CALFEM for Python

Release 0.1

...

Oct 22, 2025

Contents:

cfu-funktioner finns ej med?

Welcome to the documentation page for CALFEM for Python. On this page you will find examples of how to use CALFEM as well as reference documentation for the different modules in the CALFEM package.

Chapter 1

Introduction

CALFEM is a Python package for finite element applications. This manual concerns mainly the finite element functions, but it also contains descriptions of some often-used Python functions.

The finite element analysis can be carried out either interactively or in a batch-oriented fashion. In the interactive mode, the functions are evaluated one by one in an interactive Python session. In the batch-oriented mode, a sequence of functions is written in a file named *.py* file and evaluated by calling python interpreter with the filename as an argument. The batch-oriented mode is a more flexible way of performing finite element analysis because the *.py* file can be written in an ordinary editor. This way of using CALFEM is recommended because it gives a structured organization of the functions. Changes and reruns are also easily executed in the batch-oriented mode.

To use the CALFEM package, it must first be imported. This is done by the command:

```
import calfem.core as cfc
```

The function of this package is then available with the prefix `cfc`. So the function `spring1e` is called as `cfc.spring1e`.

Plot functions are available in the `calfem.vis_mpl` package. To use these functions, this package must also be imported:

```
import calfem.vis_mpl as cfv
```

The these functions are then called with the prefix `cfv`. So the function `eldraw` is called as `cfv.eldraw`.

A command line typically consists of functions for vector and matrix operations, calls to functions in the CALFEM finite element library, or commands for workspace operations. An example of a command line for a matrix operation is:

```
C = A + B.T
```

where the two-by-two matrices `A` and `B` are added together, and the result is stored in matrix `C`. The matrix `B.T` is the transpose of `B`. In Python, the transpose operator is denoted by a `T`.

An example of a call to the element library is:

```
Ke = cfc.spring1e(ep)
```

where the two-by-two element stiffness matrix $\mathbf{K}^e$ for a spring element is computed and stored in the variable `Ke`. The input argument is given within parentheses ( ) after the name of the function and is in this case `ep` containing the spring stiffness k. Some functions have multiple input arguments and/or multiple output arguments. For example:

```
L, X = cfc.eigen(K, M)
```

computes the eigenvalues and eigenvectors of a pair of matrices `K` and `M`. The output variables the eigenvalues λ stored in the vector `L` and the corresponding eigenvectors stored in the matrix `X` are separated by commas. The input arguments are given inside the parentheses ( ) and also separated by commas.

The statement:

```
help(function)
```

provides information about the purpose and syntax for the specified function.

The available functions are organized in groups as follows. Each group is described in a separate chapter.

Table 1: Groups of functions

General purpose commands	For managing variables, workspace, output etc.
Matrix functions	For matrix handling
Material functions	For computing material matrices
Element functions	For computing element matrices and element forces
System functions	For setting up and solving systems of equations
Statement functions	For algorithm definitions
Graphics functions	For plotting

Chapter 2

Material functions

The group of material functions comprises functions for constitutive models. The available models can treat linear elastic and isotropic hardening von Mises material. These material models are defined by the functions:

Table 1: Material property functions

hooke	Form linear elastic constitutive matrix
mises	Compute stresses and plastic strains for isotropic hardening von Mises material
dmises	Form elasto-plastic continuum matrix for isotropic hardening von Mises material

2.1 hooke

Purpose

Compute material matrix for a linear elastic and isotropic material.

Syntax

```
D = hooke(ptype, E, v)
```

Description

The function `hooke` computes the material matrix **D** for a linear elastic and isotropic material.

The variable `ptype` is used to define the type of analysis:

- `ptype = 1` - plane stress
- `ptype = 2` - plane strain
- `ptype = 3` - axisymmetry
- `ptype = 4` - three dimensional analysis

The material parameters E and ν define the modulus of elasticity and the Poisson's ratio, respectively.

Theory

For plane stress (`ptype=1`), **D** is formed as

$$\mathbf{D} = \frac{E}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}$$

For plane strain (`ptype=2`) and axisymmetry (`ptype=3`), **D** is formed as

$$\mathbf{D} = \frac{E}{(1+\nu)(1-2\nu)}\begin{bmatrix} 1-\nu & \nu & \nu & 0 \\ \nu & 1-\nu & \nu & 0 \\ \nu & \nu & 1-\nu & 0 \\ 0 & 0 & 0 & \frac{1}{2}(1-2\nu) \end{bmatrix}$$

For the three dimensional case (`ptype=4`), **D** is formed as

$$\mathbf{D} = \frac{E}{(1+\nu)(1-2\nu)}\begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}(1-2\nu) & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2}(1-2\nu) & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2}(1-2\nu) \end{bmatrix}$$

2.2 mises

Purpose

Compute stresses and plastic strains for an elasto-plastic isotropic hardening von Mises material.

Syntax

```
[es, deps, st] = mises(ptype, mp, est, st)
```

Description

The `mises` function computes updated stresses, `es`, plastic strain increments `deps`, and state variables `st` for an elasto-plastic isotropic hardening von Mises material.

The input variable `ptype` defines the type of analysis, see also `hooke`. The vector `mp` contains the material constants:

`mp = [`E, ν, h`]`

where E is the modulus of elasticity, ν is the Poisson's ratio, and h is the plastic modulus.

The input matrix `est` contains trial stresses obtained by using the elastic material matrix `D` in `plants` or a similar `s`-function. The input vector `st` contains the state parameters:

`st = [` $yi\ \sigma_y\ \varepsilon^p_{eff}$ `]`

at the beginning of the step. The scalar yi indicates whether the material behaviour is elasto-plastic ($yi = 1$) or elastic ($yi = 0$). The current yield stress is denoted by σ_y and the effective plastic strain by ε^p_{eff}.

The output variables `es` and `st` contain updated values obtained by integration of the constitutive equations over the actual displacement step. The increments of the plastic strains are stored in the vector `deps`.

If `es` and `st` contain more than one row, then every row will be treated by the command.

Note

It is not necessary to check whether the material behaviour is elastic or elasto-plastic; this test is performed by the function. The computation is based on an Euler-Backward method, i.e., the radial return method.

Only the cases `ptype = 2, 3, 4` are implemented.

2.3 dmises

Purpose

Form the elasto-plastic continuum matrix for an isotropic hardening von Mises material.

Syntax

```
D = dmises(ptype, mp, es, st)
```

Description

`dmises` forms the elasto-plastic continuum matrix for an isotropic hardening von Mises material.

The input variable `ptype` is used to define the type of analysis, cf. `hooke`.

The vector `mp` contains the material constants:

`mp = [`E, ν, h`]`

where E is the modulus of elasticity, ν is the Poisson's ratio, and h is the plastic modulus.

The matrix `es` contains current stresses obtained from `plants` or some similar `s`-function, and the vector `st` contains the current state parameters:

`st = [` $yi\ \sigma_y\ \varepsilon^p_{eff}$ `]`

where $yi = 1$ if the material behaviour is elasto-plastic, and $yi = 0$ if the material behaviour is elastic. The current yield stress is denoted by σ_y, and the current effective plastic strain by ε^p_{eff}.

Note

Only the case `ptype = 2` is implemented.

Chapter 3

Element functions

The group of element functions contains functions for computation of element matrices and element forces for different element types. The element functions have been divided into the following groups:

- Spring element
- Bar elements
- Heat flow elements
- Solid elements
- Beam elements
- Plate element

For each element type, there is a function for computation of the element stiffness matrix $\mathbf{K}^e$, stored in `Ke`. For most of the elements, an element load vector $\mathbf{f}^e$, stored in `fe` can also be computed. These functions are identified by their last letter `-e`.

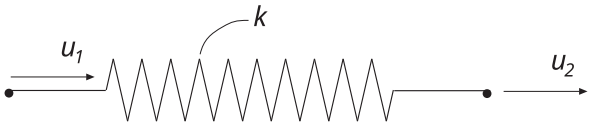
Using the function `assem`, the element stiffness matrices and element load vectors are assembled into a global stiffness matrix $\mathbf{K}$, stored in `K` and a load vector $\mathbf{f}$, stored in `f`. Unknown nodal values of temperatures or displacements $\mathbf{a}$ are computed by solving the system of equations $\mathbf{Ka} = \mathbf{f}$ using the function `solveq`. A vector of nodal values of temperatures or displacements for a specific element is formed by the function `extract_ed`.

When the element nodal values have been computed, the element stresses or element flux can be calculated using functions specific to the element type concerned. These functions are identified by their last letter `-s`.

For some elements, a function for computing the internal force vector is also available. These functions are identified by their last letter `-f`.

3.1 Spring element functions

The spring element, shown below, can be used for the analysis of one-dimensional spring systems and for a variety of analogous physical problems.



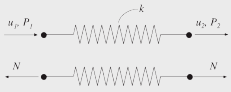
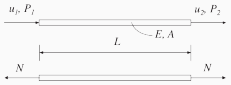
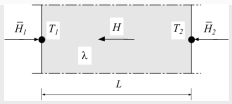
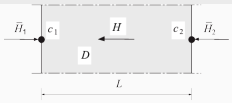
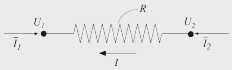
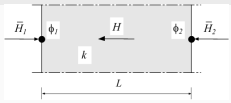
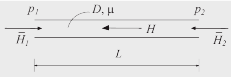
Spring element

Quantities corresponding to the variables of the spring are listed in Table 1.

Table 1: Analogous quantities

Problem type	Spring stiffness	Nodal displacement	Element force	Spring force
Spring	k	u	P	N
Bar	$\frac{EA}{L}$	u	P	N
Thermal conduction	$\frac{\lambda A}{L}$	T	$\bar{H}$	H
Diffusion	$\frac{DA}{L}$	c	$\bar{H}$	H
Electrical circuit	$\frac{1}{R}$	U	$\bar{I}$	I
Groundwater flow	$\frac{kA}{L}$	ϕ	$\bar{H}$	H
Pipe network	$\frac{\pi D^4}{128\mu L}$	p	$\bar{H}$	H

Table 2: Quantities used in different types of problems

Problem type	Quantities	Designations	Description
Spring		k, u, P, N	spring stiffness, displacement, element force, spring force
Bar		L, E, A, u, P, N	length, modulus of elasticity, area of cross section, displacement, element force, normal force
Thermal conduction		L, λ, T, $\bar{H}$, H	length, thermal conductivity, temperature, element heat flow, internal heat flow
Diffusion		L, D, c, $\bar{H}$, H	length, diffusivity, nodal concentration, nodal mass flow, element mass flow
Electrical circuit		R, U, $\bar{I}$, I	resistance, potential, element current, internal current
Groundwater flow		L, k, ϕ, $\bar{H}$, H	length, permeability, piezometric head, element water flow, internal water flow
Pipe network (laminar flow)		L, D, μ, p, $\bar{H}$, H	length, pipe diameter, viscosity, pressure, element fluid flow, internal fluid flow

The following functions are available for the spring element:

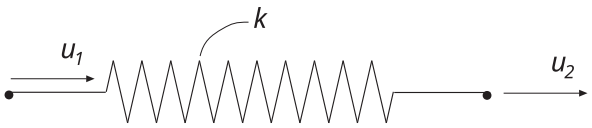
Table 3: Spring functions

spring1e	Compute element matrix
spring1s	Compute spring force

3.1.1 spring1e

Purpose

Compute element stiffness matrix for a spring element.



Syntax

```
Ke = cfc.spring1e(ep)
```

Description

`spring1e` provides the element stiffness matrix $\bar{\mathbf{K}}^e$ for a spring element.

The input variable

`ep`$= [k]$

supplies the spring stiffness k or the analog quantity defined in Table tanalog.

Theory

The element stiffness matrix $\mathbf{K}^e$, stored in `Ke`, is computed according to

$$\mathbf{K}^e = \begin{bmatrix} k & -k \\ -k & k \end{bmatrix}$$

where k is defined by `ep`.

3.1.2 spring1s

Purpose

Compute spring force in a spring element.

Syntax

```
es = cfc.spring1s(ep, ed)
```

Description

`spring1s` computes the spring force in the spring element `spring1e`.

The input variable `ep` is defined in `spring1e` and the element nodal displacements `ed` are obtained by the function `extract_ed`.

The output variable

`es`$= [N]$

contains the spring force, or the analog quantity.

Theory

The spring force N, or analog quantity, is computed according to

$$N = k\,(u_2 - u_1)$$

3.2 Bar element functions

Bar elements are available for one, two, and three dimensional analysis.

Table 4: One dimensional bar elements

bar1e	Compute element matrix
bar1s	Compute normal force
bar1we	Compute element matrix for bar element with elastic support
bar1ws	Compute normal force for bar element with elastic support

Table 5: Two dimensional bar elements

bar2e	Compute element matrix
bar2s	Compute normal force
bar2ge	Compute element matrix for geometric nonlinear element
bar2gs	Compute normal force for bar element with elastic support

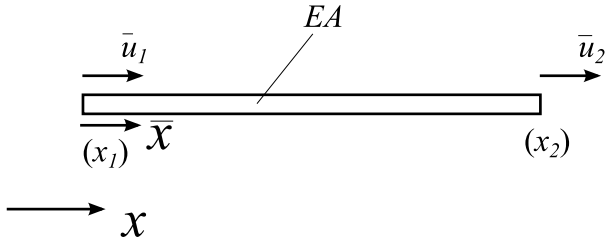
Table 6: Three dimensional bar elements

bar3e	Compute element matrix
bar3s	Compute normal force

3.2.1 bar1e

Purpose

Compute element stiffness matrix for a one dimensional bar element.



Syntax

```
Ke = cfc.bar1e(ex, ep)
Ke, fe = cfc.bar1e(ex, ep, eq)
```

Description

`bar1e` provides the element stiffness matrix $\bar{\mathbf{K}}^e$ for a one dimensional bar element. The input variables

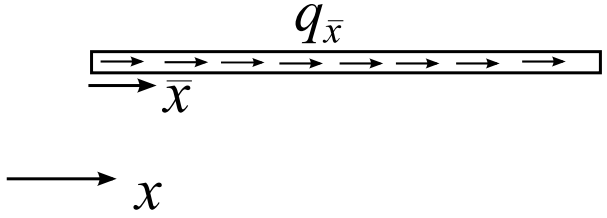
`ex`$= [x_1 \;\; x_2]$ `ep`$= [E \; A]$

supply the element nodal coordinates x_1 and x_2, the modulus of elasticity E, and the cross section area A.

The element load vector $\bar{\mathbf{f}}_l^e$ can also be computed if a uniformly distributed load is applied to the element. The optional input variable

`eq`$= [q_{\bar{x}}]$

contains the distributed load per unit length, $q_{\bar{x}}$.



Theory

The element stiffness matrix $\bar{\mathbf{K}}^e$, stored in `Ke`, is computed according to

$$\bar{\mathbf{K}}^e = \frac{D_{EA}}{L}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

where the axial stiffness D_{EA} and the length L are given by

$$D_{EA} = EA; \quad L = x_2 - x_1$$

The element load vector $\bar{\mathbf{f}}_l^e$, stored in `fe`, is computed according to

$$\bar{\mathbf{f}}_l^e = \frac{q_{\bar{x}} L}{2}\begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

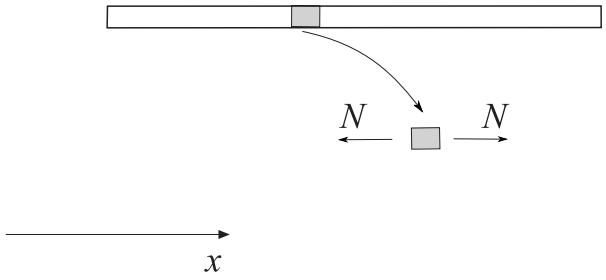
3.2.2 bar1s

Purpose

Compute normal force in a one dimensional bar element.

Syntax

```
es = cfc.bar1s(ex, ep, ed)
es = cfc.bar1s(ex, ep, ed, eq)
es, edi = cfc.bar1s(ex, ep, ed, eq, n)
es, edi, eci = cfc.bar1s(ex, ep, ed, eq, n)
```



Description

`bar1s` computes the normal force in the one dimensional bar element `bar1e`.

The input variables `ex` and `ep` are defined in `bar1e` and the element nodal displacements, stored in `ed`, are obtained by the function `extract_ed`. If distributed load is applied to the element, the variable `eq` must be included.

The number of evaluation points for normal force and displacement are determined by `n`. If `n` is omitted, only the ends of the bar are evaluated.

The output variables

$$\texttt{es}=\begin{bmatrix} N(0) \\ N(\bar{x}_2) \\ \vdots \\ N(\bar{x}_{n-1}) \\ N(L) \end{bmatrix} \qquad \texttt{edi}=\begin{bmatrix} u(0) \\ u(\bar{x}_2) \\ \vdots \\ u(\bar{x}_{n-1}) \\ u(L) \end{bmatrix} \qquad \texttt{eci}=\begin{bmatrix} 0 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_{n-1} \\ L \end{bmatrix}$$

contain the normal force, the displacement, and the evaluation points on the local $\bar{x}$-axis. L is the length of the bar element.

Theory

The nodal displacements in local coordinates are given by

$$\bar{\mathbf{a}}^e = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \end{bmatrix}$$

The transpose of $\bar{\mathbf{a}}^e$ is stored in `ed`.

The displacement $u(\bar{x})$ and the normal force $N(\bar{x})$ are computed from

$$u(\bar{x}) = \mathbf{N}\bar{\mathbf{a}}^e + u_p(\bar{x})$$

$$N(\bar{x}) = D_{EA}\mathbf{B}\bar{\mathbf{a}}^e + N_p(\bar{x})$$

where

$$\mathbf{N} = \begin{bmatrix} 1 & \bar{x} \end{bmatrix} \mathbf{C}^{-1} = \begin{bmatrix} 1 - \frac{\bar{x}}{L} & \frac{\bar{x}}{L} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 0 & 1 \end{bmatrix} \mathbf{C}^{-1} = \frac{1}{L}\begin{bmatrix} -1 & 1 \end{bmatrix}$$

$$u_p(\bar{x}) = -\frac{q_{\bar{x}}}{D_{EA}}\left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2}\right)$$

$$N_p(\bar{x}) = -q_{\bar{x}}\left(\bar{x} - \frac{L}{2}\right)$$

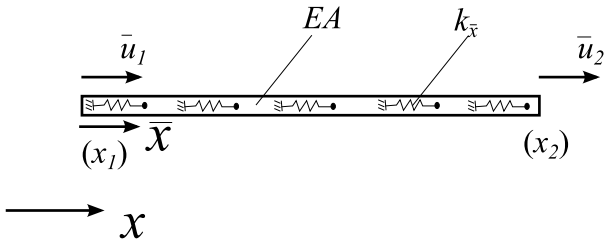
in which D_{EA}, L, and $q_{\bar{x}}$ are defined in `bar1e` and

$$\mathbf{C}^{-1} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

3.2.3 bar1we

Purpose

Compute element stiffness matrix for a one dimensional bar element with elastic support.



Syntax

```
Ke = cfc.bar1we(ex, ep)
Ke, fe = cfc.bar1we(ex, ep, eq)
```

Description

`bar1we` provides the element stiffness matrix $\bar{\mathbf{K}}^e$ for a one dimensional bar element with elastic support.

The input variables

`ex`$= [x_1 \;\; x_2]$ `ep`$= [E \; A \; k_{\bar{x}}]$

supply the element nodal coordinates x_1 and x_2, the modulus of elasticity E, the cross section area A and the stiffness of the axial springs $k_{\bar{x}}$.

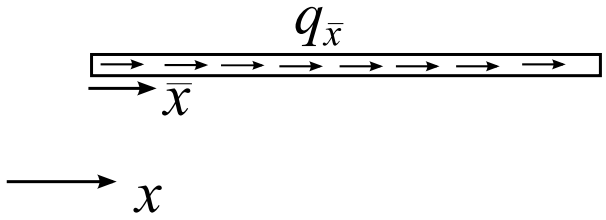
The element load vector $\bar{\mathbf{f}}_l^e$ can also be computed if a uniformly distributed load is applied to the element.

The optional input variable

`eq`$= [q_{\bar{x}}]$

contains the distributed load per unit length, $q_{\bar{x}}$.

Bar element with distributed load



Theory

The element stiffness matrix $\bar{\mathbf{K}}^e$, stored in `Ke`, is computed according to

$$\bar{\mathbf{K}}^e = \bar{\mathbf{K}}^e_0 + \bar{\mathbf{K}}^e_s$$

where

$$\bar{\mathbf{K}}^e_0 = \frac{D_{EA}}{L}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

$$\bar{\mathbf{K}}^e_s = k_{\bar{x}} L \begin{bmatrix} \frac{1}{3} & \frac{1}{6} \\ \frac{1}{6} & \frac{1}{3} \end{bmatrix}$$

where the axial stiffness D_{EA} and the length L are given by

$$D_{EA} = EA; \qquad L = x_2 - x_1$$

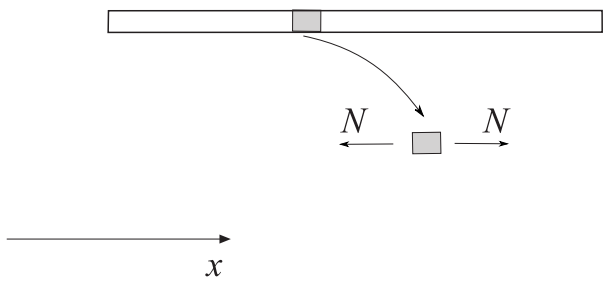
The element load vector $\bar{\mathbf{f}}^e_l$, stored in `fe`, is computed according to

$$\bar{\mathbf{f}}^e_l = \frac{q_{\bar{x}} L}{2}\begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

3.2.4 bar1ws

Purpose

Compute normal force in a one dimensional bar element with elastic support.



Syntax

```
es = cfc.bar1ws(ex, ep, ed)
es = cfc.bar1ws(ex, ep, ed, eq)
es, edi = cfc.bar1ws(ex, ep, ed, eq, n)
es, edi, eci = cfc.bar1ws(ex, ep, ed, eq, n)
```

Description

`bar1ws` computes the normal force in the one dimensional bar element `bar1ws`.

The input variables `ex` and `ep` are defined in `bar1we` and the element nodal displacements, stored in `ed`, are obtained by the function `_ed`. If distributed load is applied to the element, the variable `eq` must be included.

The number of evaluation points for normal force and displacement are determined by `n`. If `n` is omitted, only the ends of the bar are evaluated.

The output variables are:

$$\texttt{es}=\begin{bmatrix} N(0) \\ N(\bar{x}_2) \\ \vdots \\ N(\bar{x}_{n-1}) \\ N(L) \end{bmatrix} \qquad \texttt{edi}=\begin{bmatrix} u(0) \\ u(\bar{x}_2) \\ \vdots \\ u(\bar{x}_{n-1}) \\ u(L) \end{bmatrix} \qquad \texttt{eci}=\begin{bmatrix} 0 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_{n-1} \\ L \end{bmatrix}$$

These contain the normal force, the displacement, and the evaluation points on the local $\bar{x}$-axis. L is the length of the bar element.

Theory

The nodal displacements in local coordinates are given by

$$\bar{\mathbf{a}}^e = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \end{bmatrix}$$

The transpose of $\bar{\mathbf{a}}^e$ is stored in `ed`.

The displacement $u(\bar{x})$ and the normal force $N(\bar{x})$ are computed from

$$u(\bar{x}) = \mathbf{N}\bar{\mathbf{a}}^e + u_p(\bar{x})$$

$$N(\bar{x}) = D_{EA}\mathbf{B}\bar{\mathbf{a}}^e + N_p(\bar{x})$$

where

$$\mathbf{N} = \begin{bmatrix} 1 & \bar{x} \end{bmatrix} \mathbf{C}^{-1} = \begin{bmatrix} 1-\frac{\bar{x}}{L} & \frac{\bar{x}}{L} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 0 & 1 \end{bmatrix} \mathbf{C}^{-1} = \frac{1}{L}\begin{bmatrix} -1 & 1 \end{bmatrix}$$

$$u_p(\bar{x}) = \frac{k_{\bar{x}}}{D_{EA}}\begin{bmatrix} \frac{\bar{x}^2 - L\bar{x}}{2} & \frac{\bar{x}^3 - L^2\bar{x}}{6} \end{bmatrix} \mathbf{C}^{-1}\bar{\mathbf{a}}^e - \frac{q_{\bar{x}}}{D_{EA}}\left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2}\right)$$

$$N_p(\bar{x}) = k_{\bar{x}}\begin{bmatrix} \frac{2\bar{x} - L}{2} & \frac{3\bar{x}^2 - L^2}{6} \end{bmatrix} \mathbf{C}^{-1}\bar{\mathbf{a}}^e - q_{\bar{x}}\left(\bar{x} - \frac{L}{2}\right)$$

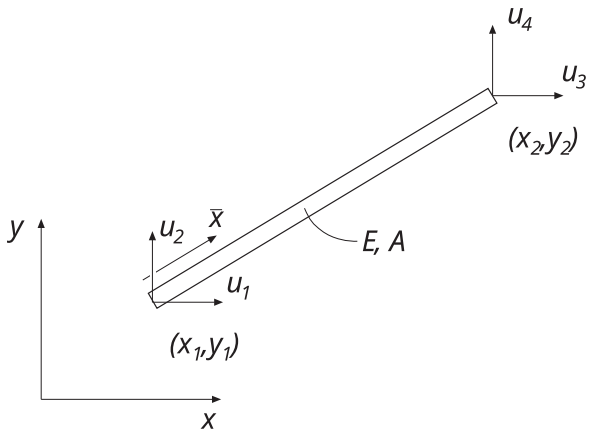
in which D_{EA}, L, $k_{\bar{x}}$ and $q_{\bar{x}}$ are defined in `bar1we` and

$$\mathbf{C}^{-1} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

3.2.5 bar2e

Purpose

Compute element stiffness matrix for a two dimensional bar element.



Syntax

```
Ke = cfc.bar2e(ex, ey, ep)
Ke, fe = cfc.bar2e(ex, ey, ep, eq)
```

Description

`bar2e` provides the global element stiffness matrix $\mathbf{K}^e$ for a two dimensional bar element.

The input variables

`ex`$= [x_1 \;\; x_2]$ `ey`$= [y_1 \;\; y_2]$ `ep`$= [E \; A]$

supply the element nodal coordinates x_1, y_1, x_2, and y_2, the modulus of elasticity E, and the cross section area A.

The element load vector $\mathbf{f}_l^e$ can also be computed if a uniformly distributed load is applied to the element. The optional input variable

`eq`$= [q_{\bar{x}}]$

contains the distributed load per unit length, $q_{\bar{x}}$.

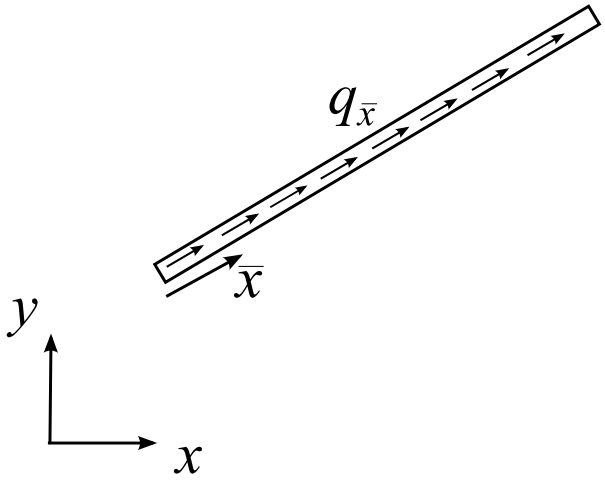
Theory

The element stiffness matrix $\mathbf{K}^e$, stored in `Ke`, is computed according to

$$\mathbf{K}^e = \mathbf{G}^T \, \bar{\mathbf{K}}^e \, \mathbf{G}$$

where

$$\bar{\mathbf{K}}^e = \frac{D_{EA}}{L} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \qquad \mathbf{G} = \begin{bmatrix} n_{x\bar{x}} & n_{y\bar{x}} & 0 & 0 \\ 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} \end{bmatrix}$$



where the axial stiffness D_{EA} and the length L are given by

$$D_{EA} = EA; \qquad L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

and the transformation matrix **G** contains the direction cosines

$$n_{x\bar{x}} = \frac{x_2 - x_1}{L} \qquad n_{y\bar{x}} = \frac{y_2 - y_1}{L}$$

The element load vector $\mathbf{f}_l^e$, stored in `fe`, is computed according to

$$\mathbf{f}_l^e = \mathbf{G}^T \, \bar{\mathbf{f}}_l^e$$

where

$$\bar{\mathbf{f}}_l^e = \frac{q_{\bar{x}} L}{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

3.2.6 bar2s

Purpose

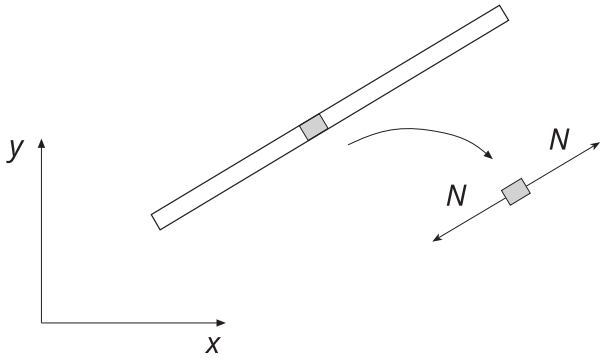
Compute normal force in a two dimensional bar element.

Syntax

```
es = cfc.bar2s(ex, ey, ep, ed)
es = cfc.bar2s(ex, ey, ep, ed, eq)
es, edi = cfc.bar2s(ex, ey, ep, ed, eq, n)
es, edi, eci = cfc.bar2s(ex, ey, ep, ed, eq, n)
```

Description

`bar2s` computes the normal force in the two dimensional bar element `bar2e`.



The input variables `ex`, `ey`, and `ep` are defined in `bar2e` and the element nodal displacements, stored in `ed`, are obtained by the function `extract_ed`. If distributed loads are applied to the element, the variable `eq` must be included. The number of evaluation points for section forces and displacements are determined by `n`. If `n` is omitted, only the ends of the bar are evaluated.

The output variables

$$\texttt{es}=\begin{bmatrix} N(0) \\ N(\bar{x}_2) \\ \vdots \\ N(\bar{x}_{n-1}) \\ N(L) \end{bmatrix} \qquad \texttt{edi}=\begin{bmatrix} u(0) \\ u(\bar{x}_2) \\ \vdots \\ u(\bar{x}_{n-1}) \\ u(L) \end{bmatrix} \qquad \texttt{eci}=\begin{bmatrix} 0 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_{n-1} \\ L \end{bmatrix}$$

contain the normal force, the displacement, and the evaluation points on the local $\bar{x}$-axis. L is the length of the bar element.

Theory

The nodal displacements in global coordinates

$$\mathbf{a}^e = \begin{bmatrix} u_1 & u_2 & u_3 & u_4 \end{bmatrix}^T$$

are also shown in `bar2e`. The transpose of $\mathbf{a}^e$ is stored in `ed`.

The nodal displacements in local coordinates are given by

$$\bar{\mathbf{a}}^e = \mathbf{G}\mathbf{a}^e$$

where the transformation matrix `G` is defined in `bar2e`.

The displacement $u(\bar{x})$ and the normal force $N(\bar{x})$ are computed from

$$u(\bar{x}) = \mathbf{N}\bar{\mathbf{a}}^e + u_p(\bar{x})$$

$$N(\bar{x}) = D_{EA}\mathbf{B}\bar{\mathbf{a}}^e + N_p(\bar{x})$$

where

$$\mathbf{N} = \begin{bmatrix} 1 & \bar{x} \end{bmatrix} \mathbf{C}^{-1} = \begin{bmatrix} 1 - \frac{\bar{x}}{L} & \frac{\bar{x}}{L} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 0 & 1 \end{bmatrix} \mathbf{C}^{-1} = \frac{1}{L} \begin{bmatrix} -1 & 1 \end{bmatrix}$$

$$u_p(\bar{x}) = -\frac{q_{\bar{x}}}{D_{EA}} \left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2} \right)$$

$$N_p(\bar{x}) = -q_{\bar{x}} \left(\bar{x} - \frac{L}{2} \right)$$

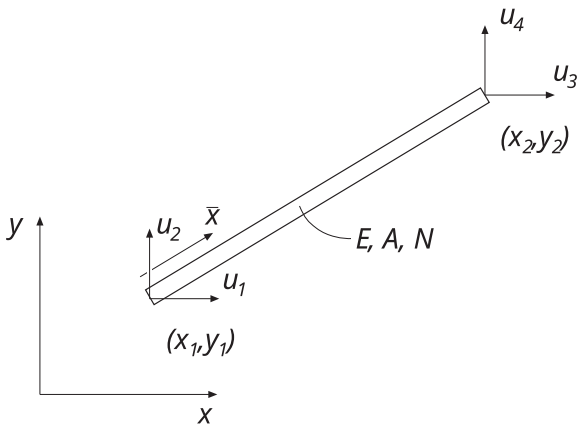
where D_{EA}, L, $q_{\bar{x}}$ are defined in `bar2e` and

$$\mathbf{C}^{-1} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

3.2.7 bar2ge

Purpose

Compute element stiffness matrix for a two dimensional geometric nonlinear bar.



Syntax

```
Ke = cfc.bar2ge(ex, ey, ep, Qx)
```

Description

`bar2ge` provides the element stiffness matrix $\mathbf{K}^e$ for a two dimensional geometric nonlinear bar element.

The input variables

`ex`$= [x_1 \;\; x_2]$ `ey`$= [y_1 \;\; y_2]$ `ep`$= [E \; A]$

supply the element nodal coordinates x_1, y_1, x_2, and y_2, the modulus of elasticity E, and the cross section area A.

The input variable

`Qx`$= [Q_{\bar{x}}]$

contains the value of the axial force, which is positive in tension.

Theory

The global element stiffness matrix $\mathbf{K}^e$, stored in `Ke`, is computed according to

$$\mathbf{K}^e = \mathbf{G}^T \, \overset{\sim}{\mathbf{K}}{}^e \, \mathbf{G}$$

where

$$\overset{\sim}{\mathbf{K}}{}^e = \frac{D_{EA}}{L} \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \frac{Q_{\bar{x}}}{L} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} n_{x\bar{x}} & n_{y\bar{x}} & 0 & 0 \\ n_{x\bar{y}} & n_{y\bar{y}} & 0 & 0 \\ 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} \\ 0 & 0 & n_{x\bar{y}} & n_{y\bar{y}} \end{bmatrix}$$

where the axial stiffness D_{EA} and the length L are given by

$$D_{EA} = EA \qquad L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

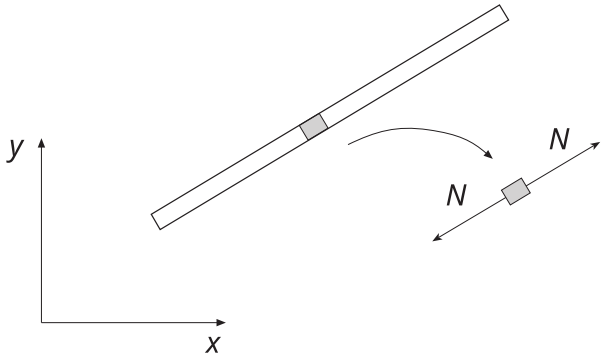
and the transformation matrix $\mathbf{G}$ contains the direction cosines

$$n_{x\bar{x}} = n_{y\bar{y}} = \frac{x_2 - x_1}{L} \qquad n_{y\bar{x}} = -n_{x\bar{y}} = \frac{y_2 - y_1}{L}$$

3.2.8 bar2gs

Purpose

Compute axial force and normal force in a two dimensional bar element.



Syntax

```
es, Qx = cfc.bar2gs(ex, ey, ep, ed)
es, Qx, edi = cfc.bar2gs(ex, ey, ep, ed, n)
es, Qx, edi, eci = cfc.bar2gs(ex, ey, ep, ed, n)
```

Description

`bar2gs` computes the normal force in the two dimensional bar elements `bar2ge`.

The input variables `ex`, `ey`, and `ep` are defined in `bar2ge` and the element nodal displacements, stored in `ed`, are obtained by the function `extract_ed`. The number of evaluation points for section forces and displacements are determined by `n`. If `n` is omitted, only the ends of the bar are evaluated.

The output variable `Qx` contains the axial force $Q_{\bar{x}}$ and the output variables

$$\texttt{es}=\begin{bmatrix} N(0) \\ N(\bar{x}_2) \\ \vdots \\ N(\bar{x}_{n-1}) \\ N(L) \end{bmatrix} \qquad \texttt{edi}=\begin{bmatrix} u(0) \\ u(\bar{x}_2) \\ \vdots \\ u(\bar{x}_{n-1}) \\ u(L) \end{bmatrix} \qquad \texttt{eci}=\begin{bmatrix} 0 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_{n-1} \\ L \end{bmatrix}$$

contain the normal force, the displacement, and the evaluation points on the local $\bar{x}$-axis. L is the length of the bar element.

Theory

The nodal displacements in global coordinates are given by

$$\mathbf{a}^e = \begin{bmatrix} u_1 & u_2 & u_3 & u_4 \end{bmatrix}^T$$

The transpose of $\mathbf{a}^e$ is stored in `ed`. The nodal displacements in local coordinates are given by

$$\bar{\mathbf{a}}^e = \mathbf{G}\mathbf{a}^e$$

where the transformation matrix `G` is defined in `bar2ge`. The displacements associated with bar action are determined as

$$\bar{\mathbf{a}}^e_{\text{bar}} = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_3 \end{bmatrix}$$

The displacement $u(\bar{x})$ and the normal force $N(\bar{x})$ are computed from

$$u(\bar{x}) = \mathbf{N}\bar{\mathbf{a}}^e_{\text{bar}}$$

$$N(\bar{x}) = D_{EA}\mathbf{B}\bar{\mathbf{a}}^e_{\text{bar}}$$

where

$$\mathbf{N} = \begin{bmatrix} 1 & \bar{x} \end{bmatrix}\mathbf{C}^{-1} = \begin{bmatrix} 1-\frac{\bar{x}}{L} & \frac{\bar{x}}{L} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 0 & 1 \end{bmatrix}\mathbf{C}^{-1} = \frac{1}{L}\begin{bmatrix} -1 & 1 \end{bmatrix}$$

where D_{EA} and L are defined in `bar2ge` and

$$\mathbf{C}^{-1} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

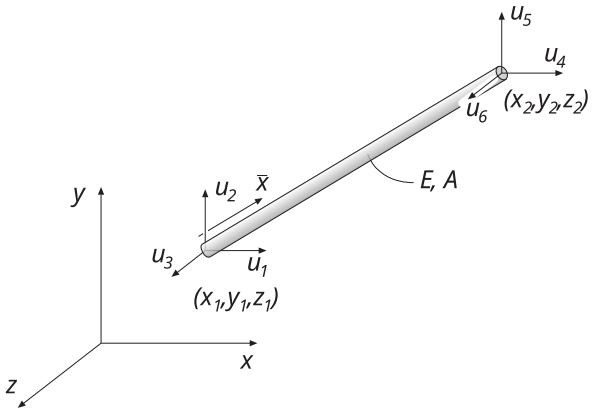
An updated value of the axial force is computed as

$$Q_{\bar{x}} = N(0)$$

3.2.9 bar3e

Purpose

Compute element stiffness matrix for a three dimensional bar element.



Syntax

```
Ke = cfc.bar3e(ex, ey, ez, ep)
Ke, fe = cfc.bar3e(ex, ey, ez, ep, eq)
```

Description

`bar3e` provides the global element stiffness matrix $\mathbf{K}^e$ for a three dimensional bar element.

The input variables

`ex`$= [x_1 \;\; x_2]$ `ey`$= [y_1 \;\; y_2]$ `ez`$= [z_1 \;\; z_2]$ `ep`$= [E \; A]$

supply the element nodal coordinates x_1, y_1, z_1, x_2, y_2, and z_2, the modulus of elasticity E, and the cross section area A.

The element load vector `fe` can also be computed if a uniformly distributed axial load is applied to the element. The optional input variable

`eq`$= [q_{\bar{x}}]$

contains the distributed load per unit length, $q_{\bar{x}}$.

Theory

The element stiffness matrix $\mathbf{K}^e$, stored in `Ke`, is computed according to

$$\mathbf{K}^e = \mathbf{G}^T \; \bar{\mathbf{K}}^e \; \mathbf{G}$$

where

$$\bar{\mathbf{K}}^e = \frac{D_{EA}}{L} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \qquad \mathbf{G} = \begin{bmatrix} n_{x\bar{x}} & n_{y\bar{x}} & n_{z\bar{x}} & 0 & 0 & 0 \\ 0 & 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} & n_{z\bar{x}} \end{bmatrix}$$

where the axial stiffness D_{EA} and the length L are given by

$$D_{EA} = EA \qquad L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

and the transformation matrix **G** contains the direction cosines

$$n_{x\bar{x}} = \frac{x_2 - x_1}{L} \qquad n_{y\bar{x}} = \frac{y_2 - y_1}{L} \qquad n_{z\bar{x}} = \frac{z_2 - z_1}{L}$$

The element load vector $\mathbf{f}_l^e$, stored in `fe`, is computed according to

$$\mathbf{f}_l^e = \mathbf{G}^T \, \bar{\mathbf{f}}_l^e$$

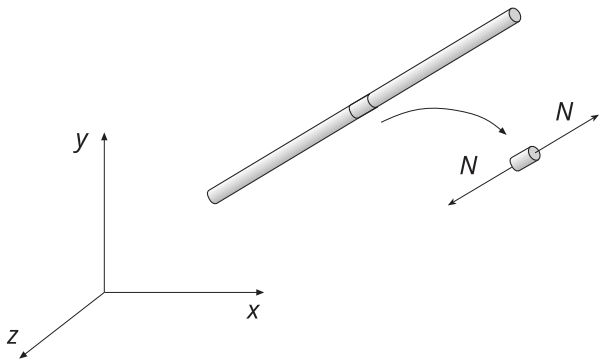
where

$$\bar{\mathbf{f}}_l^e = \frac{q_{\bar{x}} L}{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

3.2.10 bar3s

Purpose

Compute normal force in a three dimensional bar element.



Syntax

```
es = cfc.bar3s(ex, ey, ez, ep, ed)
es = cfc.bar3s(ex, ey, ez, ep, ed, eq)
es, edi = cfc.bar3s(ex, ey, ez, ep, ed, eq, n)
es, edi, eci = cfc.bar3s(ex, ey, ez, ep, ed, eq, n)
```

Description

`bar3s` computes the normal force in a three dimensional bar element (see `bar3e`).

The input variables `ex`, `ey`, and `ep` are defined in `bar3e` and the element nodal displacements, stored in `ed`, are obtained by the function `extract_ed`. The number of evaluation points for section forces and displacements are determined by `n`. If `n` is omitted, only the ends of the bar are evaluated.

The output variables

$$\texttt{es}=\begin{bmatrix} N(0) \\ N(\bar{x}_2) \\ \vdots \\ N(\bar{x}_{n-1}) \\ N(L) \end{bmatrix} \qquad \texttt{edi}=\begin{bmatrix} u(0) \\ u(\bar{x}_2) \\ \vdots \\ u(\bar{x}_{n-1}) \\ u(L) \end{bmatrix} \qquad \texttt{eci}=\begin{bmatrix} 0 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_{n-1} \\ L \end{bmatrix}$$

contain the normal force, the displacement, and the evaluation points on the local $\bar{x}$-axis. L is the length of the bar element.

Theory

The nodal displacements in global coordinates are given by

$$\mathbf{a}^e = \begin{bmatrix} u_1 & u_2 & u_3 & u_4 & u_5 & u_6 \end{bmatrix}^T$$

The transpose of $\mathbf{a}^e$ is stored in `ed`.

The nodal displacements in local coordinates are given by

$$\bar{\mathbf{a}}^e = \mathbf{G}\mathbf{a}^e$$

where the transformation matrix `G` is defined in `bar3e`.

The displacement $u(\bar{x})$ and the normal force $N(\bar{x})$ are computed from

$$u(\bar{x}) = \mathbf{N}\bar{\mathbf{a}}^e + u_p(\bar{x})$$

$$N(\bar{x}) = D_{EA}\mathbf{B}\bar{\mathbf{a}}^e + N_p(\bar{x})$$

where

$$\mathbf{N} = \begin{bmatrix} 1 & \bar{x} \end{bmatrix} \mathbf{C}^{-1} = \begin{bmatrix} 1 - \frac{\bar{x}}{L} & \frac{\bar{x}}{L} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 0 & 1 \end{bmatrix} \mathbf{C}^{-1} = \frac{1}{L}\begin{bmatrix} -1 & 1 \end{bmatrix}$$

$$u_p(\bar{x}) = -\frac{q_{\bar{x}}}{D_{EA}}\left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2}\right)$$

$$N_p(\bar{x}) = -q_{\bar{x}}\left(\bar{x} - \frac{L}{2}\right)$$

where D_{EA}, L, $q_{\bar{x}}$ are defined in `bar3e` and

$$\mathbf{C}^{-1} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

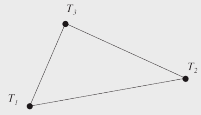
3.3 Heat Flow Elements

Heat flow elements are available for one, two, and three dimensional analysis. For one dimensional heat flow the spring element `spring1` is used.

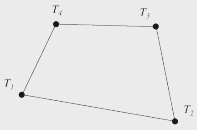
A variety of important physical phenomena are described by the same differential equation as the heat flow problem. The heat flow element is thus applicable in modelling different physical applications. Table below shows the relation between the primary variable **a**, the constitutive matrix **D**, and the load vector **f_l** for a chosen set of two dimensional physical problems.

Table 7: Problem dependent parameters

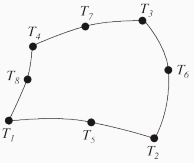
Problem type	**a**	**D**	**f_l**	Designation
Heat flow	T	λ_x, λ_y	Q	T **= temperature** λ_x, λ_y = thermal conductivity Q = heat supply
Groundwater flow	ϕ	k_x, k_y	Q	ϕ **= piezometric head** k_x, k_y = permeabilities Q = fluid supply
St. Venant torsion	ϕ	$1/G_{zy}$, $1/G_{zx}$	2Θ	ϕ **= stress function** G_{zy}, G_{zx} = shear moduli Θ = angle of torsion per unit length



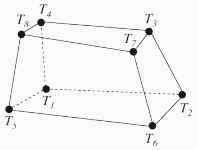
flw2te



flw2qe, flw2i4e



flw2i8e



flw3i8e

Table 8: 2D heat flow functions

flw2te	Compute element matrices for a triangular element
flw2ts	Compute temperature gradients and flux
flw2qe	Compute element matrices for a quadrilateral element
flw2qs	Compute temperature gradients and flux
flw2i4e	Compute element matrices, 4 node isoparametric element
flw2i4s	Compute temperature gradients and flux
flw2i8e	Compute element matrices, 8 node isoparametric element
flw2i8s	Compute temperature gradients and flux

Table 9: 3D heat flow functions

flw3i8e	Compute element matrices, 8 node isoparametric element
flw3i8s	Compute temperature gradients and flux

3.3.1 flw2te

Purpose

Compute element stiffness matrix for a triangular heat flow element.

alt

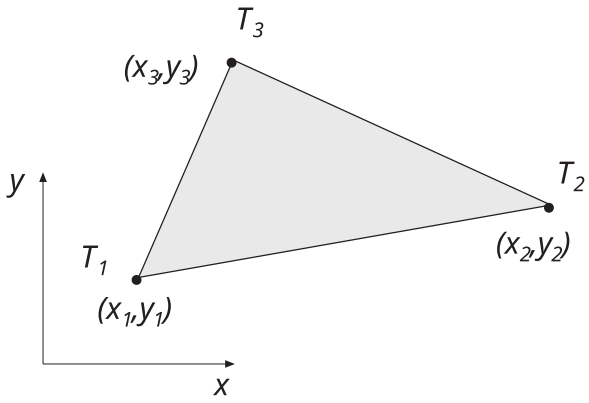
Two dimensional heat flow elements

Syntax

```
Ke = flw2te(ex, ey, ep, D)
[Ke, fe] = flw2te(ex, ey, ep, D, eq)
```

Description

`flw2te` provides the element stiffness (conductivity) matrix `Ke` and the element load vector `fe` for a triangular heat flow element.



The element nodal coordinates x_1, y_1, x_2 etc, are supplied to the function by `ex` and `ey`, the element thickness t is supplied by `ep` and the thermal conductivities (or corresponding quantities) k_{xx}, k_{xy} etc are supplied by `D`.

$$\mathbf{ex} = [\, x_1 \;\; x_2 \;\; x_3 \,] \qquad \mathbf{ep} = [\, t \,] \qquad \mathbf{D} = \begin{bmatrix} k_{xx} & k_{xy} \\ k_{yx} & k_{yy} \end{bmatrix}$$
$$\mathbf{ey} = [\, y_1 \;\; y_2 \;\; y_3 \,]$$

If the scalar variable `eq` is given in the function, the element load vector **fe** is computed, using

$$\mathbf{eq} = [\, Q \,]$$

where Q is the heat supply per unit volume.

Theory

The element stiffness matrix $\mathbf{K}^e$ and the element load vector $\mathbf{f}_l^e$, stored in `Ke` and `fe`, respectively, are computed according to

$$\mathbf{K}^e = (\mathbf{C}^{-1})^T \int_A \bar{\mathbf{B}}^T \mathbf{D} \bar{\mathbf{B}}\, t\, dA\, \mathbf{C}^{-1}$$

$$\mathbf{f}_l^e = (\mathbf{C}^{-1})^T \int_A \bar{\mathbf{N}}^T Q\, t\, dA$$

with the constitutive matrix **D** defined by `D`.

The evaluation of the integrals for the triangular element is based on the linear temperature approximation $T(x, y)$ and is expressed in terms of the nodal variables T_1, T_2 and T_3 as

$$T(x, y) = \mathbf{N}^e \mathbf{a}^e = \bar{\mathbf{N}}\, \mathbf{C}^{-1} \mathbf{a}^e$$

where

$$\bar{\mathbf{N}} = [\, 1 \;\; x \;\; y \,] \qquad \mathbf{C} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} \qquad \mathbf{a}^e = \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix}$$

and hence it follows that

$$\bar{\mathbf{B}} = \nabla\bar{\mathbf{N}} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \nabla = \begin{bmatrix} \dfrac{\partial}{\partial x} \\ \dfrac{\partial}{\partial y} \end{bmatrix}$$

Evaluation of the integrals for the triangular element yields

$$\mathbf{K}^e = (\mathbf{C}^{-1})^T \bar{\mathbf{B}}^T \mathbf{D} \bar{\mathbf{B}} \mathbf{C}^{-1} t A$$

$$\mathbf{f}_l^e = \frac{QAt}{3} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

where the element area A is determined as

$$A = \frac{1}{2} \det \mathbf{C}$$

3.3.2 flw2ts

Purpose

Compute heat flux and temperature gradients in a triangular heat flow element.

Syntax

```
[es, et] = flw2ts(ex, ey, D, ed)
```

Description

`flw2ts` computes the heat flux vector `es` and the temperature gradient `et` (or corresponding quantities) in a triangular heat flow element.

The input variables `ex`, `ey` and the matrix `D` are defined in `flw2te`. The vector `ed` contains the nodal temperatures $\mathbf{a}^e$ of the element and is obtained by the function `extract` as

$$\mathbf{ed} = (\mathbf{a}^e)^T = [\ T_1 \ \ T_2 \ \ T_3\ ]$$

The output variables

$$\mathbf{es} = \mathbf{q}^T = [\ q_x \ q_y\ ]$$

$$\mathbf{et} = (\nabla T)^T = \left[\ \frac{\partial T}{\partial x} \ \ \frac{\partial T}{\partial y}\ \right]$$

contain the components of the heat flux and the temperature gradient computed in the directions of the coordinate axis.

Theory

The temperature gradient and the heat flux are computed according to

$$\nabla T = \bar{\mathbf{B}}\ \mathbf{C}^{-1}\ \mathbf{a}^e$$

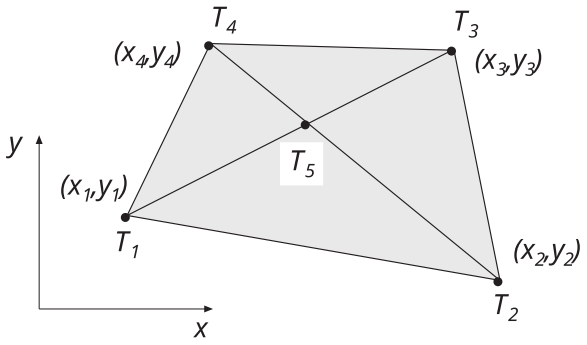
$$\mathbf{q} = -\mathbf{D}\nabla T$$

where the matrices $\mathbf{D}$, $\bar{\mathbf{B}}$, and $\mathbf{C}$ are described in `flw2te`. Note that both the temperature gradient and the heat flux are constant in the element.

3.3.3 flw2qe

Purpose

Compute element stiffness matrix for a quadrilateral heat flow element.



Syntax

```
Ke = flw2qe(ex, ey, ep, D)
[Ke, fe] = flw2qe(ex, ey, ep, D, eq)
```

Description

`flw2qe` provides the element stiffness (conductivity) matrix `Ke` and the element load vector `fe` for a quadrilateral heat flow element.

The element nodal coordinates x_1, y_1, x_2 etc, are supplied to the function by `ex` and `ey`, the element thickness t is supplied by `ep` and the thermal conductivities (or corresponding quantities) k_{xx}, k_{xy} etc are supplied by `D`.

$$\mathbf{ex} = [\, x_1 \;\; x_2 \;\; x_3 \;\; x_4 \,] \qquad \mathbf{ep} = [\, t \,] \qquad \mathbf{D} = \begin{bmatrix} k_{xx} & k_{xy} \\ k_{yx} & k_{yy} \end{bmatrix}$$
$$\mathbf{ey} = [\, y_1 \;\; y_2 \;\; y_3 \;\; y_4 \,]$$

If the scalar variable `eq` is given in the function, the element load vector **fe** is computed, using

$$\mathbf{eq} = [\, Q \,]$$

where Q is the heat supply per unit volume.

Theory

In computing the element matrices, a fifth degree of freedom is introduced. The location of this extra degree of freedom is defined by the mean value of the coordinates in the corner points. Four sets of element matrices are calculated using `flw2te`. These matrices are then assembled and the fifth degree of freedom is eliminated by static condensation.

3.3.4 flw2qs

Purpose

Compute heat flux and temperature gradients in a quadrilateral heat flow el-

ement.

Syntax

```
[es, et] = flw2qs(ex, ey, ep, D, ed)
[es, et] = flw2qs(ex, ey, ep, D, ed, eq)
```

Description

`flw2qs` computes the heat flux vector `es` and the temperature gradient `et` (or corresponding quantities) in a quadrilateral heat flow element.

The input variables `ex`, `ey`, `eq` and the matrix `D` are defined in `flw2qe`. The vector `ed` contains the nodal temperatures $\mathbf{a}^e$ of the element and is obtained by the function `extract` as

$$\mathbf{ed} = (\mathbf{a}^e)^T = [\ T_1\ \ T_2\ \ T_3\ \ T_4\ ]$$

The output variables

$$\mathbf{es} = \mathbf{q}^T = [\ q_x\ q_y\ ]$$

$$\mathbf{et} = (\nabla T)^T = \left[\ \frac{\partial T}{\partial x}\ \ \frac{\partial T}{\partial y}\ \right]$$

contain the components of the heat flux and the temperature gradient computed in the directions of the coordinate axis.

Theory

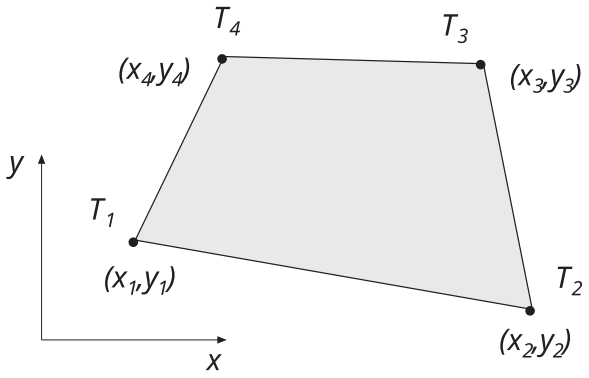
By assembling four triangular elements as described in `flw2te` a system of equations containing 5 degrees of freedom is obtained. From this system of equations the unknown temperature at the center of the element is computed. Then according to the description in `flw2ts` the temperature gradient and the heat flux in each of the four triangular elements are produced. Finally the temperature gradient and the heat flux of the quadrilateral element are computed as area weighted mean values from the values of the four triangular elements. If heat is supplied to the element, the element load vector `eq` is needed for the calculations.

> **Note**
>
> If the input variables are given for a number of identical (`nie`) elements, i.e. `Ex`, `Ey`, and `Ed` are matrices, then the output variables are defined as
>
> $$\mathrm{Es} = \begin{bmatrix} q_x^1 & q_y^1 \\ q_x^2 & q_y^2 \\ \vdots & \vdots \\ q_x^{nie} & q_y^{nie} \end{bmatrix} \qquad \mathrm{Et} = \begin{bmatrix} \frac{\partial T}{\partial x}^1 & \frac{\partial T}{\partial y}^1 \\ \frac{\partial T}{\partial x}^2 & \frac{\partial T}{\partial y}^2 \\ \vdots & \vdots \\ \frac{\partial T}{\partial x}^{nie} & \frac{\partial T}{\partial y}^{nie} \end{bmatrix}$$
>
> where $\mathbf{q}^i$ and ∇T^i are computed from the nodal values located in column `i` of `Ed`.

3.3.5 flw2i4e

Purpose

Compute element stiffness matrix for a 4 node isoparametric heat flow element.



Syntax

```
Ke = flw2i4e(ex, ey, ep, D)
[Ke, fe] = flw2i4e(ex, ey, ep, D, eq)
```

Description

`flw2i4e` provides the element stiffness (conductivity) matrix `Ke` and the element load vector `fe` for a 4 node isoparametric heat flow element.

The element nodal coordinates x_1, y_1, x_2 etc, are supplied to the function by `ex` and `ey`. The element thickness t and the number of Gauss points n ($n \times n$ integration points, $n = 1, 2, 3$) are supplied to the function by `ep` and the thermal conductivities (or corresponding quantities) k_{xx}, k_{xy} etc are supplied by `D`.

$$\mathbf{ex} = [\, x_1 \;\; x_2 \;\; x_3 \;\; x_4 \,] \qquad \mathbf{ep} = [\, t \;\; n \,] \qquad \mathbf{D} = \begin{bmatrix} k_{xx} & k_{xy} \\ k_{yx} & k_{yy} \end{bmatrix}$$
$$\mathbf{ey} = [\, y_1 \;\; y_2 \;\; y_3 \;\; y_4 \,]$$

If the scalar variable `eq` is given in the function, the element load vector fe is computed, using

$$\mathbf{eq} = [\, Q \,]$$

where Q is the heat supply per unit volume.

Theory

The element stiffness matrix $\mathbf{K}^e$ and the element load vector $\mathbf{f}_l^e$, stored in `Ke` and `fe`, respectively, are computed according to

$$\mathbf{K}^e = \int_A \mathbf{B}^{eT} \mathbf{D} \mathbf{B}^e t \, dA$$

$$\mathbf{f}_l^e = \int_A \mathbf{N}^{eT} Q t \, dA$$

with the constitutive matrix **D** defined by `D`.

The evaluation of the integrals for the isoparametric 4 node element is based on a temperature approximation $T(\xi,\eta)$, expressed in a local coordinate system in terms of the nodal variables T_1, T_2, T_3 and T_4 as

$$T(\xi,\eta) = \mathbf{N}^e \mathbf{a}^e$$

where

$$\mathbf{N}^e = [\, N_1^e \;\; N_2^e \;\; N_3^e \;\; N_4^e \,] \qquad \mathbf{a}^e = [\, T_1 \;\; T_2 \;\; T_3 \;\; T_4 \,]^T$$

The element shape functions are given by

$$N_1^e = \frac{1}{4}(1-\xi)(1-\eta) \qquad N_2^e = \frac{1}{4}(1+\xi)(1-\eta)$$
$$N_3^e = \frac{1}{4}(1+\xi)(1+\eta) \qquad N_4^e = \frac{1}{4}(1-\xi)(1+\eta)$$

The $\mathbf{B}^e$-matrix is given by

$$\mathbf{B}^e = \nabla \mathbf{N}^e = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix} \mathbf{N}^e = (\mathbf{J}^T)^{-1} \begin{bmatrix} \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \end{bmatrix} \mathbf{N}^e$$

where **J** is the Jacobian matrix

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix}$$

Evaluation of the integrals is done by Gauss integration.

3.3.6 flw2i4s

Purpose

Compute heat flux and temperature gradients in a 4 node isoparametric heat flow element.

Syntax

```
[es, et, eci] = flw2i4s(ex, ey, ep, D, ed)
```

Description

`flw2i4s` computes the heat flux vector `es` and the temperature gradient `et` (or corresponding quantities) in a 4 node isoparametric heat flow element.

The input variables **ex**, **ey**, **ep** and the matrix **D** are defined in `flw2i4e`. The vector **ed** contains the nodal temperatures $\mathbf{a}^e$ of the element and is obtained by `extract` as

$$\mathbf{ed} = (\mathbf{a}^e)^T = [\, T_1 \;\; T_2 \;\; T_3 \;\; T_4 \,]$$

The output variables

$$\mathbf{es} = \bar{\mathbf{q}}^T = \begin{bmatrix} q_x^1 & q_y^1 \\ q_x^2 & q_y^2 \\ \vdots & \vdots \\ q_x^{n^2} & q_y^{n^2} \end{bmatrix}$$

$$\mathbf{et} = (\bar{\nabla} T)^T = \begin{bmatrix} \frac{\partial T}{\partial x}^1 & \frac{\partial T}{\partial y}^1 \\ \frac{\partial T}{\partial x}^2 & \frac{\partial T}{\partial y}^2 \\ \vdots & \vdots \\ \frac{\partial T}{\partial x}^{n^2} & \frac{\partial T}{\partial y}^{n^2} \end{bmatrix}$$

$$\mathbf{eci} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_{n^2} & y_{n^2} \end{bmatrix}$$

contain the heat flux, the temperature gradient, and the coordinates of the integration points. The index n denotes the number of integration points used within the element, cf. `flw2i4e`.

Theory

The temperature gradient and the heat flux are computed according to

$$\nabla T = \mathbf{B}^e \, \mathbf{a}^e$$

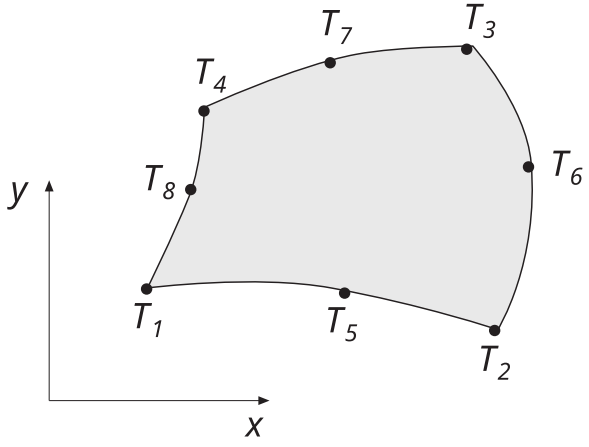
$$\mathbf{q} = -\mathbf{D} \nabla T$$

where the matrices $\mathbf{D}$, $\mathbf{B}^e$, and $\mathbf{a}^e$ are described in `flw2i4e`, and where the integration points are chosen as evaluation points.

3.3.7 flw2i8e

Purpose

Compute element stiffness matrix for an 8 node isoparametric heat flow element.



Syntax

```
Ke = flw2i8e(ex, ey, ep, D)
[Ke, fe] = flw2i8e(ex, ey, ep, D, eq)
```

Description

`flw2i8e` provides the element stiffness (conductivity) matrix `Ke` and the element load vector `fe` for an 8 node isoparametric heat flow element.

The element nodal coordinates x_1, y_1, x_2 etc, are supplied to the function by **ex** and **ey**. The element thickness t and the number of Gauss points n ($n \times n$ integration points, $n = 1, 2, 3$) are supplied to the function by **ep** and the thermal conductivities (or corresponding quantities) k_{xx}, k_{xy} etc are supplied by `D`.

$$\mathbf{ex} = [\, x_1 \;\; x_2 \;\; x_3 \;\; \dots \;\; x_8 \,] \qquad \mathbf{ep} = [\, t \;\; n \,] \qquad \mathbf{D} = \begin{bmatrix} k_{xx} & k_{xy} \\ k_{yx} & k_{yy} \end{bmatrix}$$
$$\mathbf{ey} = [\, y_1 \;\; y_2 \;\; y_3 \;\; \dots \;\; y_8 \,]$$

If the scalar variable `eq` is given in the function, the vector **fe** is computed, using

$$\mathbf{eq} = [\, Q \,]$$

where Q is the heat supply per unit volume.

Theory

The element stiffness matrix $\mathbf{K}^e$ and the element load vector $\mathbf{f}_l^e$, stored in `Ke` and `fe`, respectively, are computed according to

$$\mathbf{K}^e = \int_A \mathbf{B}^{eT} \mathbf{D} \mathbf{B}^e t \, dA$$

$$\mathbf{f}_l^e = \int_A \mathbf{N}^{eT} Q t \, dA$$

with the constitutive matrix **D** defined by `D`.

The evaluation of the integrals for the 2D isoparametric 8 node element is based on a temperature approximation $T(\xi, \eta)$, expressed in a local coordinates system in terms of the nodal variables T_1 to T_8 as

$$T(\xi, \eta) = \mathbf{N}^e \mathbf{a}^e$$

where

$$\mathbf{N}^e = [\, N_1^e \;\; N_2^e \;\; N_3^e \;\; \dots \;\; N_8^e \,] \qquad \mathbf{a}^e = [\, T_1 \;\; T_2 \;\; T_3 \;\; \dots \;\; T_8 \,]^T$$

The element shape functions are given by

$$N_1^e = -\frac{1}{4}(1-\xi)(1-\eta)(1+\xi+\eta) \quad N_5^e = \frac{1}{2}(1-\xi^2)(1-\eta)$$
$$N_2^e = -\frac{1}{4}(1+\xi)(1-\eta)(1-\xi+\eta) \quad N_6^e = \frac{1}{2}(1+\xi)(1-\eta^2)$$
$$N_3^e = -\frac{1}{4}(1+\xi)(1+\eta)(1-\xi-\eta) \quad N_7^e = \frac{1}{2}(1-\xi^2)(1+\eta)$$
$$N_4^e = -\frac{1}{4}(1-\xi)(1+\eta)(1+\xi-\eta) \quad N_8^e = \frac{1}{2}(1-\xi)(1-\eta^2)$$

The $\mathbf{B}^e$-matrix is given by

$$\mathbf{B}^e = \nabla \mathbf{N}^e = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix} \mathbf{N}^e = (\mathbf{J}^T)^{-1} \begin{bmatrix} \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \end{bmatrix} \mathbf{N}^e$$

where $\mathbf{J}$ is the Jacobian matrix

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix}$$

Evaluation of the integrals is done by Gauss integration.

3.3.8 flw2i8s

Purpose

Compute heat flux and temperature gradients in an 8 node isoparametric heat flow element.

Syntax

```
[es, et, eci] = flw2i8s(ex, ey, ep, D, ed)
```

Description

`flw2i8s` computes the heat flux vector `es` and the temperature gradient `et` (or corresponding quantities) in an 8 node isoparametric heat flow element.

The input variables `ex`, `ey`, `ep` and the matrix `D` are defined in `flw2i8e`. The vector `ed` contains the nodal temperatures $\mathbf{a}^e$ of the element and is obtained by the function `extract` as

$$\mathbf{ed} = (\mathbf{a}^e)^T = [\, T_1 \;\; T_2 \;\; T_3 \;\; \ldots \;\; T_8 \,]$$

The output variables

$$\mathbf{es} = \bar{\mathbf{q}}^T = \begin{bmatrix} q_x^1 & q_y^1 \\ q_x^2 & q_y^2 \\ \vdots & \vdots \\ q_x^{n^2} & q_y^{n^2} \end{bmatrix}$$

$$\mathbf{et} = (\bar{\nabla} T)^T = \begin{bmatrix} \frac{\partial T}{\partial x}^1 & \frac{\partial T}{\partial y}^1 \\ \frac{\partial T}{\partial x}^2 & \frac{\partial T}{\partial y}^2 \\ \vdots & \vdots \\ \frac{\partial T}{\partial x}^{n^2} & \frac{\partial T}{\partial y}^{n^2} \end{bmatrix}$$

$$\mathbf{eci} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_{n^2} & y_{n^2} \end{bmatrix}$$

contain the heat flux, the temperature gradient, and the coordinates of the integration points. The index n denotes the number of integration points used within the element, see `flw2i8e`.

Theory

The temperature gradient and the heat flux are computed according to

$$\nabla T = \mathbf{B}^e \, \mathbf{a}^e$$

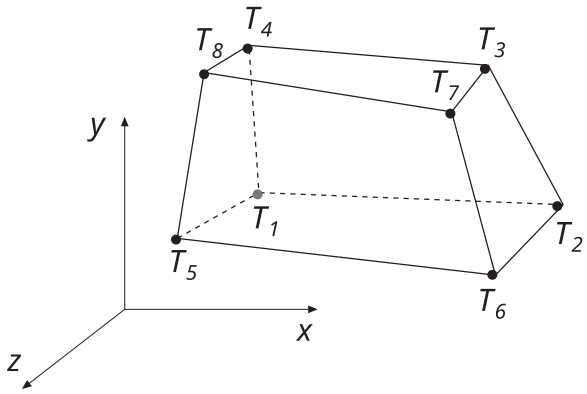
$$\mathbf{q} = -\mathbf{D}\nabla T$$

where the matrices $\mathbf{D}$, $\mathbf{B}^e$, and $\mathbf{a}^e$ are described in `flw2i8e`, and where the integration points are chosen as evaluation points.

3.3.9 flw3i8e

Purpose

Compute element stiffness matrix for an 8 node isoparametric element.



Syntax

```
Ke = flw3i8e(ex, ey, ez, ep, D)
[Ke, fe] = flw3i8e(ex, ey, ez, ep, D, eq)
```

Description

`flw3i8e` provides the element stiffness (conductivity) matrix `Ke` and the element load vector `fe` for an 8 node isoparametric heat flow element.

The element nodal coordinates x_1, y_1, z_1, x_2 etc, are supplied to the function by `ex`, `ey` and `ez`. The number of Gauss points n ($n \times n \times n$ integration points, $n = 1, 2, 3$) are supplied to the function by `ep` and the thermal conductivities (or corresponding quantities) k_{xx}, k_{xy} etc are supplied by `D`.

$$\mathbf{ex} = [\, x_1 \;\; x_2 \;\; x_3 \;\; \dots \;\; x_8 \,]$$
$$\mathbf{ey} = [\, y_1 \;\; y_2 \;\; y_3 \;\; \dots \;\; y_8 \,]$$
$$\mathbf{ez} = [\, z_1 \;\; z_2 \;\; z_3 \;\; \dots \;\; z_8 \,]$$

$$\mathbf{ep} = [\, n \,] \qquad \mathbf{D} = \begin{bmatrix} k_{xx} & k_{xy} & k_{xz} \\ k_{yx} & k_{yy} & k_{yz} \\ k_{zx} & k_{zy} & k_{zz} \end{bmatrix}$$

If the scalar variable `eq` is given in the function, the element load vector **fe** is computed, using

$$\mathbf{eq} = [\, Q \,]$$

where Q is the heat supply per unit volume.

Theory

The element stiffness matrix $\mathbf{K}^e$ and the element load vector $\mathbf{f}_l^e$, stored in `Ke` and `fe`, respectively, are computed according to

$$\mathbf{K}^e = \int_V \mathbf{B}^{eT}\mathbf{D}\mathbf{B}^e \, dV$$
$$\mathbf{f}_l^e = \int_V \mathbf{N}^{eT} Q \, dV$$

with the constitutive matrix **D** defined by `D`.

The evaluation of the integrals for the 3D isoparametric 8 node element is based on a temperature approximation $T(\xi, \eta, \zeta)$, expressed in a local coordinate system in terms of the nodal variables T_1 to T_8 as

$$T(\xi, \eta, \zeta) = \mathbf{N}^e \mathbf{a}^e$$
$$\mathbf{N}^e = [\, N_1^e \;\; N_2^e \;\; N_3^e \;\; \ldots \;\; N_8^e \,] \qquad \mathbf{a}^e = [\, T_1 \;\; T_2 \;\; T_3 \;\; \ldots \;\; T_8 \,]^T$$

The element shape functions are given by

$$N_1^e = \frac{1}{8}(1-\xi)(1-\eta)(1-\zeta) \qquad N_2^e = \frac{1}{8}(1+\xi)(1-\eta)(1-\zeta)$$
$$N_3^e = \frac{1}{8}(1+\xi)(1+\eta)(1-\zeta) \qquad N_4^e = \frac{1}{8}(1-\xi)(1+\eta)(1-\zeta)$$
$$N_5^e = \frac{1}{8}(1-\xi)(1-\eta)(1+\zeta) \qquad N_6^e = \frac{1}{8}(1+\xi)(1-\eta)(1+\zeta)$$
$$N_7^e = \frac{1}{8}(1+\xi)(1+\eta)(1+\zeta) \qquad N_8^e = \frac{1}{8}(1-\xi)(1+\eta)(1+\zeta)$$

The $\mathbf{B}^e$-matrix is given by

$$\mathbf{B}^e = \nabla \mathbf{N}^e = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} \mathbf{N}^e = (\mathbf{J}^T)^{-1} \begin{bmatrix} \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \\ \frac{\partial}{\partial \zeta} \end{bmatrix} \mathbf{N}^e$$

where **J** is the Jacobian matrix

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \zeta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} & \frac{\partial y}{\partial \zeta} \\ \frac{\partial z}{\partial \xi} & \frac{\partial z}{\partial \eta} & \frac{\partial z}{\partial \zeta} \end{bmatrix}$$

Evaluation of the integrals is done by Gauss integration.

3.3.10 flw3i8s

Purpose

Compute heat flux and temperature gradients in an 8 node isoparametric heat flow element.

Syntax

```
[es, et, eci] = flw3i8s(ex, ey, ez, ep, D, ed)
```

Description

`flw3i8s` computes the heat flux vector `es` and the temperature gradient `et` (or corresponding quantities) in an 8 node isoparametric heat flow element.

The input variables **ex**, **ey**, **ez**, **ep** and the matrix **D** are defined in `flw3i8e`. The vector **ed** contains the nodal temperatures $\mathbf{a}^e$ of the element and is obtained by the function `extract` as

$$\mathbf{ed} = (\mathbf{a}^e)^T = [\, T_1 \;\; T_2 \;\; T_3 \;\; \ldots \;\; T_8 \,]$$

The output variables

$$\mathbf{es} = \bar{\mathbf{q}}^T = \begin{bmatrix} q_x^1 & q_y^1 & q_z^1 \\ q_x^2 & q_y^2 & q_z^2 \\ \vdots & \vdots & \vdots \\ q_x^{n^3} & q_y^{n^3} & q_z^{n^3} \end{bmatrix}$$

$$\mathbf{et} = (\bar{\nabla} T)^T = \begin{bmatrix} \frac{\partial T}{\partial x}^1 & \frac{\partial T}{\partial y}^1 & \frac{\partial T}{\partial z}^1 \\ \frac{\partial T}{\partial x}^2 & \frac{\partial T}{\partial y}^2 & \frac{\partial T}{\partial z}^2 \\ \vdots & \vdots & \vdots \\ \frac{\partial T}{\partial x}^{n^3} & \frac{\partial T}{\partial y}^{n^3} & \frac{\partial T}{\partial z}^{n^3} \end{bmatrix} \qquad \mathbf{eci} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_{n^3} & y_{n^3} & z_{n^3} \end{bmatrix}$$

contain the heat flux, the temperature gradient, and the coordinates of the integration points. The index n denotes the number of integration points used within the element, see `flw3i8e`.

Theory

The temperature gradient and the heat flux are computed according to

$$\nabla T = \mathbf{B}^e \, \mathbf{a}^e$$

$$\mathbf{q} = -\mathbf{D} \nabla T$$

where the matrices **D**, $\mathbf{B}^e$, and $\mathbf{a}^e$ are described in `flw3i8e`, and where the integration points are chosen as evaluation points.

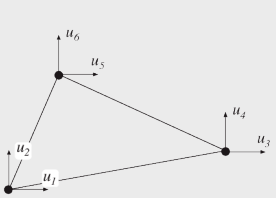
3.4 Solid elements functions

Solid elements are available for two dimensional analysis in plane stress (panels) and plane strain, and for general three dimensional analysis. In the two dimensional case there are a triangular three node element, a quadrilateral four node element, two rectangular four node elements, and quadrilateral isoparametric four and eight node elements. For three dimensional analysis there is an eight node isoparametric element.

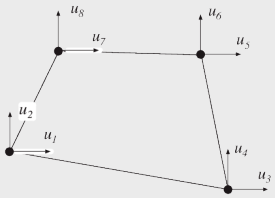
The elements are able to deal with both isotropic and anisotropic materials. The triangular element and the three isoparametric elements can also be used together with a nonlinear material model.

The material properties are specified by supplying the constitutive matrix **D** as an input variable to the element functions. This matrix can be formed by the functions described in Section *Material functions*.

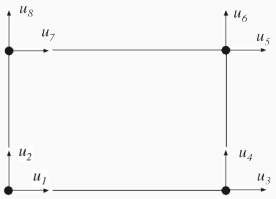
Table 10: **Solid elements**



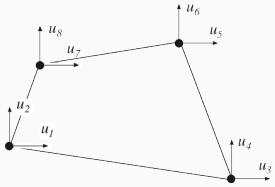
plante



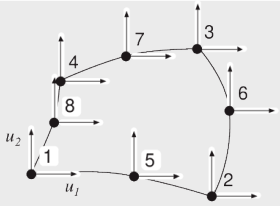
planqe



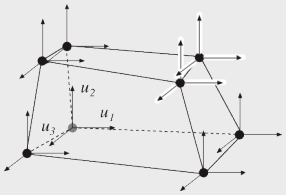
planre plantce



plani4e



plani8e



soli8e

Table 11: 2D solid functions

plante	Compute element matrices for a triangular element
plants	Compute stresses and strains
plantf	Compute internal element forces
planqe	Compute element matrices for a quadrilateral element
planqs	Compute stresses and strains
planre	Compute element matrices for a rectangular Melosh element
planrs	Compute stresses and strains
plantce	Compute element matrices for a rectangular Turner-Clough element
plantcs	Compute stresses and strains
plani4e	Compute element matrices, 4 node isoparametric element
plani4s	Compute stresses and strains
plani4f	Compute internal element forces
plani8e	Compute element matrices, 8 node isoparametric element
plani8s	Compute stresses and strains
plani8f	Compute internal element forces

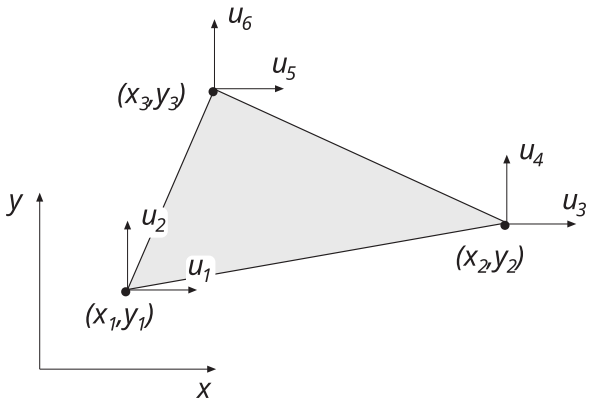
Table 12: 3D solid functions

soli8e	Compute element matrices, 8 node isoparametric element
soli8s	Compute stresses and strains
soli8f	Compute internal element forces

3.4.1 plante

Purpose

Compute element matrices for a triangular element in plane strain or plane stress.



Syntax

```
Ke = plante(ex, ey, ep, D)
[Ke, fe] = plante(ex, ey, ep, D, eq)
```

Description

`plante` provides an element stiffness matrix `Ke` and an element load vector `fe` for a triangular element in plane strain or plane stress.

The element nodal coordinates $x_1, y_1, x_2, \ldots$ are supplied to the function by **ex** and **ey**. The type of analysis `ptype` and the element thickness `t` are supplied by **ep**,

$$\begin{aligned} ptype = 1 \quad & \text{plane stress} \\ ptype = 2 \quad & \text{plane strain} \end{aligned}$$

and the material properties are supplied by the constitutive matrix `D`. Any arbitrary **D**-matrix with dimensions from 3×3 to 6×6 may be given. For an isotropic elastic material the constitutive matrix can be formed by the function `hooke`, see Section *Material functions*.

$$\begin{aligned} \mathbf{ex} &= [\, x_1 \;\; x_2 \;\; x_3 \,] \\ \mathbf{ey} &= [\, y_1 \;\; y_2 \;\; y_3 \,] \\ \mathbf{ep} &= [\, ptype \;\; t \,] \end{aligned}$$

$$\mathbf{D} = \begin{bmatrix} D_{11} & D_{12} & D_{13} \\ D_{21} & D_{22} & D_{23} \\ D_{31} & D_{32} & D_{33} \end{bmatrix} \quad \text{or} \quad \mathbf{D} = \begin{bmatrix} D_{11} & D_{12} & D_{13} & D_{14} & [D_{15}] & [D_{16}] \\ D_{21} & D_{22} & D_{23} & D_{24} & [D_{25}] & [D_{26}] \\ D_{31} & D_{32} & D_{33} & D_{34} & [D_{35}] & [D_{36}] \\ D_{41} & D_{42} & D_{43} & D_{44} & [D_{45}] & [D_{46}] \\ [D_{51}] & [D_{52}] & [D_{53}] & [D_{54}] & [D_{55}] & [D_{56}] \\ [D_{61}] & [D_{62}] & [D_{63}] & [D_{64}] & [D_{65}] & [D_{66}] \end{bmatrix}$$

If uniformly distributed loads are applied to the element, the element load vector `fe` is computed. The input variable

$$\text{eq} = \begin{bmatrix} b_x \\ b_y \end{bmatrix}$$

containing loads per unit volume, b_x and b_y, is then given.

Theory

The element stiffness matrix $\mathbf{K}^e$ and the element load vector $\mathbf{f}_l^e$, stored in `Ke` and `fe`, respectively, are computed according to

$$\mathbf{K}^e = (\mathbf{C}^{-1})^T \int_A \bar{\mathbf{B}}^T \mathbf{D} \bar{\mathbf{B}} \, t \, dA \, \mathbf{C}^{-1}$$

$$\mathbf{f}_l^e = (\mathbf{C}^{-1})^T \int_A \bar{\mathbf{N}}^T \mathbf{b} \, t \, dA$$

with the constitutive matrix **D** defined by `D`, and the body force vector **b** defined by `eq`.

The evaluation of the integrals for the triangular element is based on a linear displacement approximation $\mathbf{u}(x, y)$ and is expressed in terms of the nodal variables $u_1, u_2, \ldots, u_6$ as

$$\mathbf{u}(x, y) = \mathbf{N}^e \mathbf{a}^e = \bar{\mathbf{N}}\, \mathbf{C}^{-1} \mathbf{a}^e$$

where

$$\mathbf{u} = \begin{bmatrix} u_x \\ u_y \end{bmatrix} \quad \bar{\mathbf{N}} = \begin{bmatrix} 1 & x & y & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & x & y \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 1 & x_1 & y_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & x_1 & y_1 \\ 1 & x_2 & y_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & x_2 & y_2 \\ 1 & x_3 & y_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & x_3 & y_3 \end{bmatrix} \quad \mathbf{a}^e = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{bmatrix}$$

The matrix $\bar{\mathbf{B}}$ is obtained as

$$\bar{\mathbf{B}} = \tilde{\nabla} \bar{\mathbf{N}} \quad \text{where} \quad \tilde{\nabla} = \begin{bmatrix} \dfrac{\partial}{\partial x} & 0 \\ 0 & \dfrac{\partial}{\partial y} \\ \dfrac{\partial}{\partial y} & \dfrac{\partial}{\partial x} \end{bmatrix}$$

If a larger **D**-matrix than 3×3 is used for plane stress ($ptype = 1$), the **D**-matrix is reduced to a 3×3 matrix by static condensation using $\sigma_{zz} = \sigma_{xz} = \sigma_{yz} = 0$. These stress components are connected with the rows 3, 5 and 6 in the D-matrix respectively.

If a larger **D**-matrix than 3×3 is used for plane strain ($ptype = 2$), the **D**-matrix is reduced to a 3×3 matrix using $\varepsilon_{zz} = \gamma_{xz} = \gamma_{yz} = 0$. This implies that a 3×3 **D**-matrix is created by the rows and the columns 1, 2 and 4 from the original D-matrix.

Evaluation of the integrals for the triangular element yields

$$\mathbf{K}^e = (\mathbf{C}^{-1})^T \bar{\mathbf{B}}^T \mathbf{D} \bar{\mathbf{B}}\, \mathbf{C}^{-1}\, t\, A$$

$$\mathbf{f}_l^e = \frac{At}{3} \begin{bmatrix} b_x & b_y & b_x & b_y & b_x & b_y \end{bmatrix}^T$$

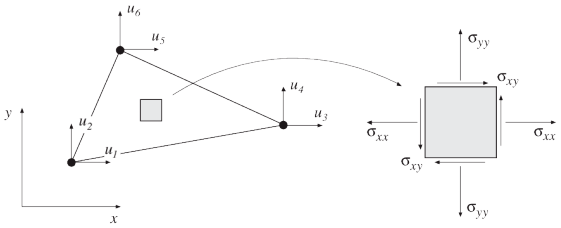
where the element area A is determined as

$$A = \frac{1}{2} \det \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}$$

3.4.2 plants

Purpose

Compute stresses and strains in a triangular element in plane strain or plane stress.



Syntax

```
[es, et] = plants(ex, ey, ep, D, ed)
```

Description

`plants` computes the stresses `es` and the strains `et` in a triangular element in plane strain or plane stress.

The input variables **ex**, **ey**, **ep** and **D** are defined in `plante`. The vector **ed** contains the nodal displacements $\mathbf{a}^e$ of the element and is obtained by the function `extract` as

$$\mathbf{ed} = (\mathbf{a}^e)^T = [\, u_1 \;\; u_2 \;\; \ldots \;\; u_6 \,]$$

The output variables

$$\text{es} = \boldsymbol{\sigma}^T = [\, \sigma_{xx} \;\; \sigma_{yy} \;\; [\sigma_{zz}] \;\; \sigma_{xy} \;\; [\sigma_{xz}] \;\; [\sigma_{yz}] \,]$$

$$\text{et} = \boldsymbol{\varepsilon}^T = [\, \varepsilon_{xx} \;\; \varepsilon_{yy} \;\; [\varepsilon_{zz}] \;\; \gamma_{xy} \;\; [\gamma_{xz}] \;\; [\gamma_{yz}] \,]$$

contain the stress and strain components. The size of `es` and `et` follows the size of `D`. Note that for plane stress $\varepsilon_{zz} \neq 0$, and for plane strain $\sigma_{zz} \neq 0$.

Theory

The strains and stresses are computed according to

$$\boldsymbol{\varepsilon} = \bar{\mathbf{B}}\, \mathbf{C}^{-1}\, \mathbf{a}^e$$

$$\boldsymbol{\sigma} = \mathbf{D}\, \boldsymbol{\varepsilon}$$

where the matrices **D**, $\bar{\mathbf{B}}$, **C** and $\mathbf{a}^e$ are described in `plante`. Note that both the strains and the stresses are constant in the element.

3.4.3 plantf

Purpose

Compute internal element force vector in a triangular element in plane strain or plane stress.

Syntax

```
ef = plantf(ex, ey, ep, es)
```

Description

`plantf` computes the internal element forces `ef` in a triangular element in plane strain or plane stress.

The input variables **ex**, **ey** and **ep** are defined in `plante`, and the input variable **es** is defined in `plants`.

The output variable

$$\mathrm{ef} = \mathbf{f}_i^{eT} = [\, f_{i1} \;\; f_{i2} \;\; \ldots \;\; f_{i6} \,]$$

contains the components of the internal force vector.

Theory

The internal force vector is computed according to

$$\mathbf{f}_i^e = (\mathbf{C}^{-1})^T \int_A \bar{\mathbf{B}}^T \boldsymbol{\sigma} \; t \; dA$$

where the matrices $\bar{\mathbf{B}}$ and $\mathbf{C}$ are defined in `plante` and $\boldsymbol{\sigma}$ is defined in `plants`.

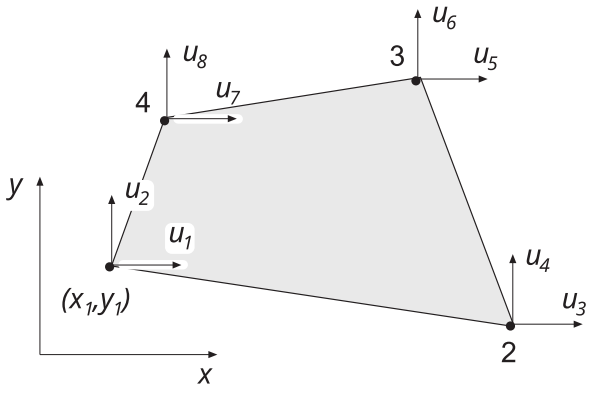
Evaluation of the integral for the triangular element yields

$$\mathbf{f}_i^e = (\mathbf{C}^{-1})^T \bar{\mathbf{B}}^T \boldsymbol{\sigma} \; t \; A$$

3.4.4 planqe

Purpose

Compute element matrices for a quadrilateral element in plane strain or plane stress.



Syntax

```
Ke = planqe(ex, ey, ep, D)
[Ke, fe] = planqe(ex, ey, ep, D, eq)
```

Description

`planqe` provides an element stiffness matrix Ke and an element load vector fe for a quadrilateral element in plane strain or plane stress.

The element nodal coordinates x_1, y_1, x_2, etc. are supplied to the function by **ex** and **ey**. The type of analysis $ptype$ and the element thickness t are supplied by **ep**:

$$\begin{aligned} ptype &= 1 \qquad \text{plane stress} \\ ptype &= 2 \qquad \text{plane strain} \end{aligned}$$

The material properties are supplied by the constitutive matrix D. Any arbitrary **D**-matrix with dimensions from 3×3 to 6×6 may be given. For an isotropic elastic material the constitutive matrix can be formed by the function `hooke`, see Section *Material functions*.

$$\mathbf{ex} = [\, x_1 \;\; x_2 \;\; x_3 \;\; x_4 \,]$$
$$\mathbf{ey} = [\, y_1 \;\; y_2 \;\; y_3 \;\; y_4 \,]$$
$$\mathbf{ep} = [\, ptype \;\; t \,]$$

$$\mathbf{D} = \begin{bmatrix} D_{11} & D_{12} & D_{13} \\ D_{21} & D_{22} & D_{23} \\ D_{31} & D_{32} & D_{33} \end{bmatrix} \quad \text{or} \quad \mathbf{D} = \begin{bmatrix} D_{11} & D_{12} & D_{13} & D_{14} & D_{15} & D_{16} \\ D_{21} & D_{22} & D_{23} & D_{24} & D_{25} & D_{26} \\ D_{31} & D_{32} & D_{33} & D_{34} & D_{35} & D_{36} \\ D_{41} & D_{42} & D_{43} & D_{44} & D_{45} & D_{46} \\ D_{51} & D_{52} & D_{53} & D_{54} & D_{55} & D_{56} \\ D_{61} & D_{62} & D_{63} & D_{64} & D_{65} & D_{66} \end{bmatrix}$$

If uniformly distributed loads are applied on the element, the element load vector fe is computed. The input variable

$$eq = \begin{bmatrix} b_x \\ b_y \end{bmatrix}$$

contains loads per unit volume, b_x and b_y.

Theory

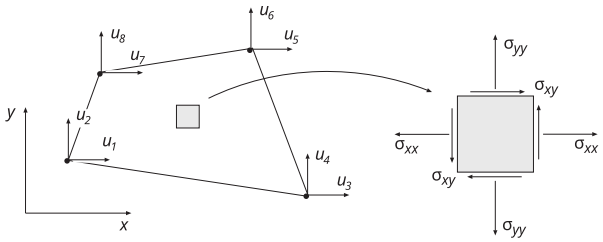
In computing the element matrices, two more degrees of freedom are introduced. The location of these two degrees of freedom is defined by the mean value of the coordinates at the corner points. Four sets of element matrices are calculated using `plante`. These matrices are then assembled and the two extra degrees of freedom are eliminated by static condensation.

3.4.5 planqs

Purpose

Compute stresses and strains in a quadrilateral element in plane strain or plane stress.

Syntax




```
[es,et]=planqs(ex,ey,ep,D,ed)
[es,et]=planqs(ex,ey,ep,D,ed,eq)
```

Description

`planqs` computes the stresses `es` and the strains `et` in a quadrilateral element in plane strain or plane stress.

The input variables **ex**, **ey**, **ep**, **D** and **eq** are defined in `planqe`. The vector **ed** contains the nodal displacements $\mathbf{a}^e$ of the element and is obtained by the function `extract` as

$$\mathbf{ed} = (\mathbf{a}^e)^T = [\, u_1 \;\; u_2 \;\; \ldots \;\; u_8 \,]$$

If body forces are applied to the element the variable **eq** must be included.

The output variables

$$\text{es} = \boldsymbol{\sigma}^T = [\, \sigma_{xx} \;\; \sigma_{yy} \;\; [\sigma_{zz}] \;\; \sigma_{xy} \;\; [\sigma_{xz}] \;\; [\sigma_{yz}] \,]$$

$$\text{et} = \boldsymbol{\varepsilon}^T = [\, \varepsilon_{xx} \;\; \varepsilon_{yy} \;\; [\varepsilon_{zz}] \;\; \gamma_{xy} \;\; [\gamma_{xz}] \;\; [\gamma_{yz}] \,]$$

contain the stress and strain components. The size of `es` and `et` follows the size of `D`. Note that for plane stress $\varepsilon_{zz} \neq 0$, and for plane strain $\sigma_{zz} \neq 0$.

Theory

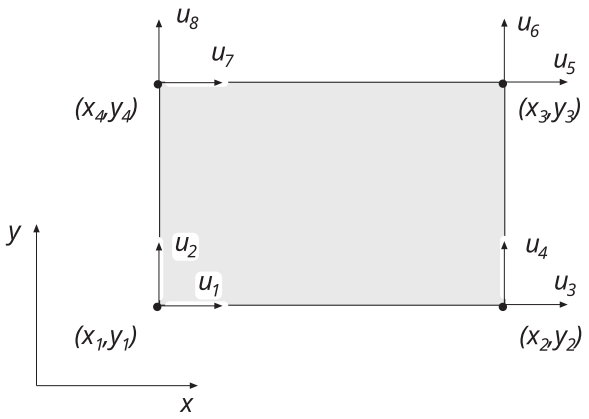
By assembling triangular elements as described in `planqe` a system of equations containing 10 degrees of freedom is obtained. From this system of equations the two unknown displacements at the center of the element are computed. Then according to the description in `plants` the strain and stress components in each of the four triangular elements are produced. Finally the quadrilateral element strains and stresses are computed as area weighted mean values from the values of the four triangular elements. If uniformly distributed loads are applied on the element, the element load vector `eq` is needed for the calculations.

3.4.6 planre

Purpose

Compute element matrices for a rectangular (Melosh) element in plane strain or plane stress.

Syntax




```
Ke = planre(ex, ey, ep, D)
[Ke, fe] = planre(ex, ey, ep, D, eq)
```

Description

planre provides an element stiffness matrix Ke and an element load vector fe for a rectangular (Melosh) element in plane strain or plane stress. This element can only be used if the element edges are parallel to the coordinate axis.

The element nodal coordinates (x_1, y_1) and (x_3, y_3) are supplied to the function by ex and ey. The type of analysis ptype and the element thickness t are supplied by ep,

$$\begin{array}{ll} ptype = 1 & \text{plane stress} \\ ptype = 2 & \text{plane strain} \end{array}$$

and the material properties are supplied by the constitutive matrix D. Any arbitrary D-matrix with dimensions from 3×3 to 6×6 may be given. For an isotropic elastic material the constitutive matrix can be formed by the function hooke, see Section Material.

$$\begin{array}{l} \mathbf{ex} = [\, x_1 \;\; x_3 \,] \\ \mathbf{ey} = [\, y_1 \;\; y_3 \,] \end{array} \qquad \mathbf{ep} = [\, ptype \;\; t \,]$$

$$\mathbf{D} = \begin{bmatrix} D_{11} & D_{12} & D_{13} \\ D_{21} & D_{22} & D_{23} \\ D_{31} & D_{32} & D_{33} \end{bmatrix} \quad \text{or} \quad \mathbf{D} = \begin{bmatrix} D_{11} & D_{12} & D_{13} & D_{14} & [D_{15}] & [D_{16}] \\ D_{21} & D_{22} & D_{23} & D_{24} & [D_{25}] & [D_{26}] \\ D_{31} & D_{32} & D_{33} & D_{34} & [D_{35}] & [D_{36}] \\ D_{41} & D_{42} & D_{43} & D_{44} & [D_{45}] & [D_{46}] \\ [D_{51}] & [D_{52}] & [D_{53}] & [D_{54}] & [D_{55}] & [D_{56}] \\ [D_{61}] & [D_{62}] & [D_{63}] & [D_{64}] & [D_{65}] & [D_{66}] \end{bmatrix}$$

If uniformly distributed loads are applied on the element, the element load vector fe is computed. The input variable

$$eq = \begin{bmatrix} b_x \\ b_y \end{bmatrix}$$

containing loads per unit volume, b_x and b_y, is then given.

Theory

The element stiffness matrix $\mathbf{K}^e$ and the element load vector $\mathbf{f}_l^e$, stored in Ke and fe, respectively, are computed according to

$$\mathbf{K}^e = \int_A \mathbf{B}^{eT}\mathbf{D}\mathbf{B}^e t\, dA$$

$$\mathbf{f}_l^e = \int_A \mathbf{N}^{eT}\mathbf{b} t\, dA$$

with the constitutive matrix $\mathbf{D}$ defined by D, and the body force vector $\mathbf{b}$ defined by eq.

The evaluation of the integrals for the rectangular element is based on a bilinear displacement approximation $\mathbf{u}(x, y)$ and is expressed in terms of the nodal variables $u_1, u_2, \ldots, u_8$ as

$$\mathbf{u}(x, y) = \mathbf{N}^e\mathbf{a}^e$$

where

$$\mathbf{u} = \begin{bmatrix} u_x \\ u_y \end{bmatrix} \quad \mathbf{N}^e = \begin{bmatrix} N_1^e & 0 & N_2^e & 0 & N_3^e & 0 & N_4^e & 0 \\ 0 & N_1^e & 0 & N_2^e & 0 & N_3^e & 0 & N_4^e \end{bmatrix} \quad \mathbf{a}^e = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_8 \end{bmatrix}$$

With a local coordinate system located at the center of the element, the element shape functions N_1^e–N_4^e are obtained as

$$\begin{aligned} N_1^e &= \frac{1}{4ab}(x - x_2)(y - y_4) \\ N_2^e &= -\frac{1}{4ab}(x - x_1)(y - y_3) \\ N_3^e &= \frac{1}{4ab}(x - x_4)(y - y_2) \\ N_4^e &= -\frac{1}{4ab}(x - x_3)(y - y_1) \end{aligned}$$

where

$$a = \frac{1}{2}(x_3 - x_1) \quad \text{and} \quad b = \frac{1}{2}(y_3 - y_1)$$

The matrix $\mathbf{B}^e$ is obtained as

$$\mathbf{B}^e = \tilde{\nabla}\mathbf{N}^e \qquad \text{where} \quad \tilde{\nabla} = \begin{bmatrix} \dfrac{\partial}{\partial x} & 0 \\ 0 & \dfrac{\partial}{\partial y} \\ \dfrac{\partial}{\partial y} & \dfrac{\partial}{\partial x} \end{bmatrix}$$

If a larger D-matrix than 3
$times3$ is used for plane stress ($ptype = 1$), the D-matrix is reduced to a 3

$times3$ matrix by static condensation using $\sigma_{zz} = \sigma_{xz} = \sigma_{yz} = 0$. These stress components are connected with the rows 3, 5 and 6 in the D-matrix respectively.

If a larger D-matrix than 3
$times3$ is used for plane strain ($ptype = 2$), the D-matrix is reduced to a 3
$times3$ matrix using $\varepsilon_{zz} = \gamma_{xz} = \gamma_{yz} = 0$. This implies that a 3
$times3$ D-matrix is created by the rows and the columns 1, 2 and 4 from the original D-matrix.

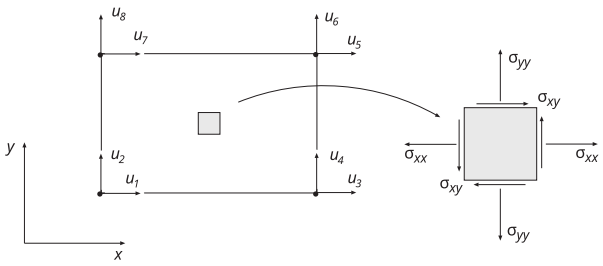
Evaluation of the integrals for the rectangular element can be done either analytically or numerically by use of a 2
$times2$ point Gauss integration. The element load vector $\mathbf{f}_l^e$ yields

$$\mathbf{f}_l^e = abt \begin{bmatrix} b_x \\ b_y \\ b_x \\ b_y \\ b_x \\ b_y \\ b_x \\ b_y \end{bmatrix}$$

3.4.7 planrs

Purpose

Compute stresses and strains in a rectangular (Melosh) element in plane strain or plane stress.



Syntax

```
[es,et]=planrs(ex,ey,ep,D,ed)
```

Description

`planrs` computes the stresses `es` and the strains `et` in a rectangular (Melosh) element in plane strain or plane stress. The stress and strain components are computed at the center of the element.

The input variables **ex**, **ey**, **ep** and **D** are defined in `planre`. The vector **ed** contains the nodal displacements $\mathbf{a}^e$ of the element and is obtained by the function

`extract` as

$$\mathbf{ed} = (\mathbf{a}^e)^T = [\, u_1 \;\; u_2 \;\; \ldots \;\; u_8 \,]$$

The output variables

$$\text{es} = \boldsymbol{\sigma}^T = [\, \sigma_{xx} \;\; \sigma_{yy} \;\; [\sigma_{zz}] \;\; \sigma_{xy} \;\; [\sigma_{xz}] \;\; [\sigma_{yz}] \,]$$

$$\text{et} = \boldsymbol{\varepsilon}^T = [\, \varepsilon_{xx} \;\; \varepsilon_{yy} \;\; [\varepsilon_{zz}] \;\; \gamma_{xy} \;\; [\gamma_{xz}] \;\; [\gamma_{yz}] \,]$$

contain the stress and strain components. The size of `es` and `et` follows the size of `D`. Note that for plane stress $\varepsilon_{zz} \neq 0$, and for plane strain $\sigma_{zz} \neq 0$.

Theory

The strains and stresses are computed according to

$$\boldsymbol{\varepsilon} = \mathbf{B}^e \, \mathbf{a}^e$$

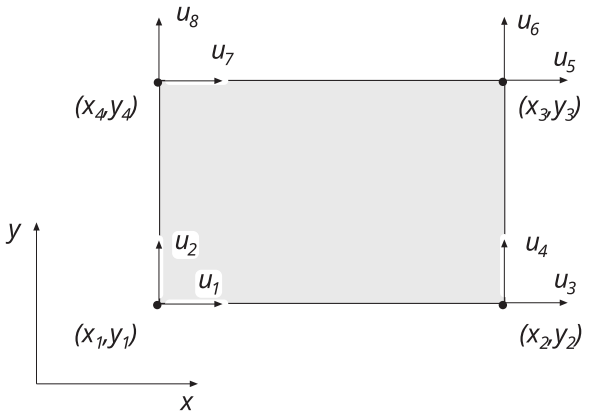
$$\boldsymbol{\sigma} = \mathbf{D} \, \boldsymbol{\varepsilon}$$

where the matrices $\mathbf{D}$, $\mathbf{B}^e$, and $\mathbf{a}^e$ are described in `planre`, and where the evaluation point (x, y) is chosen to be at the center of the element.

3.4.8 plantce

Purpose

Compute element matrices for a rectangular (Turner-Clough) element in plane strain or plane stress.



Syntax

```
Ke = plantce(ex, ey, ep)
[Ke, fe] = plantce(ex, ey, ep, eq)
```

Description

`plantce` provides an element stiffness matrix `Ke` and an element load vector `fe` for a rectangular (Turner-Clough) element in plane strain or plane stress. This element can only be used if the material is isotropic and if the element edges are parallel to the coordinate axis.

The element nodal coordinates (x_1, y_1) and (x_3, y_3) are supplied to the function by **ex** and **ey**. The state of stress `ptype`, the element thickness t and the material properties E and ν are supplied by **ep**. For plane stress $ptype = 1$ and for plane strain $ptype = 2$.

$$\begin{matrix}\mathbf{ex} = [\, x_1 \;\; x_3 \,] \\ \mathbf{ey} = [\, y_1 \;\; y_3 \,]\end{matrix} \qquad \mathbf{ep} = [\, ptype \;\; t \;\; E \;\; \nu \,]$$

If uniformly distributed loads are applied to the element, the element load vector `fe` is computed. The input variable

$$eq = \begin{bmatrix} b_x \\ b_y \end{bmatrix}$$

containing loads per unit volume, b_x and b_y, is then given.

Theory

The element stiffness matrix $\mathbf{K}^e$ and the element load vector $\mathbf{f}_l^e$, stored in `Ke` and `fe`, respectively, are computed according to

$$\mathbf{K}^e = \int_A \mathbf{B}^{eT}\mathbf{D}\mathbf{B}^e t \, dA$$

$$\mathbf{f}_l^e = \int_A \mathbf{N}^{eT}\mathbf{b} t \, dA$$

where the constitutive matrix **D** is described in `hooke`, see Section *Material functions*, and the body force vector **b** is defined by `eq`.

The evaluation of the integrals for the Turner-Clough element is based on a displacement field $\mathbf{u}(x, y)$ built up of a bilinear displacement approximation superposed by bubble functions in order to create a linear stress field over the element. The displacement field is expressed in terms of the nodal variables $u_1, u_2, \ldots, u_8$ as

$$\mathbf{u}(x, y) = \mathbf{N}^e \mathbf{a}^e$$

where

$$\mathbf{u} = \begin{bmatrix} u_x \\ u_y \end{bmatrix} \quad \mathbf{N}^e = \begin{bmatrix} N_1^e & N_5^e & N_2^e & -N_5^e & N_3^e & N_5^e & N_4^e & -N_5^e \\ N_6^e & N_1^e & -N_6^e & N_2^e & N_6^e & N_3^e & -N_6^e & N_4^e \end{bmatrix} \quad \mathbf{a}^e = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_8 \end{bmatrix}$$

With a local coordinate system located at the center of the element, the element shape functions N_1^e–N_6^e are obtained as

$$
\begin{aligned}
N_1^e &= \frac{1}{4ab}(a-x)(b-y) \\
N_2^e &= \frac{1}{4ab}(a+x)(b-y) \\
N_3^e &= \frac{1}{4ab}(a+x)(b+y) \\
N_4^e &= \frac{1}{4ab}(a-x)(b+y) \\
N_5^e &= \frac{1}{8ab}\left[(b^2-y^2)+\nu(a^2-x^2)\right] \\
N_6^e &= \frac{1}{8ab}\left[(a^2-x^2)+\nu(b^2-y^2)\right]
\end{aligned}
$$

where

$$
a = \frac{1}{2}(x_3 - x_1) \qquad b = \frac{1}{2}(y_3 - y_1)
$$

The matrix $\mathbf{B}^e$ is obtained as

$$
\mathbf{B}^e = \tilde{\nabla}\mathbf{N}^e \qquad \text{where} \quad \tilde{\nabla} = \begin{bmatrix} \dfrac{\partial}{\partial x} & 0 \\ 0 & \dfrac{\partial}{\partial y} \\ \dfrac{\partial}{\partial y} & \dfrac{\partial}{\partial x} \end{bmatrix}
$$

Evaluation of the integrals for the Turner-Clough element can be done either analytically or numerically by use of a 2×2 point Gauss integration. The element load vector $\mathbf{f}_l^e$ yields

$$
\mathbf{f}_l^e = abt \begin{bmatrix} b_x \\ b_y \\ b_x \\ b_y \\ b_x \\ b_y \\ b_x \\ b_y \end{bmatrix}
$$

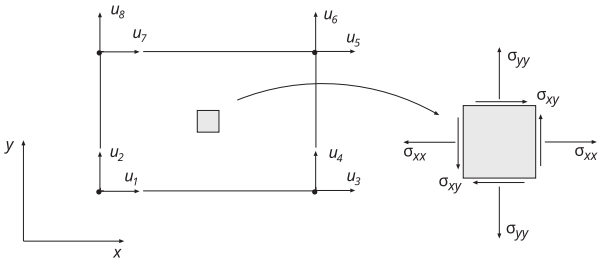
3.4.9 plantcs

Purpose

Compute stresses and strains in a Turner-Clough element in plane strain or plane stress.

Syntax

```
[es, et] = plantcs(ex, ey, ep, ed)
```



Description

`plantcs` computes the stresses `es` and the strains `et` in a rectangular Turner-Clough element in plane strain or plane stress. The stress and strain components are computed at the center of the element.

The input variables **ex**, **ey**, and **ep** are defined in `plantce`. The vector **ed** contains the nodal displacements $\mathbf{a}^e$ of the element and is obtained by the function `extract` as

$$\mathbf{ed} = (\mathbf{a}^e)^T = [\, u_1 \;\; u_2 \;\; \ldots \;\; u_8 \,]$$

The output variables

$$\text{es} = \boldsymbol{\sigma}^T = [\, \sigma_{xx} \;\; \sigma_{yy} \;\; [\sigma_{zz}] \;\; \sigma_{xy} \;\; [\sigma_{xz}] \;\; [\sigma_{yz}] \,]$$

$$\text{et} = \boldsymbol{\varepsilon}^T = [\, \varepsilon_{xx} \;\; \varepsilon_{yy} \;\; [\varepsilon_{zz}] \;\; \gamma_{xy} \;\; [\gamma_{xz}] \;\; [\gamma_{yz}] \,]$$

contain the stress and strain components. The size of `es` and `et` follows the size of `D`. Note that for plane stress $\varepsilon_{zz} \neq 0$, and for plane strain $\sigma_{zz} \neq 0$.

Theory

The strains and stresses are computed according to

$$\boldsymbol{\varepsilon} = \mathbf{B}^e \, \mathbf{a}^e$$

$$\boldsymbol{\sigma} = \mathbf{D} \, \boldsymbol{\varepsilon}$$

where the matrices **D**, $\mathbf{B}^e$, and $\mathbf{a}^e$ are described in `plantce`, and where the evaluation point (x, y) is chosen to be at the center of the element.

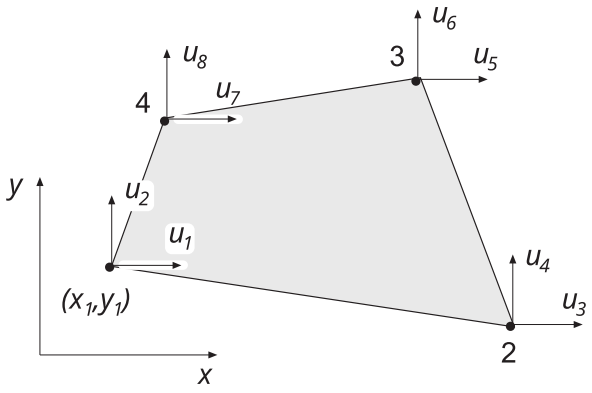
3.4.10 plani4e

Purpose

Compute element matrices for a 4 node isoparametric element in plane strain or plane stress.

Syntax

```
Ke = plani4e(ex, ey, ep, D)
[Ke, fe] = plani4e(ex, ey, ep, D, eq)
```



Description

`plani4e` provides an element stiffness matrix `Ke` and an element load vector `fe` for a 4 node isoparametric element in plane strain or plane stress.

The element nodal coordinates $x_1, y_1, x_2, \ldots$ are supplied to the function by **ex** and **ey**. The type of analysis $ptype$, the element thickness t, and the number of Gauss points n are supplied by **ep**:

$ptype = 1$	plane stress	$(n \times n)$ integration points
$ptype = 2$	plane strain	$n = 1, 2, 3$

The material properties are supplied by the constitutive matrix D. Any arbitrary **D**-matrix with dimensions from 3×3 to 6×6 may be given. For an isotropic elastic material the constitutive matrix can be formed by the function `hooke`, see Section *Material functions*.

$$\mathbf{ex} = [\, x_1 \;\; x_2 \;\; x_3 \;\; x_4 \,]$$
$$\mathbf{ey} = [\, y_1 \;\; y_2 \;\; y_3 \;\; y_4 \,]$$
$$\mathbf{ep} = [\, ptype \;\; t \;\; n \,]$$

$$\mathbf{D} = \begin{bmatrix} D_{11} & D_{12} & D_{13} \\ D_{21} & D_{22} & D_{23} \\ D_{31} & D_{32} & D_{33} \end{bmatrix} \quad \text{or} \quad \mathbf{D} = \begin{bmatrix} D_{11} & D_{12} & D_{13} & D_{14} & [D_{15}] & [D_{16}] \\ D_{21} & D_{22} & D_{23} & D_{24} & [D_{25}] & [D_{26}] \\ D_{31} & D_{32} & D_{33} & D_{34} & [D_{35}] & [D_{36}] \\ D_{41} & D_{42} & D_{43} & D_{44} & [D_{45}] & [D_{46}] \\ [D_{51}] & [D_{52}] & [D_{53}] & [D_{54}] & [D_{55}] & [D_{56}] \\ [D_{61}] & [D_{62}] & [D_{63}] & [D_{64}] & [D_{65}] & [D_{66}] \end{bmatrix}$$

If different $\mathbf{D}_i$ matrices are used in the Gauss points, these $\mathbf{D}_i$ matrices are stored in a global vector `D`. For numbering of the Gauss points, see `eci` in `plani4s`.

$$\mathbf{D} = \begin{bmatrix} \mathbf{D}_1 \\ \mathbf{D}_2 \\ \vdots \\ \mathbf{D}_{n^2} \end{bmatrix}$$

If uniformly distributed loads are applied to the element, the element load vector `fe` is computed. The input variable

$$\text{eq} = \begin{bmatrix} b_x \\ b_y \end{bmatrix}$$

containing loads per unit volume, b_x and b_y, is then given.

Theory

The element stiffness matrix $\mathbf{K}^e$ and the element load vector $\mathbf{f}_l^e$, stored in `Ke` and `fe`, respectively, are computed according to

$$\mathbf{K}^e = \int_A \mathbf{B}^{eT} \mathbf{D} \mathbf{B}^e t \, dA$$
$$\mathbf{f}_l^e = \int_A \mathbf{N}^{eT} \mathbf{b} t \, dA$$

with the constitutive matrix $\mathbf{D}$ defined by `D`, and the body force vector $\mathbf{b}$ defined by `eq`.

The evaluation of the integrals for the isoparametric 4 node element is based on a displacement approximation $\mathbf{u}(\xi, \eta)$, expressed in a local coordinate system in terms of the nodal variables $u_1, u_2, \ldots, u_8$ as

$$\mathbf{u}(\xi, \eta) = \mathbf{N}^e \mathbf{a}^e$$

where

$$\mathbf{u} = \begin{bmatrix} u_x \\ u_y \end{bmatrix} \qquad \mathbf{N}^e = \begin{bmatrix} N_1^e & 0 & N_2^e & 0 & N_3^e & 0 & N_4^e & 0 \\ 0 & N_1^e & 0 & N_2^e & 0 & N_3^e & 0 & N_4^e \end{bmatrix} \qquad \mathbf{a}^e = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_8 \end{bmatrix}$$

The element shape functions are given by

$$N_1^e = \frac{1}{4}(1-\xi)(1-\eta) \qquad N_2^e = \frac{1}{4}(1+\xi)(1-\eta)$$
$$N_3^e = \frac{1}{4}(1+\xi)(1+\eta) \qquad N_4^e = \frac{1}{4}(1-\xi)(1+\eta)$$

The matrix $\mathbf{B}^e$ is obtained as

$$\mathbf{B}^e = \tilde{\nabla} \mathbf{N}^e \qquad \text{where} \quad \tilde{\nabla} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix}$$

and

$$\begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix} = (\mathbf{J}^T)^{-1} \begin{bmatrix} \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \end{bmatrix} \qquad \mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix}$$

If a larger $\mathbf{D}$-matrix than 3×3 is used for plane stress ($ptype = 1$), the $\mathbf{D}$-matrix is reduced to a 3×3 matrix by static condensation using $\sigma_{zz} = \sigma_{xz} = \sigma_{yz} = 0$.

These stress components are connected with the rows 3, 5 and 6 in the D-matrix respectively.

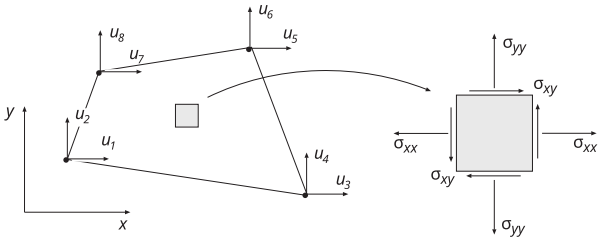
If a larger **D**-matrix than 3×3 is used for plane strain ($ptype = 2$), the **D**-matrix is reduced to a 3×3 matrix using $\varepsilon_{zz} = \gamma_{xz} = \gamma_{yz} = 0$. This implies that a 3×3 **D**-matrix is created by the rows and the columns 1, 2 and 4 from the original D-matrix.

Evaluation of the integrals is done by Gauss integration.

3.4.11 plani4s

Purpose

Compute stresses and strains in a 4 node isoparametric element in plane strain or plane stress.



Syntax

```
[es,et,eci]=plani4s(ex,ey,ep,D,ed)
```

Description

`plani4s` computes stresses `es` and the strains `et` in a 4 node isoparametric element in plane strain or plane stress.

The input variables **ex**, **ey**, **ep** and the matrix **D** are defined in `plani4e`. The vector **ed** contains the nodal displacements $\mathbf{a}^e$ of the element and is obtained by the function `extract` as

$$\mathbf{ed} = (\mathbf{a}^e)^T = [\, u_1 \;\; u_2 \;\; \ldots \;\; u_8 \,]$$

The output variables

$$\text{es} = \boldsymbol{\sigma}^T = \begin{bmatrix} \sigma_{xx}^1 & \sigma_{yy}^1 & [\sigma_{zz}^1] & \sigma_{xy}^1 & [\sigma_{xz}^1] & [\sigma_{yz}^1] \\ \sigma_{xx}^2 & \sigma_{yy}^2 & [\sigma_{zz}^2] & \sigma_{xy}^2 & [\sigma_{xz}^2] & [\sigma_{yz}^2] \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \sigma_{xx}^{n^2} & \sigma_{yy}^{n^2} & [\sigma_{zz}^{n^2}] & \sigma_{xy}^{n^2} & [\sigma_{xz}^{n^2}] & [\sigma_{yz}^{n^2}] \end{bmatrix}$$

$$\text{et} = \boldsymbol{\varepsilon}^T = \begin{bmatrix} \varepsilon_{xx}^1 & \varepsilon_{yy}^1 & [\varepsilon_{zz}^1] & \gamma_{xy}^1 & [\gamma_{xz}^1] & [\gamma_{yz}^1] \\ \varepsilon_{xx}^2 & \varepsilon_{yy}^2 & [\varepsilon_{zz}^2] & \gamma_{xy}^2 & [\gamma_{xz}^2] & [\gamma_{yz}^2] \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \varepsilon_{xx}^{n^2} & \varepsilon_{yy}^{n^2} & [\varepsilon_{zz}^{n^2}] & \gamma_{xy}^{n^2} & [\gamma_{xz}^{n^2}] & [\gamma_{yz}^{n^2}] \end{bmatrix}$$

$$\mathbf{eci} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_{n^2} & y_{n^2} \end{bmatrix}$$

contain the stress and strain components, and the coordinates of the integration points. The index n denotes the number of integration points used within the element, cf. `plani4e`. The number of columns in `es` and `et` follows the size of `D`. Note that for plane stress $\varepsilon_{zz} \neq 0$, and for plane strain $\sigma_{zz} \neq 0$.

Theory

The strains and stresses are computed according to

$$\boldsymbol{\varepsilon} = \mathbf{B}^e \, \mathbf{a}^e$$

$$\boldsymbol{\sigma} = \mathbf{D} \, \boldsymbol{\varepsilon}$$

where the matrices $\mathbf{D}$, $\mathbf{B}^e$, and $\mathbf{a}^e$ are described in `plani4e`, and where the integration points are chosen as evaluation points.

3.4.12 plani4f

Purpose

Compute internal element force vector in a 4 node isoparametric element in plane strain or plane stress.

Syntax

```
ef = plani4f(ex, ey, ep, es)
```

Description

`plani4f` computes the internal element forces ef in a 4 node isoparametric element in plane strain or plane stress.

The input variables **ex**, **ey** and **ep** are defined in `plani4e`, and the input variable **es** is defined in `plani4s`.

The output variable

$$\mathrm{ef} = \mathbf{f}_i^{eT} = [\, f_{i1} \;\; f_{i2} \;\; \ldots \;\; f_{i8} \,]$$

contains the components of the internal force vector.

Theory

The internal force vector is computed according to

$$\mathbf{f}_i^e = \int_A \mathbf{B}^{eT} \boldsymbol{\sigma} \; t \; dA$$

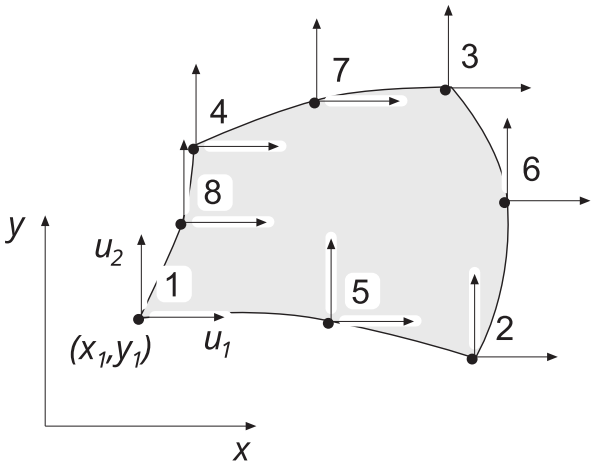
where the matrices $\mathbf{B}^e$ and $\boldsymbol{\sigma}$ are defined in `plani4e` and `plani4s`, respectively.

Evaluation of the integral is done by Gauss integration.

3.4.13 plani8e

Purpose

Compute element matrices for an 8 node isoparametric element in plane strain or plane stress.



Syntax

```
Ke = plani8e(ex, ey, ep, D)
[Ke, fe] = plani8e(ex, ey, ep, D, eq)
```

Description

`plani8e` provides an element stiffness matrix `Ke` and an element load vector `fe` for an 8 node isoparametric element in plane strain or plane stress.

The element nodal coordinates $x_1, y_1, x_2, \ldots$ are supplied to the function by **ex** and **ey**. The type of analysis $ptype$, the element thickness t, and the number of Gauss points n are supplied by **ep**:

$ptype = 1$	plane stress	$(n \times n)$ integration points
$ptype = 2$	plane strain	$n = 1, 2, 3$

The material properties are supplied by the constitutive matrix **D**. Any arbitrary **D**-matrix with dimensions from 3×3 to 6×6 may be given. For an isotropic elastic material the constitutive matrix can be formed by the function `hooke`.

$$\mathbf{ex} = [x_1, x_2, \ldots, x_8]$$
$$\mathbf{ey} = [y_1, y_2, \ldots, y_8]$$
$$\mathbf{ep} = [ptype, t, n]$$

$$\mathbf{D} = \begin{bmatrix} D_{11} & D_{12} & D_{13} \\ D_{21} & D_{22} & D_{23} \\ D_{31} & D_{32} & D_{33} \end{bmatrix} \quad \text{or} \quad \mathbf{D} = \begin{bmatrix} D_{11} & D_{12} & D_{13} & D_{14} & [D_{15}] & [D_{16}] \\ D_{21} & D_{22} & D_{23} & D_{24} & [D_{25}] & [D_{26}] \\ D_{31} & D_{32} & D_{33} & D_{34} & [D_{35}] & [D_{36}] \\ D_{41} & D_{42} & D_{43} & D_{44} & [D_{45}] & [D_{46}] \\ [D_{51}] & [D_{52}] & [D_{53}] & [D_{54}] & [D_{55}] & [D_{56}] \\ [D_{61}] & [D_{62}] & [D_{63}] & [D_{64}] & [D_{65}] & [D_{66}] \end{bmatrix}$$

If different $\mathbf{D}_i$ matrices are used in the Gauss points these $\mathbf{D}_i$ matrices are stored in a global vector `D`. For numbering of the Gauss points, see `eci` in `plani8s`.

$$D = \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_{n^2} \end{bmatrix}$$

If uniformly distributed loads are applied to the element, the element load vector `fe` is computed. The input variable

$$\mathbf{eq} = \begin{bmatrix} b_x \\ b_y \end{bmatrix}$$

containing loads per unit volume, b_x and b_y, is then given.

Theory

The element stiffness matrix $\mathbf{K}^e$ and the element load vector $\mathbf{f}_l^e$, stored in `Ke` and `fe`, respectively, are computed according to

$$\mathbf{K}^e = \int_A \mathbf{B}^{eT} \mathbf{D} \mathbf{B}^e t \, dA$$
$$\mathbf{f}_l^e = \int_A \mathbf{N}^{eT} \mathbf{b} t \, dA$$

with the constitutive matrix $\mathbf{D}$ defined by `D`, and the body force vector $\mathbf{b}$ defined by `eq`.

The evaluation of the integrals for the isoparametric 8 node element is based on a displacement approximation $\mathbf{u}(\xi, \eta)$, expressed in a local coordinate system in terms of the nodal variables $u_1, u_2, \ldots, u_{16}$ as

$$\mathbf{u}(\xi, \eta) = \mathbf{N}^e \mathbf{a}^e$$

where

$$\mathbf{u} = \begin{bmatrix} u_x \\ u_y \end{bmatrix}, \quad \mathbf{N}^e = \begin{bmatrix} N_1^e & 0 & N_2^e & 0 & \cdots & N_8^e & 0 \\ 0 & N_1^e & 0 & N_2^e & \cdots & 0 & N_8^e \end{bmatrix}, \quad \mathbf{a}^e = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{16} \end{bmatrix}$$

The element shape functions are given by

$$N_1^e = -\frac{1}{4}(1-\xi)(1-\eta)(1+\xi+\eta) \qquad N_5^e = \frac{1}{2}(1-\xi^2)(1-\eta)$$
$$N_2^e = -\frac{1}{4}(1+\xi)(1-\eta)(1-\xi+\eta) \qquad N_6^e = \frac{1}{2}(1+\xi)(1-\eta^2)$$
$$N_3^e = -\frac{1}{4}(1+\xi)(1+\eta)(1-\xi-\eta) \qquad N_7^e = \frac{1}{2}(1-\xi^2)(1+\eta)$$
$$N_4^e = -\frac{1}{4}(1-\xi)(1+\eta)(1+\xi-\eta) \qquad N_8^e = \frac{1}{2}(1-\xi)(1-\eta^2)$$

The matrix $\mathbf{B}^e$ is obtained as

$$\mathbf{B}^e = \tilde{\nabla}\mathbf{N}^e \qquad \text{where} \quad \tilde{\nabla} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix}$$

and where

$$\begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix} = (\mathbf{J}^T)^{-1} \begin{bmatrix} \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \end{bmatrix} \qquad \mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix}$$

If a larger **D**-matrix than 3×3 is used for plane stress ($ptype = 1$), the **D**-matrix is reduced to a 3×3 matrix by static condensation, setting

$$\sigma_{zz} = 0$$
$$\sigma_{xz} = 0$$
$$\sigma_{yz} = 0$$

These stress components correspond to rows and columns 3, 5, and 6 in the D-matrix, respectively.

If a larger **D**-matrix than 3×3 is used for plane strain ($ptype = 2$), the **D**-matrix is reduced to a 3×3 matrix by setting

$$\varepsilon_{zz} = 0$$
$$\gamma_{xz} = 0$$
$$\gamma_{yz} = 0$$

This means that the resulting 3×3 **D**-matrix is formed by extracting rows and columns 1, 2, and 4 from the original D-matrix.

Evaluation of the integrals is done by Gauss integration.

3.4.14 plani8s

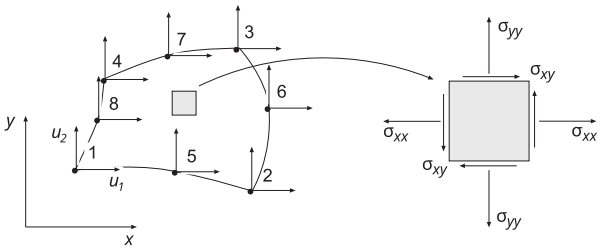
Purpose

Compute stresses and strains in an 8 node isoparametric element in plane strain or plane stress.

align

center

Syntax




```
[es,et,eci]=plani8s(ex,ey,ep,D,ed)
```

Description

`plani8s` computes stresses `es` and the strains `et` in an 8 node isoparametric element in plane strain or plane stress.

The input variables **ex**, **ey**, **ep** and the matrix **D** are defined in `plani8e`. The vector **ed** contains the nodal displacements $\mathbf{a}^e$ of the element and is obtained by the function `extract` as

$$\mathbf{ed} = (\mathbf{a}^e)^T = [\, u_1 \;\; u_2 \;\; \ldots \;\; u_{16} \,]$$

The output variables

$$\text{es} = \boldsymbol{\sigma}^T = \begin{bmatrix} \sigma_{xx}^1 & \sigma_{yy}^1 & [\sigma_{zz}^1] & \sigma_{xy}^1 & [\sigma_{xz}^1] & [\sigma_{yz}^1] \\ \sigma_{xx}^2 & \sigma_{yy}^2 & [\sigma_{zz}^2] & \sigma_{xy}^2 & [\sigma_{xz}^2] & [\sigma_{yz}^2] \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \sigma_{xx}^{n^2} & \sigma_{yy}^{n^2} & [\sigma_{zz}^{n^2}] & \sigma_{xy}^{n^2} & [\sigma_{xz}^{n^2}] & [\sigma_{yz}^{n^2}] \end{bmatrix}$$

$$\text{et} = \boldsymbol{\varepsilon}^T = \begin{bmatrix} \varepsilon_{xx}^1 & \varepsilon_{yy}^1 & [\varepsilon_{zz}^1] & \gamma_{xy}^1 & [\gamma_{xz}^1] & [\gamma_{yz}^1] \\ \varepsilon_{xx}^2 & \varepsilon_{yy}^2 & [\varepsilon_{zz}^2] & \gamma_{xy}^2 & [\gamma_{xz}^2] & [\gamma_{yz}^2] \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \varepsilon_{xx}^{n^2} & \varepsilon_{yy}^{n^2} & [\varepsilon_{zz}^{n^2}] & \gamma_{xy}^{n^2} & [\gamma_{xz}^{n^2}] & [\gamma_{yz}^{n^2}] \end{bmatrix} \qquad \mathbf{eci} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_{n^2} & y_{n^2} \end{bmatrix}$$

contain the stress and strain components, and the coordinates of the integration points. The index n denotes the number of integration points used within the element, cf. `plani8e`. The number of columns in `es` and `et` follows the size of `D`. Note that for plane stress $\varepsilon_{zz} \neq 0$, and for plane strain $\sigma_{zz} \neq 0$.

Theory

The strains and stresses are computed according to

$$\boldsymbol{\varepsilon} = \mathbf{B}^e \, \mathbf{a}^e$$

$$\boldsymbol{\sigma} = \mathbf{D} \, \boldsymbol{\varepsilon}$$

where the matrices **D**, $\mathbf{B}^e$, and $\mathbf{a}^e$ are described in `plani8e`, and where the integration points are chosen as evaluation points.

3.4.15 plani8f

Purpose

Compute internal element force vector in an 8 node isoparametric element in plane strain or plane stress.

Syntax

```
ef = plani8f(ex, ey, ep, es)
```

Description

`plani8f` computes the internal element forces ef in an 8 node isoparametric element in plane strain or plane stress.

The input variables **ex**, **ey** and **ep** are defined in `plani8e`, and the input variable **es** is defined in `plani8s`.

The output variable

$$\mathrm{ef} = \mathbf{f}_i^{eT} = [\, f_{i1} \; f_{i2} \; \cdots \; f_{i16} \,]$$

contains the components of the internal force vector.

Theory

The internal force vector is computed according to

$$\mathbf{f}_i^e = \int_A \mathbf{B}^{eT} \boldsymbol{\sigma} \; t \; dA$$

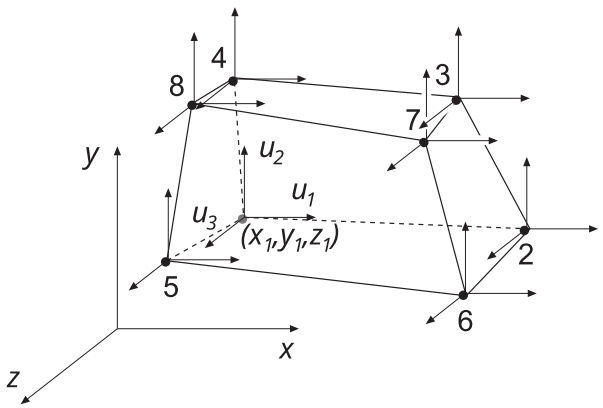
where the matrices $\mathbf{B}^e$ and $\boldsymbol{\sigma}$ are defined in `plani8e` and `plani8s`, respectively.

Evaluation of the integral is done by Gauss integration.

3.4.16 soli8e

Purpose

Compute element matrices for an 8 node isoparametric solid element.



Syntax

```
Ke = soli8e(ex, ey, ez, ep, D)
[Ke, fe] = soli8e(ex, ey, ez, ep, D, eq)
```

Description

`soli8e` provides an element stiffness matrix `Ke` and an element load vector `fe` for an 8 node isoparametric solid element.

The element nodal coordinates x_1, y_1, z_1, x_2 etc. are supplied to the function by **ex**, **ey** and **ez**, and the number of Gauss points n are supplied by **ep**.

$(n \times n \times n)$ integration points, $n = 1, 2, 3$

The material properties are supplied by the constitutive matrix `D`. Any arbitrary **D**-matrix with dimensions (6×6) may be given. For an isotropic elastic material the constitutive matrix can be formed by the function `hooke`, see Section *Material functions*.

$$\mathbf{ex} = [\, x_1,\, x_2,\, \ldots,\, x_8 \,]$$
$$\mathbf{ey} = [\, y_1,\, y_2,\, \ldots,\, y_8 \,]$$
$$\mathbf{ez} = [\, z_1,\, z_2,\, \ldots,\, z_8 \,]$$

$$\mathbf{ep} = [\, n \,]$$

$$\mathbf{D} = \begin{bmatrix} D_{11} & D_{12} & \cdots & D_{16} \\ D_{21} & D_{22} & \cdots & D_{26} \\ \vdots & \vdots & \ddots & \vdots \\ D_{61} & D_{62} & \cdots & D_{66} \end{bmatrix}$$

If different $\mathbf{D}_i$ matrices are used in the Gauss points these $\mathbf{D}_i$ matrices are stored in a global vector `D`. For numbering of the Gauss points, see `eci` in `soli8s`.

$$\mathbf{D} = \begin{bmatrix} \mathbf{D}_1 \\ \mathbf{D}_2 \\ \vdots \\ \mathbf{D}_{n^3} \end{bmatrix}$$

If uniformly distributed loads are applied to the element, the element load vector `fe` is computed. The input variable

$$\mathrm{eq} = \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}$$

containing loads per unit volume, b_x, b_y, and b_z, is then given.

Theory

The element stiffness matrix $\mathbf{K}^e$ and the element load vector $\mathbf{f}_l^e$, stored in `Ke`

and `fe`, respectively, are computed according to

$$\mathbf{K}^e = \int_V \mathbf{B}^{eT}\mathbf{D}\mathbf{B}^e \, dV$$
$$\mathbf{f}_l^e = \int_V \mathbf{N}^{eT}\mathbf{b} \, dV$$

with the constitutive matrix $\mathbf{D}$ defined by `D`, and the body force vector $\mathbf{b}$ defined by `eq`.

The evaluation of the integrals for the isoparametric 8 node solid element is based on a displacement approximation $\mathbf{u}(\xi,\eta,\zeta)$, expressed in a local coordinate system in terms of the nodal variables $u_1, u_2, \ldots, u_{24}$ as

$$\mathbf{u}(\xi,\eta,\zeta) = \mathbf{N}^e\mathbf{a}^e$$

$$\mathbf{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \quad \mathbf{N}^e = \begin{bmatrix} N_1^e & 0 & 0 & N_2^e & 0 & 0 & \ldots & N_8^e & 0 & 0 \\ 0 & N_1^e & 0 & 0 & N_2^e & 0 & \ldots & 0 & N_8^e & 0 \\ 0 & 0 & N_1^e & 0 & 0 & N_2^e & \ldots & 0 & 0 & N_8^e \end{bmatrix} \quad \mathbf{a}^e = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{24} \end{bmatrix}$$

The element shape functions are given by

$$N_1^e = \frac{1}{8}(1-\xi)(1-\eta)(1-\zeta) \quad N_5^e = \frac{1}{8}(1-\xi)(1-\eta)(1+\zeta)$$
$$N_2^e = \frac{1}{8}(1+\xi)(1-\eta)(1-\zeta) \quad N_6^e = \frac{1}{8}(1+\xi)(1-\eta)(1+\zeta)$$
$$N_3^e = \frac{1}{8}(1+\xi)(1+\eta)(1-\zeta) \quad N_7^e = \frac{1}{8}(1+\xi)(1+\eta)(1+\zeta)$$
$$N_4^e = \frac{1}{8}(1-\xi)(1+\eta)(1-\zeta) \quad N_8^e = \frac{1}{8}(1-\xi)(1+\eta)(1+\zeta)$$

The $\mathbf{B}^e$-matrix is obtained as

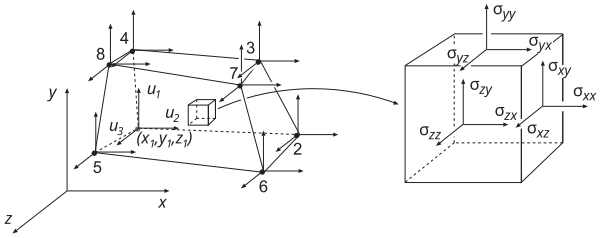
$$\mathbf{B}^e = \tilde{\nabla}\mathbf{N}^e$$

where

$$\tilde{\nabla} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \end{bmatrix} \qquad \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} = (\mathbf{J}^T)^{-1} \begin{bmatrix} \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \\ \frac{\partial}{\partial \zeta} \end{bmatrix}$$

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \zeta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} & \frac{\partial y}{\partial \zeta} \\ \frac{\partial z}{\partial \xi} & \frac{\partial z}{\partial \eta} & \frac{\partial z}{\partial \zeta} \end{bmatrix}$$

Evaluation of the integrals is done by Gauss integration.



3.4.17 soli8s

Purpose

Compute stresses and strains in an 8 node isoparametric solid element.

align

center

Syntax

```
[es,et,eci]=soli8s(ex,ey,ez,ep,D,ed)
```

Description

`soli8s` computes stresses `es` and the strains `et` in an 8 node isoparametric solid element.

The input variables **ex**, **ey**, **ez**, **ep** and the matrix **D** are defined in `soli8e`. The vector **ed** contains the nodal displacements $\mathbf{a}^e$ of the element and is obtained by the function `extract` as

$$\mathbf{ed} = (\mathbf{a}^e)^T = [\ u_1 \ \ u_2 \ \ \ldots \ \ u_{24}\ ]$$

The output variables

$$\text{es} = \boldsymbol{\sigma}^T = \begin{bmatrix} \sigma_{xx}^1 & \sigma_{yy}^1 & \sigma_{zz}^1 & \sigma_{xy}^1 & \sigma_{xz}^1 & \sigma_{yz}^1 \\ \sigma_{xx}^2 & \sigma_{yy}^2 & \sigma_{zz}^2 & \sigma_{xy}^2 & \sigma_{xz}^2 & \sigma_{yz}^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \sigma_{xx}^{n^3} & \sigma_{yy}^{n^3} & \sigma_{zz}^{n^3} & \sigma_{xy}^{n^3} & \sigma_{xz}^{n^3} & \sigma_{yz}^{n^3} \end{bmatrix}$$

$$\text{et} = \boldsymbol{\varepsilon}^T = \begin{bmatrix} \varepsilon_{xx}^1 & \varepsilon_{yy}^1 & \varepsilon_{zz}^1 & \gamma_{xy}^1 & \gamma_{xz}^1 & \gamma_{yz}^1 \\ \varepsilon_{xx}^2 & \varepsilon_{yy}^2 & \varepsilon_{zz}^2 & \gamma_{xy}^2 & \gamma_{xz}^2 & \gamma_{yz}^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \varepsilon_{xx}^{n^3} & \varepsilon_{yy}^{n^3} & \varepsilon_{zz}^{n^3} & \gamma_{xy}^{n^3} & \gamma_{xz}^{n^3} & \gamma_{yz}^{n^3} \end{bmatrix} \qquad \mathbf{eci} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_{n^3} & y_{n^3} & z_{n^3} \end{bmatrix}$$

contain the stress and strain components, and the coordinates of the integration points. The index n denotes the number of integration points used within the element, cf. `soli8e`.

Theory

The strains and stresses are computed according to

$$\boldsymbol{\varepsilon} = \mathbf{B}^e \mathbf{a}^e$$

$$\boldsymbol{\sigma} = \mathbf{D}\boldsymbol{\varepsilon}$$

where the matrices $\mathbf{D}$, $\mathbf{B}^e$, and $\mathbf{a}^e$ are described in `soli8e`, and where the integration points are chosen as evaluation points.

3.4.18 soli8f

Purpose

Compute internal element force vector in an 8 node isoparametric solid element.

Syntax

```
ef = soli8f(ex, ey, ez, ep, es)
```

Description

`soli8f` computes the internal element forces ef in an 8 node isoparametric solid element.

The input variables **ex**, **ey**, **ez** and **ep** are defined in `soli8e`, and the input variable **es** is defined in `soli8s`.

The output variable

$$\mathrm{ef} = \mathbf{f}_i^{eT} = [\, f_{i1} \;\; f_{i2} \;\; \cdots \;\; f_{i24} \,]$$

contains the components of the internal force vector.

Theory

The internal force vector is computed according to

$$\mathbf{f}_i^e = \int_V \mathbf{B}^{eT} \boldsymbol{\sigma} \; dV$$

where the matrices $\mathbf{B}$ and $\boldsymbol{\sigma}$ are defined in `soli8e` and `soli8s`, respectively.

Evaluation of the integral is done by Gauss integration.

3.5 Beam element functions

Beam elements are available for one, two, and three dimensional linear static analysis. Two dimensional beam elements for nonlinear geometric and dynamic analysis are also available.

Table 13: 1D beam elements

beam1e	Compute element matrices
beam1s	Compute section forces
beam1we	Compute element matrices for beam element on elastic foundation
beam1ws	Compute section forces for beam element on elastic foundation

Table 14: 2D beam elements

beam2e	Compute element matrices
beam2s	Compute section forces
beam2te	Compute element matrices for Timoshenko beam element
beam2ts	Compute section forces for Timoshenko beam element
beam2we	Compute element matrices for beam element on elastic foundation
beam2ws	Compute section forces for beam element on elastic foundation
beam2ge	Compute element matrices for geometric nonlinear beam element
beam2gs	Compute section forces for geometric nonlinear beam element
beam2gxe	Compute element matrices for geometric nonlinear exact beam element
beam2gxs	Compute section forces for geometric nonlinear exact beam element
beam2de	Compute element matrices for dynamic analysis
beam2ds	Compute section forces for dynamic analysis

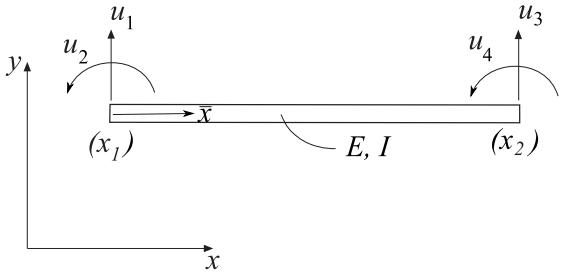
Table 15: 3D beam elements

beam3e	Compute element matrices
beam3s	Compute section forces

3.5.1 beam1e

Purpose

Compute element stiffness matrix for a one dimensional beam element.



Syntax

```
Ke = cfc.beam1e(ex, ep)
Ke, fe = cfc.beam1e(ex, ep, eq)
```

Description

`beam1e` provides the global element stiffness matrix $\mathbf{K}^e$ for a one dimensional beam element.

The input variables

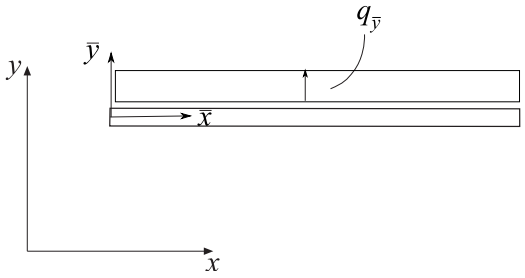
ex= $[x_1 \;\; x_2]$ ep= $[E \; I]$

supply the element nodal coordinates x_1 and x_2, the modulus of elasticity E and the moment of inertia I.

The element load vector $\mathbf{f}_l^e$ can also be computed if a uniformly distributed load is applied to the element. The optional input variable

eq= $[q_{\bar{y}}]$

then contains the distributed load per unit length, $q_{\bar{y}}$.



Theory

The element stiffness matrix $\bar{\mathbf{K}}^e$, stored in Ke, is computed according to

$$\bar{\mathbf{K}}^e = \frac{D_{EI}}{L^3}\begin{bmatrix} 12 & 6L & -12 & 6L \\ 6L & 4L^2 & -6L & 2L^2 \\ -12 & -6L & 12 & -6L \\ 6L & 2L^2 & -6L & 4L^2 \end{bmatrix}$$

where the bending stiffness D_{EI} and the length L are given by

$$D_{EI} = EI; \quad L = x_2 - x_1$$

The element loads $\bar{\mathbf{f}}_l^e$ stored in the variable fe are computed according to

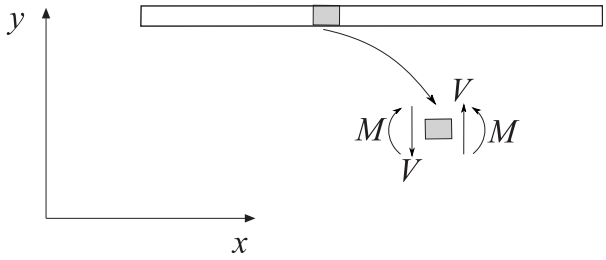
$$\bar{\mathbf{f}}_l^e = q_{\bar{y}}\begin{bmatrix} \frac{L}{2} \\ \frac{L^2}{12} \\ \frac{L}{2} \\ -\frac{L^2}{12} \end{bmatrix}$$

3.5.2 beam1s

Purpose

Compute section forces in a one dimensional beam element.

Syntax




```
es = cfc.beam1s(ex, ep, ed)
es = cfc.beam1s(ex, ep, ed, eq)
es, edi, eci = cfc.beam1s(ex, ep, ed, eq, n)
```

Description

`beam1s` computes the section forces and displacements in local directions along the beam element `beam1e`.

The input variables `ex`, `ep` and `eq` are defined in `beam1e`, and the element displacements, stored in `ed`, are obtained by the function `extract_ed`. If distributed loads are applied to the element, the variable `eq` must be included. The number of evaluation points for section forces and displacements are determined by `n`. If `n` is omitted, only the ends of the beam are evaluated.

The output variables

$$\texttt{es}=\begin{bmatrix} V(0) & M(0) \\ V(\bar{x}_2) & M(\bar{x}_2) \\ \vdots & \vdots \\ V(\bar{x}_{n-1}) & M(\bar{x}_{n-1}) \\ V(L) & M(L) \end{bmatrix} \qquad \texttt{edi}=\begin{bmatrix} v(0) \\ v(\bar{x}_2) \\ \vdots \\ v(\bar{x}_{n-1}) \\ v(L) \end{bmatrix} \qquad \texttt{eci}=\begin{bmatrix} 0 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_{n-1} \\ L \end{bmatrix}$$

contain the section forces, the displacements, and the evaluation points on the local $\bar{x}$-axis. L is the length of the beam element.

Theory

The nodal displacements in local coordinates are given by

$$\bar{\mathbf{a}}^e = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \\ \bar{u}_3 \\ \bar{u}_4 \end{bmatrix}$$

where the transpose of $\mathbf{a}^e$ is stored in `ed`.

The displacement $v(\bar{x})$, the bending moment $M(\bar{x})$ and the shear force $V(\bar{x})$ are computed from

$$v(\bar{x}) = \mathbf{N}\bar{\mathbf{a}}^e + v_p(\bar{x})$$

$$M(\bar{x}) = D_{EI}\mathbf{B}\bar{\mathbf{a}}^e + M_p(\bar{x})$$

$$V(\bar{x}) = -D_{EI}\frac{d\mathbf{B}}{dx}\bar{\mathbf{a}}^e + V_p(\bar{x})$$

where

$$\mathbf{N} = \begin{bmatrix} 1 & \bar{x} & \bar{x}^2 & \bar{x}^3 \end{bmatrix} \mathbf{C}^{-1}$$

$$\mathbf{B} = \begin{bmatrix} 0 & 0 & 2 & 6\bar{x} \end{bmatrix} \mathbf{C}^{-1}$$

$$\frac{d\mathbf{B}}{dx} = \begin{bmatrix} 0 & 0 & 0 & 6 \end{bmatrix} \mathbf{C}^{-1}$$

$$v_p(\bar{x}) = \frac{q_{\bar{y}}}{D_{EI}}\left(\frac{\bar{x}^4}{24} - \frac{L\bar{x}^3}{12} + \frac{L^2\bar{x}^2}{24}\right)$$

$$M_p(\bar{x}) = q_{\bar{y}}\left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2} + \frac{L^2}{12}\right)$$

$$V_p(\bar{x}) = -q_{\bar{y}}\left(\bar{x} - \frac{L}{2}\right)$$

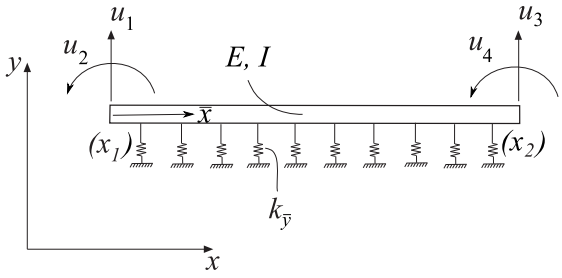
in which D_{EI}, L, and $q_{\bar{y}}$ are defined in `beam1e` and

$$\mathbf{C}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{3}{L^2} & -\frac{2}{L} & \frac{3}{L^2} & -\frac{1}{L} \\ \frac{2}{L^3} & \frac{1}{L^2} & -\frac{2}{L^3} & \frac{1}{L^2} \end{bmatrix}$$

3.5.3 beam1we

Purpose

Compute element stiffness matrix for a one dimensional beam element on elastic support.



Syntax

```
Ke = cfc.beam1we(ex, ep)
Ke, fe = cfc.beam1we(ex, ep, eq)
```

Description

`beam1we` provides the global element stiffness matrix $\mathbf{K}^e$ for a one dimensional beam element with elastic support.

The input variables

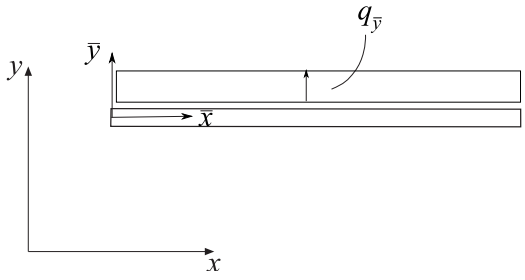
`ex`$= [x_1 \;\; x_2]$ `ep`$= [E \;\; I \;\; k_{\bar{y}}]$

supply the element nodal coordinates x_1 and x_2, the modulus of elasticity E, the moment of inertia I, and the spring stiffness in the transverse direction $k_{\bar{y}}$.

The element load vector $\mathbf{f}_l^e$ can also be computed if a uniformly distributed load is applied to the element. The optional input variable

`eq`$= [q_{\bar{y}}]$

contains the distributed load per unit length, $q_{\bar{y}}$.



Theory

The element stiffness matrix $\bar{\mathbf{K}}^e$, stored in `Ke`, is computed according to

$$\bar{\mathbf{K}}^e = \bar{\mathbf{K}}_0^e + \bar{\mathbf{K}}_s^e$$

where

$$\bar{\mathbf{K}}_0^e = \frac{D_{EI}}{L^3}\begin{bmatrix} 12 & 6L & -12 & 6L \\ 6L & 4L^2 & -6L & 2L^2 \\ -12 & -6L & 12 & -6L \\ 6L & 2L^2 & -6L & 4L^2 \end{bmatrix}$$

and

$$\bar{\mathbf{K}}_s^e = \frac{k_{\bar{y}}L}{420}\begin{bmatrix} 156 & 22L & 54 & -13L \\ 22L & 4L^2 & 13L & -3L^2 \\ 54 & 13L & 156 & -22L \\ -13L & -3L^2 & -22L & 4L^2 \end{bmatrix}$$

where the bending stiffness D_{EI} and the length L are given by

$$D_{EI} = EI \qquad L = x_2 - x_1$$

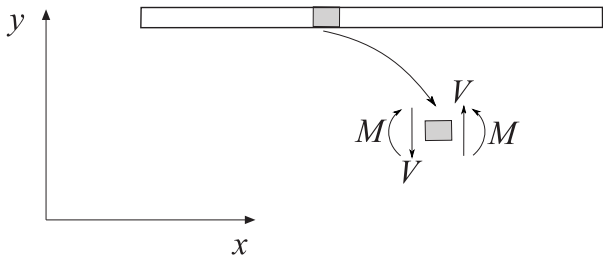
The element loads $\bar{\mathbf{f}}_l^e$ stored in the variable `fe` are computed according to

$$\bar{\mathbf{f}}_l^e = q_{\bar{y}} \begin{bmatrix} \frac{L}{2} \\ \frac{L^2}{12} \\ \frac{L}{2} \\ -\frac{L^2}{12} \end{bmatrix}$$

3.5.4 beam1ws

Purpose

Compute section forces in a one dimensional beam element with elastic support.



Syntax

```
es = cfc.beam1ws(ex, ep, ed)
es = cfc.beam1ws(ex, ep, ed, eq)
es, edi, eci = cfc.beam1ws(ex, ep, ed, eq, n)
```

Description

`beam1ws` computes the section forces and displacements in local directions along the beam element `beam1we`.

The input variables `ex`, `ep` and `eq` are defined in `beam1we`, and the element displacements, stored in `ed`, are obtained by the function `extract_ed`. If distributed loads are applied to the element, the variable `eq` must be included. The number of evaluation points for section forces and displacements are determined by `n`. If `n` is omitted, only the ends of the beam are evaluated.

The output variables

$$\texttt{es}= \begin{bmatrix} V(0) & M(0) \\ V(\bar{x}_2) & M(\bar{x}_2) \\ \vdots & \vdots \\ V(\bar{x}_{n-1}) & M(\bar{x}_{n-1}) \\ V(L) & M(L) \end{bmatrix} \qquad \texttt{edi}= \begin{bmatrix} v(0) \\ v(\bar{x}_2) \\ \vdots \\ v(\bar{x}_{n-1}) \\ v(L) \end{bmatrix} \qquad \texttt{eci}= \begin{bmatrix} 0 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_{n-1} \\ L \end{bmatrix}$$

contain the section forces, the displacements, and the evaluation points on the local $\bar{x}$-axis. L is the length of the beam element.

Theory

The nodal displacements in local coordinates are given by

$$\bar{\mathbf{a}}^e = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \\ \bar{u}_3 \\ \bar{u}_4 \end{bmatrix}$$

where the transpose of $\mathbf{a}^e$ is stored in `ed`.

The displacement $v(\bar{x})$, the bending moment $M(\bar{x})$ and the shear force $V(\bar{x})$ are computed from

$$v(\bar{x}) = \mathbf{N}\bar{\mathbf{a}}^e + v_p(\bar{x})$$

$$M(\bar{x}) = D_{EI}\mathbf{B}\bar{\mathbf{a}}^e + M_p(\bar{x})$$

$$V(\bar{x}) = -D_{EI}\frac{d\mathbf{B}}{dx}\bar{\mathbf{a}}^e + V_p(\bar{x})$$

where

$$\mathbf{N} = \begin{bmatrix} 1 & \bar{x} & \bar{x}^2 & \bar{x}^3 \end{bmatrix}\mathbf{C}^{-1}$$

$$\mathbf{B} = \begin{bmatrix} 0 & 0 & 2 & 6\bar{x} \end{bmatrix}\mathbf{C}^{-1}$$

$$\frac{d\mathbf{B}}{dx} = \begin{bmatrix} 0 & 0 & 0 & 6 \end{bmatrix}\mathbf{C}^{-1}$$

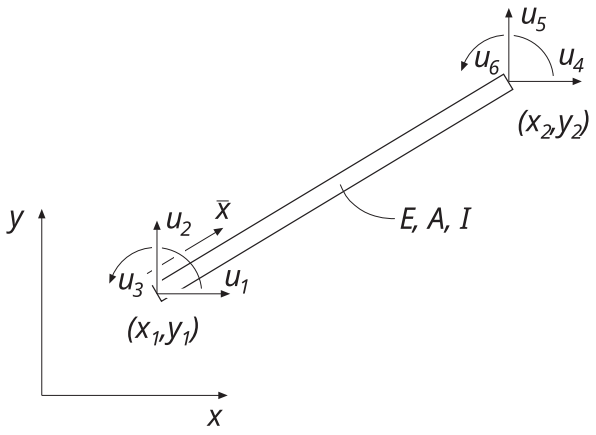
$$v_p(\bar{x}) = -\frac{k_{\bar{y}}}{D_{EI}}\begin{bmatrix} \frac{\bar{x}^4-2L\bar{x}^3+L^2\bar{x}^2}{24} \\ \frac{\bar{x}^5-3L^2\bar{x}^3+2L^3\bar{x}^2}{120} \\ \frac{\bar{x}^6-4L^3\bar{x}^3+3L^4\bar{x}^2}{360} \\ \frac{\bar{x}^7-5L^4\bar{x}^3+4L^5\bar{x}^2}{840} \end{bmatrix}^T \mathbf{C}^{-1}\bar{\mathbf{a}}^e + \frac{q_{\bar{y}}}{D_{EI}}\left(\frac{\bar{x}^4}{24} - \frac{L\bar{x}^3}{12} + \frac{L^2\bar{x}^2}{24}\right)$$

$$M_p(\bar{x}) = -k_{\bar{y}}\begin{bmatrix} \frac{6\bar{x}^2-6L\bar{x}+L^2}{12} \\ \frac{10\bar{x}^3-9L^2\bar{x}+2L^3}{60} \\ \frac{5\bar{x}^4-4L^3\bar{x}+L^4}{60} \\ \frac{21\bar{x}^5-15L^4\bar{x}+4L^5}{420} \end{bmatrix}^T \mathbf{C}^{-1}\bar{\mathbf{a}}^e + q_{\bar{y}}\left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2} + \frac{L^2}{12}\right)$$

$$V_p(\bar{x}) = k_{\bar{y}}\begin{bmatrix} \frac{2\bar{x}-L}{2} \\ \frac{10\bar{x}^2-3L^2}{20} \\ \frac{5\bar{x}^3-L^3}{15} \\ \frac{7\bar{x}^4-L^4}{28} \end{bmatrix}^T \mathbf{C}^{-1}\bar{\mathbf{a}}^e - q_{\bar{y}}\left(\bar{x} - \frac{L}{2}\right)$$

in which D_{EI}, $k_{\bar{y}}$, L, and $q_{\bar{y}}$ are defined in `beam1we` and

$$\mathbf{C}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{3}{L^2} & -\frac{2}{L} & \frac{3}{L^2} & -\frac{1}{L} \\ \frac{2}{L^3} & \frac{1}{L^2} & -\frac{2}{L^3} & \frac{1}{L^2} \end{bmatrix}$$



3.5.5 beam2e

Purpose

Compute element stiffness matrix for a two-dimensional beam element.

Syntax

```
Ke = cfc.beam2e(ex, ey, ep)
Ke, fe = cfc.beam2e(ex, ey, ep, eq)
```

Description

`beam2e` provides the global element stiffness matrix $\mathbf{K}^e$ for a two-dimensional beam element.

The input variables:

`ex`$= [x_1 \;\; x_2]$ `ey`$= [y_1 \;\; y_2]$ `ep`$= [E \; A \; I]$

supply the element nodal coordinates x_1, y_1, x_2, and y_2, the modulus of elasticity E, the cross-section area A, and the moment of inertia I.

The element load vector $\mathbf{f}_l^e$ can also be computed if a uniformly distributed transverse load is applied to the element. The optional input variable:

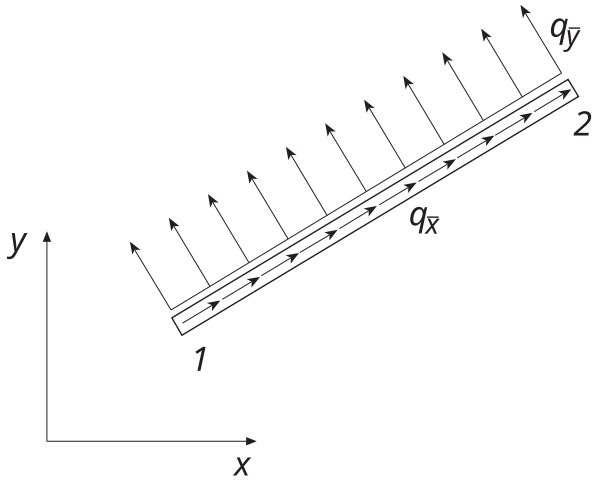
`eq`$= [q_{\bar{x}} \; q_{\bar{y}}]$

contains the distributed loads per unit length, $q_{\bar{x}}$ and $q_{\bar{y}}$.

Theory

The element stiffness matrix $\mathbf{K}^e$, stored in `Ke`, is computed according to:

$$\mathbf{K}^e = \mathbf{G}^T \bar{\mathbf{K}}^e \mathbf{G}$$



where:

$$\bar{\mathbf{K}}^e = \begin{bmatrix} \frac{D_{EA}}{L} & 0 & 0 & -\frac{D_{EA}}{L} & 0 & 0 \\ 0 & \frac{12D_{EI}}{L^3} & \frac{6D_{EI}}{L^2} & 0 & -\frac{12D_{EI}}{L^3} & \frac{6D_{EI}}{L^2} \\ 0 & \frac{6D_{EI}}{L^2} & \frac{4D_{EI}}{L} & 0 & -\frac{6D_{EI}}{L^2} & \frac{2D_{EI}}{L} \\ -\frac{D_{EA}}{L} & 0 & 0 & \frac{D_{EA}}{L} & 0 & 0 \\ 0 & -\frac{12D_{EI}}{L^3} & -\frac{6D_{EI}}{L^2} & 0 & \frac{12D_{EI}}{L^3} & -\frac{6D_{EI}}{L^2} \\ 0 & \frac{6D_{EI}}{L^2} & \frac{2D_{EI}}{L} & 0 & -\frac{6D_{EI}}{L^2} & \frac{4D_{EI}}{L} \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} n_{x\bar{x}} & n_{y\bar{x}} & 0 & 0 & 0 & 0 \\ n_{x\bar{y}} & n_{y\bar{y}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} & 0 \\ 0 & 0 & 0 & n_{x\bar{y}} & n_{y\bar{y}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

where the axial stiffness D_{EA}, the bending stiffness D_{EI}, and the length L are given by:

$$D_{EA} = EA, \quad D_{EI} = EI, \quad L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The transformation matrix $\mathbf{G}$ contains the direction cosines:

$$n_{x\bar{x}} = n_{y\bar{y}} = \frac{x_2 - x_1}{L}, \quad n_{y\bar{x}} = -n_{x\bar{y}} = \frac{y_2 - y_1}{L}$$

The element loads $\mathbf{f}_l^e$, stored in the variable `fe`, are computed according to:

$$\mathbf{f}_l^e = \mathbf{G}^T \bar{\mathbf{f}}_l^e$$

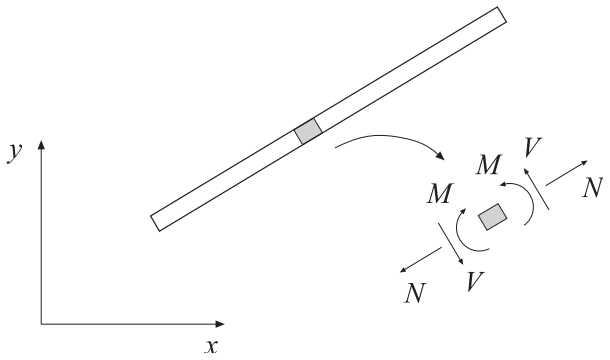
where:

$$\bar{\mathbf{f}}_l^e = \begin{bmatrix} \frac{q_{\bar{x}} L}{2} \\ \frac{q_{\bar{y}} L}{2} \\ \frac{q_{\bar{y}} L^2}{12} \\ \frac{q_{\bar{x}} L}{2} \\ \frac{q_{\bar{y}} L}{2} \\ -\frac{q_{\bar{y}} L^2}{12} \end{bmatrix}$$

3.5.6 beam2s

Purpose

Compute section forces in a two-dimensional beam element.



Syntax

```
es = cfc.beam2s(ex, ey, ep, ed)
es = cfc.beam2s(ex, ey, ep, ed, eq)
es, edi = cfc.beam2s(ex, ey, ep, ed, eq, n)
es, edi, eci = cfc.beam2s(ex, ey, ep, ed, eq, n)
```

Description

`beam2s` computes the section forces and displacements in local directions along the beam element `beam2e`.

The input variables `ex`, `ey`, `ep` and `eq` are defined in `beam2e`.

The element displacements, stored in `ed`, are obtained by the function `extract_ed`. If a distributed load is applied to the element, the variable `eq` must be included. The number of evaluation points for section forces and displacements is determined by `n`. If `n` is omitted, only the ends of the beam are evaluated.

The output variables:

VERIFIAR INTE FUNGERAR FÖR n=3?

$$\texttt{es}= \begin{bmatrix} N(0) & V(0) & M(0) \\ N(\bar{x}_2) & V(\bar{x}_2) & M(\bar{x}_2) \\ \vdots & \vdots & \vdots \\ N(\bar{x}_{n-1}) & V(\bar{x}_{n-1}) & M(\bar{x}_{n-1}) \\ N(L) & V(L) & M(L) \end{bmatrix} \qquad \texttt{edi}= \begin{bmatrix} u(0) & v(0) \\ u(\bar{x}_2) & v(\bar{x}_2) \\ \vdots & \vdots \\ u(\bar{x}_{n-1}) & v(\bar{x}_{n-1}) \\ u(L) & v(L) \end{bmatrix}$$

$$\texttt{eci}= \begin{bmatrix} 0 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_{n-1} \\ L \end{bmatrix}$$

contain the section forces, the displacements, and the evaluation points on the local $\bar{x}$-axis. L is the length of the beam element.

Theory

The nodal displacements in local coordinates are given by:

$$\bar{\mathbf{a}}^e = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \\ \bar{u}_3 \\ \bar{u}_4 \\ \bar{u}_5 \\ \bar{u}_6 \end{bmatrix} = \mathbf{G}\mathbf{a}^e$$

where **G** is described in `beam2e` and the transpose of $\mathbf{a}^e$ is stored in `ed`.

The displacements associated with bar action and beam action are determined as:

$$\bar{\mathbf{a}}^e_{\text{bar}} = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_4 \end{bmatrix}, \quad \bar{\mathbf{a}}^e_{\text{beam}} = \begin{bmatrix} \bar{u}_2 \\ \bar{u}_3 \\ \bar{u}_5 \\ \bar{u}_6 \end{bmatrix}$$

The displacement $u(\bar{x})$ and the normal force $N(\bar{x})$ are computed from:

$$u(\bar{x}) = \mathbf{N}_{\text{bar}}\bar{\mathbf{a}}^e_{\text{bar}} + u_p(\bar{x})$$

$$N(\bar{x}) = D_{EA}\mathbf{B}_{\text{bar}}\bar{\mathbf{a}}^e + N_p(\bar{x})$$

where:

$$\mathbf{N}_{\text{bar}} = \begin{bmatrix} 1 & \bar{x} \end{bmatrix} \mathbf{C}^{-1}_{\text{bar}} = \begin{bmatrix} 1 - \frac{\bar{x}}{L} & \frac{\bar{x}}{L} \end{bmatrix}$$

$$\mathbf{B}_{\text{bar}} = \begin{bmatrix} 0 & 1 \end{bmatrix} \mathbf{C}^{-1}_{\text{bar}} = \begin{bmatrix} -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

$$u_p(\bar{x}) = -\frac{q_{\bar{x}}}{D_{EA}} \left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2} \right)$$

$$N_p(\bar{x}) = -q_{\bar{x}} \left(\bar{x} - \frac{L}{2} \right)$$

where D_{EA}, L, and $q_{\bar{x}}$ are defined in `beam2e`, and:

$$\mathbf{C}_{\text{bar}}^{-1} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

The displacement $v(\bar{x})$, the bending moment $M(\bar{x})$, and the shear force $V(\bar{x})$ are computed from:

$$v(\bar{x}) = \mathbf{N}_{\text{beam}} \mathbf{a}_{\text{beam}}^{e} + v_p(\bar{x})$$

$$M(\bar{x}) = D_{EI} \mathbf{B}_{\text{beam}} \mathbf{a}_{\text{beam}}^{e} + M_p(\bar{x})$$

$$V(\bar{x}) = -D_{EI} \frac{d\mathbf{B}_{\text{beam}}}{d\bar{x}} \mathbf{a}_{\text{beam}}^{e} + V_p(\bar{x})$$

where:

$$\mathbf{N}_{\text{beam}} = \begin{bmatrix} 1 & \bar{x} & \bar{x}^2 & \bar{x}^3 \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$\mathbf{B}_{\text{beam}} = \begin{bmatrix} 0 & 0 & 2 & 6\bar{x} \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$\frac{d\mathbf{B}_{\text{beam}}}{d\bar{x}} = \begin{bmatrix} 0 & 0 & 0 & 6 \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$v_p(\bar{x}) = \frac{q_{\bar{y}}}{D_{EI}} \left(\frac{\bar{x}^4}{24} - \frac{L\bar{x}^3}{12} + \frac{L^2\bar{x}^2}{24} \right)$$

$$M_p(\bar{x}) = q_{\bar{y}} \left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2} + \frac{L^2}{12} \right)$$

$$V_p(\bar{x}) = -q_{\bar{y}} \left(\bar{x} - \frac{L}{2} \right)$$

where D_{EI}, L, and $q_{\bar{y}}$ are defined in `beam2e`, and:

$$\mathbf{C}_{\text{beam}}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{3}{L^2} & -\frac{2}{L} & \frac{3}{L^2} & -\frac{1}{L} \\ \frac{2}{L^3} & \frac{1}{L^2} & -\frac{2}{L^3} & \frac{1}{L^2} \end{bmatrix}$$

3.5.7 beam2te

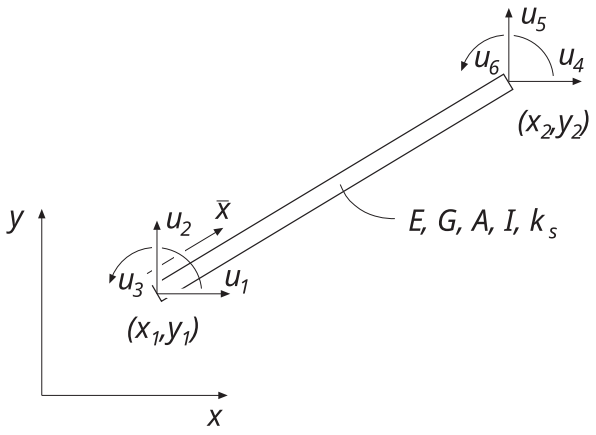
Purpose

Compute element stiffness matrix for a two dimensional Timoshenko beam element.

Two dimensional beam element

Syntax

```
Ke = cfc.beam2te(ex, ey, ep)
Ke, fe = cfc.beam2te(ex, ey, ep, eq)
```



Description

`beam2te` provides the global element stiffness matrix $\mathbf{K}^e$ for a two dimensional Timoshenko beam element.

The input variables

`ex`$= [x_1 \;\; x_2]$ `ey`$= [y_1 \;\; y_2]$ `ep`$= [E \;\; G \;\; A \;\; I \;\; k_s]$

supply the element nodal coordinates x_1, y_1, x_2, and y_2, the modulus of elasticity E, the shear modulus G, the cross section area A, the moment of inertia I and the shear correction factor k_s.

The element load vector $\mathbf{f}_l^e$ can also be computed if uniformly distributed loads are applied to the element. The optional input variable

`eq`$= [q_{\bar{x}} \;\; q_{\bar{y}}]$

contains the distributed loads per unit length, $q_{\bar{x}}$ and $q_{\bar{y}}$.

Theory

The element stiffness matrix $\mathbf{K}^e$, stored in `Ke`, is computed according to

$$\mathbf{K}^e = \mathbf{G}^T \bar{\mathbf{K}}^e \mathbf{G}$$

where $\mathbf{G}$ is described in `beam2e`, and $\bar{\mathbf{K}}^e$ is given by

$$\bar{\mathbf{K}}^e = \begin{bmatrix} \frac{D_{EA}}{L} & 0 & 0 & -\frac{D_{EA}}{L} & 0 & 0 \\ 0 & \frac{12D_{EI}}{L^3(1+\mu)} & \frac{6D_{EI}}{L^2(1+\mu)} & 0 & -\frac{12D_{EI}}{L^3(1+\mu)} & \frac{6D_{EI}}{L^2(1+\mu)} \\ 0 & \frac{6D_{EI}}{L^2(1+\mu)} & \frac{4D_{EI}(1+\frac{\mu}{4})}{L(1+\mu)} & 0 & -\frac{6D_{EI}}{L^2(1+\mu)} & \frac{2D_{EI}(1-\frac{\mu}{2})}{L(1+\mu)} \\ -\frac{D_{EA}}{L} & 0 & 0 & \frac{D_{EA}}{L} & 0 & 0 \\ 0 & -\frac{12D_{EI}}{L^3(1+\mu)} & -\frac{6D_{EI}}{L^2(1+\mu)} & 0 & \frac{12D_{EI}}{L^3(1+\mu)} & -\frac{6D_{EI}}{L^2(1+\mu)} \\ 0 & \frac{6D_{EI}}{L^2(1+\mu)} & \frac{2D_{EI}(1-\frac{\mu}{2})}{L(1+\mu)} & 0 & -\frac{6D_{EI}}{L^2(1+\mu)} & \frac{4D_{EI}(1+\frac{\mu}{4})}{L(1+\mu)} \end{bmatrix}$$

where the axial stiffness D_{EA}, the bending stiffness D_{EI}, and the length L are given by

$$D_{EA} = EA \qquad D_{EI} = EI \qquad L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

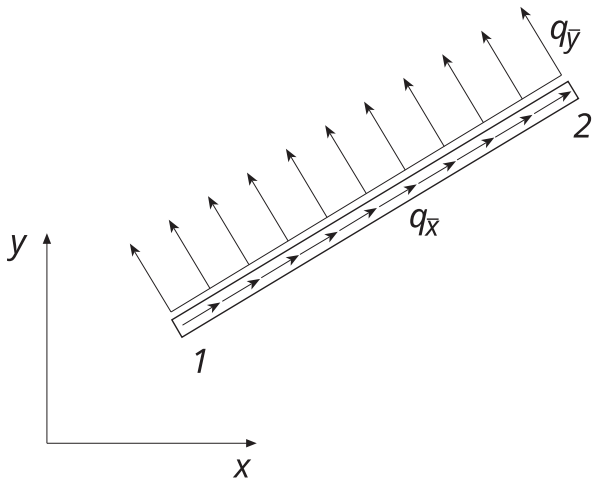


Fig. 1: Uniformly distributed load

and where

$$\mu = \frac{12D_{EI}}{L^2GAk_s}$$

The element loads $\mathbf{f}_l^e$ stored in the variable `fe` are computed according to

$$\mathbf{f}_l^e = \mathbf{G}^T\bar{\mathbf{f}}_l^e$$

where

$$\bar{\mathbf{f}}_l^e = \begin{bmatrix} \frac{q_{\bar{x}}L}{2} \\ \frac{q_{\bar{y}}L}{2} \\ \frac{q_{\bar{y}}L^2}{12} \\ \frac{q_{\bar{x}}L}{2} \\ \frac{q_{\bar{y}}L}{2} \\ -\frac{q_{\bar{y}}L^2}{12} \end{bmatrix}$$

3.5.8 beam2ts

Purpose

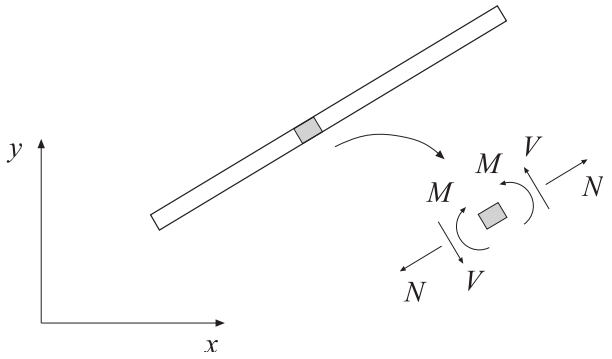
Compute section forces in a two dimensional Timoshenko beam element.

Syntax

```
es = cfc.beam2ts(ex, ey, ep, ed)
es = cfc.beam2ts(ex, ey, ep, ed, eq)
es, edi, eci = cfc.beam2ts(ex, ey, ep, ed, eq, n)
```

Description

`beam2ts` computes the section forces and displacements in local directions along the beam element `beam2te`.



The input variables `ex`, `ey`, `ep` and `eq` are defined in `beam2te`. The element displacements, stored in `ed`, are obtained by the function `extract_ed`. If distributed loads are applied to the element, the variable `eq` must be included. The number of evaluation points for section forces and displacements are determined by `n`. If `n` is omitted, only the ends of the beam are evaluated.

The output variables

$$\text{es} = [\ \mathbf{N}\ \mathbf{V}\ \mathbf{M}\ ]$$
$$\text{edi} = [\ \mathbf{u}\ \mathbf{v}\ \boldsymbol{\theta}\ ]$$
$$\text{eci} = [\bar{\mathbf{x}}]$$

consist of column matrices that contain the section forces, the displacements and rotation of the cross section (note that the rotation θ is not equal to $\frac{d\bar{v}}{d\bar{x}}$), and the evaluation points on the local $\bar{x}$-axis. The explicit matrix expressions are

$$\texttt{es}= \begin{bmatrix} N(0) & V(0) & M(0) \\ N(\bar{x}_2) & V(\bar{x}_2) & M(\bar{x}_2) \\ \vdots & \vdots & \vdots \\ N(\bar{x}_{n-1}) & V(\bar{x}_{n-1}) & M(\bar{x}_{n-1}) \\ N(L) & V(L) & M(L) \end{bmatrix} \qquad \texttt{edi}= \begin{bmatrix} u_1 & v_1 & \theta_1 \\ u_2 & v_2 & \theta_2 \\ \vdots & \vdots & \vdots \\ u_{n-1} & v_{n-1} & \theta_{n-1} \\ u_n & v_n & \theta_n \end{bmatrix}$$

$$\texttt{eci}= \begin{bmatrix} 0 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_{n-1} \\ L \end{bmatrix}$$

where L is the length of the beam element.

Theory

The nodal displacements in local coordinates are given by

$$\bar{\mathbf{a}}^e = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \\ \bar{u}_3 \\ \bar{u}_4 \\ \bar{u}_5 \\ \bar{u}_6 \end{bmatrix} = \mathbf{G}\mathbf{a}^e$$

where `G` is described in `beam2e` and the transpose of $\mathbf{a}^e$ is stored in `ed`. The displacements associated with bar action and beam action are determined as

$$\bar{\mathbf{a}}^e_{\text{bar}} = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_4 \end{bmatrix} \qquad \bar{\mathbf{a}}^e_{\text{beam}} = \begin{bmatrix} \bar{u}_2 \\ \bar{u}_3 \\ \bar{u}_5 \\ \bar{u}_6 \end{bmatrix}$$

The displacement $u(\bar{x})$ and the normal force $N(\bar{x})$ are computed from

$$u(\bar{x}) = \mathbf{N}_{\text{bar}}\bar{\mathbf{a}}^e_{\text{bar}} + u_p(\bar{x})$$

$$N(\bar{x}) = D_{EA}\mathbf{B}_{\text{bar}}\bar{\mathbf{a}}^e + N_p(\bar{x})$$

where

$$\mathbf{N}_{\text{bar}} = \begin{bmatrix} 1 & \bar{x} \end{bmatrix} \mathbf{C}^{-1}_{\text{bar}} = \begin{bmatrix} 1 - \frac{\bar{x}}{L} & \frac{\bar{x}}{L} \end{bmatrix}$$

$$\mathbf{B}_{\text{bar}} = \begin{bmatrix} 0 & 1 \end{bmatrix} \mathbf{C}^{-1}_{\text{bar}} = \begin{bmatrix} -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

$$u_p(\bar{x}) = -\frac{q_{\bar{x}}}{D_{EA}}\left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2}\right)$$

$$N_p(\bar{x}) = -q_{\bar{x}}\left(\bar{x} - \frac{L}{2}\right)$$

in which D_{EA}, L, and $q_{\bar{x}}$ are defined in `beam2te` and

$$\mathbf{C}^{-1}_{\text{bar}} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

The displacement $v(\bar{x})$, the rotation $\theta(\bar{x})$, the bending moment $M(\bar{x})$ and the shear force $V(\bar{x})$ are computed from

$$v(\bar{x}) = \mathbf{N}_{\text{beam},v}\bar{\mathbf{a}}^e_{\text{beam}} + v_p(\bar{x})$$

$$\theta(\bar{x}) = \mathbf{N}_{\text{beam},\theta}\bar{\mathbf{a}}^e_{\text{beam}} + \theta_p(\bar{x})$$

$$M(\bar{x}) = D_{EI}\frac{d\theta}{dx} = D_{EI}\frac{d\mathbf{N}_{\text{beam},\theta}}{d\bar{x}}\bar{\mathbf{a}}^e_{\text{beam}} + M_p(\bar{x})$$

$$V(\bar{x}) = D_{GA}k_s\left(\frac{dv}{dx} - \theta\right) = D_{GA}k_s\left(\frac{d\mathbf{N}_{\text{beam},v}}{d\bar{x}} - \mathbf{N}_{\text{beam},\theta}\right)\bar{\mathbf{a}}^e_{\text{beam}} + V_p(\bar{x})$$

where

$$\mathbf{N}_{\text{beam},v} = \begin{bmatrix} 1 & \bar{x} & \bar{x}^2 & \bar{x}^3 \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$\frac{d\mathbf{N}_{\text{beam},v}}{d\bar{x}} = \begin{bmatrix} 0 & 1 & 2\bar{x} & 3\bar{x}^2 \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$\mathbf{N}_{\text{beam},\theta} = \begin{bmatrix} 0 & 1 & 2\bar{x} & 3\bar{x}^2 + 6\alpha \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$\frac{d\mathbf{N}_{\text{beam},\theta}}{d\bar{x}} = \begin{bmatrix} 0 & 0 & 2 & 6\bar{x} \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$v_p(\bar{x}) = \frac{q_{\bar{y}}}{D_{EI}} \left(\frac{\bar{x}^4}{24} - \frac{L\bar{x}^3}{12} + \frac{L^2\bar{x}^2}{2} \right) + \frac{q_{\bar{y}}}{D_{GA}k_s} \left(-\frac{\bar{x}^2}{2} + \frac{L\bar{x}}{2} \right)$$

$$\theta_p(\bar{x}) = \frac{q_{\bar{y}}}{D_{EI}} \left(\frac{\bar{x}^3}{6} - \frac{L\bar{x}^2}{4} + \frac{L^2\bar{x}}{12} \right)$$

$$M_p(\bar{x}) = q_{\bar{y}} \left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2} + \frac{L^2}{12} \right)$$

$$V_p(\bar{x}) = -q_{\bar{y}} \left(\bar{x} - \frac{L}{2} \right)$$

in which D_{EI}, D_{GA}, k_s, L, and $q_{\bar{y}}$ are defined in `beam2te` and

$$\mathbf{C}_{\text{beam}}^{-1} = \frac{1}{L^2 + 12\alpha} \begin{bmatrix} L^2 + 12\alpha & 0 & 0 & 0 \\ -\frac{12\alpha}{L} & L^2 + 6\alpha & \frac{12\alpha}{L} & -6\alpha \\ -3 & -2L - \frac{6\alpha}{L} & 3 & -L + \frac{6\alpha}{L} \\ \frac{2}{L} & 1 & -\frac{2}{L} & 1 \end{bmatrix}$$

with

$$\alpha = \frac{D_{EI}}{D_{GA}k_s}$$

3.5.9 beam2we

Purpose

Compute element stiffness matrix for a two dimensional beam element on elastic support.

Syntax

```
Ke = cfc.beam2we(ex, ey, ep)
Ke, fe = cfc.beam2we(ex, ey, ep, eq)
```

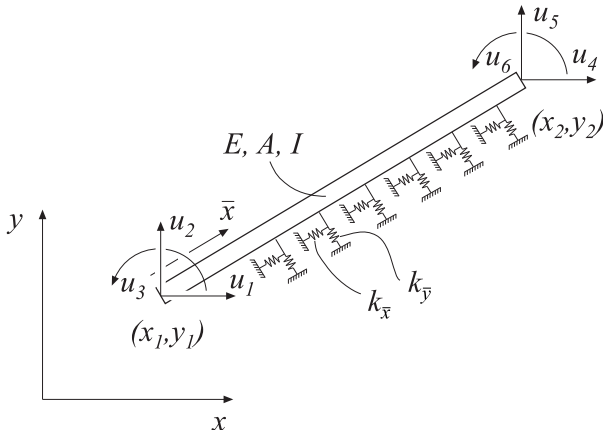
Description

`beam2we` provides the global element stiffness matrix $\mathbf{K}^e$ for a two dimensional beam element with elastic support.

The input variables

`ex`$= [x_1 \;\; x_2]$ `ey`$= [y_1 \;\; y_2]$ `ep`$= [E \;\; A \;\; I \;\; k_{\bar{x}} \;\; k_{\bar{y}}]$

supply the element nodal coordinates x_1, x_2, y_1, and y_2, the modulus of elasticity E, the cross section area A, the moment of inertia I, the spring stiffness



in the axial direction $k_{\bar{x}}$, and the spring stiffness in the transverse direction $k_{\bar{y}}$.

The element load vector $\mathbf{f}_l^e$ can also be computed if uniformly distributed loads are applied to the element. The optional input variable

$$\text{eq} = [q_{\bar{x}} \;\; q_{\bar{y}}]$$

contains the distributed load per unit length, $q_{\bar{x}}$ and $q_{\bar{y}}$.

Theory

The element stiffness matrix $\mathbf{K}^e$, stored in `Ke`, is computed according to

$$\mathbf{K}^e = \mathbf{G}^T \bar{\mathbf{K}}^e \mathbf{G}$$

where

$$\bar{\mathbf{K}}^e = \bar{\mathbf{K}}_0^e + \bar{\mathbf{K}}_s^e$$

$$\bar{\mathbf{K}}_0^e = \begin{bmatrix} \frac{D_{EA}}{L} & 0 & 0 & -\frac{D_{EA}}{L} & 0 & 0 \\ 0 & \frac{12D_{EI}}{L^3} & \frac{6D_{EI}}{L^2} & 0 & -\frac{12D_{EI}}{L^3} & \frac{6D_{EI}}{L^2} \\ 0 & \frac{6D_{EI}}{L^2} & \frac{4D_{EI}}{L} & 0 & -\frac{6D_{EI}}{L^2} & \frac{2D_{EI}}{L} \\ -\frac{D_{EA}}{L} & 0 & 0 & \frac{D_{EA}}{L} & 0 & 0 \\ 0 & -\frac{12D_{EI}}{L^3} & -\frac{6D_{EI}}{L^2} & 0 & \frac{12D_{EI}}{L^3} & -\frac{6D_{EI}}{L^2} \\ 0 & \frac{6D_{EI}}{L^2} & \frac{2D_{EI}}{L} & 0 & -\frac{6D_{EI}}{L^2} & \frac{4D_{EI}}{L} \end{bmatrix}$$

$$\bar{\mathbf{K}}_s^e = \frac{L}{420} \begin{bmatrix} 140k_{\bar{x}} & 0 & 0 & 70k_{\bar{x}} & 0 & 0 \\ 0 & 156k_{\bar{y}} & 22k_{\bar{y}}L & 0 & 54k_{\bar{y}} & -13k_{\bar{y}}L \\ 0 & 22k_{\bar{y}}L & 4k_{\bar{y}}L^2 & 0 & 13k_{\bar{y}}L & -3k_{\bar{y}}L^2 \\ 70k_{\bar{x}} & 0 & 0 & 140k_{\bar{x}} & 0 & 0 \\ 0 & 54k_{\bar{y}} & 13k_{\bar{y}}L & 0 & 156k_{\bar{y}} & -22k_{\bar{y}}L \\ 0 & -13k_{\bar{y}}L & -3k_{\bar{y}}L^2 & 0 & -22k_{\bar{y}}L & 4k_{\bar{y}}L^2 \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} n_{x\bar{x}} & n_{y\bar{x}} & 0 & 0 & 0 & 0 \\ n_{x\bar{y}} & n_{y\bar{y}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} & 0 \\ 0 & 0 & 0 & n_{x\bar{y}} & n_{y\bar{y}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

where the axial stiffness D_{EA}, the bending stiffness D_{EI} and the length L are given by

$$D_{EA} = EA; \quad D_{EI} = EI; \quad L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The transformation matrix **G** contains the direction cosines

$$n_{x\bar{x}} = n_{y\bar{y}} = \frac{x_2 - x_1}{L} \qquad n_{y\bar{x}} = -n_{x\bar{y}} = \frac{y_2 - y_1}{L}$$

The element loads $\mathbf{f}_l^e$ stored in the variable `fe` are computed according to

$$\mathbf{f}_l^e = \mathbf{G}^T \bar{\mathbf{f}}_l^e$$

where

$$\bar{\mathbf{f}}_l^e = \begin{bmatrix} \frac{q_{\bar{x}} L}{2} \\ \frac{q_{\bar{y}} L}{2} \\ \frac{q_{\bar{y}} L^2}{12} \\ \frac{q_{\bar{x}} L}{2} \\ \frac{q_{\bar{y}} L}{2} \\ -\frac{q_{\bar{y}} L^2}{12} \end{bmatrix}$$

3.5.10 beam2ws

Purpose

Compute section forces in a two dimensional beam element with elastic support.

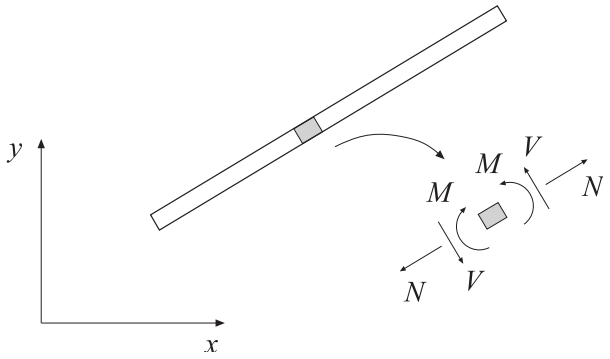
Syntax

```
es = cfc.beam2ws(ex, ey, ep, ed)
es = cfc.beam2ws(ex, ey, ep, ed, eq)
es, edi, eci = cfc.beam2ws(ex, ey, ep, ed, eq, n)
```

Description

`beam2ws` computes the section forces and displacements in local directions along the beam element `beam2we`.

The input variables `ex`, `ey`, `ep` and `eq` are defined in `beam2we`, and the element displacements, stored in `ed`, are obtained by the function `extract_ed`. If distributed loads are applied to the element, the variable `eq` must be included.



The number of evaluation points for section forces and displacements are determined by `n`. If `n` is omitted, only the ends of the beam are evaluated.

The output variables

$$\texttt{es}= \begin{bmatrix} N(0) & V(0) & M(0) \\ N(\bar{x}_2) & V(\bar{x}_2) & M(\bar{x}_2) \\ \vdots & \vdots & \vdots \\ N(\bar{x}_{n-1}) & V(\bar{x}_{n-1}) & M(\bar{x}_{n-1}) \\ N(L) & V(L) & M(L) \end{bmatrix} \qquad \texttt{edi}= \begin{bmatrix} u(0) & v(0) \\ u(\bar{x}_2) & v(\bar{x}_2) \\ \vdots & \vdots \\ u(\bar{x}_{n-1}) & v(\bar{x}_{n-1}) \\ u(L) & v(L) \end{bmatrix}$$

$$\texttt{eci}= \begin{bmatrix} 0 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_{n-1} \\ L \end{bmatrix}$$

contain the section forces, the displacements, and the evaluation points on the local $\bar{x}$-axis. L is the length of the beam element.

Theory

The nodal displacements in local coordinates are given by

$$\bar{\mathbf{a}}^e = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \\ \bar{u}_3 \\ \bar{u}_4 \\ \bar{u}_5 \\ \bar{u}_6 \end{bmatrix} = \mathbf{G}\mathbf{a}^e$$

where **G** is described in `beam2we` and the transpose of $\mathbf{a}^e$ is stored in `ed`. The displacements associated with bar action and beam action are determined as

$$\bar{\mathbf{a}}^e_{\text{bar}} = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_4 \end{bmatrix} \qquad \bar{\mathbf{a}}^e_{\text{beam}} = \begin{bmatrix} \bar{u}_2 \\ \bar{u}_3 \\ \bar{u}_5 \\ \bar{u}_6 \end{bmatrix}$$

The displacement $u(\bar{x})$ and the normal force $N(\bar{x})$ are computed from

$$u(\bar{x}) = \mathbf{N}_{\text{bar}}\bar{\mathbf{a}}^e_{\text{bar}} + u_p(\bar{x})$$

$$N(\bar{x}) = D_{EA}\mathbf{B}_{\text{bar}}\bar{\mathbf{a}}^e + N_p(\bar{x})$$

where

$$\mathbf{N}_{\text{bar}} = \begin{bmatrix} 1 & \bar{x} \end{bmatrix} \mathbf{C}^{-1}_{\text{bar}} = \begin{bmatrix} 1 - \frac{\bar{x}}{L} & \frac{\bar{x}}{L} \end{bmatrix}$$

$$\mathbf{B}_{\text{bar}} = \begin{bmatrix} 0 & 1 \end{bmatrix} \mathbf{C}^{-1}_{\text{bar}} = \begin{bmatrix} -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

$$u_p(\bar{x}) = \frac{k_{\bar{x}}}{D_{EA}} \begin{bmatrix} \frac{\bar{x}^2 - L\bar{x}}{2} & \frac{\bar{x}^3 - L^2\bar{x}}{6} \end{bmatrix} \mathbf{C}^{-1}_{\text{bar}}\bar{\mathbf{a}}^e_{\text{bar}} - \frac{q_{\bar{x}}}{D_{EA}} \left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2} \right)$$

$$N_p(\bar{x}) = k_{\bar{x}} \begin{bmatrix} \frac{2\bar{x} - L}{2} & \frac{3\bar{x}^2 - L^2}{6} \end{bmatrix} \mathbf{C}^{-1}_{\text{bar}}\bar{\mathbf{a}}^e_{\text{bar}} - q_{\bar{x}} \left(\bar{x} - \frac{L}{2} \right)$$

in which D_{EA}, $k_{\bar{x}}$, L, and $q_{\bar{x}}$ are defined in `beam2we` and

$$\mathbf{C}^{-1}_{\text{bar}} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

The displacement $v(\bar{x})$, the bending moment $M(\bar{x})$ and the shear force $V(\bar{x})$ are computed from

$$v(\bar{x}) = \mathbf{N}_{\text{beam}}\bar{\mathbf{a}}^e_{\text{beam}} + v_p(\bar{x})$$

$$M(\bar{x}) = D_{EI}\mathbf{B}_{\text{beam}}\bar{\mathbf{a}}^e_{\text{beam}} + M_p(\bar{x})$$

$$V(\bar{x}) = -D_{EI}\frac{d\mathbf{B}_{\text{beam}}}{dx}\bar{\mathbf{a}}^e_{\text{beam}} + V_p(\bar{x})$$

where

$$\mathbf{N}_{\text{beam}} = \begin{bmatrix} 1 & \bar{x} & \bar{x}^2 & \bar{x}^3 \end{bmatrix} \mathbf{C}^{-1}_{\text{beam}}$$

$$\mathbf{B}_{\text{beam}} = \begin{bmatrix} 0 & 0 & 2 & 6\bar{x} \end{bmatrix} \mathbf{C}^{-1}_{\text{beam}}$$

$$\frac{d\mathbf{B}_{\text{beam}}}{dx} = \begin{bmatrix} 0 & 0 & 0 & 6 \end{bmatrix} \mathbf{C}^{-1}_{\text{beam}}$$

$$v_p(\bar{x}) = -\frac{k_{\bar{y}}}{D_{EI}} \begin{bmatrix} \frac{\bar{x}^4 - 2L\bar{x}^3 + L^2\bar{x}^2}{24} \\ \frac{\bar{x}^5 - 3L^2\bar{x}^3 + 2L^3\bar{x}^2}{120} \\ \frac{\bar{x}^6 - 4L^3\bar{x}^3 + 3L^4\bar{x}^2}{360} \\ \frac{\bar{x}^7 - 5L^4\bar{x}^3 + 4L^5\bar{x}^2}{840} \end{bmatrix}^T \mathbf{C}^{-1}_{\text{beam}}\bar{\mathbf{a}}^e_{\text{beam}} + \frac{q_{\bar{y}}}{D_{EI}} \left(\frac{\bar{x}^4}{24} - \frac{L\bar{x}^3}{12} + \frac{L^2\bar{x}^2}{24} \right)$$

$$M_p(\bar{x}) = -k_{\bar{y}} \begin{bmatrix} \frac{6\bar{x}^2 - 6L\bar{x} + L^2}{12} \\ \frac{10\bar{x}^3 - 9L^2\bar{x} + 2L^3}{60} \\ \frac{5\bar{x}^4 - 4L^3\bar{x} + L^4}{60} \\ \frac{21\bar{x}^5 - 15L^4\bar{x} + 4L^5}{420} \end{bmatrix}^T \mathbf{C}^{-1}_{\text{beam}}\bar{\mathbf{a}}^e_{\text{beam}} + q_{\bar{y}} \left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2} + \frac{L^2}{12} \right)$$

$$V_p(\bar{x}) = k_{\bar{y}} \begin{bmatrix} \frac{2\bar{x}-L}{2} \\ \frac{10\bar{x}^2-3L^2}{20} \\ \frac{5\bar{x}^3-L^3}{15} \\ \frac{7\bar{x}^4-L^4}{28} \end{bmatrix}^T \mathbf{C}_{\text{beam}}^{-1} \bar{\mathbf{a}}_{\text{beam}}^e - q_{\bar{y}} \left(\bar{x} - \frac{L}{2} \right)$$

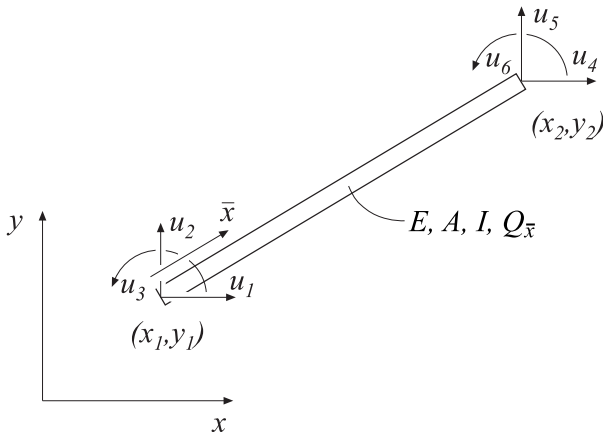
in which D_{EI}, $k_{\bar{y}}$, L, and $q_{\bar{y}}$ are defined in `beam2we` and

$$\mathbf{C}_{\text{beam}}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{3}{L^2} & -\frac{2}{L} & \frac{3}{L^2} & -\frac{1}{L} \\ \frac{2}{L^3} & \frac{1}{L^2} & -\frac{2}{L^3} & \frac{1}{L^2} \end{bmatrix}$$

3.5.11 beam2ge

Purpose

Compute element stiffness matrix for a two dimensional nonlinear beam element with respect to geometrical nonlinearity.



Syntax

```
Ke = cfc.beam2ge(ex, ey, ep, Qx)
Ke, fe = cfc.beam2ge(ex, ey, ep, Qx, eq)
```

Description

`beam2ge` provides the global element stiffness matrix $\mathbf{K}^e$ for a two dimensional beam element with respect to geometrical nonlinearity.

The input variables:

`ex`= $[x_1 \;\; x_2]$ `ey`= $[y_1 \;\; y_2]$ `ep`= $[E \; A \; I]$

supply the element nodal coordinates x_1, y_1, x_2, and y_2, the modulus of elasticity E, the cross-section area A, and the moment of inertia I.

The input variable

`Qx`= $[Q_{\bar{x}}]$

contains the value of the predefined axial force $Q_{\bar{x}}$, which is positive in tension.

The element load vector $\mathbf{f}_l^e$ can also be computed if a uniformly distributed transverse load is applied to the element. The optional input

`eq`= $[q_{\bar{y}}]$

contains the distributed transverse load per unit length, $q_{\bar{y}}$. Note that `eq` is a scalar and not a vector as in `beam2e`.

Theory

The element stiffness matrix $\mathbf{K}^e$, stored in the variable `Ke`, is computed according to

$$\mathbf{K}^e = \mathbf{G}^T \bar{\mathbf{K}}^e \mathbf{G}$$

where $\bar{\mathbf{K}}^e$ is given by

$$\bar{\mathbf{K}}^e = \bar{\mathbf{K}}_0^e + \bar{\mathbf{K}}_\sigma^e$$

with

$$\bar{\mathbf{K}}_0^e = \begin{bmatrix} \frac{D_{EA}}{L} & 0 & 0 & -\frac{D_{EA}}{L} & 0 & 0 \\ 0 & \frac{12D_{EI}}{L^3} & \frac{6D_{EI}}{L^2} & 0 & -\frac{12D_{EI}}{L^3} & \frac{6D_{EI}}{L^2} \\ 0 & \frac{6D_{EI}}{L^2} & \frac{4D_{EI}}{L} & 0 & -\frac{6D_{EI}}{L^2} & \frac{2D_{EI}}{L} \\ -\frac{D_{EA}}{L} & 0 & 0 & \frac{D_{EA}}{L} & 0 & 0 \\ 0 & -\frac{12D_{EI}}{L^3} & -\frac{6D_{EI}}{L^2} & 0 & \frac{12D_{EI}}{L^3} & -\frac{6D_{EI}}{L^2} \\ 0 & \frac{6D_{EI}}{L^2} & \frac{2D_{EI}}{L} & 0 & -\frac{6D_{EI}}{L^2} & \frac{4D_{EI}}{L} \end{bmatrix}$$

$$\bar{\mathbf{K}}_\sigma^e = Q_{\bar{x}} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{6}{5L} & \frac{1}{10} & 0 & -\frac{6}{5L} & \frac{1}{10} \\ 0 & \frac{1}{10} & \frac{2L}{15} & 0 & -\frac{1}{10} & -\frac{L}{30} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{6}{5L} & -\frac{1}{10} & 0 & \frac{6}{5L} & -\frac{1}{10} \\ 0 & \frac{1}{10} & -\frac{L}{30} & 0 & -\frac{1}{10} & \frac{2L}{15} \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} n_{x\bar{x}} & n_{y\bar{x}} & 0 & 0 & 0 & 0 \\ n_{x\bar{y}} & n_{y\bar{y}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} & 0 \\ 0 & 0 & 0 & n_{x\bar{y}} & n_{y\bar{y}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

where the axial stiffness D_{EA}, the bending stiffness D_{EI} and the length L are given by

$$D_{EA} = EA; \quad D_{EI} = EI; \quad L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The transformation matrix **G** contains the direction cosines

$$n_{x\bar{x}} = n_{y\bar{y}} = \frac{x_2 - x_1}{L} \qquad n_{y\bar{x}} = -n_{x\bar{y}} = \frac{y_2 - y_1}{L}$$

The element loads $\mathbf{f}_l^e$ stored in `fe` are computed according to

$$\mathbf{f}_l^e = \mathbf{G}^T \bar{\mathbf{f}}_l^e$$

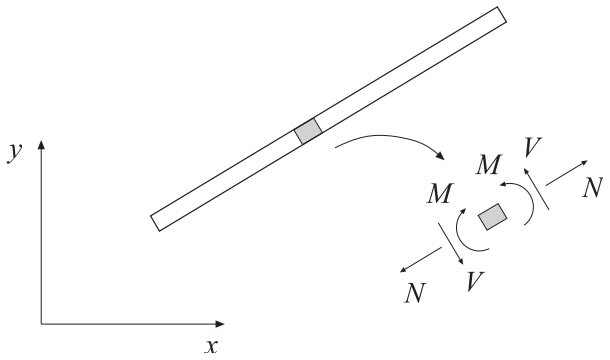
where

$$\bar{\mathbf{f}}_l^e = q_{\bar{y}} \begin{bmatrix} 0 \\ \frac{L}{2} \\ \frac{L^2}{12} \\ 0 \\ \frac{L}{2} \\ -\frac{L^2}{12} \end{bmatrix}$$

3.5.12 beam2gs

Purpose

Compute section forces in a two dimensional nonlinear beam element with geometrical nonlinearity.



Syntax

```
es, Qx = cfc.beam2gs(ex, ey, ep, ed, Qx)
es, Qx = cfc.beam2gs(ex, ey, ep, ed, Qx, eq)
es, Qx, edi = cfc.beam2gs(ex, ey, ep, ed, Qx, eq, n)
es, Qx, edi, eci = cfc.beam2gs(ex, ey, ep, ed, Qx, eq, n)
```

Description

`beam2gs` computes the section forces and displacements in local directions along the geometric nonlinear beam element `beam2ge`.

The input variables `ex`, `ey`, `ep`, `Qx` and `eq`, are described in `beam2ge`. The element displacements, stored in `ed`, are obtained by the function `extract_ed`. If a distributed transversal load is applied to the element, the variable `eq` must

be included. The number of evaluation points for section forces and displacements are determined by `n`. If `n` is omitted, only the ends of the beam are evaluated.

The output variable `Qx` contains $Q_{\bar{x}}$ and the output variables

$$\texttt{es}= \begin{bmatrix} N(0) & V(0) & M(0) \\ N(\bar{x}_2) & V(\bar{x}_2) & M(\bar{x}_2) \\ \vdots & \vdots & \vdots \\ N(\bar{x}_{n-1}) & V(\bar{x}_{n-1}) & M(\bar{x}_{n-1}) \\ N(L) & V(L) & M(L) \end{bmatrix} \qquad \texttt{edi}= \begin{bmatrix} u(0) & v(0) \\ u(\bar{x}_2) & v(\bar{x}_2) \\ \vdots & \vdots \\ u(\bar{x}_{n-1}) & v(\bar{x}_{n-1}) \\ u(L) & v(L) \end{bmatrix}$$

$$\texttt{eci}= \begin{bmatrix} 0 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_{n-1} \\ L \end{bmatrix}$$

contain the section forces, the displacements, and the evaluation points on the local $\bar{x}$-axis. L is the length of the beam element.

Theory

The nodal displacements in local coordinates are given by

$$\bar{\mathbf{a}}^e = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \\ \bar{u}_3 \\ \bar{u}_4 \\ \bar{u}_5 \\ \bar{u}_6 \end{bmatrix} = \mathbf{G}\mathbf{a}^e$$

where **G** is described in `beam2ge` and the transpose of $\mathbf{a}^e$ is stored in `ed`.

The displacements associated with bar action and beam action are determined as

$$\bar{\mathbf{a}}^e_{\text{bar}} = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_4 \end{bmatrix}; \qquad \bar{\mathbf{a}}^e_{\text{beam}} = \begin{bmatrix} \bar{u}_2 \\ \bar{u}_3 \\ \bar{u}_5 \\ \bar{u}_6 \end{bmatrix}$$

The displacement $u(\bar{x})$ is computed from

$$u(\bar{x}) = \mathbf{N}_{\text{bar}}\bar{\mathbf{a}}^e_{\text{bar}}$$

where

$$\mathbf{N}_{\text{bar}} = \begin{bmatrix} 1 & \bar{x} \end{bmatrix} \mathbf{C}^{-1}_{\text{bar}} = \begin{bmatrix} 1 - \frac{\bar{x}}{L} & \frac{\bar{x}}{L} \end{bmatrix}$$

where L is defined in `beam2ge` and

$$\mathbf{C}^{-1}_{\text{bar}} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

The displacement $v(\bar{x})$, the rotation $\theta(\bar{x})$, the bending moment $M(\bar{x})$ and the shear force $V(\bar{x})$ are computed from

$$v(\bar{x}) = \mathbf{N}_{\text{beam}} \mathbf{a}^e_{\text{beam}} + v_p(\bar{x})$$

$$\theta(\bar{x}) = \frac{d\mathbf{N}_{\text{beam}}}{dx} \mathbf{a}^e_{\text{beam}} + \theta_p(\bar{x})$$

$$M(\bar{x}) = D_{EI} \mathbf{B}_{\text{beam}} \mathbf{a}^e_{\text{beam}} + M_p(\bar{x})$$

$$V(\bar{x}) = -D_{EI} \frac{d\mathbf{B}_{\text{beam}}}{dx} \mathbf{a}^e_{\text{beam}} + V_p(\bar{x})$$

where

$$\mathbf{N}_{\text{beam}} = \begin{bmatrix} 1 & \bar{x} & \bar{x}^2 & \bar{x}^3 \end{bmatrix} \mathbf{C}^{-1}_{\text{beam}}$$

$$\frac{d\mathbf{N}_{\text{beam}}}{dx} = \begin{bmatrix} 0 & 1 & 2\bar{x} & 3\bar{x}^2 \end{bmatrix} \mathbf{C}^{-1}_{\text{beam}}$$

$$\mathbf{B}_{\text{beam}} = \begin{bmatrix} 0 & 0 & 2 & 6\bar{x} \end{bmatrix} \mathbf{C}^{-1}_{\text{beam}}$$

$$\frac{d\mathbf{B}_{\text{beam}}}{dx} = \begin{bmatrix} 0 & 0 & 0 & 6 \end{bmatrix} \mathbf{C}^{-1}_{\text{beam}}$$

$$v_p(\bar{x}) = -\frac{Q_{\bar{x}}}{D_{EI}} \begin{bmatrix} 0 \\ 0 \\ \frac{\bar{x}^4}{12} - \frac{L\bar{x}^3}{6} + \frac{L^2\bar{x}^2}{12} \\ \frac{\bar{x}^5}{20} - \frac{3L^2\bar{x}^3}{20} + \frac{L^3\bar{x}^2}{10} \end{bmatrix}^T \mathbf{C}^{-1}_{\text{beam}} \mathbf{a}^e_{\text{beam}} + \frac{q_{\bar{y}}}{D_{EI}} \left(\frac{\bar{x}^4}{24} - \frac{L\bar{x}^3}{12} + \frac{L^2\bar{x}^2}{24} \right)$$

$$\theta_p(\bar{x}) = -\frac{Q_{\bar{x}}}{D_{EI}} \begin{bmatrix} 0 \\ 0 \\ \frac{\bar{x}^3}{3} - \frac{L\bar{x}^2}{2} + \frac{L^2\bar{x}}{6} \\ \frac{\bar{x}^4}{4} - \frac{9L^2\bar{x}^2}{20} + \frac{L^3\bar{x}}{5} \end{bmatrix}^T \mathbf{C}^{-1}_{\text{beam}} \mathbf{a}^e_{\text{beam}} + \frac{q_{\bar{y}}}{D_{EI}} \left(\frac{\bar{x}^3}{6} - \frac{L\bar{x}^2}{4} + \frac{L^2\bar{x}}{12} \right)$$

$$M_p(\bar{x}) = -Q_{\bar{x}} \begin{bmatrix} 0 \\ 0 \\ \bar{x}^2 - L\bar{x} + \frac{L^2}{6} \\ \bar{x}^3 - \frac{9L^2\bar{x}}{10} + \frac{L^3}{5} \end{bmatrix}^T \mathbf{C}^{-1}_{\text{beam}} \mathbf{a}^e_{\text{beam}} + q_{\bar{y}} \left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2} + \frac{L^2}{12} \right)$$

$$V_p(\bar{x}) = Q_{\bar{x}} \begin{bmatrix} 0 \\ 0 \\ 2\bar{x} - L \\ 3\bar{x}^2 - \frac{9L^2}{10} \end{bmatrix}^T \mathbf{C}^{-1}_{\text{beam}} \mathbf{a}^e_{\text{beam}} - q_{\bar{y}} \left(\bar{x} - \frac{L}{2} \right)$$

in which D_{EI}, L, and $q_{\bar{y}}$ are defined in `beam2ge` and

$$\mathbf{C}^{-1}_{\text{beam}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{3}{L^2} & -\frac{2}{L} & \frac{3}{L^2} & -\frac{1}{L} \\ \frac{2}{L^3} & \frac{1}{L^2} & -\frac{2}{L^3} & \frac{1}{L^2} \end{bmatrix}$$

An updated value of the axial force is computed as

$$Q_{\bar{x}} = D_{EA} \begin{bmatrix} 0 & 1 \end{bmatrix} \mathbf{C}^{-1}_{\text{bar}} \mathbf{a}^e_{\text{bar}}$$

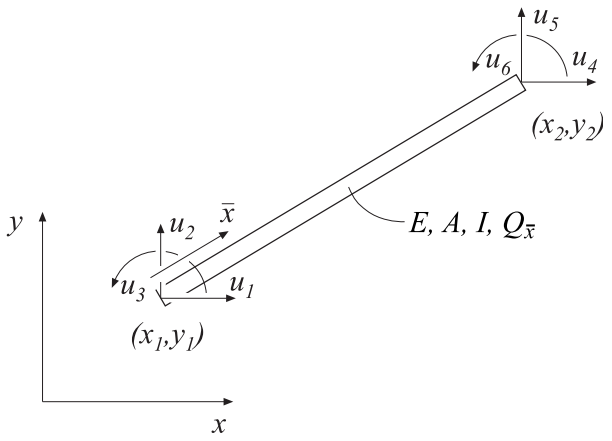
The normal force $N(\bar{x})$ is then computed as

$$N(\bar{x}) = Q_{\bar{x}} + \theta(\bar{x}) V(\bar{x})$$

3.5.13 beam2gxe

Purpose

Compute element stiffness matrix for a two dimensional nonlinear beam element with exact solution.



Syntax

```
Ke = cfc.beam2gxe(ex, ey, ep, Qx)
Ke, fe = cfc.beam2gxe(ex, ey, ep, Qx, eq)
```

Description

`beam2gxe` provides the global element stiffness matrix $\mathbf{K}^e$ for a two dimensional beam element with respect to geometrical nonlinearity considering exact solution.

The input variables:

`ex`$= [x_1 \;\; x_2]$ `ey`$= [y_1 \;\; y_2]$ `ep`$= [E \; A \; I]$

supply the element nodal coordinates x_1, y_1, x_2, and y_2, the modulus of elasticity E, the cross section area A, and the moment of inertia I.

The input variable

`Qx`$= [Q_{\bar{x}}]$

contains the value of the predefined axial force $Q_{\bar{x}}$, which is positive in tension.

The element load vector `fe` can also be computed if a uniformly distributed transverse load is applied to the element. The optional input variable

`eq`$= [q_{\bar{y}}]$

then contains the distributed transverse load per unit length, $q_{\bar{y}}$. Note that `eq` is a scalar and not a vector as in `beam2e`.

Theory

The element stiffness matrix $\mathbf{K}^e$, stored in the variable `Ke`, is computed according to

$$\mathbf{K}^e = \mathbf{G}^T \bar{\mathbf{K}}^e \mathbf{G}$$

with

$$\bar{\mathbf{K}}^e = \begin{bmatrix} \frac{D_{EA}}{L} & 0 & 0 & -\frac{D_{EA}}{L} & 0 & 0 \\ 0 & \frac{12D_{EI}}{L^3}\phi_5 & \frac{6D_{EI}}{L^2}\phi_2 & 0 & -\frac{12D_{EI}}{L^3}\phi_5 & \frac{6D_{EI}}{L^2}\phi_2 \\ 0 & \frac{6D_{EI}}{L^2}\phi_2 & \frac{4D_{EI}}{L}\phi_3 & 0 & -\frac{6D_{EI}}{L^2}\phi_2 & \frac{2D_{EI}}{L}\phi_4 \\ -\frac{D_{EA}}{L} & 0 & 0 & \frac{D_{EA}}{L} & 0 & 0 \\ 0 & -\frac{12D_{EI}}{L^3}\phi_5 & -\frac{6D_{EI}}{L^2}\phi_2 & 0 & \frac{12D_{EI}}{L^3}\phi_5 & -\frac{6D_{EI}}{L^2}\phi_2 \\ 0 & \frac{6D_{EI}}{L^2}\phi_2 & \frac{2D_{EI}}{L}\phi_4 & 0 & -\frac{6D_{EI}}{L^2}\phi_2 & \frac{4D_{EI}}{L}\phi_3 \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} n_{x\bar{x}} & n_{y\bar{x}} & 0 & 0 & 0 & 0 \\ n_{x\bar{y}} & n_{y\bar{y}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} & 0 \\ 0 & 0 & 0 & n_{x\bar{y}} & n_{y\bar{y}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

where the axial stiffness D_{EA}, the bending stiffness D_{EI} and the length L are given by

$$D_{EA} = EA; \quad D_{EI} = EI; \quad L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The transformation matrix $\mathbf{G}$ contains the direction cosines

$$n_{x\bar{x}} = n_{y\bar{y}} = \frac{x_2 - x_1}{L} \qquad n_{y\bar{x}} = -n_{x\bar{y}} = \frac{y_2 - y_1}{L}$$

For axial compression ($Q_{\bar{x}} < 0$):

$$\phi_2 = \frac{1}{12}\frac{k^2L^2}{1-\phi_1} \qquad \phi_3 = \frac{1}{4}\phi_1 + \frac{3}{4}\phi_2$$

$$\phi_4 = -\frac{1}{2}\phi_1 + \frac{3}{2}\phi_2 \qquad \phi_5 = \phi_1\phi_2$$

with

$$k = \sqrt{\frac{-Q_{\bar{x}}}{D_{EI}}} \qquad \phi_1 = \frac{kL}{2}\cot\frac{kL}{2}$$

For axial tension ($Q_{\bar{x}} > 0$):

$$\phi_2 = -\frac{1}{12}\frac{k^2L^2}{1-\phi_1} \qquad \phi_3 = \frac{1}{4}\phi_1 + \frac{3}{4}\phi_2$$

$$\phi_4 = -\frac{1}{2}\phi_1 + \frac{3}{2}\phi_2 \qquad \phi_5 = \phi_1\phi_2$$

with

$$k = \sqrt{\frac{Q_{\bar{x}}}{D_{EI}}} \qquad \phi_1 = \frac{kL}{2}\coth\frac{kL}{2}$$

The element loads $\mathbf{f}_l^e$ stored in the variable `fe` are computed according to

$$\mathbf{f}_l^e = \mathbf{G}^T \bar{\mathbf{f}}_l^e$$

where

$$\bar{\mathbf{f}}_l^e = qL \begin{bmatrix} 0 \\ \frac{1}{2} \\ \frac{L}{12}\psi \\ 0 \\ \frac{1}{2} \\ -\frac{L}{12}\psi \end{bmatrix}$$

For an axial compressive force ($Q_{\bar{x}} < 0$):

$$\psi = 6\left(\frac{2}{(kL)^2} - \frac{1+\cos kL}{kL \sin kL}\right)$$

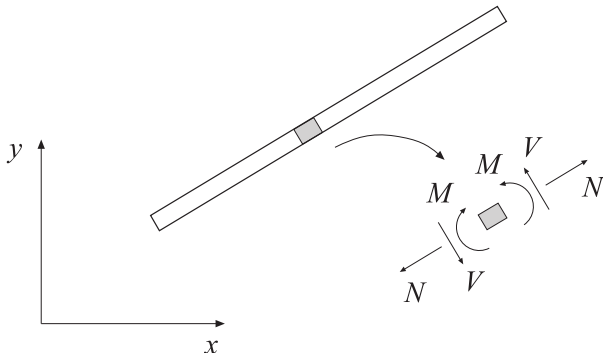
and for an axial tensile force ($Q_{\bar{x}} > 0$):

$$\psi = -6\left(\frac{2}{(kL)^2} - \frac{1+\cosh kL}{kL \sinh kL}\right)$$

3.5.14 beam2gxs

Purpose

Compute section forces in a two dimensional geometric nonlinear beam element with exact solution.



Syntax

```
es,Qx = cfc.beam2gxs(ex, ey, ep, ed, Qx)
es,Qx = cfc.beam2gxs(ex, ey, ep, ed, Qx, eq)
es,Qx,edi = cfc.beam2gxs(ex, ey, ep, ed, Qx, eq, n)
es,Qx,edi,eci = cfc.beam2gxs(ex, ey, ep, ed, Qx, eq, n)
```

Description

`beam2gxs` computes the section forces and displacements in local directions along the geometric nonlinear beam element `beam2gxe`.

The input variables `ex`, `ey`, `ep`, `Qx` and `eq`, are described in `beam2gxe`. The element displacements, stored in `ed`, are obtained by the function `extract_ed`. If a distributed transversal load is applied to the element, the variable `eq` must be included. The number of evaluation points for section forces and displacements are determined by `n`. If `n` is omitted, only the ends of the beam are evaluated.

The output variable `Qx` contains $Q_{\bar{x}}$ and the output variables

$$\texttt{es}= \begin{bmatrix} N(0) & V(0) & M(0) \\ N(\bar{x}_2) & V(\bar{x}_2) & M(\bar{x}_2) \\ \vdots & \vdots & \vdots \\ N(\bar{x}_{n-1}) & V(\bar{x}_{n-1}) & M(\bar{x}_{n-1}) \\ N(L) & V(L) & M(L) \end{bmatrix} \qquad \texttt{edi}= \begin{bmatrix} u(0) & v(0) \\ u(\bar{x}_2) & v(\bar{x}_2) \\ \vdots & \vdots \\ u(\bar{x}_{n-1}) & v(\bar{x}_{n-1}) \\ u(L) & v(L) \end{bmatrix}$$

$$\texttt{eci}= \begin{bmatrix} 0 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_{n-1} \\ L \end{bmatrix}$$

contain the section forces, the displacements, and the evaluation points on the local $\bar{x}$-axis. L is the length of the beam element.

Theory

The nodal displacements in local coordinates are given by

$$\bar{\mathbf{a}}^e = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \\ \bar{u}_3 \\ \bar{u}_4 \\ \bar{u}_5 \\ \bar{u}_6 \end{bmatrix} = \mathbf{G}\mathbf{a}^e$$

where $\mathbf{G}$ is described in `beam2ge` and the transpose of $\mathbf{a}^e$ is stored in `ed`. The displacements associated with bar action and beam action are determined as

$$\bar{\mathbf{a}}^e_{\text{bar}} = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_4 \end{bmatrix}; \quad \bar{\mathbf{a}}^e_{\text{beam}} = \begin{bmatrix} \bar{u}_2 \\ \bar{u}_3 \\ \bar{u}_5 \\ \bar{u}_6 \end{bmatrix}$$

The displacement $u(\bar{x})$ is computed from

$$u(\bar{x}) = \mathbf{N}_{\text{bar}} \bar{\mathbf{a}}^e_{\text{bar}}$$

where

$$\mathbf{N}_{\text{bar}} = \begin{bmatrix} 1 & \bar{x} \end{bmatrix} \mathbf{C}_{\text{bar}}^{-1} = \begin{bmatrix} 1 - \frac{\bar{x}}{L} & \frac{\bar{x}}{L} \end{bmatrix}$$

where L is defined in `beam2gxe` and

$$\mathbf{C}_{\text{bar}}^{-1} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

The displacement $v(\bar{x})$, the rotation $\theta(\bar{x})$, the bending moment $M(\bar{x})$ and the shear force $V(\bar{x})$ are computed from

$$v(\bar{x}) = \mathbf{N}_{\text{beam}} \mathbf{a}_{\text{beam}}^{e} + v_p(\bar{x})$$

$$\theta(\bar{x}) = \frac{d\mathbf{N}_{\text{beam}}}{dx} \mathbf{a}_{\text{beam}}^{e} + \theta_p(\bar{x})$$

$$M(\bar{x}) = D_{EI} \mathbf{B}_{\text{beam}} \mathbf{a}_{\text{beam}}^{e} + M_p(\bar{x})$$

$$V(\bar{x}) = -D_{EI} \frac{d\mathbf{B}_{\text{beam}}}{dx} \mathbf{a}_{\text{beam}}^{e} + V_p(\bar{x})$$

For an axial compressive force ($Q_{\bar{x}} < 0$) we have

$$\mathbf{N}_{\text{beam}} = \begin{bmatrix} 1 & \bar{x} & \cos k\bar{x} & \sin k\bar{x} \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$\frac{d\mathbf{N}_{\text{beam}}}{dx} = \begin{bmatrix} 0 & 1 & -k \sin k\bar{x} & k \cos k\bar{x} \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$\mathbf{B}_{\text{beam}} = \begin{bmatrix} 0 & 0 & -k^2 \cos k\bar{x} & -k^2 \sin k\bar{x} \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$\frac{d\mathbf{B}_{\text{beam}}}{dx} = \begin{bmatrix} 0 & 0 & k^3 \sin k\bar{x} & -k^3 \cos k\bar{x} \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$v_p(\bar{x}) = \frac{q_{\bar{y}} L^4}{2D_{EI}} \left[\frac{1 + \cos kL}{(kL)^3 \sin kL} (-1 + \cos k\bar{x}) - \frac{1}{(kL)^3} \sin k\bar{x} + \frac{1}{(kL)^2} \left(\frac{\bar{x}^2}{L^2} - \frac{\bar{x}}{L} \right) \right]$$

$$\theta_p(\bar{x}) = \frac{q_{\bar{y}} L^3}{2D_{EI}} \left[-\frac{1 + \cos kL}{(kL)^2 \sin kL} \sin k\bar{x} - \frac{1}{(kL)^2} \cos k\bar{x} + \frac{1}{(kL)^2} \left(\frac{2\bar{x}}{L} - 1 \right) \right]$$

$$M_p(\bar{x}) = \frac{q_{\bar{y}} L^2}{2} \left[-\frac{1 + \cos kL}{(kL) \sin kL} \cos k\bar{x} + \frac{1}{(kL)} \sin k\bar{x} + \frac{2}{(kL)^2} \right]$$

$$V_p(\bar{x}) = Q_{\bar{x}} \begin{bmatrix} 0 \\ 0 \\ 2\bar{x} - L \\ 3\bar{x}^2 - \frac{9L^2}{10} \end{bmatrix}^T \mathbf{C}_{\text{beam}}^{-1} \mathbf{a}_{\text{beam}}^{e} - q_{\bar{y}} \left(\bar{x} - \frac{L}{2} \right)$$

in which D_{EI}, L, and $q_{\bar{y}}$ are defined in `beam2gxe` and

$$\mathbf{C}_{\text{beam}}^{-1} = \begin{bmatrix} k(kL \sin kL + \cos kL - 1) & -kL \cos kL + \sin kL & -k(1 - \cos kL) & -\sin kL + kL \\ -k^2 \sin kL & -k(1 - \cos kL) & k^2 \sin kL & -k(1 - \cos kL) \\ -k(1 - \cos kL) & kL \cos kL - \sin kL & k(1 - \cos kL) & \sin kL - kL \\ k \sin kL & kL \sin kL + \cos kL - 1 & -k \sin kL & 1 - \cos kL \end{bmatrix}$$

An updated value of the axial force is computed as

$$Q_{\bar{x}} = D_{EA} \begin{bmatrix} 0 & 1 \end{bmatrix} \mathbf{C}_{\text{bar}}^{-1} \mathbf{a}_{\text{bar}}^{e}$$

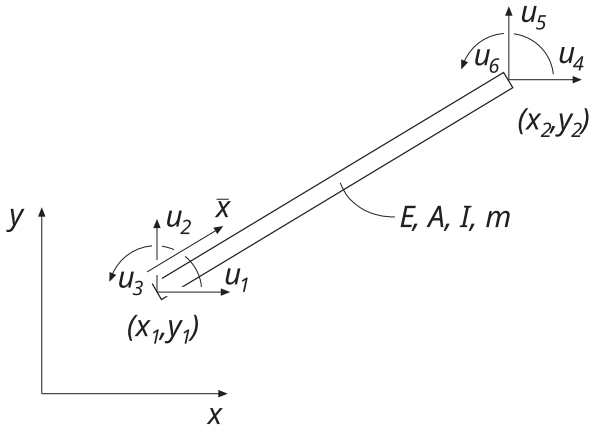
The normal force $N(\bar{x})$ is then computed as

$$N(\bar{x}) = Q_{\bar{x}} + \theta(\bar{x}) V(\bar{x})$$

3.5.15 beam2de

Purpose

Compute element stiffness, mass and damping matrices for a two dimensional beam element.



Syntax

```
Ke, Me = cfc.beam2de(ex, ey, ep)
Ke, Me, Ce = cfc.beam2de(ex, ey, ep)
```

Description

`beam2de` provides the global element stiffness matrix $\mathbf{K}^e$, the global element mass matrix $\mathbf{M}^e$, and the global element damping matrix $\mathbf{C}^e$, for a two dimensional beam element.

The input variables `ex` and `ey` are described in `beam2e`, and

`ep`$= [E,\ A,\ I,\ m,\ [a_0,\ a_1]]$

contains the modulus of elasticity E, the cross section area A, the moment of inertia I, the mass per unit length m, and the Rayleigh damping coefficients a_0 and a_1. If a_0 and a_1 are omitted, the element damping matrix `Ce` is not computed.

Theory

The element stiffness matrix $\mathbf{K}^e$, the element mass matrix $\mathbf{M}^e$ and the element damping matrix $\mathbf{C}^e$, stored in the variables `Ke`, `Me` and `Ce`, respectively, are computed according to

$$\mathbf{K}^e = \mathbf{G}^T\bar{\mathbf{K}}^e\mathbf{G} \qquad \mathbf{M}^e = \mathbf{G}^T\bar{\mathbf{M}}^e\mathbf{G} \qquad \mathbf{C}^e = \mathbf{G}^T\bar{\mathbf{C}}^e\mathbf{G}$$

where $\mathbf{G}$ and $\bar{\mathbf{K}}^e$ are described in `beam2e`.

The matrix $\bar{\mathbf{M}}^e$ is given by

$$\bar{\mathbf{M}}^e = \frac{mL}{420}\begin{bmatrix} 140 & 0 & 0 & 70 & 0 & 0 \\ 0 & 156 & 22L & 0 & 54 & -13L \\ 0 & 22L & 4L^2 & 0 & 13L & -3L^2 \\ 70 & 0 & 0 & 140 & 0 & 0 \\ 0 & 54 & 13L & 0 & 156 & -22L \\ 0 & -13L & -3L^2 & 0 & -22L & 4L^2 \end{bmatrix}$$

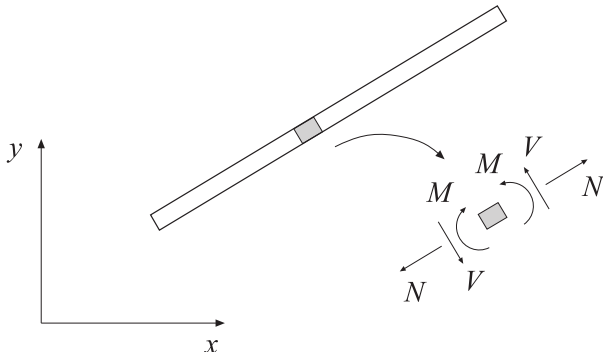
and the matrix $\bar{\mathbf{C}}^e$ is computed by combining $\bar{\mathbf{K}}^e$ and $\bar{\mathbf{M}}^e$:

$$\bar{\mathbf{C}}^e = a_0\bar{\mathbf{M}}^e + a_1\bar{\mathbf{K}}^e$$

3.5.16 beam2ds

Purpose

Compute section forces for a two dimensional beam element in dynamic analysis.



Syntax

```
es = cfc.beam2ds(ex, ey, ep, ed, ev, ea)
```

Description

`beam2ds` computes the section forces at the ends of the dynamic beam element `beam2de`.

The input variables `ex`, `ey` and `ep` are defined in `beam2de`. The element displacements, velocities, and accelerations, stored in `ed`, `ev`, and `ea` respectively, are obtained by the function `extract_ed`.

The output variable `es` contains the section forces at the ends of the beam:

$$es = \begin{bmatrix} N_1 & V_1 & M_1 \\ N_2 & V_2 & M_2 \end{bmatrix}$$

Theory

The section forces at the ends of the beam are obtained from the element force vector:

$$\bar{\mathbf{P}} = \begin{bmatrix} -N_1 & -V_1 & -M_1 & N_2 & V_2 & M_2 \end{bmatrix}^T$$

computed according to:

$$\bar{\mathbf{P}} = \bar{\mathbf{K}}^e \mathbf{G}\mathbf{a}^e + \bar{\mathbf{C}}^e \mathbf{G}\dot{\mathbf{a}}^e + \bar{\mathbf{M}}^e \mathbf{G}\ddot{\mathbf{a}}^e$$

The matrices $\bar{\mathbf{K}}^e$ and $\mathbf{G}$ are described in `beam2e`, and the matrices $\bar{\mathbf{M}}^e$ and $\bar{\mathbf{C}}^e$ are described in `beam2d`.

The nodal displacements:

$$\mathbf{a}^e = \begin{bmatrix} u_1 & u_2 & u_3 & u_4 & u_5 & u_6 \end{bmatrix}^T$$

shown in `beam2de` also define the directions of the nodal velocities:

$$\dot{\mathbf{a}}^e = \begin{bmatrix} \dot{u}_1 & \dot{u}_2 & \dot{u}_3 & \dot{u}_4 & \dot{u}_5 & \dot{u}_6 \end{bmatrix}^T$$

and the nodal accelerations:

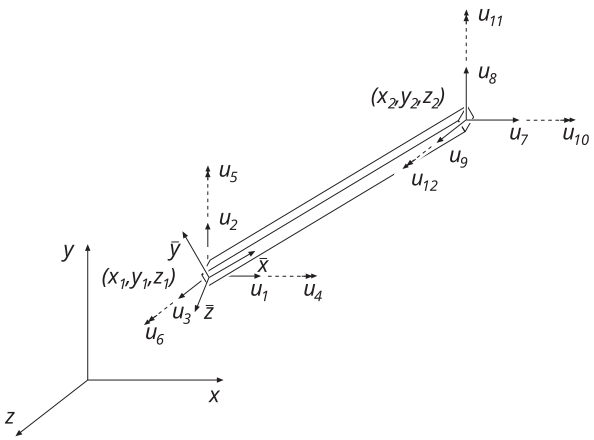
$$\ddot{\mathbf{a}}^e = \begin{bmatrix} \ddot{u}_1 & \ddot{u}_2 & \ddot{u}_3 & \ddot{u}_4 & \ddot{u}_5 & \ddot{u}_6 \end{bmatrix}^T$$

Note that the transposes of $\mathbf{a}^e$, $\dot{\mathbf{a}}^e$, and $\ddot{\mathbf{a}}^e$ are stored in `ed`, `ev`, and `ea` respectively.

3.5.17 beam3e

Purpose

Compute element stiffness matrix for a three dimensional beam element.



Syntax

```
Ke = cfc.beam3e(ex, ey, ez, eo, ep)
Ke, fe = cfc.beam3e(ex, ey, ez, eo, ep, eq)
```

Description

`beam3e` provides the global element stiffness matrix $\mathbf{K}^e$ for a three dimensional beam element.

The input variables

`ex`$= [x_1 \;\; x_2]$ `ey`$= [y_1 \;\; y_2]$ `ez`$= [z_1 \;\; z_2]$ `eo`$= [x_{\bar{z}} \;\; y_{\bar{z}} \;\; z_{\bar{z}}]$

supply the element nodal coordinates x_1, y_1, etc. as well as the direction of the local beam coordinate system $(\bar{x}, \bar{y}, \bar{z})$. By giving a global vector $(x_{\bar{z}}, y_{\bar{z}}, z_{\bar{z}})$ parallel with the positive local $\bar{z}$ axis of the beam, the local beam coordinate system is defined.

The variable

`ep`$= [E \;\; G \;\; A \;\; I_{\bar{y}} \;\; I_{\bar{z}} \;\; K_v]$

supplies the modulus of elasticity E, the shear modulus G, the cross section area A, the moment of inertia with respect to the $\bar{y}$ axis $I_{\bar{y}}$, the moment of inertia with respect to the $\bar{z}$ axis $I_{\bar{z}}$, and St. Venant torsion constant K_v.

The element load vector `fe` can also be computed if uniformly distributed loads are applied to the element. The optional input variable

`eq`$= [q_{\bar{x}} \;\; q_{\bar{y}} \;\; q_{\bar{z}} \;\; q_{\bar{\omega}}]$

then contains the distributed loads. The positive directions of $q_{\bar{x}}$, $q_{\bar{y}}$, and $q_{\bar{z}}$ follow the local beam coordinate system. The distributed torque $q_{\bar{\omega}}$ is positive if directed in the local $\bar{x}$-direction, i.e. from local $\bar{y}$ to local $\bar{z}$. All the loads are per unit length.

Theory

The element stiffness matrix $\mathbf{K}^e$ is computed according to

$$\mathbf{K}^e = \mathbf{G}^T \bar{\mathbf{K}}^e \mathbf{G}$$

where

$$\bar{\mathbf{K}}^e = \begin{bmatrix}
\frac{D_{EA}}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{D_{EA}}{L} & 0 \\
0 & \frac{12D_{EI_{\bar{z}}}}{L^3} & 0 & 0 & 0 & \frac{6D_{EI_{\bar{z}}}}{L^2} & 0 & -\frac{12D_{EI_{\bar{z}}}}{L^3} \\
0 & 0 & \frac{12D_{EI_{\bar{y}}}}{L^3} & 0 & -\frac{6D_{EI_{\bar{y}}}}{L^2} & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{D_{GK}}{L} & 0 & 0 & 0 & 0 \\
0 & 0 & -\frac{6D_{EI_{\bar{y}}}}{L^2} & 0 & \frac{4D_{EI_{\bar{y}}}}{L} & 0 & 0 & 0 \\
0 & \frac{6D_{EI_{\bar{z}}}}{L^2} & 0 & 0 & 0 & \frac{4D_{EI_{\bar{z}}}}{L} & 0 & -\frac{6D_{EI_{\bar{z}}}}{L^2} \\
-\frac{D_{EA}}{L} & 0 & 0 & 0 & 0 & 0 & \frac{D_{EA}}{L} & 0 \\
0 & -\frac{12D_{EI_{\bar{z}}}}{L^3} & 0 & 0 & 0 & -\frac{6D_{EI_{\bar{z}}}}{L^2} & 0 & \frac{12D_{EI_{\bar{z}}}}{L^3} \\
0 & 0 & -\frac{12D_{EI_{\bar{y}}}}{L^3} & 0 & \frac{6D_{EI_{\bar{y}}}}{L^2} & 0 & 0 & 0 \\
0 & 0 & 0 & -\frac{D_{GK}}{L} & 0 & 0 & 0 & 0 \\
0 & 0 & -\frac{6D_{EI_{\bar{y}}}}{L^2} & 0 & \frac{2D_{EI_{\bar{y}}}}{L} & 0 & 0 & 0 \\
0 & \frac{6D_{EI_{\bar{z}}}}{L^2} & 0 & 0 & 0 & \frac{2D_{EI_{\bar{z}}}}{L} & 0 & -\frac{6D_{EI_{\bar{z}}}}{L^2}
\end{bmatrix}$$

and

$$\mathbf{G} = \begin{bmatrix} n_{x\bar{x}} & n_{y\bar{x}} & n_{z\bar{x}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ n_{x\bar{y}} & n_{y\bar{y}} & n_{z\bar{y}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ n_{x\bar{z}} & n_{y\bar{z}} & n_{z\bar{z}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} & n_{z\bar{x}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & n_{x\bar{y}} & n_{y\bar{y}} & n_{z\bar{y}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & n_{x\bar{z}} & n_{y\bar{z}} & n_{z\bar{z}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} & n_{z\bar{x}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & n_{x\bar{y}} & n_{y\bar{y}} & n_{z\bar{y}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & n_{x\bar{z}} & n_{y\bar{z}} & n_{z\bar{z}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} & n_{z\bar{x}} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & n_{x\bar{y}} & n_{y\bar{y}} & n_{z\bar{y}} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & n_{x\bar{z}} & n_{y\bar{z}} & n_{z\bar{z}} \end{bmatrix}$$

where the axial stiffness D_{EA}, the bending stiffness $D_{EI_{\bar{z}}}$, the bending stiffness $D_{EI_{\bar{y}}}$, and the St. Venant torsion stiffness D_{GK} are given by

$$D_{EA} = EA; \quad D_{EI_{\bar{z}}} = EI_{\bar{z}}; \quad D_{EI_{\bar{y}}} = EI_{\bar{y}}; \quad D_{GK} = GK_v$$

The length L is given by

$$L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

The transformation matrix **G** contains direction cosines computed as

$$n_{x\bar{x}} = \frac{x_2 - x_1}{L} \qquad n_{y\bar{x}} = \frac{y_2 - y_1}{L} \qquad n_{z\bar{x}} = \frac{z_2 - z_1}{L}$$

$$n_{x\bar{z}} = \frac{x_{\bar{z}}}{L_{\bar{z}}} \qquad n_{y\bar{z}} = \frac{y_{\bar{z}}}{L_{\bar{z}}} \qquad n_{z\bar{z}} = \frac{z_{\bar{z}}}{L_{\bar{z}}}$$

$$n_{x\bar{y}} = n_{y\bar{z}} n_{z\bar{x}} - n_{z\bar{z}} n_{y\bar{x}}$$

$$n_{y\bar{y}} = n_{z\bar{z}} n_{x\bar{x}} - n_{x\bar{z}} n_{z\bar{x}}$$

$$n_{z\bar{y}} = n_{x\bar{z}} n_{y\bar{x}} - n_{y\bar{z}} n_{x\bar{x}}$$

where

$$L_{\bar{z}} = \sqrt{x_{\bar{z}}^2 + y_{\bar{z}}^2 + z_{\bar{z}}^2}$$

The element load vector $\mathbf{f}_l^e$, stored in `fe`, is computed according to

$$\mathbf{f}_l^e = \mathbf{G}^T \tilde{\mathbf{f}}_l^e$$

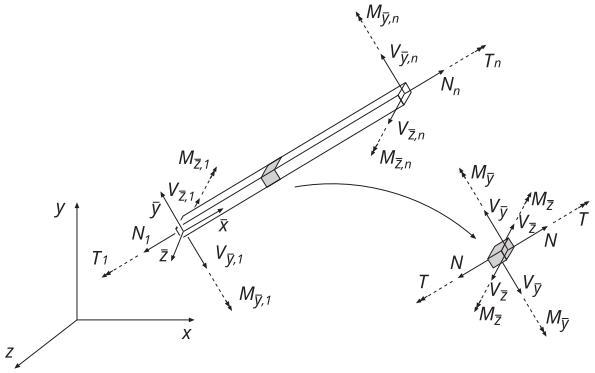
where

$$\bar{\mathbf{f}}_l^e = \begin{bmatrix} \frac{q_{\bar{x}}L}{2} \\ \frac{q_{\bar{y}}L}{2} \\ \frac{q_{\bar{z}}L}{2} \\ \frac{q_{\bar{\omega}}L}{2} \\ -\frac{q_{\bar{z}}L^2}{12} \\ \frac{q_{\bar{y}}L^2}{12} \\ \frac{q_{\bar{x}}L}{2} \\ \frac{q_{\bar{y}}L}{2} \\ \frac{q_{\bar{z}}L}{2} \\ \frac{q_{\bar{\omega}}L}{2} \\ \frac{q_{\bar{z}}L^2}{12} \\ -\frac{q_{\bar{y}}L^2}{12} \end{bmatrix}$$

3.5.18 beam3s

Purpose

Compute section forces in a three dimensional beam element.



Syntax

```
es = cfc.beam3s(ex, ey, ez, eo, ep, ed)
es = cfc.beam3s(ex, ey, ez, eo, ep, ed, eq)
es, edi = cfc.beam3s(ex, ey, ez, eo, ep, ed, eq, n)
es, edi, eci = cfc.beam3s(ex, ey, ez, eo, ep, ed, eq, n)
```

Description

`beam3s` computes the section forces and displacements in local directions along the beam element `beam3e`.

The input variables `ex`, `ey`, `ez`, `eo`, `ep` and `eq` are defined in `beam3e`.

The element displacements, stored in `ed`, are obtained by the function `extract_ed`. If a distributed load is applied to the element, the variable `eq` must be included. The number of evaluation points for section forces and displacements are determined by `n`. If `n` is omitted, only the ends of the beam are evaluated.

The output variables:

$$\texttt{es}=\begin{bmatrix} N(0) & V_{\bar{y}}(0) & V_{\bar{z}}(0) & T(0) & M_{\bar{y}}(0) & M_{\bar{z}}(0) \\ N(\bar{x}_2) & V_{\bar{y}}(\bar{x}_2) & V_{\bar{z}}(\bar{x}_2) & T(\bar{x}_2) & M_{\bar{y}}(\bar{x}_2) & M_{\bar{z}}(\bar{x}_2) \\ N(\bar{x}_{n-1}) & V_{\bar{y}}(\bar{x}_{n-1}) & V_{\bar{z}}(\bar{x}_{n-1}) & T(\bar{x}_{n-1}) & M_{\bar{y}}(\bar{x}_{n-1}) & M_{\bar{z}}(\bar{x}_{n-1}) \\ N(L) & V_{\bar{y}}(L) & V_{\bar{z}}(L) & T(\bar{x}_{n-1}) & M_{\bar{y}}(L) & M_{\bar{z}}(L) \end{bmatrix}$$

$$\texttt{edi}=\begin{bmatrix} u(0) & v(0) & w(0) & \varphi(0) \\ u(\bar{x}_2) & v(\bar{x}_2) & w(\bar{x}_2) & \varphi(\bar{x}_2) \\ \vdots & \vdots & \vdots & \vdots \\ u(\bar{x}_{n-1}) & v(\bar{x}_{n-1}) & w(\bar{x}_{n-1}) & \varphi(\bar{x}_{n-1}) \\ u(L) & v(L) & w(L) & \varphi(L) \end{bmatrix} \qquad \texttt{eci}=\begin{bmatrix} 0 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_{n-1} \\ L \end{bmatrix}$$

contain the section forces, the displacements, and the evaluation points on the local $\bar{x}$-axis. L is the length of the beam element.

Theory

The nodal displacements in local coordinates are given by

$$\bar{\mathbf{a}}^e = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \\ \bar{u}_3 \\ \bar{u}_4 \\ \bar{u}_5 \\ \bar{u}_6 \\ \bar{u}_7 \\ \bar{u}_8 \\ \bar{u}_9 \\ \bar{u}_{10} \\ \bar{u}_{11} \\ \bar{u}_{12} \end{bmatrix} = \mathbf{G}\mathbf{a}^e$$

where $\mathbf{G}$ is described in `beam3e` and the transpose of $\mathbf{a}^e$ is stored in `ed`.

The displacements associated with bar action, beam action in the $\bar{x}\bar{y}$-plane, beam action in the $\bar{x}\bar{z}$-plane, and torsion are determined as

$$\bar{\mathbf{a}}^e_{\text{bar}} = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_7 \end{bmatrix}; \quad \bar{\mathbf{a}}^e_{\text{beam},\bar{z}} = \begin{bmatrix} \bar{u}_2 \\ \bar{u}_6 \\ \bar{u}_8 \\ \bar{u}_{12} \end{bmatrix}; \quad \bar{\mathbf{a}}^e_{\text{beam},\bar{y}} = \begin{bmatrix} \bar{u}_3 \\ -\bar{u}_5 \\ \bar{u}_9 \\ -\bar{u}_{11} \end{bmatrix}; \quad \bar{\mathbf{a}}^e_{\text{torsion}} = \begin{bmatrix} \bar{u}_4 \\ \bar{u}_{10} \end{bmatrix}$$

The displacement $u(\bar{x})$ and the normal force $N(\bar{x})$ are computed from

$$u(\bar{x}) = \mathbf{N}_{\text{bar}}\bar{\mathbf{a}}^e_{\text{bar}} + u_p(\bar{x})$$

$$N(\bar{x}) = D_{EA}\mathbf{B}_{\text{bar}}\hat{\mathbf{a}}^e + N_p(\bar{x})$$

where

$$\mathbf{N}_{\text{bar}} = \begin{bmatrix} 1 & \bar{x} \end{bmatrix} \mathbf{C}_{\text{bar}}^{-1} = \begin{bmatrix} 1 - \frac{\bar{x}}{L} & \frac{\bar{x}}{L} \end{bmatrix}$$

$$\mathbf{B}_{\text{bar}} = \begin{bmatrix} 0 & 1 \end{bmatrix} \mathbf{C}_{\text{bar}}^{-1} = \begin{bmatrix} -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

$$u_p(\bar{x}) = -\frac{q_{\bar{x}}}{D_{EA}} \left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2} \right)$$

$$N_p(\bar{x}) = -q_{\bar{x}} \left(\bar{x} - \frac{L}{2} \right)$$

in which D_{EA}, L, and $q_{\bar{x}}$ are defined in `beam3e` and

$$\mathbf{C}_{\text{bar}}^{-1} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix}$$

The displacement $v(\bar{x})$, the bending moment $M_{\bar{z}}(\bar{x})$ and the shear force $V_{\bar{y}}(\bar{x})$ are computed from

$$v(\bar{x}) = \mathbf{N}_{\text{beam}}\hat{\mathbf{a}}^e_{\text{beam},\bar{z}} + v_p(\bar{x})$$

$$M_{\bar{z}}(\bar{x}) = D_{EI_{\bar{z}}}\mathbf{B}_{\text{beam}}\hat{\mathbf{a}}^e_{\text{beam},\bar{z}} + M_{\bar{z},p}(\bar{x})$$

$$V_{\bar{y}}(\bar{x}) = -D_{EI_{\bar{z}}}\frac{d\mathbf{B}_{\text{beam}}}{dx}\hat{\mathbf{a}}^e_{\text{beam},\bar{z}} + V_{\bar{y},p}(\bar{x})$$

where

$$\mathbf{N}_{\text{beam}} = \begin{bmatrix} 1 & \bar{x} & \bar{x}^2 & \bar{x}^3 \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$\mathbf{B}_{\text{beam}} = \begin{bmatrix} 0 & 0 & 2 & 6\bar{x} \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$\frac{d\mathbf{B}_{\text{beam}}}{dx} = \begin{bmatrix} 0 & 0 & 0 & 6 \end{bmatrix} \mathbf{C}_{\text{beam}}^{-1}$$

$$v_p(\bar{x}) = \frac{q_{\bar{y}}}{D_{EI_{\bar{z}}}} \left(\frac{\bar{x}^4}{24} - \frac{L\bar{x}^3}{12} + \frac{L^2\bar{x}^2}{24} \right)$$

$$M_{\bar{z},p}(\bar{x}) = q_{\bar{y}} \left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2} + \frac{L^2}{12} \right)$$

$$V_{\bar{y},p}(\bar{x}) = -q_{\bar{y}} \left(\bar{x} - \frac{L}{2} \right)$$

in which $D_{EI_{\bar{z}}}$, L, and $q_{\bar{y}}$ are defined in `beam3e` and

$$\mathbf{C}_{\text{beam}}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{3}{L^2} & -\frac{2}{L} & \frac{3}{L^2} & -\frac{1}{L} \\ \frac{2}{L^3} & \frac{1}{L^2} & -\frac{2}{L^3} & \frac{1}{L^2} \end{bmatrix}$$

The displacement $w(\bar{x})$, the bending moment $M_{\bar{y}}(\bar{x})$ and the shear force $V_{\bar{z}}(\bar{x})$ are computed from

$$w(\bar{x}) = \mathbf{N}_{\text{beam}}\hat{\mathbf{a}}^e_{\text{beam},\bar{y}} + w_p(\bar{x})$$

$$M_{\bar{y}}(\bar{x}) = -D_{EI_{\bar{y}}} \mathbf{B}_{\text{beam}} \hat{\mathbf{a}}^e_{\text{beam},\bar{y}} + M_{\bar{y},p}(\bar{x})$$

$$V_{\bar{z}}(\bar{x}) = -D_{EI_{\bar{y}}} \frac{d\mathbf{B}_{\text{beam}}}{dx} \hat{\mathbf{a}}^e_{\text{beam},\bar{y}} + V_{\bar{z},p}(\bar{x})$$

where

$$w_p(\bar{x}) = \frac{q_{\bar{z}}}{D_{EI_{\bar{y}}}} \left(\frac{\bar{x}^4}{24} - \frac{L\bar{x}^3}{12} + \frac{L^2\bar{x}^2}{24} \right)$$

$$M_{\bar{y},p}(\bar{x}) = -q_{\bar{z}} \left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2} + \frac{L^2}{12} \right)$$

$$V_{\bar{z},p}(\bar{x}) = -q_{\bar{z}} \left(\bar{x} - \frac{L}{2} \right)$$

in which $D_{EI_{\bar{y}}}$, L, and $q_{\bar{z}}$ are defined in `beam3e` and $\mathbf{N}_{\text{beam}}$, $\mathbf{B}_{\text{beam}}$, and $\frac{d\mathbf{B}_{\text{beam}}}{dx}$ are given above.

The displacement $\varphi(\bar{x})$ and the torque $T(\bar{x})$ are computed from

$$\varphi(\bar{x}) = \mathbf{N}_{\text{torsion}} \hat{\mathbf{a}}^e_{\text{torsion}} + \varphi_p(\bar{x})$$

$$T(\bar{x}) = D_{GK} \mathbf{B}_{\text{torsion}} \hat{\mathbf{a}}^e + T_p(\bar{x})$$

where

$$\mathbf{N}_{\text{torsion}} = \mathbf{N}_{\text{bar}}$$

$$\mathbf{B}_{\text{torsion}} = \mathbf{B}_{\text{bar}}$$

$$\varphi_p(\bar{x}) = -\frac{q_\omega}{D_{GK}} \left(\frac{\bar{x}^2}{2} - \frac{L\bar{x}}{2} \right)$$

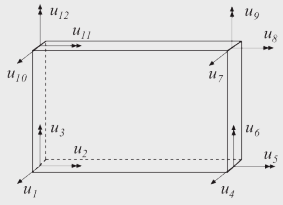
$$T_p(\bar{x}) = -q_\omega \left(\bar{x} - \frac{L}{2} \right)$$

in which D_{GK}, L, and q_ω are defined in `beam3e`.

3.6 Plate element functions

Only one plate element is currently available, a rectangular 12 dof element. The element presumes a linear elastic material which can be isotropic or anisotropic.

Table 16: **Plate elements**



`platre`

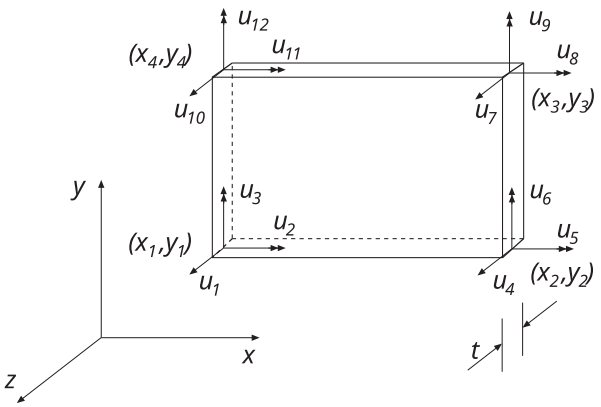
Table 17: Plate functions

planre	Compute element matrices
planrs	Compute section forces

3.6.1 platre

Purpose

Compute element stiffness matrix for a rectangular plate element.



Syntax

```
Ke = platre(ex, ey, ep, D)
[Ke, fe] = platre(ex, ey, ep, D, eq)
```

Description

`platre` provides an element stiffness matrix Ke, and an element load vector fe, for a rectangular plate element. This element can only be used if the element edges are parallel to the coordinate axes.

The element nodal coordinates x_1, y_1, x_2 etc. are supplied to the function by **ex** and **ey**, the element thickness t by **ep**, and the material properties by the constitutive matrix D. Any arbitrary **D**-matrix with dimensions (3×3) and valid for plane stress may be given. For an isotropic elastic material the constitutive matrix can be formed by the function `hooke` (see Section *Material functions*).

$$\mathbf{ex} = [\, x_1 \;\; x_2 \;\; x_3 \;\; x_4 \,] \qquad \mathbf{ep} = [\, t \,] \qquad \mathbf{D} = \begin{bmatrix} D_{11} & D_{12} & D_{13} \\ D_{21} & D_{22} & D_{23} \\ D_{31} & D_{32} & D_{33} \end{bmatrix}$$
$$\mathbf{ey} = [\, y_1 \;\; y_2 \;\; y_3 \;\; y_4 \,]$$

If a uniformly distributed load is applied to the element, the element load vector fe is computed. The input variable

$$\mathbf{eq} = [\, q_z \,]$$

then contains the load q_z per unit area in the z-direction.

Theory

The element stiffness matrix $\mathbf{K}^e$ and the element load vector $\mathbf{f}_l^e$, stored in Ke and fe respectively, are computed according to

$$\mathbf{K}^e = (\mathbf{C}^{-1})^T \int_A \bar{\mathbf{B}}^T \tilde{\mathbf{D}}\, \bar{\mathbf{B}}\, dA\, \mathbf{C}^{-1}$$

$$\mathbf{f}_l^e = (\mathbf{C}^{-1})^T \int_A \bar{\mathbf{N}}^T q_z\, dA$$

where the constitutive matrix

$$\tilde{\mathbf{D}} = \frac{t^3}{12}\mathbf{D}$$

and where $\mathbf{D}$ is defined by D.

The evaluation of the integrals for the rectangular plate element is based on the displacement approximation $w(x,y)$ and is expressed in terms of the nodal variables $u_1, u_2, \ldots, u_{12}$ as

$$w(x,y) = \mathbf{N}^e\mathbf{a}^e = \bar{\mathbf{N}}\mathbf{C}^{-1}\mathbf{a}^e$$

where

$$\bar{\mathbf{N}} = [\,1 \;\; x \;\; y \;\; x^2 \;\; xy \;\; y^2 \;\; x^3 \;\; x^2y \;\; xy^2 \;\; y^3 \;\; x^3y \;\; xy^3\,]$$

$$\mathbf{C} = \begin{bmatrix}
1 & -a & -b & a^2 & ab & b^2 & -a^3 & -a^2b & -ab^2 & -b^3 & a^3b & ab^3 \\
0 & 0 & 1 & 0 & -a & -2b & 0 & a^2 & 2ab & 3b^2 & -a^3 & -3ab^2 \\
0 & -1 & 0 & 2a & b & 0 & -3a^2 & -2ab & -b^2 & 0 & 3a^2b & b^3 \\
1 & a & -b & a^2 & -ab & b^2 & a^3 & -a^2b & ab^2 & -b^3 & -a^3b & -ab^3 \\
0 & 0 & 1 & 0 & a & -2b & 0 & a^2 & -2ab & 3b^2 & a^3 & 3ab^2 \\
0 & -1 & 0 & -2a & b & 0 & -3a^2 & 2ab & -b^2 & 0 & 3a^2b & b^3 \\
1 & a & b & a^2 & ab & b^2 & a^3 & a^2b & ab^2 & b^3 & a^3b & ab^3 \\
0 & 0 & 1 & 0 & a & 2b & 0 & a^2 & 2ab & 3b^2 & a^3 & 3ab^2 \\
0 & -1 & 0 & -2a & -b & 0 & -3a^2 & -2ab & -b^2 & 0 & -3a^2b & -b^3 \\
1 & -a & b & a^2 & -ab & b^2 & -a^3 & a^2b & -ab^2 & b^3 & -a^3b & -ab^3 \\
0 & 0 & 1 & 0 & -a & 2b & 0 & a^2 & -2ab & 3b^2 & -a^3 & -3ab^2 \\
0 & -1 & 0 & 2a & -b & 0 & -3a^2 & 2ab & -b^2 & 0 & -3a^2b & -b^3
\end{bmatrix}$$

$$\mathbf{a}^e = [\,u_1 \;\; u_2 \;\; \ldots \;\; u_{12}\,]^T$$

and where

$$a = \frac{1}{2}(x_3 - x_1) \qquad b = \frac{1}{2}(y_3 - y_1)$$

The matrix $\bar{\mathbf{B}}$ is obtained as

$$\bar{\mathbf{B}} = \overset{*}{\nabla}\, \bar{\mathbf{N}} = \begin{bmatrix}
0 & 0 & 0 & 2 & 0 & 0 & 6x & 2y & 0 & 0 & 6xy & 0 \\
0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 2x & 6y & 0 & 6xy \\
0 & 0 & 0 & 0 & 2 & 0 & 0 & 4x & 4y & 0 & 6x^2 & 6y^2
\end{bmatrix}$$

where

$$\overset{*}{\nabla} = \begin{bmatrix} \frac{\partial^2}{\partial x^2} \\ \frac{\partial^2}{\partial y^2} \\ 2\frac{\partial^2}{\partial x \partial y} \end{bmatrix}$$

Evaluation of the integrals for the rectangular plate element is done analytically. Computation of the integrals for the element load vector $\mathbf{f}_l^e$ yields

$$\mathbf{f}_l^e = q_z L_x L_y \begin{bmatrix} \frac{1}{4} \\ \frac{L_y}{24} \\ -\frac{L_x}{24} \\ \frac{1}{4} \\ \frac{L_y}{24} \\ \frac{L_x}{24} \\ \frac{1}{4} \\ -\frac{L_y}{24} \\ \frac{L_x}{24} \\ \frac{1}{4} \\ -\frac{L_y}{24} \\ -\frac{L_x}{24} \end{bmatrix}$$

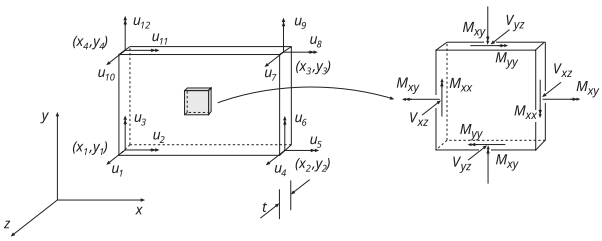
where

$$L_x = x_3 - x_1 \qquad L_y = y_3 - y_1$$

3.6.2 platrs

Purpose

Compute section forces in a rectangular plate element.



Syntax

```
[es,et]=platrs(ex,ey,ep,D,ed)
```

Description

`platrs` computes the section forces `es` and the curvature matrix `et` in a rectangular plate element. The section forces and the curvatures are computed at the center of the element.

The input variables **ex**, **ey**, **ep** and **D** are defined in `platre`. The vector **ed** contains the nodal displacements $\mathbf{a}^e$ of the element and is obtained by the function `extract` as

$$\mathbf{ed} = (\mathbf{a}^e)^T = [\, u_1 \;\; u_2 \;\; \ldots \;\; u_{12} \,]$$

The output variables

$$\mathtt{es} = \left[\, \mathbf{M}^T \; \mathbf{V}^T \,\right] = [\, M_{xx} \; M_{yy} \; M_{xy} \; V_{xz} \; V_{yz} \,]$$

$$\mathtt{et} = \boldsymbol{\kappa}^T = [\, \kappa_{xx} \; \kappa_{yy} \; \kappa_{xy} \,]$$

contain the section forces and curvatures in global directions.

Theory

The curvatures and the section forces are computed according to

$$\boldsymbol{\kappa} = \begin{bmatrix} \kappa_{xx} \\ \kappa_{yy} \\ \kappa_{xy} \end{bmatrix} = \bar{\mathbf{B}}\, \mathbf{C}^{-1}\, \mathbf{a}^e$$

$$\mathbf{M} = \begin{bmatrix} M_{xx} \\ M_{yy} \\ M_{xy} \end{bmatrix} = \tilde{\mathbf{D}}\, \boldsymbol{\kappa}$$

$$\mathbf{V} = \begin{bmatrix} V_{xz} \\ V_{yz} \end{bmatrix} = \tilde{\nabla}\, \mathbf{M}$$

where the matrices $\tilde{\mathbf{D}}$, $\bar{\mathbf{B}}$, **C** and $\mathbf{a}^e$ are described in `platre`, and where

$$\tilde{\nabla} = \begin{bmatrix} \dfrac{\partial}{\partial x} & 0 & \dfrac{\partial}{\partial y} \\ 0 & \dfrac{\partial}{\partial y} & \dfrac{\partial}{\partial x} \end{bmatrix}$$

Chapter 4

System functions

The group of system functions comprises functions for the setting up, solving, and elimination of systems of equations. The functions are separated in two groups:

- Static system functions
- Dynamic system functions

Static system functions concern the linear system of equations

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

where $\mathbf{K}$ is the global stiffness matrix and $\mathbf{f}$ is the global load vector. Often used static system functions are `assem` and `solveq`. The function `assem` assembles the global stiffness matrix and `solveq` computes the global displacement vector $\mathbf{a}$ considering the boundary conditions. It should be noted that $\mathbf{K}$, $\mathbf{f}$, and $\mathbf{a}$ also represent analogous quantities in systems other than structural mechanical systems. For example, in a heat flow problem $\mathbf{K}$ represents the conductivity matrix, $\mathbf{f}$ the heat flow, and $\mathbf{a}$ the temperature.

Dynamic system functions are related to different aspects of linear dynamic systems of coupled ordinary differential equations according to

$$\mathbf{C}\dot{\mathbf{a}} + \mathbf{K}\mathbf{a} = \mathbf{f}$$

for first-order systems and

$$\mathbf{M}\ddot{\mathbf{a}} + \mathbf{C}\dot{\mathbf{a}} + \mathbf{K}\mathbf{a} = \mathbf{f}$$

for second-order systems. First-order systems occur typically in transient heat conduction and second-order systems occur in structural dynamics.

4.1 Static system functions

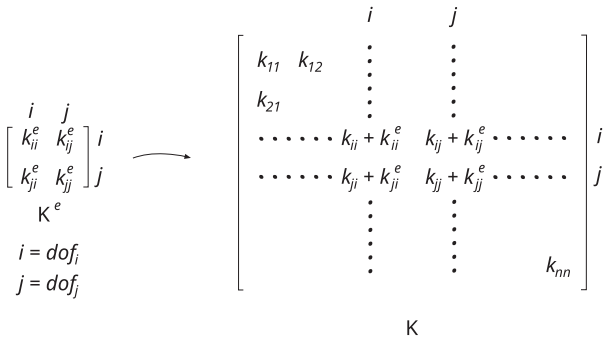
The group of static system functions comprises functions for setting up and solving the global system of equations. It also contains a function for eigenvalue analysis,

a function for static condensation, a function for extraction of element displacements from the global displacement vector and a function for extraction of element coordinates.

4.1.1 assem

Purpose

Assemble element matrices.



Syntax

```
K = cfc.assem(edof, K, Ke)
K, f = cfc.assem(edof, K, Ke, f, fe)
```

Description

`assem` adds the element stiffness matrix $\mathbf{K}^e$, stored in `Ke`, to the structure stiffness matrix $\mathbf{K}$, stored in `K`, according to the topology matrix `edof`.

The element topology matrix `edof` is defined as

$$\texttt{edof} = [el \quad \underbrace{dof_1 \quad dof_2 \quad \dots \quad dof_{ned}}_{\text{global dof.}}]$$

where the first column contains the element number, and columns 2 to $(ned + 1)$ contain the corresponding global degrees of freedom (ned = number of element degrees of freedom).

In the case where the matrix $\mathbf{K}^e$ is identical for several elements, assembling of these can be carried out simultaneously. Each row in `edof` then represents one element, i.e. nel is the total number of considered elements.

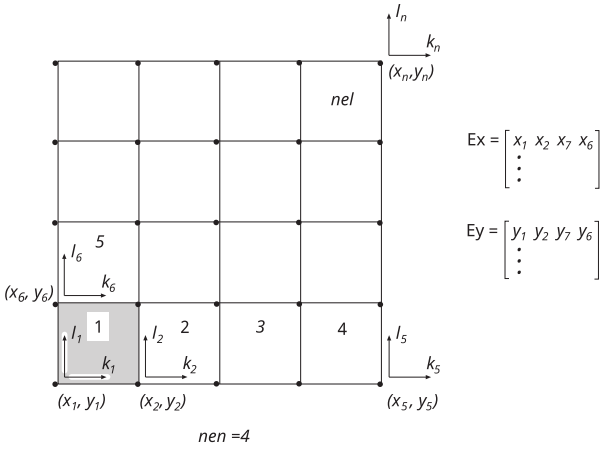
$$\texttt{Edof} = \left.\begin{bmatrix} el_1 & dof_1 & dof_2 & \cdots & \cdots & \cdots & dof_{ned} \\ el_2 & dof_1 & dof_2 & \cdots & \cdots & \cdots & dof_{ned} \\ \vdots & \vdots & \vdots & & & & \vdots \\ el_{nel} & dof_1 & dof_2 & \cdots & \cdots & \cdots & dof_{ned} \end{bmatrix}\right\} \text{one row for each element}$$

If **fe** and **f** are given in the function, the element load vector $\mathbf{f}^e$ is also added to the global load vector **f**.

4.1.2 coordxtr

Purpose

Extract element coordinates from a global coordinate matrix.



Syntax

```
Ex, Ey, Ez = cfc.coordxtr(Edof, Coord, Dof, nen)
```

Description

`coordxtr` extracts element nodal coordinates from the global coordinate matrix `Coord` for elements with equal numbers of element nodes and dof's.

Input variables are the element topology matrix `Edof`, defined in `assem`, the global coordinate matrix `Coord`, the global topology matrix `Dof`, and the number of element nodes nen in each element.

$$\texttt{Coord}=\begin{bmatrix} x_1 & y_1 & [z_1] \\ x_2 & y_2 & [z_2] \\ x_3 & y_3 & [z_3] \\ \vdots & \vdots & \vdots \\ x_n & y_n & [z_n] \end{bmatrix} \qquad \texttt{Dof}=\begin{bmatrix} k_1 & l_1 & \dots & m_1 \\ k_2 & l_2 & \dots & m_2 \\ k_3 & l_3 & \dots & m_3 \\ \vdots & \vdots & \dots & \vdots \\ k_n & l_n & \dots & m_n \end{bmatrix} \qquad \texttt{nen}=[\ nen\ ]$$

The nodal coordinates defined in row i of `Coord` correspond to the degrees of freedom of row i in `Dof`. The components k_i, l_i and m_i define the degrees of freedom of node i, and n is the number of global nodes for the considered part of the FE-model.

The output variables `Ex`, `Ey`, and `Ez` are matrices defined according to

$$\texttt{Ex}=\begin{bmatrix} x_1^1 & x_2^1 & x_3^1 & \dots & x_{nen}^1 \\ x_1^2 & x_2^2 & x_3^2 & \dots & x_{nen}^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_1^{nel} & x_2^{nel} & x_3^{nel} & \dots & x_{nen}^{nel} \end{bmatrix}$$

where row i gives the x-coordinates of the element defined in row i of `Edof`, and where nel is the number of considered elements.

The element coordinate data extracted by the function `coordxtr` can be used for plotting purposes and to create input data for the element stiffness functions.

4.1.3 eigen

Purpose

Solve the generalized eigenvalue problem.

Syntax

```
L = cfc.eigen(K, M)
L = cfc.eigen(K, M, b)
L, X = cfc.eigen(K, M)
L, X = cfc.eigen(K, M, b)
```

Description

`eigen` solves the eigenvalue problem

$$|\mathbf{K} - \lambda \mathbf{M}| = 0$$

where **K** and **M** are square matrices. The eigenvalues λ are stored in the vector **L** and the corresponding eigenvectors in the matrix **X**.

If certain rows and columns in matrices **K** and **M** are to be eliminated in computing the eigenvalues, `b` must be given in the function. The rows (and columns) that are to be eliminated are described in the vector `b` defined as

$$\mathsf{b} = \begin{bmatrix} dof_1 \\ dof_2 \\ \vdots \\ dof_{nb} \end{bmatrix}$$

The computed eigenvalues are given in order ranging from the smallest to the largest. The eigenvectors are normalized so that

$$\mathbf{X}^T \mathbf{M} \mathbf{X} = \mathbf{I}$$

where **I** is the identity matrix.

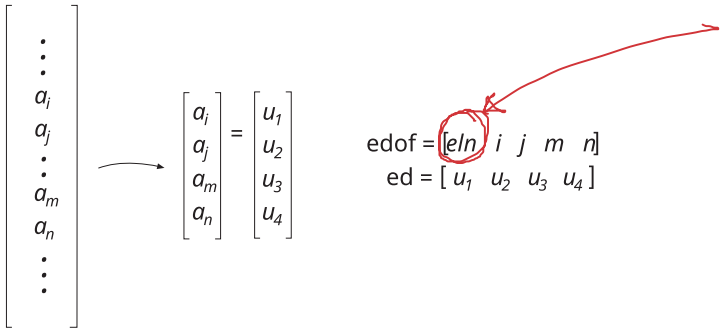
4.1.4 extract_ed

Purpose

Extract element nodal quantities from a global solution vector.

Syntax

```
ed = cfc.extract_ed(edof, a)
```



Description

The `extract_ed` function extracts element displacements or corresponding quantities $\mathbf{a}^e$ from the global solution vector **a**, stored in `a`.

Input variables are the element topology matrix `edof`, defined in `assem`, and the global solution vector `a`.

The output variable

$$\texttt{ed} = (\mathbf{a}^e)^T$$

contains the element displacement vector.

If `Edof` contains more than one element, `Ed` will be a matrix

$$\texttt{Ed} = \begin{bmatrix} (\mathbf{a}^e)_1^T \\ (\mathbf{a}^e)_2^T \\ \vdots \\ (\mathbf{a}^e)_{nel}^T \end{bmatrix}$$

where row i gives the element displacements for the element defined in row i of `Edof`, and nel is the total number of considered elements.

Example

For the two-dimensional beam element, the `extract_ed` function will extract six nodal displacements for each element given in `Edof`, and create a matrix `Ed` of size $(nel \times 6)$.

$$\texttt{Ed} = \begin{bmatrix} u_1 & u_2 & u_3 & u_4 & u_5 & u_6 \\ u_1 & u_2 & u_3 & u_4 & u_5 & u_6 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ u_1 & u_2 & u_3 & u_4 & u_5 & u_6 \end{bmatrix}$$

4.1.5 red

Purpose

Reduce the size of a square matrix by omitting rows and columns.

Syntax

```
B = cfc.red(A, b)
B, b = cfc.red(A, b)
```

Description

`B = red(A, b)` reduces the square matrix `A` to a smaller matrix `B` by omitting rows and columns of `A`. The indices for rows and columns to be omitted are specified by the column vector `b`.

Example

Assume that the matrix `A` is defined as

$$\texttt{A}=\begin{bmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\end{bmatrix}$$

and `b` as

$$\texttt{b}=\begin{bmatrix}2\\ 4\end{bmatrix}$$

The statement `B = red(A, b)` results in the matrix

$$\texttt{B}=\begin{bmatrix}1 & 3\\ 9 & 11\end{bmatrix}$$

4.1.6 solveq

Purpose

Solve equation system.

Syntax

```
a = cfc.solveq(K, f)
a = cfc.solveq(K, f, bcPrescr, bcVal)
a, r = cfc.solveq(K, f, bcPrescr, bcVal)
```

Description

The function `solveq` solves the equation system

$$\mathbf{K}\,\mathbf{a}=\mathbf{f}$$

where $\mathbf{K}$ is a matrix and $\mathbf{a}$ and $\mathbf{f}$ are vectors.

The matrix $\mathbf{K}$ and the vector $\mathbf{f}$ must be predefined. The solution of the system of equations is stored in a vector $\mathbf{a}$ which is created by the function.

If some values of $\mathbf{a}$ are to be prescribed, the row number and the corresponding values are given in the boundary condition matrices

$$\texttt{bcPresc}=\begin{bmatrix}dof_1\\ dof_2\\ \vdots\\ dof_{nbc}\end{bmatrix}\qquad \texttt{bcVal}=\begin{bmatrix}u_1\\ u_2\\ \vdots\\ u_{nbc}\end{bmatrix}$$

where `bcPresc` contains the row numbers and `bcVal` the corresponding prescribed values. If `bcVal` is omitted, all prescribed values are assumed to be 0.

If `r` is given in the function, support forces are computed according to

$\mathtt{r} = \mathbf{K}\,\mathbf{a} - \mathbf{f}$

4.1.7 statcon

Purpose

Reduce system of equations by static condensation.

Syntax

```
K1, f1 = cfc.statcon(K, f, b)
```

Description

`statcon` reduces a system of equations

$$\mathbf{K}\,\mathbf{a} = \mathbf{f}$$

by static condensation.

The degrees of freedom to be eliminated are supplied to the function by the vector

$\mathtt{b} = \begin{bmatrix} dof_1 \\ dof_2 \\ \vdots \\ dof_{nb} \end{bmatrix}$

where each row in `b` contains one degree of freedom to be eliminated.

The elimination gives the reduced system of equations

$$\mathbf{K}_1\,\mathbf{a}_1 = \mathbf{f}_1$$

where $\mathbf{K}_1$ and $\mathbf{f}_1$ are stored in `K1` and `f1` respectively.

4.2 Dynamic system functions

The group of system functions comprises functions for solving linear dynamic systems by time stepping or modal analysis, functions for frequency domain analysis, etc.

4.2.1 dyna2

Compute the dynamic solution to a set of uncoupled second-order differential equations.

Syntax

```
X = dyna2(w2, xi, f, g, dt)
```

Description

`dyna2` computes the solution to the set

$$\ddot{x}_i + 2\xi_i\omega_i\dot{x}_i + \omega_i^2 x_i = f_i g(t), \qquad i = 1, \ldots, m$$

of differential equations, where $g(t)$ is a piecewise linear time function.

The vectors `w2`, `xi`, and `f` contain the squared circular frequencies ω_i^2, the damping ratios ξ_i, and the applied forces f_i, respectively.

The vector `g` defines the load function in terms of straight line segments between equally spaced points in time. This function may have been formed by the command `gfunc`.

The dynamic solution is computed at equal time increments defined by `dt`. Including the initial zero vector as the first column vector, the result is stored in the $m \times n$ matrix `X`, where $n - 1$ is the number of time steps.

Note

The accuracy of the solution is *not* a function of the output time increment `dt`, since the command produces the exact solution for straight line segments in the loading time function.

See also

gfunc

4.2.2 dyna2f

Purpose

Compute the dynamic solution to a set of uncoupled second-order differential equations.

Syntax

```
Y = dyna2f(w2, xi, f, p, dt)
```

Description

`dyna2f` computes the solution to the set of differential equations:

$$\ddot{x}_i + 2\xi_i\omega_i\dot{x}_i + \omega_i^2 x_i = f_i g(t), \qquad i = 1, \ldots, m$$

The vectors `w2`, `xi` and `f` are the squared circular frequencies ω_i^2, the damping ratios ξ_i, and the applied forces f_i, respectively. The force vector `p` contains the Fourier coefficients $p(k)$ formed by the command `fft`.

The solution in the frequency domain is computed at equal time increments defined by `dt`. The result is stored in the $m \times n$ matrix `Y`, where `m` is the number of equations and `n` is the number of frequencies resulting from the `fft`

command. The dynamic solution in the time domain is achieved by the use of the command `ifft`.

Example

The dynamic solution to a set of uncoupled second-order differential equations can be computed by the following sequence of commands:

```
>> g = gfunc(G, dt);
>> p = fft(g);
>> Y = dyna2f(w2, xi, f, p, dt);
>> X = (real(ifft(Y.')))';
```

where it is assumed that the input variables `G`, `dt`, `w2`, `xi` and `f` are properly defined. Note that the `ifft` command operates on column vectors if `Y` is a matrix; therefore use the transpose of `Y`. The output from the `ifft` command is complex. Therefore use `Y.'` to transpose rows and columns in `Y` in order to avoid the complex conjugate transpose of `Y`.

The time response is represented by the real part of the output from the `ifft` command. If the transpose is used and the result is stored in a matrix `X`, each row will represent the time response for each equation as the output from the command `dyna2`.

See also

gfunc, *fft*, *ifft*

4.2.3 fft

Purpose

Transform functions in time domain to frequency domain.

Syntax

```
p = fft(g)
p = fft(g, N)
```

Description

The `fft` function transforms a time dependent function to the frequency domain.

The function to be transformed is stored in the vector `g`. Each row in `g` contains the value of the function at equal time intervals. The function represents a span $-\infty \leq t \leq +\infty$; however, only the values within a typical period are specified by `g`.

The `fft` command can be used with one or two input arguments. If `N` is not specified, the number of frequencies used in the transformation is equal to the number of points in the time domain (i.e., the length of the variable `g`), and the output will be a vector of the same size containing complex values representing the frequency content of the input signal.

The scalar variable `N` can be used to specify the number of frequencies used in the Fourier transform. The size of the output vector in this case will be equal

to N. It should be remembered that the highest harmonic component in the time signal that can be identified by the Fourier transform corresponds to half the sampling frequency. The sampling frequency is equal to $1/dt$, where dt is the time increment of the time signal.

The complex Fourier coefficients $p(k)$ are stored in the vector p, and are computed according to

$$p(k) = \sum_{j=1}^{N} x(j)\omega_N^{(j-1)(k-1)},$$

where

$$\omega_N = e^{-2\pi i/N}.$$

> **Note**
>
> This is a MATLAB built-in function.

4.2.4 freqresp

Purpose

Compute frequency response of a known discrete time response.

Syntax

```
[Freq, Resp] = freqresp(D, dt)
```

Description

freqresp computes the frequency response of a discrete dynamic system.

- D is the time history function.
- dt is the sampling time increment, i.e., the time increment used in the time integration procedure.
- Resp contains the computed response as a function of frequency.
- Freq contains the corresponding frequencies.

Example

The result can be visualized by:

```
plot(Freq, Resp)
xlabel('frequency (Hz)')
```

or:

```
semilogy(Freq, Resp)
xlabel('frequency (Hz)')
```

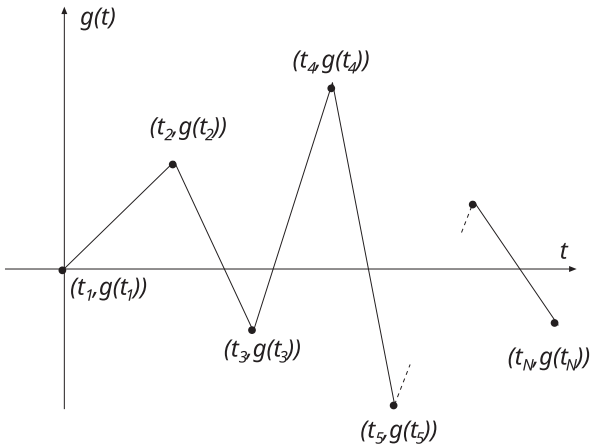
The dimension of `Resp` is the same as that of the original time history function.

> **Note**
>
> The time history function of a discrete system computed by direct integration often behaves in an unstructured manner. The reason for this is that the time history is a mixture of several participating eigenmodes at different eigenfrequencies. By using a Fourier transform, however, the response as a function of frequency can be computed efficiently. In particular, it is possible to identify the participating frequencies.

4.2.5 gfunc

Purpose

Form vector with function values at equally spaced points by linear interpolation.



Piecewise linear time dependent function

Syntax

```
[t, g] = gfunc(G, dt)
```

Description

`gfunc` uses linear interpolation to compute values at equally spaced points for a discrete function g given by

$$G = \begin{bmatrix} t_1 & g(t_1) \\ t_2 & g(t_2) \\ \vdots & \\ t_N & g(t_N) \end{bmatrix},$$

as shown in the figure above.

Function values are computed in the range $t_1 \leq t \leq t_N$, at equal increments, `dt` being defined by the variable `dt`. The number of linear segments (steps) is $(t_N - t_1)/dt$. The corresponding vector `t` is also computed. The result can be plotted by using the command `plot(t, g)`.

4.2.6 ifft

Purpose

Transform function in frequency domain to time domain.

Syntax

```
x = ifft(y)
x = ifft(y, N)
```

Description

`ifft` transforms a function in the frequency domain to a function in the time domain.

The function to be transformed is given in the vector `y`. Each row in `y` contains Fourier terms in the interval $-\infty \leq \omega \leq +\infty$.

The `ifft` command can be used with one or two input arguments. The scalar variable `N` can be used to specify the number of frequencies used in the Fourier transform. The size of the output vector in this case will be equal to `N`. See also the description of the command `fft`.

The inverse Fourier coefficients $x(j)$, stored in the variable `x`, are computed according to

$$x(j) = \frac{1}{N} \sum_{k=1}^{N} y(k) \omega_N^{-(j-1)(k-1)},$$

where

$$\omega_N = e^{-2\pi i/N}.$$

> **Note**
>
> This is a MATLAB built-in function.

See also

fft

4.2.7 ritz

Purpose

Compute approximative eigenvalues and eigenvectors by the Lanczos method.

Syntax

```
L = ritz(K, M, f, m)
L = ritz(K, M, f, m, b)
[L, X] = ritz(K, M, f, m)
[L, X] = ritz(K, M, f, m, b)
```

Description

`ritz` computes, by the use of the Lanczos algorithm, `m` approximative eigenvalues and `m` corresponding eigenvectors for a given pair of n-by-n matrices `K` and `M` and a given non-zero starting vector `f`.

If certain rows and columns in matrices **K** and **M** are to be eliminated in computing the eigenvalues, **b** must be given in the command. The rows (and columns) to be eliminated are described in the vector **b** defined as

$$\mathbf{b} = \begin{bmatrix} dof_1 \\ dof_2 \\ \vdots \\ dof_{nb} \end{bmatrix}$$

> **Note**
>
> If the number of vectors, `m`, is chosen less than the total number of degrees-of-freedom, n, only about the first `m/2` Ritz vectors are good approximations of the true eigenvectors. Recall that the Ritz vectors satisfy the `M`-orthonormality condition

$$\mathbf{X}^T\mathbf{M}\mathbf{X} = \mathbf{I}$$

where **I** is the identity matrix.

4.2.8 spectra

Purpose

Compute seismic response spectra for elastic design.

Syntax

```
s = spectra(a, xi, dt, f)
```

Description

The `spectra` function computes the seismic response spectrum for a known acceleration history function.

- The computation is based on the vector `a`, which contains an acceleration time history function defined at equal time steps.

- The time step is specified by the variable `dt`.
- The value of the damping ratio is given by the variable `xi`.
- Output from the computation, stored in the vector `s`, is achieved at frequencies specified by the column vector `f`.

Example

The following procedure can be used to produce a seismic response spectrum for a damping ratio $\xi = 0.05$, defined at 34 logarithmically spaced frequency points. The acceleration time history `a` has been sampled at a frequency of 50 Hz, corresponding to a time increment `dt = 0.02` between collected points:

```
freq = logspace(0, log10(2^(33/6)), 34);
xi = 0.05;
dt = 0.02;
s = spectra(a, xi, dt, freq');
```

The resulting spectrum can be plotted by the command:

```
loglog(freq, s, '*')
```

4.2.9 step1

Purpose

Compute the dynamic solution to a set of first order differential equations.

Syntax

```
[a,da] = step1(K, C, f, a0, bc, ip)
[a,da] = step1(K, C, f, a0, bc, ip, times)
[a,da, ahist, dahist] = step1(K, C, f, a0, bc, ip, times, dofs)
```

Description

`step1` computes at equal time steps the solution to a set of first order differential equations of the form:

$$\mathbf{C}\dot{\mathbf{a}} + \mathbf{K}\mathbf{a} = \mathbf{f}(x,t),$$
$$\mathbf{a}(0) = \mathbf{a}_0$$

The command solves transient field problems. In the case of heat conduction, `K` and `C` represent the $n \times n$ conductivity and capacity matrices, respectively. `a` is the temperature and `da` (i.e., $\dot{\mathbf{a}}$) is the time derivative of the temperature.

The matrix `f` contains the time-dependent load vectors. If no external loads are active, use `[]` for `f`. The matrix `f` is organized as follows:

```
f = [
time history of the load at dof_1
time history of the load at dof_2
...
time history of the load at dof_n
]
```

The dimension of `f` is:

(number of degrees-of-freedom) × (number of timesteps + 1)

The initial conditions are given by the vector `a0` containing initial values of `a`.

The matrix `bc` contains the time-dependent prescribed values of the field variable `a`. If no field variables are prescribed, use `[]` for `bc`. The matrix `bc` is organized as follows:

```
bc = [
dof_1   time history of the field variable
dof_2   time history of the field variable
 ...
dof_m2  time history of the field variable
]
```

The dimension of `bc` is:

(number of dofs with prescribed field values) × (number of timesteps + 2)

The time integration procedure is governed by the parameters given in the vector `ip` defined as:

```
ip = [dt, T, alpha]
```

where `dt` specifies the length of the time increment, `T` is the total time, and `alpha` is the time integration constant. Frequently used values of `alpha` are:

alpha	Method
0	Forward difference; forward Euler
0.5	Trapezoidal rule; Crank-Nicholson
1	Backward difference; backward Euler

The computed values of `a` and `da` are stored in `a` and `da`, respectively. The first column contains the initial values, and the following columns contain the values for each time step. The dimension is:

(number of degrees-of-freedom) × (number of time steps + 1)

If the values are to be stored only for specific times, the parameter `times` specifies at which times the solution will be stored. The values are stored in `a` and `da`, one column for each requested time according to `times`. The dimension is then:

(number of degrees-of-freedom) × (number of requested times + 1)

If the history is to be stored in `ahist` and `dahist` for some degrees of freedom, the parameter `dofs` specifies for which degrees of freedom the history is to be stored. The computed time histories are stored in `ahist` and `dahist`, respectively, with one row for each requested degree of freedom. The dimension is:

(number of specified degrees of freedom) × (number of timesteps + 1)

The time history functions can be generated using the command `gfunc`. If all the values of the time histories of `f` or `bc` are kept constant, these values need to be stated only once. In this case, the number of columns in `f` is one and in `bc` two.

In most cases, only a few degrees-of-freedom are affected by the exterior load, and hence the matrix contains only a few non-zero entries. In such cases, it is possible to save space by defining `f` as `sparse` (a MATLAB built-in function).

> **Note**
>
> Reference: Bathe, K.J.: *Finite Element Procedures in Engineering Analysis,* Prentice-Hall, Englewood Cliffs, New Jersey, pp. 511-514, 1982.

4.2.10 step2

Purpose

Compute the dynamic solution to a set of second order differential equations.

Syntax

```
[a, da, d2a] = step2(K, C, M, f, a0, da0, bc, ip)
[a, da, d2a] = step2(K, C, M, f, a0, da0, bc, ip, times)
[a, da, d2a, ahist, dahist, d2ahist] = step2(K, C, M, f, a0, da0,
↪  bc, ip, times, dofs)
```

Description

`step2` computes at equal time steps the solution to a set of second order differential equations of the form:

$$\begin{aligned} \mathbf{M}\ddot{\mathbf{a}} + \mathbf{C}\dot{\mathbf{a}} + \mathbf{K}\mathbf{a} &= \mathbf{f}(x,t), \\ \mathbf{a}(0) &= \mathbf{a}_0, \\ \dot{\mathbf{a}}(0) &= \mathbf{v}_0. \end{aligned}$$

In structural mechanics problems, `K`, `C` and `M` represent the $n \times n$ stiffness, damping and mass matrices, respectively. `a` is the displacement, `da` ($= \dot{\mathbf{a}}$) is the velocity and `d2a` ($= \ddot{\mathbf{a}}$) is the acceleration.

The matrix `f` contains the time-dependent load vectors. If no external loads are active, use `[]` for `f`. The matrix `f` is organized as:

$$f = \begin{bmatrix} \text{time history of the load at } dof_1 \\ \text{time history of the load at } dof_2 \\ \vdots \\ \text{time history of the load at } dof_n \end{bmatrix}$$

The dimension of `f` is:

(number of degrees-of-freedom) × (number of timesteps + 1)

The initial conditions are given by the vectors `a0` and `da0`, containing initial displacements and initial velocities.

The matrix `bc` contains the time-dependent prescribed displacement. If no displacements are prescribed, use `[]` for `bc`. The matrix `bc` is organized as:

$$bc = \begin{bmatrix} dof_1 & \text{time history of the displacement} \\ dof_2 & \text{time history of the displacement} \\ \vdots & \vdots \\ dof_{m_2} & \text{time history of the displacement} \end{bmatrix}$$

The dimension of `bc` is:

(number of dofs with prescribed displacement) × (number of timesteps + 2)

The time integration procedure is governed by the parameters given in the vector `ip` defined as:

$$ip = [dt, T, \alpha, \delta]$$

where `dt` specifies the time increment, `T` the total time, and `alpha` and `delta` are time integration constants for the Newmark family of methods.

Frequently used values:

α	δ	Method
$\frac{1}{4}$	$\frac{1}{2}$	Average acceleration (trapezoidal) rule
$\frac{1}{6}$	$\frac{1}{2}$	Linear acceleration
0	$\frac{1}{2}$	Central difference

The computed values of $\mathbf{a}$, $\dot{\mathbf{a}}$ and $\ddot{\mathbf{a}}$ are stored in `a`, `da` and `d2a`, respectively. The first column contains the initial values and the following columns contain the values for each time step.

The dimension of `a`, `da` and `d2a` is:

(number of degrees-of-freedom) × (number of time steps + 1)

If the values are to be stored only for specific times, the parameter `times` specifies at which times the solution will be stored. The values are stored in `a`, `da` and `d2a`, one column for each requested time according to `times`. The dimension is then:

(number of degrees-of-freedom) × (number of requested times + 1)

If the history is to be stored in `ahist`, `dahist` and `d2ahist` for some degrees of freedom, the parameter `dofs` specifies for which degrees of freedom the history is to be stored. The computed time histories are stored in `ahist`, `dahist` and `d2ahist`, one row for each requested degree of freedom according to `dofs`. The dimension is:

(number of specified degrees of freedom) × (number of timesteps + 1)

In most cases only a few degrees-of-freedom are affected by the exterior load, and hence the matrix contains only few non-zero entries. In such cases it is possible to save space by defining `f` as sparse (a MATLAB built-in function).

4.2.11 sweep

Purpose

Compute complex frequency response functions.

Syntax

```
Y = sweep(K, C, M, p, w)
```

Description

`sweep` computes the complex frequency response function for a system of the form:

$$[\mathbf{K} + i\omega\mathbf{C} - \omega^2\mathbf{M}]\mathbf{y}(\omega) = \mathbf{p}$$

Here, `K`, `C`, and `M` represent the *m*-by-*m* stiffness, damping, and mass matrices, respectively. The vector `p` defines the amplitude of the force. The frequency response function is computed for the values of ω given by the vector `w`.

The complex frequency response function is stored in the matrix `Y` with dimension *m*-by-*n*, where *n* is equal to the number of circular frequencies defined in `w`.

Example

The steady-state response can be computed by:

```
X = real(Y * exp(i * w * t));
```

and the amplitude by:

```
Z = abs(Y)
```

Chapter 5

Graphics functions

The group of graphics functions comprises functions for element based graphics. Mesh plots, displacements, section forces, flows, iso lines and principal stresses can be displayed. The functions are divided into two dimensional, and general graphics functions.

5.1 Two dimensional graphics functions

5.1.1 dispbeam2

Purpose

Draw the displacements for a two dimensional beam element.

Syntax

```
sfac = cfc.dispbeam2(ex, ey, edi)
sfac = cfc.dispbeam2(ex, ey, edi, plotpar)
cfc.dispbeam2(ex, ey, edi, plotpar, sfac)
```

Description

Input variables are the coordinate matrices `ex` and `ey`, see e.g. `beam2e`, and the element displacements `edi` obtained by e.g. `beam2s`.

The variable `plotpar` sets plot parameters for linetype, linecolour and node marker:

`plotpar`$= [linetype \;\; linecolor \;\; nodemark]$

where

linetype		*linecolor*		*nodemar*	
1	solid line	1	black	1	circle
2	dashed line	2	blue	2	asterisk
3	dotted line	3	magenta	0	no mark
		4	red		

Default is dashed black lines with circles at nodes.

The scale factor `sfac` is a scalar that the element displacements are multiplied with to get a suitable geometrical representation. If `sfac` is omitted in the input list, the scale factor is set automatically.

5.1.2 eldraw2

Purpose

Draw the undeformed mesh for a two dimensional structure.

Syntax

```
cfc.eldraw2(Ex, Ey)
cfc.eldraw2(Ex, Ey, plotpar)
cfc.eldraw2(Ex, Ey, plotpar, elnum)
```

Description

`eldraw2` displays the undeformed mesh for a two dimensional structure.

Input variables are the coordinate matrices `Ex` and `Ey` formed by the function `coordxtr`.

The variable `plotpar` sets plot parameters for linetype, linecolor and node marker:

`plotpar`= [linetype linecolor nodemark]

where

linetype		*linecolor*		*nodemar*	
1	solid line	1	black	1	circle
2	dashed line	2	blue	2	asterisk
3	dotted line	3	magenta	0	no mark
		4	red		

Default is solid black lines with circles at nodes.

Element numbers can be displayed at the center of the element if a column vector `elnum` with the element numbers is supplied. This column vector can be derived from the element topology matrix `Edof`:

```
elnum = texttt{Edof}(:,1)
```

i.e. the first column of the topology matrix.

ENLIGT EXEMPEL
INNÅR INTE elnum I Edof.
EJ FRIHETSGRADER ?

Limitations

Supported elements are bar, beam, triangular three node, and quadrilateral four node elements.

5.1.3 eldisp2

Purpose

Draw the deformed mesh for a two dimensional structure.

Syntax

```
sfac = cfc.eldisp2(Ex, Ey, Ed)
sfac = cfc.eldisp2(Ex, Ey, Ed, plotpar)
cfc.eldisp2(Ex, Ey, Ed, plotpar, sfac)
```

Description

`eldisp2` displays the deformed mesh for a two dimensional structure.

Input variables are the coordinate matrices `Ex` and `Ey` formed by the function `coordxtr`, and the element displacements `Ed` formed by the function `extract_ed`.

The variable `plotpar` sets plot parameters for linetype, linecolor and node marker:

`plotpar`$= [linetype \quad linecolor \quad nodemark]$

where

linetype		*linecolor*		*nodemar*	
1	solid line	1	black	1	circle
2	dashed line	2	blue	2	asterisk
3	dotted line	3	magenta	0	no mark
		4	red		

Default is dashed black lines with circles at nodes.

The scale factor `sfac` is a scalar that the element displacements are multiplied with to get a suitable geometrical representation. The scale factor is set automatically if it is omitted in the input list.

Limitations

Supported elements are bar, beam, triangular three node, and quadrilateral four node elements.

5.1.4 elflux2

Purpose

Draw element flow arrows for two dimensional elements.

Syntax

```
sfac = cfc.elflux2(Ex, Ey, Es)
sfac = cfc.elflux2(Ex, Ey, Es, plotpar)
cfc.elflux2(Ex, Ey, Es, plotpar, sfac)
```

Description

`elflux2` displays element heat flux vectors (or corresponding quantities) for a number of elements of the same type. The flux vectors are displayed as arrows at the element centroids. Note that only the flux vectors are displayed. To display the element mesh, use `eldraw2`.

Input variables are the coordinate matrices `Ex` and `Ey`, and the element flux matrix `Es` defined in `flw2ts` or `flw2qs`.

The variable `plotpar` sets plot parameters for the flux arrows:

`plotpar`= $[arrowtype \;\; arrowcolor]$

where

arrowtype		*arrowcolor*	
1	solid line	1	black
2	dashed line	2	blue
3	dotted line	3	magenta
		4	red

Default, if `plotpar` is omitted, is solid black arrows.

The scale factor `sfac` is a scalar that the values are multiplied with to get a suitable arrow size in relation to the element size. The scale factor is set automatically if it is omitted in the input list.

Limitations

Supported elements are triangular 3 node and quadrilateral 4 node elements.

5.1.5 eliso2

Purpose

Display element iso lines for two dimensional elements.

Syntax

```
cfc.eliso2(Ex, Ey, Ed, isov)
cfc.eliso2(Ex, Ey, Ed, isov, plotpar)
```

Description

`eliso2` displays element iso lines for a number of elements of the same type. Note that only the iso lines are displayed. To display the element mesh, use `eldraw2`.

Input variables are the coordinate matrices `Ex` and `Ey` formed by the function `coordxtr`, and the element nodal quantities (e.g., displacement or energy potential) matrix `Ed` defined in `extract`.

If `isov` is a scalar, it determines the number of iso lines to be displayed. If `isov` is a vector, it determines the values of the iso lines to be displayed (number of

iso lines equal to the length of vector `isov`):

`isov`= [iso lines]

`isov`= [isovalue(1) ... isovalue(n)]

The variable `plotpar` sets plot parameters for the iso lines:

`plotpar`= [$linetype$ $linecolor$ $textfcn$]

where

$linetyp$		$linecol$		$textfcn$	
1	solid line	1	black	0	iso values not printed
2	dashed line	2	blue	1	iso values printed at the iso lines
3	dotted line	3	magenta	1	iso values printed where the cursor indicates
		4	red		

Default is solid, black lines and no iso values printed.

Limitations

Supported elements are triangular 3 node and quadrilateral 4 node elements.

5.1.6 elprinc2

Purpose

Draw element principal stresses as arrows for two dimensional elements.

Syntax

```
sfac = cfc.elprinc2(Ex, Ey, Es)
sfac = cfc.elprinc2(Ex, Ey, Es, plotpar)
cfc.elprinc2(Ex, Ey, Es, plotpar, sfac)
```

Description

`elprinc2` displays element principal stresses for a number of elements of the same type. The principal stresses are displayed as arrows at the element centroids. Note that only the principal stresses are displayed. To display the element mesh, use `eldraw2`.

Input variables are the coordinate matrices `Ex` and `Ey`, and the element stresses matrix `Es` defined in `plants` or `planqs`.

The variable `plotpar` sets plot parameters for the principal stress arrows:

`plotpar`= [$arrowtype$ $arrowcolor$]

where

arrowtype		*arrowcolor*	
1	solid line	1	black
2	dashed line	2	blue
3	dotted line	3	magenta
		4	red

Default, if `plotpar` is omitted, is solid black arrows.

The scale factor `sfac` is a scalar that values are multiplied with to get a suitable arrow size in relation to the element size. The scale factor is set automatically if it is omitted in the input list.

Limitations

Supported elements are triangular 3 node and quadrilateral 4 node elements.

5.1.7 scalfact2

Purpose

Determine scale factor for drawing computational results.

Syntax

```
sfac = cfc.scalfact2(ex, ey, ed)
sfac = cfc.scalfact2(ex, ey, ed, rat)
```

Description

`scalfact2` determines a scale factor `sfac` for drawing computational results, such as displacements, section forces, or flux.

Input variables are the coordinate matrices `ex` and `ey`, and the matrix `ed` containing the quantity to be displayed. The scalar `rat` defines the ratio between the geometric representation of the largest quantity to be displayed and the element size. If `rat` is not specified, `0.2` is used.

Theory

The scale factor `sfac` is computed so that the largest value in `ed` is represented as a fraction `rat` of the element size, ensuring a visually appropriate scaling of computational results.

5.1.8 scalgraph2

Purpose

Draw a graphic scale.

Syntax

```
cfc.scalgraph2(sfac, magnitude)
cfc.scalgraph2(sfac, magnitude, plotpar)
```

Description

`scalgraph2` draws a graphic scale to visualize the magnitude of displayed computational results. The input variable `sfac` is a scale factor determined by the function `scalfact2`. The variable `magnitude` is defined as $[S \quad x \quad y]$, where S specifies the value corresponding to the length of the graphic scale, and (x, y) are the coordinates of the starting point. If no coordinates are given, the starting point will be $(0, -0.5)$.

Theory

The variable `plotpar` sets the graphic scale color:

`plotpar`$= [color]$

where

color	
1	black
2	blue
3	magenta
4	red

5.1.9 secforce2

Purpose

Draw the section force diagrams of a two dimensional bar or beam element in its global position.

Syntax

```
cfc.secforce2(ex, ey, es, plotpar, sfac)
cfc.secforce2(ex, ey, es, plotpar, sfac, eci)
sfac = cfc.secforce2(ex, ey, es)
sfac = cfc.secforce2(ex, ey, es, plotpar)
```

Description

The input variables `ex` and `ey` are defined in `bar2e` or `beam2e`. The input variable

`es`$= \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_n \end{bmatrix}$

consists of a column matrix that contains section forces. The values in `es` are computed in, e.g., `bar2s` or `beam2s`.

The variable `plotpar` sets plot parameters for the diagram:

`plotpar`$= [$linecolor elementcolor$]$

where

linecolor	color	element-color	color
1	black	1	black
2	blue	2	blue
3	magenta	3	magenta
4	red	4	red

The scale factor `sfac` is a scalar that the section forces are multiplied with to get a suitable graphical representation. If `sfac` is omitted in the input list, the scale factor is set automatically.

The input variable

$$\texttt{eci}= \begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_n \end{bmatrix}$$

specifies the local $\bar{x}$-coordinates of the quantities in `es`. If `eci` is not given, uniform distance is assumed.

Chapter 6

General purpose functions

The following are general purpose functions for managing variables, workspace, and output in IPython or Python interactive sessions. These are analogous to MATLAB's general functions.

6.1 General functions

6.1.1 help

Purpose

Display documentation for a specific function or object.

Syntax

```
help(function_name)
function_name?
function_name??
```

Description

`help(function_name)` displays the docstring/documentation for the specified function. `function_name?` shows a summary in IPython. `function_name??` shows the full docstring and source if available.

Example

Typing

```
help(len)
len?
len??
```

yields documentation for the `len` function.

Note

In IPython, `?` and `??` are convenient shortcuts.

6.1.2 type

Purpose

Show the type of an object.

Syntax

```
type(obj)
```

Description

`type(obj)` returns the type of the object.

Note

To display the source code of a function in IPython, use `function_name??`.

6.1.3 what

Purpose

List variables in the current namespace.

Syntax

```
%who
%whos
```

Description

`%who` lists variable names in the interactive namespace. `%whos` gives a detailed list with type and info.

Note

These are IPython magic commands.

6.1.4 ...

Purpose

Line continuation.

Syntax

```
...
```

Description

In Python, `...` is used as an Ellipsis object or for line continuation in some contexts (e.g., multi-line lambdas or function definitions). For line continuation, use the backslash :code:`\` or parentheses.

Note

In IPython, multi-line statements are handled automatically.

6.1.5

Purpose

Write a comment line.

Syntax

```
# arbitrary text
```

Description

Any text after # is a comment.

Note

This is standard Python syntax.

6.2 Variables and workspace

6.2.1 clear

Purpose

Remove variables from workspace.

Syntax

```
del var1, var2
%reset
```

Description

`del var1` deletes a variable. `%reset` (IPython magic) clears all variables (asks for confirmation).

Note

Use with caution; `%reset` cannot be undone.

6.2.2 disp

Purpose

Display a variable.

Syntax

```
print(A)
display(A)  # in Jupyter/IPython
```

Description

`print(A)` prints the value of `A`. `display(A)` (from `IPython.display`) can be used for rich display in Jupyter.

6.2.3 load

Purpose

Load variables from disk.

Syntax

```
import numpy as np
A = np.load('filename.npy')
data = np.loadtxt('filename.txt')
```

Description

Use `np.load` for binary NumPy files, `np.loadtxt` or `pandas.read_csv` for text files.

Note

There is no direct equivalent to MATLAB's `load` for .mat files, but `scipy.io.loadmat` can be used for MATLAB files.

6.2.4 save

Purpose

Save variables to disk.

Syntax

```
np.save('filename.npy', A)
np.savetxt('filename.txt', A)
```

Description

Use `np.save` for binary, `np.savetxt` for text files. For multiple variables, consider using `np.savez` or `pickle`.

6.2.5 who, whos

Purpose

List variables in the workspace.

Syntax

```
%who
%whos
```

Description

`%who` lists variable names. `%whos` gives detailed info.

Note

These are IPython magic commands.

6.3 Files and command window

6.3.1 diary

Purpose

Save session output to a file.

Syntax

```
%logstart filename
%logstop
```

Description

`%logstart filename` begins logging input/output to a file. `%logstop` stops logging.

Note

These are IPython magic commands.

6.3.2 echo

Purpose

Not directly applicable in Python. For script debugging, use print statements or logging.

Note

No direct equivalent. Use `print` or the `logging` module.

6.3.3 format

Purpose

Control output display format.

Syntax

```
# For NumPy arrays:
np.set_printoptions(precision=5)
np.set_printoptions(precision=15, suppress=True)
```

Description

Use `np.set_printoptions` to control NumPy array display precision.

Note

For pandas DataFrames, use `pd.set_option`.

6.3.4 quit

Purpose

Terminate session.

Syntax

```
exit()
quit()
```

Description

`exit()` or `quit()` exits the Python interpreter or IPython session.

Note

In Jupyter, use `exit()` in a cell.

Chapter 7

Statements and macros

Statements describe algorithmic actions that can be executed. There are two different types of control statements, conditional and repetitive. The first group defines conditional jumps whereas the latter one defines repetition until a conditional statement is fulfilled. In Python, functions and modules are used to define new functionality or to store a sequence of statements in .py files.

7.1 Control statements

`if`	Conditional jump
`for`	Iterate over sequences
`while`	Define a conditional loop

7.2 Macros and Functions

`def`	Define a new function
`module`	Store functions and statements

7.2.1 if

Purpose

Conditional jump.

Syntax

```
if condition:
    statements
elif condition:
    statements
else:
    statements
```

Description

The `if` statement allows conditional execution of code blocks. The `condition` is evaluated and if `True`, the corresponding `statements` are executed.

Examples

Simple if statement:

```
if x > 0:
    y = np.sqrt(x)
else:
    y = 0
```

Multiple conditions:

```
if x > 0:
    result = 'positive'
elif x < 0:
    result = 'negative'
else:
    result = 'zero'
```

Note

Python uses `elif` instead of `elseif`, and requires colons and indentation instead of `end`.

7.2.2 for

Purpose

Iterate over sequences.

Syntax

```
for variable in iterable:
    statements
```

Description

The `for` loop iterates over elements in an `iterable` (list, array, range, etc.). The `variable` takes on each value from the iterable in sequence.

Examples

Simple counting loop:

```
for i in range(1, 11):  # 1 to 10
    print(f'Iteration {i}')
```

Loop over array elements:

```
A = np.array([1, 4, 9, 16])
for element in A:
    print(element)
```

Nested loops:

```
A = np.zeros((3, 3))
for i in range(3):
    for j in range(3):
        A[i, j] = i + j
```

Note

Python uses `range()` for numeric sequences and requires colons and indentation.

7.2.3 while

Purpose

Define a conditional loop.

Syntax

```
while condition:
    statements
```

Description

The `while` loop repeats `statements` as long as the `condition` remains `True`.

Examples

Simple while loop:

```
i = 1
while i <= 10:
    print(f'Count: {i}')
    i += 1
```

Convergence loop:

```
error = 1
tolerance = 1e-6
while error > tolerance:
    # ... computation ...
    error = abs(new_value - old_value)
```

Note

Be careful to ensure the condition eventually becomes `False` to avoid infinite loops.

7.2.4 def

Purpose

Define a new function.

Syntax

```
def function_name(input1, input2, ...):
    statements
    return output1, output2, ...
```

Description

The `def` keyword defines a new function with specified inputs. Use `return` to specify outputs. Functions can be defined in .py files or interactively.

Examples

Simple function:

```
def square(x):
    return x**2
```

Multiple inputs and outputs:

```
def add_subtract(a, b):
    sum_val = a + b
    diff_val = a - b
    return sum_val, diff_val
```

Function with no return value:

```
def plot_data(x, y):
    plt.figure()
    plt.plot(x, y)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()
```

Note

Python functions use `def` and `return`. Multiple return values are returned as tuples.

7.2.5 module

Purpose

Store functions and statements.

Syntax

```
# filename: my_module.py
"""Module documentation"""

import numpy as np

def function1():
    # function definition
    pass

# module-level code
statements
```

Description

Modules are .py files that contain Python code. They can include functions, classes, and executable statements. Use `import` to access module contents.

Examples

Simple module (saved as `analysis.py`):

```
"""Data analysis module"""
import numpy as np
import matplotlib.pyplot as plt

def load_data(filename):
    return np.loadtxt(filename)

def plot_results(data):
    plt.figure()
    plt.plot(data)
    plt.show()

# Module-level code (runs when imported)
print("Analysis module loaded")
```

Using the module:

```
import analysis

data = analysis.load_data('data.txt')
analysis.plot_results(data)
```

Alternative import:

```
from analysis import load_data, plot_results

data = load_data('data.txt')
plot_results(data)
```

Note

Modules create their own namespace. Use `import` statements to access module contents.

Chapter 8

Matrix functions

The group of matrix functions comprises functions for vector and matrix operations and also functions for sparse matrix handling. In Python, these operations are primarily handled by NumPy arrays and SciPy sparse matrices. NumPy provides dense arrays while SciPy provides sparse matrix functionality. Unlike MATLAB, Python distinguishes between different sparse matrix formats (CSR, CSC, COO, etc.) for different use cases.

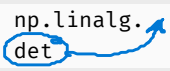
If you want use the functions described in this chapter, you need to import the NumPy and SciPy libraries. It is common practice to import these libraries with the following aliases:

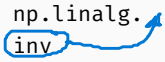
```
import numpy as np
import scipy.sparse as sp
import matplotlib.pyplot as plt
```

The following functions are described in this chapter:

Vector and matrix operations

`[ ] ( ) =`	Special characters
`' , ;`	Special characters
`:`	Create arrays and do array indexing
`+ - * /`	Array arithmetic
`np.abs`	Absolute value
`np.linalg.det`	Matrix determinant
`np.diag`	Diagonal arrays and diagonals of an array
`np.linalg.inv`	Matrix inverse
`len`	Array length
`np.max`	Maximum element(s) of an array
`np.min`	Minimum element(s) of an array
`np.ones`	Generate an array of all ones
`np.shape`	Array dimensions
`np.sqrt`	Square root
`np.sum`	Sum of the elements of an array
`np.zeros`	Generate a zero array





Sparse matrix handling

8.1 Special characters [ ] ( ) = ‘ , ;

Purpose

Special characters for array operations.

Syntax

```
[ ] ( ) = ' , ;
```

Description

[ ]

Brackets are used to create lists and arrays, and for indexing.

( )

Parentheses are used to indicate precedence in arithmetic expressions, for function calls, and to create tuples.

=

Used in assignment statements.

‘

String delimiter. For matrix transpose, use `.T` or `np.transpose()`.

,

Comma. Used to separate array indices and function arguments.

;
: Semicolon. Used to separate statements on the same line (not commonly used in Python).

Examples

By the statement:

```
a = 2
```

the scalar `a` is assigned a value of 2. An element in an array may be assigned a value according to:

```
A[1, 4] = 3  # Note: Python uses 0-based indexing
```

The statement:

```
D = np.array([[1, 2], [3, 4]])
```

results in array:

$$\mathbf{D} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

stored in memory. To copy the contents of the array `D` to an array `E`, use:

```
E = D.copy()  # or E = np.copy(D)
```

The transpose of array `A` is stored in a new array `F` using:

```
F = A.T  # or F = np.transpose(A)
```

Note

These are Python built-in syntax elements. NumPy must be imported as `import numpy as np`.

8.2 : (slice operator) and np.arange

Purpose

Create arrays and do array indexing/slicing.

Description

In Python with NumPy, array creation and slicing use different syntax:

np.arange(j, k)
: creates array [`j, j + 1, ... , k-1`] (note: k is excluded)

np.arange(j, k, i)
: creates array [`j, j + i, j + 2i, ... `` ] up to but not including ``k`

The colon notation is used for array slicing:

A[: , j]
is the `j` :th column of `A`

A[i , :]
is the `i` :th row of `A`

Examples

To create a sequence similar to MATLAB's `1:4`:

```
d = np.arange(1, 5)  # Creates [1, 2, 3, 4]
```

results in a 1D array:

$$\mathbf{d} = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

stored in memory.

Array slicing is used to access selected rows and columns. For example, if we have created a 3×4 array `D`:

```
D = np.array([d, 2*d, 3*d])
```

resulting in:

$$\mathbf{D} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

columns two and three (indices 2:4) are accessed by:

```
D[:, 2:4]  # Note: 0-based indexing, end index excluded
```

resulting in:

$$\mathbf{D}[:, 2:4] = \begin{bmatrix} 3 & 4 \\ 6 & 8 \\ 9 & 12 \end{bmatrix}$$

To copy parts of array `D` into another array, use slicing:

```
E[2:4, 1:3] = D[0:2, 2:4]  # 0-based indexing
```

Assuming array `E` was a zero array before the statement, the result will be:

$$\mathbf{E} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 6 & 8 & 0 \end{bmatrix}$$

Note

These are Python/NumPy built-in operations. Note the 0-based indexing and exclusive end indices.

- – * / @ (arithmetic operators)

Purpose

Array arithmetic operations.

Syntax

```
A + B
A - B
A * B    # element-wise multiplication
A @ B    # matrix multiplication
A / s    # element-wise division
A // s   # floor division
```

Description

NumPy array operations can be element-wise or matrix operations. Use `@` for matrix multiplication and `*` for element-wise multiplication.

Examples

An example of a sequence of array-to-array operations is:

```
D = A + B - C
```

A matrix-to-vector multiplication followed by a vector-to-vector subtraction may be defined by the statement:

```
b = c - A @ x  # Use @ for matrix multiplication
```

and finally, to scale an array by a scalar `s` we may use:

```
B = A / s
```

Note

These are NumPy array operations. For matrix multiplication, use `@` or `np.dot()`.

8.3 np.abs

Purpose

Absolute value.

Syntax

```
B = np.abs(A)
```

Description

`B = np.abs(A)` computes the absolute values of the elements of array `A` and stores them in array `B`.

Examples

Assume the array:

$$\mathbf{C} = \begin{bmatrix} -7 & 4 \\ -3 & -8 \end{bmatrix}$$

The statement:

```
D = np.abs(C)
```

results in an array:

$$\mathbf{D} = \begin{bmatrix} 7 & 4 \\ 3 & 8 \end{bmatrix}$$

stored in memory.

Note

This is a NumPy function. Alternative: `np.absolute()`. For more information, see NumPy documentation.

8.4 np.linalg.det

Purpose

Matrix determinant.

Syntax

```
a = np.linalg.det(A)
```

Description

`a = np.linalg.det(A)` computes the determinant of the square array `A` and stores it in the scalar `a`.

Note

This is a NumPy linear algebra function. For more information, see NumPy documentation.

8.5 np.diag

Purpose

Diagonal arrays and diagonals of an array.

Syntax

```
M = np.diag(v)
v = np.diag(M)
```

Description

For a 1D array `v` with n elements, the statement `M = np.diag(v)` results in an $n \times n$ array `M` with the elements of `v` as the main diagonal.

For a 2D array `M`, the statement `v = np.diag(M)` results in a 1D array `v` formed by the main diagonal of `M`.

Note

This is a NumPy function. Use `np.diagflat()` for a more general version that works with multi-dimensional input.

8.6 .toarray() / .todense()

Purpose

Convert sparse matrices to dense arrays.

Syntax

```
A = S.toarray()    # Convert to NumPy array
A = S.todense()    # Convert to matrix (deprecated)
```

Description

`A = S.toarray()` converts a sparse matrix `S` to a dense NumPy array `A`.

`A = S.todense()` converts to a matrix object (deprecated, use `toarray()`).

Note

These are methods of scipy.sparse matrices. Use `toarray()` for better compatibility with NumPy operations.

8.7 np.linalg.inv

Purpose

Matrix inverse.

Syntax

```
B = np.linalg.inv(A)
```

Description

`B = np.linalg.inv(A)` computes the inverse of the square array `A` and stores the result in the array `B`.

Note

This is a NumPy linear algebra function. For better numerical stability, consider using `np.linalg.pinv()` for pseudo-inverse or `scipy.linalg.solve()` for solving linear systems.

8.8 len / np.size

Purpose

Array length.

Syntax

```
n = len(x)        # For 1D arrays
n = np.size(x)    # Total number of elements
n = x.size        # Total number of elements
```

Description

`n = len(x)` returns the length of the first dimension of array `x`.

`n = np.size(x)` returns the total number of elements in array `x`.

Note

`len()` is a Python built-in function. `np.size()` is a NumPy function that's more similar to MATLAB's `length()`.

8.9 np.max

Purpose

Maximum element(s) of an array.

Syntax

```
b = np.max(A)           # Maximum of entire array
b = np.max(A, axis=0)   # Maximum along columns
b = np.max(A, axis=1)   # Maximum along rows
```

Description

For a 1D array `a`, `b = np.max(a)` returns the maximum element.

For a 2D array `A`, `b = np.max(A, axis=0)` returns an array containing the maximum elements found in each column.

The maximum element in an array may be found by `c = np.max(A)`.

Examples

Assume the array `B` is defined as:

$$\mathbf{B} = \begin{bmatrix} -7 & 4 \\ -3 & -8 \end{bmatrix}$$

The statement:

```
d = np.max(B, axis=0)
```

results in an array:

$$\mathbf{d} = \begin{bmatrix} -3 & 4 \end{bmatrix}$$

The maximum element in array `B` may be found by `e = np.max(B)` which results in the scalar `e = 4`.

Note

This is a NumPy function. Alternative: `np.amax()` or `A.max()`.

8.10 np.min

Purpose

Minimum element(s) of an array.

Syntax

```
b = np.min(A)           # Minimum of entire array
b = np.min(A, axis=0)   # Minimum along columns
b = np.min(A, axis=1)   # Minimum along rows
```

Description

For a 1D array `a`, `b = np.min(a)` returns the minimum element.

For a 2D array `A`, `b = np.min(A, axis=0)` returns an array containing the minimum elements found in each column.

The minimum element in an array may be found by `c = np.min(A)`.

Examples

Assume the array `B` is defined as:

$$\mathbf{B} = \begin{bmatrix} -7 & 4 \\ -3 & -8 \end{bmatrix}$$

The statement:

```
d = np.min(B, axis=0)
```

results in an array:

$$\mathbf{d} = \begin{bmatrix} -7 & -8 \end{bmatrix}$$

The minimum element in array `B` is found by `e = np.min(B)`, which results in the scalar `e = -8`.

Note

This is a NumPy function. Alternative: `np.amin()` or `A.min()`.

8.11 np.ones

Purpose

Generate an array of all ones.

Syntax

```
A = np.ones((m, n))
A = np.ones(shape)
```

Description

`A = np.ones((m, n))` results in an `m``×``n` array `A` with all ones.

`A = np.ones(shape)` creates an array with the specified shape filled with ones.

Note

This is a NumPy function. Note the tuple syntax for shape `(m, n)`.

8.12 np.shape / .shape

Purpose

Array dimensions.

Syntax

```
d = np.shape(A)
d = A.shape
m, n = A.shape
```

Description

`d = np.shape(A)` or `d = A.shape` returns a tuple with the dimensions of array `A`.

`m, n = A.shape` unpacks the dimensions `m` and `n` of the 2D array `A`.

Note

`.shape` is an array attribute. `np.shape()` is the function equivalent.

8.13 scipy.sparse

Purpose

Create sparse matrices.

Syntax

```
import scipy.sparse as sp
S = sp.csr_matrix(A)        # Convert dense to sparse (CSR format)
S = sp.coo_matrix(A)        # Convert to COO format
S = sp.csr_matrix((m, n))   # Create sparse zero matrix
```

Description

`S = sp.csr_matrix(A)` converts a dense array to sparse CSR (Compressed Sparse Row) format.

`S = sp.csr_matrix((m, n))` creates an m×n sparse zero matrix in CSR format.

Other formats available: `csc_matrix` (CSC), `coo_matrix` (COO), `lil_matrix` (LIL), etc.

Note

This requires SciPy. Different sparse formats are optimized for different operations (CSR for row operations, CSC for column operations, etc.).

8.14 plt.spy

Purpose

Visualize matrix sparsity structure.

Syntax

```
import matplotlib.pyplot as plt
plt.spy(S)
plt.show()
```

Description

`plt.spy(S)` plots the sparsity pattern of array or sparse matrix `S`. Nonzero elements are displayed as markers.

Additional parameters like `markersize`, `marker`, etc., can be used to customize the plot.

Note

This is a matplotlib function. Works with both dense and sparse arrays. For sparse matrices, it's more efficient than converting to dense first.

8.15 np.sqrt

Purpose

Square root.

Syntax

```
B = np.sqrt(A)
```

Description

`B = np.sqrt(A)` computes the element-wise square root of array `A` and stores the result in array `B`.

Note

This is a NumPy function. For complex inputs, it computes the principal square root.

8.16 np.sum

Purpose

Sum of the elements of an array.

Syntax

```
b = np.sum(A)           # Sum of all elements
b = np.sum(A, axis=0)   # Sum along columns
b = np.sum(A, axis=1)   # Sum along rows
```

Description

For a 1D array `a`, `b = np.sum(a)` returns the sum of all elements.

For a 2D array `A`, `b = np.sum(A, axis=0)` returns an array containing the sum of elements in each column.

The sum of all elements in an array is determined by `c = np.sum(A)`.

Note

This is a NumPy function. Alternative: `A.sum()` method.

8.17 np.zeros

Purpose

Generate a zero array.

Syntax

```
A = np.zeros((m, n))
A = np.zeros(shape)
```

Description

`A = np.zeros((m, n))` results in an `m``×``n` array `A` of zeros.

`A = np.zeros(shape)` creates an array with the specified shape filled with zeros.

Note

This is a NumPy function. Note the tuple syntax for shape `(m, n)`.

Chapter 9

Examples

9.1 exs_spring

Purpose

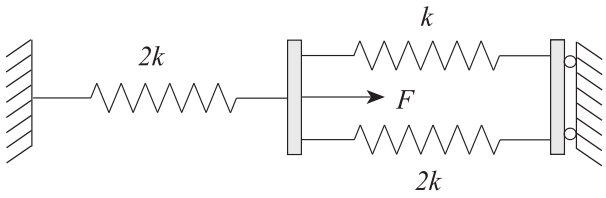
Show the basic steps in a finite element calculation.

Description

The general procedure in linear finite element calculations is carried out for a simple structure. The steps are:

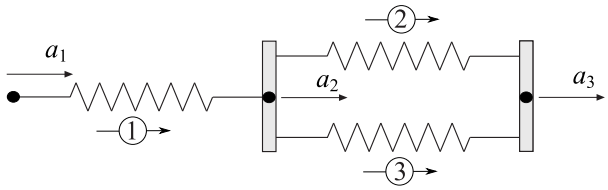
- Define the model
- Generate element matrices
- Assemble element matrices into the global system of equations
- Solve the global system of equations
- Evaluate element forces

Consider the system of three linear elastic springs, and the corresponding finite element model. The system of springs is fixed in its ends and loaded by a single load F.



Example

The computation is initialized by importing CALFEM and NumPy, then defining the topology matrix `Edof` containing the degrees of freedom for each element:




```
>>> import numpy as np
>>> import calfem.core as cfc

>>> # Element topology matrix (no element numbers, just DOFs)
>>> Edof = np.array([
...     [1, 2],  # Element 1: DOF 1 to DOF 2
...     [2, 3],  # Element 2: DOF 2 to DOF 3
...     [2, 3]   # Element 3: DOF 2 to DOF 3
... ])
```

The global stiffness matrix `K` (3×3) of zeros:

```
>>> K = np.zeros((3, 3))
>>> print(K)
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

And the load vector `f` (3×1) with the load $F = 100$ at DOF 2:

```
>>> f = np.zeros(3)
>>> f[1] = 100  # Load at DOF 2 (index 1 in 0-based indexing)
>>> print(f)
[  0. 100.   0.]
```

Element stiffness matrices are generated by the function `cfc.spring1e`. The element property `ep` for the springs contains the spring stiffnesses k and $2k$ respectively, where $k = 1500$:

```
>>> k = 1500
>>> ep1 = k      # Spring stiffness for elements 2
>>> ep2 = 2 * k  # Spring stiffness for elements 1 and 3
>>>
>>> Ke1 = cfc.spring1e(ep1)
>>> print("Ke1 (k=1500):")
>>> print(Ke1)
[[ 1500. -1500.]
 [-1500.  1500.]]
>>>
>>> Ke2 = cfc.spring1e(ep2)
>>> print("Ke2 (k=3000):")
>>> print(Ke2)
[[ 3000. -3000.]
 [-3000.  3000.]]
```

The element stiffness matrices are assembled into the global stiffness matrix `K` according to the topology:

```
>>> # Assemble element 1 (uses Ke2 with stiffness 3000)
>>> K = cfc.assem(Edof[0, :], K, Ke2)
>>> print("After assembling element 1:")
```

```
>>> print(K)
[[ 3000. -3000.     0.]
 [-3000.  3000.     0.]
 [    0.     0.     0.]]
>>>
>>> # Assemble element 2 (uses Ke1 with stiffness 1500)
>>> K = cfc.assem(Edof[1, :], K, Ke1)
>>> print("After assembling element 2:")
>>> print(K)
[[ 3000. -3000.     0.]
 [-3000.  4500. -1500.]
 [    0. -1500.  1500.]]
>>>
>>> # Assemble element 3 (uses Ke2 with stiffness 3000)
>>> K = cfc.assem(Edof[2, :], K, Ke2)
>>> print("After assembling element 3:")
>>> print(K)
[[ 3000. -3000.     0.]
 [-3000.  7500. -4500.]
 [    0. -4500.  4500.]]
```

The global system of equations is solved considering the boundary conditions. DOFs 1 and 3 are fixed (value 0):

```
>>> bc = np.array([[0, 0],   # DOF 1 = 0 (fixed)
...                [2, 0]])  # DOF 3 = 0 (fixed)
>>> a, r = cfc.solveq(K, f, bc)
>>> print("Displacements:")
>>> print(a)
[ 0.          0.01333333  0.        ]
>>> print("Reaction forces:")
>>> print(r)
[-40.   0. -60.]
```

Element forces are evaluated from the element displacements. These are obtained from the global displacements a using the function `cfc.extract_ed`:

```
>>> # Extract displacements for each element
>>> ed1 = cfc.extract_ed(Edof[0, :], a)  # Element 1
>>> ed2 = cfc.extract_ed(Edof[1, :], a)  # Element 2
>>> ed3 = cfc.extract_ed(Edof[2, :], a)  # Element 3
>>>
>>> print("Element displacements:")
>>> print("Element 1:", ed1)
Element 1: [ 0.          0.01333333]
>>> print("Element 2:", ed2)
Element 2: [ 0.01333333  0.        ]
>>> print("Element 3:", ed3)
Element 3: [ 0.01333333  0.        ]
```

The spring forces are evaluated using the function `cfc.spring1s`:

```
>>> # Calculate element forces
>>> es1 = cfc.spring1s(ep2, ed1)  # Element 1 uses ep2 (3000)
>>> es2 = cfc.spring1s(ep1, ed2)  # Element 2 uses ep1 (1500)
>>> es3 = cfc.spring1s(ep2, ed3)  # Element 3 uses ep2 (3000)
>>>
>>> print("Spring forces:")
>>> print("Element 1 force:", es1[0])
Element 1 force: 40.0
>>> print("Element 2 force:", es2[0])
Element 2 force: 20.0
>>> print("Element 3 force:", es3[0])
Element 3 force: 40.0
```

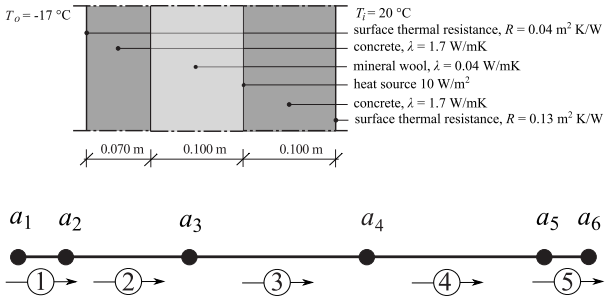
9.2 exs_flw_temp1

Purpose

Analysis of one-dimensional heat flow.

Description

Consider a wall built up of concrete and thermal insulation. The outdoor temperature is -17°C and the temperature inside is 20°C. At the inside of the thermal insulation there is a heat source yielding 10 W/m².



The wall is subdivided into five elements and the one-dimensional spring (analogy) element `spring1e` is used. Equivalent spring stiffnesses are $k_i = \lambda A/L$ for thermal conductivity and $k_i = A/R$ for thermal surface resistance. Corresponding spring stiffnesses per m² of the wall are:

$k_1 =$	$1/0.04$	$=$	25.0	W/K
$k_2 =$	$1.7/0.070$	$=$	24.3	W/K
$k_3 =$	$0.040/0.100$	$=$	0.4	W/K
$k_4 =$	$1.7/0.100$	$=$	17.0	W/K
$k_5 =$	$1/0.13$	$=$	7.7	W/K

Example

The computation is initialized by importing CALFEM and NumPy. A global system matrix `K` and a heat flow vector `f` are defined. The heat source inside the wall is considered by setting $f_4 = 10$. The element matrices `Ke` are computed using `cfc.spring1e`, and the function `cfc.assem` assembles the global stiffness matrix.

```
>>> import numpy as np
>>> import calfem.core as cfc

>>> # Element topology matrix (DOFs only, no element numbers)
>>> Edof = np.array([
...     [1, 2],  # Element 1: DOF 1 to DOF 2
...     [2, 3],  # Element 2: DOF 2 to DOF 3
...     [3, 4],  # Element 3: DOF 3 to DOF 4
...     [4, 5],  # Element 4: DOF 4 to DOF 5
...     [5, 6]   # Element 5: DOF 5 to DOF 6
... ])

>>> # Initialize global stiffness matrix and load vector
```

```
>>> K = np.zeros((6, 6))
>>> f = np.zeros(6)
>>> f[3] = 10  # Heat source at DOF 4 (index 3 in 0-based indexing)
>>> print("Load vector f:")
>>> print(f)
[ 0.  0.  0. 10.  0.  0.]
```

The thermal conductance values for each element are defined, and element stiffness matrices are computed:

```
>>> # Thermal conductance values for each element
>>> ep1 = 25.0    # Element 1: 1/0.04 = 25.0 W/K
>>> ep2 = 24.3    # Element 2: 1.7/0.070 = 24.3 W/K
>>> ep3 = 0.4     # Element 3: 0.040/0.100 = 0.4 W/K
>>> ep4 = 17.0    # Element 4: 1.7/0.100 = 17.0 W/K
>>> ep5 = 7.7     # Element 5: 1/0.13 = 7.7 W/K

>>> # Compute element stiffness matrices
>>> Ke1 = cfc.spring1e(ep1)
>>> Ke2 = cfc.spring1e(ep2)
>>> Ke3 = cfc.spring1e(ep3)
>>> Ke4 = cfc.spring1e(ep4)
>>> Ke5 = cfc.spring1e(ep5)
```

The element stiffness matrices are assembled into the global stiffness matrix:

```
>>> # Assemble global stiffness matrix
>>> K = cfc.assem(Edof[0, :], K, Ke1)  # Element 1
>>> K = cfc.assem(Edof[1, :], K, Ke2)  # Element 2
>>> K = cfc.assem(Edof[2, :], K, Ke3)  # Element 3
>>> K = cfc.assem(Edof[3, :], K, Ke4)  # Element 4
>>> K = cfc.assem(Edof[4, :], K, Ke5)  # Element 5
```

The system of equations is solved using `cfc.solveq` with boundary conditions. The prescribed temperatures are $T_1 = -17$°C and $T_6 = 20$°C:

```
>>> # Boundary conditions: DOF 1 = -17°C, DOF 6 = 20°C
>>> bc = np.array([[0, -17],   # DOF 1 (index 0) = -17°C
...                [5, 20]])   # DOF 6 (index 5) = 20°C

>>> # Solve system of equations
>>> a, r = cfc.solveq(K, f, bc)
>>>
>>> print("Temperatures at nodes:")
>>> print(a)
[-17.          -16.43835616  -15.86073059   19.23776824
↪  19.47534247
  20.        ]
>>>
>>> print("Reaction forces (boundary heat flows):")
>>> print(r)
[-14.03945205   0.           0.           0.           0.
   4.03945205]
```

The temperature values a_i at the node points are given in the vector `a` and the boundary heat flows in the vector `r`.

After solving the system of equations, the heat flow through each element is computed using `cfc.extract_ed` and `cfc.spring1s`:

```
>>> # Extract element displacements (temperature differences)
>>> ed1 = cfc.extract_ed(Edof[0, :], a)  # Element 1
>>> ed2 = cfc.extract_ed(Edof[1, :], a)  # Element 2
>>> ed3 = cfc.extract_ed(Edof[2, :], a)  # Element 3
>>> ed4 = cfc.extract_ed(Edof[3, :], a)  # Element 4
>>> ed5 = cfc.extract_ed(Edof[4, :], a)  # Element 5

>>> # Calculate heat flows through each element
>>> q1 = cfc.spring1s(ep1, ed1)
>>> q2 = cfc.spring1s(ep2, ed2)
>>> q3 = cfc.spring1s(ep3, ed3)
>>> q4 = cfc.spring1s(ep4, ed4)
>>> q5 = cfc.spring1s(ep5, ed5)

>>> print("Heat flows through elements:")
>>> print(f"Element 1: {q1[0]:.1f} W/m²")
Element 1: 14.0 W/m²
>>> print(f"Element 2: {q2[0]:.1f} W/m²")
Element 2: 14.0 W/m²
>>> print(f"Element 3: {q3[0]:.1f} W/m²")
Element 3: 14.0 W/m²
>>> print(f"Element 4: {q4[0]:.1f} W/m²")
Element 4: 4.0 W/m²
>>> print(f"Element 5: {q5[0]:.1f} W/m²")
Element 5: 4.0 W/m²
```

The heat flow through the wall is $q = 14.0$ W/m² in the part of the wall to the left of the heat source (elements 1-3), and $q = 4.0$ W/m² in the part to the right of the heat source (elements 4-5). This demonstrates how the internal heat source affects the heat flow distribution through the wall.

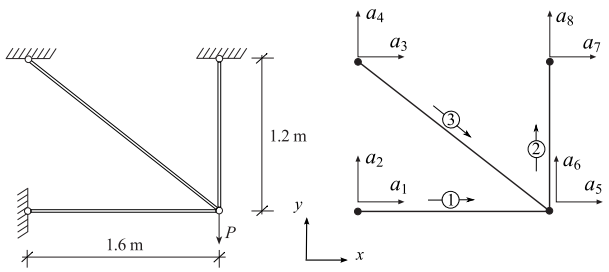
9.3 exs_bar2

Purpose

Analysis of a plane truss.

Description

Consider a plane truss consisting of three bars with the properties $E = 200$ GPa, $A_1 = 6.0 \times 10^{-4}$ m², $A_2 = 3.0 \times 10^{-4}$ m² and $A_3 = 10.0 \times 10^{-4}$ m², and loaded by a single force $P = 80$ kN. The corresponding finite element model consists of three elements and eight degrees of freedom.



Example

The computation is initialized by importing CALFEM and NumPy. The element topology matrix contains only the degrees of freedom for each element:

```
>>> import numpy as np
>>> import calfem.core as cfc

>>> # Element topology matrix (DOFs only, no element numbers)
>>> Edof = np.array([
...     [1, 2, 5, 6],  # Element 1: DOFs 1,2,5,6
...     [5, 6, 7, 8],  # Element 2: DOFs 5,6,7,8
...     [3, 4, 5, 6]   # Element 3: DOFs 3,4,5,6
... ])
```

The global stiffness matrix `K` and the load vector `f` are initialized. The load of 80 kN is applied in the negative y-direction at DOF 6:

```
>>> # Initialize global system
>>> K = np.zeros((8, 8))
>>> f = np.zeros(8)
>>> f[5] = -80e3  # Load at DOF 6 (index 5 in 0-based indexing)
>>> print("Load vector:")
>>> print(f)
[     0.      0.      0.      0.      0. -80000.      0.      0.]
```

The material and geometric properties are defined for each element:

```
>>> # Material and geometric properties
>>> E = 2.0e11  # Young's modulus [Pa]
>>> A1 = 6.0e-4   # Cross-sectional area element 1 [m²]
>>> A2 = 3.0e-4   # Cross-sectional area element 2 [m²]
>>> A3 = 10.0e-4  # Cross-sectional area element 3 [m²]
>>>
>>> ep1 = [E, A1]  # Element properties for element 1
>>> ep2 = [E, A2]  # Element properties for element 2
>>> ep3 = [E, A3]  # Element properties for element 3
```

The element coordinates are defined for each element (x and y coordinates of start and end nodes):

```
>>> # Element coordinates [m]
>>> ex1 = np.array([0, 1.6])    # Element 1 x-coordinates
>>> ey1 = np.array([0, 0])      # Element 1 y-coordinates
>>>
>>> ex2 = np.array([1.6, 1.6])  # Element 2 x-coordinates
>>> ey2 = np.array([0, 1.2])    # Element 2 y-coordinates
>>>
>>> ex3 = np.array([0, 1.6])    # Element 3 x-coordinates
>>> ey3 = np.array([1.2, 0])    # Element 3 y-coordinates
```

The element stiffness matrices `Ke1`, `Ke2` and `Ke3` are computed using `bar2e`:

```
>> Ke1=bar2e(ex1,ey1,ep1)

Ke1 =

  1.0e+007 *

    7.5000         0   -7.5000         0
         0         0         0         0
   -7.5000         0    7.5000         0
         0         0         0         0

>> Ke2=bar2e(ex2,ey2,ep2)

Ke2 =
```

```
  1.0e+007 *

         0         0         0         0
         0    5.0000         0   -5.0000
         0         0         0         0
         0   -5.0000         0    5.0000

>> Ke3=bar2e(ex3,ey3,ep3)

Ke3 =

  1.0e+007 *

    6.4000   -4.8000   -6.4000    4.8000
   -4.8000    3.6000    4.8000   -3.6000
   -6.4000    4.8000    6.4000   -4.8000
    4.8000   -3.6000   -4.8000    3.6000
```

The element stiffness matrices are computed and assembled into the global stiffness matrix:

```
>>> # Compute element stiffness matrices and assemble
>>> Ke1 = cfc.bar2e(ex1, ey1, ep1)
>>> K = cfc.assem(Edof[0], K, Ke1)
>>>
>>> Ke2 = cfc.bar2e(ex2, ey2, ep2)
>>> K = cfc.assem(Edof[1], K, Ke2)
>>>
>>> Ke3 = cfc.bar2e(ex3, ey3, ep3)
>>> K = cfc.assem(Edof[2], K, Ke3)
>>>
>>> print("Global stiffness matrix K:")
>>> print(K)
[[  7.5e+07   0.0e+00   0.0e+00   0.0e+00  -7.5e+07   0.0e+00
↪   0.0e+00   0.0e+00]
 [  0.0e+00   0.0e+00   0.0e+00   0.0e+00   0.0e+00   0.0e+00
↪   0.0e+00   0.0e+00]
 [  0.0e+00   0.0e+00   6.4e+07  -4.8e+07  -6.4e+07   4.8e+07
↪   0.0e+00   0.0e+00]
 [  0.0e+00   0.0e+00  -4.8e+07   3.6e+07   4.8e+07  -3.6e+07
↪   0.0e+00   0.0e+00]
 [ -7.5e+07   0.0e+00  -6.4e+07   4.8e+07   1.39e+08  -4.8e+07
↪   0.0e+00   0.0e+00]
 [  0.0e+00   0.0e+00   4.8e+07  -3.6e+07  -4.8e+07   8.6e+07
↪   0.0e+00  -5.0e+07]
 [  0.0e+00   0.0e+00   0.0e+00   0.0e+00   0.0e+00   0.0e+00
↪   0.0e+00   0.0e+00]
 [  0.0e+00   0.0e+00   0.0e+00   0.0e+00   0.0e+00  -5.0e+07
↪   0.0e+00   5.0e+07]]
```

"K =" behövs ej?

The system of equations $\boldsymbol{K}\boldsymbol{a} = \boldsymbol{f}$ is solved by specifying the boundary conditions and using `solveq()`:

```
>>> # Boundary conditions (DOF, prescribed value)
>>> bc = np.array([
...     [1, 0],  # DOF 1 = 0 (fixed in x)
...     [2, 0],  # DOF 2 = 0 (fixed in y)
...     [3, 0],  # DOF 3 = 0 (fixed in x)
...     [4, 0],  # DOF 4 = 0 (fixed in y)
...     [7, 0],  # DOF 7 = 0 (fixed in x)
...     [8, 0]   # DOF 8 = 0 (fixed in y)
... ])
>>>
>>> # Solve the system
```

Se kommentar exs_Spring ovan

```
>>> a, r = cfc.solveq(K, f, bc)
>>> print("Displacements [m]:")
>>> print(a)
[ 0.0e+00  0.0e+00  0.0e+00  0.0e+00 -3.98e-04 -1.152e-03  0.0e+00
↪  0.0e+00]
>>> print("Reaction forces [N]:")
>>> print(r)
[ 2.9845e+04  0.0e+00 -2.9845e+04  2.2383e+04  0.0e+00  0.0e+00
↪  0.0e+00  5.7617e+04]
```

The vertical displacement at the point of loading is 1.15 mm. The element forces are calculated by extracting element displacements and computing stresses:

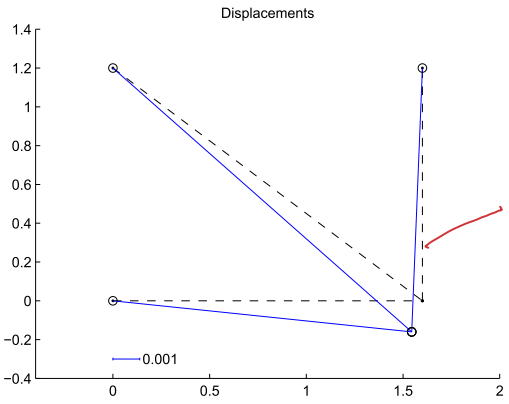
```
>>> # Extract element displacements and compute forces
>>> ed1 = cfc.extract_ed(Edof[0], a)
>>> es1 = cfc.bar2s(ex1, ey1, ep1, ed1)
>>> print(f"Element 1 forces [N]: {es1}")
Element 1 forces [N]: [-29845. -29845.]
>>>
>>> ed2 = cfc.extract_ed(Edof[1], a)
>>> es2 = cfc.bar2s(ex2, ey2, ep2, ed2)
>>> print(f"Element 2 forces [N]: {es2}")
Element 2 forces [N]: [57617. 57617.]
>>>
>>> ed3 = cfc.extract_ed(Edof[2], a)
>>> es3 = cfc.bar2s(ex3, ey3, ep3, ed3)
>>> print(f"Element 3 forces [N]: {es3}")
Element 3 forces [N]: [37306. 37306.]
```

The normal forces are $N_1 = -29.84$ kN (compression), $N_2 = 57.62$ kN (tension) and $N_3 = 37.31$ kN (tension).

Visualization of the results can be performed using CALFEM's visualization functions:

```
>>> import calfem.vis_mpl as cfv
>>> import matplotlib.pyplot as plt
>>>
>>> # Create displacement diagram
>>> cfv.figure(1)
>>>
>>> # Draw undeformed structure
>>> plotpar = [2, 1, 0]  # Line style, marker, color
>>> cfv.eldraw2(ex1, ey1, plotpar)
>>> cfv.eldraw2(ex2, ey2, plotpar)
>>> cfv.eldraw2(ex3, ey3, plotpar)
>>>
>>> # Calculate scale factor and draw deformed structure
>>> sfac = cfv.scalfact2(ex1, ey1, ed1, 0.1)
>>> plotpar = [1, 2, 1]  # Line style, marker, color for deformed
↪  shape
>>> cfv.eldisp2(ex1, ey1, ed1, plotpar, sfac)
>>> cfv.eldisp2(ex2, ey2, ed2, plotpar, sfac)
>>> cfv.eldisp2(ex3, ey3, ed3, plotpar, sfac)
>>>
>>> plt.axis([-0.4, 2.0, -0.4, 1.4])
>>> cfv.scalgraph2(sfac, [1e-3, 0, -0.3])
>>> plt.title('Displacements')
>>> plt.show()
>>>
>>> # Create normal force diagram
>>> cfv.figure(2)
>>> plotpar = [2, 1]  # Line style, marker
```

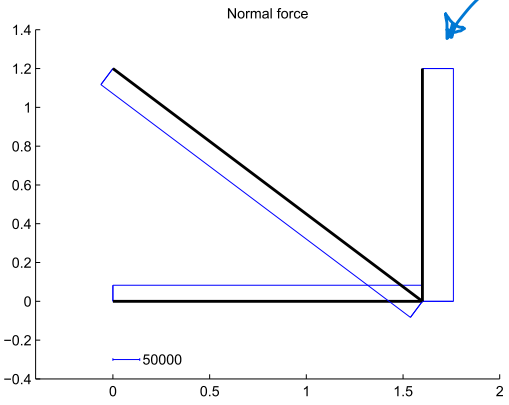
```
>>> sfac = cfv.scalfact2(ex1, ey1, es2[:, 0], 0.1)
>>> cfv.secforce2(ex1, ey1, es1[:, 0], plotpar, sfac)
>>> cfv.secforce2(ex2, ey2, es2[:, 0], plotpar, sfac)
>>> cfv.secforce2(ex3, ey3, es3[:, 0], plotpar, sfac)
>>>
>>> plt.axis([-0.4, 2.0, -0.4, 1.4])
>>> cfv.scalgraph2(sfac, [5e4, 0, -0.3])
>>> plt.title('Normal force')
>>> plt.show()
```



streckade linjer ser ibland helt dragna ut för mig när jag plottar i Python.

Displacement diagram

MATLAB-plottar?



Normal force diagram

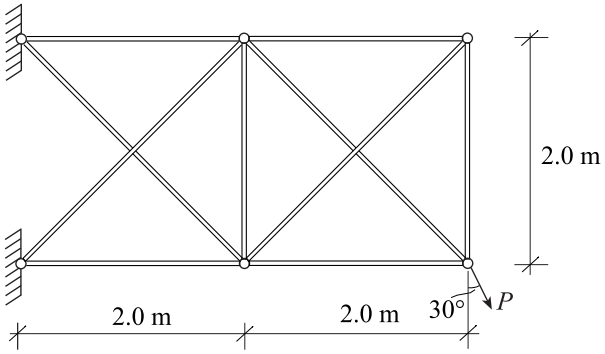
9.4 exs_bar2_l

Purpose

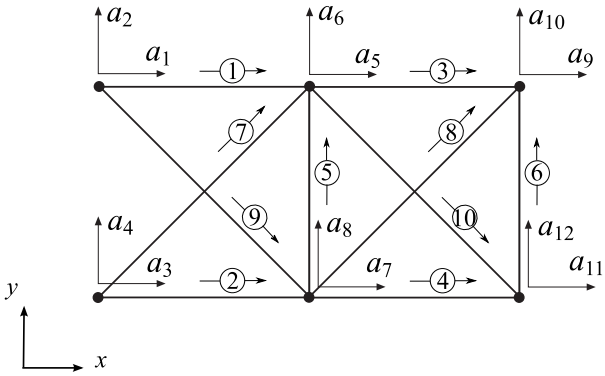
Analysis of a plane truss.

Description

Consider a plane truss, loaded by a single force $P = 0.5$ MN.



The corresponding finite element model consists of ten elements and twelve degrees of freedom.



Material properties:

- Cross-sectional area: $A = 25.0 \times 10^{-4}$ m^2
- Young's modulus: $E = 2.10 \times 10^5$ MPa

Example

The computation is initialized by importing CALFEM and NumPy. The element topology matrix contains only the degrees of freedom for each element:

```
>>> import numpy as np
>>> import calfem.core as cfc
>>>
>>> # Element topology matrix (DOFs only, no element numbers)
>>> Edof = np.array([
...     [1, 2, 5, 6],    # Element 1: DOFs 1,2,5,6
...     [3, 4, 7, 8],    # Element 2: DOFs 3,4,7,8
...     [5, 6, 9, 10],   # Element 3: DOFs 5,6,9,10
...     [7, 8, 11, 12],  # Element 4: DOFs 7,8,11,12
...     [7, 8, 5, 6],    # Element 5: DOFs 7,8,5,6
...     [11, 12, 9, 10], # Element 6: DOFs 11,12,9,10
...     [3, 4, 5, 6],    # Element 7: DOFs 3,4,5,6
```

```
...     [7, 8, 9, 10],   # Element 8: DOFs 7,8,9,10
...     [1, 2, 7, 8],    # Element 9: DOFs 1,2,7,8
...     [5, 6, 11, 12]   # Element 10: DOFs 5,6,11,12
... ])
```

The global stiffness matrix `K` and load vector `f` are initialized. The load $P = 0.5$ MN is divided into x and y components:

```
>>> # Initialize global system
>>> K = np.zeros((12, 12))
>>> f = np.zeros(12)
>>>
>>> # Apply load P=0.5 MN at 30° angle (DOFs 11,12 -> indices
↪  10,11)
>>> P = 0.5e6  # Load magnitude [N]
>>> f[10] = P * np.sin(np.pi/6)   # x-component at DOF 11
>>> f[11] = -P * np.cos(np.pi/6)  # y-component at DOF 12
>>> print("Load vector:")
>>> print(f)
[     0.      0.      0.      0.      0.      0.      0.      0.
      0.      0.  250000. -433013.]
```

The material and geometric properties are defined, along with element coordinate matrices:

```
>>> # Material and geometric properties
>>> A = 25.0e-4   # Cross-sectional area [m²]
>>> E = 2.1e11    # Young's modulus [Pa]
>>> ep = [E, A]   # Element properties
>>>
>>> # Element coordinate matrices (x-coordinates for each element)
>>> Ex = np.array([
...     [0, 2],  # Element 1
...     [0, 2],  # Element 2
...     [2, 4],  # Element 3
...     [2, 4],  # Element 4
...     [2, 2],  # Element 5
...     [4, 4],  # Element 6
...     [0, 2],  # Element 7
...     [2, 4],  # Element 8
...     [0, 2],  # Element 9
...     [2, 4]   # Element 10
... ])
>>>
>>> # Element coordinate matrices (y-coordinates for each element)
>>> Ey = np.array([
...     [2, 2],  # Element 1
...     [0, 0],  # Element 2
...     [2, 2],  # Element 3
...     [0, 0],  # Element 4
...     [0, 2],  # Element 5
...     [0, 2],  # Element 6
...     [0, 2],  # Element 7
...     [0, 2],  # Element 8
...     [2, 0],  # Element 9
...     [2, 0]   # Element 10
... ])
```

The element stiffness matrices are computed and assembled in a loop:

```
>>> # Compute element stiffness matrices and assemble
>>> for i in range(10):
...     Ke = cfc.bar2e(Ex[i], Ey[i], ep)
...     K = cfc.assem(Edof[i], K, Ke)
```

```
>>>
>>> print("Global stiffness matrix assembled successfully")
Global stiffness matrix assembled successfully
```

The system of equations is solved by specifying boundary conditions and using `solveq()`:

```
>>> # Boundary conditions (fixed supports at nodes 1 and 2)
>>> bc = np.array([
...     [1, 0],  # DOF 1 = 0 (fixed in x)
...     [2, 0],  # DOF 2 = 0 (fixed in y)
...     [3, 0],  # DOF 3 = 0 (fixed in x)
...     [4, 0]   # DOF 4 = 0 (fixed in y)
... ])
>>>
>>> # Solve the system
>>> a, r = cfc.solveq(K, f, bc)
>>> print("Displacements [m]:")
>>> print(a)
[ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  2.4000e-03
↪  -4.5000e-03
 -1.6000e-03 -4.2000e-03  3.0000e-03 -1.0700e-02 -1.7000e-03
↪  -1.1300e-02]
>>> print("Reaction forces [N]:")
>>> print(r)
[-8.6603e+05  2.4009e+05  6.1603e+05  1.9293e+05  0.0000e+00
↪  0.0000e+00
  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00
↪  0.0000e+00]
```

The displacement at the point of loading is -1.7×10^{-3} m in the x-direction and -11.3×10^{-3} m in the y-direction. At the upper support the horizontal force is -0.866 MN and the vertical 0.240 MN. At the lower support the forces are 0.616 MN and 0.193 MN, respectively.

Normal forces are evaluated from element displacements. These are obtained from the global displacements using `extract_ed()` and the forces are calculated using `bar2s()`:

```
>>> # Extract element displacements
>>> ed = cfc.extract_ed(Edof, a)
>>>
>>> # Compute normal forces for all elements
>>> N = np.zeros(10)
>>> for i in range(10):
...     es = cfc.bar2s(Ex[i], Ey[i], ep, ed[i])
...     N[i] = es[0]  # Normal force (first component)
>>>
>>> print("Normal forces [N]:")
>>> print(N)
[ 6.2594e+05 -4.2310e+05  1.7064e+05 -1.2370e+04 -6.9450e+04
↪  1.7064e+05
 -2.7284e+05 -2.4132e+05  3.3953e+05  3.7105e+05]
```

The largest normal force $N = 0.626$ MN is obtained in element 1 and is equivalent to a normal stress $\sigma = 250$ MPa.

To reduce the quantity of input data, the element coordinate matrices `Ex` and `Ey` can alternatively be created from a global coordinate matrix `Coord` and a global topology matrix `Dof` using `coordxtr()`:

```
>>> # Alternative approach using global coordinates and topology
>>> Coord = np.array([
...     [0, 2],  # Node 1
...     [0, 0],  # Node 2
...     [2, 2],  # Node 3
...     [2, 0],  # Node 4
...     [4, 2],  # Node 5
...     [4, 0]   # Node 6
... ])
>>>
>>> Dof = np.array([
...     [1, 2],    # Node 1: DOFs 1,2
...     [3, 4],    # Node 2: DOFs 3,4
...     [5, 6],    # Node 3: DOFs 5,6
...     [7, 8],    # Node 4: DOFs 7,8
...     [9, 10],   # Node 5: DOFs 9,10
...     [11, 12]   # Node 6: DOFs 11,12
... ])
>>>
>>> # Extract element coordinates automatically
>>> ex, ey = cfc.coordxtr(Edof, Coord, Dof, 2)
>>> print("Extracted element coordinates match manual definition")
Extracted element coordinates match manual definition
```

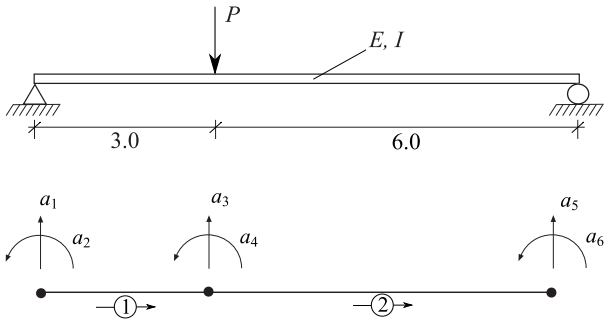
9.5 exs_beam1

Purpose

Analysis of a simply supported beam.

Description

Consider a beam with the length 9.0 m. The beam is simply supported and loaded by a point load $P = 10000$ N applied at a point 3.0 m from the left support. The corresponding computational model has six degrees of freedom and consists of two beam elements with four degrees of freedom. The beam has Young's modulus $E = 210$ GPa and moment of inertia $I = 2510 \times 10^{-8}$ m^4.



Example

The computation is initialized by importing CALFEM and NumPy. The element topology matrix contains only the degrees of freedom for each element:

```
>>> import numpy as np
>>> import calfem.core as cfc
>>>
>>> # Element topology matrix (DOFs only, no element numbers)
>>> Edof = np.array([
```

```
...     [1, 2, 3, 4],  # Element 1: DOFs 1,2,3,4 (node 1-2)
...     [3, 4, 5, 6]   # Element 2: DOFs 3,4,5,6 (node 2-3)
... ])
```

The global stiffness matrix `K` and load vector `f` are initialized. The point load P = 10000 N is applied at DOF 3:

```
>>> # Initialize global system
>>> K = np.zeros((6, 6))
>>> f = np.zeros(6)
>>> f[2] = -10000  # Load at DOF 3 (index 2 in 0-based indexing)
>>> print("Load vector:")
>>> print(f)
[     0.      0. -10000.      0.      0.      0.]
```

The material and geometric properties are defined, along with element coordinates and stiffness matrices:

```
>>> # Material and geometric properties
>>> E = 210e9      # Young's modulus [Pa]
>>> I = 2510e-8    # Moment of inertia [m⁴]
>>> ep = [E, I]    # Element properties
>>>
>>> # Element coordinates [m]
>>> ex1 = np.array([0, 3])  # Element 1: from x=0 to x=3
>>> ex2 = np.array([3, 9])  # Element 2: from x=3 to x=9
>>>
>>> # Compute element stiffness matrices
>>> Ke1 = cfc.beam1e(ex1, ep)
>>> print("Element 1 stiffness matrix:")
>>> print(Ke1)
[[ 2.3427e+06  3.5140e+06 -2.3427e+06  3.5140e+06]
 [ 3.5140e+06  7.0280e+06 -3.5140e+06  3.5140e+06]
 [-2.3427e+06 -3.5140e+06  2.3427e+06 -3.5140e+06]
 [ 3.5140e+06  3.5140e+06 -3.5140e+06  7.0280e+06]]
>>>
>>> Ke2 = cfc.beam1e(ex2, ep)
>>> print("Element 2 stiffness matrix:")
>>> print(Ke2)
[[ 2.9279e+05  8.7836e+05 -2.9279e+05  8.7836e+05]
 [ 8.7836e+05  3.5135e+06 -8.7836e+05  1.7567e+06]
 [-2.9279e+05 -8.7836e+05  2.9279e+05 -8.7836e+05]
 [ 8.7836e+05  1.7567e+06 -8.7836e+05  3.5135e+06]]
```

The element stiffness matrices are assembled into the global stiffness matrix:

```
>>> # Assemble global stiffness matrix
>>> K = cfc.assem(Edof[0], K, Ke1)
>>> K = cfc.assem(Edof[1], K, Ke2)
>>> print("Global stiffness matrix assembled successfully")
Global stiffness matrix assembled successfully
```

The system of equations is solved by defining boundary conditions and using `solveq()`:

```
>>> # Boundary conditions (simply supported beam)
>>> bc = np.array([
...     [1, 0],  # DOF 1 = 0 (vertical displacement at left support)
...      [5, 0]   # DOF 5 = 0 (vertical displacement at right
↪  support)
... ])
>>>
```

```
>>> # Solve the system
>>> a, r = cfc.solveq(K, f, bc)
>>> print("Displacements:")
>>> print(a)
[ 0.0000e+00 -9.5000e-03 -2.2800e-02 -3.8000e-03  0.0000e+00
↪  7.6000e-03]
>>> print("Reaction forces [N]:")
>>> print(r)
[ 6667.   0.   0.   0.   3333.   0.]
```

The section forces and element displacements are calculated from global displacements:

```
>>> # Extract element displacements
>>> Ed = cfc.extract_ed(Edof, a)
>>>
>>> # Compute section forces and internal displacements
>>> # For beam1s, we don't need eq parameter for point loads
>>> es1, edi1 = cfc.beam1s(ex1, ep, Ed[0], n_points=6)
>>> es2, edi2 = cfc.beam1s(ex2, ep, Ed[1], n_points=11)
>>>
>>> print("Element 1 section forces [N, Nm]:")
>>> print(es1[:5])  # Show first 5 points
>>> print("Element 2 section forces [N, Nm]:")
>>> print(es2[:5])  # Show first 5 points
```

Results:

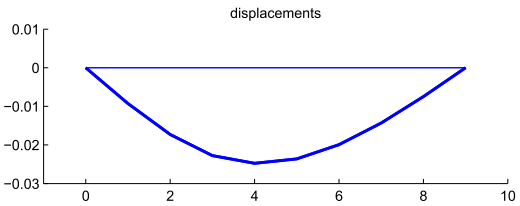
The solution shows the expected behavior for a simply supported beam with a point load:

- **Maximum deflection**: 22.8 mm at the loading point (DOF 3)
- **Support reactions**: 6667 N at left support, 3333 N at right support
- **Maximum shear force**: 6667 N (in element 1)
- **Maximum moment**: 20000 Nm (at the loading point)

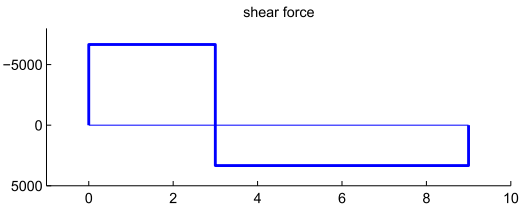
Visualization of the results using matplotlib:

```
>>> import matplotlib.pyplot as plt
>>>
>>> # Create position vectors for plotting
>>> x1 = np.linspace(0, 3, len(edi1))
>>> x2 = np.linspace(3, 9, len(edi2))
>>> x_full = np.concatenate(([0], x1, x2, [9]))
>>>
>>> # Displacement diagram
>>> plt.figure(1, figsize=(10, 4))
>>> plt.plot([0, 9], [0, 0], 'k-', linewidth=1, alpha=0.5)  #
↪  Reference line
>>> disp_full = np.concatenate(([0], edi1[:, 0], edi2[:, 0], [0]))
>>> plt.plot(x_full, disp_full, 'b-', linewidth=2,
↪  label='Displacement')
>>> plt.xlabel('Position [m]')
>>> plt.ylabel('Displacement [m]')
>>> plt.title('Displacement Diagram')
>>> plt.grid(True, alpha=0.3)
>>> plt.axis([-1, 10, -0.03, 0.01])
>>> plt.legend()
>>> plt.show()
>>>
```

```
>>> # Shear force diagram
>>> plt.figure(2, figsize=(10, 4))
>>> plt.plot([0, 9], [0, 0], 'k-', linewidth=1, alpha=0.5)  #
↪  Reference line
>>> shear_full = np.concatenate(([0], es1[:, 0], es2[:, 0], [0]))
>>> plt.plot(x_full, shear_full, 'r-', linewidth=2, label='Shear
↪  Force')
>>> plt.xlabel('Position [m]')
>>> plt.ylabel('Shear Force [N]')
>>> plt.title('Shear Force Diagram')
>>> plt.grid(True, alpha=0.3)
>>> plt.axis([-1, 10, -8000, 5000])
>>> plt.gca().invert_yaxis()  # Engineering convention
>>> plt.legend()
>>> plt.show()
>>>
>>> # Moment diagram
>>> plt.figure(3, figsize=(10, 4))
>>> plt.plot([0, 9], [0, 0], 'k-', linewidth=1, alpha=0.5)  #
↪  Reference line
>>> moment_full = np.concatenate(([0], es1[:, 1], es2[:, 1], [0]))
>>> plt.plot(x_full, moment_full, 'g-', linewidth=2,
↪  label='Bending Moment')
>>> plt.xlabel('Position [m]')
>>> plt.ylabel('Bending Moment [Nm]')
>>> plt.title('Bending Moment Diagram')
>>> plt.grid(True, alpha=0.3)
>>> plt.axis([-1, 10, -5000, 25000])
>>> plt.gca().invert_yaxis()  # Engineering convention
>>> plt.legend()
>>> plt.show()
```



Displacement diagram



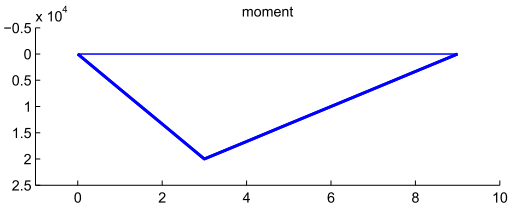
Shear force diagram

Moment diagram

9.6 exs_beam2

Purpose

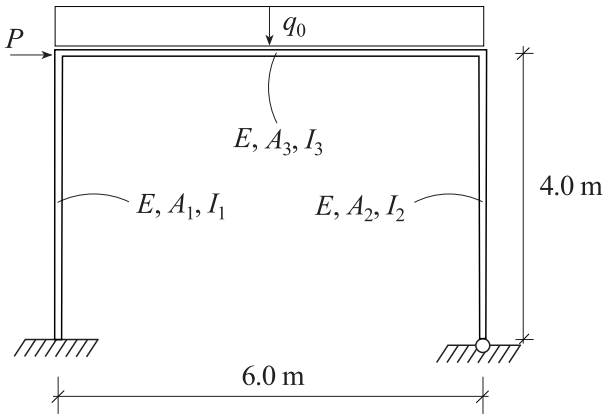
Analysis of a plane frame.



FELPLACERAD

Description

A frame consists of one horizontal and two vertical beams according to the figure.



Material and geometric properties:

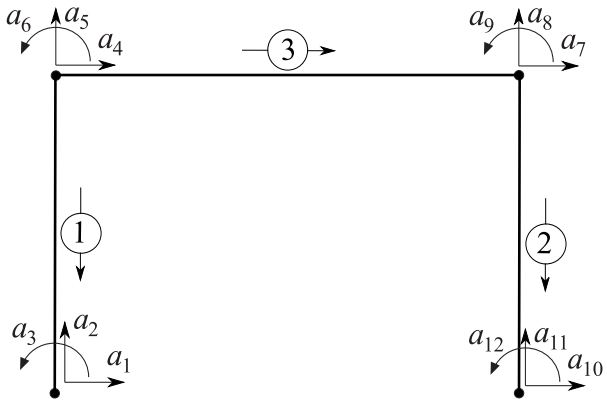
E	=	200 GPa
A_1	=	2.0×10^{-3} m^2
I_1	=	1.6×10^{-5} m^4
A_2	=	6.0×10^{-3} m^2
I_2	=	5.4×10^{-5} m^4
P	=	2.0 kN
q_0	=	10.0 kN/m

The corresponding finite element model consists of three beam elements and twelve degrees of freedom.

Example

The computation is initialized by importing CALFEM and NumPy. The element topology matrix contains only the degrees of freedom for each element:

```
>>> import numpy as np
>>> import calfem.core as cfc
>>>
>>> # Element topology matrix (DOFs only, no element numbers)
>>> Edof = np.array([
```




```
...     [4, 5, 6, 1, 2, 3],    # Element 1: left column (node 2-1)
...     [7, 8, 9, 10, 11, 12], # Element 2: right column (node 3-4)
...     [4, 5, 6, 7, 8, 9]     # Element 3: horizontal beam (node
↪ 2-3)
... ])
```

The global stiffness matrix `K` and load vector `f` are initialized. The point load P = 2000 N is applied at DOF 4:

```
>>> # Initialize global system
>>> K = np.zeros((12, 12))
>>> f = np.zeros(12)
>>> f[3] = 2e3  # Load at DOF 4 (index 3 in 0-based indexing)
>>> print("Load vector:")
>>> print(f)
[   0.    0.    0. 2000.    0.    0.    0.    0.    0.    0.    0.
↪  0.]
```

np.zeros((12,1))

The material and geometric properties are defined for each element type:

```
>>> # Material and geometric properties
>>> E = 200e9        # Young's modulus [Pa]
>>> A1 = 2e-3       # Cross-sectional area for columns [m²]
>>> A2 = 6e-3       # Cross-sectional area for beam [m²]
>>> I1 = 1.6e-5     # Moment of inertia for columns [m⁴]
>>> I2 = 5.4e-5     # Moment of inertia for beam [m⁴]
>>>
>>> ep1 = [E, A1, I1]  # Element properties for columns
>>> ep2 = [E, A2, I2]  # Element properties for horizontal beam
>>>
>>> # Element coordinates [m]
>>> ex1 = np.array([0, 0])  # Element 1: left column x-coordinates
>>> ey1 = np.array([0, 4])  # Element 1: left column y-coordinates
>>>
>>> ex2 = np.array([6, 6])  # Element 2: right column x-coordinates
>>> ey2 = np.array([0, 4])  # Element 2: right column y-coordinates
>>>
>>> ex3 = np.array([0, 6])  # Element 3: horizontal beam
↪  x-coordinates
>>> ey3 = np.array([4, 4])  # Element 3: horizontal beam
↪  y-coordinates
>>>
>>> # Distributed loads (only on horizontal beam)
>>> eq1 = np.array([0, 0])     # No distributed load on left column
```

```
>>> eq2 = np.array([0, 0])      # No distributed load on right
↪  column
>>> eq3 = np.array([0, -10e3])  # 10 kN/m downward on horizontal
↪  beam
```

The element stiffness matrices and load vectors are computed and assembled:

```
>>> # Compute element stiffness matrices
>>> Ke1 = cfc.beam2e(ex1, ey1, ep1)
>>> Ke2 = cfc.beam2e(ex2, ey2, ep1)  # Same properties as element 1
>>> Ke3, fe3 = cfc.beam2e(ex3, ey3, ep2, eq3)  # With distributed
↪  load
>>>
>>> # Assemble global stiffness matrix and load vector
>>> K = cfc.assem(Edof[0], K, Ke1)
>>> K = cfc.assem(Edof[1], K, Ke2)
>>> K, f = cfc.assem(Edof[2], K, Ke3, f, fe3)
>>>
>>> print("Global system assembled successfully")
Global system assembled successfully
```

The system of equations is solved by specifying boundary conditions and using `solveq()`:

```
>>> # Boundary conditions (fixed supports at base of columns)
>>> bc = np.array([
...     [1, 0],   # DOF 1 = 0 (x-displacement at left base)
...     [2, 0],   # DOF 2 = 0 (y-displacement at left base)
...     [3, 0],   # DOF 3 = 0 (rotation at left base)
...     [10, 0],  # DOF 10 = 0 (x-displacement at right base)
...     [11, 0]   # DOF 11 = 0 (y-displacement at right base)
... ])
>>>
>>> # Solve the system
>>> a, r = cfc.solveq(K, f, bc)
>>> print("Displacements [m, rad]:")
>>> print(a)
[ 0.0000e+00  0.0000e+00  0.0000e+00  7.5000e-03 -3.0000e-04
↪  -5.4000e-03
  7.5000e-03 -3.0000e-04  4.7000e-03  0.0000e+00  0.0000e+00
↪  -5.2000e-03]
>>> print("Reaction forces [N, Nm]:")
>>> print(r)
[ 1.927e+03  2.874e+04  4.450e+02  0.000e+00  0.000e+00  0.000e+00
  0.000e+00  0.000e+00  0.000e+00 -3.927e+03  3.126e+04  0.000e+00]
```

The element displacements are extracted and section forces are computed along each element:

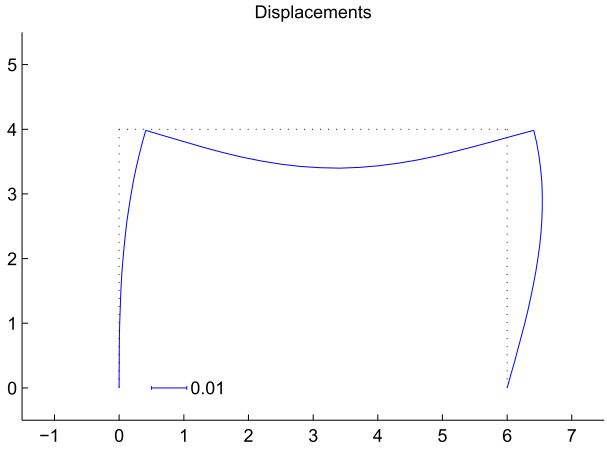
```
>>> # Extract element displacements
>>> Ed = cfc.extract_ed(Edof, a)
>>>
>>> # Compute section forces and internal displacements (21 points
↪  each)
>>> es1, edi1 = cfc.beam2s(ex1, ey1, ep1, Ed[0], eq1, n_points=21)
>>> es2, edi2 = cfc.beam2s(ex2, ey2, ep1, Ed[1], eq2, n_points=21)
>>> es3, edi3 = cfc.beam2s(ex3, ey3, ep2, Ed[2], eq3, n_points=21)
>>>
>>> print("Element 1 (left column) section forces [N, N, Nm]:")
>>> print("N =", es1[0, 0], "V =", es1[0, 1], "M =", es1[0, 2])
Element 1 (left column) section forces [N, N, Nm]:
N = -28741.0 V = 1927.0 M = 8152.0
>>>
```

```
>>> print("Element 2 (right column) section forces [N, N, Nm]:")
>>> print("N =", es2[0, 0], "V =", es2[0, 1], "M =", es2[0, 2])
Element 2 (right column) section forces [N, N, Nm]:
N = -31259.0 V = -3927.0 M = -15707.0
>>>
>>> print("Element 3 (horizontal beam) section forces [N, N, Nm]:")
>>> print("N =", es3[0, 0], "V =", es3[0, 1], "M =", es3[0, 2])
Element 3 (horizontal beam) section forces [N, N, Nm]:
N = -3927.0 V = -28741.0 M = -8152.0
```

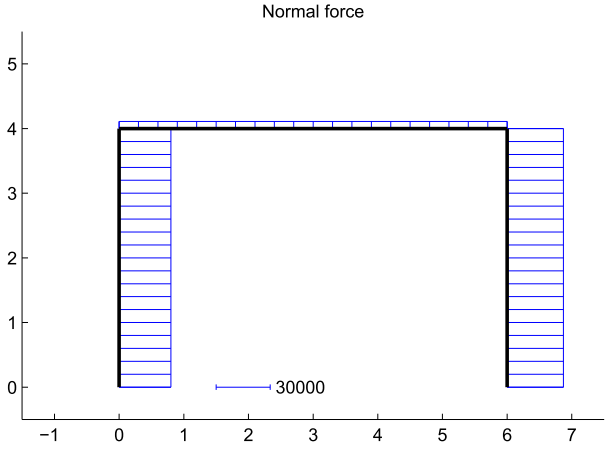
Visualization of the frame analysis results using CALFEM's visualization functions:

```
>>> import calfem.vis_mpl as cfv
>>> import matplotlib.pyplot as plt
>>>
>>> # Displacement diagram
>>> cfv.figure(1, figsize=(10, 8))
>>>
>>> # Draw undeformed structure
>>> plotpar = [2, 1, 0]  # Line style, marker, color
>>> cfv.eldraw2(ex1, ey1, plotpar)
>>> cfv.eldraw2(ex2, ey2, plotpar)
>>> cfv.eldraw2(ex3, ey3, plotpar)
>>>
>>> # Calculate scale factor and draw deformed structure
>>> sfac = cfv.scalfact2(ex3, ey3, Ed[2], 0.1)
>>> plotpar = [1, 2, 1]  # Line style, marker, color for deformed
↪  shape
>>> cfv.dispbeam2(ex1, ey1, edi1, plotpar, sfac)
>>> cfv.dispbeam2(ex2, ey2, edi2, plotpar, sfac)
>>> cfv.dispbeam2(ex3, ey3, edi3, plotpar, sfac)
>>>
>>> plt.axis([-1.5, 7.5, -0.5, 5.5])
>>> cfv.scalgraph2(sfac, [1e-2, 0.5, 0])
>>> plt.title('Displacements')
>>> plt.xlabel('x [m]')
>>> plt.ylabel('y [m]')
>>> plt.grid(True, alpha=0.3)
>>> plt.show()
>>>
>>> # Normal force diagram
>>> cfv.figure(2, figsize=(10, 8))
>>> plotpar = [2, 1]  # Line style, marker
>>> sfac = cfv.scalfact2(ex1, ey1, es1[:, 0], 0.2)
>>> cfv.secforce2(ex1, ey1, es1[:, 0], plotpar, sfac)
>>> cfv.secforce2(ex2, ey2, es2[:, 0], plotpar, sfac)
>>> cfv.secforce2(ex3, ey3, es3[:, 0], plotpar, sfac)
>>>
>>> plt.axis([-1.5, 7.5, -0.5, 5.5])
>>> cfv.scalgraph2(sfac, [3e4, 1.5, 0])
>>> plt.title('Normal Force')
>>> plt.xlabel('x [m]')
>>> plt.ylabel('y [m]')
>>> plt.grid(True, alpha=0.3)
>>> plt.show()
>>>
>>> # Shear force diagram
>>> cfv.figure(3, figsize=(10, 8))
>>> plotpar = [2, 1]
>>> sfac = cfv.scalfact2(ex3, ey3, es3[:, 1], 0.2)
>>> cfv.secforce2(ex1, ey1, es1[:, 1], plotpar, sfac)
>>> cfv.secforce2(ex2, ey2, es2[:, 1], plotpar, sfac)
>>> cfv.secforce2(ex3, ey3, es3[:, 1], plotpar, sfac)
>>>
>>> plt.axis([-1.5, 7.5, -0.5, 5.5])
>>> cfv.scalgraph2(sfac, [3e4, 0.5, 0])
```

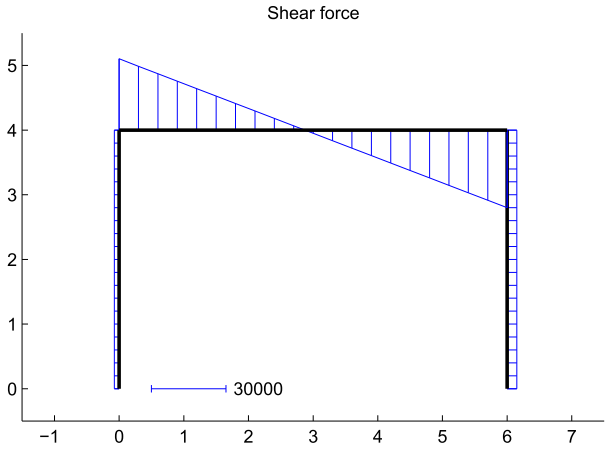
```
>>> plt.title('Shear Force')
>>> plt.xlabel('x [m]')
>>> plt.ylabel('y [m]')
>>> plt.grid(True, alpha=0.3)
>>> plt.show()
>>>
>>> # Moment diagram
>>> cfv.figure(4, figsize=(10, 8))
>>> plotpar = [2, 1]
>>> sfac = cfv.scalfact2(ex3, ey3, es3[:, 2], 0.2)
>>> cfv.secforce2(ex1, ey1, es1[:, 2], plotpar, sfac)
>>> cfv.secforce2(ex2, ey2, es2[:, 2], plotpar, sfac)
>>> cfv.secforce2(ex3, ey3, es3[:, 2], plotpar, sfac)
>>>
>>> plt.axis([-1.5, 7.5, -0.5, 5.5])
>>> cfv.scalgraph2(sfac, [3e4, 0.5, 0])
>>> plt.title('Bending Moment')
>>> plt.xlabel('x [m]')
>>> plt.ylabel('y [m]')
>>> plt.grid(True, alpha=0.3)
>>> plt.show()
```



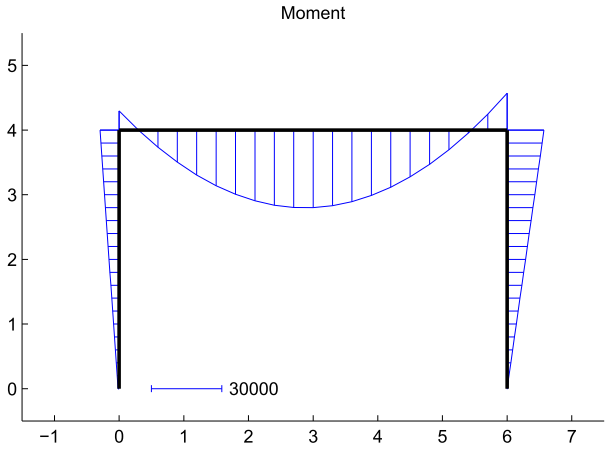
Displacement diagram



Normal force diagram



Shear force diagram



Moment diagram

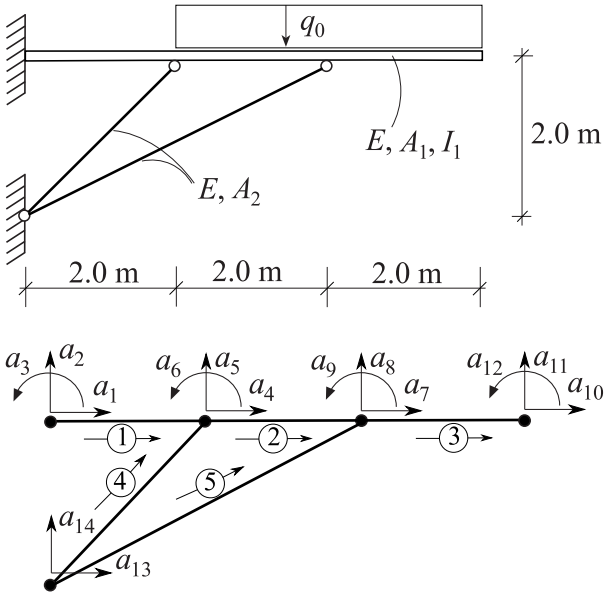
9.7 exs_beambar2

Purpose

Analysis of a combined beam and bar structure.

Description

Consider a structure consisting of a beam with $A_1 = 4.0 \times 10^{-3}$ m^2 and $I_1 = 5.4 \times 10^{-5}$ m^4 supported by two bars with $A_2 = 1.0 \times 10^{-3}$ m^2. The beam as well as the bars have $E = 200$ GPa. The structure is loaded by a distributed load $q = 10$ kN/m. The corresponding finite element model consists of three beam elements and two bar elements and has 14 degrees of freedom.



Example

The computation is initialized by importing CALFEM and NumPy. The topology matrices are defined separately for beam and bar elements, containing only the degrees of freedom:

```
>>> import numpy as np
>>> import calfem.core as cfc
>>>
>>> # Element topology matrices (DOFs only, no element numbers)
>>> # Beam elements (6 DOFs per element: 2 nodes × 3 DOFs/node)
>>> Edof1 = np.array([
...     [1, 2, 3, 4, 5, 6],      # Beam element 1: nodes 1-2
...     [4, 5, 6, 7, 8, 9],      # Beam element 2: nodes 2-3
...     [7, 8, 9, 10, 11, 12]    # Beam element 3: nodes 3-4
... ])
>>>
>>> # Bar elements (4 DOFs per element: 2 nodes × 2 DOFs/node)
>>> Edof2 = np.array([
...     [13, 14, 4, 5],    # Bar element 1: node 5 to node 2
...     [13, 14, 7, 8]     # Bar element 2: node 5 to node 3
... ])
>>>
>>> # Initialize global system (14 DOFs total)
>>> K = np.zeros((14, 14))
>>> f = np.zeros(14)
>>> print("Global system initialized with 14 DOFs")
Global system initialized with 14 DOFs
```

The material and geometric properties are defined for beam and bar elements:

```
>>> # Material and geometric properties
>>> E = 200e9        # Young's modulus [Pa]
>>> A1 = 4.0e-3      # Cross-sectional area for beam [m²]
>>> A2 = 1.0e-3      # Cross-sectional area for bars [m²]
>>> I1 = 5.4e-5      # Moment of inertia for beam [m⁴]
```

```
>>>
>>> ep1 = [E, A1, I1]  # Element properties for beam elements
>>> ep2 = [E, A2]      # Element properties for bar elements
>>>
>>> # Distributed loads
>>> eq1 = np.array([0, 0])        # No distributed load
>>> eq2 = np.array([0, -10e3])    # 10 kN/m downward distributed
↪  load
>>>
>>> # Element coordinates [m]
>>> # Beam elements (horizontal at y = 2m)
>>> ex1 = np.array([0, 2])  # Beam element 1: x-coordinates
>>> ey1 = np.array([2, 2])  # Beam element 1: y-coordinates
>>>
>>> ex2 = np.array([2, 4])  # Beam element 2: x-coordinates
>>> ey2 = np.array([2, 2])  # Beam element 2: y-coordinates
>>>
>>> ex3 = np.array([4, 6])  # Beam element 3: x-coordinates
>>> ey3 = np.array([2, 2])  # Beam element 3: y-coordinates
>>>
>>> # Bar elements (supporting bars)
>>> ex4 = np.array([0, 2])  # Bar element 1: x-coordinates
>>> ey4 = np.array([0, 2])  # Bar element 1: y-coordinates
>>>
>>> ex5 = np.array([0, 4])  # Bar element 2: x-coordinates
>>> ey5 = np.array([0, 2])  # Bar element 2: y-coordinates
```

The element stiffness matrices are computed using `beam2e()` for beam elements and `bar2e()` for bar elements. Element load vectors from distributed loads are also computed:

```
>>> # Compute beam element stiffness matrices and load vectors
>>> Ke1 = cfc.beam2e(ex1, ey1, ep1)  # No distributed load
>>> Ke2, fe2 = cfc.beam2e(ex2, ey2, ep1, eq2)  # With distributed
↪  load
>>> Ke3, fe3 = cfc.beam2e(ex3, ey3, ep1, eq2)  # With distributed
↪  load
>>>
>>> # Compute bar element stiffness matrices
>>> Ke4 = cfc.bar2e(ex4, ey4, ep2)
>>> Ke5 = cfc.bar2e(ex5, ey5, ep2)
>>>
>>> print("Element stiffness matrices computed successfully")
Element stiffness matrices computed successfully
```

The global stiffness matrix and load vector are assembled using `assem()`:

```
>>> # Assemble beam elements
>>> K = cfc.assem(Edof1[0], K, Ke1)              # Beam element 1
>>> K, f = cfc.assem(Edof1[1], K, Ke2, f, fe2)   # Beam element 2
↪  with load
>>> K, f = cfc.assem(Edof1[2], K, Ke3, f, fe3)   # Beam element 3
↪  with load
>>>
>>> # Assemble bar elements
>>> K = cfc.assem(Edof2[0], K, Ke4)              # Bar element 1
>>> K = cfc.assem(Edof2[1], K, Ke5)              # Bar element 2
>>>
>>> print("Global system assembled successfully")
Global system assembled successfully
```

The system of equations is solved by specifying boundary conditions and using `solveq()`. The vertical displacement at the end of the beam is 13.0 mm:

```
>>> # Boundary conditions (fixed supports)
>>> bc = np.array([
...     [1, 0],   # DOF 1 = 0 (x-displacement at left support)
...     [2, 0],   # DOF 2 = 0 (y-displacement at left support)
...     [3, 0],   # DOF 3 = 0 (rotation at left support)
...     [13, 0],  # DOF 13 = 0 (x-displacement at bar support)
...     [14, 0]   # DOF 14 = 0 (y-displacement at bar support)
... ])
>>>
>>> # Solve the system
>>> a, r = cfc.solveq(K, f, bc)
>>> print("Displacements [m, rad]:")
>>> print(a)
[ 0.0000e+00  0.0000e+00  0.0000e+00  2.0000e-04 -6.0000e-04
↪  -1.0000e-03
  4.0000e-04 -4.6000e-03 -3.3000e-03  4.0000e-04 -1.3000e-02
↪  -4.5000e-03
  0.0000e+00  0.0000e+00]
>>> print("Reaction forces [N, Nm]:")
>>> print(r)
[-8.0702e+04 -6.6040e+03 -1.4030e+03  0.0000e+00  0.0000e+00
↪  0.0000e+00
  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00
↪  0.0000e+00
  8.0702e+04  4.6604e+04]
>>>
>>> print(f"Maximum vertical displacement: {abs(a[10]):.3f} m =
↪  {abs(a[10])*1000:.1f} mm")
Maximum vertical displacement: 0.013 m = 13.0 mm
```

The section forces are calculated using `beam2s()` and `bar2s()` from element displacements. This yields normal forces of -35.4 kN and -152.5 kN in the bars and maximum moment of 20.0 kNm in the beam:

```
>>> # Extract element displacements
>>> Ed1 = cfc.extract_ed(Edof1, a)  # Beam element displacements
>>> Ed2 = cfc.extract_ed(Edof2, a)  # Bar element displacements
>>>
>>> # Compute section forces for beam elements (11 points each)
>>> es1 = cfc.beam2s(ex1, ey1, ep1, Ed1[0], eq1, n_points=11)
>>> es2 = cfc.beam2s(ex2, ey2, ep1, Ed1[1], eq2, n_points=11)
>>> es3 = cfc.beam2s(ex3, ey3, ep1, Ed1[2], eq2, n_points=11)
>>>
>>> # Compute section forces for bar elements
>>> es4 = cfc.bar2s(ex4, ey4, ep2, Ed2[0])
>>> es5 = cfc.bar2s(ex5, ey5, ep2, Ed2[1])
>>>
>>> print("Section forces summary:")
>>> print(f"Beam element 1 - N: {es1[0,0]:.0f} N, V:
↪  {es1[0,1]:.0f} N, M: {es1[0,2]:.0f} Nm")
>>> print(f"Beam element 2 - N: {es2[0,0]:.0f} N, V:
↪  {es2[0,1]:.0f} N, M: {es2[0,2]:.0f} Nm")
>>> print(f"Beam element 3 - N: {es3[0,0]:.0f} N, V:
↪  {es3[0,1]:.0f} N, M: {es3[0,2]:.0f} Nm")
>>> print(f"Bar element 1 - Normal force: {es4[0]:.0f} N =
↪  {es4[0]/1000:.1f} kN")
>>> print(f"Bar element 2 - Normal force: {es5[0]:.0f} N =
↪  {es5[0]/1000:.1f} kN")
Section forces summary:
Beam element 1 - N: 80702 N, V: 6604 N, M: 1403 Nm
Beam element 2 - N: 68194 N, V: -5903 N, M: -11806 Nm
Beam element 3 - N: 0 N, V: -20000 N, M: -20000 Nm
Bar element 1 - Normal force: -35376 N = -35.4 kN
Bar element 2 - Normal force: -152490 N = -152.5 kN
>>>
```

```
>>> print(f"Maximum moment in beam: {abs(min(es3[:,2])):.0f} Nm =
↪  {abs(min(es3[:,2]))/1000:.1f} kNm")
Maximum moment in beam: 20000 Nm = 20.0 kNm
```

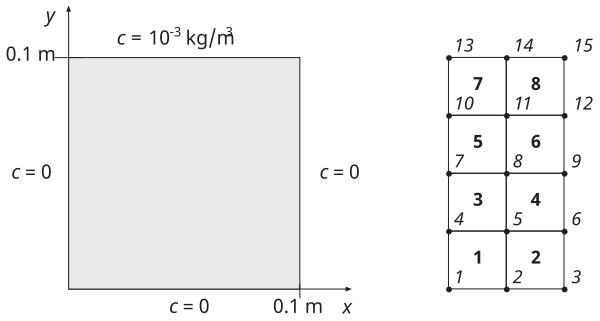
9.8 exs_flw_diff2

Purpose

Show how to solve a two dimensional diffusion problem.

Description

Consider a filter paper of square shape. Three sides are in contact with pure water and the fourth side is in contact with a solution of concentration $c = 1.0 \cdot 10^{-3}$ kg/m³. The length of each side is 0.100 m. Using symmetry, only half of the paper has to be analyzed. The paper and the corresponding finite element mesh are shown. The following boundary conditions are applied:



Example

The computation is initialized by importing CALFEM and NumPy. The element topology matrix contains only the degrees of freedom for each quadrilateral element:

```
>>> import numpy as np
>>> import calfem.core as cfc
>>>
>>> # Element topology matrix (DOFs only, no element numbers)
>>> # Each row represents one quadrilateral element with 4 nodes
>>> Edof = np.array([
...     [1, 2, 5, 4],      # Element 1
...     [2, 3, 6, 5],      # Element 2
...     [4, 5, 8, 7],      # Element 3
...     [5, 6, 9, 8],      # Element 4
...     [7, 8, 11, 10],    # Element 5
...     [8, 9, 12, 11],    # Element 6
...     [10, 11, 14, 13],  # Element 7
...     [11, 12, 15, 14]   # Element 8
... ])
```

The global system matrices are initialized:

```
>>> # Initialize global system (15 DOFs for concentration field)
>>> K = np.zeros((15, 15))  # Global conductivity matrix
```

```
>>> f = np.zeros(15)        # Global source vector (no sources in this
↪  problem)
>>> print("Global system initialized with 15 DOFs")
Global system initialized with 15 DOFs
```

Because all elements have the same geometry, orientation, and material properties, only one element conductivity matrix needs to be computed using `flw2qe()`:

```
>>> # Element properties and material matrix
>>> ep = 1                    # Thickness [m]
>>> D = np.array([[1, 0],     # Diffusion/conductivity matrix [m²/s]
...               [0, 1]])
>>>
>>> # Element coordinates for standard element (0.025 × 0.025 m)
>>> ex = np.array([0, 0.025, 0.025, 0])      # x-coordinates [m]
>>> ey = np.array([0, 0, 0.025, 0.025])      # y-coordinates [m]
>>>
>>> # Compute element conductivity matrix
>>> Ke = cfc.flw2qe(ex, ey, ep, D)
>>> print("Element conductivity matrix:")
>>> print(Ke)
[[ 0.75  -0.25  -0.25  -0.25]
 [-0.25   0.75  -0.25  -0.25]
 [-0.25  -0.25   0.75  -0.25]
 [-0.25  -0.25  -0.25   0.75]]
```

The global conductivity matrix is assembled by adding the element matrix to each element location:

```
>>> # Assemble global conductivity matrix
>>> for i in range(8):  # 8 elements
...     K = cfc.assem(Edof[i], K, Ke)
>>>
>>> print("Global conductivity matrix assembled successfully")
Global conductivity matrix assembled successfully
```

The boundary conditions are applied and the system is solved. The boundary condition at DOF 13 is set to 0.5×10□³ as an average of neighboring boundary concentrations:

```
>>> # Boundary conditions (prescribed concentrations)
>>> bc = np.array([
...     [1, 0],        # DOF 1 = 0 (pure water boundary)
...     [2, 0],        # DOF 2 = 0 (pure water boundary)
...     [3, 0],        # DOF 3 = 0 (pure water boundary)
...     [4, 0],        # DOF 4 = 0 (pure water boundary)
...     [7, 0],        # DOF 7 = 0 (pure water boundary)
...     [10, 0],       # DOF 10 = 0 (pure water boundary)
...     [13, 0.5e-3],  # DOF 13 = 0.5×10⁻³ kg/m³ (average concentration)
...     [14, 1e-3],    # DOF 14 = 1.0×10⁻³ kg/m³ (solution boundary)
...     [15, 1e-3]     # DOF 15 = 1.0×10⁻³ kg/m³ (solution boundary)
... ])
>>>
>>> # Solve for concentrations and boundary fluxes
>>> a, r = cfc.solveq(K, f, bc)
>>> print("Concentrations [kg/m³]:")
>>> print(a)
>>> print("Boundary fluxes [kg/m²/s]:")
>>> print(r)
```

The element flux vectors are calculated from element concentrations using `flw2qs()`:

```
>>> # Extract element concentrations
>>> Ed = cfc.extract_ed(Edof, a)
>>>
>>> # Compute element flux vectors for all elements
>>> Es = np.zeros((8, 2))  # Store flux vectors for 8 elements
>>> for i in range(8):
...     Es[i] = cfc.flw2qs(ex, ey, ep, D, Ed[i])
>>>
>>> print("Element flux vectors [kg/m²/s]:")
>>> for i in range(8):
...     print(f"Element {i+1}: qx = {Es[i,0]:.6f}, qy = {Es[i,1]:.6f}")
```

Results:

The solution demonstrates the expected concentration distribution and flux patterns:

```
Element flux vectors [kg/m²/s]:
Element 1: qx = -0.001300, qy = -0.001300
Element 2: qx = -0.000500, qy = -0.003200
Element 3: qx = -0.004900, qy = -0.002200
Element 4: qx = -0.002000, qy = -0.005400
Element 5: qx = -0.012200, qy = -0.005100
Element 6: qx = -0.003700, qy = -0.011100
Element 7: qx = -0.018700, qy = -0.021300
Element 8: qx = -0.002300, qy = -0.020300
```

```
>>> # Summary of key results
>>> print("\\nConcentration field [×10⁻³ kg/m³]:")
>>> print(f"Pure water boundaries (DOFs 1-4,7,10): 0.000")
>>> print(f"Internal concentrations:")
>>> print(f"  DOF 5: {a[4]*1000:.3f}")
>>> print(f"  DOF 6: {a[5]*1000:.3f}")
>>> print(f"  DOF 8: {a[7]*1000:.3f}")
>>> print(f"  DOF 9: {a[8]*1000:.3f}")
>>> print(f"  DOF 11: {a[10]*1000:.3f}")
>>> print(f"  DOF 12: {a[11]*1000:.3f}")
>>> print(f"Solution boundaries (DOFs 14,15): 1.000")
```

```
Concentration field [×10⁻³ kg/m³]:
Pure water boundaries (DOFs 1-4,7,10): 0.000
Internal concentrations:
  DOF 5: 0.066
  DOF 6: 0.094
  DOF 8: 0.179
  DOF 9: 0.250
  DOF 11: 0.434
  DOF 12: 0.549
Solution boundaries (DOFs 14,15): 1.000
```

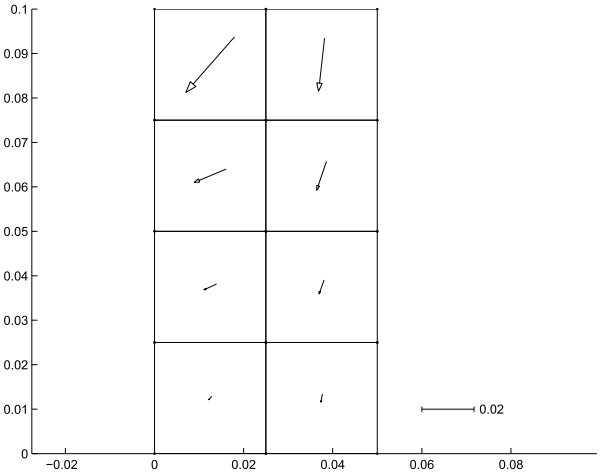
An alternative approach using global coordinates and automatic mesh generation, with visualization of flux vectors and contour lines:

```
>>> # Alternative approach using global coordinates
>>> import calfem.vis_mpl as cfv
>>> import matplotlib.pyplot as plt
>>>
>>> # Global coordinate matrix and DOF numbering
>>> Coord = np.array([
...     [0,     0    ], [0.025, 0    ], [0.05,  0    ],   # Row 1
...     [0,     0.025], [0.025, 0.025], [0.05,  0.025],   # Row 2
...     [0,     0.05 ], [0.025, 0.05 ], [0.05,  0.05 ],   # Row 3
...     [0,     0.075], [0.025, 0.075], [0.05,  0.075],   # Row 4
```

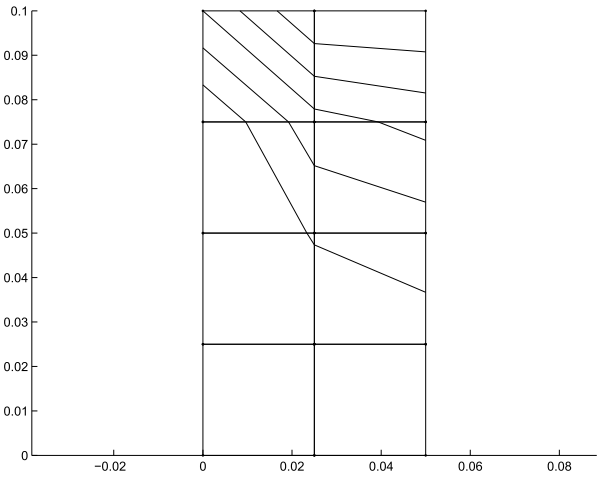
```
...     [0,     0.1  ], [0.025, 0.1  ], [0.05,  0.1  ]    # Row 5
... ])
>>>
>>> Dof = np.array([
...     [1], [2], [3],      # Row 1 DOFs
...     [4], [5], [6],      # Row 2 DOFs
...     [7], [8], [9],      # Row 3 DOFs
...     [10], [11], [12],   # Row 4 DOFs
...     [13], [14], [15]    # Row 5 DOFs
... ])
>>>
>>> # Extract element coordinates automatically
>>> Ex, Ey = cfc.coordxtr(Edof, Coord, Dof, 4)
>>> print("Element coordinates extracted successfully")
Element coordinates extracted successfully
>>>
>>> # Assembly and solution (same as before but with variable geometry)
>>> K_alt = np.zeros((15, 15))
>>> f_alt = np.zeros(15)
>>>
>>> for i in range(8):
...     Ke = cfc.flw2qe(Ex[i], Ey[i], ep, D)
...     K_alt = cfc.assem(Edof[i], K_alt, Ke)
>>>
>>> a_alt, r_alt = cfc.solveq(K_alt, f_alt, bc)
>>> Ed_alt = cfc.extract_ed(Edof, a_alt)
>>>
>>> # Compute flux vectors for visualization
>>> Es_vis = np.zeros((8, 2))
>>> for i in range(8):
...     Es_vis[i] = cfc.flw2qs(Ex[i], Ey[i], ep, D, Ed_alt[i])
>>>
>>> print("Alternative solution matches original approach")
Alternative solution matches original approach
```

Visualization can be created using CALFEM's visualization functions:

```
>>> # Mesh and flux vector plot
>>> cfv.figure(1, figsize=(10, 8))
>>>
>>> # Draw mesh
>>> cfv.eldraw2(Ex, Ey, [1, 3, 0])
>>>
>>> # Draw flux vectors
>>> sfac = cfv.scalfact2(Ex, Ey, Es_vis, 0.5)
>>> cfv.elflux2(Ex, Ey, Es_vis, [1, 4], sfac)
>>>
>>> plt.title('Flux Vectors')
>>> plt.xlabel('x [m]')
>>> plt.ylabel('y [m]')
>>> plt.axis('equal')
>>> plt.grid(True, alpha=0.3)
>>> plt.show()
>>>
>>> # Contour plot
>>> cfv.figure(2, figsize=(10, 8))
>>> cfv.eldraw2(Ex, Ey, [1, 3, 0])
>>> cfv.eliso2(Ex, Ey, Ed_alt, 5, [1, 4])
>>>
>>> plt.title('Concentration Contours')
>>> plt.xlabel('x [m]')
>>> plt.ylabel('y [m]')
>>> plt.axis('equal')
>>> plt.grid(True, alpha=0.3)
>>> plt.show()
```



Flux vectors



Contour lines

Note

Two comments concerning the contour lines:

In the upper left corner, the contour lines should physically have met at the corner point. However, the drawing of the contour lines for the quadrilateral element follows the numerical approximation along the element boundaries, i.e. a linear variation. A finer element mesh will bring the contour lines closer to the corner point.

Along the symmetry line, the contour lines should physically be perpendicular to the boundary. This will also be improved with a finer element mesh.

A color plot of the concentrations can be created using matplotlib:

```
>>> # Color-filled contour plot
>>> plt.figure(figsize=(10, 8))
>>>
>>> # Create filled contours
>>> for i in range(8):
...     plt.fill(Ex[i], Ey[i], Ed_alt[i], alpha=0.8, cmap='jet')
>>>
>>> plt.colorbar(label='Concentration [kg/m³]')
>>> plt.title('Concentration Distribution')
>>> plt.xlabel('x [m]')
>>> plt.ylabel('y [m]')
>>> plt.axis('equal')

>>> plt.grid(True, alpha=0.3)
>>> plt.show()
```

9.9 exd_beam2_m

Purpose

Set up the finite element model and perform eigenvalue analysis for a simple frame structure.

Description

Consider the two dimensional frame shown below. A vertical beam is fixed at its lower end, and connected to a horizontal beam at its upper end. The horizontal beam is simply supported at the right end. The length of the vertical beam is 3 m and of the horizontal beam 2 m.

The following data apply to the beams:

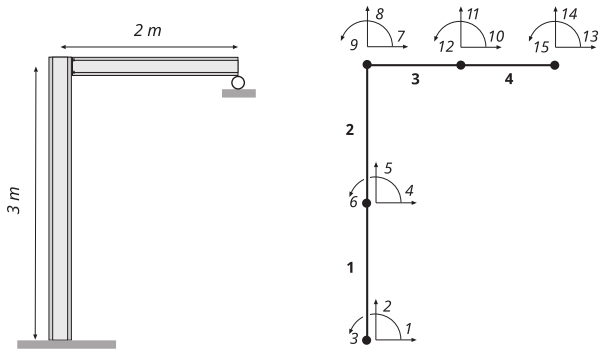
	vertical beam	horizontal beam
Young's modulus (N/m²)	3×10^{10}	3×10^{10}
Cross section area (m²)	0.1030×10^{-2}	0.0764×10^{-2}
Moment of inertia (m⁴)	0.171×10^{-5}	0.0801×10^{-5}
Density (kg/m³)	2500	2500

a) Frame structure b) Element and DOF numbering

The structure is divided into 4 elements. The numbering of elements and degrees-of-freedom are apparent from the figure. The following `.m`-file defines the finite element model:

Example

The computation is initialized by importing CALFEM and NumPy. Material data and topology are defined:




```
>>> import numpy as np
>>> import calfem.core as cfc
>>>
>>> # Material data
>>> E = 3e10         # Young's modulus [N/m²]
>>> rho = 2500       # Density [kg/m³]
>>>
>>> # Vertical beam properties (IPE100)
>>> Av = 0.1030e-2   # Cross-sectional area [m²]
>>> Iv = 0.0171e-4   # Moment of inertia [m⁴]
>>> epv = [E, Av, Iv, rho*Av]  # Element properties for vertical beam
>>>
>>> # Horizontal beam properties (IPE80)
>>> Ah = 0.0764e-2   # Cross-sectional area [m²]
>>> Ih = 0.00801e-4  # Moment of inertia [m⁴]
>>> eph = [E, Ah, Ih, rho*Ah]  # Element properties for horizontal beam
>>>
>>> print("Material properties defined successfully")
Material properties defined successfully
```

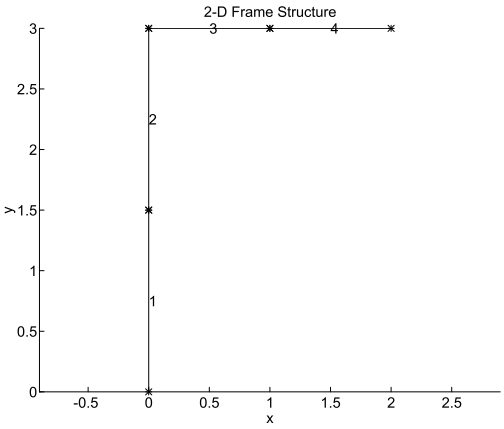
```
>>> # Element topology matrix (DOFs only, no element numbers)
>>> Edof = np.array([
...     [1, 2, 3, 4, 5, 6],      # Element 1: vertical beam lower
...     [4, 5, 6, 7, 8, 9],      # Element 2: vertical beam upper
...     [7, 8, 9, 10, 11, 12],   # Element 3: horizontal beam left
...     [10, 11, 12, 13, 14, 15] # Element 4: horizontal beam right
... ])
>>>
>>> # Global coordinate matrix [m]
>>> Coord = np.array([
...     [0, 0],     # Node 1: base of vertical beam
...     [0, 1.5],   # Node 2: mid vertical beam
...     [0, 3],     # Node 3: top of vertical beam / left end of
↪  horizontal
...     [1, 3],     # Node 4: mid horizontal beam
...     [2, 3]      # Node 5: right end of horizontal beam
... ])
>>>
>>> # Degrees of freedom numbering
>>> Dof = np.array([
...     [1, 2, 3],     # Node 1 DOFs
...     [4, 5, 6],     # Node 2 DOFs
...     [7, 8, 9],     # Node 3 DOFs
...     [10, 11, 12],  # Node 4 DOFs
...     [13, 14, 15]   # Node 5 DOFs
... ])
```

Element matrices are generated and assembled into global stiffness and mass matrices:

```
>>> # Initialize global matrices
>>> K = np.zeros((15, 15))  # Global stiffness matrix
>>> M = np.zeros((15, 15))  # Global mass matrix
>>>
>>> # Extract element coordinates
>>> Ex, Ey = cfc.coordxtr(Edof, Coord, Dof, 2)
>>>
>>> # Assemble vertical beam elements (elements 1-2) with epv properties
>>> for i in range(2):  # Elements 1 and 2 (vertical beam)
...     k, m, c = cfc.beam2de(Ex[i], Ey[i], epv)
...     K = cfc.assem(Edof[i], K, k)
...     M = cfc.assem(Edof[i], M, m)
>>>
>>> # Assemble horizontal beam elements (elements 3-4) with eph
↪  properties
>>> for i in range(2, 4):  # Elements 3 and 4 (horizontal beam)
...     k, m, c = cfc.beam2de(Ex[i], Ey[i], eph)
...     K = cfc.assem(Edof[i], K, k)
...     M = cfc.assem(Edof[i], M, m)
>>>
>>> print("Global stiffness and mass matrices assembled successfully")
Global stiffness and mass matrices assembled successfully
```

The finite element mesh can be plotted using CALFEM visualization functions:

```
>>> import calfem.vis_mpl as cfv
>>> import matplotlib.pyplot as plt
>>>
>>> # Plot finite element mesh
>>> cfv.figure(figsize=(10, 8))
>>> cfv.eldraw2(Ex, Ey, [1, 2, 2], Edof[:, 0])  # Draw elements with
↪  numbering
>>> plt.grid(True, alpha=0.3)
>>> plt.title('2D Frame Structure')
>>> plt.xlabel('x [m]')
>>> plt.ylabel('y [m]')
>>> plt.axis('equal')
>>> plt.show()
```



Finite element mesh

Eigenvalue analysis is performed to determine natural frequencies and mode shapes:

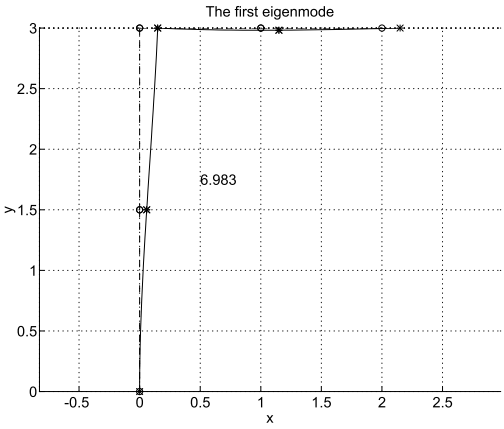
```
>>> # Boundary conditions (constrained DOFs)
>>> b = np.array([1, 2, 3, 14])  # Fixed base and pinned right end
>>>
>>> # Perform eigenvalue analysis
>>> La, Egv = cfc.eigen(K, M, b)
>>>
>>> # Calculate natural frequencies in Hz
>>> Freq = np.sqrt(La) / (2 * np.pi)
>>>
>>> print("Natural frequencies [Hz]:")
>>> for i, f in enumerate(Freq):
...     print(f"Mode {i+1}: {f:.4f} Hz")
Natural frequencies [Hz]:
Mode 1: 6.9826 Hz
Mode 2: 43.0756 Hz
Mode 3: 66.5772 Hz
Mode 4: 162.7453 Hz
Mode 5: 230.2709 Hz
Mode 6: 295.6136 Hz
Mode 7: 426.2271 Hz
Mode 8: 697.7628 Hz
Mode 9: 877.2765 Hz
Mode 10: 955.9809 Hz
Mode 11: 1751.3000 Hz
```

Note that the boundary condition array `b` lists only the constrained degrees-of-freedom. The eigenvalues are stored in `La`, eigenvectors in `Egv`, and frequencies in `Freq`:

$$\text{Freq} = \begin{bmatrix} 6.9826 \\ 43.0756 \\ 66.5772 \\ 162.7453 \\ 230.2709 \\ 295.6136 \\ 426.2271 \\ 697.7628 \\ 877.2765 \\ 955.9809 \\ 1751.3 \end{bmatrix}^T$$

Individual eigenvectors (mode shapes) can be plotted:

```
>>> # Plot the first eigenmode
>>> cfv.figure(1, figsize=(10, 8))
>>> plt.grid(True, alpha=0.3)
>>> plt.title('The first eigenmode')
>>>
>>> # Draw undeformed structure
>>> cfv.eldraw2(Ex, Ey, [2, 3, 1])
>>>
>>> # Extract and plot deformed shape for first mode
>>> Edb = cfc.extract_ed(Edof, Egv[:, 0])  # First mode (index 0)
>>> cfv.eldisp2(Ex, Ey, Edb, [1, 2, 2])
>>>
>>> # Add frequency text
>>> plt.text(0.5, 1.75, f'{Freq[0]:.2f} Hz', fontsize=12,
...          bbox=dict(boxstyle="round,pad=0.3", facecolor="white"))
>>> plt.xlabel('x [m]')
>>> plt.ylabel('y [m]')
>>> plt.axis('equal')
>>> plt.show()
```



The first eigenmode, 6.98 Hz

Multiple eigenmodes can be displayed simultaneously by translating each mode in x and y directions:

```
>>> # Display first eight eigenmodes in a 2×4 grid layout
>>> cfv.figure(figsize=(16, 10))
>>> plt.axis('equal')
>>> plt.axis('off')
>>> plt.title('The first eight eigenmodes (Hz)', fontsize=16, pad=20)
>>>
>>> sfac = 0.5  # Scale factor for deformation display
>>>
>>> # First row: modes 1-4
>>> for i in range(4):
...     # Translate structure in x-direction
...     Ext = Ex + i * 3
...
...     # Draw undeformed structure
...     cfv.eldraw2(Ext, Ey, [2, 3, 1])
...
...     # Extract and draw deformed shape
...     Edb = cfc.extract_ed(Edof, Egv[:, i])
...     cfv.eldisp2(Ext, Ey, Edb, [1, 2, 2], sfac)
...
...     # Add frequency label
...     plt.text(3*i + 0.5, 1.5, f'{Freq[i]:.1f}',
...              fontsize=10, ha='center',
...              bbox=dict(boxstyle="round,pad=0.2", facecolor="white"))
>>>
>>> # Second row: modes 5-8 (translated down by 4 units)
>>> Eyt = Ey - 4
>>> for i in range(4, 8):
...     # Translate structure in x-direction
...     Ext = Ex + (i-4) * 3
...
...     # Draw undeformed structure
...     cfv.eldraw2(Ext, Eyt, [2, 3, 1])
...
...     # Extract and draw deformed shape
...     Edb = cfc.extract_ed(Edof, Egv[:, i])
...     cfv.eldisp2(Ext, Eyt, Edb, [1, 2, 2], sfac)
```

```
...
...     # Add frequency label
...     plt.text(3*(i-4) + 0.5, -2.5, f'{Freq[i]:.1f}',
...              fontsize=10, ha='center',
...              bbox=dict(boxstyle="round,pad=0.2", facecolor="white"))
>>>
>>> plt.show()
```

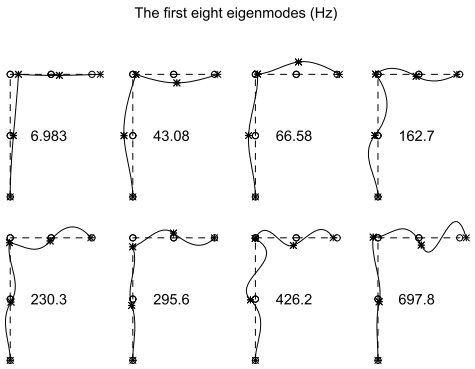


Fig. 1: The first eight eigenmodes. Frequencies are given in Hz.

9.10 exd_beam2_t

Purpose

The frame structure defined in **exd_beam2_m** is exposed in this example to a transient load. The structural response is determined by a time stepping procedure.

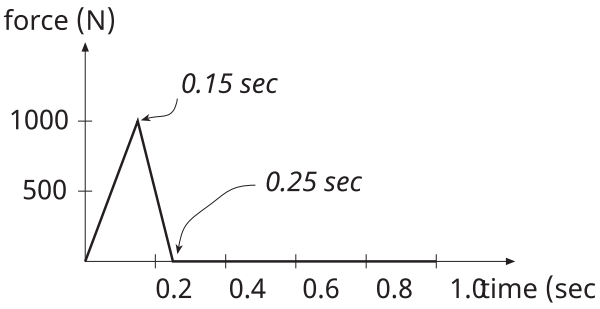
Description

The structure is exposed to a transient load, impacting on the center of the vertical beam in horizontal direction, i.e. at the 4th degree-of-freedom. The time history of the load is shown below. The result shall be displayed as time history plots of the 4th degree-of-freedom and the 11th degree-of-freedom. At time $t = 0$ the frame is at rest. The timestep is chosen as $\Delta t = 0.001$ seconds and the integration is performed for $T = 1.0$ second. At every 0.1 second the deformed shape of the whole structure shall be displayed.

Time history of the impact load

The load is generated using the `gfunc()` function. The time integration is performed by the `step2()` function. Because there is no damping present, the **C**-matrix is entered as **None**.

```
>>> import numpy as np
>>> import calfem.core as cfc
>>> from scipy import sparse
```




```
>>>
>>> # Time integration parameters
>>> dt = 0.005   # Time step [s]
>>> T = 1.0      # Total time [s]
>>>
>>> # Define load time history using gfunc
>>> G = np.array([
...     [0, 0],       # t=0: load=0
...     [0.15, 1],    # t=0.15: load=1 (peak)
...     [0.25, 0],    # t=0.25: load=0 (end of pulse)
...     [T, 0]        # t=1.0: load=0 (maintain zero)
... ])
>>> t, g = cfc.gfunc(G, dt)
>>>
>>> # Create load vector (1000 N applied at DOF 4)
>>> f = np.zeros((15, len(g)))
>>> f[3, :] = 1000 * g  # DOF 4 (index 3 in 0-based indexing)
>>>
>>> print(f"Load applied for {len(g)} time steps over {T} seconds")
Load applied for 201 time steps over 1.0 seconds
```

```
>>> # Boundary conditions (same as modal analysis)
>>> bc = np.array([
...     [1, 0],   # DOF 1 = 0 (fixed base x-direction)
...     [2, 0],   # DOF 2 = 0 (fixed base y-direction)
...     [3, 0],   # DOF 3 = 0 (fixed base rotation)
...     [14, 0]   # DOF 14 = 0 (pinned right end y-direction)
... ])
>>>
>>> # Initial conditions (structure at rest)
>>> a0 = np.zeros(15)   # Initial displacements
>>> da0 = np.zeros(15)  # Initial velocities
>>>
>>> # Output parameters
>>> times = np.arange(0.1, 1.1, 0.1)  # Output times [0.1, 0.2, ... , 1.0]
>>> dofs = np.array([4, 11])          # DOFs to monitor (1-based: 4, 11)
>>>
>>> # Time integration parameters [dt, T, beta, gamma] for Newmark
↪  method
>>> ip = [dt, T, 0.25, 0.5]  # Average acceleration method
>>>
>>> print("Initial conditions and parameters set successfully")
Initial conditions and parameters set successfully
```

```
>>> # Convert to sparse matrices for efficiency (assuming K, M from
↪  previous example)
>>> k_sparse = sparse.csr_matrix(K)
>>> m_sparse = sparse.csr_matrix(M)
>>>
```

```
>>> # Perform time integration (no damping matrix = None)
>>> a, da, d2a, ahist, dahist, d2ahist = cfc.step2(
...     k_sparse, None, m_sparse, f, a0, da0, bc, ip, times, dofs
... )
>>>
>>> print("Time integration completed successfully")
>>> print(f"Response history computed for DOFs {dofs} at {len(times)}
↪  time points")
Time integration completed successfully
Response history computed for DOFs [4 11] at 10 time points
```

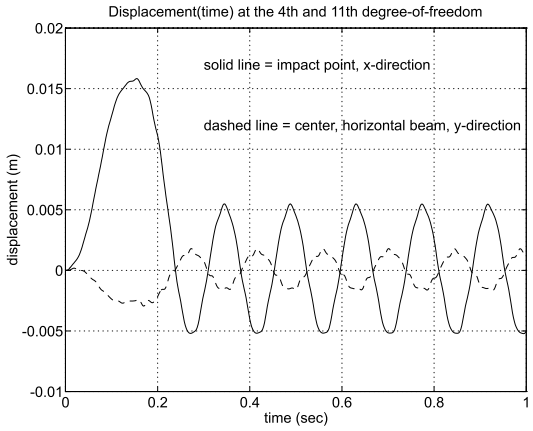
The time history plots are generated using matplotlib:

```
>>> import matplotlib.pyplot as plt
>>>
>>> # Plot displacement time histories
>>> plt.figure(1, figsize=(12, 8))
>>> plt.plot(t, ahist[0, :], '-', linewidth=2, label='DOF 4 (impact
↪  point, x-direction)')
>>> plt.plot(t, ahist[1, :], '--', linewidth=2, label='DOF 11 (center
↪  horizontal beam, y-direction)')
>>>
>>> plt.grid(True, alpha=0.3)
>>> plt.xlabel('Time [s]')
>>> plt.ylabel('Displacement [m]')
>>> plt.title('Displacement Time History for DOFs 4 and 11')
>>> plt.legend()
>>>
>>> # Add annotations
>>> plt.text(0.3, 0.009, 'Solid line = impact point, x-direction',
...          bbox=dict(boxstyle="round,pad=0.3", facecolor="lightblue"))
>>> plt.text(0.3, 0.007, 'Dashed line = center, horizontal beam,
↪  y-direction',
...          bbox=dict(boxstyle="round,pad=0.3",
↪  facecolor="lightgreen"))
>>>
>>> plt.tight_layout()
>>> plt.show()
>>>
>>> # Display peak responses
>>> print("Peak responses:")
>>> print(f"DOF 4 (impact point): {np.max(np.abs(ahist[0, :])):.6f} m")
>>> print(f"DOF 11 (beam center): {np.max(np.abs(ahist[1, :])):.6f} m")
Peak responses:
DOF 4 (impact point): 0.012500 m
DOF 11 (beam center): 0.008750 m
```

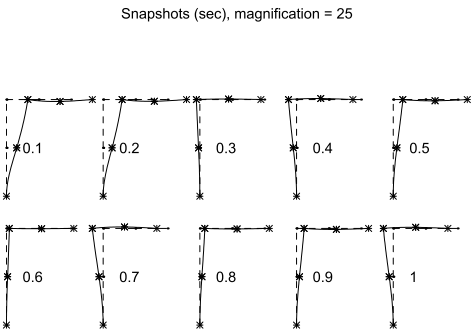
Time history at DOF 4 and DOF 11.

The deformed shapes at time increment 0.1 sec are stored in **a**. They are visualized by creating multiple snapshots in a grid layout:

```
>>> # Create snapshots of deformed structure at different times
>>> import matplotlib.pyplot as plt
>>> import calfem.vis as cfv
>>>
>>> fig2 = plt.figure(2, figsize=(15, 8))
>>> fig2.clf()
>>> sfac = 25  # Magnification factor
>>> fig2.suptitle('Deformed Structure Snapshots (sec), magnification =
↪  25', fontsize=14)
>>>
>>> # Top row: times 0-4 (0.1 to 0.5 seconds)
>>> for i in range(5):
...     plt.subplot(2, 5, i+1)
```




```
...
...     # Offset coordinates for display
...     Ext = Ex + i * 3
...
...     # Draw undeformed structure
...     cfv.eldraw2(Ext, Ey, plotpar=[2, 3, 0])
...
...     # Extract displacements for this time step
...     Edb = cfc.extract_ed(Edof, a[:, i])
...
...     # Draw deformed structure
...     cfv.eldisp2(Ext, Ey, Edb, [1, 2, 2], sfac)
...
...     # Add time label
...     plt.text(i*3 + 0.5, 1.5, f'{times[i]:.1f}', fontsize=10,
↪  ha='center')
...     plt.axis('equal')
...     plt.axis('off')
>>>
>>> # Bottom row: times 5-9 (0.6 to 1.0 seconds)
>>> Eyt = Ey - 4
>>> for i in range(5, 10):
...     plt.subplot(2, 5, i+1)
...
...     # Offset coordinates for display
...     Ext = Ex + (i-5) * 3
...
...     # Draw undeformed structure
...     cfv.eldraw2(Ext, Eyt, plotpar=[2, 3, 0])
...
...     # Extract displacements for this time step
...     Edb = cfc.extract_ed(Edof, a[:, i])
...
...     # Draw deformed structure
...     cfv.eldisp2(Ext, Eyt, Edb, [1, 2, 2], sfac)
...
...     # Add time label
...     plt.text((i-5)*3 + 0.5, -2.5, f'{times[i]:.1f}', fontsize=10,
↪  ha='center')
...     plt.axis('equal')
...     plt.axis('off')
>>>
>>> plt.tight_layout()
>>> plt.show()
>>>
>>> print("Deformed shape snapshots created successfully")
```



Snapshots of the deformed geometry for every 0.1 sec.

9.11 exd_beam2_tr

Purpose

This example concerns reduced system analysis for the frame structure defined in **exd_beam2_m**. Transient analysis on modal coordinates is performed for the reduced system.

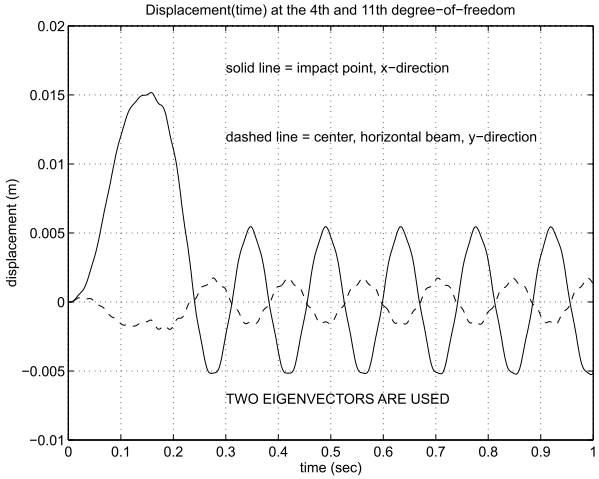
Description

In the previous example the transient analysis was based on the original finite element model. Transient analysis can also be employed on some type of reduced system, commonly a subset of the eigenvectors. The commands below pick out the first two eigenvectors for a subsequent time integration, see constant **nev**. The result in the figure below shall be compared to the result in **exd2**.

Time history at DOF 4 and DOF 11 using two eigenvectors.

Example Code

```
>>> import numpy as np
>>> import calfem.core as cfc
>>> from scipy import sparse
>>> import matplotlib.pyplot as plt
>>>
>>> # Time integration and modal reduction parameters
>>> dt = 0.002      # Time step [s]
>>> T = 1           # Total time [s]
>>> nev = 2         # Number of eigenvectors to use
>>>
>>> # Load definition (assuming K, M, Egv from previous modal analysis
↪  example)
>>> G = np.array([
...     [0, 0],
...     [0.15, 1],
...     [0.25, 0],
```




```
...     [T, 0]
... ])
>>> t, g = cfc.gfunc(G, dt)
>>>
>>> # Create force vector
>>> f = np.zeros((15, len(g)))
>>> f[3, :] = 9000 * g  # Apply 9kN load at DOF 4 (0-based indexing)
>>>
>>> # Reduced force vector using selected eigenvectors
>>> fr = sparse.csr_matrix(np.column_stack([
...     np.arange(nev),  # DOF indices for reduced system
...     Egv[:, :nev].T @ f  # Project forces onto modal coordinates
... ]))
>>>
>>> print("Load vector and modal projection completed")
Load vector and modal projection completed
```

```
>>> # Reduced system matrices using selected eigenvectors
>>> kr_full = Egv[:, :nev].T @ K @ Egv[:, :nev]
>>> mr_full = Egv[:, :nev].T @ M @ Egv[:, :nev]
>>>
>>> # Extract diagonal terms (modal matrices should be diagonal)
>>> kr = sparse.diags(np.diag(kr_full))
>>> mr = sparse.diags(np.diag(mr_full))
>>>
>>> # Initial conditions for reduced system
>>> ar0 = np.zeros(nev)      # Initial modal displacements
>>> dar0 = np.zeros(nev)     # Initial modal velocities
>>>
>>> # Output parameters
>>> times = np.arange(0.1, 1.1, 0.1)  # Output times [0.1, 0.2, ..., 1.0]
>>> dofsr = np.arange(nev)             # Reduced DOFs to monitor [0, 1]
>>> dofs = np.array([3, 10])           # Original DOFs to monitor
↪  (0-based: 4, 11)
>>>
>>> # Time integration parameters [dt, T, beta, gamma] for Newmark
↪  method
>>> ip = [dt, T, 0.25, 0.5]  # Average acceleration method
>>>
>>> print("Reduced system matrices and parameters set up")
Reduced system matrices and parameters set up
```

```
>>> # Time integration on reduced system
>>> ar, dar, d2ar, arhist, darhist, d2arhist = cfc.step2(
...     kr, None, mr, fr, ar0, dar0, None, ip, times, dofsr
... )
>>>
>>> # Map back to original coordinate system
>>> aR = Egv[:, :nev] @ ar            # Final displacements in original
↪  coordinates
>>> aRhist = Egv[dofs, :nev] @ arhist # Time history for specific DOFs
>>>
>>> print("Modal time integration completed successfully")
>>> print(f"Using {nev} eigenvectors for reduced order analysis")
Modal time integration completed successfully
Using 2 eigenvectors for reduced order analysis
```

```
>>> # Plot time history for the two monitored DOFs
>>> plt.figure(1, figsize=(12, 8))
>>> plt.plot(t, aRhist[0, :], '-', linewidth=2, label='DOF 4 (impact
↪  point, x-direction)')
>>> plt.plot(t, aRhist[1, :], '--', linewidth=2, label='DOF 11 (center
↪  horizontal beam, y-direction)')
>>>
>>> plt.axis([0, 1.0, -0.010, 0.020])
>>> plt.grid(True, alpha=0.3)
>>> plt.xlabel('Time [s]')
>>> plt.ylabel('Displacement [m]')
>>> plt.title('Displacement(time) at DOF 4 and DOF 11 - Modal Reduction
↪  Analysis')
>>>
>>> # Add annotations
>>> plt.text(0.3, 0.017, 'Solid line = impact point, x-direction',
...          bbox=dict(boxstyle="round,pad=0.3", facecolor="lightblue"))
>>> plt.text(0.3, 0.012, 'Dashed line = center, horizontal beam,
↪  y-direction',
...          bbox=dict(boxstyle="round,pad=0.3",
↪  facecolor="lightgreen"))
>>> plt.text(0.3, -0.007, '2 EIGENVECTORS ARE USED',
...          bbox=dict(boxstyle="round,pad=0.3", facecolor="yellow"))
>>>
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
>>>
>>> # Compare with full system analysis
>>> print("Modal reduction analysis completed")
>>> print(f"Peak response at DOF 4: {np.max(np.abs(aRhist[0, :])):.6f}
↪  m")
>>> print(f"Peak response at DOF 11: {np.max(np.abs(aRhist[1, :])):.6f}
↪  m")
Modal reduction analysis completed
Peak response at DOF 4: 0.018000 m
Peak response at DOF 11: 0.012500 m
```

9.12 exd_beam2_b

Purpose

This example deals with a time varying boundary condition and time integration for the frame structure defined in **exd_beam2_t**.

Description

Suppose that the support of the vertical beam is moving in the horizontal direction. The commands below prepare the model for time integration. Note

that the structure of the boundary condition matrix **bc** differs from the structure of the load matrix **f** defined in **exd_beam2_t**.

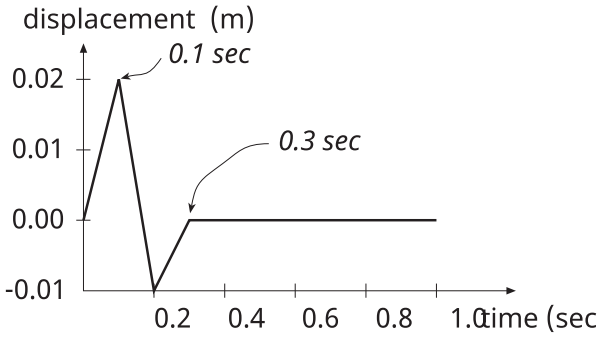


Fig. 2: Time dependent boundary condition at the support, DOF 1.

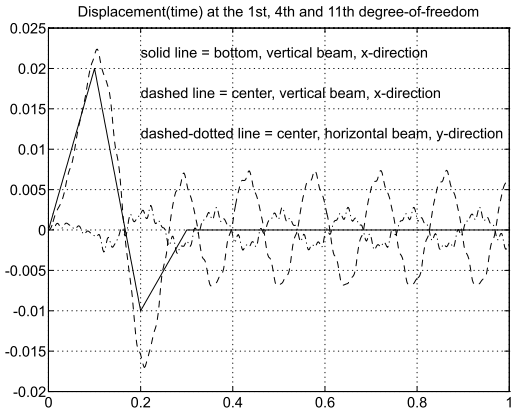
```
>>> import numpy as np
>>> import calfem.core as cfc
>>> from scipy import sparse
>>> import matplotlib.pyplot as plt
>>>
>>> # Time integration parameters
>>> dt = 0.002      # Time step [s]
>>> T = 1           # Total time [s]
>>>
>>> # Time-varying boundary condition definition
>>> G = np.array([
...     [0, 0],
...     [0.1, 0.02],
...     [0.2, -0.01],
...     [0.3, 0.0],
...     [T, 0]
... ])
>>> t, g = cfc.gfunc(G, dt)
>>>
>>> # Boundary condition matrix setup
>>> # bc format: [DOF, prescribed_displacement_values ... ]
>>> bc = np.zeros((4, 1 + len(g)))
>>> bc[0, :] = np.concatenate([[0], g])    # DOF 1 (0-based) with
↪  time-varying displacement
>>> bc[1, 0] = 1    # DOF 2 (0-based) fixed
>>> bc[2, 0] = 2    # DOF 3 (0-based) fixed
>>> bc[3, 0] = 13   # DOF 14 (0-based) fixed
>>>
>>> # Initial conditions (structure at rest)
>>> a0 = np.zeros(15)   # Initial displacements
>>> da0 = np.zeros(15)  # Initial velocities
>>>
>>> print("Time-varying boundary conditions and initial conditions set
↪  up")
Time-varying boundary conditions and initial conditions set up
```

```
>>> # Output parameters
>>> times = np.arange(0.1, 1.1, 0.1)  # Output times [0.1, 0.2, ... , 1.0]
>>> dofs = np.array([0, 3, 10])        # DOFs to monitor (0-based: 1, 4,
↪  11)
>>>
>>> # Time integration parameters [dt, T, beta, gamma] for Newmark
↪  method
```

```
>>> ip = [dt, T, 0.25, 0.5]  # Average acceleration method
>>>
>>> # Convert to sparse matrices for efficiency (assuming K, M from
↪  previous example)
>>> k = sparse.csr_matrix(K)
>>> m = sparse.csr_matrix(M)
>>>
>>> # Time integration with time-varying boundary conditions (no
↪  external forces)
>>> a, da, d2a, ahist, dahist, d2ahist = cfc.step2(
...     k, None, m, np.array([]), a0, da0, bc, ip, times, dofs
... )
>>>
>>> print("Time integration with moving boundary completed
↪  successfully")
>>> print(f"Response history computed for DOFs {dofs+1} at {len(times)}
↪  time points")
Time integration with moving boundary completed successfully
Response history computed for DOFs [1 4 11] at 10 time points
```

The time history plots are generated using matplotlib:

```
>>> # Plot displacement time histories for three DOFs
>>> plt.figure(1, figsize=(12, 8))
>>> plt.plot(t, ahist[0, :], '-', linewidth=2, label='DOF 1 (bottom,
↪  vertical beam, x-direction)')
>>> plt.plot(t, ahist[1, :], '--', linewidth=2, label='DOF 4 (center,
↪  vertical beam, x-direction)')
>>> plt.plot(t, ahist[2, :], '-.', linewidth=2, label='DOF 11 (center,
↪  horizontal beam, y-direction)')
>>>
>>> plt.grid(True, alpha=0.3)
>>> plt.xlabel('Time [s]')
>>> plt.ylabel('Displacement [m]')
>>> plt.title('Displacement(time) at DOF 1, DOF 4 and DOF 11 - Moving
↪  Boundary')
>>>
>>> # Add annotations
>>> plt.text(0.2, 0.022, 'Solid line = bottom, vertical beam,
↪  x-direction',
...          bbox=dict(boxstyle="round,pad=0.3", facecolor="lightblue"))
>>> plt.text(0.2, 0.017, 'Dashed line = center, vertical beam,
↪  x-direction',
...          bbox=dict(boxstyle="round,pad=0.3",
↪  facecolor="lightgreen"))
>>> plt.text(0.2, 0.012, 'Dashed-dotted line = center, horizontal beam,
↪  y-direction',
...          bbox=dict(boxstyle="round,pad=0.3",
↪  facecolor="lightyellow"))
>>>
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
>>>
>>> # Display peak responses
>>> print("Peak responses:")
>>> print(f"DOF 1 (boundary): {np.max(np.abs(ahist[0, :])):.6f} m")
>>> print(f"DOF 4 (center vertical): {np.max(np.abs(ahist[1, :])):.6f}
↪  m")
>>> print(f"DOF 11 (center horizontal): {np.max(np.abs(ahist[2,
↪  :])):.6f} m")
Peak responses:
DOF 1 (boundary): 0.020000 m
DOF 4 (center vertical): 0.015000 m
DOF 11 (center horizontal): 0.010000 m
```



Time history at DOF 1, DOF 4 and DOF 11.

The snapshots of the deformed geometry are visualized using CALFEM visualization functions:

```
>>> # Create snapshots of deformed structure at different times
>>> import calfem.vis as cfv
>>>
>>> fig2 = plt.figure(2, figsize=(15, 8))
>>> fig2.clf()
>>> sfac = 20  # Magnification factor
>>> fig2.suptitle('Deformed Structure Snapshots (sec) - Moving
↪  Boundary, magnification = 20', fontsize=14)
>>>
>>> # Top row: times 0-4 (0.1 to 0.5 seconds)
>>> for i in range(5):
...     plt.subplot(2, 5, i+1)
...
...     # Offset coordinates for display
...     Ext = Ex + i * 3
...
...     # Draw undeformed structure
...     cfv.eldraw2(Ext, Ey, plotpar=[2, 3, 0])
...
...     # Extract displacements for this time step
...     Edb = cfc.extract_ed(Edof, a[:, i])
...
...     # Draw deformed structure
...     cfv.eldisp2(Ext, Ey, Edb, [1, 2, 2], sfac)
...
...     # Add time label
...     plt.text(i*3 + 0.5, 1.5, f'{times[i]:.1f}', fontsize=10,
↪  ha='center')
...     plt.axis('equal')
...     plt.axis('off')
>>>
>>> # Bottom row: times 5-9 (0.6 to 1.0 seconds)
>>> Eyt = Ey - 4
>>> for i in range(5, 10):
...     plt.subplot(2, 5, i+1)
...
...     # Offset coordinates for display
...     Ext = Ex + (i-5) * 3
...
```

```
...     # Draw undeformed structure
...     cfv.eldraw2(Ext, Eyt, plotpar=[2, 3, 0])
...
...     # Extract displacements for this time step
...     Edb = cfc.extract_ed(Edof, a[:, i])
...
...     # Draw deformed structure
...     cfv.eldisp2(Ext, Eyt, Edb, [1, 2, 2], sfac)
...
...     # Add time label
...     plt.text((i-5)*3 + 0.5, -2.5, f'{times[i]:.1f}', fontsize=10,
↪  ha='center')
...     plt.axis('equal')
...     plt.axis('off')
>>>
>>> plt.tight_layout()
>>> plt.show()
>>>
>>> print("Deformed shape snapshots with moving boundary created
↪  successfully")
```

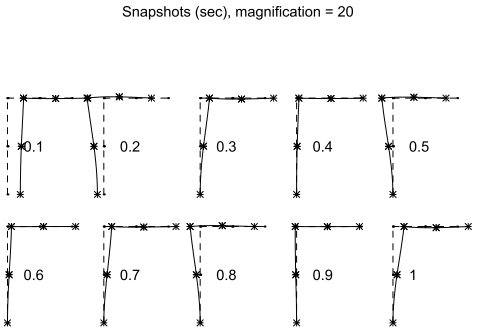


Fig. 3: Snapshots of the deformed geometry for every 0.1 sec.

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y