

YYT-C3002

Application Programming in Engineering

Spring 2019, period III, 5 credits (BSc)

Topic 7

Matlab programming for finite element methods (FEM)

Jarkko Niiranen
Assistant Professor

Department of Civil Engineering
School of Engineering
Aalto University

Lecture 10–12, Exercise 12–14, Tuesday, January 29, 2019

Notes on the course material for topic 7

The lecture material contains *extra material* (clearly indicated) which can be skipped.

The exercise material contains (1) assignments for independent studies and (2) assignments accomplished in the exercise class as guided tours:

- Theory exercise 7.1 (independent reading task)
- Theory exercise 7.2 (independent task with guidelines and hints given in the lecture session)
- ✓ Computer exercise 7.1 (accomplished in the exercise session)
- ✓ Computer exercise 7.2 (accomplished in the exercise session)
- Computer exercise 7.3 (independent hand-calculation and programming tasks)
- Computer exercise 7.4 (independent programming task with guidelines and hints given in the exercise session)
- Computer exercise 7.5 (web tutorial with step-by-step guidelines)

The total workload of a one-period course (6 weeks plus an examination week, 133 hours, 5 cr) is divided by six weeks roughly as follows (20 hours per week):

- lectures ($2 \times 2 = 4$ hours per week)
- self studies for the lecture material ($2 \times 2 = 4$ hours per week)
- exercise classes ($2 \times 2 = 4$ hours per week)
- self studies for home assignments ($2 \times 4 = 8$ hours per week)

The leftover is dedicated to the seventh week: final examination (3 hours) and preparation for the examination (10 hours).



**A
MOTIVATION
TO**

computational engineering

Motivation to computational engineering

The world is full of *data* – think about

- World Wide Web (www)
- Facebook (Fb)
- Internet of Things (IoT).



But which kind of data? Most often, quite simple data – think about

- www: words, pictures, videos,...
- Fb: likes for words, pictures, videos,...
- IoT: sensor values for coordinates, temperatures, hours,...



Data scientists/analysts collect, aggregate and analyze data – in order to form information and models for design and decisions making – by developing and utilizing *algorithms* (by means of *mathematics*, *computer science* and the “laws” of applications fields (such as population growth models or economic models).

Data science utilizes the classical tools of *statistics* and *stochastics* (for analyzing the political and economical behavior of people, for instance) and the modern trends of *data mining* (DM), *machine learning* (ML), *artificial intelligence* (AI) etc.

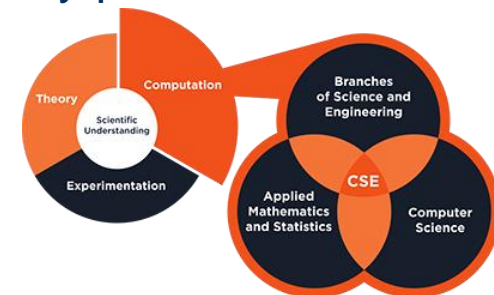
Motivation to computational engineering

The tools of data analysis are – definitely – demanding and scientific but the nature and origin of data (vote, like, purchase) is often very simple.

In *computational engineering*, most often, data originates from and is related to complex physical systems (such as buildings or machines) and the "laws" of the application field are quite complex (such as structural or fluid dynamics).

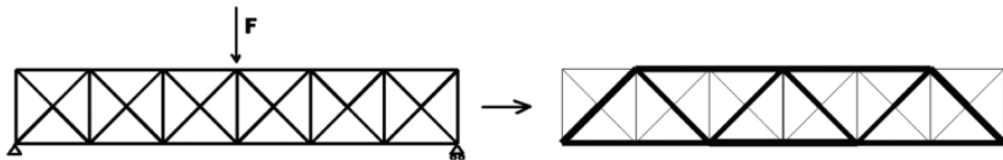
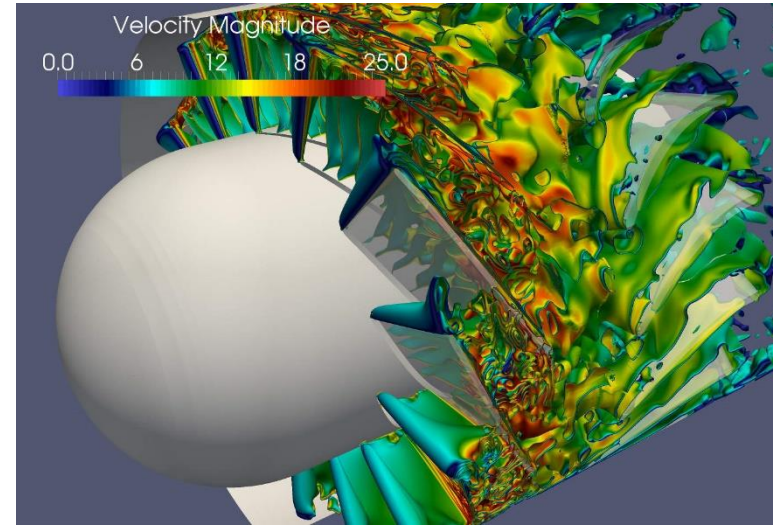
Actually, computational engineering (such as *computational mechanics* or *computational fluid dynamics*) is in large extent different than other *computational sciences* (such as *data science* or even *computational modeling* such as *geometric modeling*) although the same components and terms are typically present:

- modeling and simulation
- data structures and algorithms
- data analysis and visualization
- high-performance computing

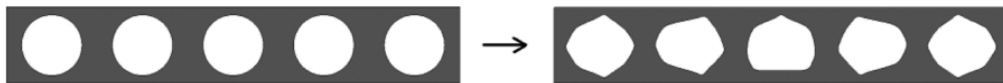


“The computational engineer uses the computer and mathematical algorithms to solve *physics-based equations* to make predictions and simulate scenarios.”

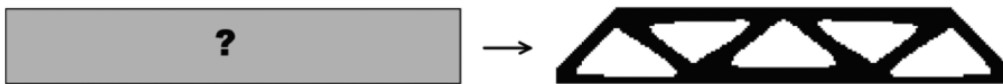
Motivation to computational engineering



Sizing optimization



Shape optimization



Topology optimization

Equation

Equation form:

Study controlled

Show equation assuming:

Study 1, Time Dependent

$$\rho \frac{\partial^2 \mathbf{u}}{\partial t^2} - \nabla \cdot \boldsymbol{\sigma} = \mathbf{F}_v$$

$$\rho C_p \frac{\partial T}{\partial t} + \rho C_p \mathbf{u} \cdot \nabla T = \nabla \cdot (k \nabla T) + Q$$

Last Time
Weighted Residual Formulations

Consider a general representation of a governing equation on a region V

$$L u = P$$

L is a differential operator

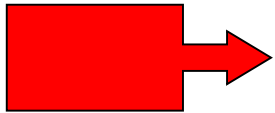
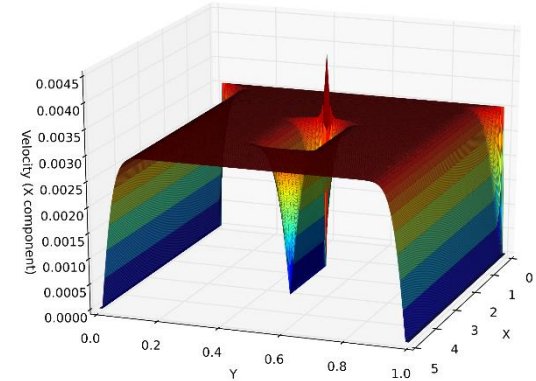
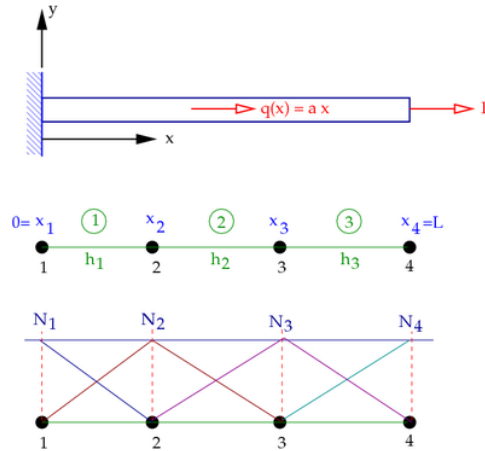
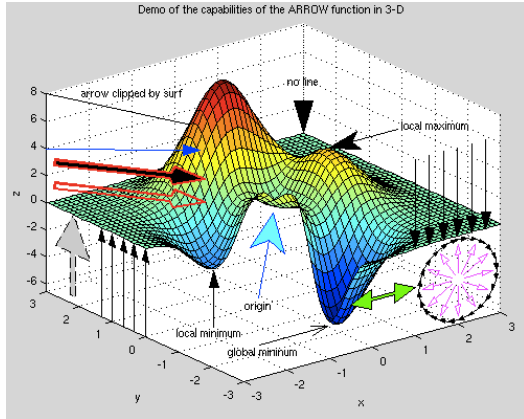
eg. For Axial element $\frac{d}{dx} \left(E A \frac{du}{dx} \right) = 0$

$$L = \frac{d}{dx} E A \frac{d}{dx} ()$$

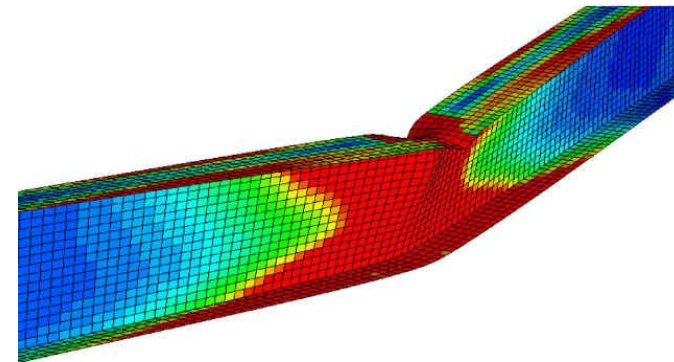
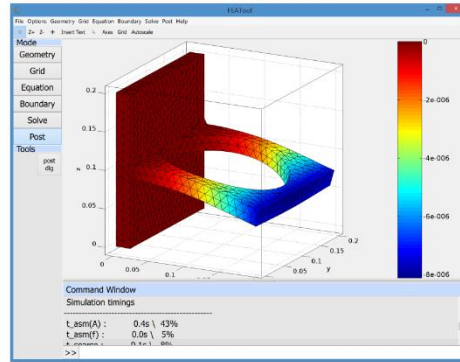
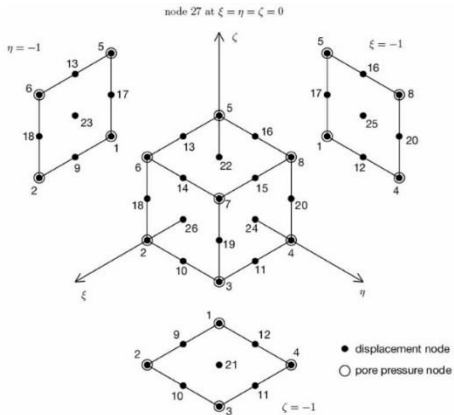
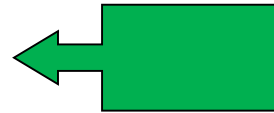

**BACK
TO**

basic material

7.0 Familiar with Matlab or finite element methods?



How much have you used Matlab
– how about FEM?



7 Matlab programming for finite element methods (FEM)

Contents

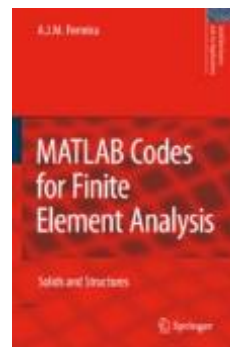
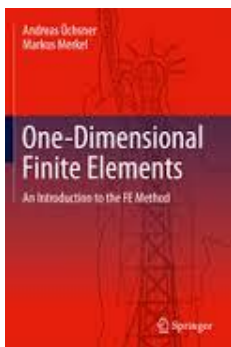
1. *General Matlab features for computational engineering*
2. *Strong form and weak form for 1D and 2D model problems*
3. *Finite element formulations for 1D and 2D model problems*



Learning outcome

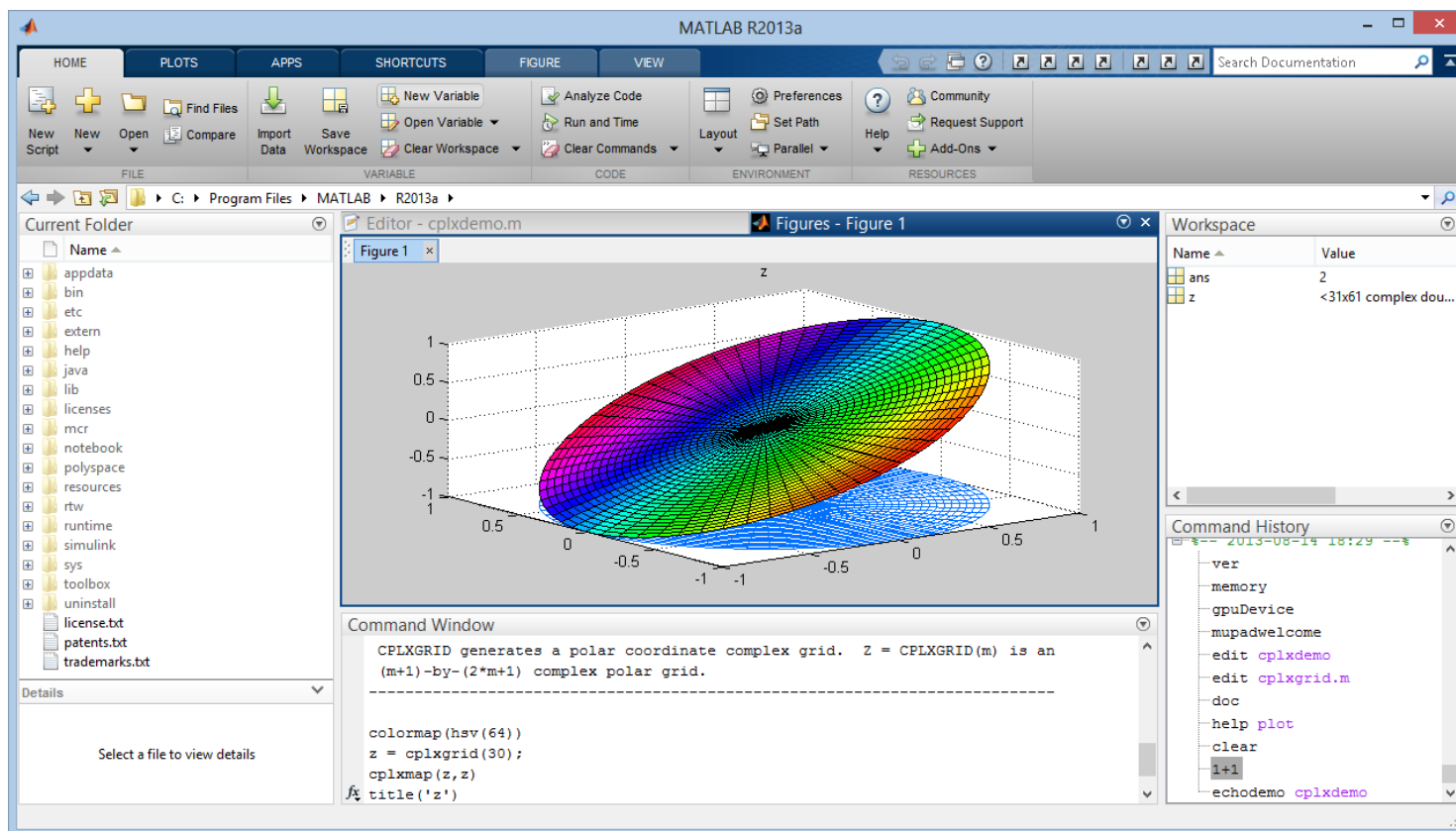
- A. *Understanding of the main principles of Matlab programming*
- B. *Ability to apply Matlab features to 1D and 2D finite element model problems*

References



7.1 General Matlab features for computational engineering

MATLAB (*MATrix LABoratory*) is a (technical) computing (easy-to-use) environment for high-performance numeric computation and visualization based on MATLAB programming language.



7.1 General Matlab features for computational engineering

MATLAB (*MATrix LABoratory*) is a (technical) computing (easy-to-use) environment for high-performance numeric computation and visualization based on MATLAB programming language.

In MATLAB, problems and solutions are expressed in familiar mathematical notation.

```
>> y = cos(x)
```

MATLAB actually is all of the following things:

- Language
- Working environment
- Graphics tools
- Mathematical function library
- Application Program Interface (API)



Remark. The **Octave** language is quite similar to Matlab so that most programs are easily portable. Octave is distributed under the terms of the [GNU General Public License](#).



7.1 General Matlab features for computational engineering

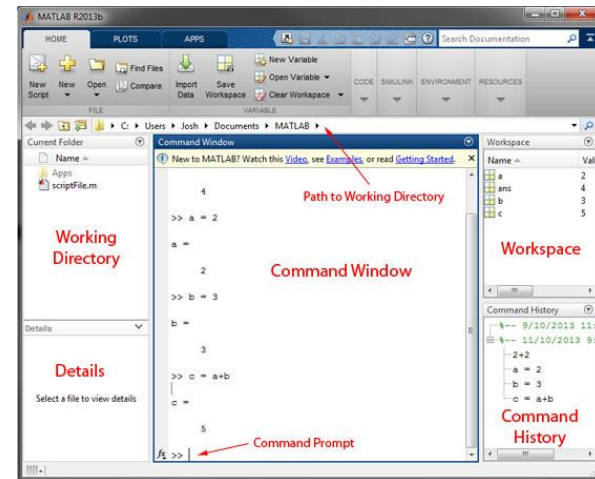
MATLAB language:

- a high-level matrix/array language
- with control flow statements, functions, data structures, input/output, and object-oriented programming features
- allows programming in small and large scale (memory and processor time)

```
a=1;
if a==1
    disp('Is 1')
elseif a==2
    disp('Is 2')
else
    disp('I don''t know')
end
```

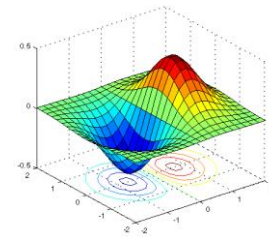
MATLAB working environment:

- a set of tools and facilities managing the variables in your workspace
- importing and exporting data
- tools for developing, managing, debugging, and profiling M-files (applications)



MATLAB Graphics tools:

- high-level commands for two-dimensional and three-dimensional data visualization, image processing, animation, and presentation graphics
- low-level commands for customizing the appearance of graphics
- allows building Graphical User Interfaces (GUI) on applications



7.1 General Matlab features for computational engineering

The MATLAB mathematical function library:

- a vast collection of computational **algorithms**
- **elementary functions** like sum, sine, cosine, and complex arithmetic
- more **sophisticated functions** like matrix inverse, matrix eigenvalues, Bessel functions, and fast Fourier transforms

cos(x)	Cosine	abs(x)	Absolute value
sin(x)	Sine	sign(x)	Signum function
tan(x)	Tangent	max(x)	Maximum value
acos(x)	Arc cosine	min(x)	Minimum value
asin(x)	Arc sine	ceil(x)	Round towards $+\infty$
atan(x)	Arc tangent	floor(x)	Round towards $-\infty$
exp(x)	Exponential	round(x)	Round to nearest integer
sqrt(x)	Square root	rem(x)	Remainder after division
log(x)	Natural logarithm	angle(x)	Phase angle
log10(x)	Common logarithm	conj(x)	Complex conjugate

The MATLAB Application Program Interface (API):

- library allowing one to write **C** and **Fortran** programs that interact with MATLAB (call C, C++, or Fortran programs, defined as **MEX-files**, from the MATLAB command line as if they were built-in functions)
- facilities for **calling routines** from MATLAB (dynamic linking)
- facilities for calling MATLAB as a computational engine facilities for reading and writing **MAT-files** (which allows one to access and change variables directly in a MAT-file, without having to load the variables into memory)



HISTORY

extra material

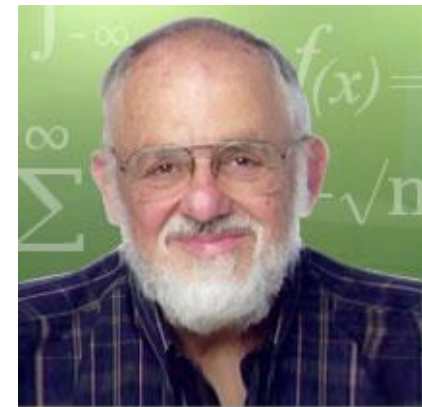
7.1 General Matlab features for computational engineering

MATLAB was originally written to provide easy access to matrix software developed by the **LINPACK** and **EISPACK** projects, which together represent the state-of-the-art in software for matrix computation.

Matlab History

In the 1970's, Cleve Moler "Professor of Math & Computer Science, Chief Author of MatLab and one of the Founders of Mathworks.Inc" participated in developing (**EISPACK**) and (**LINPACK**). Those were collection of Fortran subroutines for solving linear equations and Eigen value problems.

Later, when teaching courses in mathematics, Moler wanted his students to be able to use LINPACK and EISPACK without requiring knowledge of Fortran, so he developed the first MATLAB in 1977 as an interactive system to access LINPACK and EISPACK.



7.1 General Matlab features for computational engineering

- **LINPACK** is a software library for performing *numerical linear algebra* on digital computers. It was written in Fortran by Jack Dongarra, Jim Bunch, Cleve Moler, and Gilbert Stewart, and was intended for use on *supercomputers* in the 1970s and early 1980s.
- **EISPACK**, written in Fortran as well, is a software library for numerical computation of *eigenvalues* and *eigenvectors* of matrices.
- Both packages originated from [Argonne National Laboratory](#), has always been free, and aims to be *portable*, *robust* and *reliable*.

EISPACK and LINPACK

◆ EISPACK

- Design for the algebraic eigenvalue problem,
 $Ax = \lambda x$ and $Ax = \lambda Bx$.
- work of J. Wilkinson and colleagues in the 70's.
- Fortran 77 software based on translation of ALGOL.

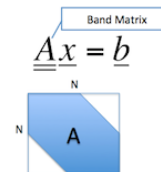
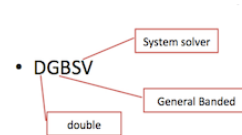
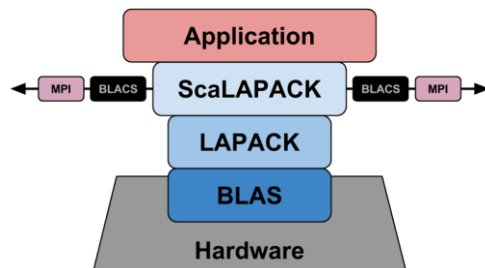
◆ LINPACK

- Design for the solving systems of equations,
 $Ax = b$.
- Fortran 77 software using the Level 1 BLAS.



7.1 General Matlab features for computational engineering

- **LAPACK** (Linear Algebra Package) is a standard software library for *numerical linear algebra*.
- The original goal of the LAPACK project was to make the widely used EISPACK and LINPACK libraries run efficiently on **shared-memory vector** and **parallel processors**.
- LAPACK routines are written so that as much as possible of the computation is performed by calls to the **Basic Linear Algebra Subprograms (BLAS)**.
- The LAPACK project has been sponsored in part by [MathWorks](#) and [Intel](#).



LAPACK

- ◆ **Linear Algebra library in Fortran 77 (binding to c)**
 - **State of the art numerical routines**
 - **Extensive coverage**
 - **Solution of systems of equations**
 - **Solution of eigenvalue problems**
- ◆ **Block algorithms**
 - **Parameterized for memory hierarchies**
 - » Built on the Level 1, 2, and 3 BLAS
 - **Efficient on a wide range of computers**
 - » RISC, Vector, SMPs
- ◆ **User interface provides similar calls in:**
 - **Single, Double, Complex, Double Complex**
- ◆ **Used by vendors: HP-48G to Teraflop/s Machines**

**BACK
TO**

basic material

7.1 General Matlab features for computational engineering

MATLAB fundamentals

In MATLAB, every object is a *complex matrix* in which real entries are displayed as real and integer as integer (numbers are 1×1 matrices).

Variables.

- no need to (cannot) declare in advance
- no need to specify type
- can switch from one type to another
- assign value with '='

To define a *complex number*, use either i or j:

```
>> c=2+i*7  
c = 2.0000 + 7.0000i  
>> d=(1+j*3)^3  
d = -26.0000 -18.0000i
```

To access the real and imaginary parts, use the commands **real** and **imag**.

7.1 General Matlab features for computational engineering

Vectors.

A *vector* of equispaced elements can be generated using the general format:

```
[{beginning number} : {step increment} : {last number}]
```

If the step is 1 then it can be omitted:

```
[{beginning number} : {last number}]
```

To define a row or column vector, respectively, with three entries:

```
>> x=[10 20 30]
```

```
x =
```

```
    10    20    30
```

```
>> x=[10; 20; 30]
```

```
x =
```

```
    10
```

```
    20
```

```
    30
```

You can define a negative incremental step size, if the beginning number is smaller than the last number.

7.1 General Matlab features for computational engineering

Matrices.

To define an $n \times m$ *matrix* A:

```
>> A=[1 2 3; 4 5 6]
```

```
A =
```

```
     1     2     3
     4     5     6
```

To find the size (i.e, the number of rows and columns) of a matrix:

```
>> [n,m]=size(A);
```

Operations *matrix algebra* can be defined in a natural way (with proper dimensions):

```
>> A*B;
```

```
>> A-B;
```

```
>> A+B;
```

Many other commands exist too:

```
>> transpose(A); A';
```

```
>> inv(A)
```

```
>> det(A)
```

```
>> eig(A)
```

7.1 General Matlab features for computational engineering

For defining submatrices:

- the i th row is $A(i,:)$
- the j th column is $A(:,j)$
- $A(i:j,p:q)$ gives a part of the matrix A (with $i < j < n$, $p < q < m$)
- $A([i \ k \ j],[p \ q])$ gives a part of the matrix A (with $i,j,k < n$, $p,q < m$)

Try the following commands:

```
>> B=eye(3)
>> C=ones(2,3)
>> D=diag([1 5 6 8])
>> E=zeros(3,2)
>> F=rand(1,5)
>> G=randn(5,1)
```

In many applications, as in finite element methods, system matrices are *sparse matrices*, or even *band matrices*. Therefore, for efficiency, one needs to define them in the following form:

```
>> A = sparse(n,m)
```

7.1 General Matlab features for computational engineering

Hadamard (or “dot”) operations.

Hadamard multiply .* works as

$C=A.*B$ has entries $c(i,j)=a(i,j)b(i,j)$

```
>> A = [2 2; 2 2];
```

```
>> C = A.*A
```

C =

```
     4     4
     4     4
```

Hadamard divide ./ works as

$C=A./B$ has entries $c(i,j)=a(i,j)/b(i,j)$

$C=A.\backslash B$ has entries $c(i,j)=b(i,j)/a(i,j)$

Hadamard exponentiate .^ works as

$C=A.^B$ has entries $c(i,j)=a(i,j)^{b(i,j)}$

$C=A.^r$ has entries $c(i,j)=a(i,j)^r$, where r is a number.

$C=r.^A$ has entries $c(i,j)=r^{a(i,j)}$

7.1 General Matlab features for computational engineering

Vectorization.

Many tasks that may be implemented with loop structures can be more efficiently executed with **vectorization**:

Let us Generate a signal function $y = \sin(x)^2$:

```
>> x=0:pi/12:4*pi;  
>> y=sin(x).^2;
```

Let's view the signal function pointwise:

```
>> stem(x,y)
```

Let's label the axis and give a title for the figure:

```
>> xlabel('x'); ylabel('y'); title('Signal Function')
```

A. Let's calculate the average signal value by a for-loop:

```
>> S=0;  
>> for k=1:length(y)  
    S=S+y(k);  
end  
avg=S/length(y)
```

B. Let's calculate the average signal value more efficiently by "vectorizing":

```
>> avg=sum(y)/length(y)
```


7.1 General Matlab features for computational engineering

Printing to screen.

1. Type the variable or expression without semicolon.

```
>> x = 0:0.2:1
```

```
x =
```

```
    0    0.2000    0.4000    0.6000    0.8000    1.0000
```

2. Use the **disp** function.

```
>> disp(x)
```

3. Use **fprintf** (sends output to screen or a file).

```
>> fprintf(1, 'value of x is %7.1f \n', x)
```

```
value of x is    0.0
```

```
value of x is    0.2
```

```
value of x is    0.4
```

```
value of x is    0.6
```

```
value of x is    0.8
```

```
value of x is    1.0
```

4. Use **sprintf** (sends output to a string variable) .

7.1 General Matlab features for computational engineering

File import/export.

Variables can be saved from the Matlab workspace with the **save** command:

```
>> save filename
```

The file will be a MAT-file (*.mat) and is readable only by Matlab.

To import the variables use the **load** command:

```
>> load filename
```

The file must be in Matlab's path (see the **path** command) or the current directory should be set to the directory containing the file (see the M-file help).

For importing data from files, see a list of commands used for io:

```
>> help iofun
```

These low-level commands deal with ASCII files, etc.

You can check the current directory and see a list of available *.m and *.mat files:

```
>> pwd
```

```
>> what
```

You can change the directory the **cd** command.

7.1 General Matlab features for computational engineering

Printing to a file.

Open a file, write to it, close the file:

```
>> fid = fopen(filename, 'w')
>> fprintf(fid, 'string %s and integer %d\n', str, int)
>> fclose(fid)
```

Create a text file called exp.txt containing a short table of the exponential function.

```
>> x = 0:0.2:1;
>> y = [x; exp(x)];
>> fid = fopen('exp.txt','w');
>> fprintf(fid,'%6.2f %12.8f\n',y);
>> fclose(fid);
```

Examine the contents of exp.txt:

```
>> type exp.txt
0.00      1.00000000
0.20      1.22140276
0.40      1.49182470
0.60      1.82211880
0.80      2.22554093
1.00      2.71828183
```

7.1 General Matlab features for computational engineering

Operators.

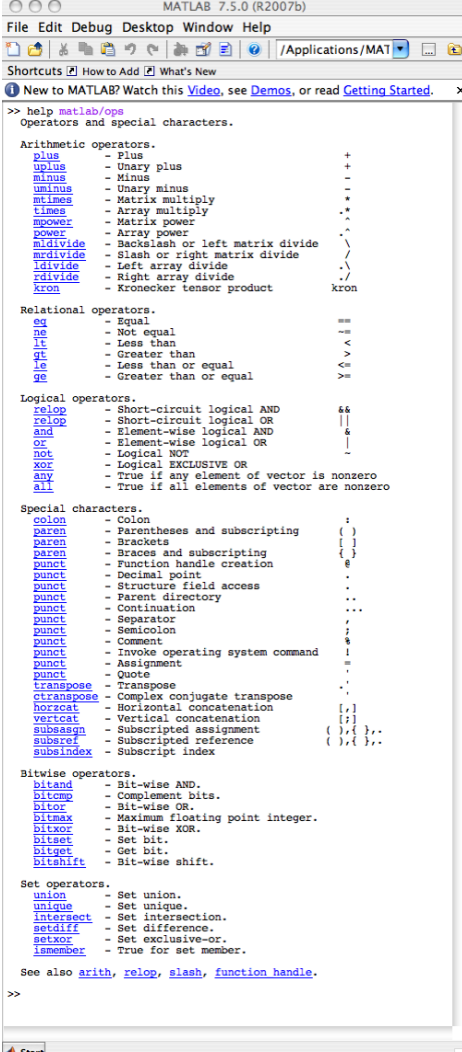
There are lots of mathematical operators defined: just type **help matlab/ops** for a list of the basic operators.

Built-in functions.

There are lots and lots and lots of mathematical built-in functions: just type **help matlab/elfun** for the list of functions.

Efficient coding requires using operators and built-in functions.

cos(x)	Cosine	abs(x)	Absolute value
sin(x)	Sine	sign(x)	Signum function
tan(x)	Tangent	max(x)	Maximum value
acos(x)	Arc cosine	min(x)	Minimum value
asin(x)	Arc sine	ceil(x)	Round towards $+\infty$
atan(x)	Arc tangent	floor(x)	Round towards $-\infty$
exp(x)	Exponential	round(x)	Round to nearest integer
sqrt(x)	Square root	rem(x)	Remainder after division
log(x)	Natural logarithm	angle(x)	Phase angle
log10(x)	Common logarithm	conj(x)	Complex conjugate



```
MATLAB 7.5.0 (R2007b)
File Edit Debug Desktop Window Help
/Applications/MAT
Shortcuts How to Add What's New
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
>> help matlab/ops
Operators and special characters.

Arithmetic operators.
plus      - Plus                +
uplus     - Unary plus         +
minus     - Minus              -
uminus    - Unary minus        -
mlines    - Matrix multiply    .*
lines     - Array multiply     .*
mpower    - Matrix power       ^
power     - Array power        ^
mldivide  - Backslash or left matrix divide \
mrdivide  - Slash or right matrix divide /
ldivide   - Left array divide  ./
rdivide   - Right array divide ./
kron      - Kronecker tensor product kron

Relational operators.
eq        - Equal              ==
ne        - Not equal          ~=
lt        - Less than          <
gt        - Greater than       >
le        - Less than or equal <=
ge        - Greater than or equal >=

Logical operators.
relop     - Short-circuit logical AND &&
orlop     - Short-circuit logical OR ||
and       - Element-wise logical AND &
or        - Element-wise logical OR |
not       - Logical NOT        ~
xor       - Logical EXCLUSIVE OR ^
any       - True if any element of vector is nonzero
all       - True if all elements of vector are nonzero

Special characters.
colon     - Colon              :
paren     - Parentheses and subscripting ( )
paren     - Brackets           [ ]
paren     - Braces and subscripting { }
punct     - Function handle creation @
punct     - Decimal point      .
punct     - Structure field access .
punct     - Parent directory ..
punct     - Continuation      ...
punct     - Separator          ;
punct     - Semicolon         ;
punct     - Comment           %
punct     - Invoke operating system command !
punct     - Assignment         =
punct     - Quote             '
transpose - Transpose          .'
ctranspose - Complex conjugate transpose .'
horzcat   - Horizontal concatenation [ ]
vertical  - Vertical concatenation {; }
subasgn   - Subscripted assignment ( ) { } ,
subref    - Subscripted reference ( ) { } ,
subindex  - Subscript index   ( ) { } ,

Bitwise operators.
bitand    - Bit-wise AND.
bitcmp    - Complement bits.
bitor     - Bit-wise OR.
bitmax    - Maximum floating point integer.
bitxor    - Bit-wise XOR.
bitset    - Set bit.
bitget    - Get bit.
bitshift  - Bit-wise shift.

Set operators.
union     - Set union.
unique    - Set unique.
intersect - Set intersection.
setdiff   - Set difference.
setxor    - Set exclusive-or.
ismember  - True for set member.

See also arith, relop, slash, function handle.
>>
```

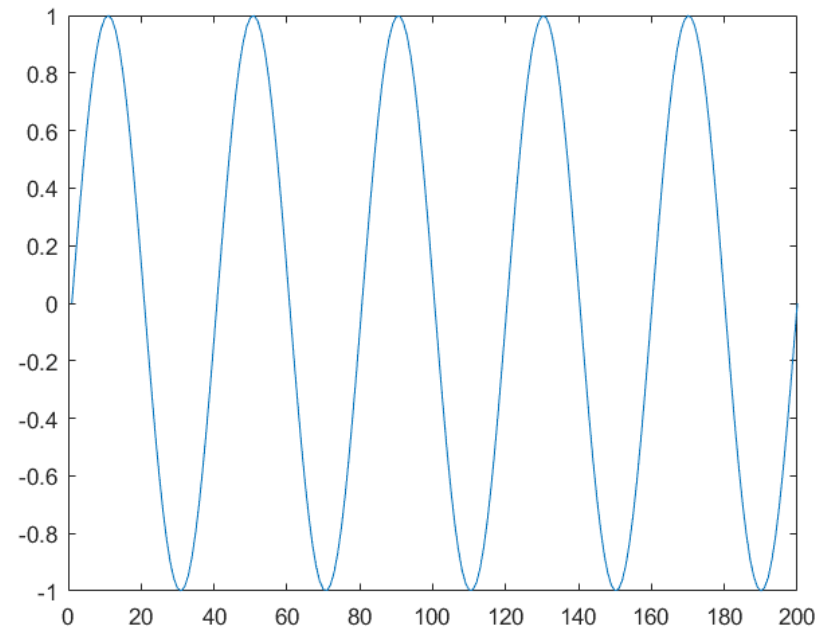
7.1 General Matlab features for computational engineering

Elementary functions are evaluated in an **element/point/entry wise** sense:

```
>> x=linspace(0,10*pi,200);  
>> y=sin(x);  
>> plot(y)
```

A list of commands worth checking out:

```
>> help conv  
>> help sum  
>> help roots  
>> help fft  
>> helpfliplr  
>> help sound  
>> help max  
>> help min  
>> help abs  
>> help length  
>> help real  
>> help for  
>> help num2str  
>> help disp  
>> help pause  
>> help whos
```



7.1 General Matlab features for computational engineering

Toolboxes.

MATLAB features a family of application-specific solutions called toolboxes – comprehensive collections of MATLAB functions (M-files).

PDEToolbox is a simple tools for simple model problems of [Partial Differential Equations](#) (PDE) solved by using simple tools of [Finite Element Methods](#) (FEM).

M-files.

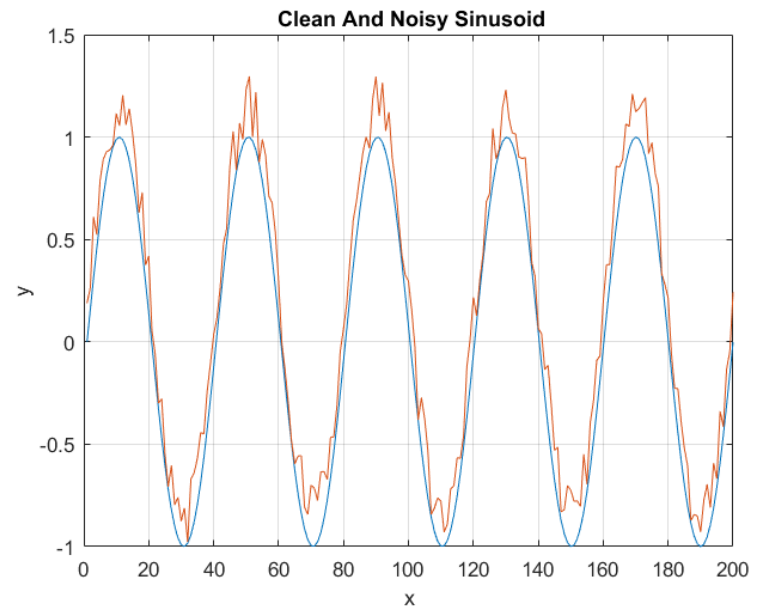
For writing your own programs, use m-files:

- use any regular ASCII text editor
- Open a file with the extension *.m
- Edit line by line the sequence of Matlab commands you want to include in your program.
- Save the file and execute the program by typing the name of the file (without .m) on the Matlab command line.

7.1 General Matlab features for computational engineering

Example program written in the file *signals.m*:

```
% This is a program that generates a clean and noisy signal
x=linspace(0,10*pi,200);
% Compute and plot the clear signal in Fig. 1
y=sin(x);
figure(1); plot(x,y); hold on;
% Compute the noise
z=0.3*rand(1,200);
% Add the noise to the signal
y=y+z
% Plot the noisy signal in Fig.1
plot(x,y); grid
% Finalize the figure
title('Clean And Noisy Sinusoid')
xlabel('x')
ylabel('y')
hold off;
```



Hint. For seeing the **time** elapsed for certain parts of your program code:

```
>> time = clock; x = A*b; pause(10); time = etime(clock,time)
>> t = cputime; x = A*b; e = cputime - t
```

7.1 General Matlab features for computational engineering

User-defined functions.

User defined functions work just like commands in Matlab.

Functions have the following format:

```
function [returned_variable_1, returned_variable_2, ...]= function_name(arugments)
```

The variables defied in a function are local and only available within the function,
see:

```
>> help function
```

Example saved in file *stat.m*:

```
function [mean,stdev] = stat(x)
n = length(x);
mean = sum(x) / n;
stdev = sqrt(sum((x - mean).^2)/n);
```

The usual programming language **statements** can be used in M-files:
for-end, if-else-break-end, while-end.

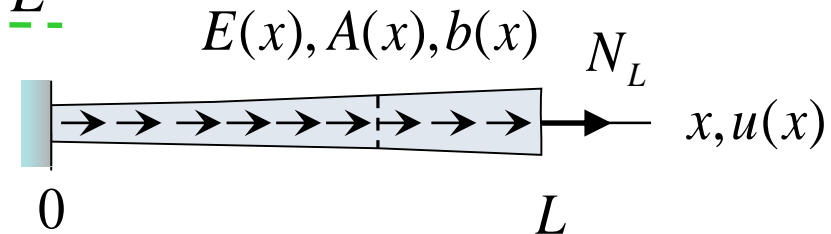
7.2 Strong forms and weak forms for 1D and 2D model problems

Strong form. The differential equation and boundary conditions for an *elastic bar/rod/column in tension/compression* read as follows: Find u such that

$$(1\text{-DE}) \quad -(\underline{EAu'})'(x) = \underline{b(x)}, \quad 0 < x < \underline{L}$$

$$(2\text{-eBC}) \quad \underline{u(0)} = \underline{u_0}$$

$$(3\text{-nBC}) \quad (\underline{EAu'}) (L) = \underline{N_L}$$



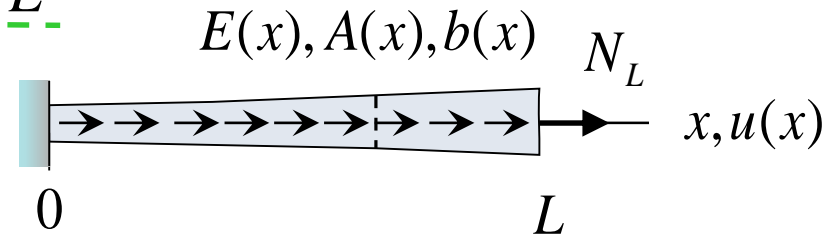
7.2 Strong forms and weak forms for 1D and 2D model problems

Strong form. The differential equation and boundary conditions for an *elastic bar/rod/column in tension/compression* read as follows: Find u such that

$$(1\text{-DE}) \quad -(\underline{EAu'})'(x) = \underline{b(x)}, \quad 0 < x < \underline{L}$$

$$(2\text{-eBC}) \quad \underline{u(0) = u_0}$$

$$(3\text{-nBC}) \quad (\underline{EAu'})(L) = \underline{N_L}$$



u axial displacement (unknown function)

E Young's modulus (given material data)

A cross-sectional area (given geometrical data)

b axial body load (given loading data)

L length (given geometrical data)

u_0 axial end point displacement (given essential/geometric boundary data)

N_L axial end point force (given natural/force boundary data).

7.2 Strong forms and weak forms for 1D and 2D model problems

Weak form. Find u such that it satisfies $u(0) = u_0$ and

$$\int_0^L EA u' v' dx = N_L v(L) + \int_0^L b v dx,$$

for all test functions v satisfying $v(0) = 0$.

Remark. Why do we formulate the problem in a weak, or variational, form?

The finite element method (FEM) is based on the weak form which actually present the problem in a form of **energy balance**:

1. the left hand side corresponds to *strain energy (stored energy, internal energy)* (the derivative of the axial displacement is the axial strain);
2. the right hand side corresponds to *loading energy (external energy)* (work done by a force equals to the product of the force and the corresponding displacement).

**HOW TO
DERIVE THE WEAK FORM?**
extra material

7.2 Strong forms and weak forms for 1D and 2D model problems

0. Start from the differential equation (1) and use the boundary conditions (2) and (3):

$$(1) \quad -(EAu')'(x) = b(x) \quad 0 < x < L$$

$$(2) \quad u(0) = u_0$$

$$(3) \quad (EAu')(L) = N_L$$

What shall we do
with the differential equation and the boundary conditions
– one page with a few lines is enough –
in order to reach the integral form below?

Do some problem solving work
for a few minutes...

$$\Rightarrow (1) \quad \int_0^L (EAu')(x) v'(x) dx = N_L v(L) + \int_0^L b(x) v(x) dx$$

$$(2) \quad u(0) = u_0$$

7.2 Strong forms and weak forms for 1D and 2D model problems

1. Multiply the differential equation (1) by a (smooth) *test function* (specified later):

$$-(EAu')'(x) = b(x) \quad \Rightarrow \quad -(EAu')'(x) v(x) = b(x) v(x), \quad 0 < x < L$$

2. Integrate over the *domain* (interval):

$$\Rightarrow \quad -\int_0^L (EAu')'(x) v(x) dx = \int_0^L b(x) v(x) dx$$

3. Integrate by parts (the left hand side) for moving one derivative from u to v :

$$\Rightarrow \quad -\underline{\underline{(EAu')(L)}} v(L) + \underline{\underline{(EAu')(0)}} v(0) + \int_0^L EAu' v' dx = \int_0^L b v dx$$

4. Utilize the *natural boundary condition* (3): $\underline{\underline{(EAu')(L)}} = N_L$

5. Set a zero *essential boundary condition* (2) for the test function: $\underline{\underline{v(0)}} = 0$

$$\Rightarrow \quad \int_0^L EAu' v' dx = N_L v(L) + \int_0^L b v dx$$

7.2 Strong forms and weak forms for 1D and 2D model problems

Weak form. Find u such that it satisfies $u(0) = u_0$ and

$$\int_0^L EAu' v' dx = N_L v(L) + \int_0^L b v dx,$$

for all v satisfying $v(0) = 0$.

Remark. Note that the solution and the test function, respectively, have to satisfy the **boundary conditions** $u(0) = u_0$, $v(0) = 0$ and, in principle, **regularity conditions** as well:

$$\int_0^L (u')^2 dx < \infty, \quad \int_0^L (v')^2 dx < \infty.$$

Then the solution and the test function are called ***kinematically admissible***.

Remark. By starting from the weak form, we could correspondingly derive the strong form (integrating by parts "backwards").

**BACK
TO**

basic material

7.2 Strong forms and weak forms for 1D and 2D model problems

Generalization of the 1D bar extension weak form to 1D heat diffusion:

$$\int_0^L EA u' v' dx = \int_0^L b v dx \quad \rightarrow \quad \int_0^L k T' v' dx = \int_0^L f v dx$$

EA, b, L

•

$u = u_0 \quad \Omega$

k, f, L

•

$T = T_0 \quad \Omega$

Generalizations of the 1D heat diffusion weak form to 2D heat diffusion:

$$\int_0^L k T' v' dx = \int_0^L f v dx \quad \rightarrow \quad \int_{\Omega} k \nabla T \cdot \nabla v d\Omega = \int_{\Omega} f v d\Omega$$

k, f, L

•

$T = T_0 \quad \Omega$

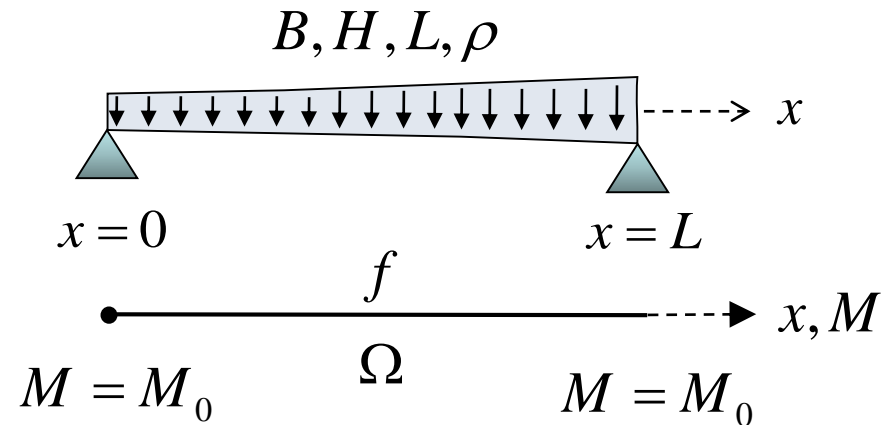
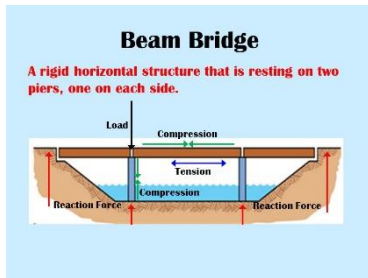
$T = T_0$

$\Omega \quad k, f$

A
MOTIVATION
FROM
exercises...

Home exercise 7.2

Let us consider a vertically gravity-loaded (statically determined) *beam* with *bending moment* M as the primary variable:



Formulate the *weak form* of the problem (serving as a basis for FE formulations).

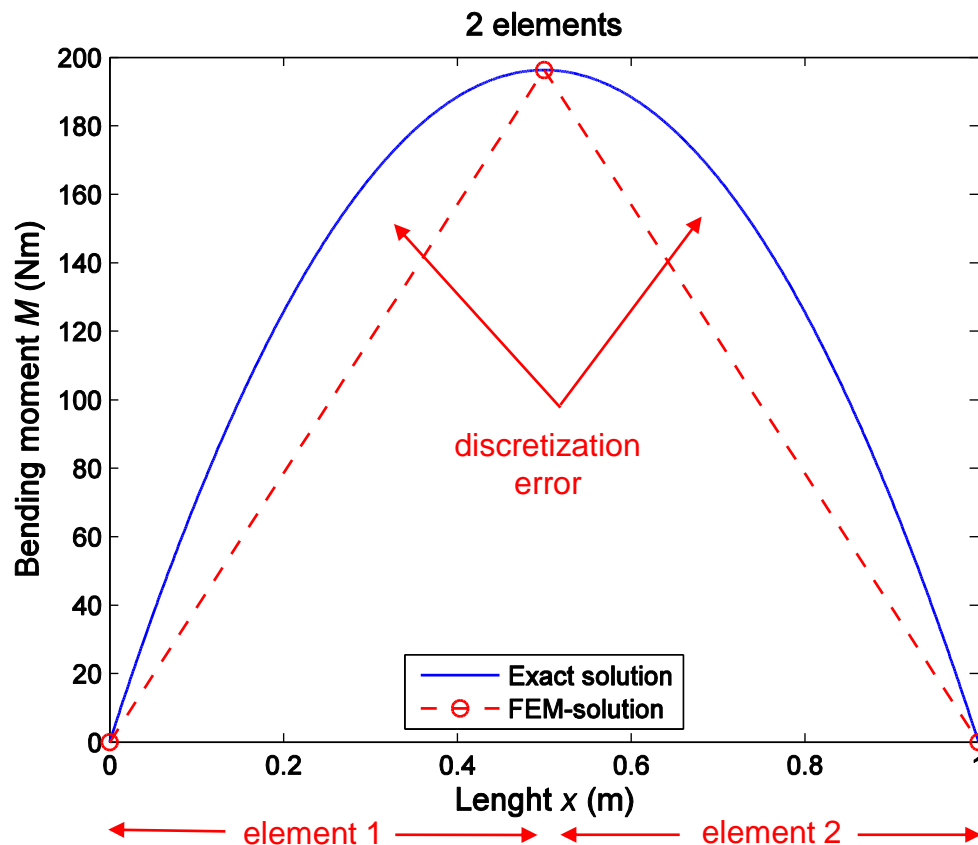
Hint: First, find out the strong form, i.e., the differential equation and boundary conditions, of the problem by recalling your previous studies (or Wikipedia).

Since the strong form of the problem is analogous to the one of the bar (or the heat diffusion) problem, you can simply imitate the weak form of the bar problem (or the heat diffusion problem).



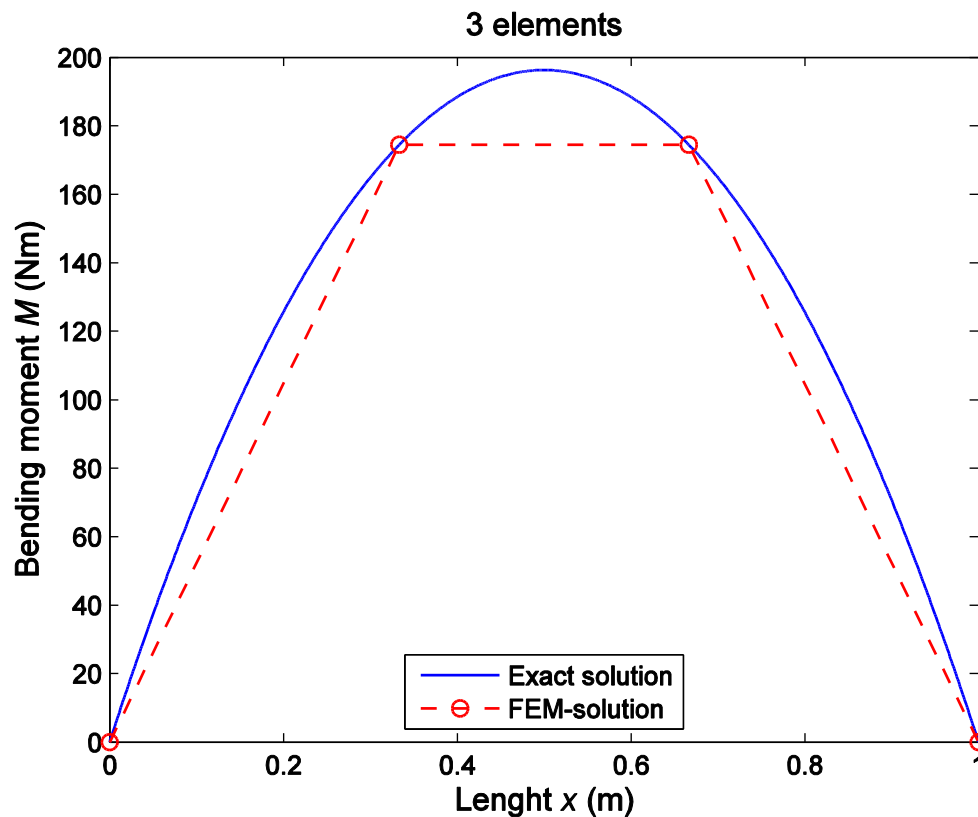
Possible computer exercise – Matlab

(i) Implement the **finite element method** with **linear basis functions** for the vertically loaded beam in **MATLAB** with the following initial data: ...



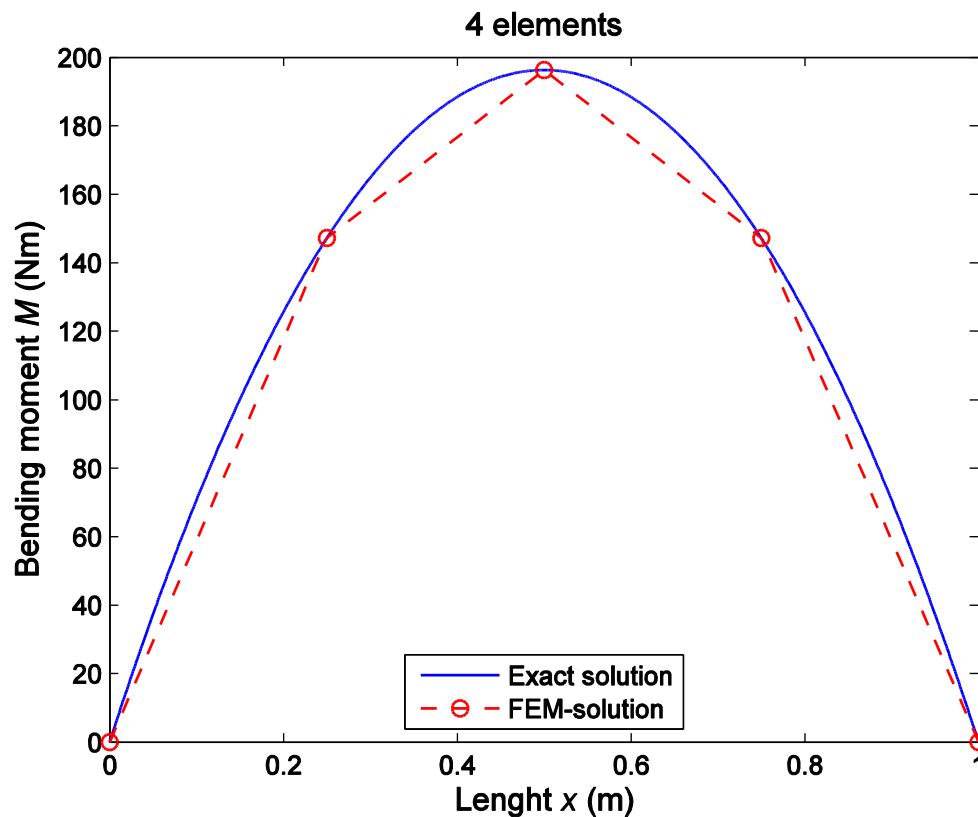
Possible computer exercise – Matlab

(i) Implement the **finite element method** with **linear basis functions** for the vertically loaded beam in **MATLAB** with the following initial data: ...



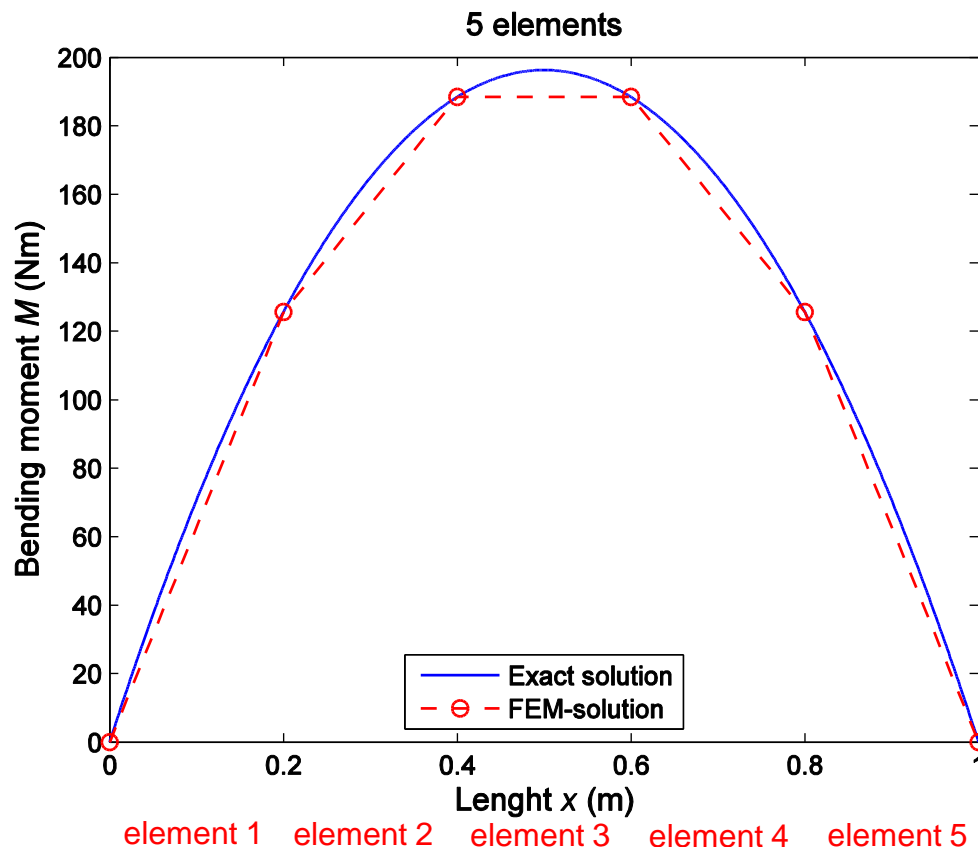
Possible computer exercise – Matlab

(i) Implement the **finite element method** with **linear basis functions** for the vertically loaded beam in **MATLAB** with the following initial data: ...



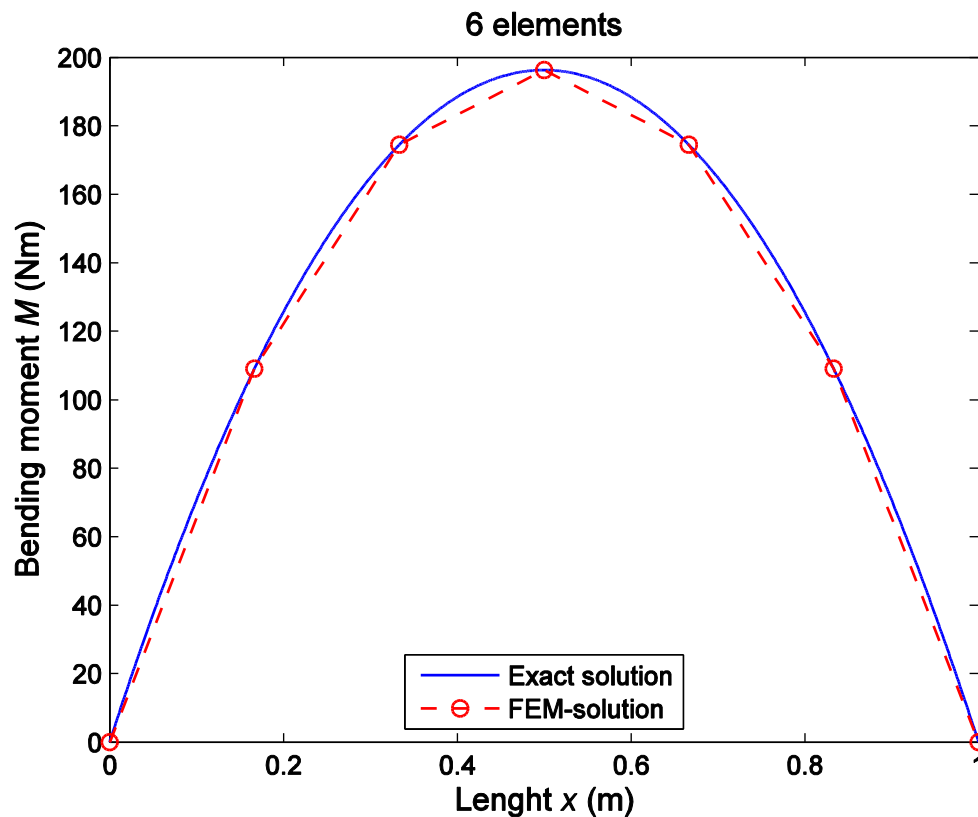
Possible computer exercise – Matlab

(i) Implement the finite element method with linear basis functions for the vertically loaded beam in **MATLAB** with the following initial data: ...



Possible computer exercise – Matlab

(i) Implement the **finite element method** with **linear basis functions** for the vertically loaded beam in **MATLAB** with the following initial data: ...



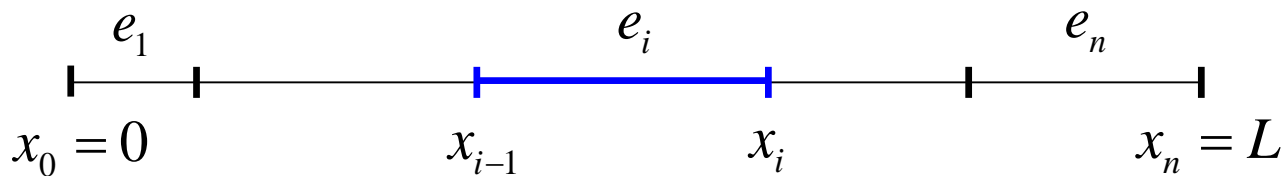
**BACK
TO**

basic material

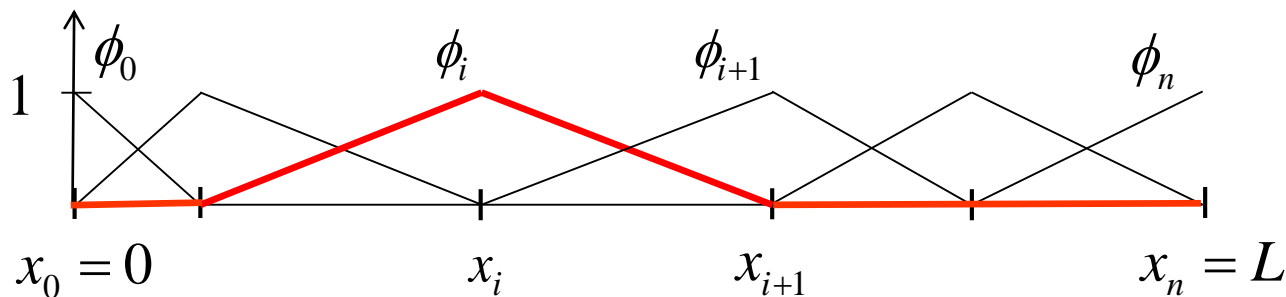
7.3 Finite element formulations for 1D and 2D model problems

1D bar in tension/compression:

Divide the solution interval (domain) into n subintervals e_i (*elements*) with *nodes* x_i and *element size* $h_i = x_i - x_{i-1}$:



In each element, the displacement field is approximated by (linear) polynomial *basis functions* which are now functions of the x -coordinate.



7.3 Finite element formulations for 1D and 2D model problems

This results in an **equation system** (initialize $K = \text{sparse}(m, m)$ in Matlab etc.)

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

with the **stiffness matrix** \mathbf{K} (computable for $i, j = 1, \dots, n$), **force vector** \mathbf{f} (computable for $i = 1, \dots, n$) and the **displacement vector** \mathbf{d} (unknown for $i = 1, \dots, n$):

$$\mathbf{K} = [K_{ij}], \quad K_{ij} = \int_0^L EA \phi_i' \phi_j' dx,$$

$$\mathbf{f} = [f_i], \quad f_i = \int_0^L b \phi_i dx + N_L \phi_i(L) - u_0 \int_0^L \frac{d\phi_i}{dx} AE \frac{d\phi_0}{dx} dx, \quad \mathbf{d} = [d_j]$$

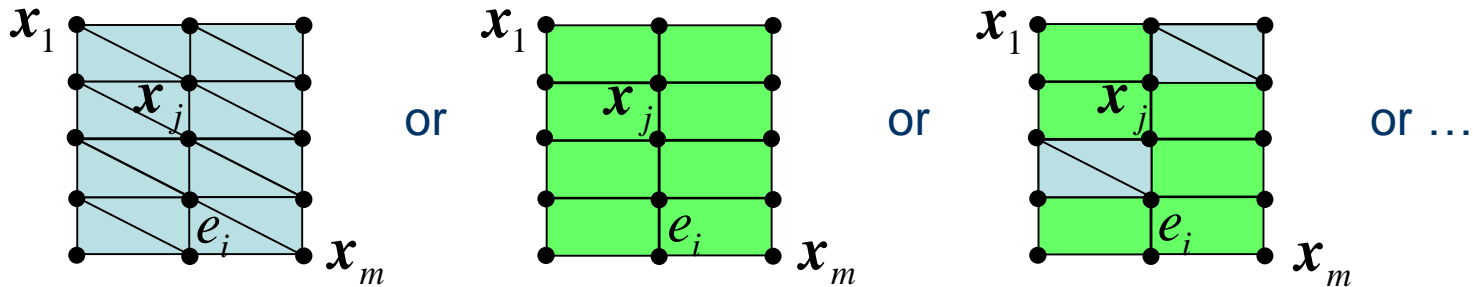
Remark. The stiffness matrix is (very often) **symmetric** (due to derivative orders) and its entries are concentrated in a narrow diagonal band forming a **band matrix** (due to local trial and test functions). These features can be utilized in computer implementation – implying small amounts of **memory needs** and quick **processing**.

Remark. Test and trial functions have to be (only) once locally differentiable (and will be then integrated over the domain) and (only) locally evaluable on the boundary.

7.3 Finite element formulations for 1D and 2D model problems

Generalization to 2D heat diffusion:

Divide the solution area (domain) into n subdomains, *elements* e_i (*triangles, quadrangles, ...*) with *nodes* $\mathbf{x}_j = (x_j, y_j)$ and *element size* $h_i = \text{diam}(e_i)$:



In each element, the temperature field is approximated by (linear) polynomial basis functions.

All functions are now functions of the plane coordinates x and y (instead of x alone). Accordingly, all integrals are now domain integrals (instead of line integrals).

7.3 Finite element formulations for 1D and 2D model problems

This finally results (details in the extra material) in an algebraic equation system

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

with the "*stiffness*" *matrix* (computable for $i, j = 1, \dots, m-p$), "*force*" *vector* (computable for $i = 1, \dots, m-p$) and the "*displacement*" *vector* (unknown for $i = 1, \dots, m-p$):

$$\mathbf{K} = [K_{ij}], \quad K_{ij} = \int_{\Omega} (k \nabla \phi_j) \cdot \nabla \phi_i \, d\Omega,$$

$$\mathbf{f} = [f_i], \quad f_i = \int_{\Omega} f \phi_i \, d\Omega - \int_{\Gamma_q} q_0 \phi_i \, ds - \sum_{\mathbf{x}_j \in \Gamma_T} T_0(\mathbf{x}_j) K_{ij},$$

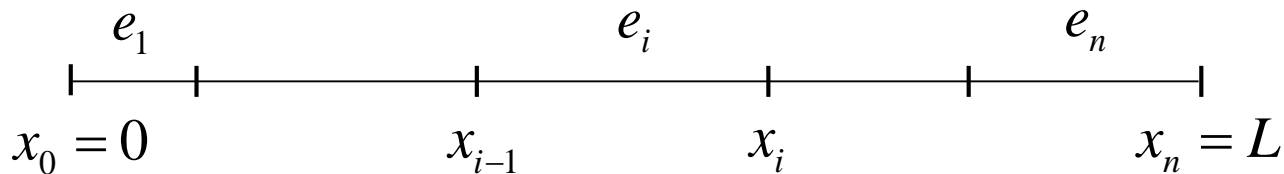
$$\mathbf{d} = [d_j].$$

Remark. The final equation system – matrix times vector equals vector – is analogous to the 1D case. This is one of the powerful features of the finite element method (or mathematics in general).

**HOW TO
DERIVE THE FINITE ELEMENT SYSTEM?**
extra material

7.3 Finite element formulations for 1D and 2D model problems

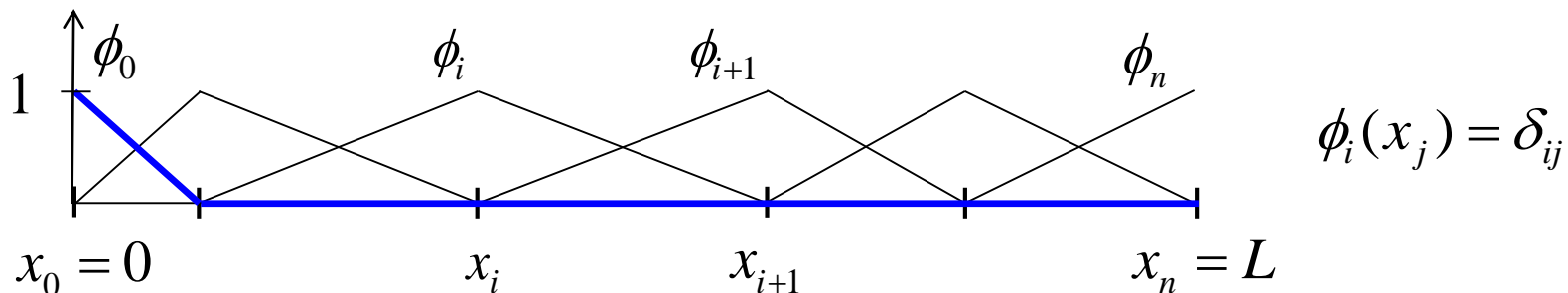
1. Divide the solution interval (domain) into n subintervals e_i (*elements*) with *nodes* x_i and the *element size* $h_i = x_i - x_{i-1}$:



2. Choose a *trial function* for the finite element approximation as a sum

$$u_h(x) = \phi_0(x)d_0 + \phi_1(x)d_1 + \dots + \phi_n(x)d_n = \sum_{j=0}^n \phi_j(x)d_j$$

with suitable *local basis functions* ϕ_i of some polynomial order (now linear)



The unknown scalar values $d_i = u_h(x_i)$ are called the *degrees of freedom*.

7.3 Finite element formulations for 1D and 2D model problems

Ensure that the trial function satisfies the essential boundary conditions:

$$u_0 = u_h(0) = \phi_0(0)d_0 + \phi_1(0)d_1 + \dots + \phi_n(0)d_n = d_0$$

3. Choose a test function of a similar form (*Galerkin method*) with the corresponding condition:

$$v(x) = \phi_0(x)c_0 + \phi_1(x)c_1 + \dots + \phi_n(x)c_n = \sum_{i=0}^n \phi_i(x)c_i$$

$$0 = v(0) \Rightarrow c_0 = 0$$

4. Insert the functions – trial and test – into the weak form:

$$\int_0^L EA u_h' v' dx = N_L v(L) + \int_0^L b v dx$$

$$\Rightarrow \int_0^L EA \left[\sum_{j=0}^n \phi_j' d_j \right] \left[\sum_{i=0}^n \phi_i' c_i \right] dx = N_L \left[\sum_{i=0}^n \phi_i(L) c_i \right] + \int_0^L b \left[\sum_{i=0}^n \phi_i c_i \right] dx$$

7.3 Finite element formulations for 1D and 2D model problems

This results in an **equation system** (initialize $K = \text{sparse}(m, m)$ in Matlab etc.)

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

with the **stiffness matrix** \mathbf{K} (computable for $i, j = 1, \dots, n$), **force vector** \mathbf{f} (computable for $i = 1, \dots, n$) and the **displacement vector** \mathbf{d} (unknown for $i = 1, \dots, n$):

$$\mathbf{K} = [K_{ij}], \quad K_{ij} = \int_0^L EA \phi_i' \phi_j' dx,$$

$$\mathbf{f} = [f_i], \quad f_i = \int_0^L b \phi_i dx + N_L \phi_i(L) - u_0 \int_0^L \frac{d\phi_i}{dx} AE \frac{d\phi_0}{dx} dx, \quad \mathbf{d} = [d_j]$$

7.3 Finite element formulations for 1D and 2D model problems

5. Use an appropriate *solver* for the equation system ($d = K \setminus f$ in Matlab):

$$d = K^{-1} f \Rightarrow u_h(x) = \sum_{j=1}^n \phi_j(x) d_j$$

6. Recover (and *postprocess*) the stress quantities and visualize:

$$\Rightarrow N_h(x) = (EAu_h)'(x) = \sum_{j=1}^n (EA\phi_j)'(x) d_j \Rightarrow \sigma_h(x) = \frac{N_h(x)}{A(x)}$$

7. Evaluate possible *error indicators*, change the *discretization* (steps 1–4) ... rerun ...

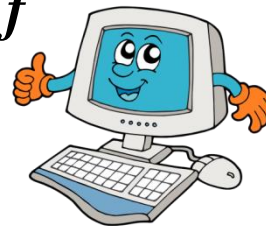
Remark. Steps 1–7 are *automated* – by means of mathematics and programming:

1. Elements e_i

2–3. Basis functions ϕ_i

4. Matrix entries K_{ij}, f_i

$$K d = f$$



$u_h(x)$

5. Equation solution

6. Visualization

7. Error evaluation

QUESTIONS?

ANSWERS”

LECTURE BREAK!