TENSORLAB 2.0 **tutorial**

1 Introduction

TENSORLAB is a MATLAB toolbox which facilitates the use of vectors and tensors of arbitrary order. The toolbox also allows for the definition of matrices, or even multi-dimensional arrays, which have tensors as their elements. The use of TENSORLAB results in programs – for instance finite element programs – which are more readable than traditional implementations based on compact matrix notation and it allows one to virtually reproduce written derivations in computer code. To this end, TENSORLAB defines a new MATLAB class called *tensor*. When operations are preformed on objects of this class, MATLAB automatically chooses the operations defined for it.
The purpose of this tutorial is to give an introduction to the use of TENSORLAB types. It is assumed that the reader is familiar with MATLAB and procedural programming in MATLAB (for example: functions, conditional statements, m-files). A comprehensive understanding of object oriented programming is not necessary.

2 Installing the TENSORLAB **toolbox**

Installation of the TENSORLAB toolbox for MATLAB is done as follows.

1. Download and save the file `TensorLab20.zip` to an arbitrary folder.
2. Select an installation folder, preferably the existing MATLAB folder; as an example, we use:

   ```
   D:\Documents\MATLAB\
   ```

3. Extract the zip-file to this folder, whereby the subfolders in the zip-file have to be preserved (for WINZIP make sure the option *Use folder names* is switched on).
4. Finally, the newly created `TensorLab20\` folder has to be added to the search path of MATLAB. To this end start MATLAB and select *Set Path* in the *Home* tab. In the pop-up window select *Add with subfolders*. Then navigate to the folder you have just created (above). After selecting *OK*, press *Save*. The TENSORLAB toolbox is now permanently added to the search path of MATLAB. You may now *Close* the window and start using the TENSORLAB toolbox.

After a successful installation, the file `TensorLab20.zip` can be safely deleted.
TENSORLAB can be removed from your computer at any given time by deleting the folder in which it is installed and removing this folder from the MATLAB path.

3 Becoming familiar with MATLAB **classes**

In order to clarify the role of the tensor class, we first recall the the use of a common MATLAB class: the floating point number with double precision called *double*. Typing

```
a = 1
```

at the MATLAB prompt creates a scalar variable *a* which takes the value 1.0. The class to which this variable belongs, i.e. the type of data it can contain, may be examined using

```
class(a)
```

which results in MATLAB replying by

```
ans =

double
```

MATLAB's definition of the double class is special in the sense that it can also hold *matrices* of (double precision) scalars. Virtually all users of MATLAB will have some experience with this. For example, typing

```
B = [1 2; 3 4]
C = [1 2 3; 4 5 6; 7 8 9]
```

at the MATLAB prompt creates the 2×2 and 3×3 (double) matrices $\underline{B}$ and $\underline{C}$ respectively. It can easily be verified that these objects indeed belong to the double class using the `class()` function.
The purpose of the double matrix class provided by MATLAB is to allow the user to define and manipulate matrices in a similar fashion as in writing. A double matrix can be assigned to a variable (see `B` and `C` above), after which we can call functions or methods which manipulate the data stored in the matrix. A small selection of standard operators and methods defined in MATLAB for double matrices is as follows (MATLAB operator between parentheses):

- addition and subtraction (`+`, `-`)
- matrix multiplication (`*`)
- solving of a linear system (\)
- transposition (`'`)
- inversion (`inv()`)
- eigenvalue extraction (`eig()`)

Type for instance

```
D = inv(B)
```

to immediately obtain the inverse of the matrix $\underline{B}$ and

```
B * D
```

to verify that $\underline{D}$ is indeed the inverse of $\underline{B}$.
Some form of type checking is performed if these operations and methods are used. For instance, the inner dimensions of the two double matrix objects in a matrix multiplication are checked, and the operation is not performed if the inner dimensions do not match. Try for instance

```
B * C
```

for the matrices defined above and observe that an error results and the program is halted.
Now that we have seen that we have already been using MATLAB classes and their methods intuitively as mathematical objects, let us return to the objective of this tutorial and introduce some of the standard and non-standard methods provided by TENSORLAB.

4 Defining vectors and tensors

The TENSORLAB toolbox defines a tensor class which can be used in a similarly natural way as the standard MATLAB matrix objects discussed above. The methods (operations) available for tensors safeguard the user against the incorrect construction of mathematical objects and incorrect use of operators.
To create a vector or a tensor, we first need to define a vector basis. TENSORLAB allows the user to define two- or three-dimensional Cartesian bases. Here we largely restrict ourselves to the two-dimensional case, but the three-dimensional counterparts are defined fully analogically.
A column matrix $\underline{\vec{e}}$ containing the two basis vectors of a two-dimensional Cartesian basis is defined by typing

```
e = cartesianbasis2d('ex', 'ey')
```

resulting in the screen output

```
e =

    1.0000*ex + 0.0000*ey
    0.0000*ex + 1.0000*ey
```

The arguments `'ex'` and `'ey'` of the above statement are the names assigned to the two basis vectors. These names are used in showing vectors and tensors on the screen, as evidenced by the screen output above. If no names are provided, the default definitions `e(1)` and `e(2)` are used.
To construct vectors or tensors in terms of the vector basis, it is useful to define each of the basis vectors contained in the column matrix $\underset{\sim}{\vec{e}}$ as a separate variable:

```
ex = e(1);
ey = e(2);
```

As a result, `ex` and `ey` have become the two individual basis vectors $\vec{e}_x$ and $\vec{e}_y$.
We are now able to define vectors and tensors in terms of the basis vectors $\vec{e}_x$ and $\vec{e}_y$. For example, to define the vector

$$\vec{v} = 3\,\vec{e}_x + \vec{e}_y$$

type

```
v = 3*ex + ey
```

MATLAB replies with

```
v = 3.0000*ex + 1.0000*ey
```

The class to which the newly defined object belongs may be examined by typing

```
class(v)
```

which results in

```
ans =

tensor
```

This shows that `v` is of the class tensor. Indeed, vectors are regarded by TENSORLAB simply as tensors of order one, as can be verified by determining the tensorial order of $\vec{v}$ via

```
order(v)
```

Higher-order tensors can be defined in the same natural fashion. For instance, to define the second-order tensor $\boldsymbol{T}$ and the fourth-order tensor ${}^4\boldsymbol{U}$ according to

$$\boldsymbol{T} = 4\,\vec{e}_x\vec{e}_x + 10\,\vec{e}_y\vec{e}_y$$
$${}^4\boldsymbol{U} = 2\,\vec{e}_x\vec{e}_x\vec{e}_x\vec{e}_x - 2\,\vec{e}_y\vec{e}_y\vec{e}_y\vec{e}_y$$

type

```
T = 4*ex*ex + 10*ey*ey
U = 2*ex*ex*ex*ex - 2*ey*ey*ey*ey
```

and verify using the `class()` and `order()` methods that the resulting objects are indeed tensors of order two and four.

5 Operations defined for the tensor class

Similar to the standard MATLAB double class, a number of operators have been defined for TENSORLAB's tensor class. A brief summary of the most important ones is as follows (operator between parentheses):

- addition and subtraction (`+`, `-`)
- dyadic product, i.e. product without contraction (`*`)
- inner product or dot product (`dot()`)
- solution of a vector equation (\)
- the norm of a vector (`norm()`)
- transposition of tensors (`'`, `ttranspose()`)
- determinant of a tensor (`det()`)
- tensor inverse (`inv()`)
- eigenvalue extraction (`eig()`)

These operations are discussed in more detail below using simple examples.

5.1 Addition and subtraction

We have already tacitly used the addition of tensor objects in defining the vector $\vec{v}$ and tensors $\boldsymbol{T}$ and ${}^{4}\boldsymbol{U}$ above.
As a further example, define the vectors

$$\vec{p} = 10\,\vec{e}_x + 2\,\vec{e}_y$$
$$\vec{q} = 20\,\vec{e}_x + 4\,\vec{e}_y$$

by typing

```
p = 10*ex + 2*ey
q = 20*ex + 4*ey
```

The sum $\vec{r}$ of $\vec{p}$ and $\vec{q}$ is computed simply by

```
r = p + q
```

Verify that the result is the one you would expect and check the class to which it belongs. Also try

```
p - q
```

Notice how MATLAB automatically selects a tensor operation (addition or subtraction) when variables of the tensor class are involved.

5.2 Dyadic product

The automatic selection of tensor operators also holds for products between tensor objects, as well as between scalars and tensors. In order to define the vector $\vec{p}$ above, we multiplied the scalars 10 and 2 by the vectors $\vec{e}_x$ and $\vec{e}_y$, respectively, using the operator `*`. As no contraction is involved in this product, it is essentially a dyadic (or tensor) product.
Similarly, the vector

$$\vec{s} = 3\,\vec{p}$$

can be computed via

```
s = 3*p
```

The dyadic product of two vectors, e.g. $\vec{p}$ and $\vec{q}$, results in a second-order tensor:

```
V = p * q
```

gives

```
V =

    200.0000*ex*ex + 40.0000*ex*ey + 40.0000*ey*ex + 8.0000*ey*ey
```

Try also

```
p * q * p
p * V
V * V
```

and verify the results.

5.3 Dot product

A counterpart of the dyadic product is the dot product or inner product. This product involves a contraction of two basis vectors and therefore reduces the combined tensorial order of its argument by two. For vectors, the dot product therefore results in a scalar.
For instance define two new vectors $\vec{a}$ and $\vec{b}$ as

$$\vec{a} = 5\,\vec{e}_x + 2\,\vec{e}_y$$
$$\vec{b} = 2\,\vec{e}_x - 5\,\vec{e}_y$$

and compute their dot product

$$c = \vec{a} \cdot \vec{b}$$

by typing

```
c = dot(a, b)
```

Based on the result, would you say $\vec{a}$ and $\vec{b}$ are orthogonal?
We can also use the `dot()` function for tensors of orders higher than one. For instance, if we have the tensor

$$\boldsymbol{\sigma} = 3\,\vec{e}_x\vec{e}_x + 1.5\left(\vec{e}_x\vec{e}_y + \vec{e}_y\vec{e}_x\right) + \vec{e}_y\vec{e}_y$$

and the vector

$$\vec{n} = \frac{1}{\sqrt{5}}\left(\vec{e}_x + 2\,\vec{e}_y\right)$$

and wish to determine the vector

$$\vec{t} = \boldsymbol{\sigma} \cdot \vec{n}$$

this can be done by

```
sigma = 3*ex*ex + 1.5*(ex*ey + ey*ex) + ey*ey
n = 1/sqrt(5) * (ex + 2*ey)
t = dot(sigma, n)
```

The projection t_n of $\vec{t}$ on $\vec{n}$ is subsequently computed as

```
tn = dot(t, n)
```

or directly via

```
tn = dot(n, sigma, n)
```

The latter shows that the function `dot()` can take more than two arguments. To further explore this feature, compute also:

```
dot(sigma, sigma, n)
dot(n, sigma, sigma, sigma, n)
dot(n, n, n)
```

Why does this final instruction result in an error?
Similar to the single dot product, a double, triple and quadruple dot product are also defined in TENSORLAB; they perform two, three and four contractions, respectively. Examples of their use are:

```
ddot(sigma, sigma)
ddot(sigma*sigma, sigma*sigma)
ddot(sigma, sigma*sigma, sigma)
ddddot(sigma*sigma, sigma*sigma)
```

5.4 Solution of a vector equation

A vector equation is of the form

$$\boldsymbol{K} \cdot \vec{u} = \vec{f}$$

where the second-order tensor $\boldsymbol{K}$ and the vector $\vec{f}$ are known and we wish to find the vector $\vec{u}$ such that the above relation holds.
One way to determine $\vec{u}$ would be by pre-multiplying the left hand side and right hand side of the equation by the inverse of $\boldsymbol{K}$ to obtain $\vec{u} = \boldsymbol{K}^{-1} \cdot \vec{f}$; see below for the TENSORLAB instructions needed to compute the inverse.
However, a more efficient way is to directly compute the solution $\vec{u}$ by solving for it in a similar way as solving for a column matrix in linear algebra. Like for scalar matrices in standard MATLAB, TENSORLAB uses the \ (backslash) operator for this purpose.
Define for instance the tensor $\boldsymbol{K}$ and vector $\vec{f}$ as

```
K = 5*ex*ex - 2*ex*ey - 2*ey*ex + 5*ey*ey
f = 12*ex - 9*ey
```

and compute $\vec{u}$ via

```
u = K \ f
```

Verify that $\vec{u}$ is indeed the solution to the above equation by computing the product $\boldsymbol{K} \cdot \vec{u}$.
The tensorial order of the solution and the number or inner products implied in the equation to be solved are determined from the orders of the two arguments of the \ operator. Try for instance the following operation:

```
K \ (f*f)
```

5.5 Norm of a vector

The norm (or length) of a vector is found using `norm()`. Verify for instance that the vector

$$\vec{d} = 4\,\vec{e}_x + 3\,\vec{e}_y$$

has length 5 by typing

```
d = 4*ex + 3*ey
norm(d)
```

Note that the same result could have been obtained using the definition of the 2-norm:

```
sqrt( dot(d, d) )
```

5.6 Transpose of a tensor

The transpose $\boldsymbol{R}$ of a second-order tensor

$$\boldsymbol{Q} = \vec{e}_x\vec{e}_y - \vec{e}_y\vec{e}_x$$

is obtained in TENSORLAB by

```
R = ttranspose(Q)
```

or, more compact:

```
R = Q'
```

For higher-order tensors, e.g.

$${}^4\boldsymbol{S} = \vec{e}_x\vec{e}_y\vec{e}_x\vec{e}_y - \vec{e}_y\vec{e}_x\vec{e}_y\vec{e}_x$$

full transposition reverses the order of all basis vectors, i.e.

$${}^4\boldsymbol{S}^{\mathrm{T}} = -\vec{e}_x\vec{e}_y\vec{e}_x\vec{e}_y + \vec{e}_y\vec{e}_x\vec{e}_y\vec{e}_x$$

Verify this using TENSORLAB.
For partial transposition, the functions `ltranspose()` and `rtranspose()` are available. For tensors of even order $2n$, they determine respectively the left transpose (first n vectors reversed) and right transpose (final n reversed). Based on this, predict the results of

```
ltranspose(S)
rtranspose(S)
rtranspose( ltranspose(S) )
ttranspose( ttranspose(S) )
```

and verify them using TENSORLAB.

5.7 Determinant of a tensor

As for standard MATLAB matrices, the determinant of a second-order tensor is computed using `det()`. For instance, the determinant of the (two dimensional) identity tensor can be computed as

```
I = ex*ex + ey*ey
det(I)
```

Not surprisingly, the result is one.

5.8 Tensor inverse

The inverse of a tensor is determined using `inv()`. Try for instance

```
inv(I)
inv(2*I)
inv(ex*ex)
```

What is the problem in the final statement?

5.9 Eigenvalues and eigenvectors

The eigenvalues of a second-order tensor

$$\boldsymbol{W} = \vec{e}_x\vec{e}_x - \vec{e}_y\vec{e}_y$$

are determined by

```
W = ex*ex - ey*ey
lambda = eig(W)
```

This statement results in a column matrix which contains the eigenvalues of the tensor.
The eigenvectors may be extracted as well by

```
[N, Lambda] = eig(W)
```

Note that the eigenvalues are now returned in a diagonal matrix rather than a column matrix, so that we can reconstruct the original tensor by the usual spectral form

```
N * Lambda * N'
```

6 Tensor matrices

Similar to the double class in standard MATLAB, TENSORLAB's tensor type can also take multiple values in the form of a matrix, i.e. matrices of tensors (or vectors). Many of the operators (methods) defined for tensors also accept such matrices.
We have already seen a first example of a (column) matrix of tensors (vectors) when we defined the basis $\vec{\underset{\sim}{e}}$ at the start of Section 4. This object is essentially a 2×1 matrix of order one tensor objects. To see this, determine the class, tensorial order and matrix dimensions of $\vec{\underset{\sim}{e}}$ by typing at the MATLAB prompt

```
class(e)
order(e)
size(e)
```

Instead of the latter statement we could also have used the `length()` method, which determines the number of elements in a column or row matrix.

6.1 Constructing tensor matrices

A matrix of tensors can be constructed in a number of ways, which are all similar to those used for scalar matrices in standard MATLAB. For example, to define the matrix of vectors

$$\vec{\mathrm{A}} = \begin{bmatrix} \vec{x} & -\vec{x} \\ -\vec{x} & \vec{x} \end{bmatrix}$$

where

$$\vec{x} = 3\,\vec{e}_x + 5\,\vec{e}_y$$

type

```
x = 3*ex + 5*ey
A = [ x  -x ;  -x  x ]
```

Note that, for a matrix which contains more than one column, the elements of the matrix are listed element-wise in order, where the numbers between brackets indicate each element's position:

```
A =

   (1,1)      3.0000*ex + 5.0000*ey
   (2,1)     -3.0000*ex - 5.0000*ey
   (1,2)     -3.0000*ex - 5.0000*ey
   (2,2)      3.0000*ex + 5.0000*ey
```

A matrix of second-order tensors is similarly defined as

```
B = [ (x*x)  -(x*x) ;  -(x*x)  (x*x) ]
```

Rows and columns may be appended to an existing matrix in the usual way as well:

```
C = [ B ;  (ex*ex)  (ey*ey) ]
D = [ B  [(ex*ex); (ey*ey)] ]
```

but not

```
[ C  D ]
```

as the number of rows of these matrices do not agree.
Elements may also be extracted from a matrix of tensors in the same way as for scalar matrices, e.g.

```
A(1, 1)
```

returns a vector which is equal to $\vec{x}$. Index matrices and the colon (`:`) can be employed to simultaneously extract a number of elements, entire columns, rows or sub-matrices. For example, the following instructions all return the first column of $\vec{\mathrm{A}}$:

```
A([1 2], 1)
A(1:2, 1)
A(:, 1)
```

6.2 Adding and subtracting matrices

Similar to ordinary MATLAB matrices (i.e. matrices of scalars), we can can add and subtract matrices of tensors. For instance, if we define

```
B = -A
```

we find that the result of

```
A + B
```

is a matrix of zeros vectors and

```
A - B
```

returns a matrix equal to $2\vec{\mathrm{A}}$.

6.3 Matrix products

Matrix multiplications obey the usual 'row times column' convention. This holds for the dyadic product (`*`), as well as for the various degrees of dot products (`dot()`, `ddot()`, etc.). Try for instance

```
A * A
dot(A, A)
ddot(B, B)
dot(B, B, B)
```

As usual, products with a single variable (i.e. a 1×1 matrix) imply multiplication of all elements of a matrix by that variable, e.g.

```
x * A
```

returns a matrix equal to $\mathbf{B}$.

6.4 Solution of systems of vector equations

The operator \ (backslash) can deal with linear systems of vector equations. So to solve the linear vector system

$$\mathbf{M} \cdot \vec{\underset{\sim}{a}} = \vec{\underset{\sim}{q}}$$

where $\mathbf{M}$ is a matrix of second-order tensors and $\vec{\underset{\sim}{q}}$ a column of vectors, type

```
a = M \ q
```

6.5 Transposition

A little care is required in transposing tensor matrices, as the transposition may apply to the the matrix dimensions or to the components of the tensors within the matrix, or to both. A number of operators are therefore available; we use the tensor matrix

$$\mathbf{G} = \begin{bmatrix} \boldsymbol{G}_{11} & \boldsymbol{G}_{12} & \boldsymbol{G}_{13} \\ \boldsymbol{G}_{21} & \boldsymbol{G}_{22} & \boldsymbol{G}_{23} \end{bmatrix}$$

to illustrate their definitions:

- `'` or `ctranspose()` transposes both the matrix and the tensors contained in it; `G'` thus results in the matrix

$$\begin{bmatrix} \boldsymbol{G}_{11}^{\mathrm{T}} & \boldsymbol{G}_{21}^{\mathrm{T}} \\ \boldsymbol{G}_{12}^{\mathrm{T}} & \boldsymbol{G}_{22}^{\mathrm{T}} \\ \boldsymbol{G}_{13}^{\mathrm{T}} & \boldsymbol{G}_{23}^{\mathrm{T}} \end{bmatrix}$$

- `.'` or `transpose()` transposes only the matrix while leaving the tensors in it untouched, so `G.'` gives

$$\begin{bmatrix} \boldsymbol{G}_{11} & \boldsymbol{G}_{21} \\ \boldsymbol{G}_{12} & \boldsymbol{G}_{22} \\ \boldsymbol{G}_{13} & \boldsymbol{G}_{23} \end{bmatrix}$$

- `ttranspose()`, as well as `ltranspose()` and `rtranspose()`, operates only on the tensors components and leaves the matrix structure intact; e.g. the result of `ttranspose(G)` reads

$$\begin{bmatrix} \boldsymbol{G}_{11}^{\mathrm{T}} & \boldsymbol{G}_{12}^{\mathrm{T}} & \boldsymbol{G}_{13}^{\mathrm{T}} \\ \boldsymbol{G}_{21}^{\mathrm{T}} & \boldsymbol{G}_{22}^{\mathrm{T}} & \boldsymbol{G}_{23}^{\mathrm{T}} \end{bmatrix}$$

7 Special tensors

7.1 Identity tensor

Identity tensors $\boldsymbol{I}$, ${}^4\boldsymbol{I}$, etc. are defined such that for any $\vec{x}$, $\boldsymbol{X}$, etc. we have

$$\boldsymbol{I} \cdot \vec{x} = \vec{x}$$

$${}^4\boldsymbol{I} : \boldsymbol{X} = \boldsymbol{X}$$

Note that this definition implies that identity tensors are of even order.
TENSORLAB provides the function `identity()` to define these special tensors. This function takes the order of the tensor and the vector basis in which is defined as arguments. In terms of the basis $\vec{\underset{\sim}{e}}$ which we have defined we can thus have

```
I  = identity(2, e)
I4 = identity(4, e)
```

etc. Verify that these tensors indeed have the property demonstrated above.
Note that the fourth-order symmetrisation tensor ${}^4\boldsymbol{I}^{\mathrm{S}}$ can now be constructed using

```
I4S = 1/2 * (I4 + rtranspose(I4))
```

7.2 Zero tensor

In a similar fashion, a zero tensor can be defined using `zeros()`. It, too, takes the order of the tensor and the basis as its arguments. This function can furthermore be used to construct matrices of zero-tensors by also specifying the matrix dimensions. For example, the result of

```
O = zeros(2, e, 4, 4)
```

is a 4×4 matrix of second-order zero-tensors.

8 Plotting

The TENSORLAB toolbox offers some functions which allow one to visualise vectors and tensors graphically. As for the other functionality, their use largely follows that of MATLAB functions such as `scatter()` (for scalar data) and `quiver()` (for vectors).
A column of position vectors $\underset{\sim}{\vec{\mathrm{x}}}$ is defined, as well as a data column which may have a scalar, vector or tensor character. Figure 1 shows the results obtained for these different categories for $\underset{\sim}{\vec{\mathrm{x}}}$ according to

$$\underset{\sim}{\vec{\mathrm{x}}} = \begin{bmatrix} \vec{0} & \vec{a} & 2\vec{a} & 3\vec{a} \end{bmatrix}^{\mathrm{T}}$$

with $\vec{a}$ the three-dimensional vector

$$\vec{a} = \vec{e}_x + \vec{e}_y + \vec{e}_z$$

and each of the following data:

- a scalar column given by

  $$\underset{\sim}{\mathrm{u}} = \begin{bmatrix} 160 & 120 & 80 & 40 \end{bmatrix}^{\mathrm{T}}$$

  the result is a scatter plot, with circles (or spheres in three dimensions) at the positions $\vec{x}_i$ which have a size and colour which depends on the value u_i. For this specific example we have used:

  ```
  scatter(x, 2*abs(u), u, 'filled')
  colorbar
  ```

  wherein the second argument scales the size of the circles and the third argument determines their colour. The (optional) last arguments causes the circles to be filled. To obtain circles of the same size, specify an empty array for the second argument.

- a column of vectors equal to

  $$\underset{\sim}{\vec{\mathrm{u}}} = \begin{bmatrix} 4\vec{e}_z & 3\vec{e}_z & 2\vec{e}_z & \vec{e}_z \end{bmatrix}^{\mathrm{T}}$$

  these vectors are plotted starting at $\vec{x}_i$ and their colour reflects the length of each vector. We execute:

  ```
  quiver(x, u, 0)
  ```

wherein the last (optional) argument turns off the automatic scaling, thus ensuring that the arrows in the plot are exactly the vectors' length.

- a column of second-order tensors

$$\underset{\sim}{\mathbf{u}} = \begin{bmatrix} 4\boldsymbol{C} & 3\boldsymbol{C} & 2\boldsymbol{C} & \boldsymbol{C} \end{bmatrix}^{\mathrm{T}}$$

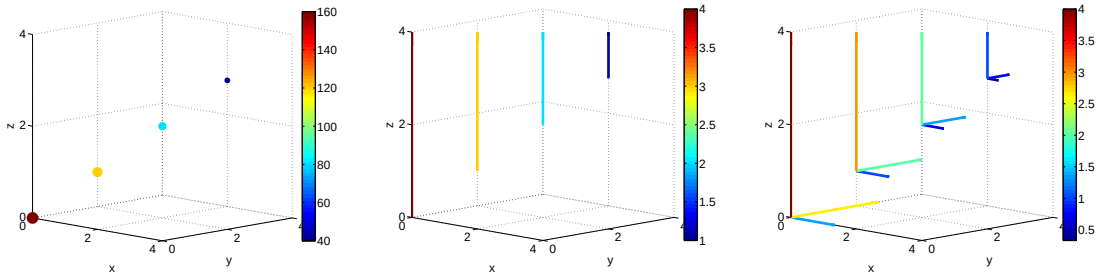
with

$$\boldsymbol{C} = \tfrac{1}{3}\,\vec{e}_x\vec{e}_x + \tfrac{2}{3}\,\vec{e}_y\vec{e}_y + \vec{e}_z\vec{e}_z$$

for which the eigenvectors of the tensors $\boldsymbol{u}_i$ are plotted and their length and colour represent the corresponding eigenvalues. Again, the following command can be used:

```
quiver(x, U, 0)
```

As before, the latter (optional) argument turns off the automatic scaling.



(a) scalar (b) vector (c) tensor

Figure 1: Different visualisation examples for different types of data.

9 Constructing and plotting finite element meshes

To facilitate the programming of finite elements, TENSORLAB also has the possibility to plot a finite element mesh using `femplot()`. The input for this function is formed by a column of nodal position vectors $\underset{\sim}{\vec{x}}$, together with the connectivity matrix. The connectivity matrix defines the node numbers associated with each element. Each row of this matrix represents a single finite element in the mesh. The numbers listed in this row are the numbers of the nodes to which the element is attached.
We illustrate the use of `femplot()` and its arguments by considering an example mesh which consists of four bi-quadratic (nine-node) elements in both $\vec{e}_x$- and $\vec{e}_y$-direction. This mesh – together with the node numbers – is shown in Figure 2. The nodal positions are given in this example by

$$\underset{\sim}{\vec{x}} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} \vec{0} \\ \frac{1}{8}\,\vec{e}_x \\ \vdots \end{bmatrix}$$

and the connectivity matrix starts as

$$\mathrm{conn} = \begin{bmatrix} 1 & 3 & 21 & 19 & 2 & 12 & 20 & 10 & 11 \\ 3 & 5 & 23 & 21 & 4 & 14 & 22 & 12 & 13 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

We can see from this definition that the node numbering per element is counter-clockwise, starting at the lower-left corner, and taking first the corner nodes, then the mid-side nodes and finally the center node. Once these matrices are defined, the mesh can be plotted using

```
femplot(x, conn)
```

In addition, we may choose to add any of the following optional arguments (in arbitrary order):

```
femplot(x, conn, 'Color', 'black', 'Nodes', 'on', ...
  'NodeNumbers', 'on', 'ElementNumbers', 'on')
```

where the default colour is blue and neither the nodes nor the node and element numbers are shown by default.

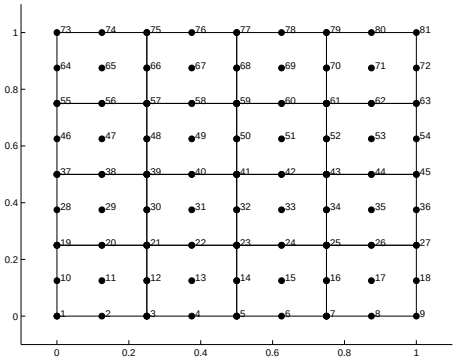


Figure 2: An example mesh consisting of four bi-quadratic finite elements in both directions.

Nodal positions and connectivity matrices for rectangular regions such as that of the example may be constructed automatically using the function `femgrid()`. The general format of a call to `femgrid()` is as follows:

```
[x, conn] = femgrid(dimensions, number_of_elements, e, ...
  'ElementType', type)
```

where `dimensions` is a scalar matrix which contains the horizontal and vertical dimensions of the rectangle, `number_of_elements` a matrix which contains the number of elements in these directions, `e` the relevant vector basis and the final couple of arguments sets the element type to be used. The latter can take the values `'Linear'` (for four-node linear elements), `'Lagrange'` (nine-node quadratic) and `'Serendipity'` (eight-node quadratic); if it is not specified, the linear four-node element is used.
The example mesh of Figure 2 has been constructed using

```
e = cartesianbasis2d;
[x, conn] = femgrid([1 1], [4 4], e, 'ElementType', ...
  'Lagrange')
```

10 Getting help

This tutorial provides a basic introduction to the TENSORLAB toolbox. When using the toolbox, one may need additional information on specific functions and operators. For this purpose, built-in help is available. For instance, to get to know more about the `dot()` product, type

```
help tensor/dot
```

To obtain a listing of all functions available for the tensor class, type

```
help tensor
```