Simplified implementation in Matlab

- After covering more general FEM implementation, we focus on the implementation of a more simplified (and limited version) in Matlab.
- The simplifications are:
  1. Fixed set and number of dof per node (*cf.* slide 410 and (448)):

$$n_{\mathrm{dpn}} := \text{ndofpn is a fixed number}$$

  2. No physics, fields, components hierarchy: As ndofpn (fixed number of dof per node) exists for the problem, we do not need to maintain the objects physics, fields, and components in elements.
  3. Limited sources of force: Natural and Essential boundary conditions and concentrated forces are the only forces.
- Also, we very briefly discuss the options to implement the FEM solver without actually resorting to object-oriented approaches.

Simplified objects: 1. Element

Given the simplifications in this section, the element object described from slide 392 to 394 can be fully described by:

- neNodes (n_{n}^e): Number of element nodes (*e.g.*, for element 1 $n_{\mathrm{n}}^e = 4$).
- eNodes (LEM): Indices of element nodes in global system; *e.g.*, for e_1:
- number of element dof (nedof: n_{f}^e): In fact, (nedof $=$ neNodes $\times$ ndofpn).
- edofs ($\mathbf{a}^e$): n_{f}^e vector of dofs.
- dofMap ($\mathbf{M}_t^e$): map from element to global dofs.
- stiffness matrix (ke: $\mathbf{k}^e$): $n_{\mathrm{f}}^e \times n_{\mathrm{f}}^e$ local stiffness (conductivity, *etc.*) matrix.
- force vectors (fde, foe, fee: $\mathbf{f}_D^e, \mathbf{f}_o^e, \mathbf{f}_e^e$): Essential BC, forces other force, and total force vectors. For term project, $\mathbf{f}_o^e = 0$, so $\mathbf{f}_D^e$ and $\mathbf{f}_e^e$ or even one of them are sufficient.
- eType: One Digit encapsulates element type. Let's adapt the following convention:

$$\text{eType } = 1 : \text{bar element} \qquad \text{eType } = 2 : \text{beam element} \tag{451}$$

$$\text{eType } = 3 : \text{truss element} \qquad \text{eType } = 4 : \text{frame element} \tag{452}$$

$$\text{eType } = 5 : \text{2D thermal element} \qquad \text{eType } = 6 : \text{2D solid element} \tag{453}$$

Simplified objects: 2. (Combined) Node + Dof

Again from our simplifications, the node and dof objects on slides 397 and 398 can be combined into,

- coordinate: a vector of size dimension of node coordinates.
- prescribed (dof_p) vector of size ndofpn having dof booleans of being prescribed dof.
- position (dof_pos) vector of size ndofpn having dof positions.
- value (dof_v) vector of size ndofpn having dof values.
- force (dof_f) vector of size ndofpn having dof forces.

Notable simplifications are:

1. All members starting with "dof_" are all that was used to be inside the set ({dof}) as a member of node object (*cf.* to {ndof} on slide 397).
2. Node's member {nndof} is no longer required as it is always equal to ndofpn.

Simplified objects: 3. FEM Solver

Following the same trend, FEM Solver on slide 401 is simplified to :

- dim, n_{dim} spatial dimension for the problem (1D, 2D, and 3D)
- number of nodes (nNodes, n_{n}).
- nodes {node}: vector of nodes in the domain.
- number of elements (ne, n_{e}) in the the domain.
- elements {element}: vector of elements in the domain.
- free dofs (dofs: $\mathbf{a}$): vector of global free dofs.
- number of free dof (nf, n_{f}).
- number of prescribed dof (np, n_{p}).
- number of dof (ndof, $n_{\mathrm{dof}} = n_{\mathrm{f}} + n_{\mathrm{p}}$).
- stiffness matrix ($\mathbf{K} = \mathbf{K}_{ff}$); $n_{\mathrm{f}} \times n_{\mathrm{f}}$ matrix.
- force vector ($\mathbf{F} = \mathbf{F}_{f}$); $n_{\mathrm{f}}(\times 1)$ vector.
- prescribed force vector ($\mathbf{F}_{p}$); $n_{\mathrm{p}}(\times 1)$ vector.
- number of materials: nmats.
- material database {mats}: a nmat size cell(Matlab) of vectors.

Object-oriented language

- Among many advantages of objected-oriented language we observe that objects encapsulate data.
- For example, for ne element, we only need a set (cell in Matlab) of elements. A different approach is not to use classes and represent "elements" by several vectors all of size ne.
- If we do not want to use objects (classes) we can do the following in Matlab:
  1. FEMSolver: A Function (*e.g.*, FEMSolver function) has all members listed on slide 447 as its internal parameters.
  2. nodes: Alternative to {node}, where "node" refers to node object on slide 446:
     - node_coordinate: Matrix (nNodes $\times$ dim) of node coordinates.
     - prescribed (node_dof_p): Matrix (nNodes $\times$ ndofpn) of node prescribed dof flags.
     - position (node_dof_pos): Matrix (nNodes $\times$ ndofpn) of node dof positions.
     - value (node_dof_v): Matrix (nNodes $\times$ ndofpn) of node dof values.
     - force (node_dof_f): Matrix (nNodes $\times$ ndofpn) of node dof forces.

Alternative to object-oriented Programming: Solver, Nodes

- Among many advantages of objected-oriented language we observe that objects encapsulate data.
- For example, for ne element, we only need a set (cell in Matlab) of elements. A different approach is not to use classes and represent "elements" by several vectors all of size ne.
- If we do not want to use classes(objects) we can do the following in Matlab:
  1. FEMSolver: A Function (*e.g.*, FEMSolver function) has all members listed on slide 447 as its internal parameters.
  2. nodes: Alternative to {node} ("node" refers to node object on slide 446):
     - node_coordinate: Matrix (nNodes $\times$ dim) of node coordinates.
     - prescribed (node_dof_p): Matrix (nNodes $\times$ ndofpn) of node prescribed dof flags.
     - position (node_dof_pos): Matrix (nNodes $\times$ ndofpn) of node dof positions.
     - value (node_dof_v): Matrix (nNodes $\times$ ndofpn) of node dof values.
     - force (node_dof_f): Matrix (nNodes $\times$ ndofpn) of node dof forces.

Alternative to object-oriented Programming: Elements

3. elements: Alternative to {element} (set of element object; *cf.* slide 445):
   - element_neNodes (n_{n}^e): ne vector; element_neNodes(en) = number of nodes for element en.
   - element_eNodes (LEM): ne cell; element_eNodes{en} = vector of size element_neNodes(en) of element en node ids.
   - element_nedof (n_{f}^e): ne vector; element_nedof(en) = number of dofs (= neNodes $\times$ ndofpn) for element en.
   - element_edofs ($\mathbf{a}^e$): ne cell; element_edofs{en} = vector of size element_nedof(en) of element dofs.
   - element_dofMap ($\mathbf{M}_t^e$): ne cell; element_dofMap{en} = vector of size element_nedof(en) of element dof positions in global system (dofMap).
   - element_ke ($\mathbf{k}^e$): ne cell; element_ke{en} = element stiffness matrix of size element_nedof(en) $\times$ element_nedof(en).
   - element_fee ($\mathbf{f}_e^e$): ne cell; element_fee{en} = element total force vector of size element_nedof(en).
   - element_foe ($\mathbf{f}_o^e$): ne cell; element_foe{en} = element other forces (all but essential) vector of size element_nedof(en).
     only needed if we have "other" load types (*e.g.*, source term).
   - element_eType: : ne vector; element_eType(en) = one integer (eType) specifying the element en's type; *cf.* (451).

Alternative to object-oriented Programming: Materials

4. Material properties: Alternative to {mats} (set of material properties objects; *cf.* slide 447):
   - mat: nmats cell; nmats{en} = vector of arbitrary size that defines the material needed for corresponding elements. For the following element types the order of material parameters are given,

eType $= 1$	Bar	mat$\{en\} = [E\ A]$	(454a)
eType $= 2$	Beam	mat$\{en\} = [E\ I]$	(454b)
eType $= 3$	Truss	mat$\{en\} = [E\ A]$	(454c)
eType $= 4$	Frame	mat$\{en\} = [E\ I]$	(454d)
eType $= 5$	2D thermal	mat$\{en\} = [\kappa\ t]$	(454e)
eType $= 6$	2D solid	mat$\{en\} = [E\ \nu\ t]$	(454f)

Alternative to object-oriented Programming: Summary

- In simplified FEMSolver class there were sets of three objects: {nodes}, {elements}, {mats}.
- On slides 449, 450, and 451 respectively we discussed how to replace these members as well as FEMSolver as on object with simple Matlab vectors and cells.
- It is evident in this alternative approach, we would be dealing with larger number of data at solver level and book keeping of them can be more difficult.
- Also, we lose the notion of objects in this approach.
- Difference between vectors and cells in Matlab:

	data indexed	indexed by
vector	numbers	(i)
cell	arbitrary: objects(*e.g.*, tensors) of different types & sizes	{i}

- In Matlab, we need to use cells to store variable data. For example, on slide 450 cells are used for storing possibly variable data (*e.g.*, vectors of variable size). Same can be observed on slide 451.

Alternative to virtual functions

- We have encountered several functions that are shared among all different realizations of an object, but may have different implementations. Some examples were:
  1. Compute element stiffness matrix $\mathbf{k}^e$.
  2. Compute various forces for elements: $\mathbf{f}_r^e$, $\mathbf{f}_N^e$, and in general all forces contributing to $\mathbf{f}_o^e$.
  3. Element output functions. For example for two node bar element we only need to output the axial force, whereas for beam elements a different output is needed.
- In we are not employing an object oriented language along with polymorphism (to enable virtual functions) we can compute or operate the right routines based on element type for all examples above:

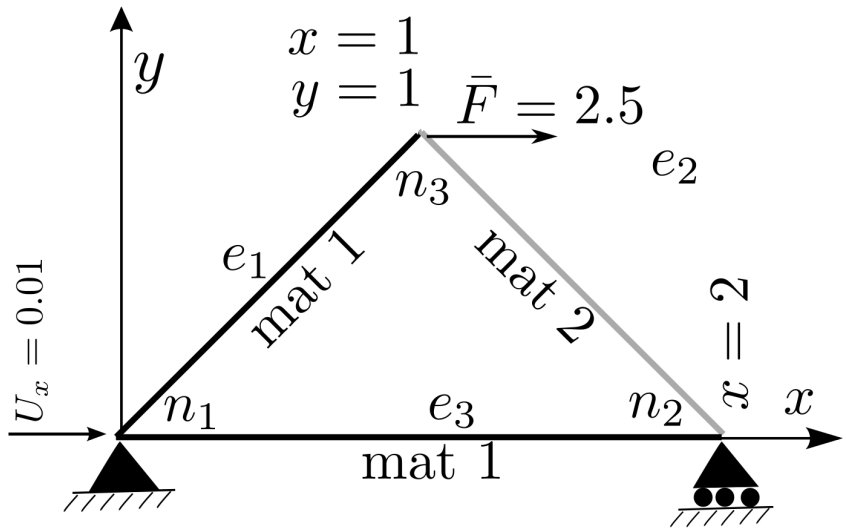
  Compute Stiffness:
  if (eType $==$ 1) % implementation to calculate ke for bar elements.
  elseif (eType $==$ 2) % ke for beam element
  $\vdots$
  end

Input file format

- Our input file has the following blocks:
  1. Header: It includes dim and ndofpn
  2. Nodes
  3. Elements
  4. Prescribed dofs
  5. Free dofs (with nonzero nodal force)
  6. Materials
- A sample input file is shown on the next page.
- We include description of all inputs (*e.g.*, id elementType matID neNodes eNodes) to make input file more readable.

Input file format

```
dim 2
ndofpn 2
Nodes
nNodes 3
id crd
1 0 0
2 2 0
3 1 1
Elements
ne 3
id elementType matID neNodes eNodes
1 3 1 2 1 3
2 3 2 2 3 2
3 3 1 2 1 2
PrescribedDOF
np 3
node node_dof_index value
1 1 0.01
1 2 0
2 2 0
FreeDOFs
nNonZeroForceFDOFs 1
node node_dof_index value
3 1 2.5
Materials
nMat 2
id numPara Paras
1 2 100 1
2 2 200 2
```



$E_1 = 100, A_1 = 1$
$E_2 = 200, A_2 = 2$

Output file format

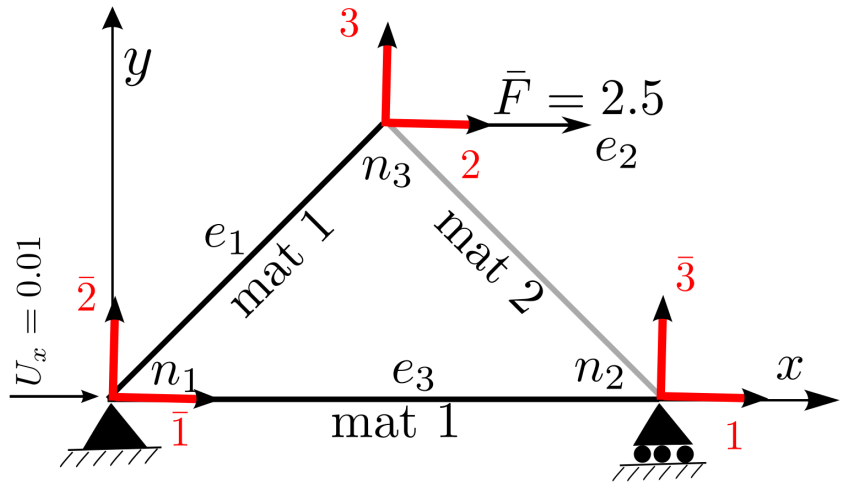
- Main FEMSolver function would be:

  function FEMSolver(runName, verboseOutput)

- runName is the name of the run (*e.g.*, TrussA).
- FEMSolver opens a file with name runName.txt to read the problem (with the format conforming to the sample on slide 457).
- This function, computes and outputs results in file runName.out.
- verboseOutput: if it is true more data will be output (explained on the next slide).
- Format of output file is
  1. Nodes: Node dof values and forces (plus dof position and prescribed boolean with verbose option).
  2. Elements: All element nodal forces (plus element specific output; *e.g.*, axial force for bars and trusses, *etc.*).

Output file format

```
Nodes
nNodes 3
id crd
values
forces
position(verbose)
prescribed_boolean(verbose)
1 0 0
a1_1 a1_2
F1_1 F1_2
-1 -2 (verbose)
1 1 (verbose)
2 2 0
a2_1 a2_2
F2_1 F2_2
1 -3 (verbose)
0 1 (verbose)
a3_1 a3_2
F3_1 F3_2
2 3 (verbose)
0 0 (verbose)
Elements
ne 3
id elementType
forces(verbose)
specific output
1 3
fee1_1 fee1_2 fee1_3 fee1_4 (verbose)
Te1
2 3
fee2_1 fee2_2 fee2_3 fee2_4 (verbose)
Te2
3 3
fee3_1 fee3_2 fee3_3 fee3_4 (verbose)
Te3
```



- lines with (verbose) are only output for verboseOutput == 1. Obviously (verbose) is not printed in either case and is only printed for clarity here.
- ai_j: is value (solution) for node i dof number j;*e.g.*, a3_1 is x displacement at node 3 ($x = 1, y = 1$).
- Fi_j: is force for node i dof number j;*e.g.*, F3_1 is x force at node 3 (which should be equal to 2.5, why?)
- feei_j: is total force (foe + fDe) for element i dof number j;*e.g.*, fee3_1 is the x force at its left node (global n_1).
- Last item of element output is specific to its type.
- For 2 node bar and truss elements Tei is the axial force in the element.

Programming hints for Matlab

- In Matlab input arguments can only be passed to functions by copy.
- That is, Matlab does not provide the option of sending arguments by reference.
- One approach to limit the number of arguments passed to functions is to have some of input arguments global in the function and from where the function is called.
- This not only reduces memory overhead by argument copy (first item) but also reduced the number of input arguments which are generally much larger if classes are not employed (*e.g.*, compare sending an element object to a function versus all its internal data through arguments on slide 450).
- As mentioned before, If classed are not employed various data (*e.g.*, those pertained to elements) should be stored in cells and Matlab vectors (or matrices) are not appropriate.
- Finally, functions that depend on the type of element can have internal "if else or switch" statements based on elementType, instead of streamlined polymorphism (virtual function) aspect offered by object oriented languages.