reactions. The sign is different because M_1 is defined in the global coordinate system and $M\left(0\right)$ is defined from our agreements on positive internal moments (leading to positive stresses at the positive z-side.)

Workshop 2

> ⓘ **_Added in version v2025.2.0:_** After workshop 2
>
> Solutions workshop 2 in downloads

> ❗ **Attention**
>
> This pages shows a preview of the assignment including its solution. Please fork and clone the assignment to work on it locally from GitHub

During today's workshop you'll extend your implementation of the matrix method.

Implementation

Contents

> ❗ **Attention**
>
> This page shows a preview of the assignment. Please fork and clone the assignment to work on it locally from [GitHub](#)

> ❗ *Added in version v2025.2.0:* After workshop 2
>
> Solutions workshop 2 in text and downloads

In this notebook you will continue to implement the matrix method and check it with some sanity checks.

> 🧩 **Exercise (0)**
>
> Check whether your implementation of last week was correct using the provided solution

```python
import numpy as np
import matplotlib as plt
import matrixmethod as mm
%config InlineBackend.figure_formats = ['svg']

%load_ext autoreload
%autoreload 2
```

1. The Node class

The `Node` class from last week is unchanged and complete

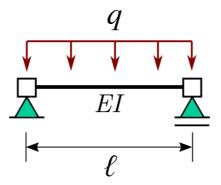# 2. The Element class

The implementation is incomplete:

- The function `add_distributed_load` should compute the equivalent load vector for a constant load q in $\bar{x}$ and $\bar{z}$ direction (we'll ignore a distributed moment load now) and moves those loads to the nodes belonging to the element. Remember to use the `add_load` function of the `Node` class to store the equivalent loads (remember we have two nodes per element). Also keep local/global transformations in mind and store `self.q = q` for later use;

- The function `bending_moments` receives the nodal displacements of the element in the global coordinate system (`u_global`) and uses it to compute the value of the bending moment at `num_points` equally-spaced points along the element length. Keep local/global transformations in mind and use the ODE approach in SymPy / Maple / pen and paper to compute an expression for M. Do the same for for w in the function `full_displacement`.

> 🧩 **Exercise (Workshop 2 - 2.1)**
>
> Add the missing pieces to the code, before you perform the checks below.

> 📑 **Solution to Exercise (Workshop 2 - 2.1)**   ⌄

Having made your implementations, it is now time to verify the first addition of your code with a simple sanity check. We would like to solve the following simply-supported beam:

Choose appropriate values yourself.

Exercise (Workshop 2 - 2.2)

Use the code blocks below to set up this problem. After you've added the load, print the element using `print(YOUR ELEMENT)`. Do the shown values for the nodal loads correspond with what you'd expect?

```
#YOUR CODE HERE
```

```
print(#YOUR ELEMENT HERE
```

Solution to Exercise (Workshop 2 - 2.2) ⌄

Exercise (Workshop 2 - 2.3)

Now solve the nodal displacements. Once you are done, compare the rotation at the right end of the beam. Does it match a solution you already know?

```
#YOUR CODE HERE
```

Solution to Exercise (Workshop 2 - 2.3) ⌄

Exercise (Workshop 2 - 2.4)

Calculate the bending moment at midspan and plot the moment distribution using `plot_moment_diagram`. Do the values and shape match with what you'd expect?

```
u_elem = con.full_disp(#YOUR CODE HERE)
#YOUR CODE HERE
```

Solution to Exercise (Workshop 2 - 2.4) ⌄

Calculate the deflection at midspan and plot the deflected structure using `plot_displaced`. Do the values and shape match with what you'd expect?

```
#YOUR CODE HERE
```

📋 **Solution to Exercise (Workshop 2 - 2.5)** ⌄

3. The Constrainer class

We're going to expand our Constrainer class, but the implementation is incomplete:
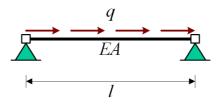
- The constrainer class should be able to handle non-zero boundary conditions too. `constrain` should be adapted to do so + the docstring of the class itself. Furthermore, the assert statement of `fix_dof` should be removed.

- The function `support_reactions` is incomplete. Since the constrainer is always first going to get `constrain` called, here we already have access to `self.free_dofs`. Together with `self.cons_dofs`, you should have all you need to compute reactions. Note that `f` is also passed as argument. Make sure you take into account the contribution of equivalent element loads that go directly into the supports without deforming the structure.

⁂ **Exercise (Workshop 2 - 3.1)**

Add the missing pieces to the code and docstring, before you perform the checks below.

📋 **Solution to Exercise (Workshop 2 - 3.1)** ⌄

We're going to verify our implementation. Therefore, we're going to solve an extension bar, supported at both ends, with a load q.

$$q$$
$$EA$$
$$l$$

Choose appropriate values yourself.

🧩 **Exercise (Workshop 2 - 3.2)**
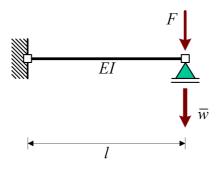
Can you say on beforehand what will be the displacements? And what will be the support reactions?

Use the code blocks below to set up and solve this problem and check the required quantities to make sure your implementation is correct.

```
#YOUR CODE HERE
```

📋 **Solution to Exercise (Workshop 2 - 3.2)**   ⌄

Again, we're going to verify our implementation. Therefore, we're going solve a beam, with a load F and support displacement $\bar{w}$ for the right support.

$$F$$
$$EI$$
$$\bar{w}$$
$$l$$

Choose appropriate values yourself.

🧩 Exercise (Workshop 2 - 3.3)

Use the code blocks below to set up and solve this problem and check the required quantities to make sure your implementation is correct.

```
#YOUR CODE HERE
```

📋 Solution to Exercise (Workshop 2 - 3.3)   ⌄

Apply

Contents

- Two-element frame

> **⚠ Attention**
>
> This page shows a preview of the assignment. Please fork and clone the assignment to work on it locally from GitHub

> **❶ Added in version v2025.2.0:** After workshop 2
>
> Solutions workshop 2 in text and downloads

In this notebook you will solve a 2-element frame at the end of the notebook.
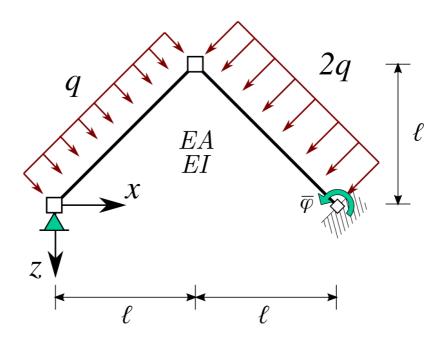
Our matrix method implementation is now completely stored in a local package, consisting of three classes.

Two-element frame

With:

- $EI = 1500$
- $EA = 1000$
- $q = 9$
- $L = 5$
- $\bar{\varphi} = 0.15$

🧩 **Exercise (Workshop 2 - Apply)**

The final example for the workshops is the two-element frame above. Here you should make use of all the new code you implemented:

- Set up the problem and compute a solution for `u_free`. Remember to consider the prescribed horizontal displacement $\bar{u}$ at the right end of the structure.
- Compute and plot bending moment lines for both elements (in the local and global coordinate systems)
- Compute reactions at both supports

```
import numpy as np
import matplotlib as plt
import matrixmethod as mm
%config InlineBackend.figure_formats = ['svg']
```

```
#YOUR CODE HERE
```

```
for elem in elems:
    u_elem = con.full_disp(#YOUR CODE HERE)[#YOUR CODE HERE.global_dofs()]
    elem.plot_displaced #YOUR CODE HERE
```

💡 **Hint**   ⌄

✅ **Solution to Exercise (Workshop 2 - Apply)**   ⌄

Additional assignments

> ⓘ **Added in version v2025.2.0:** After workshop 2
>
> Solutions additional assignments in downloads

> ⚠ **Attention**
>
> This pages shows a preview of the assignments including their solution. Please fork and clone the assignment to work on it locally from GitHub

Additional assignments are provided to extend your implementation of the matrix method and apply it to other structures.

Beam
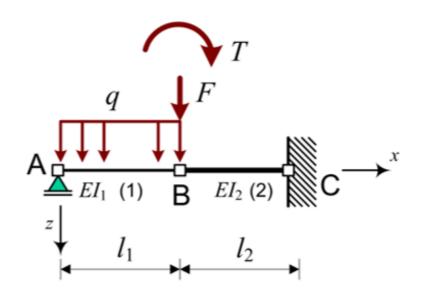
> ⚠️ **Attention**
>
> This page shows a preview of the assignment. Please fork and clone the assignment to work on it locally from GitHub

> ❗ *Added in version v2025.2.0:* After workshop 2
>
> Solutions additional assignments in text and downloads

Given is the following beam [HansWfDUoTechnology22]:

With:

- $l_1 = 5.5$
- $l_2 = 5.0$
- $EI_1 = 5000$
- $EI_2 = 8000$
- $q = 6$
- $F = 40$
- $T = 50$

Exercise (Beam)

Solve this problem.

```python
import numpy as np
import matplotlib as plt
import matrixmethod as mm
%config InlineBackend.figure_formats = ['svg']
```

```python
#YOUR_CODE_HERE
```

Solution to Exercise (Beam) ⌄

Kinked beam
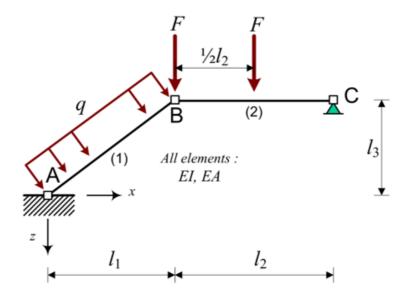
> **ℹ Added in version v2025.2.0:** After workshop 2
>
> Solutions additional assignments in text and downloads

Given is the following beam [HansWfDUoTechnology22]:

With:

- $l_1 = 4$
- $l_2 = 5$
- $l_3 = 3$
- $EI = 5000$

- $EA = 15000$
- $q = 6$
- $F = 40$

> **🧩 Exercise (Kinked beam)**
>
> Solve this problem.
>
> ```python
> import numpy as np
> import matplotlib as plt
> import matrixmethod as mm
> %config InlineBackend.figure_formats = ['svg']
> ```
>
> ```python
> #YOUR_CODE_HERE
> ```

Solution to Exercise (Kinked beam)   ⌄

Frame
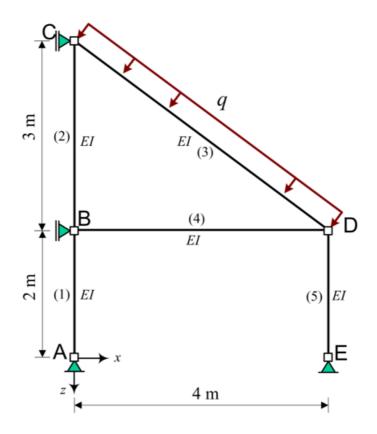
> ⚠ *Added in version v2025.2.0:* After workshop 2
>
> Solutions additional assignments in text and downloads

Given is the following beam [HansWfDUoTechnology22]:

With:

- $EI = 3000$
- $q = 12$
- $EA = \infty$

Exercise (Frame)

Solve this problem by simplifying the stiffness matrix first.

```python
import numpy as np
import matplotlib as plt
import matrixmethod as mm
%config InlineBackend.figure_formats = ['svg']
```

```python
#YOUR_CODE_HERE
```

Solution to Exercise (Frame)   ⌄
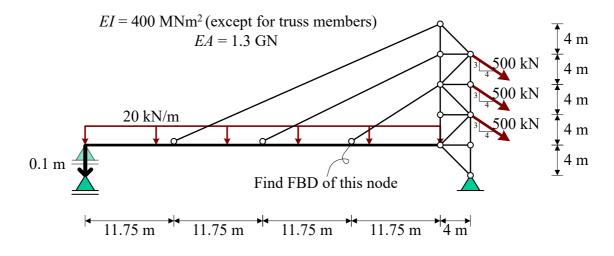
Graded assignment

When you've finished the workshops, you can start with the graded assignment. The process is very similar to the workshops, but now there's a deadline and you're required to write a report. You're going to solve the following model for displacements and internal forces using the Matrix Method.

This model is based on the bridge crossing the L723 in Walldorf, Germany:

Fig. 17 WiWa-Lokal [WiWaLokal17] (Foto: Pfeifer)

First, analyse the model and the questions you're required to answer:

1. Explain what effect hinges have on the matrix method. The following questions might help you:

   ○ Which deformations would you expect for an element with hinges on both ends? How about an element with a hinge on one end? What would you expect for $w(x)$?

   ○ Which internal forces would you expect for an element with hinges on both ends? And how do those forces relate to forces in the global coordinate system? How about an element with a hinge on one end? What would you expect for $M(x)$?

   ○ Which degrees-of-freedom are relevant for an element with hinges on both ends?

   ○ Can you calculate the rotation of hinged nodes? And if so, what is the meaning of those rotations? Does it matter if a node is connected with a hinge to either one or all of the neighboring elements?

   ○ If degrees of freedom are irrelevant, what would that mean for the size of your K-matrix? How could you solve it?

   ○ Can you reuse the element classes derived before? If yes, which tricks do you need to make it work? Are those tricks exact and do they mimic physical behaviour? If not, what new element types do you need? How do the values in those element types correspond to each other?

   ○ Is the stiffness matrix of an element with two hinges contained within the stiffness matrix of the full element? Is the K matrix of an element with just one hinge contained within the system matrix of the full element? If not, what are the differences?

2. Explain in words and math how you adapted/added code and/or procedures to solve this structure including hinges.

3. Describe alternatives you considered for your implementation in the previous steps.

4. Explain why your sanity checks prove that all of your code implementations are correct.

5. Make a table of all nodal displacement and show the displaced structure in a figure. Indicate how you identify nodes.

6. Show the moment diagram of the structure in a figure.

7. Provide a figure of a free body diagram of the full structure in which you show all the forces working on the structure (including support reactions) with numerical values from your code. This specific figure can be hand drawn.

8. Provide a figure of a free body diagram of the indicated node with numerical values from your code. This specific figure can be hand drawn. If you implement code for this in the matrixmethod package, make sure to perform sanity checks.

9. Comment on any potential mistakes you observed in your final answers.

10. If you had the time to expand this Matrix Method with an additional feature, what would that be?

Then, add potential new implementations to your code in `./matrixmethod/`. Please note that `./matrixmethod/` doesn't include any solutions from the workshops. If you've made new implementations, provide sanity checks to your implementations in `Graded_Implement.ipynb`. Then, solve and postprocess the problem in `Graded_Apply.ipynb`. Add a report in `.pdf` or `.md` format in which you included answers and reasoning for all the questions. Make sure all the values/figures you use in the report are solved/created with your code. Except for question 7 and 8: the free-body-diagrams can be hand-drawn. Furthermore, you're expected to provide to logically organise any auxiliary files you may use.

The deadline of the assignment is April 18th, 23:59, although you're encouraged to finish it directly after completing workshop 2. Doing so allows you to split the workload evenly. Commit and push all your files to the provided GitHub Classroom repository to hand in your assignment. Your latest commit before the deadline in the `main` branch will be graded. Incomplete assignments will be graded with a 1. The full solution won't be provided.

You can take the resit of this assignment in Q4. If you choose to do so, you can improve your first submission by resubmitting to the same repository. The deadline of the resit is June 20th, 23:59. For more information, see Learning objectives, activities and assessment 💡.

__init__.py

```python
from .node import *
from .elements import *
from .constrainer import *
```

node.py

> **⚠ Attention**
>
> This page shows a preview of the `matrixmethod` package. Please fork and clone the practice assignments to work on it locally from [GitHub](#)

```python
import numpy as np
```

```python
class Node:
    """
    The Node class is used to store node information and keep track of the total number o
    Degrees of Freedom (DOFs) of the problem. It introduces automatic bookkeeping in its
    initialization, which efficiently keeps track of which DOFs belong to which nodes. Th
    makes it easier to assemble matrices from multiple elements.

    Attributes:
        x (float): The x-coordinate of the node.
        z (float): The z-coordinate of the node.
        p (numpy.array):  The load vector of the node.
        dofs (list): The Degrees of Freedom associated with the node.

    Methods:
        clear(): Clears the counting of degrees of freedom and number of nodes.
        __init__(x, z): The constructor for Node class.
        add_load(p): Adds the given loads to the node.
        get_coords(): Returns the coordinates of the node.
        __str__(): Returns a string representation of the node.
    """
    ndof = 0
    nn   = 0

    def clear():
        """
        Clears the counting of degrees of freedom and number of nodes.

        This method resets the class-level counters for degrees of freedom and number of
        It should be used when you want to start a new problem from scratch.
        """
        Node.ndof = 0
        Node.nn = 0

    def __init__ (self, x, z):
        """
        The constructor for Node class.

        Parameters:
            x (float):       The x-coordinate of the node.
            z (float):       The z-coordinate of the node.
            p (numpy.array): The load vector of the node.
            dofs (list):     The Degrees of Freedom (u (in direction of x), w (in direct
        """

        self.x    = x
        self.z    = z
        self.p    = np.zeros(3)

        self.dofs = [Node.ndof, Node.ndof+1, Node.ndof+2]

        Node.ndof += 3
        Node.nn   += 1

    def add_load (self, p):
        """
        Adds the given loads to the node.

        The load is a vector p, which includes the load in the x and y direction and a mo
        These loads are added to the existing loads of the node.

        Parameters:
            p (numpy.array): A vector containing the load in the x direction, the load in
                             and the moment.
```

```python
        """
        self.p += p

    def get_coords(self):
        """
        Returns the coordinates of the node.

        Returns:
            numpy.array: An array containing the x and z coordinates of the node.
        """
        return np.array([self.x, self.z])

    def __str__(self):
        """
        Returns a string representation of the node.

        Returns:
            str: A string representation of the node.
        """
        return f"This node has:\n - x coordinate={self.x},\n - z coordinate={self.z},\n -
```

elements.py

> ⊕ **Added in version v2025.2.0:** After workshop 2
>
> Solutions workshop 2 and additional assignments in text and downloads

> ⊕ **Added in version v2025.1.0:** After workshop 1
>
> Solutions workshop 1 in text and downloads

> ⚠ **Attention**
>
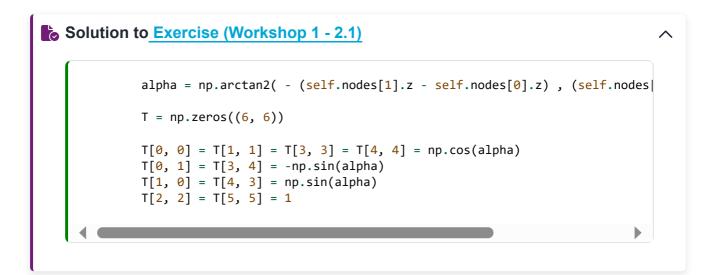> This page shows a preview of the `matrixmethod` package. Please fork and clone the practice assignments to work on it locally from [GitHub](#)
>
> After each workshop, the solution will be added to this preview and to the [GitHub-repository](#)

```python
import numpy as np
import matplotlib.pyplot as plt
```

```python
class Element:
    """
    The Element class keeps track of each element in the model, including cross-section p
    element orientation (for coordinate system transformations), and the nodes that make
    With the help of the Node class, it also keeps track of which Degrees of Freedom (DOF

    This class is responsible for providing the element stiffness matrix in the global co
    (for subsequent assembly) and postprocessing element-level fields.

    This class describes an element combining extension and Euler-Bernoulli bending. A si
    class could also be implemented for different element types (e.g., shear beam, Timosh
    For simplicity, it is assumed that elements are all arranged in a 2D plane.

    Attributes:
        node1 (Node): The first node of the element.
        node2 (Node): The second node of the element.
        EA (float): The axial stiffness of the element.
        EI (float): The flexural stiffness of the element.

    Methods:
        clear(): Clears the counting of elements.
        __init__(self, nodes): Initializes an Element object.
        set_section(self, props): Sets the section properties of the element.
        global_dofs(self): Returns the global degrees of freedom associated with the elem
        stiffness(self): Calculate the stiffness matrix of the element.
        add_distributed_load(self, q): Adds a distributed load to the element.
        bending_moments(self, u_global, num_points=2): Calculate the bending moments alon
        full_displacement(self, u_global, num_points=2): Calculates the displacement alon
        plot_moment_diagram(self, u_elem, num_points=10, global_c=False, scale=1.0): Plot
        plot_displaced(self, u_elem, num_points=10, global_c=False, scale=1.0): Plots the
        __str__(self): Returns a string representation of the Element object.
    """

    ne = 0

    def clear():
        """
        Clears the counting of elements

        This method resets the class-level counters for number of elements.
        It should be used when you want to start a new problem from scratch.
        """
        Element.ne = 0

    def __init__(self, node1, node2):
        """
        Initializes an Element object.

        Parameters:
        - node1 (Node): The first node of the element.
        - node2 (Node): The second node of the element.

        Attributes:
        - nodes (list): A list of Node objects representing the nodes of the element.
        - L (float): Length of the element.
        - cos (float): Cosine of the element's orientation angle.
        - sin (float): Sine of the element's orientation angle.
        - T (ndarray): Transformation matrix.
        - Tt (ndarray): Transpose of the transformation matrix.

        Returns:
        None
        """
```

```python
        self.nodes = [node1, node2]

        self.L = np.sqrt((self.nodes[1].x - self.nodes[0].x)**2.0 + (self.nodes[1].z - se

        alpha = np.arctan2 #YOUR CODE HERE

        T = np.zeros((6, 6))

        T[0, 0] = T[1, 1] = T[3, 3] = T[4, 4] #YOUR CODE HERE
        T[0, 1] = T[3, 4] #YOUR CODE HERE
        T[1, 0] = T[4, 3] #YOUR CODE HERE
        T[2, 2] = T[5, 5] #YOUR CODE HERE

        self.T = T
        self.Tt = np.transpose(T)

        self.q = np.array([0,0])
        self.local_element_load = np.array([0,0,0,0,0,0])

        Element.ne += 1
```

Solution to Exercise (Workshop 1 - 2.1) ⌃

```python
        alpha = np.arctan2( - (self.nodes[1].z - self.nodes[0].z) , (self.nodes

        T = np.zeros((6, 6))

        T[0, 0] = T[1, 1] = T[3, 3] = T[4, 4] = np.cos(alpha)
        T[0, 1] = T[3, 4] = -np.sin(alpha)
        T[1, 0] = T[4, 3] = np.sin(alpha)
        T[2, 2] = T[5, 5] = 1
```

```python
    def set_section(self, props):
        """
        Sets the section properties of the element.

        Parameters:
        - props (dict): A dictionary containing the section properties.
                        The dictionary should have the following keys:
                            - 'EA': The axial stiffness of the element.
                            - 'EI': The flexural stiffness of the element.

        Returns:
        None
        """
        if 'EA' in props:
            self.EA = props['EA']
        else:
            self.EA = 1.e20
        if 'EI' in props:
            self.EI = props['EI']
        else:
            self.EI = 1.e20

    def global_dofs(self):
        """
        Returns the global degrees of freedom associated with the element.

        Returns:
            numpy.ndarray: Array containing the global degrees of freedom.
        """
        return np.hstack((self.nodes[0].dofs, self.nodes[1].dofs))

    def stiffness(self):
        """
        Calculate the stiffness matrix of the element.

        Returns:
        np.ndarray: The stiffness matrix of the element.
        """
        k = np.zeros((6, 6))

        EA = self.EA
        EI = self.EI
        L = self.L

        #YOUR CODE HERE

        return np.matmul(np.matmul(self.Tt, k), self.T)
```
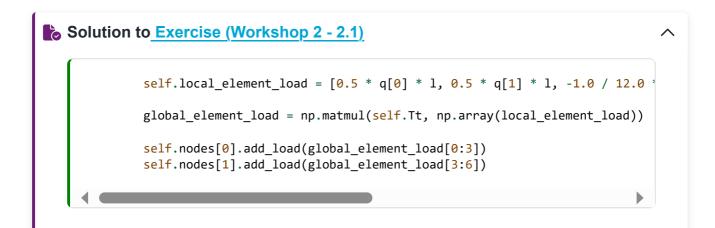
Solution to [Exercise (Workshop 1 - 2.1)](#)

```python
        # Extension contribution

        k[0, 0] = k[3, 3] = EA / L
        k[3, 0] = k[0, 3] = -EA / L

        # Bending contribution

        k[1, 1] = k[4, 4] = 12.0 * EI / L / L / L
        k[1, 4] = k[4, 1] = -12.0 * EI / L / L / L
        k[1, 2] = k[2, 1] = k[1, 5] = k[5, 1] = -6.0 * EI / L / L
        k[2, 4] = k[4, 2] = k[4, 5] = k[5, 4] = 6.0 * EI / L / L
        k[2, 2] = k[5, 5] = 4.0 * EI / L
        k[2, 5] = k[5, 2] = 2.0 * EI / L

        return np.matmul(np.matmul(self.Tt, k), self.T)
```

```python
    def add_distributed_load(self, q):
        """
        Adds a distributed load to the element.

        Parameters:
            q (list): List of distributed load in local x and z direction.

        Returns:
            None
        """

        l = self.L
        self.q = np.array(q)

        self.local_element_load # =[YOUR CODE HERE, , , , , ]

        global_element_load #YOUR CODE HERE

        self.nodes[0].add_load #YOUR CODE HERE
        self.nodes[1].add_load #YOUR CODE HERE
```

Solution to [Exercise (Workshop 2 - 2.1)](#)

```python
        self.local_element_load = [0.5 * q[0] * l, 0.5 * q[1] * l, -1.0 / 12.0 ·

        global_element_load = np.matmul(self.Tt, np.array(local_element_load))

        self.nodes[0].add_load(global_element_load[0:3])
        self.nodes[1].add_load(global_element_load[3:6])
```

```python
def bending_moments(self, u_global, num_points=2):
    """
    Calculate the bending moments along the element.

    Parameters:
    - u_global (numpy.ndarray): Global displacement vector.
    - num_points (int): Number of points to evaluate the bending moments. Default is

    Returns:
    - M (numpy.ndarray): Array of bending moments at the specified points.
    """

    l = self.L
    q = self.q[1]
    EI = self.EI

    local_x = np.linspace(0.0, l, num_points)

    local_disp #YOUR CODE HERE

    M #YOUR CODE HERE

    return M
```

Solution to [Exercise (Workshop 2 - 2.1)](#) ∧

```python
local_disp = np.matmul(self.T, u_global)

w_1 = local_disp[1]
phi_1 = local_disp[2]
w_2 = local_disp[4]
phi_2 = local_disp[5]

M = (-l ** 5.0 * q + 6.0 * l ** 4.0 * q * local_x
     - 6.0 * q * local_x * local_x * l ** 3.0 - 48.0 * (phi_1 + phi_2 /
     + 72.0 * EI * ((phi_1 + phi_2) * local_x + w_1 - w_2) * l - 144.0

return M
```

```python
    def full_displacement (self, u_global, num_points=2):
        """
        Calculates the displacement along the element.

        Args:
            u_global (numpy.ndarray): Global displacement vector of the element.
            num_points (int, optional): Number of points to calculate the bending moments

        Returns:
            numpy.ndarray: Array of displacement along the element.
        """
        #YOUR CODE HERE

        u #YOUR CODE HERE
        w #YOUR CODE HERE

        return u, w
```

Solution to [Exercise (Workshop 2 - 2.1)](#)

```python
        L = self.L
        q = self.q[1]
        q_x = self.q[0]
        EI= self.EI
        EA = self.EA

        x = np.linspace ( 0.0, L, num_points )

        ul = np.matmul ( self.T, u_global )

        u_1   = ul[0]
        w_1   = ul[1]
        phi_1 = ul[2]
        u_2   = ul[3]
        w_2   = ul[4]
        phi_2 = ul[5]

        u = q_x*(-L*x/(2*EA) + x**2/(2*EA)) + u_1*(1 - x/L) + u_2*x/L
        w = phi_1*(-x + 2*x**2/L - x**3/L**2) + phi_2*(x**2/L - x**3/L**2) + q*(

        return u, w
```

```python
def plot_moment_diagram (self, u_elem, num_points=10, global_c=False, scale=1.0):
    """
    Plots the bending moment diagram of the element.

    Args:
        u_global (numpy.ndarray): Global displacement vector of the element.
        num_points (int, optional): Number of points to calculate the bending moments
        global_c (bool, optional): If True, plots the bending moment diagram in the g
        scale (float, optional): Scale factor for the bending moment diagram. Default

    Returns:
        None
    """
    import matplotlib.pyplot as plt

    x = np.linspace ( 0.0, self.L, num_points )
    M = self.bending_moments ( u_elem, num_points )
    xM_local = np.vstack((np.hstack([0,x,x[-1]]),np.hstack([0,M,0])*scale))
    if global_c:
        xM_global = np.matmul(self.Tt[0:2,:2],xM_local)
        xz_start_node = np.vstack((np.ones(num_points+2)*self.nodes[0].x, np.ones(num
        xz_Mlijn = xM_global + xz_start_node
        p = plt.plot(xz_Mlijn[0,:],xz_Mlijn[1,:])
        X0= self.nodes[0].x
        Z0= self.nodes[0].z
        X1= self.nodes[1].x
        Z1= self.nodes[1].z
        plt.plot((X0, X1), (Z0, Z1), color=p[0].get_color())
        plt.axis('off')
        plt.axis('equal')
    else:
        p = plt.plot(xM_local[0,:],xM_local[1,:])
        plt.xlabel ( "x" )
        plt.ylabel ( "M" )
    if not plt.gca().yaxis_inverted():
        plt.gca().invert_yaxis()
    plt.gcf().patch.set_alpha(0.0)
    plt.gca().patch.set_alpha(0.0)
    plt.gca().patch.set_alpha(0.0)
    plt.title('Moment line')

def plot_displaced(self, u_elem, num_points=10, global_c=False, scale=1.0):
    """
    Plots the displacd element.

    Args:
        u_global (numpy.ndarray): Global displacement vector of the element.
        num_points (int, optional): Number of points to calculate the bending moments
        global_c (bool, optional): If True, plots the displacement diagram in the glo
        scale (float, optional): Scale factor for the displacement diagram. Default i

    Returns:
        None
    """

    x = np.linspace ( 0.0, self.L, num_points )
    u, w = self.full_displacement ( u_elem, num_points )
    uw_local = np.vstack((x+u*scale,w*scale))
    if global_c:
        uw_global = np.matmul(self.Tt[:2,:2],uw_local)
        xz_start_node = np.vstack((np.ones(num_points)*self.nodes[0].x, np.ones(num_p
        uw = uw_global + xz_start_node
        p =  plt.plot(uw[0,:],uw[1,:])
```

```python
            X0= self.nodes[0].x
            Z0= self.nodes[0].z
            X1= self.nodes[1].x
            Z1= self.nodes[1].z
            plt.plot((X0, X1), (Z0, Z1), color=p[0].get_color(),alpha=0.3)
            plt.axis('off')
            plt.axis('equal')
        else:
            p = plt.plot(uw_local[0,:],uw_local[1,:])
            plt.plot((0, self.L), (0, 0), color=p[0].get_color(),alpha=0.3)
        if not plt.gca().yaxis_inverted():
            plt.gca().invert_yaxis()
        plt.gcf().patch.set_alpha(0.0)
        plt.gca().patch.set_alpha(0.0)
        plt.gca().patch.set_alpha(0.0)
        plt.title('Displaced structure')

    def plot_numbered_structure(self,beam_number):
        """
        Plots the nodes and elements of the structure with their node and element numbers

        Returns:
            None
        """

        X0= self.nodes[0].x
        Z0= self.nodes[0].z
        X1= self.nodes[1].x
        Z1= self.nodes[1].z
        node_num = []
        node_num.append(self.nodes[0].dofs[0] // 3)
        node_num.append(self.nodes[1].dofs[0] // 3)
        plt.plot((X0, X1), (Z0, Z1), color='black',alpha=0.3)
        for i, node in enumerate(self.nodes):
            plt.text(node.x, node.z, f'[{node.dofs[0] // 3}]', fontsize=12, ha='center',
        plt.text((X0+X1)/2, (Z0+Z1)/2, f'({beam_number})', fontsize=12, ha='center', va='
        if not plt.gca().yaxis_inverted():
            plt.gca().invert_yaxis()
        plt.axis('off')
        plt.axis('equal')
        plt.gcf().patch.set_alpha(0.0)
        plt.gca().patch.set_alpha(0.0)
        plt.gca().patch.set_alpha(0.0)


    def __str__(self):
        """
        Returns a string representation of the Element object.

        The string includes the values of the node1, node2 attributes.
        """
        return f"Element connecting:\nnode #1:\n {self.nodes[0]}\nwith node #2:\n {self.n
```

Solution to Exercise (Kinked beam)

```python
class EB_point_load_element (Element):
    """
    The EB_point_load_element class describes an element combining extension and

    Attributes:
        node1 (Node): The first node of the element.
        node2 (Node): The second node of the element.
        EA (float): Axial stiffness of the element.
        EI (float): Bending stiffness of the element.
        F (float): Point load in local z direction.
        L (float): Length of the element.

    Methods:
        add_point_load_halfway(self, F): Adds a point load to the element.
        bending_moments(u_global, num_points=2): Calculates the bending moments
        full_displacement(u_global, num_points=2): Calculates the displacement a

    Inherits from:
        Element: Base class for all structural elements.
    """
    def add_point_load_halfway(self, F):
        """
        Adds a point load to the element.

        Parameters:
            F (float): Point load in local z direction.

        Returns:
            None
        """
        self.F = F
        l = self.L

        el = [0, F / 2, - F * l / 8, 0, F / 2, F * l / 8]

        eg = np.matmul(self.Tt, np.array(el))

        self.nodes[0].add_load(eg[0:3])
        self.nodes[1].add_load(eg[3:6])

    def bending_moments (self, u_global, num_points=2):
        """
        Calculates the bending moments along the element.

        Args:
            u_global (numpy.ndarray): Global displacement vector of the element.
            num_points (int, optional): Number of points to calculate the bendin

        Returns:
            numpy.ndarray: Array of bending moments along the element.
        """
        L = self.L
        F = self.F
        EI= self.EI

        x = np.linspace ( 0.0, L, num_points )
        M  = np.zeros(num_points)

        ul = np.matmul ( self.T, u_global )

        w_1   = ul[1]
        phi_1 = ul[2]
        w_2   = ul[4]
```

```python
        phi_2 = ul[5]

        M = -F*L/8 + F*x/2 + phi_1*(-4*EI/L + 6*EI*x/L**2) + phi_2*(-2*EI/L + 6
        index_halfway = int(num_points/2)
        M[index_halfway:] += - F*(-L/2 + x[index_halfway:])
        return M

    def full_displacement (self, u_global, num_points=2):
        """
        Calculates the displacement along the element.

        Args:
            u_global (numpy.ndarray): Global displacement vector of the element.
            num_points (int, optional): Number of points to calculate the bendin

        Returns:
            numpy.ndarray: Array of displacement along the element.
        """
        L = self.L
        F = self.F
        q_x = self.q[0]
        EI= self.EI
        EA = self.EA

        x = np.linspace ( 0.0, L, num_points )

        ul = np.matmul ( self.T, u_global )

        u_1   = ul[0]
        w_1   = ul[1]
        phi_1 = ul[2]
        u_2   = ul[3]
        w_2   = ul[4]
        phi_2 = ul[5]

        u = q_x*(-L*x/(2*EA) + x**2/(2*EA)) + u_1*(1 - x/L) + u_2*x/L
        w = phi_1*(-x + 2*x**2/L - x**3/L**2) + phi_2*(x**2/L - x**3/L**2) + w_1
        index_halfway = int(num_points/2)
        w[index_halfway:] += F*(x[index_halfway:] - L/2)**3/(6*EI)
        return u, w
```

constrainer.py

> ❗ **Added in version v2025.2.0:** After workshop 2
>
> Solutions workshop 2 and additional assignments in text and downloads

> ❗ **Added in version v2025.1.0:** After workshop 1
>
> Solutions workshop 1 in text and downloads

> ❗ **Attention**
>
> This page shows a preview of the `matrixmethod` package. Please fork and clone the practice assignments to work on it locally from GitHub
>
> After each workshop, the solution will be added to this preview and to the GitHub-repository

```python
import numpy as np
```

```python
class Constrainer:
    """
    A class that represents a constrainer for fixing degrees of freedom in a structural a

    Attributes:
        cons_dofs (list): A list of constrained degrees of freedom.
        cons_vals (list): A list of corresponding constraint values.

    Methods:
        fix_dof: Fixes a degree of freedom at a specific value.
        fix_node: Fixes all degrees of freedom of a node.
        full_disp: Combines the displacements of free and constrained degrees of freedom.
        constrain: Applies the constraints to the stiffness matrix and load vector.
        support_reactions: Calculates the support reactions based on the constrained disp
    """

    def __init__(self):
        """
        Initializes a new instance of the Constrainer class.

        Attributes:
            cons_dofs (list): A list of constrained degrees of freedom.
            cons_vals (list): A list of corresponding constraint values.
        """
        self.cons_dofs = []
        self.cons_vals = []

    def fix_dof (self, node, dof, value = 0):
        """
        Fixes a degree of freedom  at a specific value.

        Args:
            node (Node): The node object.
            dof (int): The index of the degree of freedom to fix.
            value (float, optional): The value to fix the degree of freedom at. Defaults
        """
        self.cons_dofs.append(node.dofs[dof])
        assert value == 0, "Only zero values are supported for now."
        self.cons_vals.append(value)

    def fix_node (self, node):
        """
        Fixes all degrees of freedom of a node.

        Args:
            node (Node): The node object.
        """
        for dof in [0,1,2]:
            self.fix_dof (node, dof)

    def full_disp (self,u_free):
        """
        Combines the displacements of free and constrained degrees of freedom.

        Args:
            u_free (numpy.ndarray): The displacements of the free degrees of freedom.

        Returns:
            numpy.ndarray: The combined displacements of all degrees of freedom.
        """
        u_full = np.zeros(len(self.free_dofs) + len(self.cons_dofs))

        u_full[self.free_dofs] = u_free
```

```
        u_full[self.cons_dofs] = self.cons_vals

        return u_full

    def constrain (self, k, f):
        """
        Applies the constraints to the stiffness matrix and load vector.

        Args:
            k (numpy.ndarray): The stiffness matrix.
            f (numpy.ndarray): The load vector.

        Returns:
            tuple: A tuple containing the stiffness matrix corresponding to free dofs and
        """
        self.free_dofs = [i for i in range(len(f)) if i not in self.cons_dofs]

        Kff #= k[np.ix_(YOUR CODE HERE)]
        Ff # YOUR CODE HERE

        return Kff, Ff
```

Solution to [Exercise (Workshop 1 - 3.1)](#) ⌃

```
        self.free_dofs = [i for i in range(len(f)) if i not in self.cons_dofs]

        Kff = k[np.ix_(self.free_dofs,self.free_dofs)]
        Ff = f[self.free_dofs]

        return Kff, Ff
```

Solution to [Exercise (Workshop 2 - 3.1)](#) ⌃
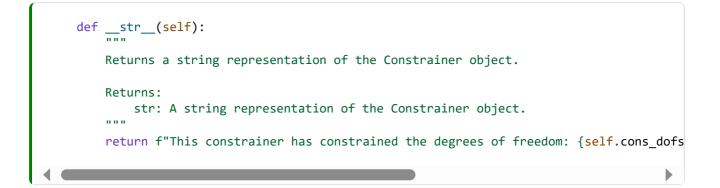
```
        self.free_dofs = [i for i in range(len(f)) if i not in self.cons_dofs]

        Kff = k[np.ix_(self.free_dofs,self.free_dofs)]
        Kfc = k[np.ix_(self.free_dofs,self.cons_dofs)]
        Ff = f[self.free_dofs]

        return Kff, Ff - np.matmul(Kfc,self.cons_vals)
```

```python
    def support_reactions (self,k,u_free,f):
        """
        Calculates the support reactions based on the constrained displacements.

        Args:
            k (numpy.ndarray): The stiffness matrix.
            u_free (numpy.ndarray): The displacements of the free degrees of freedom.
            f (numpy.ndarray): The load vector.

        Returns:
            numpy.ndarray: The support reactions.
        """
        #YOUR CODE HERE

        return #YOUR CODE HERE
```

Solution to [Exercise (Workshop 2 - 3.1)](Exercise%20(Workshop%202%20-%203.1))

```python
        Kcf = k[np.ix_(self.cons_dofs,self.free_dofs)]
        Kcc = k[np.ix_(self.cons_dofs,self.cons_dofs)]

        return np.matmul(Kcf,u_free) + np.matmul(Kcc,self.cons_vals) - f[self.cc
```

```python
    def __str__(self):
        """
        Returns a string representation of the Constrainer object.

        Returns:
            str: A string representation of the Constrainer object.
        """
        return f"This constrainer has constrained the degrees of freedom: {self.cons_dofs
```

References

[HansWfDUoTechnology22] Hans Welleman from Delft University of Technology. Cm5
(cie4190) - contents of lectures - lecture material - part 5: matrix method.
https://icozct.tudelft.nl/TUD_CT/CM5/collegestof/, 2022.

[HansWfDUoTechnology24] Hans Welleman from Delft University of Technology. Lecture 6:
newtonian formulation - create basic 1d-models - week 2 - unit 2 ciem5000.
https://brightspace.tudelft.nl/d2l/le/content/680476/viewContent/3826607/View,
2024.

[MUDETatSAfDUoTechnology24a] MUDE Teachers and the Student Army from Delft
University of Technology. Mude book.
https://mude.citg.tudelft.nl/2024/book/fem/matrix.html, 2024.

[MUDETatSAfDUoTechnology24b] MUDE Teachers and the Student Army from Delft
University of Technology. Mude book.
https://mude.citg.tudelft.nl/2024/book/fem/weak.html, 2024.

[WiWaLokal17] WiWa-Lokal. Walldorf: illumination für die brücke zum industriegebiet.
https://www.wiwa-lokal.de/walldorf-illumination-fuer-die-bruecke-zum-
industriegebiet/, November 2017.

Credits and License

Contents

You can refer to this book as:

> Tom van Woudenberg and Iuri Rocha from Delft University of Technology (2025) *Matrix method in statics*. https://ciem5000-2025.github.io/book. Source files at ⊙ CIEM5000-2025/book

You can refer to individual chapters or pages within this book as:

> `<Title of Chapter or Page>`. In Tom van Woudenberg and Iuri Rocha from Delft University of Technology (2025) *Matrix method in statics*. https://ciem5000-2025.github.io/book/. Source files at ⊙ CIEM5000-2025/book: `./book/<path to file(s)>` chapter, accessed `date`.

We anticipate that the content of this book will change significantly. Therefore, we recommend using the source code directly with the citation above that refers to the GitHub repository and lists the date and name of the file. Although content will be added over time, chapter titles and URL's in this book are expected to remain relatively static. However, we make no guarantee, so if it is important for you to reference a specific location/commit within the book.

How the book is made

This book is created using open source tools: it is a JupyterBook that is written using Markdown and Jupyter notebooks. Additional tooling is used from the TeachBooks initiative to enhance the editing and reading experience. The files are stored on a public GitHub repository. The website can be viewed at https://ciem5000-2025.github.io/book/. View the repository README file or contact the authors for additional information.

External resources

Parts of this book are taken from other external resources and reused in various ways. If an author is not listed on a particular page, it is by the Authors. Page Finite element method vs. Matrix Method includes code from MUDE Teachers and the Student Army from Delft University of Technology [MUDETatSAfDUoTechnology24a]. Original content is licensed under CC BY.

Changelog

Contents

This changelog will include all changes, except for minor adjustments like typos.

v2025.2.0, 2025-02-20 17:30, after workshop 2

- Added solutions workshop 2 to downloads Activities - Workshop 2
- Added solutions workshop 2 to text and downloads Activities - Workshop 2 - Apply
- Added solutions workshop 2 to text and downloads Activities - Workshop 2 - Implement
- Added solutions additional assignments to downloads Activities - Additional assignments
- Added solutions additional assignments to text and downloads Activities - Additional assignments - Beam
- Added solutions additional assignments to text and downloads Activities - Additional assignments - Kinked beam
- Added solutions additional assignments to text and downloads Activities - Additional assignments - Frame
- Added solutions workshop 2 and additional assignments to downloads matrixmethod package - `__init__.py`
- Added solutions workshop 2 and additional assignments to downloads matrixmethod package - `node.py`

- Added solutions workshop 2 and additional assignments to text and downloads [matrixmethod package -](#) `elements.py`
- Added solutions workshop 2 and additional assignments to text and downloads [matrixmethod package -](#) `constrainer.py`
- See full changelog [here](#)

v2025.1.1, 2025-02-18 9:30 after second lecture

- Fixed typo and removed example in [lecture slides lecture 2](#)
- Fixed various typos
- Full changelog [here](#)

v2025.1.0, 2025-02-13 17:30, after workshop 1

- Added solutions workshop 1 to downloads [Activities - Workshop 1](#)
- Added solutions workshop 1 to text and downloads [Activities - Workshop 1 - Apply](#)
- Added solutions workshop 1 to text and downloads [Activities - Workshop 1 - Implement](#)
- Added solutions workshop 1 to downloads [matrixmethod package -](#) `__init__.py`
- Added solutions workshop 1 to downloads [matrixmethod package -](#) `node.py`
- Added solutions workshop 1 to text and downloads [matrixmethod package -](#) `elements.py`
- Added solutions workshop 1 to text and downloads [matrixmethod package -](#) `constrainer.py`
- Full changelog

v2025.0.3, 2025-02-13 13:33, before workshop 1

- Fixed typo in [Workshop 1 - Exercise 2.6](#)
- Full changelog [here](#)

v2025.0.2, 2025-02-11 10:42, after first lecture

- Added html export of book as a zip to How to use this TeachBook 📘 and most other pages as additional download.
- Fixed various typos
- Full changelog here

v2025.0.1, 2025-02-10 15:02, before first lecture

- Updated lecture slides lecture 1: moved slides on python packages and updated installation requirements first workshop
- Fixed various typos
- Full changelog here

v2025.0.0, start course

- Converted material to interactive book
- Separated Activities - additional assignments
- Removed hinged beam additional assignment
- Added Activities - Lecture 1 - Recap displacement method
- New graded assignment
- Various improvements to student-experience