A multi-modular Finite Element Analysis package for Matlab

3rd October 2017

Andrey Melnik, Lili Zhang, Alexander Korsunsky, Antoine Jerusalem

University of Oxford

Contents

Listings

1 Introduction

This document presents a Finite Element Analysis package for Matlab (later referred to as "the package"). The package is developed with the purpose to support courses in finite element methods at an undergraduate level as an integral part or as a supplement. It contains m-files, which implement finite element algorithms, including procedures for mesh generation and manipulation, matrix and vector assembly, recovery and visualisation of results, and also auxiliary and service files (shape functions, Gauss points, handy representation of elastic properties, a small library of examples, verification procedures, etc.).

The package is designed in a modular manner, which provides flexibility and transparency, but requires the user to maintain the link between different procedures. This means that the solution of a problem consists in writing a *main file*, which contains definitions of problem-specific data and calls of routines that are responsible for all stages of the solving process: from discretisation to data extraction and visualisation of results. As a result, the user maintains full control over the process and can easily understand and track it at all stages.

The package is able to solve static and dynamic problems of linear elasticity, heat equation and similar PDEs. At the current stage of development, the package has limited capabilities to address nonlinear problems.

1.1 PDEs considered

The package solves linear equations of the following form:

$$\boldsymbol{\rho}\cdot\ddot{\mathbf{u}}+\boldsymbol{\mu}\cdot\dot{\mathbf{u}}-\nabla\cdot(\mathbf{c}:\nabla\mathbf{u}+\boldsymbol{\beta}\cdot\mathbf{u})-\mathbf{f}=0,\quad \text{in}\,\Omega \tag{1}$$

where the overhead dot denotes time derivative, $\mathbf{u}$ is the unknown vector field defined over the domain $\Omega\subset\mathbb{R}^{\#Dim}$, $\boldsymbol{\rho}$, $\boldsymbol{\mu}$ are second-order $\#Eq\times\#Eq$ tensors, $\mathbf{c}$ is a fourth-order $\#Eq\times\#Dim\times\#Eq\times\#Dim$ tensor, $\boldsymbol{\beta}$ is a third-order $\#Eq\times\#Dim\times\#Eq$ tensor and $\mathbf{f}\in\mathbb{R}^{\#Eq}$ is a known vector field. By convention, tensor contractions (e.g., $\mathbf{A}\cdot\mathbf{B}$, $\mathbf{A}:\mathbf{B}$) affect last indices of the first argument and first indices of the second argument, while gradient and divergence affect always last indices. Notation starting with hash symbol # refers to dimensionality of tensor and data structures specific to the formulation of the current problem, see Table 1.

We take the balance of linear momentum for a linearly elastic solid as a model problem and therefore we will refer to $\mathbf{u}$, $\dot{\mathbf{u}}$, $\ddot{\mathbf{u}}$ as displacement, velocity and acceleration vectors, respectively. As long as coefficients $\boldsymbol{\rho}$, $\boldsymbol{\mu}$, $\mathbf{c}$, $\boldsymbol{\beta}$, and $\mathbf{f}$ do not depend on $\mathbf{u}$, $\dot{\mathbf{u}}$, or $\ddot{\mathbf{u}}$, the problem remains linear, therefore time dependence and non-uniformity of the coefficients have little effect on the method of solution.

The boundary conditions can be of Dirichlet (on a subset $\partial\Omega_d$ of the boundary $\partial\Omega$) or Neumann (on a subset $\partial\Omega_n$ of the boundary $\partial\Omega$) type and prescribed independently for each component of $\mathbf{u}$,

$$\mathbf{u}=\bar{\mathbf{u}},\quad \text{on}\,\partial\Omega_d, \tag{2}$$

$$(\mathbf{c}:\nabla\mathbf{u}+\boldsymbol{\beta}\cdot\mathbf{u})\cdot\mathbf{n}=\bar{\mathbf{t}},\quad \text{on}\,\partial\Omega_n, \tag{3}$$

where $\bar{\mathbf{u}}$ and $\bar{\mathbf{t}}$ are the prescribed boundary displacement and surface traction vectors, $\mathbf{n}$ is the outward normal to the boundary $\partial\Omega_n$.

This problem can also be written in index notation in the weak form,

$$\begin{aligned}\int\limits_{\Omega}\eta_i\rho_{ij}\ddot{u}_j\mathrm{d}\Omega+\int\limits_{\Omega}\eta_i\mu_{ij}\dot{u}_j\mathrm{d}\Omega+\int\limits_{\Omega}\frac{\partial\eta_i}{\partial x_j}\left(\mathrm{c}_{ijkl}\frac{\partial u_k}{\partial x_l}+\beta_{ijk}u_k\right)\mathrm{d}\Omega&\\ -\int\limits_{\Omega}\eta_i\mathrm{f}_i\mathrm{d}V-\int\limits_{\partial\Omega_n}\eta_i\bar{\mathrm{t}}_i\mathrm{d}S&=0,\end{aligned} \tag{4}$$

where η_i are components of a test function from a suitable space. For finite element approximation, we use Bubnov-Galerkin method, in which test functions and displacement interpolants belong to the same space. In linear elasticity, the stress tensor is defined as $\boldsymbol{\sigma}=\mathbf{c}:\boldsymbol{\nabla}\mathbf{u}$ where $\mathbf{c}$ is the stiffness tensor. By extension to other constitutive models and with $\boldsymbol{\beta}=\mathbf{0}$, the definition of Neumann boundary conditions can directly be written as:

$$\boldsymbol{\sigma}\cdot\mathbf{n}=\bar{\mathbf{t}},\quad \text{on}\,\partial\Omega_n, \tag{5}$$

#	Description
$\#Eq$	Number of equations in the original system, i.e., number of degrees of freedom
$\#Dim$	Dimensionality of the domain $\Omega \subset \mathbb{R}^{\#Dim}$; current version only supports 2D problems, $\#Dim = 2$
$\#DimNat$	Dimensionality of the natural domain and natural coordinate system of an element e, $\Omega_{\text{nat}} \subset \mathbb{R}^{\#DimNat}$
$\#El$	Number of elements in the domain discretisation
$\#NpE$	Number of nodes per element
$\#EpN = \#EpN(n)$	Number of elements per node, varies from node to node
$\#Nodes$	Total number of nodes
$\#GDoF = \#Nodes \times \#Eq$	Number of global degrees of freedom
$\#FDoF$, $\#PDoF$	Numbers of free and prescribed global degrees of freedom, $\#FDoF + \#PDoF = \#GDoF$
$\#GP$	Number of Gauss points, can vary from element to element, but current version only support constant values
$\#BEl$	Number of boundary elements
$\#BNpE$	Number of nodes per boundary element
$\#BR$	Number of boundary regions
$\#AO$	Number of additional outputs of constitutive modulus

Table 1: Notation for integers that define dimensionality of tensors and data structures. In Matlab code we use similar identifier for these integers, e.g., `numEq`, `numDim`, etc.

1.2 Isoparametric FE discretisation and solution

In the following, a brief overview of the finite element method is provided. More details can be found in the literature [3, 1, 2, 4].

Assume that the domain Ω is discretised into disjunct element domains $\Omega_h^e \subset \mathbb{R}^{\#Dim}$, $m = 1, \ldots, \#El$. The element domain Ω^e is isomorphic to the element natural configuration $\Omega_{\text{nat}} \subset \mathbb{R}^{\#DimNat}$, and this isomorphism[1] can be expressed through the element's shape functions $N_1, \ldots, N_{\#NpE} : \Omega_{\text{nat}} \to \mathbb{R}$ and the element nodes of coordinates $\mathbf{x}^1, \ldots \mathbf{x}^{\#NpE} \in \Omega$,

$$\mathbf{x}(\mathbf{r}) = \sum_{a=1}^{\#NpE} N_a(\mathbf{r})\mathbf{x}^a \in \Omega^e, \qquad \mathbf{r} \in \Omega_{\text{nat}}. \tag{6}$$

The unknown displacement field $\mathbf{u}(\mathbf{x})$ is interpolated within each element by the same set of shape functions,

$$\mathbf{u}(\mathbf{x}) = \sum_{a=1}^{\#NpE} N_a(\mathbf{r})\mathbf{u}^a \in \mathbb{R}^{\#Eq}, \quad \mathbf{x} \in \Omega, \tag{7}$$

where $\mathbf{u}^1, \ldots, \mathbf{u}^{\#NpE}$ are the values of the unknown displacement field at the nodes of an element and are vectors of dimension $\#Eq$. This equation can then be rewritten:

$$\mathbf{u}(\mathbf{x}) \;=\; \underbrace{\begin{bmatrix} N_1 & \ldots & 0 & N_2 & \ldots & 0 & \ldots & N_{\#NpE} & \ldots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \ldots & \vdots & \ddots & \vdots \\ 0 & \ldots & N_1 & 0 & \ldots & N_2 & \ldots & 0 & \ldots & N_{\#NpE} \end{bmatrix}}_{\mathbf{H}} \cdot \underbrace{\begin{bmatrix} u_1^1 \\ \vdots \\ u_{\#Eq}^1 \\ u_1^2 \\ \vdots \\ u_{\#Eq}^{\#NpE} \end{bmatrix}}_{\hat{\mathbf{u}}} \tag{8}$$

where $\mathbf{H}$ is the local *displacement interpolation matrix.* By taking derivatives we obtain the interpolation for displacement gradients,

$$\begin{bmatrix} \frac{\partial \mathrm{u}_1}{\partial x_1} & \ldots & \frac{\partial \mathrm{u}_{\#Eq}}{\partial x_1} \\ \vdots & & \vdots \\ \frac{\partial \mathrm{u}_1}{\partial x_{\#Dim}} & \ldots & \frac{\partial \mathrm{u}_{\#Eq}}{\partial x_{\#Dim}} \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x_1} \sum_{a=1}^{\#NpE} N_a u_1^a & \ldots & \frac{\partial}{\partial x_1} \sum_{a=1}^{\#NpE} N_a u_{\#Eq}^a \\ \vdots & & \vdots \\ \frac{\partial}{\partial x_{\#Dim}} \sum_{a=1}^{\#NpE} N_a u_1^a & \ldots & \frac{\partial}{\partial x_{\#Dim}} \sum_{a=1}^{\#NpE} N_a u_{\#Eq}^a \end{bmatrix} \tag{9}$$

$$= \underbrace{\begin{bmatrix} \frac{\partial N}{\partial x_1} & \frac{\partial N_2}{\partial x_1} & \ldots & \frac{\partial N_{\#NpE}}{\partial x_1} \\ \vdots & \vdots & & \vdots \\ \frac{\partial N_1}{\partial x_{\#Dim}} & \frac{\partial N_2}{\partial x_{\#Dim}} & \ldots & \frac{\partial N_{\#NpE}}{\partial x_{\#Dim}} \end{bmatrix}}_{\textbf{spatDerivatives}'} \cdot \underbrace{\begin{bmatrix} u_1^1 & \ldots & u_{\#Eq}^1 \\ u_1^2 & \ldots & u_{\#Eq}^2 \\ \vdots & & \vdots \\ u_1^{\#NpE} & \ldots & u_{\#Eq}^{\#NpE} \end{bmatrix}}_{\textbf{u2\_loc}}, \tag{10}$$

[1]Note that although $\Omega_{\text{nat}}^{(m)}$ and $\Omega^{(m)} \subset \mathbb{R}^{\#Dim}$ are isomorphic, in general, $\#DimNat \leq \#Dim$, as it does not violatedim $\Omega^{(m)} \leq$ dim $\Omega \leq$ dim $\mathbb{R}^{\#Dim}$. For example, we represent the boundary as a union of elements, whose dimensionality is less than the dimensionality of the underlying space, e.g., we have $\#DimNat = 1$, $\#Dim = 2$.

$$\nabla\mathbf{u}(\mathbf{x}) = \begin{bmatrix} \partial u_1/\partial x_1 \\ \vdots \\ \partial u_1/\partial x_{\#Dim} \\ \partial u_2/\partial x_1 \\ \vdots \\ \partial u_{\#Eq}/\partial x_{\#Dim} \end{bmatrix} = \nabla\mathbf{H}\cdot\hat{\mathbf{u}} \tag{11}$$

$$= \underbrace{\begin{bmatrix} \frac{\partial N_1}{\partial x_1} & 0 & \dots & 0 & \dots & \frac{\partial N_{\#NpE}}{\partial x_1} & 0 & \dots & 0 \\ \frac{\partial N_1}{\partial x_2} & 0 & \vdots & \vdots & \dots & \frac{\partial N_{\#NpE}}{\partial x_2} & 0 & \vdots & \vdots \\ \vdots & 0 & \dots & 0 & \dots & \vdots & 0 & \dots & 0 \\ \frac{\partial N_1}{\partial x_{\#Dim}} & 0 & \dots & 0 & \dots & \frac{\partial N_{\#NpE}}{\partial x_{\#Dim}} & 0 & \dots & 0 \\ 0 & \frac{\partial N_1}{\partial x_1} & \dots & 0 & \dots & 0 & \frac{\partial N_{\#NpE}}{\partial x_1} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \vdots \\ 0 & \frac{\partial N_1}{\partial x_{\#Dim}} & 0 & 0 & \dots & 0 & \frac{\partial N_{\#NpE}}{\partial x_{\#Dim}} & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \frac{\partial N_1}{\partial x_1} & \dots & 0 & 0 & \dots & \frac{\partial N_{\#NpE}}{\partial x_1} \\ 0 & 0 & \dots & \vdots & \dots & 0 & 0 & \dots & \vdots \\ 0 & 0 & \dots & \frac{\partial N_1}{\partial x_{\#Dim}} & \dots & 0 & 0 & \dots & \frac{\partial N_{\#NpE}}{\partial x_{\#Dim}} \end{bmatrix}}_{\mathbf{gradH}} \cdot \underbrace{\begin{bmatrix} u_1^1 \\ \vdots \\ u_{\#Eq}^1 \\ u_1^2 \\ \vdots \\ u_{\#Eq}^{\#NpE} \end{bmatrix}}_{\hat{\mathbf{u}}}. \tag{12}$$

The analogous expressions in 2D and 3D for strains read, respectively,

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ 2\varepsilon_{12} \end{bmatrix} = \underbrace{\begin{bmatrix} \frac{\partial}{\partial x}N_1 & 0 & \dots & \frac{\partial}{\partial x}N_{\#NpE} & 0 \\ 0 & \frac{\partial}{\partial y}N_1 & \dots & 0 & \frac{\partial}{\partial y}N_{\#NpE} \\ \frac{\partial}{\partial y}N_1 & \frac{\partial}{\partial x}N_1 & \dots & \frac{\partial}{\partial y}N_{\#NpE} & \frac{\partial}{\partial x}N_{\#NpE} \end{bmatrix}}_{\mathbf{B}} \cdot \underbrace{\begin{bmatrix} u_1^1 \\ u_2^1 \\ \vdots \\ u_1^{\#NpE} \\ u_2^{\#NpE} \end{bmatrix}}_{\hat{\mathbf{u}}}, \tag{13}$$

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ 2\varepsilon_{23} \\ 2\varepsilon_{13} \\ 2\varepsilon_{12} \end{bmatrix} = \underbrace{\begin{bmatrix} \frac{\partial}{\partial x}N_1 & 0 & & \dots & \frac{\partial}{\partial x}N_{\#NpE} & 0 & \\ 0 & \frac{\partial}{\partial y}N_1 & & \dots & 0 & \frac{\partial}{\partial y}N_{\#NpE} & \\ & & \frac{\partial}{\partial z}N_1 & \dots & & & \frac{\partial}{\partial z}N_{\#NpE} \\ & \frac{\partial}{\partial z}N_1 & \frac{\partial}{\partial y}N_1 & \dots & & \frac{\partial}{\partial z}N_{\#NpE} & \frac{\partial}{\partial y}N_{\#NpE} \\ \frac{\partial}{\partial z}N_1 & & \frac{\partial}{\partial x}N_1 & \dots & \frac{\partial}{\partial z}N_{\#NpE} & & \frac{\partial}{\partial x}N_{\#NpE} \\ \frac{\partial}{\partial y}N_1 & \frac{\partial}{\partial x}N_1 & & \dots & \frac{\partial}{\partial y}N_{\#NpE} & \frac{\partial}{\partial x}N_{\#NpE} & \end{bmatrix}}_{\mathbf{B}} \cdot \underbrace{\begin{bmatrix} u_1^1 \\ u_3^2 \\ u_3^1 \\ \vdots \\ u_1^{\#NpE} \\ u_2^{\#NpE} \\ u_3^{\#NpE} \end{bmatrix}}_{\hat{\mathbf{u}}}. \tag{14}$$

Given that the test functions ($\hat{\boldsymbol{\eta}}$ in its "vectorised" form) are interpolated in the same manner, the discretised version of the weak form (4) reads

$$\begin{aligned} &\sum_e \int_{\Omega_h^e} \mathrm{H}_{ir}\hat{\eta}_r\rho_{ij}\mathrm{H}_{js}\ddot{\hat{\mathrm{u}}}_s \mathrm{d}\Omega + \sum_e \int_{\Omega_h^e} \mathrm{H}_{ir}\hat{\eta}_r\mu_{ij}\mathrm{H}_{js}\dot{\hat{\mathrm{u}}}_s \mathrm{d}\Omega \\ &+ \sum_e \int_{\Omega_h^e} \nabla\mathrm{H}_{Ir}\hat{\eta}_r \left(\hat{\mathrm{c}}_{IJ}\nabla\mathrm{H}_{Js}\hat{\mathrm{u}}_s + \beta_{Ik}\mathrm{H}_{ks}\hat{\mathrm{u}}_s\right) \mathrm{d}\Omega \\ &- \sum_e \int_{\Omega_h^e} \mathrm{H}_{ir}\hat{\eta}_r\mathrm{f}_i \mathrm{d}\Omega - \sum_e \int_{\partial\Omega_{h,n}^e} \mathrm{H}_{ir}\hat{\eta}_r\bar{\mathrm{t}}_i \mathrm{d}S = 0, \end{aligned} \tag{15}$$

where $\hat{c}_{IJ}$ is a $\#Eq \cdot \#Dim \times \#Eq \cdot \#Dim$ tensor, a "vectorised" form of c_{ijkl}, which is a fourth-order $\#Eq \times \#Dim \times \#Eq \times \#Dim$ tensor, similarly to β_{Ik}. $\partial\Omega^e_{h,n}$ is the Neumann boundary of element e (can be empty). The rearranging of terms yields

$$\sum_e \hat{\eta}_r \underbrace{\int_{\Omega^e_h} \mathrm{H}_{ir}\rho_{ij}\mathrm{H}_{js}\mathrm{d}\Omega}_{\mathrm{M}^e_{rs}}\, \ddot{\hat{\mathrm{u}}}_s + \sum_e \hat{\eta}_r \underbrace{\int_{\Omega^e_h} \mathrm{H}_{ir}\mu_{ij}\mathrm{H}_{js}\mathrm{d}\Omega}_{\mathrm{C}^e_{rs}}\, \dot{\hat{\mathrm{u}}}_s \tag{16}$$

$$+\sum_e \hat{\eta}_r \underbrace{\int_{\Omega^e_h} \nabla\mathrm{H}_{Ir}\left(\hat{c}_{IJ}\nabla\mathrm{H}_{Js} + \beta_{Ik}\mathrm{H}_{ks}\right)\mathrm{d}\Omega}_{\mathrm{K}^e_{rs}}\, \hat{\mathrm{u}}_s \tag{17}$$

$$-\sum_e \hat{\eta}_r \underbrace{\int_{\Omega^e_h} \mathrm{H}_{ir}\mathrm{f}_i\mathrm{d}\Omega}_{\mathrm{F^B}^e_r} - \sum_e \hat{\eta}_r \underbrace{\int_{\partial\Omega^e_{h,n}} \mathrm{H}_{ir}\bar{\mathrm{t}}_i\mathrm{d}S}_{\mathrm{F^S}^e_r} \quad = \quad 0, \tag{18}$$

where $\mathbf{M}^e$, $\mathbf{C}^e$, $\mathbf{K}^{(m)}$ and $\mathbf{F}^{\mathrm{B}^e}$, $\mathbf{F}^{\mathrm{S}^e}$ are local mass, damping, stiffness matrices and body force and surface loading vectors, respectively. Due to the arbitrarity of test functions, the discretised weak form after the assembly becomes

$$\mathbf{M} \cdot \ddot{\hat{\mathbf{u}}} + \mathbf{C}\cdot\dot{\hat{\mathbf{u}}} + \mathbf{K}\cdot\hat{\mathbf{u}} = \underbrace{\mathbf{F}^{\mathrm{B}} + \mathbf{F}^{\mathrm{S}}}_{\mathbf{F}}, \tag{19}$$

where $\ddot{\hat{\mathbf{u}}}$, $\dot{\hat{\mathbf{u}}}$ and $\hat{\mathbf{u}}$ are the global vectors of nodal acceleration, velocity, and displacement, respectively. The Neumann boundary condition was taken into account when deriving (15), whereas the Dirichlet boundary condition is incorporated directly by setting appropriate degrees of freedom at the appropriate nodes to the defined values, see Equation (2). By rebalancing the contributions of the known values of $\hat{\mathbf{u}}$ (provided through the "vectorised" form of $\bar{\mathbf{u}}$) and transfering them on the right hand side as an additional external force vector, the system reduced to the unknown values of $\hat{\mathbf{u}}$ can be solved.

The static case simplifies to:

$$\mathbf{K}\cdot\hat{\mathbf{u}} = \underbrace{\mathbf{F}^{\mathrm{B}} + \mathbf{F}^{\mathrm{S}}}_{\mathbf{F}}. \tag{20}$$

The dynamic case requires the numerical integration of (19). We consider here the Newmark algorithm to solve the system (here at time $n+1$):

$$\mathbf{u}_{n+1} = \underbrace{\mathbf{u}_n + \Delta t\mathbf{v}_n + \frac{1}{2}\Delta t^2(1-2\beta)\mathbf{a}_n}_{\mathbf{u}^0_{\mathrm{n+1}}} + \Delta t^2\beta\mathbf{a}_{n+1}, \tag{21}$$

$$\mathbf{v}_{n+1} = \underbrace{\mathbf{v}_n + \Delta t(1-\gamma)\mathbf{a}_n}_{\mathbf{v}^0_{\mathrm{n+1}}} + \Delta t\gamma\mathbf{a}_{n+1}. \tag{22}$$

Here β and γ are the Newmark parameters, $\mathbf{u}^0_{\mathrm{n+1}}$ and $\mathbf{v}^0_{\mathrm{n+1}}$ are the Newmark displacement and velocity predictors. Here and subsequently, the $\hat{}$ notation is dropped for simplicity. These expressions are substituted into the semi-discrete equation at time step $n+1$,

$$\mathbf{M} \cdot \mathbf{a}_{n+1} + \mathbf{C}\cdot\mathbf{v}_{n+1} + \mathbf{K}\cdot\mathbf{u}_{n+1} = \mathbf{F}_{n+1}, \tag{23}$$

to obtain

$$\left(\mathbf{M} + \Delta t^2\beta\mathbf{K} + \Delta t\gamma\mathbf{C}\right) \cdot \mathbf{a}_{n+1} + \mathbf{C}\cdot\mathbf{v}^0_{\mathrm{n+1}} + \mathbf{K} \cdot \mathbf{u}^0_{\mathrm{n+1}} = \mathbf{F}_{i+1}, \tag{24}$$

which is an equation for $\mathbf{a}_{n+1}$, assuming $\mathbf{u}_n$, $\mathbf{v}_n$, $\mathbf{a}_n$ are known. After the new accelerations $\mathbf{a}_{n+1}$ are found, displacements and velocities $\mathbf{u}_{n+1}$, $\mathbf{v}_{n+1}$ are updated via (21), (22). For a nonlinear system, Newton-Raphson iterations are used.

The values of β and γ are directly related to numerical stability, damping and other properties of the integration scheme. Popular choices of the Newmark parameters include $\beta = \frac{1}{4}$, $\gamma = \frac{1}{2}$ for unconditionally stable implicit scheme and $\beta = 0$, $\gamma = \frac{1}{2}$ for conditionally stable explicit scheme for diagonal $\mathbf{M}$ (potentially lumped) and in the absence of damping. In the latter case, the time step must be chosen to satisfy $\Delta t \leq L c_p^{-1}$, where L is the minimum element size and c_p is the dilatational (pressure) wave phase speed. One consistent diagonal lumped mass matrix $\bar{\mathbf{M}}$ can be obtained by summing up the columns:$\bar{\mathrm{M}}_{ii} = \sum_j \mathrm{M}_{ij}$. For further details and additional information the reader is referred to the literature [2] for Newmark's method in the context of nonlinear problems.

2 Workflow

A typical workflow consists in writing a main file, which defines problem-specific data and calls procedures, which facilitate the following steps of problem solving:

- mesh generation;
- matrix and vector assembly;
- solution of a linear system of equations;
- stress and strain (gradient) recovery;
- visualisation.

In static linear problems, this sequence of steps is normally followed literally. In nonlinear problems, an iterative method is used: a linearised system of equations is constructed and solved repeatedly until a converged solution is obtained (Newton-Raphson). Repetition of some of the above steps is also typical for linear dynamic problems. In this Section, we describe the suggested course of action at each step separately and give a minimal example to illustrate the entire workflow.

2.1 Mesh generation

Element type	Description	Natural (parent) domain	Shape functions
`'1dQ1'`	1D linear element	$-1 \leq r \leq 1$	$N_1 = \frac{1}{2}(r-1)$, $N_2 = \frac{1}{2}(r+1)$
`'1dQ2'`	1D quadratic element	$-1 \leq r \leq 1$	$N_1 = \frac{1}{2}r(r-1)$, $N_2 = 1 - r^2$, $N_3 = \frac{1}{2}r(r+1)$
`'2dP1'`	2D linear triangular element	$0 \leq r,\, s,\, r+s \leq 1$	$N_1 = r$, $N_2 = s$, $N_3 = 1 - r - s$
`'2dP2'`	2D quadratic triangular element	$0 \leq r,\, s,\, r+s \leq 1$	$N_1 = r(2r-1)$, $N_2 = s(2s-1)$ $N_3 = (1-r-s)(1-2r-2s)$, $N_4 = 4rs$ $N_5 = 4s(1-r-s)$, $N_6 = 4r(1-r-s)$
`'2dQ1'`	2D bilinear quadrilateral element	$-1 \leq r,\, s \leq 1$	$N_1 = \frac{1}{4}(r-1)(s-1)$, $N_2 = \frac{1}{4}(r+1)(s-1)$ $N_3 = \frac{1}{4}(r+1)(s+1)$, $N_4 = \frac{1}{4}(r-1)(s+1)$
`'2dQ2r'`	2D quadratic quadrilateral (serendipity) element	$-1 \leq r,\, s \leq 1$	$N_1 = \frac{1}{4}(r-1)(s-1)(1+r+s)$ $N_2 = \frac{1}{4}rs(r+1)(s-1)(1-r+s)$ $N_3 = \frac{1}{4}rs(r+1)(s+1)(1-r-s)$ $N_4 = \frac{1}{4}rs(r-1)(s+1)(1+r-s)$ $N_5 = -\frac{1}{2}s(r^2-1)(s-1)$, $N_6 = -\frac{1}{2}r(r-1)(s^2-1)$ $N_7 = -\frac{1}{2}s(r^2-1)(s+1)$, $N_8 = -\frac{1}{2}r(r-1)(s^2-1)$
`'2dQ2'`	2D quadratic quadrilateral element	$-1 \leq r,\, s \leq 1$	$N_1 = \frac{1}{4}rs(r-1)(s-1)$, $N_2 = \frac{1}{4}rs(r+1)(s-1)$ $N_3 = \frac{1}{4}rs(r+1)(s+1)$, $N_4 = \frac{1}{4}rs(r-1)(s+1)$ $N_5 = -\frac{1}{2}s(r^2-1)(s-1)$, $N_6 = -\frac{1}{2}r(r-1)(s^2-1)$ $N_7 = -\frac{1}{2}s(r^2-1)(s+1)$, $N_8 = -\frac{1}{2}r(r-1)(s^2-1)$ $N_9 = (r^2-1)(s^2-1)$
`'3dQ1'`	3D linear hexahedral element ('brick')	$-1 \leq r,\, s,\, t \leq 1$	$N_1 = \frac{1}{8}(r-1)(s-1)(t-1)$, $N_2 = \frac{1}{8}(r+1)(s-1)(t-1)$ $N_3 = \frac{1}{8}(r+1)(s+1)(t-1)$, $N_4 = \frac{1}{8}(r-1)(s+1)(t-1)$ $N_5 = \frac{1}{8}(r-1)(s-1)(t+1)$, $N_6 = \frac{1}{8}(r+1)(s-1)(t+1)$ $N_7 = \frac{1}{8}(r+1)(s+1)(t+1)$, $N_8 = \frac{1}{8}(r-1)(s+1)(t+1)$
`'3dP1'`	3D linear tetrahedral element	$0 \leq r,\, s,\, t,\, r+s+t \leq 1$	$N_1 = r$, $N_2 = s$, $N_3 = t$, $N_4 = 1 - r - s - t$

Table 2: Supported element types and corresponding sets of shape functions.

Mesh generation is the initial step of the problem discretisation, which has to be accomplished prior to matrix and vector assembly. The output of this step is the mesh data, which will be passed to assembly, recovery and visualisation procedures. The resulting mesh should consist of appropriate elements, approximate the exact domain, be of sufficient fineness and possess other properties that ensure an accurate numerical solution. There are two ways of mesh generation:

- Using the package's own procedure: uniform 2D and 3D meshes over rectangular domains (a rectangle for 2D and a rectangular parallelepiped for 3D) can be generated used the `meshRect2d()` and `meshRect3d()` functions. Triangular, quadrilateral, tetrahedral and hexahedral elements are supported. For the list of supported elements see Table 2.

- Using Matlab's PDE Toolbox: the function `meshinit()`, which is included in the PDE Toolbox, supports meshing for 2D domains that can be described as union, intersection and subtraction of polygons and ellipses. Function `edges2sublists()` allows to convert the data structure used in PDE Toolbox to the format used by the package. The current version of the PDE Toolbox does not support 6-node quadratic triangular element. However, a mesh consisting of 6-node elements can be obtained from a mesh consisting of 3-node linear elements using the `meshP1toP2() function.`

Note that the current version of the package does not include any own meshing algorithms for arbitrary domains or methods of conversion from the PDE Toolbox 3D mesh format.

The basic convention for mesh representation is that elements and nodes in a given mesh are ordered in some way. That is, each node and each element are identified with a natural number, a nodal ID or an element ID, within the range $1 \dots \#Nodes$ or $1 \dots \#El$, respectively. The mesh is defined by the following data structures:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates. Each row `nodeCoords(i,:)` contains the coordinates of the ith node.

- `IEN` - $\#El$-by-$\#NpE$ matrix of the elements' connectivity. Each row `IEN(i,:)` contains the IDs of the nodes of the ith element. The order of nodal IDs is specific to the type of elements and a permutation would in general lead to an error.

- `elementType` - a single string defining the type of elements in the mesh. The list of supported elements can be found in Appendix.

- `BIEN` - a $\#BR$-by-1 cell array. Each cell `BIEN{k}` corresponds to boundary region k and contains a $\#BEl(k)$-by-$\#BNpE$ array of element-node incidence. Each row `BIEN{k}(i,:)` contains IDs of nodes incident to $(\#Dim-1)$-dimensional element i of boundary region k. `BIEN` effectively represents the boundary of the mesh as a submesh consisting of elements of dimension $\#Dim-1$. As in `IEN`, the order of nodal IDs in each row is specific to the type of elements and a permutation would in general lead to an error.

- `bndElementType` - a single string defining the type of elements in the boundary submesh.

There are intermediate and auxiliary data structures involved in the process.

- `boundaryElementIDs` - $\#BR$-by-1 cell array. Cell `boundaryElementIDs{p}` is a l_p-by-1 matrix listing IDs of elements at boundary p.

- `boundaryNodeLocalIDs` - $\#BR$-by-1 cell array. Cell `boundaryNodeLocalIDs{p}` contains a l_p-by-q matrix listing local IDs (within an element) of corresponding nodes in a specific order. In 2D meshes `boundaryNode~ LocalIDs(m,1)` and `boundaryNodeLocalIDs(m,end)` define the endpoints of the (1D) boundary edge of the element number `boundaryElementIDs(m)`. In 3D meshes `boundaryNodeLocalIDs(m,:)` lists nodes clockwise when viewed from inside the element.

- `INE` - $\#Nodes$-by-$\#EpN$ matrix of node-element incidence. Row `INE(i,:)` corresponds to ith node and contains IDs of elements that share it. Since the number of incident elements varies over nodes, $\#EpN$ is in fact the maximum nodal valency (degree) and the unused entries in each row are filled with zeros.

- `[p, e, t]` - 2D mesh representation used in Matlab PDE Toolbox. `p` is analogous to `nodeCoords`, `t` is analogous to `IEN`, and `e` describes the boundary of the mesh. See Matlab reference for details.

- `[gd, sf, ns]` - constructive solid geometry description used in Matlab PDE Toolbox. These data is used to describe a 2D region as unions and intersections of rectangles and ellipsoids. See Matlab reference for details.

- `dl` - decomposed geometry description used in Matlab PDE Toolbox, which serves as an intermediate representation of constructive solid geometry for mesh generation.

The two ways of mesh generation are summarised in Fig. 1.a.

2.2 Matrix and vector assembly

Assembly procedures evaluate integrals using finite element approximation and return global vectors (e.g., the body force vector, $\mathbf{F}^{\mathrm{B}}$) or matrices (e.g., the stiffness matrix, $\mathbf{K}$). All integrals are computed iteratively for each element and the element contributions are assembled into global arrays. Gauss point coordinates and weights are obtained by calling `gaussPoints()` for a provided element type and the number of Gauss points. The displacements and gradients at Gauss points are interpolated using shape functions as in (8)-(12) by calling `shapeFunctions<elementType>`, where `elementType` is the type of element used. See Table 4 in Section 3 for the brief summary of matrix and vector assembly procedures.

The input of all assembly procedures consists of the mesh data (`nodeCoords, IEN, elementType, numGP`), the quantity being integrated (`CMatrix, BetaMatrix, MMatrix, bodyForce, s`) and for some procedures, optional current nodal displacements and state variables at Gauss points (`u, s`). The integrated quantity can be constant or given as a function, e.g., `CMatrix(x_,u_,gradu_,s_)`, where

- `x_` is 1-by-$\#Dim$ position vector,
- `u_` is a 1-by-$\#Eq$ displacement vector,
- `gradu_` is a $\#Dim$-by-$\#Eq$ matrix, displacement gradient,
- `s_` is a 1-by-k vector of state variables.

In this case, the position, displacement and its gradient are interpolated from the values provided at nodes (i.e., `nodeCoords` and `u`), while the values of state variables are taken directly from the input `s`, which contains this datum at all Gauss points of the mesh.

An exception is `formInternalForce0()`, for which the integrated quantity is provided not as a constant or function handle, but as an array `s` of values at Gauss points.

When assembling the stiffness matrix, it may become convenient to collect and store extra data at Gauss points throughout the domain, e.g., the updated values of state variables (such necessity may arise when algorithmic moduli are used). In order to do that, an extra output argument must be specified at the procedure call. The assembly procedure in turn requests an additional output datum from the `CMatrix()` function, which must be given as a 1-by-$\#AO$ cell array containing matrices of arbitrary size. These data are collected at each Gauss point and combined into `argout`, a 1-by-$\#AO$ cell array of additional output.

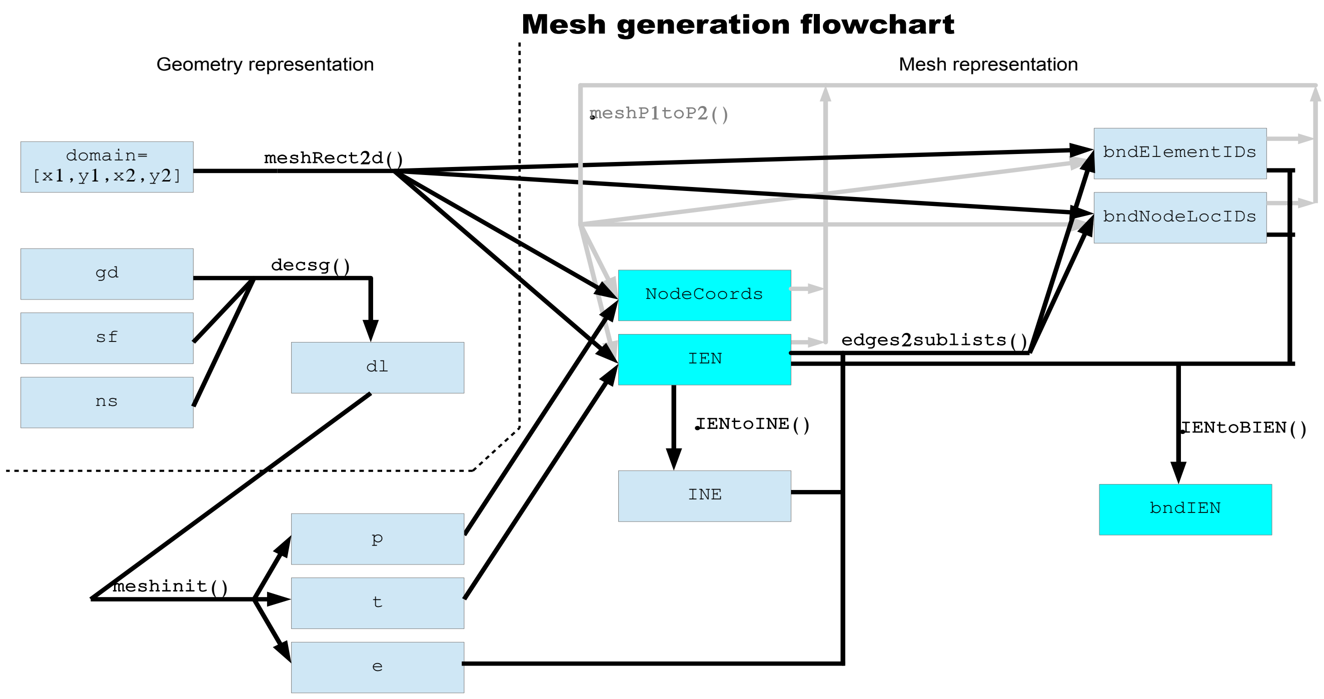
The conventions used for vectorised tensors match the following,

$$\begin{pmatrix}\sigma_{11}\\ \sigma_{21}\\ \sigma_{31}\\ \sigma_{12}\\ \sigma_{22}\\ \sigma_{32}\\ \sigma_{13}\\ \sigma_{23}\\ \sigma_{33}\end{pmatrix} = \mathbf{c}\cdot\begin{pmatrix}u_{1,1}\\ u_{1,2}\\ u_{1,3}\\ u_{2,1}\\ u_{2,2}\\ u_{2,3}\\ u_{3,1}\\ u_{3,2}\\ u_{3,3}\end{pmatrix}, \qquad \begin{pmatrix}\sigma_{11}\\ \sigma_{21}\\ \sigma_{12}\\ \sigma_{22}\end{pmatrix} = \mathbf{c}\cdot\begin{pmatrix}u_{1,1}\\ u_{1,2}\\ u_{2,1}\\ u_{2,2}\end{pmatrix}, \tag{25}$$

that is, the stress tensor $\boldsymbol{\sigma}$ and the displacement gradient $\nabla\mathbf{u}$ are vectorised in the "column first" order. Note that in problems of elasticity, the minor symmetries of modulus $\mathbf{c}$ allow to replace the displacement gradient $\nabla\mathbf{u}$ by the strain tensor $\boldsymbol{\varepsilon}$ where $\varepsilon_{ij} = \frac{1}{2}\left(u_{i,j} + u_{j,i}\right)$.

Function `formStiffnessMatrixEng()` uses Voigt notation for stiffness matrix assembly, that is, the stress and strain tensors are vectorised as follows,

a.



b.

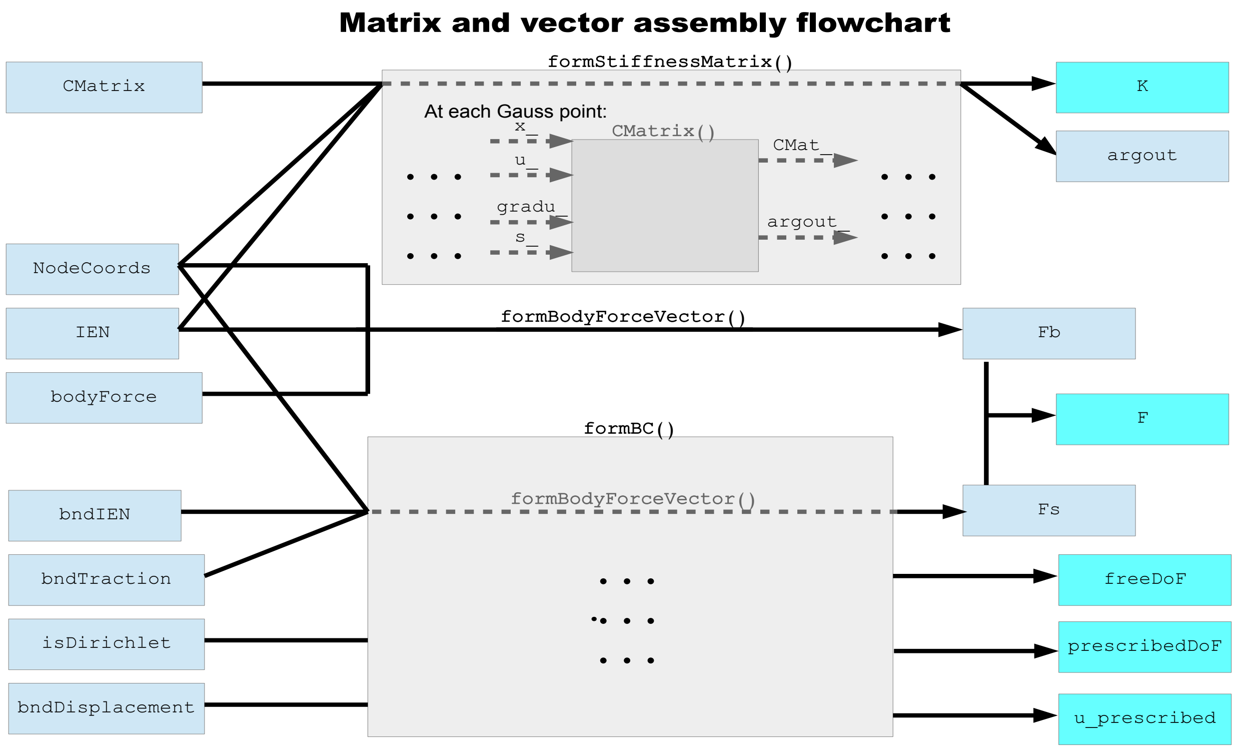


Figure 1: **a**. Mesh generation flowchart. **b.** Matrix and vector assembly flowchart.

$$\begin{pmatrix} \sigma_1 \\ \sigma_2 \\ \tau_{12} \end{pmatrix} = \mathbf{c} \cdot \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \gamma_{12} \end{pmatrix}, \qquad \begin{pmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_2 \\ \tau_{23} \\ \tau_{13} \\ \tau_{12} \end{pmatrix} = \mathbf{c}. \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \gamma_{23} \\ \gamma_{13} \\ \gamma_{12} \end{pmatrix}, \tag{26}$$

where **c** is now a 3-by-3 (2D) or 6-by-6 (3D) matrix,$\gamma_{ij} = u_{i,j} + u_{j,i}$ is the engineering shear strain.

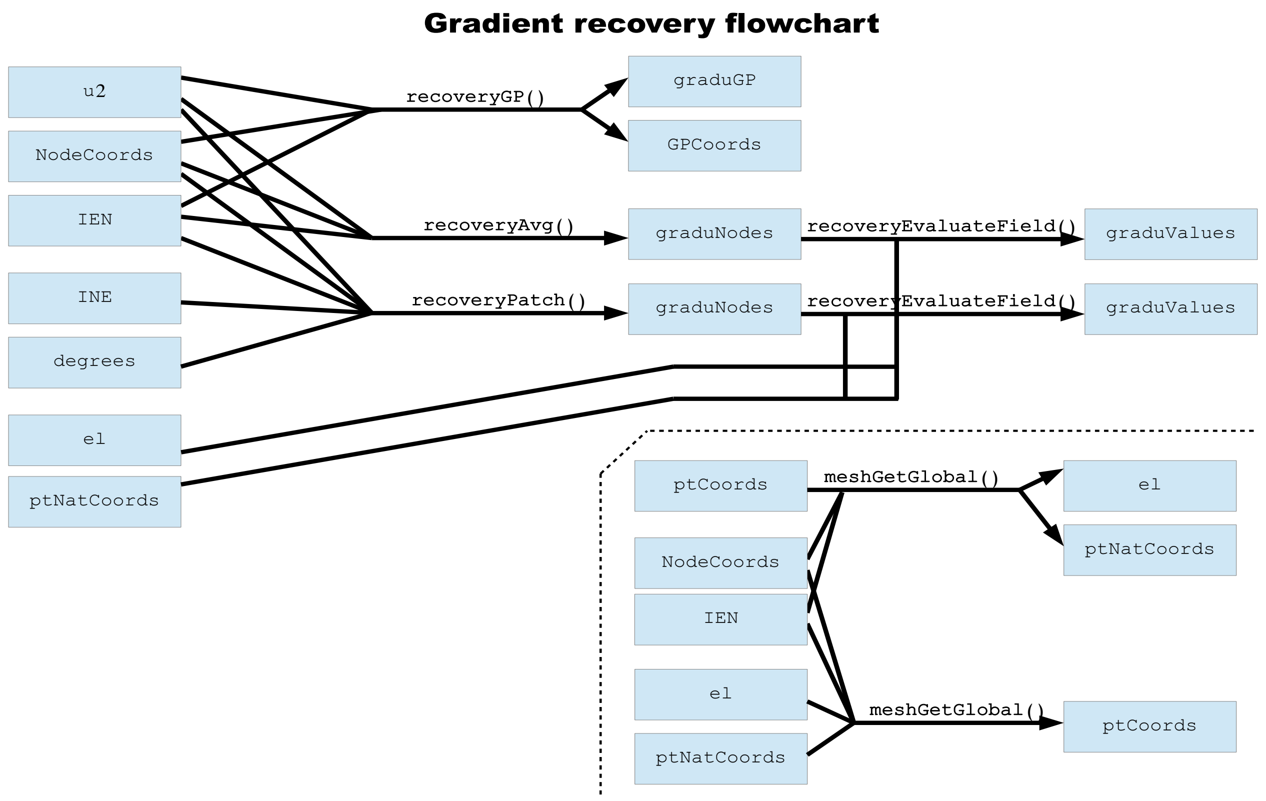


Figure 2: Recovery of gradients.

2.3 Solution

The finite element approximation of a linear static problem leads straightforwardly to a system of linear algebraic equations. No special procedures for solving the resulting linear equations are provided in the current version, as Matlab built-in functionality is deemed to be sufficient. Rows and columns of matrices can be conveniently manipulated using Matlab language, as demonstrated in the minimal example in the end of this section.

Linear and nonlinear dynamic problems with non-singular mass matrix are solved using the Newmark algorithm, see in Section 1.2. This method is implemented through the functions `solveNewmarkLa()` and `solveNewmarkNLa()`, whose usage is described in Section 1.2. By design, the input of these functions contains only semi-discretised data and no mesh or element information. If, for instance, the tangent stiffness matrix is not constant, then it must be passed as a function handle.

2.4 Recovery

After the problem is solved for the unknown field (e.g., displacements) one may be interested in computing the flux variables (e.g., displacement gradients, strains or stresses). Estimation of these dependent quantities from the solution and other a posteriori treatment is referred to as the *recovery*. We provide several procedure for the recovery of gradients, strains and stresses at nodes and Gauss points. The most sophisticated procedure included into the package is the *Superconvergent Patch Recovery*, which results in more accurate stress and strain predictions than standard methods (therefore, superconvergent).

The input of all recovery procedures consists the solution (e.g., displacements) and the mesh data. The following types of recovery methods are included into the package.

1. Recovery of gradients at Gauss points via finite element interpolation, as in equations (11), (12). This is implemented in `recoveryGP()` and `recoveryGPEng()`.
2. Recovery of gradients at elements' nodes, which done by averaging the values obtained within the elements incident to a given node. Functions `recoveryAvg()` and `recoveryAvgEng()` implement this.

3. Recovery of gradients at elements' nodes using the Superconvergent Patch Recovery technique, which consists in least-squares fitting of nodal stress values within small patches surrounding every suitable node. This method is implemented in `recoveryPatch()` and `recoveryPatchGP()`.

Finite element interpolation principle (8), (11)-(12) allows to evaluate the primary field and its gradients (or strains) at an arbitrary point defined in the natural (parent) element domain. Use `recoveryEvaluateField()`, `recoveryEvaluateGradients()` and `recoveryEvaluateStrains()` for that. In order to map points from the natural (parent) domain to the global domain and back use `meshGetGlobal()` and `meshGetNatural()`.

2.5 Visualisation

Visualisation consists in graphical depiction of original or deformed mesh (including nodes and edges) and representation of a quantity of interest by different colours and shades. Matlab functions `scatter3()` and `patch()` are well-suited for this purpose. The package includes the following visualisation routines, which effectively rearrange the input data and pass it to `scatter3()` and `patch()` Matlab functions.

- `drawElements()` - draws the mesh coloured according to values specified for every element, such as stresses or strains recovered at centroids using `recoveryGP()`.
- `drawElementsInterp()` - draws the mesh colored according to values specified at node. Can be used in conjunction with `recoveryAvg()` and `recoveryPatch()`. Note that neither geometry nor the colour data of higher order elements represent true values. This is related to the fact that `patch()` uses triangulation and linear interpolation.
- `drawNodes()` - puts a marker at each node. A useful supplement if one needs to mark all or specific nodes.

The input of the visualisation routines includes mesh data. Deformed meshes can be drawn by passing the nodal coordinates shifted by the deformation, which is amplified by a factor if necessary.

2.6 Minimal example. Plane stress equilibrium problem

We illustrate the suggested workflow by solving the equilibrium plane stress problem for a linearly elastic rectangular plate,

$$\begin{aligned}
-\nabla\cdot(\mathbf{c}:\nabla\mathbf{u})-\mathbf{f} &= 0, && \text{in } (x,y)\in[0,2]\times[0,2], && (27)\\
\mathbf{u} &= 0, && \text{on } x=0,\ y\in[0,2], && (28)\\
\mathbf{n}\cdot\nabla\cdot(\mathbf{c}:\nabla\mathbf{u}) &= 0, && \text{on } y=0,2,\ x\in[0,2], && (29)\\
\mathbf{n}\cdot\nabla\cdot(\mathbf{c}:\nabla\mathbf{u}) &= \bar{\mathbf{t}}, && \text{on } x=2,\ y\in[0,2], && (30)
\end{aligned}$$

where $\mathbf{u}=(u_1,u_2)^{\mathrm{T}}$, $\mathbf{c}$ is the fourth-order elasticity tensor, $\mathbf{f}=(0,-\rho g h)$ is the gravity force. The boundary conditions are as follows: the right edge is subject to constant uniform loading $\bar{\mathbf{t}}=(\bar{t}_1,\bar{t}_2)$, the top and the bottom edges are traction-free, and the left edge is fixed at zero displacement. The FE solution of this problem is included in the package as `example0.m`. Note that both traction forces and body forces are normalised in for simplicity in the example. For other examples the reader is referred to Section 4.

The first step is the creation of mesh for the rectangular domain. The user defines the element types (see Table 2) for solid and boundary elements, these types must be in agreement. Then the primary mesh data is generated by `meshRect2d()` and the boundary incidence array is formed by `IENtoBIEN()`. Here `BIEN` is 4-by-1 cell array, the cells contain element-node incidence matrices corresponding to the bottom, right, top, and left boundaries of the domain. Here we followed the shorter (top) route depicted in Figure 1.a.

Listing 1: Example 0 - Mesh generation

```
%% Mesh generation
elementType='2dQ1'; %define element type
elementType1d='1dQ1'; %define boundary element type — must be consistent
[nodeCoords, IEN, boundaryElementIDs, boundaryNodeLocalID]=...
    meshRect2d([0 0 2 2],elementType,[10 10]); %creating mesh 10—by—10 el.
%create boundary incidence arrays
BIEN=IENtoBIEN(IEN, boundaryElementIDs, boundaryNodeLocalID);
```

The second step is matrix and vector assembly. It start with defining the functions for boundary conditions and a matrix `isDirichlet`, which tells which boundary regions have essential (Dirichlet) boundary conditions imposed. This matrix contains 4 rows (line 11) because `BIEN` contains four cells. The boundary tractions and displacements are defined as anonymous functions (lines 4-6), but could also be defined in separate files. Next we form cell arrays for Dirichlet and Neumann boundary conditions, which must match the dimensions of `BIEN`, 4-by-1. Function `formBC()` in lines 16-18 makes all the necessary calls to integrate and assemble the global surface force vector (the number of Gauss points for integration is defined at line 13) and evaluate prescribed values of displacement at the boundary. It also returns the lists of free and prescribed degrees of freedom. The body force vector is assembled by `formBodyForceVector`. The final part consists in stiffness matrix assembly, facilitated by `formStiffnessMatrixEng`, which uses the elastic (tensor) modulus constructed in lines 29-30. The route followed here is also shown in Figure 2.

Listing 2: Example 0 - Matrix and vector assembly

```
1  %% Matrix and vector assembly
2  % Boundary conditions
3  %define function handles for BC
4  bndDisplacement=@(x)(zeros(size(x))'); %zero displacement
5  bndTraction0=@(x)(zeros(size(x))'); %zero traction
6  bndTraction1=@(x)(repmat([1; 1],size(x,1)));  %non-zero traction
7  %make cell arrays of function handles - boundary condition cell arrays must
8  %match the structure of BIEN
9  bndTractions={bndTraction0,bndTraction1,bndTraction0,bndTraction0};
10 bndDisplacements=repmat({bndDisplacement},4,1);
11 isDirichlet=[0; 0; 0; 1]; %define which boundary regions have Dirichlet BC
12 
13 numGP1d=1; %number of Gauss Points for 1d boundary elements
14 % assemble boundary load vector for Neumann BC
15 % and evaluate displacements for Dirichlet BC
16 [u_prescribed, Fs, prescribedDoF, freeDoF]=...
17     formBC(nodeCoords,BIEN,elementType1d,numGP1d,...
18     bndTractions,bndDisplacements,isDirichlet);
19 
20 % Body force
21 numGP=1; %number of Gauss Points for 2d elements
22 %define function handle for body force
23 bodyForce=@(x)(repmat([0; -1],size(x,1)));
24 % assemble body force vector
25 Fb = formBodyForceVector(nodeCoords, IEN, elementType, numGP, bodyForce);
26 
27 % Stiffness matrix
28 %define the matrix of elasticities
29 CMatrix=elasticProperties('youngsModulus',193e6,'poissonsRatio',0.253,...
30     'CPlaneStressEng'); %elasticity tensor (Voigt notation)
31 %assemble stiffness matrix
32 K = formStiffnessMatrixEng(nodeCoords, IEN, elementType, numGP, CMatrix);
```

The discretisation of the problem is now completed. We can now proceed to solving equations (27) in order to find unknown displacements at the free degrees of freedom. The effective force vector (line 3) and the free part of the stiffness matrix (line 5) make a linear system (20), which is finally solved.

Listing 3: Example 0 - Solution

```
1  %% Solution
2  F=Fb+Fs; %total load vector
3  %define equivalent load vector
4  FF=F(freeDoF)-K(freeDoF,prescribedDoF)*u_prescribed(prescribedDoF);
5  %define equivalent stiffness matrix
6  KK=K(freeDoF,freeDoF);
```

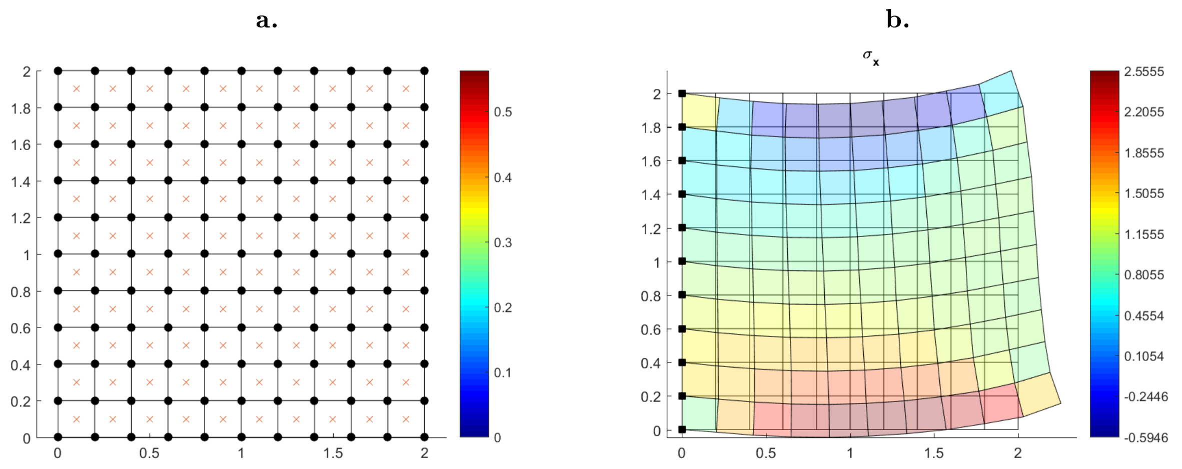


Figure 3: Figures produced in Example 0 (the minimal example). **a**. Undeformed mesh, nodes, and Gauss points. **b**. Deformed mesh. Colour reflects the xx-component of stress. Nodes on the left-hand boundary are pinned, as the Dirichlet boundary condition is prescribed there.

```
u=zeros(size(u_prescribed)); %initialise vector of displacements
%solve linear equations
u(freeDoF)=KK\FF;
```

Stress recovery requires the displacement vector to be reshaped into a list of vectors. We recover strains at Gauss points (centroids). The stresses are computed simply by multiplying the strains by the constant elasticity tensor.

Listing 4: Example 0 - Stress recovery

```
%% Stress recovery
u2=reshape(u,[2 numel(u)/2])';
%recover strains at element centroids
[strain, GPCoords]=recoveryGPEng(u2,nodeCoords,IEN,elementType,1);
%evaluate stresses at centroids
s2=CMatrix*strain;
```

The final part is visualisation. We produce two figures: the first figure shows undeformed mesh with nodes and Gauss points, the second figure plots the deformation and shows the pinned nodes at the left boundary, where the Dirichlet boundary condition is imposed, see Figure 3.

Listing 5: Example 0 - Visualisation

```
figure(1);clf;
drawElements(nodeCoords,IEN,elementType); %draw initial undeformed mesh
hold on;
drawNodes(nodeCoords); %draw nodes
drawNodes(GPCoords,'x'); %draw Gauss points

figure(2);clf;
drawElements(nodeCoords,IEN,elementType); %draw initial undeformed mesh
hold on;
factor=1e7; %scaling factor to make small deformations visible
%draw deformed mesh on top with alpha=.3
drawElements(nodeCoords+u2*factor,IEN,elementType,s2(1,:)',.3);
drawNodes(nodeCoords,BIEN{4},{'ks','filling'}); %draw pinned nodes
title('\sigma_x');
```

3 List of procedures

The package provides a number of routines, which can be categorised for clarity as shown in the Table 3. In this Section, syntax and details are listed for each routine. See Section 2 for the minimal example and suggested typical workflow. More examples can be found in Section 4.

Mesh generation and related procedures	
meshRect2d.m meshRect3d.m IENtoBIEN.m IENtoINE.m	edges2sublists.m meshP1toP2.m meshGetGlobal.m meshGetNatural.m
Assembly procedures	
formBC.m formBForceEng.m formBodyForceVector.m formInternalForce0.m formInternalForce2.m	formMassMatrix.m formStiffnessMatrix.m formStiffnessMatrix2.m formStiffnessMatrixEng.m formStiffnessMatrix_inc.m preventRigidMotion.m
Solver procedures	
\ (mldivide)	solveNewmarkLa.m solveNewmarkNLa.m
Recovery procedures	
recoveryAvg.m recoveryAvgEng.m recoveryEvaluateField.m recoveryEvaluateGradients.m recoveryEvaluateStrains.m	recoveryGP.m recoveryGPEng.m recoveryPatch.m recoveryPatchGP.m
Visualisation procedures	
drawElements.m drawElementsInterp.m drawNodes.m	
Element-specific functions	
elementData.m gaussPoints.m shapeFunctions1dQ1.m shapeFunctions1dQ2.m shapeFunctions2dP1.m shapeFunctions2dP2.m	shapeFunctions2dQ1.m shapeFunctions2dQ2.m shapeFunctions2dQ2r.m shapeFunctions3dP1.m shapeFunctions3dQ1.m
Constitutive functions	
elasticProperties.m	
Miscellaneous functions	
clonefig.m snapfig.m node2DoFs.m	tensorInvariant.m rotTensor.m
Verification procedures	
chkAbaqusBeamStatic.m chkAbaqusBeamOsc.m chkAll.m	parseAbaqRecs_1.m parseAbaqRecs_2.m import_displacements.m importMTX.m

Table 3: Routines contained in the package.

3.1 Mesh generation and related functions

3.1.1 meshRect2d

Create a mesh for a rectangular 2D domain.

Syntax:

```
[nodeCoords, IEN, boundaryElementIDs, boundaryNodeLocalIDs, e] = meshRect2d(domain, elementType, elementSize)
[nodeCoords, IEN, bndElIDs, bndNodeLocIDs] = meshRect2d([0 0; 1 1], '2dQ1', [10 10])
[nodeCoords, IEN, bndElIDs, bndNodeLocIDs] = meshRect2d([0 0 1 1], '2dQ1', .1)
```

Input parameters:

- `domain` (optional) - 1-by-4 matrix `[x1 y1 x2 y2]` or 2-by-2 `[x1 y1; x2 y2]` with coordinates defining the domain. Default value - `[-2 -2 2 2]`.
- `elementType` (optional) - single string, the type of elements. Default value - `'2dQ1'`.
- `elementSize` (optional) - a single real or 2-by-1 matrix. A single real defines the maximum diameter of elements. A 2-by-1 matrix defines the number of "cells" in each row and column of the rectangular grid. Each "cell" is a quadrilateral element or a pair of triangular elements. Default value - `sqrt(2)`.

Output parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ matrix, coordinates of nodes. $\#Dim = 2$.
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `boundaryElementIDs` - $\#BR$-by-1 cell array, $\#BR = 4$. Cell `boundaryElementIDs{p}` is a l_p-by-1 matrix listing IDs of elements at boundary p.
- `boundaryNodeLocalIDs` - $\#BR$-by-1 cell array, $\#BR = 4$. Cell `boundaryNodeLocalIDs{p}` contains a l_p-by-q matrix listing local nodal IDs (within an element). `boundaryNodeLocalIDs(m,1)` and `boundaryNodeLocalIDs(m,end)` define the endpoints of the boundary edge of the element number `boundaryElementIDs(m)`.
- `e` - edges, the data structure supported by Pde Toolbox, defines the boundaries of the domain.

The supported element types are '2dQ1', '2dQ2r', '2dQ2', '2dP1', '2dP2'. Within each row `IEN(i,:)`, vertices are listed before the midpoints. All nodes are listed anti-clockwise. The four cells in `boundaryElementIDs` and `boundaryNodeLocalIDs` correspond respectively to LOWER, RIGHT, UPPER and LEFT boundaries of the rectangular domain

3.1.2 meshRect3d

Create a mesh for a cuboid 3D domain.

Syntax:

```
[nodeCoords, IEN, bndElIDs, bndNodeLocIDs] = meshRect3d(domain, elementType, elementSize)
[nodeCoords, IEN, bndElIDs, bndNodeLocIDs] = meshRect3d([0 0 0; 1 1 1], '3dQ1', [10 10 10])
[nodeCoords, IEN, bndElIDs, bndNodeLocIDs] = meshRect3d([0 0 0 1 1 1], '3dQ1', .1)
[nodeCoords, IEN, boundaryElementIDs, boundaryNodeLocalIDs] = meshRect3d(domain, elementType, elementSize)
```

Input parameters:

- `domain` (optional) - 1-by-6 matrix `[x1 y1 z1 x2 y2 z2]` or 2-by-3 `[x1 y1 z1; x2 y2 z2]` with coordinates defining the domain. Default value - `[-2 -2 -2 2 2 2]`.
- `elementType` (optional) - single string, the type of elements. Default value - `'3dQ1'`.
- `elementSize` (optional) - a single real or 3-by-1 matrix. A single real defines the maximum diameter of elements. A 3-by-1 matrix defines the number of "cells" in each row and column of the rectangular grid. Each "cell" is a hexahedron or 5 tetrahedra depending on the element type. Default value - `sqrt(2)`.

Output parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ matrix, coordinates of nodes. $\#Dim = 3$.
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `boundaryElementIDs` - $\#BR$-by-1 cell array, $\#BR = 6$. Cell `boundaryElementIDs{p}` is a l_p-by-1 matrix listing IDs of elements at boundary p.
- `boundaryNodeLocalIDs` - $\#BR$-by-1 cell array, $\#BR = 6$. Cell `boundaryNodeLocalIDs{p}` contains a l_p-by-q matrix listing local nodal IDs (within an element). `boundaryNodeLocalIDs(m,1)` and `boundaryNodeLo~calIDs(m,end)` define the endpoints of the boundary edge of the element number `boundaryElementIDs(m)`.

The supported element types are `'3dQ1'`, `'3dP1'`. Within each row `IEN(i,:)`, vertices are listed before the midpoints. The six cells in `boundaryElementIDs` and `boundaryNodeLocalIDs` correspond respectively to BOTTOM, TOP, LEFT, RIGHT, FRONT and BACK boundaries of the domain, i.e. corresponding to $-z$, $+z$, $-x$, $+x$, $-y$, $+y$ directions.

3.1.3 IENtoBIEN

Restrict a given element-node incidence array (`IEN`) to given lists of elements and nodes within each element.

Syntax:
`BIEN = IENtoBIEN(IEN,boundaryElementIDs,boundaryNodeLocalIDs)`
Input parameters:

- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `boundaryElementIDs` - $\#BR$-by-1 cell array: p-th cell contains l_p-by-1 matrix listing boundary element IDs.
- `boundaryNodeLocalIDs` - $\#BR$-by-1 cell array: p-th cell contains l_p-by-q matrix listing local nodal IDs (within an element) of corresponding boundary nodes.

Output parameters

- `BIEN` - a $\#BR$-by-1 cell array. Cell `BIEN{k}` contains a l_p-by-$\#BNpE$ element-node incidence array corresponding to boundary region k. The order of IDs within row `BIEN{k}(i,:)` follows the order in `boundaryN~odeLocalIDs`.

This routine is useful for representing the boundary of a mesh as a sub-mesh.

3.1.4 IENtoINE

Compute INE, node-elements incidence array (dual to IEN up to permutations within each row).
`[INE, degrees, localIDs] = IENtoINE(IEN)`
Input parameters:

- IEN - $\#El$-by-$\#NpE$ array, element-node incidence matrix.

Output parameters:

- `INE` - $\#Nodes$-by-$\#EpN$ matrix of node-element incidence. Row `INE(i,:)` corresponds to ith node and contains IDs of elements that share it. Since the number of incident elements varies over nodes, $\#EpN$ is in fact the maximum nodal valency (degree) and the unused entries in each row are filled with zeros.
- `degrees` - $\#Nodes$-by-1 vector containing degrees/valencies of nodes, `degrees(i)` is the number of elements node i is incident to.
- `localIDs` - $\#Nodes$-by-$\#EpN$ array, `localIDs(i,j)` is the local ID of node i within the element `INE(i,j)`. Thus, `INE(i,:)` and `localIDs(i,:)` contain respectively the row and column subscripts of each entry of node i within `IEN`.

Computation of IEN is required when one intends to consider a node and its neighbourhood of elements. For instance, superconvergent patch recovery algorithm (SPR) considers patches surrounding each vertex node, see `recoveryPatchGP()` for details.

3.1.5 edges2sublists

Convert Matlab PDE Toolbox (2D) edge representation to sublists of boundary elements and corresponding local node IDs within each boundary element.

Syntax:

`[boundaryElementIDs, boundaryNodeLocalIDs]=edges2sublists(e,IEN,INE);`

Input parameters:

- `e` - 7-by-k matrix, edge representation produced by PDE Toolbox `initmesh()`
- `IEN` -$\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `INE` - $\#Nodes$-by-$\#EpN$ array, node-element incidence matrix, row `INE(i,:)` corresponds to ith node and contains IDs of elements that share it.

Output parameters:

- `boundaryElementIDs` - $\#BR$-by-1 cell array: p-th cell contains l_p-by-1 matrix listing boundary element IDs.
- `boundaryNodeLocalIDs` - $\#BR$-by-1 cell array: p-th cell contains l_p-by-$\#BNpE$ matrix listing local nodal IDs (within an element) of corresponding boundary nodes. `boundaryNodeLocalIDs(m,1)` and `boundaryN~ odeLocalIDs(m,end)` define the boundary edge of the element number `boundaryElementIDs(m)`. Note that $l_1 + ... + l_{\#B} \leq k$.

Examples:

```
[p,e,t] = initmesh(dl,'Hmax',hmax);
nodeCoords=p';
IEN=IEN(1:3,:)';
INE=IENtoINE(IEN);
[boundaryElementIDs, boundaryNodeLocalIDs]=edges2sublists(e,IEN,INE);
```

Only works for meshes consisting of 3-node triangular elements. Only boundaries with the exterior are taken into account, while interfaces between subdomains are disregarded.

3.1.6 meshP1toP2

Convert `'2dP1'` mesh (linear, 3-node triangles) to `'2dP2'` mesh (quadratic, 6-nodes triangles) by adding a midpoint to every edge.

Syntax:

`[nodeCoords2, IEN2, boundaryElementIDs2, boundaryNodeLocalIDs2] = meshP1toP2(nodeCoords, IEN, boundaryElementIDs, boundaryNodeLocalIDs)`

Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ matrix, coordinates of nodes.
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix, $\#NpE = 3$.
- `boundaryElementIDs` - $\#BR$-by-1 cell array. Cell `boundaryElementIDs{p}` is a l_p-by-1 matrix listing IDs of elements at boundary p.
- `boundaryNodeLocalIDs` - $\#BR$-by-1 cell array. Cell `boundaryNodeLocalIDs{p}` contains a l_p-by-q_1 matrix listing local nodal IDs (within an element). `boundaryNodeLocalIDs(m,1)` and `boundaryNodeLocalIDs(m,q_1)` define the endpoints of the boundary edge of the element number `boundaryElementIDs(m)`.

Output parameters:

- `nodeCoords2` - $\#Nodes2$-by-$\#Dim$ matrix, coordinates of nodes in the new mesh.
- `IEN2` - $\#El$-by-$\#NpE2$ array, element-node incidence matrix, $\#NpE2 = 6$.
- `boundaryElementIDs2` - $\#BR$-by-1 cell array. Cell `boundaryElementIDs2{p}` is a l_p-by-1 matrix listing IDs of elements at boundary p.

- `boundaryNodeLocalIDs2` - $\#BR$-by-1 cell array. Cell `boundaryNodeLocalIDs2{p}` contains a l_p-by-q_2 matrix listing local nodal IDs (within an element). `boundaryNodeLocalIDs2(m,1)` and `boundaryNodeLo~calIDs2(m,q_2)` define the endpoints of the boundary edge of the element number `boundaryElementIDs2(m)`.

`boundaryElementIDs2` and `boundaryNodeLocalIDs2` are computed only if `boundaryElementIDs` and `boundaryN~odeLocalIDs` were provided as input. In this implementation `boundaryElementIDs2` equals `boundaryElementIDs`, i.e. the order of elements in the boundaries is preserved.

3.1.7 meshGetNatural

Map global coordinates to natural coordinates.
Syntax:
`[el, ptNatCoords] = meshGetNatural(nodeCoords,IEN,elementType,INE,ptCoords)`
Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ matrix, coordinates of nodes.
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `elementType` - a single string defining the type of elements in the mesh.
- `ptCoords` - k-by-$\#Dim$ matrix, row `ptCoords(i,:)` contains global coordinates of the query point i.

Output parameters:

- `el` - k-by-1 matrix, element IDs of the query points.
- `ptNatCoords` - k-by-$\#ElDim$ matrix, natural coordinates of the query points.

This routine uses Newton's iterative method to find the pre-image in higher-order elements. If a preimage of some particular point was not found, a warning message is generated and the corresponding entry in the output array `el` is zeros. This routine uses the `'inElementTol'` property of `elementData`, which allows to tell whether a requested point belongs to the element's natural domain up to some tolerance. See `elementData.m` for details.

The maximum number of Newton iterations and absolute tolerance are defined in the file.

3.1.8 meshGetGlobal

Map natural coordinates to global coordinates.
Syntax:
`ptCoords=meshGetGlobal(nodeCoords,IEN,elementType,el,ptNatCoords)`
Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ matrix, coordinates of nodes.
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `elementType` - a single string defining the type of elements in the mesh.
- `el` - k-by-1 matrix, element IDs of the query points.
- `ptNatCoords` - k-by-$\#ElDim$ matrix, natural coordinates of the query points.

Output parameters:

- `ptCoords` - k-by-$\#Dim$ matris, row `ptCoords(i,:)` contains global coordinates of the query point i.

Since isoparametric elements are used, the procedure is identical to `recoveryEvaluateField().`

3.2 Matrix and vector assembly

3.2.1 formStiffnessMatrix

Compute stiffness matrix.

formStiffnessMatrix	$\sum\limits_{e}\int\limits_{\Omega_h^e} \nabla \mathrm{H}_{Ir}\hat{\mathrm{c}}_{IJ}\nabla \mathrm{H}_{Js}\mathrm{d}\Omega$
formStiffnessMatrix2	$\sum\limits_{e}\int\limits_{\Omega_h^e} \nabla \mathrm{H}_{Ir}\left(\hat{\mathrm{c}}_{IJ}\nabla \mathrm{H}_{Js} + \beta_{Ik}\mathrm{H}_{ks}\right)\mathrm{d}\Omega$
formStiffnessMatrixEng	$\sum\limits_{e}\int\limits_{\Omega_h^e} B_{Ir}\hat{\mathrm{c}}_{IJ}B_{Js}\mathrm{d}\Omega$
formMassMatrix	$\sum\limits_{e}\int\limits_{\Omega_h^e} \mathrm{H}_{ir}\rho_{ij}\mathrm{H}_{js}\mathrm{d}\Omega$
formBodyForceVector	$\sum\limits_{e}\int\limits_{\Omega_h^e} \mathrm{H}_{ir}\mathrm{f}_i\mathrm{d}\Omega, \quad \sum\limits_{e}\int\limits_{\partial\Omega_{h,n}^e} \mathrm{H}_{ir}\bar{\mathrm{t}}_i\mathrm{d}S$
formInternalForce0	$\sum\limits_{e}\int\limits_{\Omega_h^e} \nabla \mathrm{H}_{Ir}\sigma_r\mathrm{d}\Omega$
formInternalForce2	$\sum\limits_{e}\int\limits_{\Omega_h^e} \nabla \mathrm{H}_{Ir}\left(\hat{\mathrm{c}}_{IJ}\nabla \mathrm{H}_{Js} + \beta_{Ik}\mathrm{H}_{ks}\right)\hat{u}_s\mathrm{d}\Omega$

Table 4: Summary of matrix and vector assembly procedures.

Syntax:
```
K = formStiffnessMatrix(nodeCoords, IEN, elementType, numGP, CMatrix)
K = formStiffnessMatrix(nodeCoords, IEN, elementType, numGP, CMatrix, u, s)
K = formStiffnessMatrix(nodeCoords, IEN, elementType, numGP, CMatrix, u)
K = formStiffnessMatrix(nodeCoords, IEN, elementType, numGP, CMatrix, [], s)
[K, argout] = formStiffnessMatrix(...)
```
Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates. Each row `nodeCoords(i,:)` contains coordinates of the ith node.
- `IEN` - $\#El$-by-$\#NpE$ array of element-node incidence. Each row `IEN(i,:)` contains IDs of nodes incident to the ith element. The order of nodal IDs is specific to the type of element.
- `elementType` - a single string defining the type of elements used in the mesh.
- `numGP` - single scalar, number of Gauss points per element.
- `CMatrix` - $\#Eq\times\#Dim$-by-$\#Eq\times\#Dim$ matrix, a vectorised form of **c** in (1), or a function handle returning such matrix. The `CMatrix` input is `x_` (1-by-$\#Dim$), `u_` (1-by-$\#Eq$), `gradu_` ($\#Dim$-by-$\#Eq$), `s_` (1-by-k), respectively the values of position, displacement, displacement gradient and state variables at Gauss points. `x_`, `u_`, and `gradu_` are interpolated; `s_` is provided by the user. Function `CMatrix` may have a second output argument, which is a 1-by-$\#AO$ cell array. This additional output is then collected at Gauss points and returned as additional output of `formStiffnessMatrix()` at the global level.
- `u` (optional) - $\#Nodes$-by-$\#Eq$ matrix, nodal displacements used to compute `u_`, and `gradu_` at each Gauss point before passing them as arguments to `CMatrix`(). Default value - all zeros.
- `s` (optional) - $\#El\times\#GP$-by-k matrix, internal variables at Gauss points. `s(`$(m-1)*\#GP+l$`,:)` is passed to CMatrix at l-th Gauss point of m-th element. Default values - all zeros, $k=1$.

Output parameters:

- `K` - $\#Nodes\times\#Eq$-by-$\#Nodes\times\#Eq$ stiffness matrix
- `argout` - 1-by-$\#AO$ cell array, where $\#AO$ is the number of additional outputs of CMatrix(). `argout{i}` is a $\#El\times\#GP\times n1(i)$-by-$n2(i)$ matrix, where $[n1(i)\ n2(i)]$ is the size of i-th additional output of `CMatrix()`.

The procedure loops through each element computing a local element stiffness matrix and adding it to appropriate columns and rows of the global stiffness matrix. The local stiffness matrix is given by

$$\mathrm{K}_{rs}^{e} = \int\limits_{\Omega_h^e} \nabla \mathrm{H}_{Ir}\left(\hat{\mathrm{c}}_{IJ}\nabla \mathrm{H}_{Js}\right)\mathrm{d}\Omega. \tag{31}$$

3.2.2 formStiffnessMatrixEng

Compute stiffness matrix. This function uses the short form of constitutive modulus, as in (26), and is otherwise similar to `formStiffnessMatrix()`.

Syntax:

```
K = formStiffnessMatrixEng(nodeCoords, IEN, elementType, numGP, CMatrix, u, s)
K = formStiffnessMatrixEng(nodeCoords, IEN, elementType, numGP, CMatrix, u)
K = formStiffnessMatrixEng(nodeCoords, IEN, elementType, numGP, CMatrix)
K = formStiffnessMatrixEng(nodeCoords, IEN, elementType, numGP, CMatrix, [], s)
[K, argout] = formStiffnessMatrixEng(...)
```

Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates. Each row `nodeCoords(i,:)` contains coordinates of the ith node.
- `IEN` - $\#El$-by-$\#NpE$ array of element-node incidence. Each row `IEN(i,:)` contains IDs of nodes incident to the ith element. The order of nodal IDs is specific to the type of element.
- `elementType` - a single string defining the type of elements used in the mesh.
- `numGP` - single scalar, number of Gauss points per element.
- `CMatrix` - 3-by-3 or 6-by-6 matrix, the constitutive modulus in Voigt notation, or a function handle returning such matrix. The `CMatrix` input is `x_` (1-by-$\#Dim$), `u_` (1-by-$\#Eq$), `gradu_` ($\#Dim$-by-$\#Eq$), `s_` (1-by-k), respectively the values of position, displacement, displacement gradient and state variables at Gauss points. `x_`, `u_`, and `gradu_` are interpolated; `s_` is provided by the user. Function `CMatrix` may have a second output argument, which is a 1-by-$\#AO$ cell array. This additional output is then collected at Gauss points and returned at the global level as additional output of `formStiffnessMatrix()`.
- `u` (optional) - $\#Nodes$-by-$\#Eq$ matrix, nodal displacements used to compute `u_`, and `gradu_` at each Gauss point before passing them as arguments to `CMatrix`(). Default value - all zeros.
- `s` (optional) - $\#El \times \#GP$-by-k matrix, internal variables at Gauss points. `s(`$(m-1)*\#GP+l$`,:)` is passed to CMatrix at l-th Gauss point of m-th element. Default values - all zeros, $k = 1$.

Output parameters:

- `K` - $\#Nodes \times \#Eq$-by-$\#Nodes \times \#Eq$ stiffness matrix
- `argout` - 1-by-$\#AO$ cell array, where $\#AO$ is the number of additional outputs of CMatrix(). `argout{i}` is a $\#El \times \#GP \times n1(i)$-by-$n2(i)$ matrix, where $[n1(i)\ n2(i)]$ is the size of i-th additional output of `CMatrix()`.

The procedure loops through each element computing a local element stiffness matrix and adding it to appropriate columns and rows of the global stiffness matrix. The local stiffness matrix is given by

$$\mathrm{K}^e_{rs} = \int\limits_{\Omega^e_h} B_{Ir}\left(\hat{\mathrm{c}}_{IJ} B_{Js}\right) \mathrm{d}\Omega. \tag{32}$$

3.2.3 formStiffnessMatrix2

Compute stiffness matrix. This function similar to `formStiffnessMatrix()`, but includes a contribution arising directly from the field interpolation (in addition to that arising form the gradient interpolation).

Syntax:

```
K = formStiffnessMatrix2(nodeCoords, IEN, elementType, numGP, CMatrix)
K = formStiffnessMatrix2(nodeCoords, IEN, elementType, numGP, CMatrix, u, s)
K = formStiffnessMatrix2(nodeCoords, IEN, elementType, numGP, CMatrix, u)
K = formStiffnessMatrix2(nodeCoords, IEN, elementType, numGP, CMatrix, [], s)
[K, argout] = formStiffnessMatrix2(...)
```

Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates. Each row `nodeCoords(i,:)` contains coordinates of the ith node.

- `IEN` - $\#El$-by-$\#NpE$ array of element-node incidence. Each row `IEN(i,:)` contains IDs of nodes incident to the ith element. The order of nodal IDs is specific to the type of element.
- `elementType` - a single string defining the type of elements used in the mesh.
- `numGP` - single scalar, number of Gauss points per element.
- `CMatrix` - $\#Eq \times \#Dim$-by-$\#Eq \times \#Dim$ matrix, a vectorised form of **c** in (1), or a function handle returning such matrix. The `CMatrix` input is `x_` (1-by-$\#Dim$), `u_` (1-by-$\#Eq$), `gradu_` ($\#Dim$-by-$\#Eq$), `s_` (1-by-k), respectively the values of position, displacement, displacement gradient and state variables at Gauss points. `x_`, `u_`, and `gradu_` are interpolated; `s_` is provided by the user. Function `CMatrix` may have a second output argument, which is a 1-by-$\#AO$ cell array. This additional output is then collected at Gauss points and returned at the global level as additional output of `formStiffnessMatrix()`.
- `BetaMatrix` - $\#Eq \times \#Dim$-by-$\#Eq$ matrix, a vectorised form of **c** in (1), or a function handle returning such matrix. The input parameters of `BetaMatrix` are similar to those of `CMatrix`. Additional output is not collected.
- `u` (optional) - $\#Nodes$-by-$\#Eq$ matrix, nodal displacements used to compute `u_`, and `gradu_` at each Gauss point before passing them as arguments to `CMatrix`(). Default value - all zeros.
- `s` (optional) - $\#El \times \#GP$-by-k matrix, internal variables at Gauss points. `s(`$(m-1)*\#GP+l$`,:)` is passed to CMatrix at l-th Gauss point of m-th element. Default values - all zeros, $k = 1$.

Output parameters:

- `K` - $\#Nodes \times \#Eq$-by-$\#Nodes \times \#Eq$ stiffness matrix
- `argout` - 1-by-$\#AO$ cell array, where $\#AO$ is the number of additional outputs of CMatrix(). `argout{i}` is a $\#El \times \#GP \times n1(i)$-by-$n2(i)$ matrix, where $[n1(i)\ n2(i)]$ is the size of i-th additional output of `CMatrix()`.

The procedure loops through each element computing a local element stiffness matrix and adding it to appropriate columns and rows of the global stiffness matrix. The local stiffness matrix is given by

$$\mathrm{K}^e_{rs} = \int\limits_{\Omega^e_h} \nabla \mathrm{H}_{Ir} \left(\hat{\mathrm{c}}_{IJ} \nabla \mathrm{H}_{Js} + \beta_{Ik} \mathrm{H}_{ks} \right) \mathrm{d}\Omega. \tag{33}$$

3.2.4 formStiffnessMatrix_inc

Compute stiffness matrix. This function differs from formStiffnessMatrix() in that the displacement increment `du` is provided for evaluation of `CMatrix`.

Syntax:

```
K = formStiffnessMatrix_inc(nodeCoords, IEN, elementType, numGP, CMat, u, du, s)
K = formStiffnessMatrix_inc(nodeCoords, IEN, elementType, numGP, CMat, [], du, s)
K = formStiffnessMatrix_inc(nodeCoords, IEN, elementType, numGP, CMat, u, du, [])
[K, argout] = formStiffnessMatrix_inc(...)
```

Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates. Each row `nodeCoords(i,:)` contains coordinates of the ith node.
- `IEN` - $\#El$-by-$\#NpE$ array of element-node incidence. Each row `IEN(i,:)` contains IDs of nodes incident to the ith element. The order of nodal IDs is specific to the type of element.
- `elementType` - a single string defining the type of elements used in the mesh.
- `numGP` - single scalar, number of Gauss points per element.
- `CMatrix` - a function handle returning a $\#Eq \times \#Dim$-by-$\#Eq \times \#Dim$ matrix, a vectorised form of **c** in (1). The `CMatrix` input is `x_` (1-by-$\#Dim$), `u_` (1-by-$\#Eq$), `gradu_` ($\#Dim$-by-$\#Eq$), `s_` (1-by-k), `du_` (1-by-$\#Eq$), `dgradu_` ($\#Dim$-by-$\#Eq$), respectively the values of position, displacement, displacement gradient,

state variables at a Gauss points, displacement increment and displacement gradient increment. `x_`, `u_`, `gradu_`, `du_`, and `dgradu_` are interpolated; `s_` is provided by the user. Function `CMatrix` may have a second output argument, which is a 1-by-$\#AO$ cell array. This additional output is then collected at Gauss points and returned as additional output of `formStiffnessMatrix()` at the global level.

- `u` (optional) - $\#Nodes$-by-$\#Eq$ matrix, nodal displacements used to compute `u_`, and `gradu_` at each Gauss point before passing them as arguments to `CMatrix`(). Default value - all zeros.
- `s` (optional) - $\#El \times \#GP$-by-k matrix, internal variables at Gauss points. `s(`$(m-1)*\#GP+l$`,:)` is passed to `CMatrix()` at l-th Gauss point of m-th element. Default values - all zeros, $k = 1$.

Output parameters:

- `K` - $\#Nodes \times \#Eq$-by-$\#Nodes \times \#Eq$ stiffness matrix
- `argout` - 1-by-$\#AO$ cell array, where $\#AO$ is the number of additional outputs of CMatrix(). `argout{i}` is a $\#El \times \#GP \times n1(i)$-by-$n2(i)$ matrix, where $[n1(i)\ n2(i)]$ is the size of i-th additional output of `CMatrix()`.

This routine simply rearranges input and output parameters and invokes `formStiffnessMatrix()`.

3.2.5 formMassMatrix

Compute mass matrix.

Syntax:

```
M = formMassMatrix(nodeCoords, IEN, elementType, numGP, MMat)
M = formMassMatrix(nodeCoords, IEN, elementType, numGP, MMat, u)
M = formMassMatrix(nodeCoords, IEN, elementType, numGP, MMat, [], s)
M = formMassMatrix(nodeCoords, IEN, elementType, numGP, MMat, u, s)
```

Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates. Each row `nodeCoords(i,:)` contains coordinates of the ith node.
- `IEN` - $\#El$-by-$\#NpE$ array of element-node incidence. Each row `IEN(i,:)` contains IDs of nodes incident to the ith element. The order of nodal IDs is specific to the type of element.
- `elementType` - a single string defining the type of elements used in the mesh.
- `numGP` - single scalar, number of Gauss points per element.
- `MMatrix` - a $\#Eq$-by-$\#Eq$ matrix or a function handle returning such matrix. The `MMatrix` input is `x_` (1-by-$\#Dim$), `u_` (1-by-$\#Eq$), `gradu_` ($\#Dim$-by-$\#Eq$), `s_` (1-by-k), respectively the values of position, displacement, displacement gradient and state variables at Gauss points. `x_`, `u_`, and `gradu_` are interpolated; `s_` is provided by the user.
- `u` (optional) - $\#Nodes$-by-$\#Eq$ matrix, nodal displacements used to compute `u_`, and `gradu_` at each Gauss point before passing them as arguments to `MMatrix`(). Default value - all zeros.
- `s` (optional) - $\#El \times \#GP$-by-k matrix, internal variables at Gauss points. `s(`$(m-1)*\#GP+l$`,:)` is passed to `MMatrix`() at l-th Gauss point of m-th element. Default values - all zeros, $k = 1$.

Output parameters:

- `M` - $\#Nodes \times \#Eq$-by-$\#Nodes \times \#Eq$ mass matrix.

The procedure loops through each element computing a local element mass matrix and adding it to appropriate columns and rows of the global mass matrix. The local mass matrix is given by

$$\mathrm{M}^e_{rs} = \int_{\Omega^e_h} \mathrm{H}_{ir}\rho_{ij}\mathrm{H}_{js}\mathrm{d}\Omega. \tag{34}$$

This routine can also be used to compute the damping matrix **C**, which has exactly the same form as above.

3.2.6 formInternalForce2

Compute equivalent (internal) nodal force vector from given nodal displacements.

Syntax:

```
F = formNodalForce2(nodeCoords, IEN, elementType, numGP, CMatrix, BetaMatrix, u, s)
F = formNodalForce2(nodeCoords, IEN, elementType, numGP, CMatrix, BetaMatrix, u)
[F, argout] = formNodalForce2(...)
```

Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates. Each row `nodeCoords(i,:)` contains coordinates of the ith node.
- `IEN` - $\#El$-by-$\#NpE$ array of element-node incidence. Each row `IEN(i,:)` contains IDs of nodes incident to the ith element. The order of nodal IDs is specific to the type of element.
- `elementType` - a single string defining the type of elements used in the mesh.
- `numGP` - single scalar, number of Gauss points per element.
- `CMatrix` - $\#Eq2 \times \#Dim$-by-$\#Eq1 \times \#Dim$ matrix, a vectorised form of $\mathbf{c}$ in (1), or a function handle returning such matrix. The `CMatrix` input is `x_` (1-by-$\#Dim$), `u_` (1-by-$\#Eq$), `gradu_` ($\#Dim$-by-$\#Eq$), `s_` (1-by-k), respectively the values of position, displacement, displacement gradient and state variables at Gauss points. `x_`, `u_`, and `gradu_` are interpolated; `s_` is provided by the user. Function `CMatrix` may have a second output argument, which is a 1-by-$\#AO$ cell array. This additional output is then collected at Gauss points and returned at the global level as additional output of `formStiffnessMatrix()`.
- `BetaMatrix` - $\#Eq2 \times \#Dim$-by-$\#Eq1$ matrix, a vectorised form of $\mathbf{c}$ in (1), or a function handle returning such matrix. The input parameters of `BetaMatrix` are similar to those of `CMatrix`. Additional output is not collected.
- `u` - $\#Nodes$-by-$\#Eq$ matrix, nodal displacements used to compute `u_`, and `gradu_` at each Gauss point before passing them as arguments to `CMatrix`(). Default value - all zeros.
- `s` (optional) - $\#El \times \#GP$-by-k matrix, internal variables at Gauss points. `s(`$(m-1)*\#GP+l$`,:)` is passed to CMatrix at l-th Gauss point of m-th element. Default values - all zeros, $k = 1$.

Output parameters:

- `F` - $\#Nodes \times \#Eq$-by-1 force vector.

The equivalent internal forces are defined as

$$\mathbf{f}^{\mathrm{i}} = \nabla \cdot (\mathbf{c} : \nabla \mathbf{u} + \boldsymbol{\beta} \cdot \mathbf{u}). \tag{35}$$

The procedure loops through each element computing a local element force vector and adding it to appropriate rows of the global force vector. The local force vector is given by

$$\mathrm{F}^{ie}_{r} = \int\limits_{\Omega^e_h} \nabla \mathrm{H}_{Ir} \left(\hat{\mathrm{c}}_{IJ} \nabla \mathrm{H}_{Js} + \beta_{Ij} \mathrm{H}_{js} \right) \hat{\mathrm{u}}_s \mathrm{d}\Omega.$$

3.2.7 formInternalForce0

Compute equivalent internal nodal force vector from stress values given at Gauss points.

Syntax:

```
F = formNodalForce0(nodeCoords, IEN, elementType, numGP, s)
```

Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates. Each row `nodeCoords(i,:)` contains coordinates of the ith node.
- `IEN` - $\#El$-by-$\#NpE$ array of element-node incidence. Each row `IEN(i,:)` contains IDs of nodes incident to the ith element. The order of nodal IDs is specific to the type of element.

- `elementType` - a single string defining the type of elements used in the mesh.
- `s` - $\#El \times \#GP$-by-$\#Eq \times \#Dim$ matrix, stress values at Gauss points. Each row `s((m-1)*#GP+l,:)` corresponds to the stress tensor of at l-th Gauss point of m-th element.

Output parameters:

- `F` - $\#Nodes \times \#Eq$-by-1 force vector.

The equivalent internal forces are defined as

$$\mathbf{f}^{\mathrm{i}} = \nabla \cdot \boldsymbol{\sigma}. \tag{36}$$

The procedure loops through each element computing a local element force vector and adding it to appropriate rows of the global force vector. The local force vector is given by

$$\mathrm{F}_r^{ie} = \int\limits_{\Omega_h^e} \nabla \mathrm{H}_{Ir} \sigma_I \mathrm{d}\Omega.$$

3.2.8 formBodyForceVector

Compute body force vector.

Syntax:

```
F = formBodyForceVector(nodeCoords, IEN, elementType, numGP, [0 -9.8*8.05])
F = formBodyForceVector(nodeCoords, IEN, elementType, numGP, bodyForce)
```

Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates. Each row `nodeCoords(i,:)` contains coordinates of the ith node.
- `IEN` - $\#El$-by-$\#NpE$ array of element-node incidence. Each row `IEN(i,:)` contains IDs of nodes incident to the ith element. The order of nodal IDs is specific to the type of element.
- `elementType` - a single string defining the type of elements used in the mesh.
- `numGP` - single scalar, number of Gauss points per element.
- `bodyForce` - a $\#Eq$-by-1 matrix or a function handle returning such matrix. The function input is `1-by-`$\#Dim$ position vector.

Output parameters:

- `F` - $\#Nodes \times \#Eq$-by-1 body force vector.

The procedure loops through each element computing a local element force vector and adding it to appropriate rows of the global force vector. The local force vector is given by

$$\mathrm{F^{Be}}_r = \int\limits_{\Omega_h^e} \mathrm{H}_{ir} \mathrm{f}_i \mathrm{d}\Omega. \tag{37}$$

This routine is also suitable to compute surface traction vector $\mathbf{F}^{\mathrm{S}}$. In this case `IEN`, `elementType`, and `numGP` must correspond to boundary submesh. Element surface traction vector is given by

$$\mathrm{F}_{\mathrm{r}}^{\mathrm{Se}} = \int\limits_{\partial\Omega_{h,n}^e} \mathrm{H}_{ir} \bar{t}_i \mathrm{d}S.$$

3.2.9 formBC

Process boundary conditions and return prescribed displacements, force vector, and degrees of freedom lists.

Syntax:

```
[u_prescribed, Fs, prescribedDoF, freeDoF]=formBC(nodeCoords, BIEN, elementType, numGPs,
boundaryTractions, boundaryDisplacements, isDirichlets)
```

`[u_prescribed, Fs, prescribedDoF, freeDoF]=formBC(nodeCoords, BIEN, elementType, numGPs, boundaryTractions, boundaryDisplacements, isDirichlets,[1 2])`

Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates. Each row `nodeCoords(i,:)` contains coordinates of the ith node.
- `BIEN` - a single boundary incidence matrix or a $\#BR$-by-1 cell array of such matrices, where $\#BR$ is the number of boundary regions.
- `elementType` - a single string or a $\#BR$-by-1 cell array of strings, which define element types for each boundary region.
- `numGPs` - a single scalar or a $\#BR$-by-1 matrix, which defines the number of Gauss points used for boundary submesh elements within each boundary region.
- `boundaryTractions` - a single function handle or a $\#BR$-by-1 cell array of function handles. Functions must take k-by-$\#Dim$ matrix as input (coordinates) and return a $\#Eq$-by-k matrix as output (tractions).
- `boundaryDisplacements` - a single function handle or a $\#BR$-by-1 cell array of function handles. Functions must take k-by-$\#Dim$ matrix as input (coordinates) and return a $\#Eq$-by-k matrix as output (displacements).
- `isDirichlets` - a $\#BR$-by-1 or $\#BR$-by-$\#Eq$ matrix of ones and zeros, or a function handle that takes a k-by-$\#Dim$ matrix (coordinates) and returns a k-by-$\#Eq$ matrix of ones and zeros. Ones and zeros indicate Dirichlet and Neumann boundary conditions respectively.
- `pinnedDoFs` (optional) - a list of degrees of freedom IDs, which will be added to the list of `prescribedDoF` and removed from `freeDoF`. Default value - `[]`.
- `pinnedVals` (optional) - a list of values for the pinned degrees of freedom. Default value - `0`.

Output parameters:

- `u_prescribed` - $\#Nodes$-by-$\#Eq$ matrix containing Dirichlet data evaluated at boundary nodes and zeros at other nodes.
- `Fs` - $\#GDoF$-by-1 vector containing equivalent nodal forces corresponding to Neumann boundary condition.
- `prescribedDoF` - list of IDs of prescribed (fixed) degrees of freedom.
- `freeDoF` - list of IDs of free degrees of freedom.

Function `formBC()` determines prescribed and free degrees of freedoms by looping through each boundary region, evaluating prescribed displacements and boundary traction vectors using `formBodyForceVector`() function. If provided, `pinnedDoFs` is taken into account at the final stage.

3.2.10 preventRigidMotion

Append the list of prescribed DoFs if necessary to remove rigid body motions in elasticity problems.

Syntax:

`[pDoF,fDoF]=preventRigidMotion(nodeCoords,pDoF,fDoF)`

Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array, coordinates of nodes.
- `pDoF` - list of IDs of prescribed (fixed) degrees of freedom.
- `fDoF` - list of IDs of free degrees of freedom.

Output parameters:

- `pDoF` - updated list of IDs of prescribed (fixed) degrees of freedom.
- `fDoF` - updated list of IDs of free degrees of freedom.

If `pDoF` is initially empty, then no essential constraints are introduced. This is not guaranteed otherwise.

3.3 Solution

3.3.1 solveNewmarkLa.m

Newmark's method for a linear problem.

Syntax:

```
u=solveNewmarkLa(beta,gamma,ts,fDoF,pDoF,M,C,K,F,u0,v0,a0,up,vp,ap);
outputFunction=@(i,t,u,v,a,fDoF,pDoF,M,C,K,F,u0,v0,a0,up,vp,ap){t,u,v,a};
argsOut=solveNewmarkLa(beta,gamma,ts,fDoF,pDoF,M,C,K,F,u0,v0,a0,up,vp,ap,'increment',outputFunction);
```

Input parameters:

- `beta` - Newmark beta parameter.
- `gamma` - Newmark gamma parameter.
- `ts` - $TimeStep+$ 1-by-1 matrix of integration time points.
- `fDoF` - list of IDs of free degrees of freedom.
- `pDoF` - list of IDs of prescribed (fixed) degrees of freedom.
- `M` - ($\#Nodes \times \#Eq$)-by-($\#Nodes \times \#Eq$) mass matrix or a function handle returning such matrix. The input of the function is `t_` (time).
- `C` - ($\#Nodes \times \#Eq$)-by-($\#Nodes \times \#Eq$) damping matrix or a function handle returning such matrix. The input of the function is `t_` (time).
- `K` - ($\#Nodes \times \#Eq$)-by-($\#Nodes \times \#Eq$) stiffness matrix or a function handle returning such matrix. The input of the function is `t_` (time).
- `F` - ($\#Nodes \times \#Eq$)-by-1 external force vector or a function handle returning such vector. The input of the function is `t_` (time).
- `u0` - ($\#Nodes \times \#Eq$)-by-1 vector of initial displacements.
- `v0` - ($\#Nodes \times \#Eq$)-by-1 vector of initial velocities or an empty array.
- `a0` - ($\#Nodes \times \#Eq$)-by-1 vector of initial accelerations or an empty array.
- `up` - ($\#Nodes \times \#Eq$)-by-1 vector of prescribed displacements or a function handle returning such vector, or an empty array. The input of the function handle is t_ (time).
- `vp` - ($\#Nodes \times \#Eq$)-by-1 vector of prescribed velocities or a function handle returning such vector, or an empty array. The input of the function handle is t_ (time).
- `ap` - ($\#Nodes \times \#Eq$)-by-1 vector of prescribed accelerations or a function handle returning such vector, or an empty array. The input of the function handle is t_ (time).
- Options in the format `...,'Name',value,...`
  - `'increment', outputFunction` - function called at each time increment and supplied with `(i,ts(i),u,v,a,fDoF,pDo` as input. The output must be one or several matrices. The outputs are collected and returned at the end of the procedure.

Output parameters:

- If no `'increment'` option specified, then:
  - `u` -($\#Nodes \times \#Eq$)-by-1 vector of displacements at the final time point.
  - `v` - ($\#Nodes \times \#Eq$)-by-1 vector of velocities at the final time point.
  - `a` -($\#Nodes \times \#Eq$))-by-1 vector of accelerations at the final time point.
- If `'increment'` option specified, then:

  – `argsOut` - a cell array containing the outputs of outputFunction collected at each time step or Newton iteration. Each element `argsOut{k}` contains NI-by-$n1(k)$-by-...-$nq(k)$ matrix, where $n1(k)$-by-...-$nq(k)$) is the size of k-th output of `outputFunction` and NI is the total number of increments.

By default, the Dirichlet (essential) boundary conditions are given by prescribed displacements. In order to allow for different boundary conditions definition, `pDoF` can be replaced by `{pDoFu, pDoFv, pDoFa}` - lists of nodes at which displacements, velocities and accelerations are prescribed respectively. The corresponding entries of `up, vp,` and `ap` are used directly and via finite difference approximations, which normally match the approximation used in Newmark method.

The mass matrix must be non-singular (e.g., a lumped diagonal matrix)

Empty array `[]` can be passed as a value for input parameters `C, up, vp, ap, v0` and `a0`. In this case the arguments are replaced by zero arrays of appropriate size.

3.3.2 solveNewmarkNLa.m

Newmark's method for a nonlinear problem.

Syntax:

```
   u=solveNewmarkNLa(beta,gamma,ts,fDoF,pDoF,M,Ct,Kt,Ft,u0,v0,a0,up,vp,ap);
   u=solveNewmarkNLa(beta,gamma,ts,fDoF,pDoF,M,C,K,@(t,u,v)(K*u-Fext),u0,v0,a0,up,vp,ap); %linear
problem
   outputFunction1=@(i,t,u,v,a,fDoF,pDoF,M,Ct,Kt,Ft,u0,v0,a0,up,vp,ap){t,u,v,a};
   outputFunction2=@(i,t,u,v,a,fDoF,pDoF,M,Ct,Kt,Ft,u0,v0,a0,up,vp,ap)plot(t,u(1),'*');
   argsOut=solveNewmarkLa(beta,gamma,ts,fDoF,pDoF,M,C,K,F,u0,v0,a0,up,vp,ap,'increment',outputFunction2,'i
```

Input parameters:

- `beta` - Newmark beta parameter.
- `gamma` - Newmark gamma parameter.
- `ts` - $TimeStep + 1$-by-1 matrix of integration time points.
- `fDoF` - list of IDs of free degrees of freedom.
- `pDoF` - list of IDs of prescribed (fixed) degrees of freedom.
- `M` - ($\#Nodes \times \#Eq$)-by-($\#Nodes \times \#Eq$) mass matrix or a function handle returning such matrix. The input of the function is `t_, u_, v_` -

time and $\#Nodes \times \#Eq$-by-1 displacement and velocity vectors respectively.

- `C` - ($\#Nodes \times \#Eq$)-by-($\#Nodes \times \#Eq$) damping matrix or a function handle returning such matrix. The input of the function is `t_, u_, v_` -

time and $\#Nodes \times \#Eq$-by-1 displacement and velocity vectors respectively.

- `K` - ($\#Nodes \times \#Eq$)-by-($\#Nodes \times \#Eq$) stiffness matrix or a function handle returning such matrix. The input of the function is `t_, u_, v_` -

time and $\#Nodes \times \#Eq$-by-1 displacement and velocity vectors respectively.

- `F` - ($\#Nodes \times \#Eq$)-by-1 external force vector or a function handle returning such vector. The input of the function is `t_, u_, v_` -

time and $\#Nodes \times \#Eq$-by-1 displacement and velocity vectors respectively.

- `u0` - ($\#Nodes \times \#Eq$)-by-1 vector of initial displacements.
- `v0` - ($\#Nodes \times \#Eq$)-by-1 vector of initial velocities or an empty array.
- `a0` - ($\#Nodes \times \#Eq$)-by-1 vector of initial accelerations or an empty array.
- `up` - ($\#Nodes \times \#Eq$)-by-1 vector of prescribed displacements or a function handle returning such vector, or an empty array. The input of the function handle is t_ (time).

- `vp` - ($\#Nodes \times \#Eq$)-by-1 vector of prescribed velocities or a function handle returning such vector, or an empty array. The input of the function handle is t_ (time).
- `ap` - ($\#Nodes \times \#Eq$)-by-1 vector of prescribed accelerations or a function handle returning such vector, or an empty array. The input of the function handle is t_ (time).
- Options in the format `...,'Name',value,...`
  - `'iteration', outputFunction` - function called at each iteration (of Newton's iterative method) and supplied with `([i j],ts(i),u,v,a,fDoF,pDoF,Mval,Cval,Kval,Fval,u0,v0,a0,upval,vpval,apval)` as input. The output must be one or several matrices. The outputs are collected and returned at the end of the procedure.
  - `'increment', outputFunction` - function called at each time increment and supplied with `(i,ts(i),u,v,a,fDoF,pDo` as input. The output must be one or several matrices. The outputs are collected and returned at the end of the procedure. The outputs are not collected if `outputFunction` at iteration is specified. However increment outputFunction are called and can be used for plotting etc.
  - 'maxiter', `maxIter` - maximum number of Newton's iterations at each time step. If this number is reached, Newton method aborts and an error message is thrown. Default value - 10.
  - `'reltol', relTol` - relative tolerance. Iterations stop if the norm of the residual is within this tolerance relatively to the total force vector.
  - `'abstol', absTol` - absolute tolerance. Iterations stop if the norm of the residual is less or equal to absTol.

Output parameters:

- If no `'iteration' or 'increment'` option specified, then:
  - `u` -($\#Nodes \times \#Eq$)-by-1 vector of displacements at the final time point.
  - `v` - ($\#Nodes \times \#Eq$)-by-1 vector of velocities at the final time point.
  - `a` -($\#Nodes \times \#Eq$))-by-1 vector of accelerations at the final time point.
- If `'iteration' or 'increment'` option specified, then:
  - `argsOut` - a cell array containing the outputs of outputFunction collected at each time step or Newton iteration. Each element `argsOut{k}` contains NI-by-$n1(k)$-by-...-$nq(k)$ matrix, where $n1(k)$-by-...-$nq(k)$) is the size of k-th output of `outputFunction` and NI is the total number of iterations or increments.

By default, the Dirichlet (essential) boundary conditions are given by prescribed displacements. In order to allow for different boundary conditions definition, `pDoF` can be replaced by `{pDoFu, pDoFv, pDoFa}` - lists of nodes at which displacements, velocities and accelerations are prescribed respectively. The corresponding entries of `up, vp,` and `ap` are used directly and via finite difference approximations, which normally match the approximation used in Newmark method.

The mass matrix must be non-singular (e.g., a lumped diagonal matrix)

Empty array `[]` can be passed as a value for input parameters `C, up, vp, ap, v0` and `a0`. In this case the arguments are replaced by zero arrays of appropriate size.

If both `'iteration'` and `'increment'` options are specified, then outputs of the former are not collected, yet the corresponding `outputFunction` is called at the end of each iteration and can be used e.g., for plotting.

Output collection at each iteration is inefficient as the target array changes its size every time.

3.4 Recovery of the results

3.4.1 recoveryEvaluateField

Interpolate values at points given by natural coordinates from nodal values.

Syntax:

`u2_i=recoveryEvaluateField(u2,IEN,elementType,el,ptNatCoords)`

Input parameters:

- `u2` - $\#Nodes$-by-$\#Eq$ matrix containing nodal field values (e.g., displacements);
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `elementType` - a single string defining the type of elements in the mesh.
- `el` - k-by-1 matrix, element IDs of the query points.
- `ptNatCoords` - k-by-$\#ElDim$ matrix, natural coordinates of the query points.

Output parameters:

- `ptCoords` - k-by-$\#Dim$ matris, row `ptCoords(i,:)` contains global coordinates of the query point i.

Since isoparametric elements are used, the procedure is identical to `meshGetGlobal()`.

3.4.2 recoveryEvaluateGradients

Interpolate gradients from nodal field values at points given by natural coordinates.
Syntax:
`gradu2_i=recoveryEvaluateGradients(u2,nodeCoords,IEN,elementType,el,ptNatCoords)`
Input parameters:

- `u2` - $\#Nodes$-by-$\#Eq$ matrix containing nodal field values (e.g., displacements);
- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates.
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `elementType` - a single string defining the type of elements in the mesh.
- `el` - k-by-1 matrix, element IDs of the query points.
- `ptNatCoords` - k-by-$\#ElDim$ matrix, natural coordinates of the query points.

Output parameters:

- `gradu2_i` - $\#Eq \times \#Dim$-by-k matrix containing values of gradients, `gradu2_i((i-1)*#Dim+j,m)` equals to $\partial u_i/\partial x_j$ evaluated at m-th query point.

Gradients are vectorised as in equation (25).

3.4.3 recoveryEvaluateStrains

Interpolate strains from nodal field values at points given by natural coordinates.
Syntax:
`e_i=recoveryEvaluateStrains(u2,nodeCoords,IEN,elementType,el,ptNatCoords)`
Input parameters:

- `u2` - $\#Nodes$-by-$\#Eq$ matrix containing nodal field values (e.g., displacements);
- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates.
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `elementType` - a single string defining the type of elements in the mesh.
- `el` - k-by-1 matrix, element IDs of the query points.
- `ptNatCoords` - k-by-$\#ElDim$ matrix, natural coordinates of the query points.

Output parameters:

- `e_i` - 3-by-k or 6-by-k matrix containing values of strains, column `e_i(:,m)` corresponds to strains evaluated at m-th query point

Strains are vectorised as in equation (26).

3.4.4 recoveryGP

Recover gradients at Gauss points by interpolation of nodal field values.
Syntax:
`[gradu,GPCoords]=recoveryGP(u2,nodeCoords,IEN,elementType,numGP)`
`[gradu,GPCoords]=recoveryGP(u2,nodeCoords,IEN,elementType)`
`[gradu,GPCoords,GPNatCoords,el] = recoveryGP(...)`
Input parameters:

- `u2` - $\#Nodes$-by-$\#Eq$ matrix containing nodal field values (e.g., displacements);
- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates.
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `elementType` - a single string defining the type of elements in the mesh.
- `numGP` (optional) - $\#GP$, number of Gauss points to evaluate the gradient at. Default value - 1.

Output parameters:

- `gradu2_GP` - $\#Eq \times \#Dim$-by-$\#El \times \#GP$ matrix containing values of gradients recovered using the procedure, so that `gradu2_GP((i-1)*#Dim+j,l)` equals to $\partial u_i/\partial x_j$ evaluated at l-th Gauss point of m-th element.
- `GPCoords` - $\#El\times\#GP$-by-$\#Dim$ matrix containing coordinates of Gauss points, so that `GPCoords((m-1)*#GP+l,:)` are the coordinates of l-th Gauss point of m-th element.
- `GPNatCoords` (optional) - $\#El\times\#GP$-by-$\#ElDim$ matrix containing natural coordinates of Gauss points, so that `GPNatCoords((m-1)*#GP+l,:)` are the coordinates of l-th Gauss point of m-th element.
- `el` (optional) - $\#El\times\#GP$-by-1 matrix containing element IDs of corresponding Gauss points, i.e. first $\#GP$ rows contain 1, followed by $\#GP$ rows containing 2 and so on.

Gradients are vectorised as in equation (25).

3.4.5 recoveryGPEng

Recover strains at Gauss points by interpolation of displacements.
Syntax:
`[e_GP,GPCoords]=recoveryGPEng(u2,nodeCoords,IEN,elementType,numGP)`
`[e_GP,GPCoords]=recoveryGPEng(u2,nodeCoords,IEN,elementType)`
`[e_GP,GPCoords,GPNatCoords,el] = recoveryGPEng(...)`
Input parameters:

- `u2` - $\#Nodes$-by-$\#Eq$ matrix containing nodal field values (e.g., displacements);
- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates.
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `elementType` - a single string defining the type of elements in the mesh.
- `numGP` (optional) - $\#GP$, number of Gauss points to evaluate the gradient at. Default value - 1.

Output parameters:

- `e_GP` - 3-by-$\#El\times\#GP$ or 6-by-$\#El\times\#GP$ matrix containing values of strains recovered using the procedure, so that `e_GP(i,l)` equals to i-th evaluated at l-th Gauss point of m-th element.
- `GPCoords` - $\#El\times\#GP$-by-$\#Dim$ matrix containing coordinates of Gauss points, so that `GPCoords((m-1)*#GP+l,:)` are the coordinates of l-th Gauss point of m-th element.
- `GPNatCoords` - $\#El\times\#GP$-by-$\#ElDim$ matrix containing natural coordinates of Gauss points, so that `GP~NatCoords((m-1)*#GP+l,:)` are the coordinates of l-th Gauss point of m-th element.
- `el` $\#El\times\#GP$-by-1 matrix containing element IDs of corresponding Gauss points, i.e. first $\#GP$ rows contain 1, followed by $\#GP$ rows containing 2 and so on.

Strains are vectorised as in equation (26).

3.4.6 recoveryAvg

Recover gradients at nodes by averaging values obtained by interpolation within adjacent elements.
Syntax:
`gradu_n=recoveryAvg(u2,nodeCoords,IEN,elementType)`
Input parameters:

- `u2` - $\#Nodes$-by-$\#Eq$ matrix containing nodal displacements.
- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates.
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `elementType` - a single string defining the type of elements in the mesh.

Output parameters:

- `gradu_n` - $\#Eq \times \#Dim$-by-$\#Nodes$ matrix containing nodal values of gradients recovered using the procedure, so that `gradu_n((i-1)*#Dim+j,m)` equals to $\partial u_i/\partial x_j$ evaluated at m-th node.

This function calls `elementData.m` in order to determine the natural nodal coordinates for the given element type. Gradients are vectorised as in equation (25).

3.4.7 recoveryAvgEng

Recover strains at nodes by averaging values obtained by interpolation within adjacent elements.
Syntax:
`e_n=recoveryAvg(u2,nodeCoords,IEN,elementType)`
Input parameters:

- `u2` - $\#Nodes$-by-$\#Eq$ matrix containing nodal displacements.
- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates.
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `elementType` - a single string defining the type of elements in the mesh.

Output parameters:

- `e_n` - 3-by-$\#Nodes$ or 6-by-$\#Nodes$ matrix containing nodal values of strains recovered using the procedure, so that column `e_n(:,i)` equals to strains evaluated at i-th node.

This function calls `elementData.m` in order to determine the natural nodal coordinates for the given element type. Strains are vectorised as in equation (26).

3.4.8 recoveryPatch

Recover gradients at nodes from given nodal field values using superconvergent patch recovery (SPR) technique.
Syntax:
`grad_n=recoveryPatch(u2,nodeCoords,IEN,elementType,BIEN,INE,degrees)`
`grad_n=recoveryPatch(u2,nodeCoords,IEN,elementType,BIEN)`
Input parameters:

- `u2` - $\#Nodes$-by-$\#Eq$ matrix containing nodal displacements.
- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates.
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `elementType` - a single string defining the type of elements in the mesh.
- `BIEN` - a $\#BR$-by-1 cell array containing boundary incidence matrices.

- `INE` (optional) - $\#Nodes$-by-$\#EpN$ array, node-element incidence matrix, row `INE(i,:)` corresponds to ith node and contains IDs of elements that share it.
- `degrees` (optional) - $\#Nodes$-by-1 matrix containing degrees/valencies of nodes, `degrees(i)` is the number of elements in the mesh node i is incident to.

Output parameters:

- `gradu_n` - $\#Eq \times \#Dim$-by-$\#Nodes$ matrix containing nodal values of gradients recovered using the procedure, so that `gradu_n((i-1)*#Dim+j,m)` equals to $\partial u_i/\partial x_j$ evaluated at m-th node.

Gradients are vectorised as in equation (25). INE and degrees should be provided together. If not provided, they are computed using `IENtoINE`. This procedure invokes `recoveryGP()` and passes its output to `recoveryPatchGP()`, which does the rest of the job. function calls `elementData.m`

3.4.9 recoveryPatchGP

Recover gradients at nodes from gradient values given at Gauss points using least square fitting (superconvergent patch recovery, SPR)

Input parameters:

- `gradu2_GP` - $\#Eq \times \#Dim$-by-$\#El \times \#GP$ matrix containing values of gradients recovered using the procedure, so that `gradu2_GP((i-1)*#Dim+j,l)` equals to $\partial u_i/\partial x_j$ evaluated at l-th Gauss point of m-th element.
- `GPCoords` - $\#El \times \#GP$-by-$\#Dim$ matrix containing coordinates of Gauss points, so that `GPCoords((m-1)*#GP+l,:)` are the coordinates of l-th Gauss point of m-th element.
- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates.
- `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
- `elementType` - a single string defining the type of elements in the mesh.
- `BIEN` - a $\#BR$-by-1 cell array containing boundary incidence matrices.
- `INE` (optional) - $\#Nodes$-by-$\#EpN$ array, node-element incidence matrix, row `INE(i,:)` corresponds to ith node and contains IDs of elements that share it.
- `degrees` (optional) - $\#Nodes$-by-1 matrix containing degrees/valencies of nodes, `degrees(i)` is the number of elements in the mesh node i is incident to.

Output parameters:

- `gradu_n` - $\#Eq \times \#Dim$-by-$\#Nodes$ matrix containing nodal values of gradients recovered using the procedure, so that `gradu_n((i-1)*#Dim+j,m)` equals to $\partial u_i/\partial x_j$ evaluated at m-th node.

Gradients are vectorised as in equation (25). INE and degrees should be provided together. If not provided, they are computed using `IENtoINE`. This function calls `elementData.m` in order to determine natural nodal coordinates, optimal number of Gauss points, the polynomial that can be fit exactly by the element shape functions and topology of element boundary.

3.5 Visualisation

3.5.1 drawNodes

Draw mesh nodes using MATLAB `scatter3()` function.

Syntax:

```
drawNodes(nodeCoords)
drawNodes(nodeCoords,specs)
drawNodes(nodeCoords,'o')
drawNodes(nodeCoords,IEN)
drawNodes(nodeCoords,IEN,specs)
```

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates.
- `IEN or {IEN,boundaryElementsIDs} or BIEN` (optional), where
  - `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
  - `boundaryElementIDs` - $\#BR$-by-1 cell array. Cell `boundaryElementIDs{p}` is a l_p-by-1 matrix listing IDs of elements at boundary p.
  - `BIEN` - a $\#BR$-by-1 cell array containing boundary incidence matrices.
- `specs` (optional) - string or cell array of parameters, first of which is a string. The parameters are passed to `scatter3()` directly. Default value - `{'k','filled'}`

This function uses Matlab's `scatter3()` object. Only nodes of elements listed in `IEN` (or in `boundaryElementsIDs`, when applicable) are shown.

3.5.2 drawElements

Draw the mesh using MATLAB `patch()` function. Values may be specified for each element to define the colour. See also `drawElementsInterp` function.

Syntax:

```
drawElements(nodeCoords,IEN,elementType)
drawElements(nodeCoords,IEN,elementType, vals)
drawElements(nodeCoords,IEN,elementType, vals, alphas)
drawElements(nodeCoords,IEN,elementType, 0, alphas)
drawElements(nodeCoords,IEN,elementType,0,0) %plots transparent elements
```

Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates.
- `IEN or {IEN,boundaryElementsIDs} or BIEN` (optional), where
  - `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
  - `boundaryElementIDs` - $\#BR$-by-1 cell array. Cell `boundaryElementIDs{p}` is a l_p-by-1 matrix listing IDs of elements at boundary p. Providing both `IEN` and `boundaryElementsIDs` is equivalent to providing `IEN(unique([boundaryElementIDs{:}]),:)`, but element labels will differ if drawn.
  - `BIEN` - a $\#BR$-by-1 cell array containing boundary incidence matrices.
- `elementType` - a single string specifying the type of elements used;
- `vals` (optional) - a scalar or $\#El$-by-1 vector containing scalar values to display as element colours;
- `alphas` (optional) - a scalar or $\#El$-by-1 vector containing scalar which are interpreted as element opacity;
- `options` (optional) - a string `'Labels', 'elementLabels', 'nodeLabels'` or `'nodeLocalLabels'`. If specified, corresponding IDs are displayed.

Uses MATLAB's `patch()` object. The edges of elements are piece-wise linear and do not represent true geometry of higher-order elements. Only elements listed in `IEN` (or in `boundaryElementsIDs`, when applicable) are shown. When `BIEN` is passed as the second argument, `elementType` should correspond to the boundary element type. `{IEN,boundaryElementsIDs}` and `BIEN` are told apart based on the fact that `BIEN` is a column of cells.

3.5.3 drawElementsInterp

Draw the mesh using MATLAB `patch()` function. Values may be specified for each node to define the colour. See also `drawElements` function.

Syntax:

```
drawElements(nodeCoords,IEN,elementType)
drawElements(nodeCoords,IEN,elementType, vals)
drawElements(nodeCoords,IEN,elementType, vals, alphas)
drawElements(nodeCoords,IEN,elementType, 0, alphas)
drawElements(nodeCoords,IEN,elementType,0,0) %plots transparent elements
```

Input parameters:

- `nodeCoords` - $\#Nodes$-by-$\#Dim$ array of node coordinates.
- `IEN or {IEN,boundaryElementsIDs} or BIEN` (optional), where
  - `IEN` - $\#El$-by-$\#NpE$ array, element-node incidence matrix.
  - `boundaryElementIDs` - $\#BR$-by-1 cell array. Cell `boundaryElementIDs{p}` is a l_p-by-1 matrix listing IDs of elements at boundary p. Providing both `IEN` and `boundaryElementsIDs` is equivalent to providing `IEN(unique([boundaryElementIDs{:}]),:)`, but element labels will differ if drawn.
  - `BIEN` - a $\#BR$-by-1 cell array containing boundary incidence matrices.
- `elementType` - a single string specifying the type of elements used;
- `vals` (optional) - a scalar or $\#Nodes$-by-1 vector containing scalar values to display as element colours;
- `alphas` (optional) - a scalar or $\#Nodes$-by-1 vector containing scalar which are interpreted as element opacity;
- `options` (optional) - a string `'Labels'`, `'elementLabels'`, `'nodeLabels'` or `'nodeLocalLabels'`. If specified, corresponding IDs are displayed.

Uses MATLAB's `patch()` object. The edges of elements are piece-wise linear and do not represent true geometry of higher-order elements. Only elements listed in `IEN` (or in `boundaryElementsIDs`, when applicable) are shown. When `BIEN` is passed as the second argument, `elementType` should correspond to the boundary element type. `{IEN,boundaryElementsIDs}` and `BIEN` are told apart based on the fact that `BIEN` is a column of cells.

3.6 Element-specific functions

3.6.1 elementData

Specific data for a requested element type.

Syntax:

```
elData=elementData(elementType)
numGP=elementData(elementType,'numGPFull')
inElement=elementData('2dQ1','inElement')
inElement([1 1]) % ans=1
elementData('2dQ1','inElement',[2 0]) % ans=0
```

Input parameters:

- `elementType` - a single string specifying the type of elements used;
- `fieldName` (optional) - a single string specifying a requested property of the structure
- `argument` (optional) - if the requested field is a function handle, the function is applied to the argument and the result is returned.

Output parameters:

- `elData` - `elementData` structure, if no `fieldName` specified. Otherwise, the value of the requested field.

3.6.2 gaussPoints

Coordinates and weights of Gauss points for the specified element type and number of points

Syntax:

```
[GPNatCoords, GPweights]=gaussPoints('2dQ1',4)
```

Input parameters:

- `elementType` - a string specifying the type of element.
- `numGP` - single scalar specifying the requested number of Gauss points.

Output parameters:

- `GPNatCoords` - $\#GP$-by-$\#ElDim$ matrix, row `GPNatCoords(k,:)` contains the coordinates of k-th Gauss point.
- `GPweights` - $\#GP$-by-1 matrix, `weights(k,1)` is the k-th Gauss weight.

elementData structure FIELDS:

- `'nodeNatCoords'` - $\#Nodes$-by-$\#ElDim$matrix, natural nodal coordinates.
- `'numGPFull'` - number of Gauss points for exact integration in FE space.
- `'numGPSPR'` - number of Gauss points for SPR (superconvergent patch recovery).
- `'bndElementType'` - string, type of boundary element.
- `'elLocBIEN'` - $\#bndEl$-by$-\#bndNpE$ matrix, $elLocBIEN(k,:)$ lists IDs of nodes incident to k-th boundary element.
- `'polTerms'` - $\#p$-to-$\#ElDim$ ($\#ElDim$-dimensional square array with $\#p$ elements in each dimension) - array of 1s and 0s representing polynomial space spanned by shape functions
- `'p0'` - function handle, polynomial basis for FE space, takes natural coordinates `x_` (k-by-$\#ElDim$) as input and returns functions' values `p0_` (k-by-$\#Nodes$) as output.
- `'vertexLocalIDs'` - 1-by-$l1$ matrix, lists IDs of vertex nodes (as opposed to midpoints)
- `'facetsLocBIEN'` - $\#Faces$-by-$l2$ matrix, `facetsLocBIEN(k,:)` lists IDs of nodes incident to k-th facet. Used for element visualization.
- `'inElement'` - function handle, shows if a point belongs to the element natural domain. Takes natural coordinates `x_` (k-by-$\#ElDim$) as input and returns k-by-1 matrix of 1s (true) and 0s (false).
- `'inElementTol'` - function handle, shows if a point belongs to the element natural domain within certain tolerance. Takes natural coordinates `x_` (k-by-$\#ElDim$) as input and returns k-by-1 matrix of 1s (true) and 0s (false).

Table 5: `elementData` structure fields.

3.6.3 shapeFunctions<elementType>

Shape functions for the specified element type.

Syntax:

```
[shapeFunctions, natDerivatives]=shapeFunctions1dQ1(r)
[shapeFunctions, natDerivatives]=shapeFunctions2dQ2r(r)
[shapeFunctions, natDerivatives]=shapeFunctions2dQ2r(r(:,1),r(:,2))
```

Input parameters:

- `r` - k-by-$\#ElDim$ matrix, row `r(k,:)` contains natural coordinates of k-th point.

Output parameters:

- `shapeFunctions` - k-by-$\#Nodes$ matrix, `shapeFunctions(l,i)` is the value of i-th shape function at point l.
- `natDerivatives` - $\#ElDim \times k$-by-$\#Nodes$ matrix, `natDerivatives(k*(j-1)+l,i)` is the derivative of i-th shape function with respect to j-th coordinate evaluated at point l.

The shape functions for different elements are implemented in different m-files, which are named shapeFunctions<elementType>, e.g., `shapeFunctions1dQ2.m` for `'1dQ2'` element. The functions can accept points' coordinates as a single argument (e.g., `shapeFunctions1dQ2(r)`) or as separate arguments for each coordinate (e.g., `shapeFunctions1dQ2(r1,r2)`).

3.7 Constitutive functions

3.7.1 elasticProperties

```
prop = elasticProperties(parName1, parVal1, parName2, parVal2)
```

Computes various representations for properties of a linearly elastic material defined in terms of any two parameters.

Syntax:

```
prop = elasticProperties('lambda',8e7,'mu',8e7);
prop = elasticProperties('youngsModulus',2e8,'poissonsRatio',0.25);
prop.CFull % 9-by-9 matrix
elasticProperties('lambda',8e7,'mu',8e7,'CPlaneStressFull') % 4-by-4 matrix
```

Input parameters:

- `parName1, parVal1, parName2, parVal2` - where
  - `parName1`, `parName2` - strings, parameter names (see table below).
  - `parVal1`, `parVal2` - real scalars, parameter values.
- `fieldName` (optional) - a single string specifying a requested property of the structure.

Output parameters:

- `prop` - elasticProperties structure, if no `fieldName` specified. Otherwise, the value of the requested field.

The fields of the structure can be accessed via dot notation `prop.<fieldName>`, where `<fieldName>` is a placeholder defined in the following table.

<fieldName>	Value
`lambda` (can be used for input)	scalar, Lame's first parameter λ
`mu` (can be used for input)	scalar, Lame's second parameter, shear modulus μ
`youngsModulus` (can be used for input; alias: `E`)	scalar, Young's modulus E.
`poissonsRatio` (can be used for input; alias: `nu`)	scalar, Poisson's ratio ν
`CPlaneStrainFull`	4-by-4 matrix, full form of the plane strain elasticity tensor
`CPlaneStrainEng`	3-by-3 matrix, short form of the plane strain elasticity tensor
`CPlaneStressFull`	4-by-4 matrix, full form of the plane stress elasticity tensor
`CPlaneStressEng`	3-by-3 matrix, full form of the plane stress elasticity tensor
`CFull`	full form 3D 9-by-9 elasticity tensor
`CShort`	short form 3D 6-by-6 elasticity tensor

Table 6: `elasticProperties` fields.

3.8 Miscellaneous functions

3.8.1 clonefig

Clone a figure by creating an exact copy of it

Input parameters:

- `i` (optional) - graphic object identifier. See `copyobj` reference for more details.

Syntax:

```
clonefig % clones current figure;
clonefig(i) % clones figure i;
clonefig(0) % clones all figures;
```

3.8.2 snapfig

Take a snapshot of the current figure and save it in a .png file.

Syntax:

```
snapfig; % snapshot1.png created
snapfig; % snapshot2.png created and so on
```

3.8.3 tensorRotate

Rotate a 2D second-order tensor `A` by angle `alpha`.

Syntax:

```
tensorRotate([1 0 0 0],pi/4); % ans=[.5 .5 .5 .5]
tensorRotate([.5 .5 1],-pi/4,'engineering'); % ans=[1 0 0]
```

Input parameters:

- `A` - k-by-4 matrix, k-th row represents a 2d tensor asA_{11}, A_{21}, A_{12}, A_{22};
  or k-by-3 matrix, k-th row represents a 2d strain tensor as ε_{11}, ε_{22}, γ_{12} if `form='engineering'`, or as ε_{11}, ε_{22}, ε_{12}, if `form='tensorial'`
- `alpha` - a scalar or k-by-m matrix or 1-by-m matrix of rotation angles.
- `form` (when `A` is k-by-3) - a string `'engineering'` or `'tensorial'`, which specifies how shear components are represented.

Output parameters:

- `B` - k-by-$l \times m$ matrix, so that each k-by-l block corresponds to a k-by-l matrix `A` rotated respectively by m-th column of angles.

3.8.4 tensorInvariant

Compute an invariant of a second-order tensor.
Syntax:
`I1=tensorInvariant(A,str,'I1')`
Input parameters:

- A - k-by-4 matrix, k-th row represents a 2d tensor as A_{11}, A_{21}, A_{12}, A_{22};
  or k-by-9 matrix, k-th row represents a 3d tensor as A_{11}, A_{21}, A_{31}, A_{12}, A_{22}, A_{32}, A_{13}, A_{23}, A_{33};
  or k-by-3 matrix, k-th row represents a symmetric 2d tensor as A_{11}, A_{22}, A_{12};
  or k-by-6 matrix, k-th row represents a symmetric 3d tensor as A_{11}, A_{22}, A_{33}, A_{23}, A_{13}, A_{12}.
- `str` - a string identifier for the requested quantity: `'I1'`, `'I2'`, `'I3'`,`'J1'`, `'J2'`, `'J3'`, `'mises'`, `'tresca'`, `'eigen'`.

Output parameters:

- `invar` - k-by-1 matrix, k-th element is the requested invariant for the k-th input tensor;
  or k-by-2 or k-by-3 matrix, k-th row contains eigenvalues for the k-th input tensor.

Von Mises stress for 2D is consistent with 3D plane stress setting. On 'eigen', eigenvalues are returned. Exact expressions for invariants are provided below.

$$
\begin{aligned}
I_1(\mathbf{A}) &= \mathrm{tr}\mathbf{A} = A_{11} + A_{22} + A_{33}, && (38)\\
I_2(\mathbf{A}) &= \frac{1}{2}\left(\mathrm{tr}\mathbf{A}\right)^2 - \frac{1}{2}\mathrm{tr}\mathbf{A}^2, && (39)\\
&= A_{11}A_{22} + A_{22}A_{33} + A_{33}A_{11} - A_{12}A_{21} - A_{13}A_{31} - A_{32}A_{23}, && (40)\\
I_3(\mathbf{A}) &= \det\mathbf{A} = A_{11}A_{22}A_{33} + A_{12}A_{23}A_{31} + A_{13}A_{21}A_{32} && (41)\\
&-A_{13}A_{22}A_{31} - A_{12}A_{21}A_{33} - A_{11}A_{23}A_{32}. && (42)\\
J_2(\mathbf{A}) &= I_2(\mathbf{A} - \frac{1}{3}\mathrm{tr}\mathbf{A}) = \frac{1}{6}\left(\left(A_{11}-A_{22}\right)^2 + \left(A_{22}-A_{33}\right)^2 + \left(A_{33}-A_{11}\right)^2\right) && (43)\\
&+A_{23}A_{32} + A_{13}A_{31} + A_{21}A_{12}. && (44)\\
J_3(\mathbf{A}) &= I_3(\mathbf{A} - \frac{1}{3}\mathrm{tr}\mathbf{A}) = \frac{2}{27}\left(A_{11}^3 + A_{22}^3 + A_{33}^3\right) + A_{12}A_{23}A_{31} + A_{13}A_{32}A_{21} && (45)\\
&+\frac{1}{3}A_{11}\left(A_{12}A_{21} + A_{13}A_{31}\right) + \frac{1}{3}A_{22}\left(A_{12}A_{21} + A_{23}A_{32}\right) + \frac{1}{3}A_{33}\left(A_{23}A_{32} + A_{13}A_{31}\right) && (46)\\
&-\frac{2}{3}\left(A_{13}A_{22}A_{31} + A_{12}A_{21}A_{33} + A_{11}A_{23}A_{32}\right) + \frac{4}{9}A_{11}A_{22}A_{33} && (47)\\
&-\frac{1}{3}\left(A_{11}^2A_{22} + A_{22}^2A_{11} + A_{22}^2A_{33} + A_{33}^2A_{22} + A_{33}^2A_{11} + A_{11}^2A_{33}\right). && (48)
\end{aligned}
$$

$$
\begin{aligned}
\sigma_{\text{tresca}} &= \max\left(\left|\sigma_1-\sigma_2\right|, \left|\sigma_2-\sigma_3\right|, \left|\sigma_3-\sigma_1\right|\right), && (49)\\
\sigma_{\text{mises}} &= \left(\frac{1}{2}\left(\sigma_1-\sigma_2\right)^2 + \frac{1}{2}\left(\sigma_2-\sigma_3\right)^2 + \frac{1}{2}\left(\sigma_3-\sigma_1\right)^2\right)^{\frac{1}{2}} && (50)\\
&= \sqrt{\frac{1}{2}\left(\left(\sigma_{11}-\sigma_{22}\right)^2 + \left(\sigma_{22}-\sigma_{33}\right)^2 + \left(\sigma_{33}-\sigma_{11}\right)^2\right) + 3\left(\sigma_{12}^2 + \sigma_{23}^2 + \sigma_{13}^2\right)} && (51)\\
&= \sqrt{3J_2(\boldsymbol{\sigma})}, && (52)
\end{aligned}
$$

$$
\begin{aligned}
I_1(\mathbf{A}) &= \mathrm{tr}\mathbf{A} = \mathrm{A}_{11} + \mathrm{A}_{22}, && (53)\\
I_2(\mathbf{A}) &= \det \mathbf{A} = \mathrm{A}_{11}\mathrm{A}_{22} - \mathrm{A}_{12}\mathrm{A}_{21}, && (54)\\
J_2(\mathbf{A}) &= I_2(\mathbf{A} - \frac{1}{3}\mathrm{tr}\mathbf{A}) && (55)\\
&= \frac{1}{4}\left(\sigma_{11} - \sigma_{22}\right)^2 + \sigma_{12}^2, && (56)\\
\sigma_{\mathrm{tresca}} &= |\sigma_1 - \sigma_2| = \sqrt{\frac{1}{4}(\sigma_{11} - \sigma_{22})^2 + \sigma_{12}^2}, && (57)\\
\sigma_{\mathrm{mises}} &= \left(\sigma_{11}^2 - \sigma_{11}\sigma_{22} + \sigma_{22}^2 + 3\sigma_{12}^2\right)^{\frac{1}{2}}. && (58)
\end{aligned}
$$

3.8.5 node2DoFs

Convert IDs of nodes to the corresponding IDs of degrees of freedom (the two are identical only in the case of scalar problem, i.e. when $\#Eq = 1$). This function implements ordering convention used throughout this package and in particular in matrix and vector assembly.

Syntax:

```
DoF_IDs =node2DoFs(nodeIDs, numEq, eqIndex)
DoF_IDs = node2DoFs(nodeIDs, numEq)
```

Input parameters:

- `nodeIDs` - k-by-1 vector or k-by-m array;
- `numEq` - a single integer;
- `eqIndex` (optional) - 1-by-m vector or k-by-m array containing non-negative integers - equation numbers, i.e. IDs of local degrees of freedom. Zeros indicate that the corresponding entry must not be included in the output. If k-by-m array then m or k may equal 1 as long as condition `size(eqIndex)==size(nodeIDs)` holds.

Output parameters:

- `DoF_IDs` - $k \times m$-by-1 array. `DoF_IDs` =`(nodeIDs-1)*numEq + eqIndex`; or n-by-1 array ($n < k \times m$) if some elements of `eqIndex` are zero.

3.9 Verification procedures

The package includes several files, which serve the purpose to verify the solutions provided by the package. These include comparison with analytical results (`example4_1.m`, `example4_2.m`, `example5_1.m`, `example5_2.m`,) and with the solution provided by Abaqus (`chkAbaqusBeamStatic.m`, `chkAbaqusBeamOsc.m`). We omit detailed documentation of verification procedures here and refer the reader to commentaries in the following files. We note that the following functions are used to import the data from Abaqus and can be potentially reusable: `im~port_displacements.m`, `importMTX.m`, `parseAbaqRecs_1.m`, `parseAbaqRecs_2.m`, `readAbaqFil.m`.

Script `chkAll.m` calls verification procedures and every example included in the package. This script must run without errors (provided PDE Toolbox and Abaqus are installed), while warning messages are expected.

4 Examples

All examples are located in the package folder and named `example0.m`, `example1.m` and so on. The minimal example, `example0.m`, is discussed in the end of the Section 2.

4.1 Example 1. Steady state heat diffusion equation on a rectangular domain

In this example we solve a steady state heat equation

$$\nabla \cdot \nabla\theta + f = 0, \quad (x, y) \in \Omega = [0, 2] \times [0, 2], \tag{59}$$

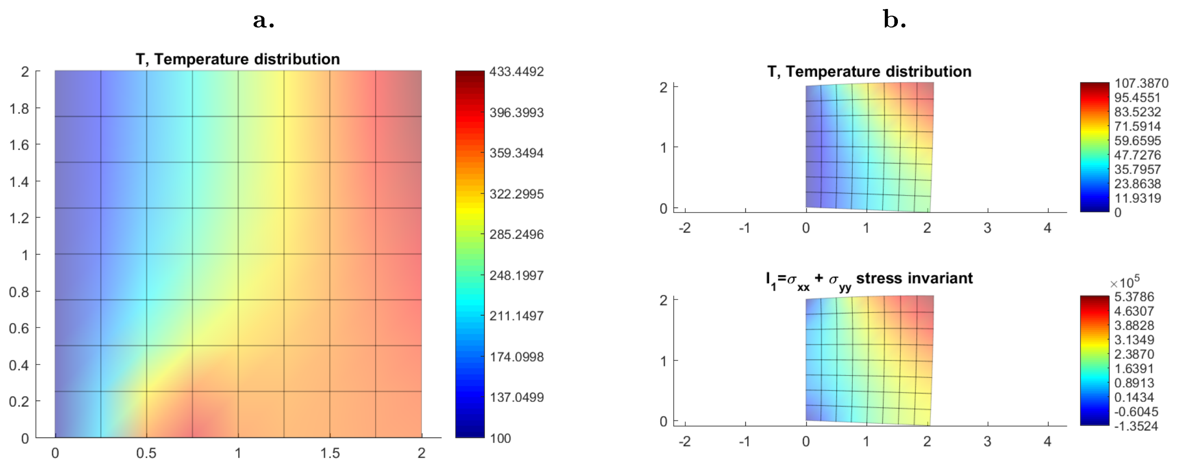


Figure 4: **a**. Example 1. Equilibrium temperature distribution. **b**. Example 2. Equilibrium temperature distribution and mechanical equilibrium on the deformed mesh.

which is a special scalar case of equation (1) with $\mathbf{u} = \theta$ and $\mathbf{f} = f$, $\mathbf{c} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and all other tensors equal zero. Full code can be found in `example1.m`, and here we only provide the excerpt showing the specification of boundary conditions and bulk heat source, which were chosen as follows,

$$f = \begin{cases} 5 \cdot 10^3, & 0.5 \leq x \leq 0.7,\ 0 \leq y \leq 0.2, \\ 0, & \text{otherwise.} \end{cases} \tag{60}$$

$$\begin{aligned} \theta = 0, &\quad \text{on } \Omega_4, \\ \mathbf{n} \cdot \nabla\theta = 0, &\quad \text{on } \Omega_3,\ \Omega_1, \\ \mathbf{n} \cdot \nabla\theta = 100y, &\quad \text{on } \Omega_3. \end{aligned}$$

Boundary temperature, boundary heat flux and distributed heat source are each given by a single function handle, as shown below. Lines 2, 5, and 8 rearrange the input coordinates, separating x and y components.

Listing 6: Example 1 - BC and heat source specification

```
1 bodyForce2d=@(x,y)([5e3*((.5<=x)&(x<=.7)&(y<=.2))]'); %distributed heat source
2 bodyForce2d=@(x)(bodyForce2d(x(:,1),x(:,2)));
3
4 boundaryTraction2d=@(x,y)([(100*y)*(x>=2-1e-6)]'); %boundary flux
5 boundaryTraction2d=@(x)(boundaryTraction2d(x(:,1),x(:,2)));
6
7 boundaryDisplacement2d=@(x,y)[100*(x<=1e-6)]'; %boundary temperature
8 boundaryDisplacement2d=@(x)(boundaryDisplacement2d(x(:,1),x(:,2)));
9 isDirichlet=[0;0;0;1];
```

4.2 Example 2. Steady state mechanical equilibrium and steady state heat equation

Here we solve a simultaneous problem of steady state heat equation and Cauchy's equation of equilibrium (or conservation of linear momentum) with thermal expansion taken into account. We assume additive decomposition of the total strain into elastic and thermal part,

$$\boldsymbol{\nabla}\mathbf{u} = \boldsymbol{\varepsilon}^{\text{tot}} = \boldsymbol{\varepsilon}^{\text{el}} + \boldsymbol{\varepsilon}^{\text{th}}.$$

The thermal strain $\boldsymbol{\varepsilon}^{\text{th}}$ is the product of the symmetric thermal expansion tensor $\boldsymbol{\alpha}$ and relative temperature θ,

$$\boldsymbol{\varepsilon}^{\text{th}} = \theta\boldsymbol{\alpha} = \theta \begin{pmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{12} & \alpha_{22} \end{pmatrix}.$$

We consider isotropic thermal expansion with the linear coefficient α_L,

$$\boldsymbol{\alpha} = \alpha_L \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

It follows that the stress is given by

$$\boldsymbol{\sigma}_{ij} = \mathbf{c}_{ijkl}\boldsymbol{\varepsilon}^{\mathrm{el}}_{kl} = \mathbf{c}_{ijkl}\frac{\partial u_k}{\partial x_l} - \mathbf{c}_{ijkl}\boldsymbol{\alpha}_{kl}\theta.$$

By substituting this into the Cauchy's equation,

$$\boldsymbol{\nabla}\cdot\boldsymbol{\sigma} + \mathbf{f}^{\mathrm{b}} = 0,$$

we obtain

$$\begin{aligned} \boldsymbol{\nabla}\cdot(\mathbf{c}\cdot\boldsymbol{\nabla}\mathbf{u} - \mathbf{c}\cdot\boldsymbol{\alpha}\theta) + \mathbf{f}^{\mathrm{b}} &= 0, \\ \boldsymbol{\nabla}\cdot\mathbf{c}\cdot\boldsymbol{\nabla}\mathbf{u} + \mathbf{f}^{\mathrm{th}} + \mathbf{f}^{\mathrm{b}} &= 0, \\ \Leftrightarrow \quad \frac{\partial}{\partial x_j}\left(\mathbf{c}_{ijkl}\frac{\partial u_k}{\partial x_l} - \mathbf{c}_{ijkl}\boldsymbol{\alpha}_{kl}\theta\right) + \mathbf{f}^{\mathrm{b}} &= 0 \end{aligned} \tag{61}$$

which is a special case of equation (1) with the body force $\mathbf{f} = \mathbf{f}^{\mathrm{el}} + \mathbf{f}^{\mathrm{th}}$, where

$$\mathbf{f}^{\mathrm{th}} = -\boldsymbol{\nabla}\cdot\left(\mathbf{c}\cdot\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\alpha\theta\right) = -\boldsymbol{\nabla}\theta\cdot\mathbf{c}\cdot\alpha\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

is the equivalent thermal force, whose nodal values can be computed using `formInternalForce2` by letting there $\boldsymbol{\beta} = \mathbf{c} : \alpha\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $\mathbf{u} = \theta$, $\mathbf{c} = \mathbf{0}$ in (35). Since we take into account only thermal expansion and disregard other thermomechanical effects, the heat equation can be solved independently. The resulting equilibrium temperature distribution is used as input for the Cauchy's equation (61). This approach is implemented in `example2.m`.

Another way of solving this problem consists in combining Cauchy's and heat equations into a single equation with the unknown vector field $\mathbf{u} = (u, v, \theta)^{\mathrm{T}}$. The joint equation reads

$$\nabla\cdot\left(\underbrace{\begin{pmatrix} \mathbf{c}^{\mathrm{el}} & \mathbf{0}_{4\times2} \\ \mathbf{0}_{2\times4} & \mathbf{c}^{\mathrm{th}} \end{pmatrix}}_{\mathbf{c}}\begin{pmatrix} \partial u/\partial x \\ \partial u/\partial y \\ \partial v/\partial x \\ \partial v/\partial y \\ \partial\theta/\partial x \\ \partial\theta/\partial y \end{pmatrix} + \underbrace{\begin{pmatrix} \mathbf{c}^{\mathrm{el}} & \mathbf{0}_{4\times2} \\ \mathbf{0}_{2\times4} & \mathbf{c}^{\mathrm{th}} \end{pmatrix}\cdot\begin{pmatrix} \mathbf{0}_{4\times2} & -\alpha\begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \\ \mathbf{0}_{2\times2} & \mathbf{0}_{2\times1} \end{pmatrix}}_{\boldsymbol{\beta}}\cdot\begin{pmatrix} u \\ v \\ \theta \end{pmatrix}\right) - \begin{pmatrix} \mathbf{f}^{\mathrm{el}} \\ f \end{pmatrix} = 0. \tag{62}$$

Such formulation allows to incorporate reciprocal thermomechanical coupling into the equilibrium problem at the cost of increase of dimensionality. This approach is demonstrated in `example2_1.m`.

4.3 Example 3. Steady state mechanical equilibrium and dynamics of heat diffusion

In this example we solve a problem consisting of simultaneous Cauchy's equation of equilibrium and heat equation. We consider a quasi-static case of a thermomechanically coupled system, in which the dynamics of heat conduction is taken into account, while the time-dependent mechanical behaviour is disregarded. As in `example2.m`, here we solve the heat equation independently from the mechanical part. The discretised heat equation at time $t = (n+1)\Delta t$ with boundary conditions taken into account is of the form

$$\mathbf{C}\cdot\mathbf{v}_{i+1} + \mathbf{K}\cdot\mathbf{u}_{i+1} = \mathbf{F}_{i+1}, \tag{63}$$

where $\mathbf{u} = \mathbf{u}^{\mathrm{th}}$ are the nodal temperatures and $\mathbf{v} = \mathbf{v}^{\mathrm{th}}$ are their time-derivatives. We use the generalised trapezoid rule, which reads

$$\frac{\mathbf{u}_{i+1} - \mathbf{u}_i}{\Delta t} = (1-\alpha)\mathbf{v}_i + \alpha\mathbf{v}_{i+1}, \qquad \alpha \in [0, 1].$$

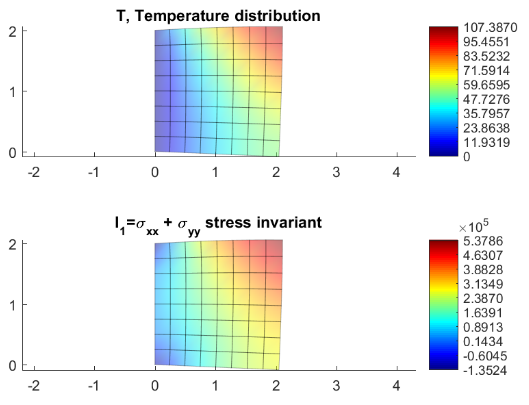


Figure 5: Example 2. Simultaneous equilibrium temperature distribution and mechanical equilibrium with thermal expansion taken into account.

Depending on whether velocity or displacement is used as the primary unknown, one of the following equivalent expressions can be used:

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \Delta t(1-\alpha)\mathbf{v}_i + \Delta t\alpha\mathbf{v}_{i+1}, \tag{64}$$

$$\mathbf{v}_{i+1} = \frac{1}{\alpha\Delta t}(\mathbf{u}_{i+1} - \mathbf{u}_i) - \frac{1-\alpha}{\alpha}\mathbf{v}_i. \tag{65}$$

Substituting these into equation (63) yields respectively

$$(\mathbf{C} + \alpha\Delta t\mathbf{K}) \cdot \mathbf{v}_{i+1} = \mathbf{F}_{i+1} - \mathbf{K} \cdot (\mathbf{u}_i + (1-\alpha)\Delta t\mathbf{v}_i), \tag{66}$$

$$\left(\mathbf{C}\frac{1}{\alpha\Delta t} + \mathbf{K}\right) \cdot \mathbf{u}_{i+1} = \mathbf{F}_{i+1} - \mathbf{C} \cdot \left(\frac{1}{\alpha\Delta t}\mathbf{u}_i + \frac{1-\alpha}{\alpha}\mathbf{v}_i\right). \tag{67}$$

We use the the latter relation and fragment of code, which implements the integration in time. See `example3.m` for full code.

Listing 7: Example 3 - Generalised trapezoid rule for heat equation

```
%initial temperature and rate
u_T=u0_T;
v_T=zeros(numGDoF_T,1);
v_T(freeDoF_T)=CC_T\(FF_T+KK_T*u_T(freeDoF_T));

for i=1:numTimesteps
    % make a time–step
    u_old_T=u_T(freeDoF_T);
    v_old_T=v_T(freeDoF_T);
    % effective right–hand side
    rhs=FF_T+CC_T*(u_old_T+(1–alpha)*dt*v_old_T)/(alpha*dt);
    % effective linear operator
    lhs=(CC_T+alpha*dt*KK_T)/(alpha*dt);
    u_T(freeDoF_T)=lhs\rhs; %solution
    v_T(freeDoF_T)=(u_T(freeDoF_T)–u_old_T–(1–alpha)*dt*v_old_T)/alpha/dt;

    %%% compute elastic componenets and display results at selected timesteps
        % ....
end
```

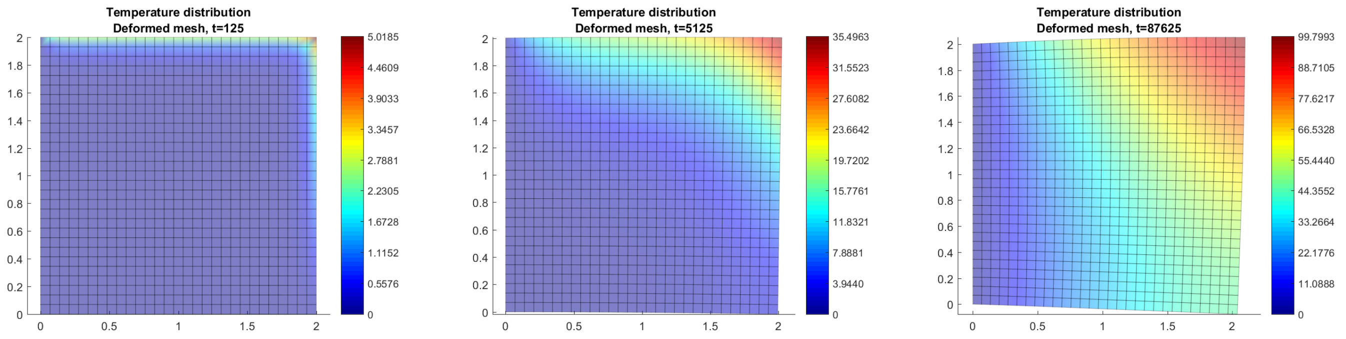


Figure 6: Example 3. Heat equation coupled with the (quasi-static) mechanical equilibrium equation.

4.4 Example 4. Benchmark problem. Rectangular plate

In this example we solve a plane strain equilibrium problem for a linear elastic material and compare our solution to the exact solution. The exact analytical solution is generated beforehand using the Airy functions. The contraction of this exact solution is taken as boundary conditions for problem that is solved numerically.

We compare various recovery procedures and observe that the superconvergent patch recovery method is superior, as expected, see Figure 7 and Figure 8.a. See `example4_1.m` and `example4_2.m` for full code.

4.5 Example 5. Benchmark problem. Plate with a hole (Kirsch problem)

Numerical solution for the Kirsch problem (infinite plate with hole) is compared with analytical results. The main difference from Example 4 is that the domain is not rectangular anymore. Therefore we generate mesh using Matlab `initmesh()` function, which is a part of PDE Toolbox, following the scheme shown in Figure 1.b. Here is the code for the mesh generation. See `example5_1.m` and `example5_2.m` for full code.

Listing 8: Example 5 - Mesh generation using PDE Toolbox function meshinit()

```
csg_rect=[3 4 0 L L 0 0 0 L L]'; %define the rectangle
csg_circ=[1 0 0 a]'; %define the circle
gd=[csg_rect, [csg_circ; zeros(numel(csg_rect)-numel(csg_circ),1)]]; %combine
ns = char('rect1','circ1')'; %Give names to the shapes - namespace matrix
sf = 'rect1-circ1';% Specify the set formula
[dl,~] = decsg(gd,sf,ns); % produces decomposed geometry

% . . .
% some code is skipped here
% . . .

% using Pde Toolbox generate triangular mesh
[nodeCoords,e,IEN] = initmesh(dl,'Hmax',hmax);
nodeCoords=nodeCoords'; % bring it to the form we use
IEN=IEN(1:3,:)'; % bring it to the form we use
INE=IENtoINE(IEN);
[boundaryElementIDs, boundaryNodeLocalIDs]=edges2sublists(e,IEN,INE);

[nodeCoords, IEN,boundaryElementIDs, boundaryNodeLocalIDs] = ...
        meshP1toP2(nodeCoords,IEN,boundaryElementIDs, boundaryNodeLocalIDs);
% convert 3-node elements into 6-node elements
```

4.6 Example 6. Thermal expansion of a bimaterial

In this example, a bimetal (brass and steel) is considered. We solve the heat equation for equilibrium temperature distribution and then solve Cauchy's equation of equilibrium adjusted for thermal expansion. The variation of

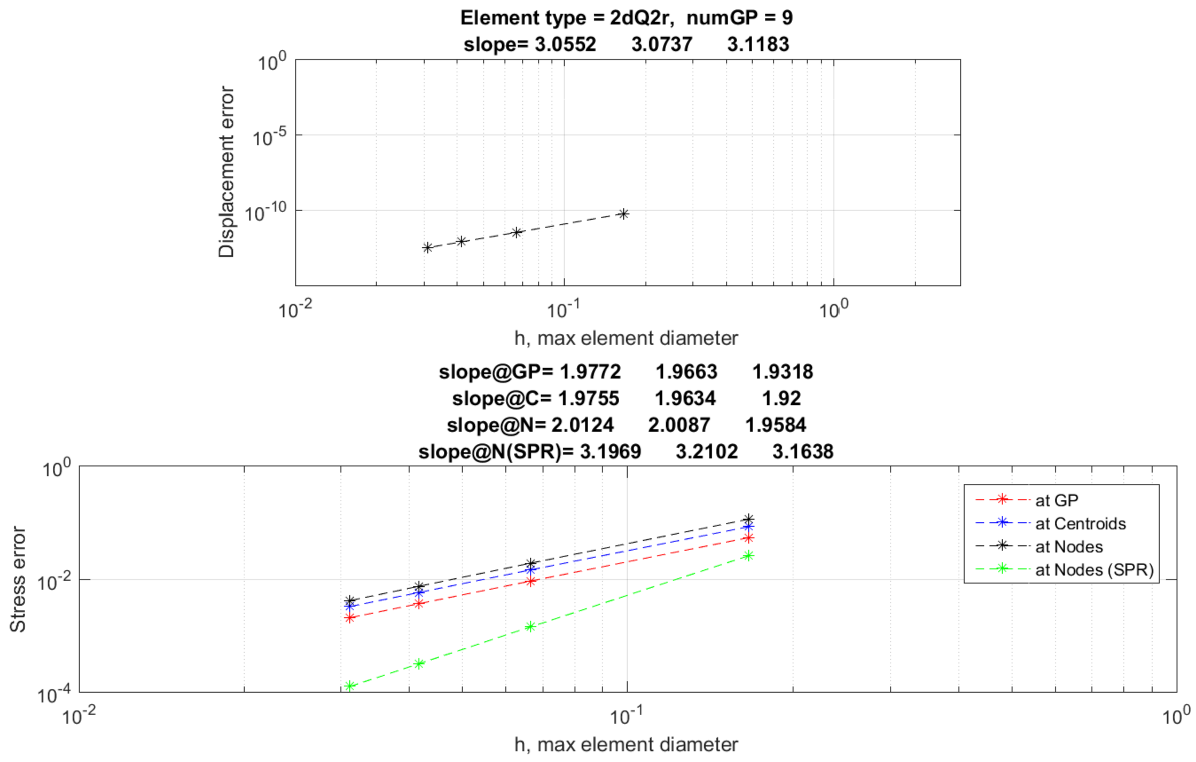


Figure 7: Example 4. Comparison of different stress recovery techniques.

a. **b.**

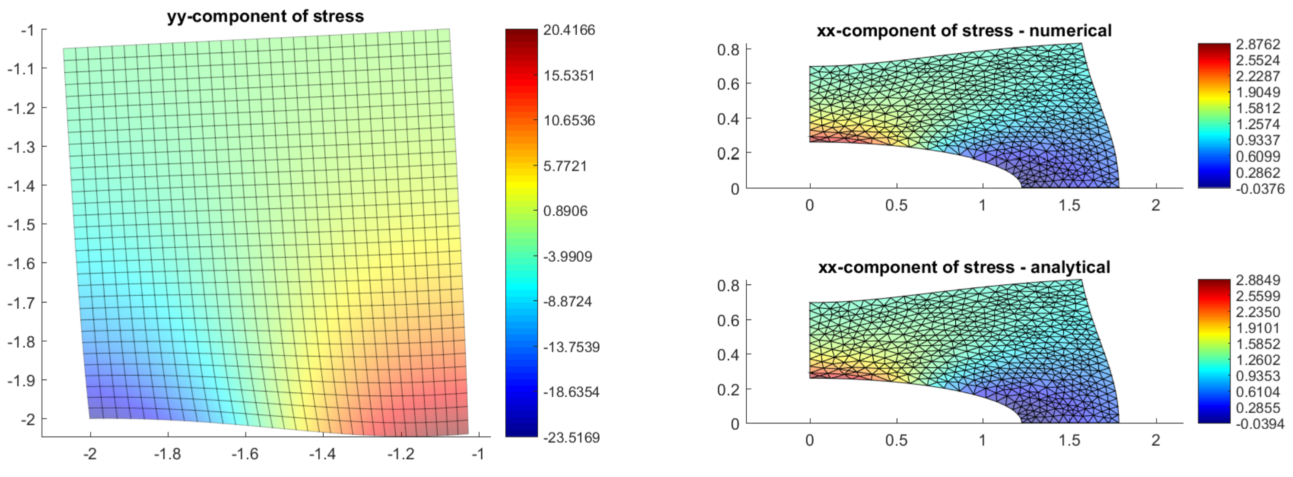


Figure 8: **a**. Example 4. The equilibrium stress field generated using arbitrarily constructed Airy function. **b**. Example 5. Kirsch problem (plate with a hole)

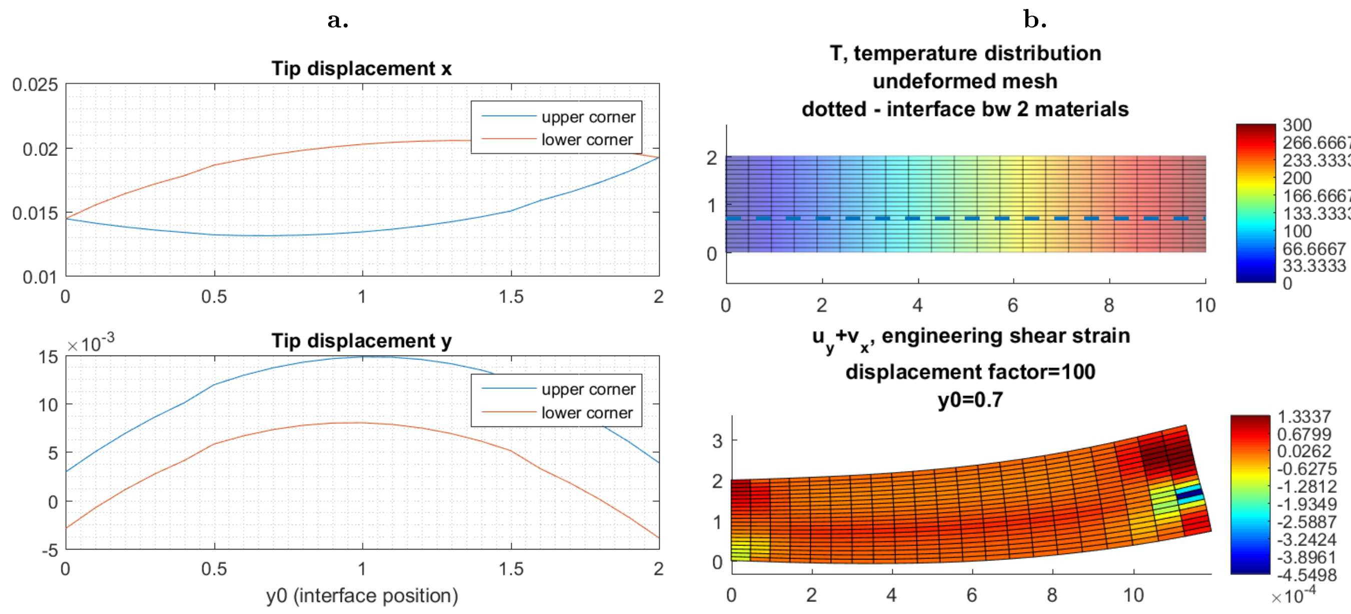


Figure 9: **a**. Deflection of the tip of the bimaterial as a function of the composition (the spatial position of the interface between two metals). **b**. Reference and deformed configurations for a bimaterial consisting of 0.7 parts of brass and 1.3 parts of steel.

physical properties in the bimetal is implemented by position dependent constitutive tensors. Here we show how it is done for the elastic modulus. Heat conductivity and thermal expansion tensors are treated in the same way. See `example6.m` and `example6_main.m` for full code.

Listing 9: Example 6 - Position-dependent material properties

```
properties1=elasticProperties('youngsModulus',youngsModulus1,'poissonsRatio',poissonsRatio1);
properties2=elasticProperties('youngsModulus',youngsModulus2,'poissonsRatio',poissonsRatio2);
CMatrix_E1=properties1.CPlaneStressFull;
CMatrix_E2=properties2.CPlaneStressFull;
    % this function defines position−dependent elasticity modulus
    function CMat=CMatrix_E(x,~,~,~)

        if nargin==0
            CMat=CMatrix_E1;
        else
            CMat=((x(2)>=y0)*CMatrix_E1+(x(2)<y0)*CMatrix_E2);
        end
    end
```

4.7 Example 7. Bending of a cantilever beam in 3D.

This example compares bending of a cantilever beam in 3D to the corresponding plane stress and plane strain formulations. The only difference between 3D and 2D workflows is found at the visualisation stage. In 3D meshes, only the surface of the domain are shown. Therefore, the boundary of the domain is displayed by passing the boundary incidence array and the boundary element types to `drawElements()` and `drawElementsInterp()` routines. Below we show two excerpts of code creating the plots: for a 2D mesh and for a 3D mesh. See `example7_1.m`, `example7_2.m`, `example7_3.m`, and `example7_4.m` for full code.

Listing 10: Example 7 - Difference between 2D and 3D visualisation

```
%===== 2D Plots ===== %
drawElements(nodeCoords,IEN,elementType,0,0)
```

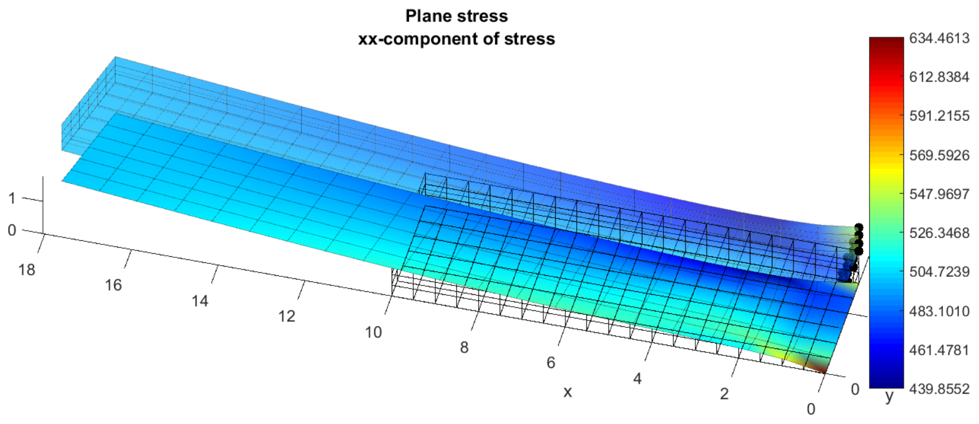


Figure 10: Example 7. Bending of a 3D cantilever beam versus the plane stress formulation.

```
drawElementsInterp(nodeCoords+u2_E*factor,IEN,elementType,s2(1,:));

% . . .
% Some code skipped here. The variables are redefined for the 3D problem.
% . . .

%===== 3D Plots ===== %
hold on;
shift=ones(size(nodeCoords,1),1)*[0 0 1];
drawElementsInterp(shift+nodeCoords+u2_E*factor,boundaryIEN,bndElementType,s2(1,:),.5);
drawNodes(shift+nodeCoords+u2_E*factor,boundaryIEN(3));
```

4.8 Example 8. Oscillation of a cantilever beam (Newmark's method)

This example compares implicit and explicit Newmark's methods. We consider oscillation of cantilever beam, which are induced by a sudden release of bending loading. Here we list the part of the code containing the call of `solveNewmarkLa()` and the definition of `outputFunction()`, which passed to the solver and needed to show results at selected time steps as well as to collect data of interest (the history of displacements, velocities and accelerations). See `example8_1.m`, `example8_2.m`, and `example8_main.m` for full code.

Listing 11: Example 8 - Newmark's method

```
[output]=solveNewmarkLa(beta,gamma,ts,freeDoF,prescribedDoF,...
    M1,[],K,F,...
    u0,v0,a0,...
    u_prescribed,v_prescribed,a_prescribed,...
    'increment',@outputFunction);
u=output{1};
v=output{2};
a=output{3};

% function passed to solveNewmarkLa() used for output and plotting
    function res=outputFunction(i,t,u,v,a,fDoF,pDoF,M,C,K,F,u0,v0,a0,up,vp,ap) %#ok<INUSL,INUSD>
        if doplot && mod(i−1,displayevery)==0
            % Plotting the results
            u2=reshape(u,[numEq numel(u)/numEq])';
            [gradu]=recoveryAvg(u2,nodeCoords,IEN,elementType);
            s2=CMatrix*gradu;

            figure(1);clf;
```

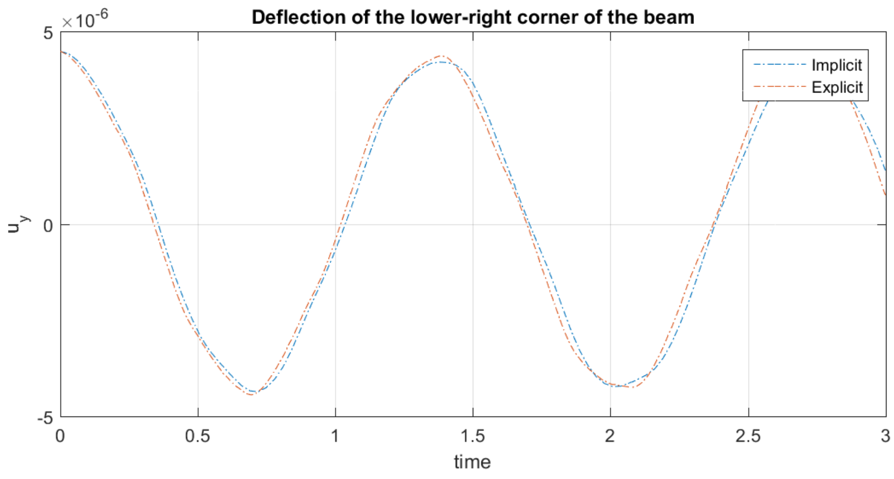


Figure 11: Example 7. Deflection of the tip of an oscillating cantilever beam as a function of time.

```
            drawElementsInterp(nodeCoords+u2*factor,IEN,elementType,s2(1,:));
            axis((eye(4)+.2*[-1 -1 0 0; 1 1 0 0; 0 0 -1 -1; 0 0 1 1])*domain(:));
            title('\sigma_{xx} stress component');
            view(2)
            drawnow;
        end
        res={u,v,a};
    end
```

References

[1] K.J. Bathe. *Finite Element Procedures.* Prentice-Hall International Series in. Prentice Hall, 1996.

[2] T. Belytschko, W.K. Liu, and B. Moran. *Nonlinear Finite Elements for Continua and Structures.* Nonlinear Finite Elements for Continua and Structures. Wiley, 2000.

[3] T.J.R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis.* Prentice Hall International Editions. Prentice-Hall, 1987.

[4] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method, The Basis.* The Finite Element Method. Wiley, 2000.