Computational Electrical Engineering

—

Numerical Lab

Piergiorgio Alotto, Paolo Bettini,
Francesco Lucchini, Riccardo Torchio

November 21, 2025

Contents

Chapter 1

Tableau and Nodal Analysis

1.1 Tableau analysis in DC

1.1.1 Example 1

Solve the DC problem in Figure 1.1 by means of Tableau analysis.

- Read the netlist (`netlist_01.cir`)
- Write a Matlab code to compute matrices **A**, **R**, **G**, and vector **s**
- Assemble the system matrix and the known term (right-hand side)
- Solve the linear system and check the results with the reference solution

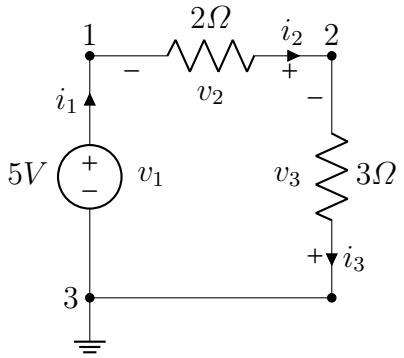


Figure 1.1: DC circuit (ideal voltage source and two ideal resistors).

i_1	i_2	i_3	v_1	v_2	v_3	p_1	p_2
+1.0 A	+1.0 A	+1.0 A	+5.0 V	-2.0 V	-3.0 V	+5.0 V	+3.0 V

Reading the netlist

The complete description (both topology and typology) of the simple circuit shown in Figure 1.1 is encoded into a Netlist.

n_1	n_2	t	v
3	1	V1	5
1	2	R1	2
2	3	R2	3

The Netlist can be read by a Matlab script:

`[nodes,types,labels,vals,err]=readcir('netlist_01.cir')`
which produces as output the following variables in the workspace:

```
Name     Size   Bytes   Class    Attributes
labels   3x1    324     cell
nodes    3x2    48      double
types    3x1    6       char
vals     3x2    48      double
```

labels contains the components' names or labels (e.g., `R1`); **nodes** contains the indices of nodes n_1 and n_2 (first and second columns, respectively); **types** contains the components' types (e.g., `R,V,I`); **vals** contains the components' values.

Construction of A, R, G, and s (simple algorithm)

We can assemble the complete incidence matrix $\mathbf{A}_c$ from the "edges-to-nodes" matrix **nodes**, with a `for` loop to fill in the matrix entries iteratively, as described in algorithm 1. The complete incidence matrix $\mathbf{A}_c$ is singular, so one row (typically the last one) is excluded and the (reduced) incidence matrix $\mathbf{A}$ is finally obtained.

Algorithm 1 Incidence matrix $\mathbf{A}$

Input: $n, e,$ **nodes**
Output: Matrix $\mathbf{A}$

1: $\mathbf{Ac} \leftarrow$ `zeros(n,e)`
2: $\mathbf{A} \leftarrow$ `zeros(n-1,e)`
3: **for** $k \leftarrow 1, e$ **do**
4: $\quad n1 \leftarrow \mathbf{nodes}(k, 1)$
5: $\quad \mathbf{Ac}(n1, k) \leftarrow -1$
6: $\quad n2 \leftarrow \mathbf{nodes}(k, 2)$
7: $\quad \mathbf{Ac}(n2, k) \leftarrow +1$
8: **end for**
9: $\mathbf{A} \leftarrow \mathbf{Ac}$`(1:end-1,:)`

Then, the matrices **R** and **G** and the vector **s** can be constructed from the components' data stored in **types** and **vals**. Following what we have studied, they can be assembled as described in algorithm 2.

Algorithm 2 Resistance matrix **R**, conductance matrix **G**, and source vector **s**

Input: $n, e,$ **types**, **vals**
Output: Matrices **R**, **G** and vector **s**

1: $\mathbf{R} \leftarrow$ `zeros(e,e)`
2: $\mathbf{G} \leftarrow$ `zeros(e,e)`
3: $\mathbf{s} \leftarrow$ `zeros(e,1)`
4: **for** $k \leftarrow 1, e$ **do**
5: $\quad t \leftarrow \mathbf{types}(k)$
6: $\quad v \leftarrow \mathbf{vals}(k)$
7: $\quad$ **if** $t ==' R'$ **then** ▷ selection of ideal resistors
8: $\quad\quad \mathbf{R}(k,k) \leftarrow +1$
9: $\quad\quad \mathbf{G}(k,k) \leftarrow 1/v$
10: $\quad\quad \mathbf{s}(k) \leftarrow 0$
11: $\quad$ **end if**
12: $\quad$ **if** $t ==' V'$ **then** ▷ selection of ideal voltage sources
13: $\quad\quad \mathbf{G}(k,k) \leftarrow +1$
14: $\quad\quad \mathbf{s}(k) \leftarrow v$
15: $\quad$ **end if**
16: $\quad$ **if** $t ==' I'$ **then** ▷ selection of ideal current sources
17: $\quad\quad \mathbf{R}(k,k) \leftarrow +1$
18: $\quad\quad \mathbf{s}(k) \leftarrow v$
19: $\quad$ **end if**
20: **end for**

Note that algorithms 1 and 2 are very easy to implement, but they are not computationally efficient, because all matrices and vectors assembled in this way are "dense", i.e., all entries are stored in memory, including all zeros. This is not a main issue for small circuits, but it leads to unnecessary waste of memory in the case of very large circuits (thousands to millions of components).

Building and solving the linear system in Matlab

The system matrix `M` can be easily assembled, block by block, provided that each block has the right size:

```
M  =  [A          zeros(n-1,e)  zeros(n-1,n-1);
       zeros(e,e)  -eye(e)       A';
       R           G             zeros(e,n-1)];
```

The known term (i.e., the right-hand side) `b` can be assembled as follows:

```
b  =  [zeros(n-1,1);  zeros(e,1);  s];
```

Finally, the solution $\mathbf{x}$ of the system of linear equations is obtained by using the "backslash" command (see 10.4.3).

```
x=M\b
```

Construction of A, R, G, and s (efficient algorithm)

Whenever a matrix contains a large enough percentage of zeros, the computational efficiency benefits from sparse linear algebra techniques, which can be exploited in Matlab by using appropriate instructions. For example, we can efficiently assemble the incidence matrix $\mathbf{A}$ from the data stored in **nodes**, as described in algorithm 3, where the `sparse` command is used (see 10.4.4).

Algorithm 3 Incidence matrix $\mathbf{A}$ (sparse implementation)

Input: $n, e,$ **nodes**
Output: Matrix $\mathbf{A}$

1: $cols \leftarrow 1 : e$
2: **Aplus** $\leftarrow$ `sparse`(**nodes**$(:, 2), cols,$ `ones`$(e, 1), n, e)$
3: **Aminus** $\leftarrow$ `sparse`(**nodes**$(:, 1), cols, -$`ones`$(e, 1), n, e)$
4: **Ac** $\leftarrow$ **Aplus** + **Aminus**
5: **A** $\leftarrow$ **Ac**`(1:end-1,:);`

Then, also $\mathbf{R}$, $\mathbf{G}$, and $\mathbf{s}$ can be constructed using computationally efficient algorithms that exploit their sparsity. For example, a sparse matrix $\mathbf{R}$ can be assembled as described in algorithm 4 for ideal resistors and current sources. A similar approach can be used for other components (e.g., capacitors). Note that no instructions are needed to deal with ideal voltage sources since the condition $R(k, k) = 0$ is automatically satisfied.

Algorithm 4 Resistance matrix **R** (sparse implementation)

Input: e, **types**, **vals**
Output: Matrix **R**

1: $idx \leftarrow \texttt{find}(\textbf{types} ==' R')$ ▷ indices of ideal resistors
2: $v \leftarrow \texttt{ones}(\texttt{numel}(idx), 1)$
3: $\mathbf{R} \leftarrow \texttt{sparse}(idx, idx, v, e, e)$
4: $idx \leftarrow \texttt{find}(\textbf{types} ==' I')$ ▷ indices of ideal current sources
5: $v \leftarrow \texttt{ones}(\texttt{numel}(idx), 1)$
6: $\mathbf{R} \leftarrow \mathbf{R} + \texttt{sparse}(idx, idx, v, e, e)$

Dually, a sparse matrix **G** can be assembled as described in algorithm 5 for ideal resistors and voltage sources. A similar approach can be used for other components (e.g., inductors). Note that no instructions are needed to deal with ideal current sources since the condition $G(k, k) = 0$ is automatically satisfied.

Algorithm 5 Conductance matrix **G** (sparse implementation)

Input: e, **types**, **vals**
Output: Matrix **G**

1: $idx \leftarrow \texttt{find}(\textbf{types} ==' R')$ ▷ indices of ideal resistors
2: $v \leftarrow \texttt{1}/\textbf{vals}(idx)$ ▷ values of the conductances
3: $\mathbf{G} \leftarrow \texttt{sparse}(idx, idx, v, e, e)$
4: $idx \leftarrow \texttt{find}(\textbf{types} ==' V')$ ▷ indices of ideal voltage sources
5: $v \leftarrow \texttt{ones}(\texttt{numel}(idx), 1)$
6: $\mathbf{G} \leftarrow \mathbf{G} + \texttt{sparse}(idx, idx, v, e, e)$

Finally, a sparse source vector **s** can be assembled as described in algorithm 6.

Algorithm 6 Source vector **s** (sparse implementation)

Input: e, **types**, **vals**
Output: Vector **s**

1: $idx \leftarrow \texttt{find}(\textbf{types} ==' V' \,|\, \textbf{types} ==' I')$ ▷ indices of ideal sources
2: $v \leftarrow \texttt{1}/\textbf{vals}(idx)$ ▷ values of voltages or currents
3: $\mathbf{s} \leftarrow \texttt{sparse}(idx, 1, v, e, 1)$

1.1.2 Example 2

Solve the DC problem in Figure 1.2 by means of Tableau analysis.

- Read the netlist (`netlist_02.cir`)
- Assemble the system matrix and the known term
- Solve the linear system and check the results with the reference solution

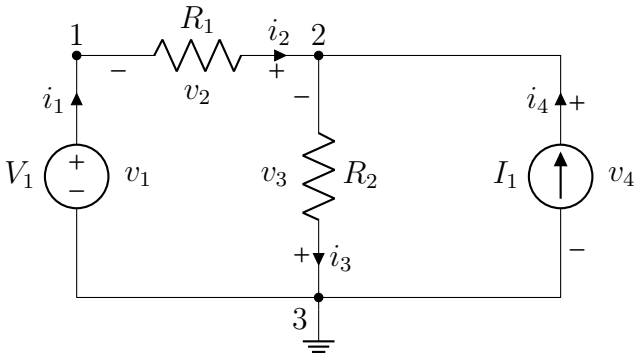


Figure 1.2: DC circuit: ideal voltage source, current source, and resistors.

i_1	i_2	i_3	i_4	v_1	v_2	v_3	v_4	p_1	p_2
$-1.0A$	$-1.0A$	$+4.0A$	$+5.0A$	$+10.0V$	$+2.0V$	$-12.0V$	$+12.0V$	$+10.0V$	$+12.0V$

Reading the netlist

The Netlist (`netlist_02.cir`) is the following:

n_1	n_2	t	v
3	1	V1	10.0
1	2	R1	2.0
2	3	R2	3.0
3	2	I1	5.0

1.2 Nonlinear Tableau analysis in DC (optional)

1.2.1 Example 1

Solve the nonlinear DC problem in Figure 1.3 by means of Tableau analysis.

- Read the netlist (`netlist_06_diodo.cir`)
- Write a Matlab code to compute matrices **A**, **R**, **G**, and vector **s**

- Assemble the system matrix and the known term (right-hand side)
- Solve the nonlinear system and check the results with the reference solution

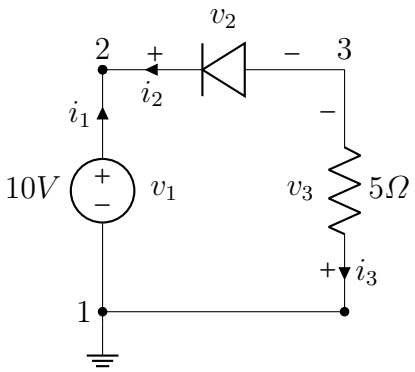


Figure 1.3: DC circuit (ideal voltage source, a diode, and a resistor).

The diode is described by the following nonlinear equation

$$i(v) = v(G_{min} + I_s(e^{\frac{v}{TV}} - 1))$$

where

- $G_{min} = 10^{-9}$
- $I_s = 10^{-7}$
- $TV = 25.85 \cdot 10^{-3}$

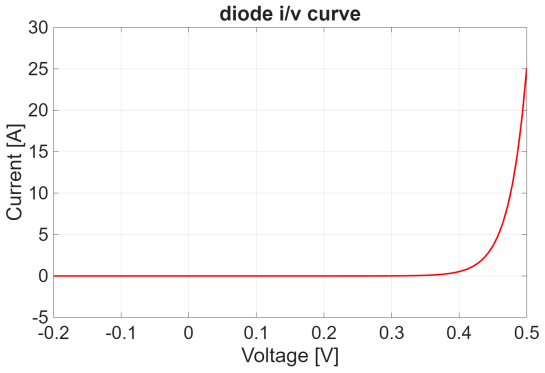


Figure 1.4: Diode voltage-current relation.

Reading the netlist

The description of the simple circuit shown in Figure 1.3 is encoded into a Netlist.

n_1	n_2	t	v
1	2	V1	10
3	2	D1	1
3	1	R1	5

In this circuit we have a nonlinear component (a diode), and the simple netlist does not provide the typological information about it.

The diode can be treated like a resistor with a non-linear conductance (i.e., the conductance depends on the diode voltage).

Thus, in MATLAB, matrix `M` must be constructed as a function handle:

```
M = @(x) assembleM_nonlin(x,...)
```

`assembleM_nonlin` is almost equivalent to the function developed for the linear case. The only difference is that it needs the (unknown) solution vector `x` as input to extract the diode's voltage and evaluate the diode nonlinear conductance as

$$G(v) = \frac{i(v)}{v} = (G_{min} + \frac{I_s(e^{\frac{v}{TV}} - 1)}{v})$$

Once we have constructed the function handle `M(x)`, we can solve the problem by using the `fsolve` nonlinear iterative solver of MATLAB:

```
x=fsolve(@(x) M(x)*x-b,x0);
```

where `x0` is the initial guest solution. `fsolve` look for a solution `x` such that `M(x)*x-b=0`.

Note that we cannot solve the problem with "\" since this is a nonlinear problem: `M(x)` is a function of the unknown solution `x`.

The solution in terms of diode voltage and current is:

v_2	i_2
$0.4334V$	$-1.9133A$

1.3 Nodal Analysis (optional)

1.3.1 Example 1 - NA

Solve the DC problem in Figure 1.5 by means of Nodal Analysis.

- Read the netlist (`netlist_03.cir`)
- Write a Matlab code to compute: **A**, **G**, **s**
- Assemble the system matrix and the known term (right hand side)
- Solve the linear system and check the results with the reference solution

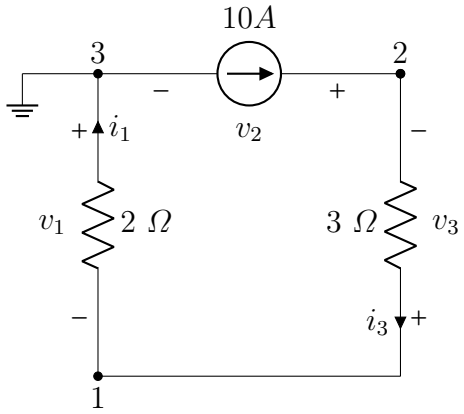


Figure 1.5: DC circuit: ideal current source and resistors.

p_1	p_2	i_1	i_2	i_3	v_1	v_2	v_3
$+20.0V$	$+50.0V$	$+10.0A$	$+10.0A$	$+10.0A$	$-20.0V$	$+50.0V$	$-30.0V$

Reading the netlist

The Netlist (`netlist_03.cir`) is the following:

n_1	n_2	t	v
1	3	R1	2
3	2	I1	10
2	1	R2	3

Construction of A, G, and s

A, **G**, and **s** can be assembled with the algorithms described in 1.1.1.

Building and solving the linear system in Matlab

The system matrix `M` and the known term `b` can be easily assembled in Matlab:

```
M=AGA'    b=As
```

Finally, the solution `x` of the system of linear equations is obtained by using the "backslash" command (see 10.4.3):

```
x=M\b
```

1.3.2 Example 2 - MNA

Solve the DC problem in Figure 1.6 by means of Modified Nodal Analysis.

- Read the netlist (`netlist_02.cir`)
- Write a Matlab code to compute: $\mathbf{A}_I$, $\mathbf{A}_V$ $\mathbf{G}_I$, $\mathbf{s}_I$, $\mathbf{s}_V$
- Assemble the system matrix and the known term (right-hand side)
- Solve the linear system and check the results with the reference solution

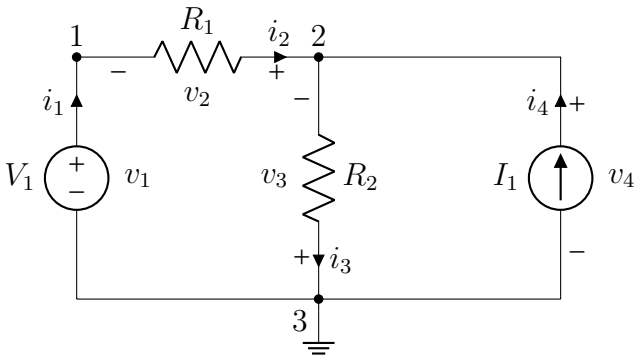


Figure 1.6: DC circuit: ideal voltage source, current source and resistors.

i_1	i_2	i_3	i_4	v_1	v_2	v_3	v_4	p_1	p_2
$-1.0A$	$-1.0A$	$+4.0A$	$+5.0A$	$+10.0V$	$+2.0V$	$-12.0V$	$+12.0V$	$+10.0V$	$+12.0V$

Reading the netlist

The Netlist (`netlist_03.cir`) is the following:

n_1	n_2	t	v
3	1	V1	10.0
1	2	R1	2.0
2	3	R2	3.0
3	2	I1	5.0

We know that the system of linear equations of MNA can be easily obtained if all components are sorted in such a way that the block of I-type components comes before the block of V-type components.

Since in general the components are in arbitrary positions in the netlist (for example, in this case `V1` is even in the first row) we need a simple way to sort the components as required.

We can either sort the "original" netlist, and then save and reload the "sorted" netlist, or operate with permutations of the entries of each variable produced in the workspace (`nodes, types, values`) when reading the data from the "original" netlist.

In the following we will refer to the latter and a simple procedure to "shuffle" the entries of all variables is shown in Algorithm 7.

Algorithm 7 Shuffle the entries of all variables (`nodes, types, values`)

Input: **nodes_o**, **types_o**, **vals_o** ▷ "original" variables
Output: **nodes_s**, **types_s**, **vals_s**, idx, r, s ▷ "sorted" variables

1: $ir \leftarrow$ `find(`**types_o** `==`$'$ R'`)` ▷ indices of ideal resistors
2: $ii \leftarrow$ `find(`**types_o** `==`$'$ I'`)` ▷ indices of ideal current sources
3: $iv \leftarrow$ `find(`**types_o** `==`$'$ V'`)` ▷ indices of ideal voltage sources
4: $idx \leftarrow$ `[ir;ii;iv]` ▷ permutation indices
5: $r \leftarrow$ `(numel(ir)+numel(ii))` ▷ number of I-type components
6: $s \leftarrow$ `numel(iv)` ▷ number of V-type components
7: **nodes_s** $\leftarrow$ **nodes_o**`(idx,:)`
8: **types_s** $\leftarrow$ **types_o**`(idx)`
9: **vals_s** $\leftarrow$ **vals_o**`(idx)`

Construction of $\mathbf{A}_I$, $\mathbf{A}_V$ $\mathbf{G}_I$, $\mathbf{s}_I$, $\mathbf{s}_V$

Once sorted the entries of all variables (`nodes, types, values`) read from the netlist, $\mathbf{A}$, $\mathbf{G}$, and $\mathbf{s}$ can be assembled with the algorithms described in 1.1.1. However, some additional steps are required to build $\mathbf{A}_I$, $\mathbf{A}_V$, $\mathbf{s}_I$, $\mathbf{s}_V$, and $\mathbf{G}_I$ as shown in Algorithm 8.

Algorithm 8 Splitting $\mathbf{A}$ into $\mathbf{A}_I$ and $\mathbf{A}_V$

Input: r, $\mathbf{A}$, $\mathbf{s}$, $\mathbf{G}$
Output: $\mathbf{A}_I$, $\mathbf{A}_V$, $\mathbf{s}_I$, $\mathbf{s}_V$, $\mathbf{G}_I$

1: $\mathbf{A}_I \leftarrow$ `A(:,1:r)`
2: $\mathbf{A}_V \leftarrow$ `A(:,r+1:end)`
3: $\mathbf{s}_I \leftarrow$ `s(1:r)`
4: $\mathbf{s}_V \leftarrow$ `s(r+1:end)`
5: $\mathbf{G}_I \leftarrow$ `G(1:r,1:r)`

Building and solving the linear system in Matlab

The system matrix `M` can be easily assembled, block by block, provided that each block has the right size:

```
M = [A_I G_I A_I'      -A_V;
     A_V'    zeros(s,s)];
```

The known term (i.e., the right-hand side) `b` can be assembled as follows:

```
b = [A_I s_I;  s_V];
```

Finally, the solution $\mathbf{x}$ of the system of linear equations is obtained by using the "backslash" command (see 10.4.3):

```
x=M\b
```

1.4 Tableau analysis in AC

1.4.1 Example 1

Solve the AC problem in Figure 1.8 by means of Tableau analysis for an angular frequency $\omega = 1000\,rad/s$.

- Read the (`netlist_04_RLCsin.cir`)
- Write a Matlab code to compute matrices **A**, **R**, **G**, **L**, **C**, and vector **s**
- Assemble the system matrix and the known term (right-hand side)
- Solve the linear system and check the results with the reference solution

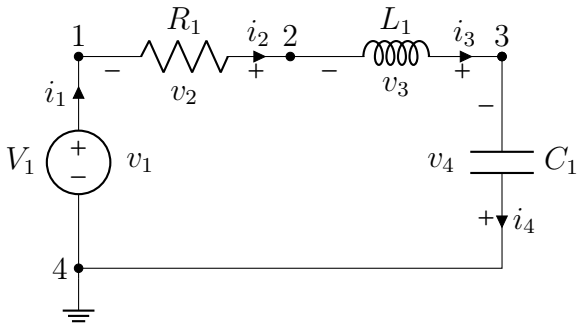


Figure 1.7: AC circuit: series RLC.

i_1	i_2	i_3	i_4	v_1	v_2	v_3	v_4	p_1	p_2	p_3
5 - 5i	5 - 5i	5 - 5i	5 - 5i	100 + 0i	-50 + 50i	-75 - 75i	25 + 25i	100 + 0i	50 + 50i	-25 - 25i

Reading the netlist

The Netlist (`netlist_04_RLCsin.cir`) is the following:

n_1	n_2	t	v_1	v_2
4	1	V1	100.0	0.0
1	2	R1	10.0	
2	3	L1	15.0e-3	
3	4	C1	200.0e-6	

Note that in AC, two parameters are used to define the ideal voltage source. The first (v_1) can indicate the amplitude or the r.m.s. value of the impressed sinusoidal voltage, in *volts*, and the second (v_2) denotes the initial phase, in *radian*. From now on, we assume that v_1 indicates the amplitude.

In this example, where $\omega = 1000\,rad/s$, we have the following expressions:

$$v_1(t) = 100\,\sin\left(1000\,t + 0\right) \quad \Leftrightarrow \quad \bar{V}_1 = 100\,e^{i\,0} = 100 + 0i$$

where we used a transformation rule that establishes a two-way correspondence between the sinusoid (amplitude, phase) and the phasor (modulus, argument).

In circuit theory it is often used a different rule which associates the sinusoid's r.m.s value to the phasor's modulus; in that case, we get:

$$\bar{V}_1 = 100/\sqrt{2}\,e^{i\,0} = 100/\sqrt{2} + 0i$$

The are no particular reasons to prefer one or the other, but we have to choose only one of the two (either amplitude or r.m.s. value) for all sinusoids.

Construction of A, R, and G

The matrices **A**, **R**, and **G** can be assembled with the algorithms in 1.1.1.

Construction of L, and C

The matrices **L** and **C** can be constructed from the components' data stored in **types** and **vals**, following the algorithm 9.

Algorithm 9 Inductance matrix **L**, Capacitance matrix **C**

Input: $n, e,$ **types**, **vals**
Output: Matrices **L**, **C**

1: **L** ← `zeros(e,e)`
2: **C** ← `zeros(e,e)`
3: **for** $k \leftarrow 1, e$ **do**
4: $t \leftarrow$ **types**(k)
5: $v \leftarrow$ **vals**$(k, 1)$
6: **if** $t ==' L'$ **then** ▷ selection of ideal inductors
7: **L**$(k, k) \leftarrow v$
8: **end if**
9: **if** $t ==' C'$ **then** ▷ selection of ideal capacitors
10: **C**$(k, k) \leftarrow v$
11: **end if**
12: **end for**

Construction of s

The vector **s** can be constructed from the components' data stored in **types** and **vals**, following the algorithm 10.

Algorithm 10 Source vector **s**

Input: $n, e,$ **types**, **vals**
Output: Vector **s**

1: **s** ← `zeros(e,1)`
2: **for** $k \leftarrow 1, e$ **do**
3: $t \leftarrow$ **types**(k)
4: **if** $t ==' V' \mid t ==' I'$ **then** ▷ selection of ideal sources
5: $v1 \leftarrow$ **vals**$(k, 1)$
6: $v2 \leftarrow$ **vals**$(k, 2)$
7: **s**$(k) \leftarrow v1 * exp(i * v2)$
8: **end if**
9: **end for**

Building and solving the linear system in Matlab

We need to assemble the following two square matrices `M1`, and `M2`, block by block, provided that each block has the right size. The former is the same as in DC Tableau Analysis. The latter takes into account the dynamic components.

```
M1  =  [A             zeros(n-1,e)  zeros(n-1,n-1);
        zeros(e,e)    -eye(e)       A';
        R             G             zeros(e,n-1)];

M2  =  [zeros(n-1,e)  zeros(n-1,e)  zeros(n-1,n-1);
        zeros(e,e)    zeros(e,e)    zeros(e,n-1);
        L             C             zeros(e,n-1)];
```

Then, we define the system matrix as:

```
M=M1+i*omega*M2;
```

The known term (i.e., the right-hand side) `b` can be assembled as follows:

```
b  =  [zeros(n-1,1);  zeros(e,1);  s];
```

Finally, the solution **x** of the system of linear equations is obtained by using the "backslash" command (see 10.4.3):

```
x=M\b
```

where each entry of **x** is a complex number (phasor).

1.4.2 Example 2

Solve the AC problem in Figure 1.8 by means of MNA analysis for an angular frequency $\omega = 1000\,rad/s$.

- Read the (`netlist_04_RLCsin.cir`)
- Write a Matlab code to compute matrices $\mathbf{A}$, $\mathbf{Y}$, and vector $\mathbf{s}$
- Assemble the system matrix and the known term (right hand side)
- Solve the linear system and check the results with the reference solution

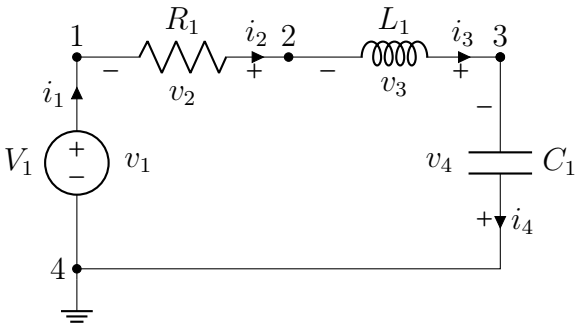


Figure 1.8: AC circuit: series RLC.

p_1	p_2	p_3	i_1
$100 + 0i$	$50 + 50i$	$-25 - 25i$	5 - 5i

Reading the netlist

The Netlist (`netlist_04_RLCsin.cir`) is the following:

n_1	n_2	t	v_1	v_2
4	1	V1	100.0	0.0
1	2	R1	10.0	
2	3	L1	15.0e-3	
3	4	C1	200.0e-6	

Construction of $\mathbf{A}_I$, $\mathbf{A}_V$, and source terms $\bar{\mathbf{S}}_I$, $\bar{\mathbf{S}}_I$

Once sorted the entries of all variables (`nodes, types, values`) read from the netlist, the matrix $\mathbf{A}$, can be assembled with the algorithm described in Algorithm 1.1.1. Then matrices $\mathbf{A}_I$, $\mathbf{A}_V$, and the source terms $\bar{\mathbf{S}}_I$, $\bar{\mathbf{S}}_I$ can be assembled as shown in Algorithm 11.

Algorithm 11 Splitting $\mathbf{A}$ into $\mathbf{A}_I$, $\mathbf{A}_V$, and $\bar{\mathbf{S}}$ into $\bar{\mathbf{S}}_I$, $\bar{\mathbf{S}}_I$

Input: r, s, $\mathbf{A}$, $\mathbf{S}$
Output: $\mathbf{A}_I$, $\mathbf{A}_V$, $\mathbf{S}_I$, $\mathbf{S}_V$

1: $\mathbf{S}_I \leftarrow$ `zeros(r,1)`
2: $\mathbf{S}_V \leftarrow$ `zeros(s,1)`
3: $\mathbf{A}_I \leftarrow \mathbf{A}$`(:,1:r)`
4: $\mathbf{A}_V \leftarrow \mathbf{A}$`(:,r+1:end)`
5: $v \leftarrow \mathbf{vals}(:,1) * exp(i * \mathbf{vals}(:,2))$
6: $\mathbf{S}_I \leftarrow v(1:r)$
7: $\mathbf{S}_V \leftarrow v(r+1:end1)$

Construction of the admittance matrix of I-type components $\mathbf{Y}_I$

The admittance matrix of I-type components $\mathbf{Y}_I$ can be assembled as shown in Algorithm 12.

Algorithm 12 Building $\dot{\mathbf{Y}}_I$

Input: $e, \mathbf{types}, \mathbf{vals}, \omega$
Output: Matrix $\mathbf{Y}_I$

1: $\mathbf{Y} \leftarrow$ `zeros(e,e)`
2: $\mathbf{Y}_I \leftarrow$ `zeros(r,r)`
3: **for** $k \leftarrow 1, e$ **do**
4: $\quad t \leftarrow \mathbf{types}(k)$
5: $\quad v \leftarrow \mathbf{vals}(k,1)$
6: $\quad$ **if** $t ==' R'$ **then** ▷ selection of ideal resistors
7: $\quad\quad \mathbf{Y}(k,k) \leftarrow 1/v$
8: $\quad$ **end if**
9: $\quad$ **if** $t ==' L'$ **then** ▷ selection of ideal inductors
10: $\quad\quad \mathbf{Y}(k,k) \leftarrow -i/(\omega * v)$
11: $\quad$ **end if**
12: $\quad$ **if** $t ==' C'$ **then** ▷ selection of ideal capacitors
13: $\quad\quad \mathbf{Y}(k,k) \leftarrow i * \omega * v$
14: $\quad$ **end if**
15: **end for**
16: $\mathbf{Y}_I \leftarrow \mathbf{Y}(1:r, 1:r)$

Building and solving the linear system in Matlab

The system matrix `M` can be easily assembled, block by block, provided that each block has the right size:

```
M = [A_I Y_I A_I'      -A_V;
       A_V'     zeros(s,s)];
```

The known term (i.e., the right-hand side) `b` can be assembled as follows:

```
b = [A_I S_I;  S_V];
```

Finally, the solution $\mathbf{x}$ of the system of linear equations is obtained by using the "backslash" command (see 10.4.3):

```
x=M\b
```

where each entry of $\mathbf{x}$ is a complex number (phasor).

1.5 Transient Tableau Analysis

1.5.1 Example 1

Solve the circuit in Figure 1.9 in the interval $[0,\ 5s]$ by means of Transient Tableau analysis with θ-method.

- Read the netlist (`netlist_05_RL.cir`)
- Compute a self-consistent set of Initial Conditions: $\mathbf{x_0}$
- Assemble the matrices $\mathbf{K_1}$, $\mathbf{K_2}$ and the known term
- Solve the linear system of equations at each time step with the θ-method: $\theta = 0.5$, $\Delta t = 0.1\,s$.
- Compare the results with the analytical solution

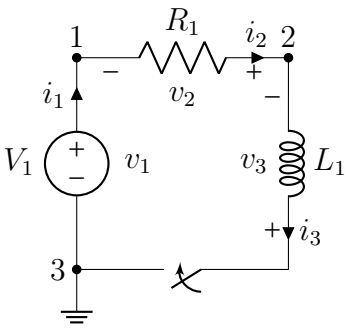


Figure 1.9: Time domain problem: RL circuit fed by a constant voltage source.

Reading the netlist

The Netlist (`netlist_05_RL.cir`) is the following:

n_1	n_2	t	v_1	v_2
3	1	V1	1.0	
1	2	R1	1.0	
2	3	L1	1.0	

Initial Conditions (ICs)

To compute a self-consistent set of ICs, substitute L_1 with an open circuit (the component's equation of an open circuit is $i = 0$, so $R_{k,k} = 1$, $G_{k,k} = 0$, and $s_k = 0$) and run the Tableau Analysis DC code. Alternatively, solve the problem with paper and pen since it is trivial. The solution at $t = 0$ provides the ICs' values for the transient problem: $\mathbf{x_0}$.

Construction of A, R, and G

The matrices **A**, **R**, and **G** can be assembled with the algorithms in 1.1.1.

Construction of L, and C

The matrices **L** and **C** can be constructed from the components' data stored in **types** and **vals**, following the algorithm 9.

Building and solving the linear system in Matlab with the Theta Method (optional)

We have to assemble the matrices `M1` and `M2`, block by block:

```
M1  =  [A             zeros(n-1,e)  zeros(n-1,n-1);
        zeros(e,e)    -eye(e)       A';
        R             G             zeros(e,n-1)];

M2  =  [zeros(n-1,e)  zeros(n-1,e)  zeros(n-1,n-1);
        zeros(e,e)    zeros(e,e)    zeros(e,n-1);
        L             C             zeros(e,n-1)];
```

Then, we assemble the matrices `K1`, and `K2` as:

```
K1  =  theta*M1+1/deltat*M2;
K2  =  -(1-theta)*M1+1/deltat*M2;
```

As far as the known term is concerned, in general, it changes at any time instant because the ideal sources can change arbitrarily in time. In this case, we have only a constant voltage source, so the known term is constant throughout the entire simulation. We denote it with `q0`.

Finally, the solution is obtained by solving at any time instant a linear system of equations, where the right-hand side is updated at any iteration as shown in Algorithm 13.

For example, the solution at the first step, `x1`, can be obtained by computing:

```
b1=K2*x0+q0
```

where `x0` are the ICs, and the solving the linear system of equations:

```
x1=K1\b1
```

Algorithm 13 Time-stepping scheme

Input: $n, e, ntmax, deltat, \mathbf{K_1}, \mathbf{K_2}, \mathbf{x_0}, \mathbf{q_0}$
Output: $\mathbf{x}$, $\mathbf{t}$

1: $\mathbf{x} \leftarrow$ `zeros(2e+n-1,ntmax)`
2: $\mathbf{t} \leftarrow$ `zeros(ntmax,1)`
3: $\mathbf{xprev} \leftarrow \mathbf{x_0}$
4: **for** $k \leftarrow 2, ntmax$ **do**
5: $\quad \mathbf{b} \leftarrow \mathbf{K_2} * \mathbf{xprev} + \mathbf{q_0}$
6: $\quad \mathbf{xnew} \leftarrow \mathbf{K_1} \backslash \mathbf{b}$
7: $\quad \mathbf{x}(:, k) \leftarrow \mathbf{xnew}$
8: $\quad \mathbf{t}(k) \leftarrow$ `deltat*(k-1)`
9: $\quad \mathbf{xprev} \leftarrow \mathbf{xnew}$
10: **end for**

Building and solving the linear system in Matlab with ODE functions

An alternative way to solve the transient problem is the use the ode solver functions of MATLAB.

To do that, we need to construct matrices `M1` and `M2` and construct a function handle of the equation we are solving.

The problem we are solving can be written as

$$\underbrace{\texttt{M2}}_{mass\ matrix}\ \dot{\texttt{x}} = \underbrace{-\texttt{M1x} + \texttt{q(t)}}_{\texttt{F(t,x)}}$$

To solve it with the ODE solver(s) of MATLAB we need the following commands:

- First, we construct the function handle `F(t,x)`:

  ```
  F=@(t,x) -M1*x+q(t)
  ```

  where `q(t)` is the time dependent source term.

- then we prepare the `options` for the ode solvers:

```
options = odeset('Mass',M2,'RelTol',1e-5);
```

with this command, we define the mass matrix of the problem (i.e., matrix `M2`) and the prescribed relative tolerance (i.e., `1e-5`).

- Finally, we call the ode solver (in this case we use `ODE23t`, but there are many other possible solvers available, check MATLAB help to see the differences). This command will solve the time domain problem in the time window `[tmin,tmax]` and with initial solution `sol0`:

```
[time,sol]=ode23t(F,[tmin,tmax],sol0,options);
```

Try to use `ODE15s` and compare the results.

A simple example taken from physics is the equation of motion of the pendulum with length L under gravity acceleration g, for small oscillations (period $T = 2\pi\sqrt{L/g}$)

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\theta. \tag{1.1}$$

Also this equation can be integrated with the proposed approach; indeed, define $v = d\theta/dt$ then (1.1) can be transformed into:

$$\begin{array}{l} \frac{d\theta}{dt} = v \\ \frac{dv}{dt} = -\frac{g}{L}\theta \end{array}. \tag{1.2}$$

If we define the array of unknowns $\mathbf{x} = [\theta, v]^\intercal$, (1.2) can be written as a system of first-order ordinary differential equations

$$\frac{d}{dt}\begin{bmatrix}\theta \\ v\end{bmatrix} = \begin{bmatrix}0 & 1 \\ -g/L & 0\end{bmatrix}\begin{bmatrix}\theta \\ v\end{bmatrix} \tag{1.3}$$

in the form $\mathbf{M_2}\dot{\mathbf{x}} = -\mathbf{M_1}\mathbf{x} + \mathbf{q}$. In this case, $\mathbf{M_2} = \mathbf{I}$ is the identity matrix, so we do not need it in the procedure.

$$\mathbf{M_1} = -\begin{bmatrix}0 & 1 \\ -g/L & 0\end{bmatrix} \tag{1.4}$$

and $\mathbf{q} = 0$.

Building and solving the linear system in Matlab Backward Euler

With Backward Euler (that coincides with the Theta method with $\theta = 1$), the problem

$$\texttt{M2}\dot{\texttt{x}} = -\texttt{M1x} + \texttt{q(t)}$$

is discretized as

$$\texttt{M2}\frac{\texttt{x}^{(\texttt{k+1})} - \texttt{x}^{(\texttt{k})}}{\texttt{dT}} = -\texttt{M1x}^{(\texttt{k+1})} + \texttt{q((k+1)*dT)}$$

Try to solve the transient problem with this approach (select a proper value for the time step `dT`).

Building and solving the linear system in Matlab Forward Euler

With Forward Euler (that coincides with the Theta method with $\theta = 0$), the problem

$$\texttt{M2}\dot{\texttt{x}} = -\texttt{M1x} + \texttt{q(t)}$$

is discretized as (spot the difference w.r.t. Backward Euler)

$$\texttt{M2}\frac{\texttt{x}^{(\texttt{k+1})} - \texttt{x}^{(\texttt{k})}}{\texttt{dT}} = -\texttt{M1x}^{(\texttt{k})} + \texttt{q((k+1)*dT)}$$

Try to solve the transient problem with this approach (select a proper value for the time step `dT`). Note: This will not work. Why?

Reference solution

The analytical solution, with the generators' rule, is the following:

$$i_i(t) = 1 - e^{-t} \qquad i_2(t) = 1 - e^{-t} \qquad i_3(t) = 1 - e^{-t}$$

$$v_1(t) = 1 \qquad v_2(t) = -R_1\, i_2(t) = -\left(1 - e^{-t}\right) \qquad v_3(t) = -L_1 \frac{di_3}{dt} = -e^{-t}$$

$$p_1(t) = 1 \qquad p_2(t) = e^{-t}$$

1.5.2 Example 2

Solve the circuit in Figure 1.11 in the time interval $[0,\ 0.1\,s]$ by means of Transient Tableau analysis with θ-method.

- Read the netlist (`netlist_06_RLC.cir`)

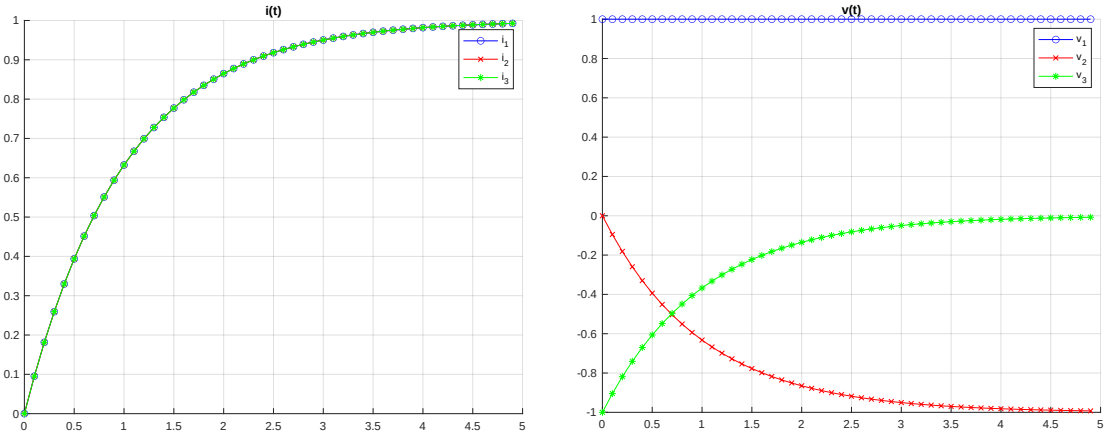


Figure 1.10: RL circuit: analytical solution. Left: currents. Right: voltages.

- Compute a self-consistent set of Initial Conditions: $\mathbf{x_0}$
- Assemble the matrices $\mathbf{K_1}$, $\mathbf{K_2}$ and the known term
- Solve the linear system of equations at each time step with the θ-method: $\theta = 0.5$, $\Delta t = 1\,ms$.
- Compare the results with the Matlab solution in Appendix 10.7.4

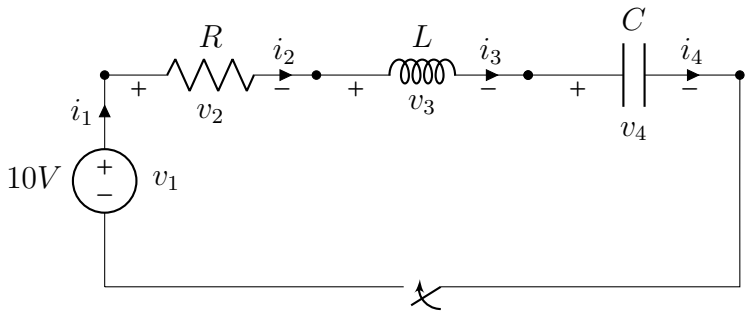


Figure 1.11: Time domain problem: RCL circuit fed by a constant voltage source

Circuit's parameters and initial conditions					
$L = 100\,mH$	$C = 10\,\mu F$	$R = 10\,\Omega$	$i_3(0) = 0\,A$	$v_4(0) = 0\,V$	$v_1 = 10\,V\,(const)$

1.6 Assignments

1.6.1 Mandatory assignments

1. A script that loads a netlist and calls the related functions for the DC (linear) Tableau Analysis (+ one of the netlist described in the exercises to test the code)

2. A script that loads a netlist and calls the related functions for the AC (linear) Tableau Analysis (+ one of the netlist described in the exercises to test the code)

3. Script that loads the netlist and calls the related functions for the transient (linear) Tableau Analysis, solved with ODE solver of MATLAB (+ one of the netlist described in the exercises to test the code)

1.6.2 Optional assignments

1. (+ 0.5 point to be checked during oral examination) Script and related functions for the Nodal Analysis (DC, AC, Transient)

2. (+ 1 points to be checked during oral examination) Script and related functions for Nonlinear (with diodes) DC Tableau Analysis

3. (+ 0.5 points to be checked during oral examination) Script and related functions for Nonlinear (with diodes) Transient Tableau Analysis

Chapter 2

LTspice

In this chapter, we use LTspice software for DC, AC, and transient simulations.

First, you need to download the LTspice software from that is freely available online.

2.1 LTspice - DC

Here we verify the solution of the simple circuit in Fig. 1.2 we previously solved with the Tableau analysis.

- Open LTspice
- Create a new schematic.
- As shown in the figure below, use the component icon (highlighted in red) to add components to the schematic (use ctrl+R to rotate the component).
- Then, connect the components by using the icon highlighted by the green box.
- Right-click on the components to add the values.
- Solve the problem by clicking the run icon (highlighted by the yellow box). Select DC op pnt (DC operational point).
- Verify the solution (note that current conventions may be different and LTspice gives as a result the nodes' potentials w.r.t. to ground).

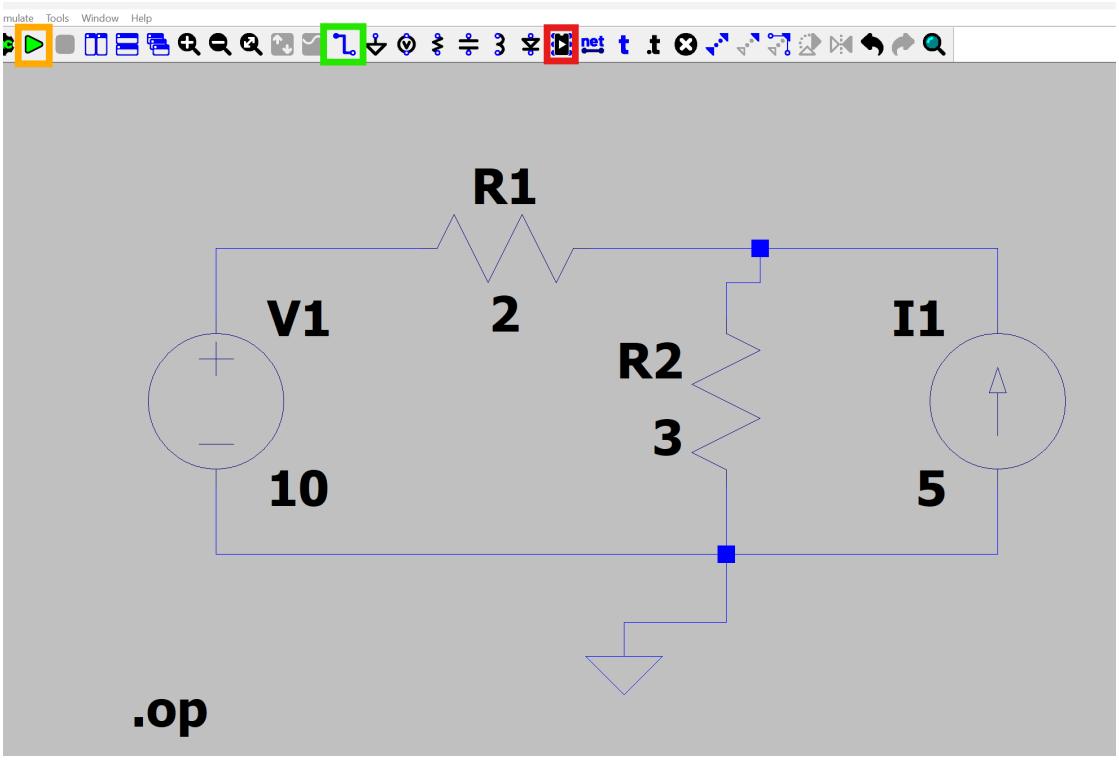


Figure 2.1: LTspice schematics.

2.2 LTspice - AC

Verify the solution of Fig. 1.8 with LTspice.

2.3 LTspice - Transient

2.3.1 Example 1 - Full-wave Rectifier

We simulate a full-wave rectifier as shown below. Ideal diodes are used.

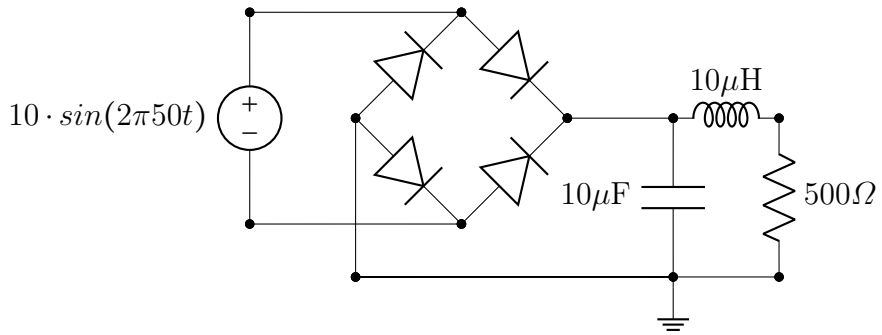


Figure 2.2: Full-wave rectifier.

The figure below shows the results obtained from LTspice (transient simulation).

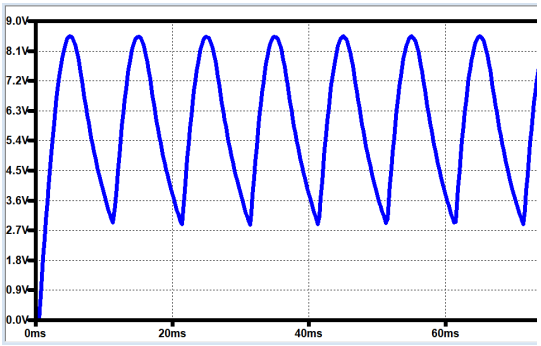


Figure 2.3: Output Voltage of the Full wave Rectifier (Resistor's voltage).

Change the capacitance value of the capacitor to obtain a smoother output voltage (the voltage of the resistor).

2.3.2 Example 2 - DC-DC Buck Converter

Here we construct the model of a DC-DC Buck Converter. The schematic is shown in the figure below.

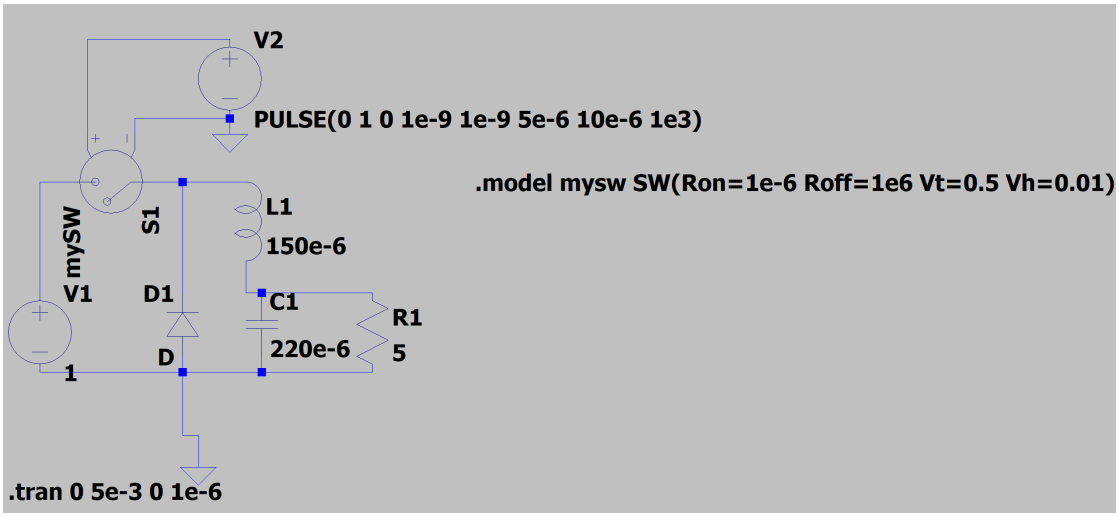


Figure 2.4: Voltage Output of the Full wave Rectifier

For the model of the switch, we use the `sw` component (controlled switch). Then, right-click on the component and change the value property to `mysw`. Then add an LTspice directive by clicking on the directive icon `.t`. Then write `.model mysw SW(Ron=1e-6 Roff=1e6 Vt=0.5 Vh=0.01)`

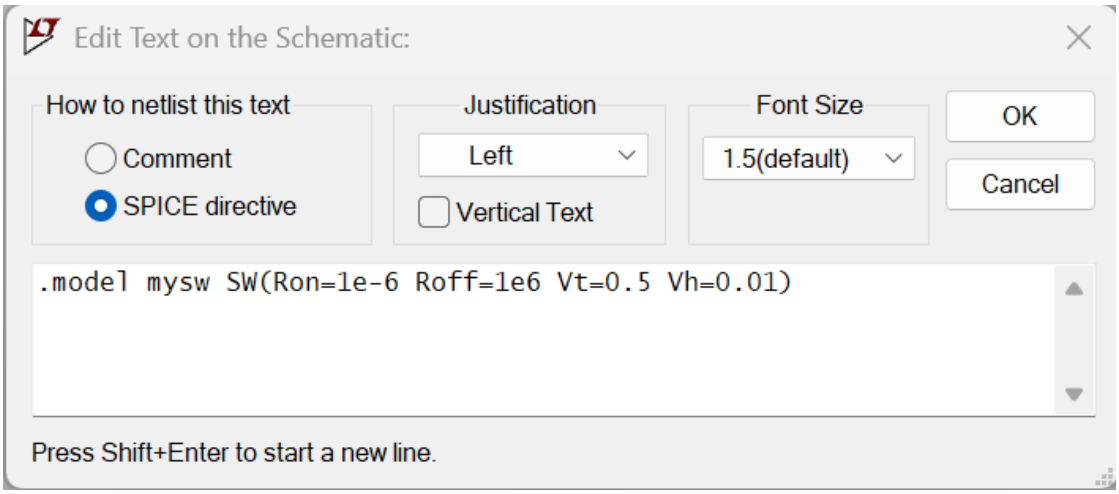


Figure 2.5: LTspice directive for the controlled switch.

By doing this, we are selecting the properties of the controlled switch (that can be seen as a resistor with varying resistance):

- `Ron` is the resistance value when the switch is closed

- `Roff` is the resistance value when the switch is open
- `Vt` is the threshold voltage for switching
- `Vh` is the Hysteresis voltage (adds noise immunity)

This model defines a hysteresis-controlled switch, meaning:

- The switch turns ON (`Ron`) when the control voltage goes above `Vt + Vh/2` → 0.505 V
- The switch turns OFF (`Roff`) when the control voltage goes below `Vt - Vh/2` → 0.495 V

This avoids erratic switching when the control voltage hovers near the threshold, which is useful in noisy environments.

The switch is controlled by a pulse voltage source defined as in the image below.

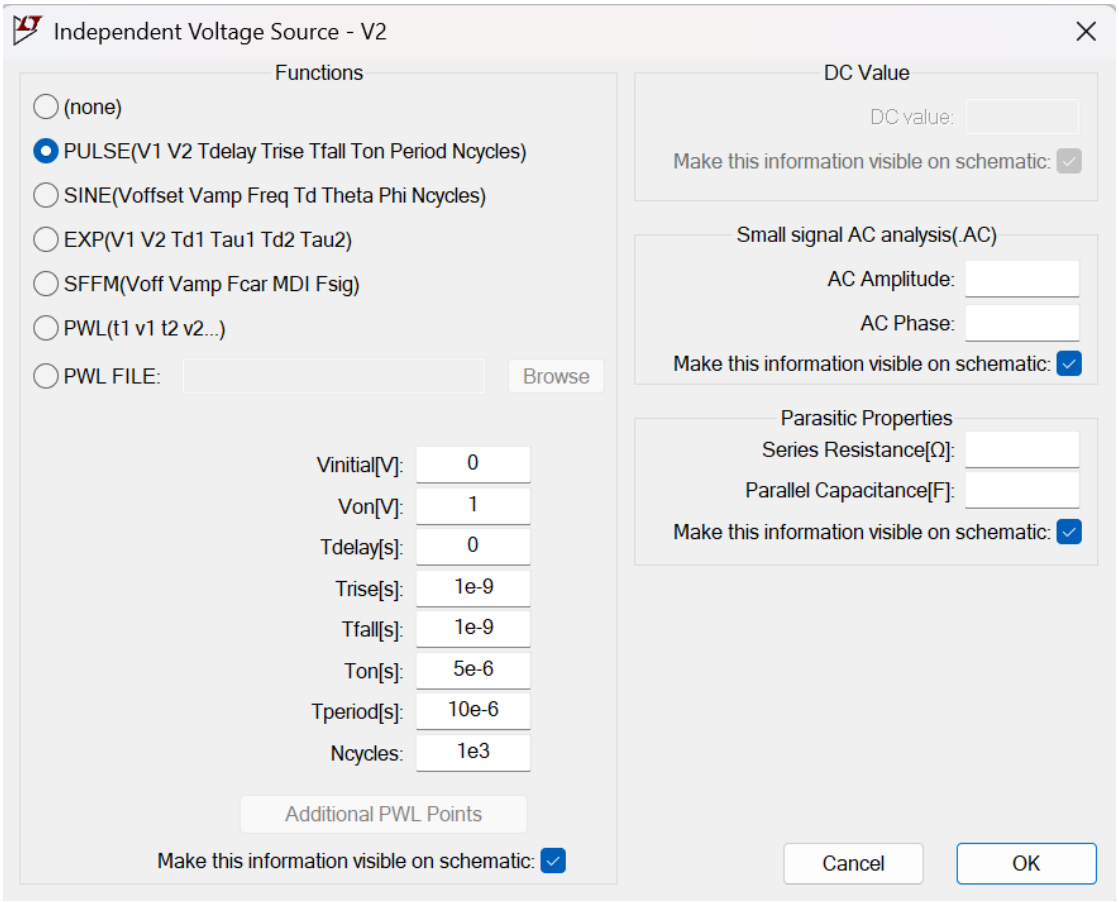


Figure 2.6: Pulse Voltage Source to control the switch.

By simulating the schematic, the following solution is obtained.

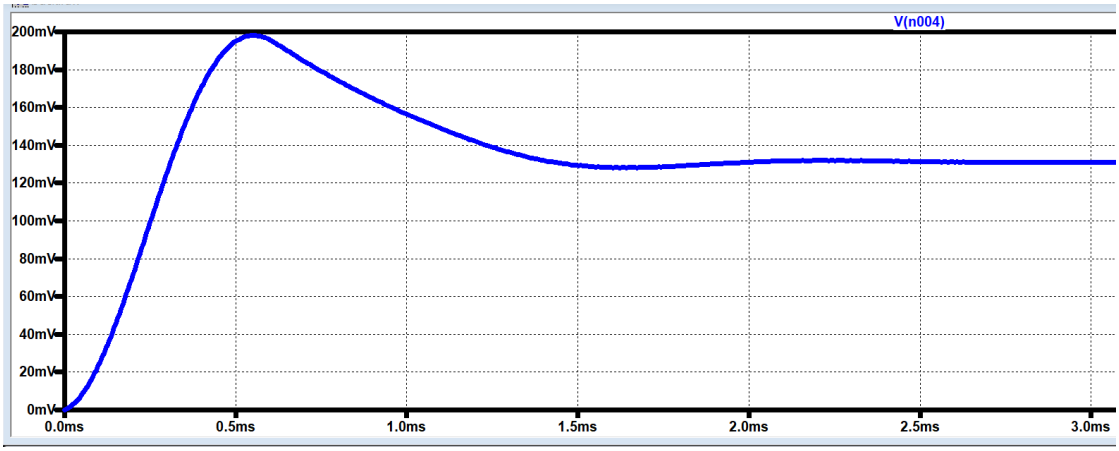


Figure 2.7: Output voltage of the DC-DC Buck converter.

2.4 Assignments

2.4.1 Mandatory assignments

- LTspice file of the Full-wave rectifier.
- LTspice file of the DC-DC Buck converter.

2.4.2 Optional assignments

- (+ 2 points to be checked during oral examination) Create a working LTspice model of a Wireless Power Transfer (WPT) system including:
  1. 50 Hz to DC Rectifier,
  2. DC to AC resonant inverter (85 kHz for automotive, 13.56 MHz for NFC antennas, or something else),
  3. two-port inductor network with compensation (transmitter and receiver coils plus capacitors for the compensation),
  4. rectifier at the receiver side,
  5. equivalent load or battery.

Chapter 3

Vector Fitting

3.1 Using Vector Fitted Models in LTspice

Apply Vector Fitting to the admittance data of a real capacitor. Then, convert the Vector Fitted model to an equivalent netlist and load it into LTspice, then perform a transient simulation.

- Download the Vector Fitting Algorithm (follow Downloads/Matrix Fitting Toolbox) and add it to the Matlab path
- Load in MATLAB the admittance frequency domain data of the real component (data are available in moodle)
- Call the Vector Fitting function to create a state space model of the component (the Vector Fitting function already converts the fitted rational function to the state space model)
- Call the MATLAB functions that generate the equivalent netlist from the state space model
- Call the MATLAB function that generates the LTspice files of the equivalent netlist model
- Follow the instructions to load the LTspice Vector Fitted model in LTspice
- Verify the model: perform a frequency domain study in LTspice to verify that the equivalent model provides the same impedance (magnitude and phase)
- use the LTspice VF capacitor model in place of the ideal capacitor previously used in the Section 2.3.2

- Compare the output voltage obtained from the DC-DC Buck converter simulation with the ideal and Vector Fitted capacitor models (also perform the FFT of the output voltage)

Using the Vector Fitting library

Once downloaded the Vector Fitting toolbox put it in the working directory and add it to the MATLAB path (in this way, MATLAB will "see" also the functions of the toolbox and we can use them). You can do it by right-clicking on the directory and then "Add to Path, Select Folder(s) and Subfolder(s)" or by adding at the beginning of the script the command

```
addpath(genpath(pwd));
```

that adds all the subfolders of the working directory to the MATLAB path.

The main function of the Vector Fitting toolbox we are going to use is `VFdriver` that can be used in this way

Listing 3.1: Calling the Vector Fitting Function

```
opts=[]; % Initialize the options struct
opts.N=3; % Model order (how many poles, try different values
   , 1,2,3,...)
opts.poletype='logcmplx';  % look inside the VFdriver
   function for more information
opts.asymp=2; % Model includes R0 and E? % Keep this value
opts.stable=1; % 1-> poles are 'flipped' into the left half
   plane
opts.passive_DE=1; % 1-> Enforce positive realness for D,E
   during fitting (Def=0)
poles=[]; % first guest location of the poles, if empty the
   code will take care
opts.cmplx_ss=1; % 1-> opts.cmplx_ss=1 -->  generate complex
   state space model with diagonal A (keep this value)
[SER,rmserr,bigYfit,opts2]=VFdriver(bigY,s,poles,opts);
```

The first two inputs of `VFdriver` are:

- `bigY`, a variable storing the admittance values (complex numbers) for all the measured frequencies. In this example, `bigY` has dimension $1\times1\times N_{freq}$, where N_{freq} is the number of measurements. Why? For multiport components, `bigY` stores the admittance *matrix*, in that case, `bigY` has dimension $N_{ports}\times N_{ports}\times N_{freq}$, where N_{ports} is the number of ports.

- `s`, a vector of size N_{freq} storing the complex frequencies of the measurements, i.e., `s=j`ω.

By running `VFdriver` function we should obtain the following Fig. 3.1 comparing the admittance values (magnitude and phase) of the measurements with the fitted model.

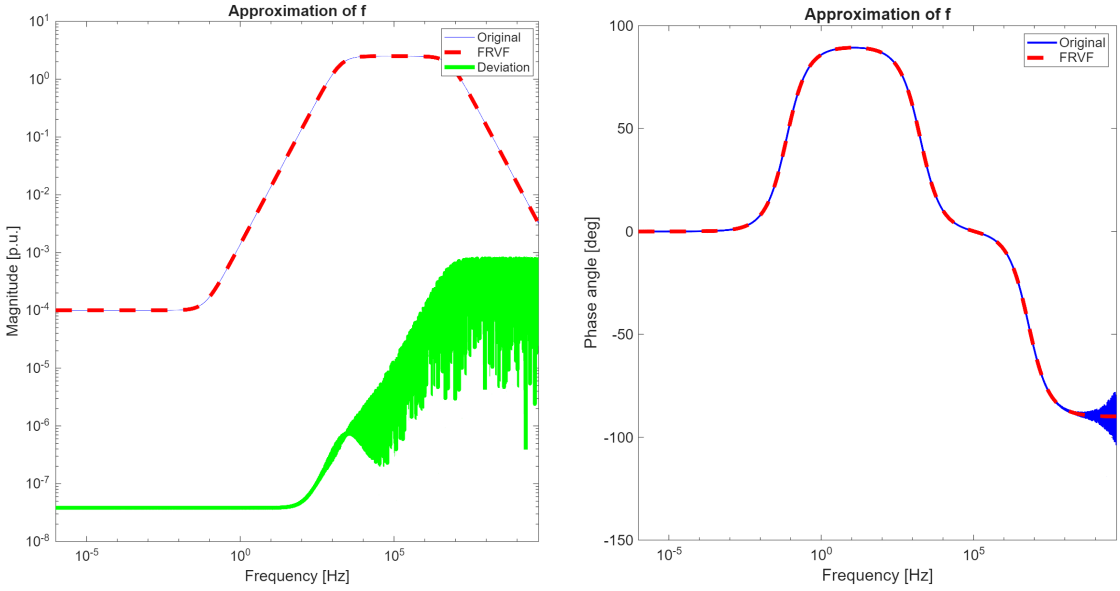


Figure 3.1: Magnetitude (in p.u.) and Phase of the admittance, measurements vs Vector Fitted model.

3.1.1 Generating Netlist

The only relevant output of the function is `SERY`, that is a struct containing the state space model of the Vector Fitted model. We can extract the State Space matrices:

Listing 3.2: Extracting the state space model

```
A=full(SERY.A);
B=full(SERY.B);
C=full(SERY.C);
D=full(SERY.D);
```

Then, we can generate the equivalent netlist by using the related function available in moodle

Listing 3.3: Generating the netlist

```
[filename,Port_node_name,Pole_node_name,ground_node_name]=
   fun_VFmodel2Netlist_MultiPort(A,B,C,D,'netlist.m');
```

where `filename` is a variable with the name of the .m file of the netlist.

This file will be something like:

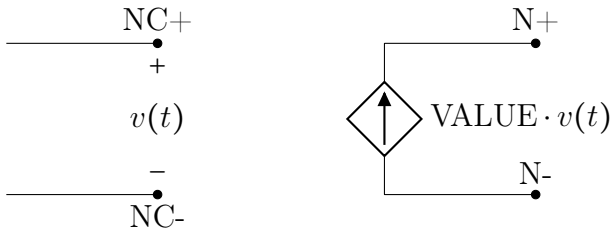
Listing 3.4: Generated the netlist

```
Gc1 1 2 3 2 1.000277e+08
Gd0 1 2 1 2 1.000000e-06
Gb1 2 3 1 2 1.000000e+00
R1 3 2 2.500727e-08
C1 3 2 1
Gc2 1 2 4 2 -2.843332e+04
Gb2 2 4 1 2 1.000000e+00
R2 4 2 8.797148e-05
C2 4 2 1
```

where Resistors, Capacitors, Voltage Controlled Current Sources are used. Voltage Controlled Current Sources are described as:

Gxxxxx N+ N- NC+ NC- VALUE

where:



3.1.2 Generating LTspice files

Once the netlist is generated, LTspice files can be generated. These files can then be uploaded to LTspice to use the Vector Fitted model in LTspice simulations.

To do that, we have to use the `generate_LTspice_netlist_MultiPort` function available in Moodle:

Listing 3.5: Generating the LTspice files

```
[lib_filename] = generate_LTspice_netlist_MultiPort(size(B,2)
    ,filename,'MyVFmodel',Port_node_name,ground_node_name);
```

where `generate_LTspice_netlist_MultiPort` is a function with no output in MATLAB, but it creates two files:

- `MyVFmodel.asy` is a file containing the graphical description of the component

- `MyVFmodel.lib` is a file containing the netlist description of the component

3.1.3 Loading and Using the Vector Fitted files in LTspice

By running `generate_ltspice_files` function some instructions are also generated in the command window to correctly place the LTspice files `MyVFmodel.asy` and `MyVFmodel.lib` so they can be loaded by the LTspice software:

```
The .lib file must be placed in:
   C:\Users\User\Documents\LTspice\lib\sub\

The .asy file must be placed in:
   C:\Users\User\Documents\LTspice\sym\MyParts\

(Create the directories if not present)

In LTspice:
 - You can find the symbol in "MyParts" when you add a new component
 - Use .include to load the .lib file, e.g.:
       .include C:\Users\User\Documents\LTspice\lib\sub\MyVFmodel.lib
```

Figure 3.2: Instructions to correctly place the LTspice files

Be careful to adapt the path accordingly.

Then, open LTpsice and create a new schematic. To load the Vector fitted model, click on the component icon, select the LTspice directory `MyParts`, and double click on MyVFmodel.

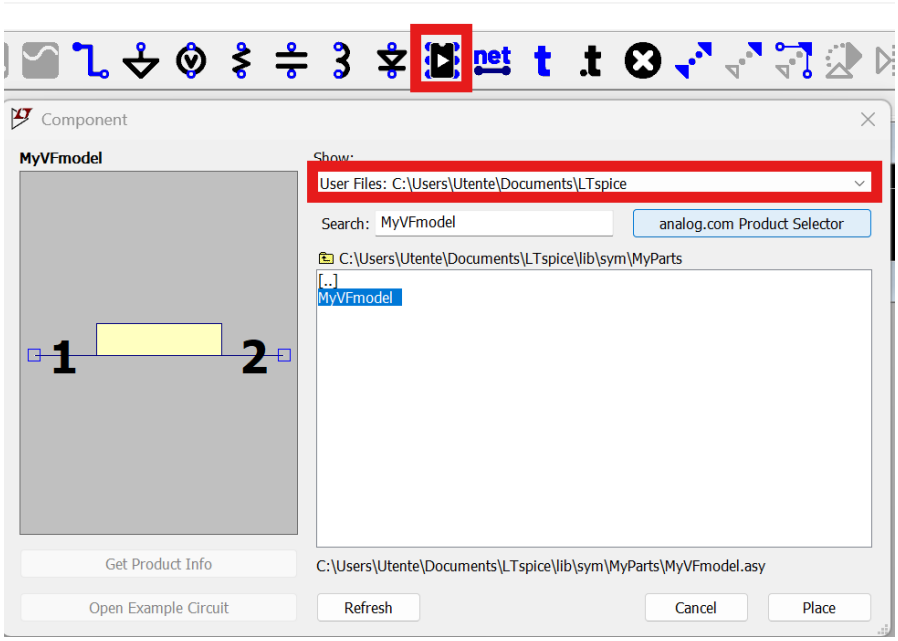


Figure 3.3: Add VF model to a schematic (note that the symbol may look different).

We can now place the component in the schematic. However, to allow LTspice to know the netlist of the component, we need to include the following Spice Directive:

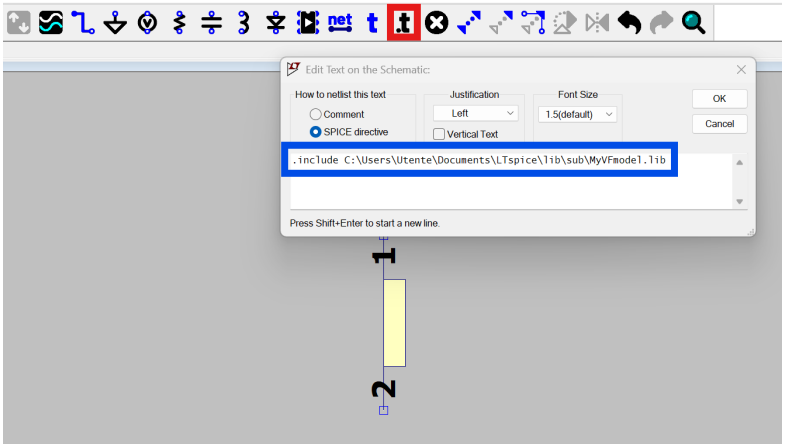


Figure 3.4: Add Spice Directive to load the netlist of the Vector Fitted model.

Again, be careful to adapt the path of the .include directive accordingly (it is the path of the .lib file).

Then, we can run an AC sweep simulation in the frequency range from 10^{-6} Hz to 5 GHz (the same range of the measurements) for validation.

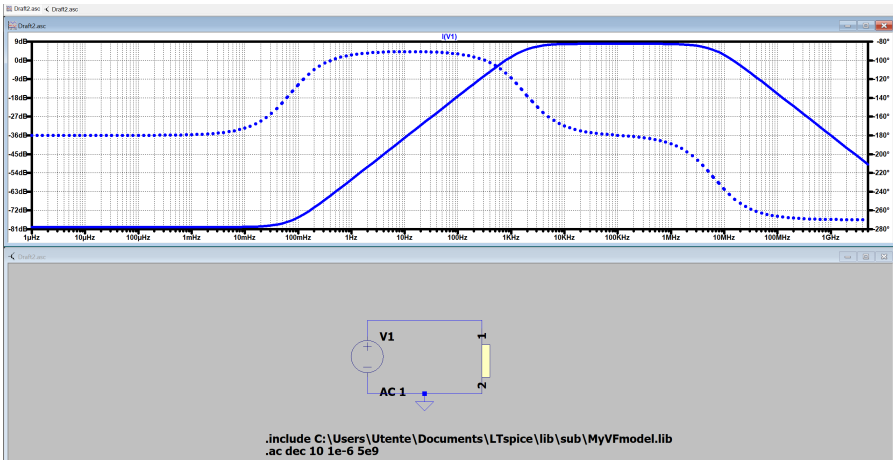


Figure 3.5: Vector Fitted Model simulated in LTspice in AC.

3.1.4 Using the Vector Fitted Model in the DC-DC Buck converter schematic

The same Vector Fitted Model of the Capacitor can now be used in place of the ideal capacitor model in the schematic of the DC-DC Buck Converter generated in Section 2.3.2.

By doing so, the following output voltage is obtained:

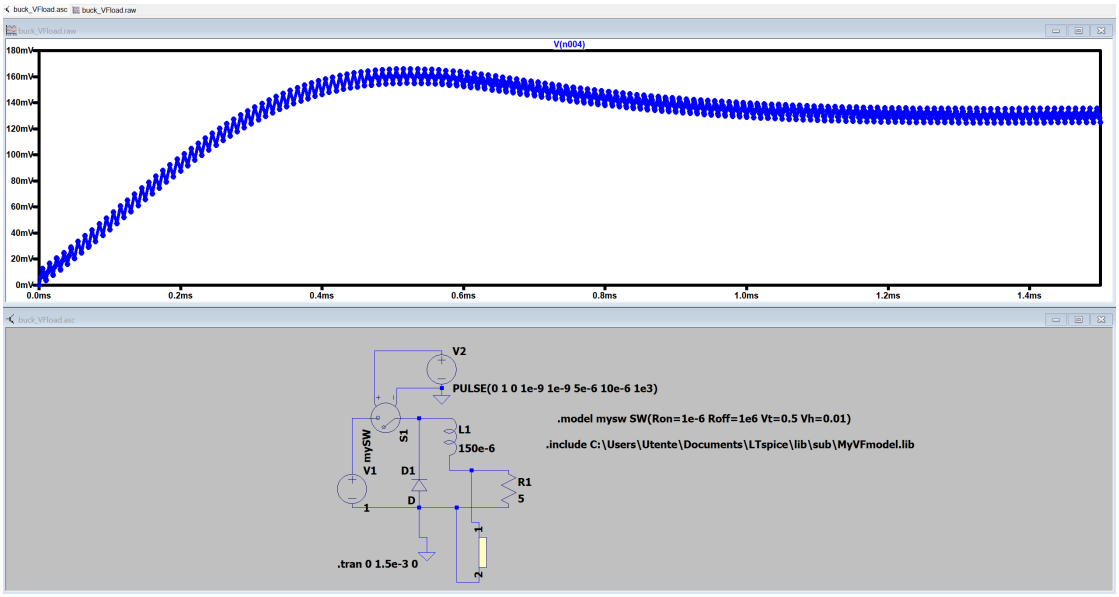


Figure 3.6: Simulation results of the Vector Fitted Model for the DC-DC Buck Converter.

As can be seen, because of the non-ideality of the capacitor model, the output voltage is much noisier than the ideal case.

By right-clicking on the graphics, we can also generate the frequency spectrum of the signal.

3.2 Assignments

3.2.1 Mandatory assignments

- Load in Moodle the MATLAB files (not including the VF library), the generated Vector Fitted files (.lib and .asy), and the LTspice schematics of Example 3.1.

3.2.2 Optional assignments

- (+ 0.5 point to be checked during oral examination) Repeat the procedure to include in the Buck DC-DC model also a real model of the inductor. The impedance data of the inductor is available in moodle.

- (+ 2 points to be checked during oral examination) Implement the Vector Fitting approach

Chapter 4

Finite Difference Method

4.1 Finite Difference in DC

4.1.1 Example 1

Solve the electrostatic problem:

$$\nabla^2 V = 0 \tag{4.1}$$

in the square domain Ω ($\Delta x = 1\,m$, $\Delta y = 1\,m$) shown in Figure 4.3, with the Finite Difference Method, by imposing the following boundary conditions:

- Dirichlet's conditions on Γ_{D_1}: $V_1 = 0V$, on Γ_{D_2}: $V_2 = +1V$
- Neumann's conditions on $\Gamma_{N_1} \cup \Gamma_{N_2}$

Assembling the system matrix A and the known term b

If we denote with N the number of nodes along the x and y axes, we have $N_u = N^2$ unknowns (one per each node), so the system matrix $\mathbf{A}$ is a square matrix of size $N_u \times N_u$, and the known term $\mathbf{b}$ is a column vector of size $N_u \times 1$.

For example, if we choose $N = 8$, we have $N_u = 64$ nodes, as shown in Figure 4.3. Therefore, the system matrix $\mathbf{A}$ is a square matrix of size 64×64, and the known term $\mathbf{b}$ is a column vector of size 64×1.

First of all, we allocate $\mathbf{A}$ as an empty sparse matrix, using the Matlab command `spalloc(Nu,Nu,5*Nu)`. Similarly, we allocate the known term $\mathbf{b}$ as an empty sparse vector, using the Matlab command `spalloc(Nu,1,Nu)`.

Then, we have to assemble the system matrix $\mathbf{A}$ and the known term $\mathbf{b}$, taking into account the position of each node (i.e., interior, Dirichlet's boundary, Neumann's boundary).

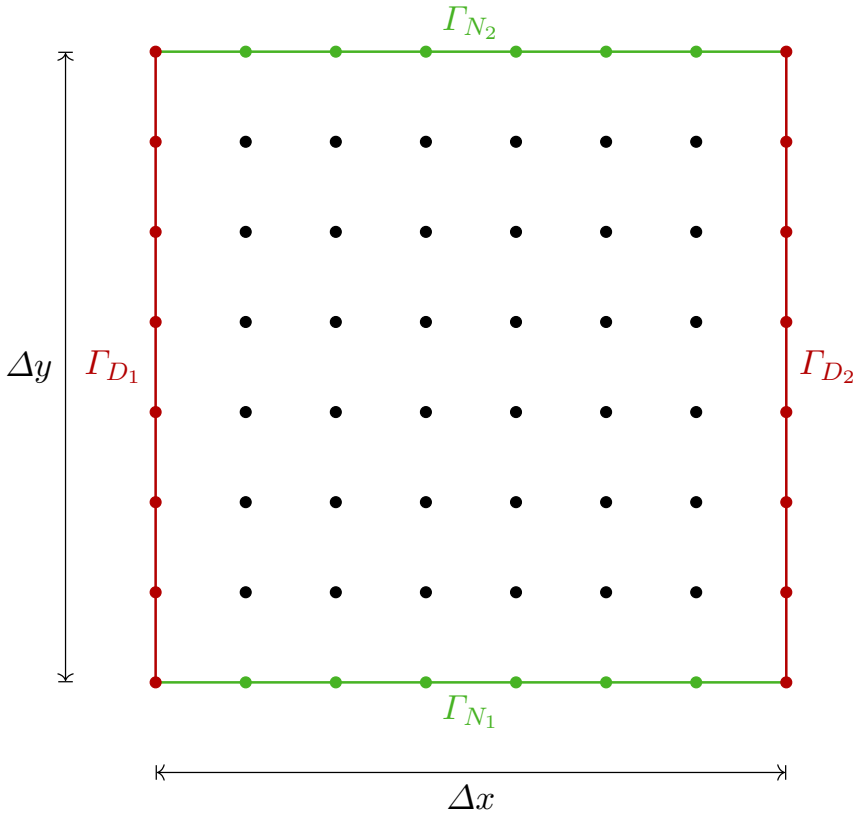


Figure 4.1: Numerical domain of the electrostatic problem $\nabla^2 V = 0$.

As far as the interior nodes and Neumann's boundary nodes are concerned, we can assemble the system matrix $\mathbf{A}$ by following the procedure in Algorithm 14.

As far as the Dirichlet's boundary conditions are concerned, for those nodes which sit on the left boundary (Γ_{D_1}) we can follow Algorithm 15, and for those nodes that sit on the right boundary (Γ_{D_2}) we can follow Algorithm 16.

Numerical solution vs reference solution

The reference solution is $V(x, y) = x, \quad 0 \leq x \leq 1$.

The equipotential lines of the numerical solution are shown in Figure 4.2. Note that the numerical solution is exactly up to machine precision because Taylor's expansions of $V(x, y)$ along x and y have got $O(a^4) = 0$.

Algorithm 14 Interior nodes and Neumann's boundary nodes

```
for i ← 2, N − 1 do
    for j ← 1, N do
        k ← (i − 1)N + j
        A(k, k) ← 4
        A(k, k + N) ← −1
        A(k, k − N) ← −1
        if (j > 1) & (j < N) then                ▷ Interior nodes
            A(k, k + 1) ← −1
            A(k, k − 1) ← −1
        end if
        if (j == 1) then                         ▷ Bottom boundary nodes
            A(k, k + 1) ← −2
        end if
        if (j == N) then                         ▷ Top boundary nodes
            A(k, k − 1) ← −2
        end if
    end for
end for
```

Algorithm 15 Dirichlet's boundary nodes on Γ_{D_1}

```
for j ← 1, N do
    k ← j
    A(k, k) ← 1
    b(k) ← V_1
end for
```

Algorithm 16 Dirichlet's boundary nodes on Γ_{D_2}

```
for j ← 1, N do
    k ← (N − 1)N + j
    A(k, k) ← 1
    b(k) ← V_2
end for
```

4.1.2 Example 2

Solve the electrostatic Poisson's problem:

$$\nabla^2 V = f \tag{4.2}$$

in the square domain Ω ($\Delta x = 1\,m$, $\Delta y = 1\,m$) shown in Figure 4.3, with the Finite Difference Method, by imposing Dirichlet's boundary conditions on $\Gamma_{D_1} \cup \Gamma_{D_2} \cup$

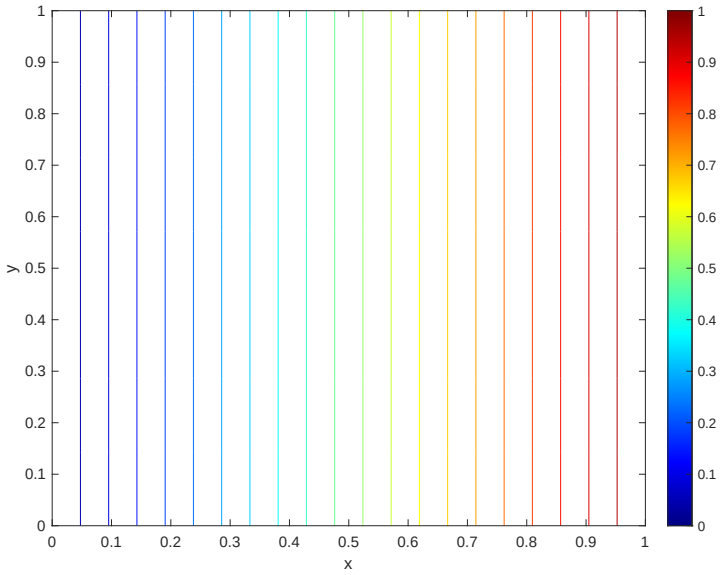


Figure 4.2: Numerical solution of the electrostatic problem (Laplace equation) in the square domain Ω ($\Delta x = 1\,m$, $\Delta y = 1\,m$).

$\Gamma_{D_3} \cup \Gamma_{D_4}$: $V_1 = 0V$.

Consider the following expressions of $f(x, y)$:

1. $f_1(x, y) = 2\,(x^2 - x + y^2 - y)$
2. $f_2(x, y) = -8\pi^2 \cdot \sin(2\pi x) \cdot \sin(2\pi y)$

Assembling the system matrix A and the known term b

Similarly to 4.1.1, first of all, we allocate **A** as an empty sparse matrix, using the Matlab command `spalloc(Nu,Nu,5*Nu)`. Similarly, we allocate the known term **b** as an empty sparse vector, using the Matlab command `spalloc(Nu,1,Nu)`.

Then, we have to assemble the system matrix **A** and the known term **b**, taking into account the position of each node (i.e., interior, Dirichlet's boundary).

As far as the interior nodes are concerned, we can assemble the system matrix **A** by following the procedure in Algorithm 17. If we denote with N the number of nodes along the x and y axes, the square side is $a = \Delta x/(N-1)$.

In this example, we consider $f(x, y) = f_1(x, y)$.

As far as the Dirichlet's boundary conditions are concerned, we can follow Algorithm 18 for the left boundary (Γ_{D_1}) and repeat the procedure for the other boundary parts (Γ_{D_2}, Γ_{D_3}, and Γ_{D_4}).

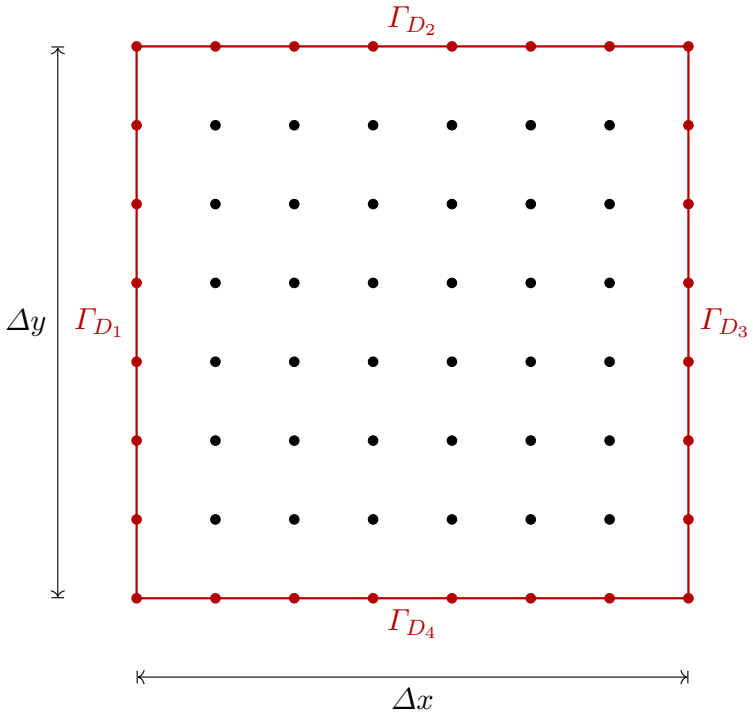


Figure 4.3: Numerical domain of the electrostatic problem $\nabla^2 V = f$.

Algorithm 17 Interior nodes

```
for i ← 2, N − 1 do
    for j ← 2, N − 1 do
        k ← (i − 1)N + j
        x(k) ← (i − 1) ∗ a
        y(k) ← (j − 1) ∗ a
        A(k, k) ← 4
        A(k, k + N) ← −1
        A(k, k − N) ← −1
        A(k, k + 1) ← −1
        A(k, k − 1) ← −1
        b(k) =← −a² ∗ 2 ∗ [x(k)² − x(k) + y(k)² − y(k)]        ▷ known term
    end for
end for
```

Numerical solution vs reference solution

The reference (analytical) solutions of problems 1 and 2 are, respectively:

Algorithm 18 Dirichlet's boundary nodes on Γ_{D_1}

```
for k ← 1, N do
    A(k, k) ← 1
end for
```

1. $V(x, y) = (x^2 - x) \cdot (y^2 - y)$
2. $V(x, y) = \sin(2\pi x) \cdot \sin(2\pi y)$

In Figure 4.4 we compare the numerical solution ($N = 20$) with the reference solution of problem 1. Note that the numerical solution is exactly up to machine precision because Taylor's expansions of $V(x, y)$ along x and y have got $O(a^4) = 0$.

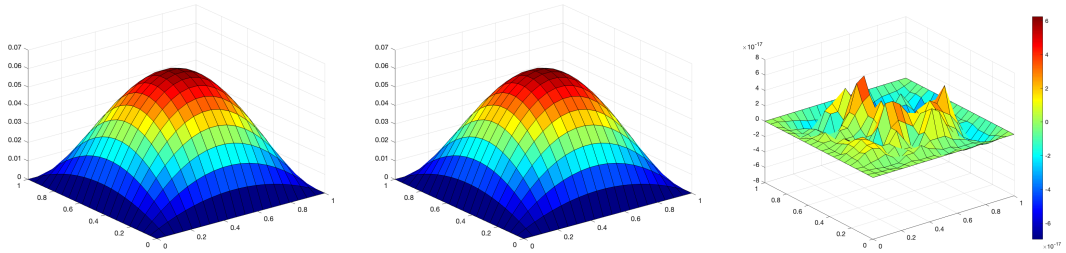


Figure 4.4: Solution of the electrostatic problem $\nabla^2 V = 2 \cdot (x^2 - x + y^2 - y)$. Left: numerical. Middle: reference. Right: error.

In Figure 4.5 we compare the numerical solution ($N = 20$) with the reference solution of problem 2. Note that the numerical solution in this case is much less accurate than before (maximum error $9.10 \cdot 10^{-3}$). To increase the accuracy we should increase the number of nodes N (i.e., reduce the node spacing a).

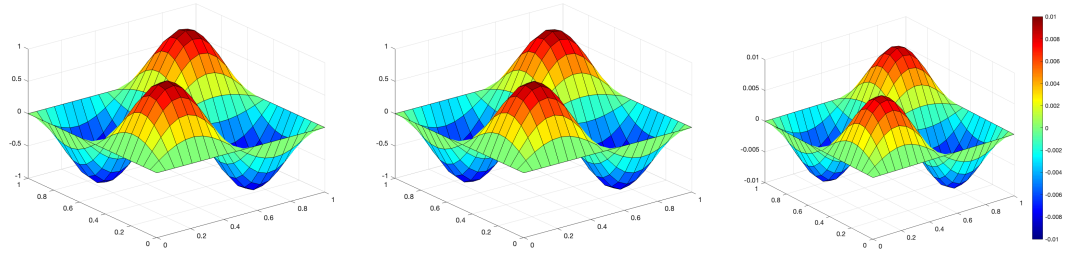


Figure 4.5: Solution of the electrostatic problem $\nabla^2 V = -8\pi^2 \cdot \sin(2\pi x) \cdot \sin(2\pi y)$. Left: numerical. Middle: reference. Right: error.

In Figure 4.6 we compare the error of the numerical solution for $N = [20, 40, 80]$. The maximum error for the three cases is $[9.10, 2.16, 0.53] \cdot 10^{-3}$, respectively.

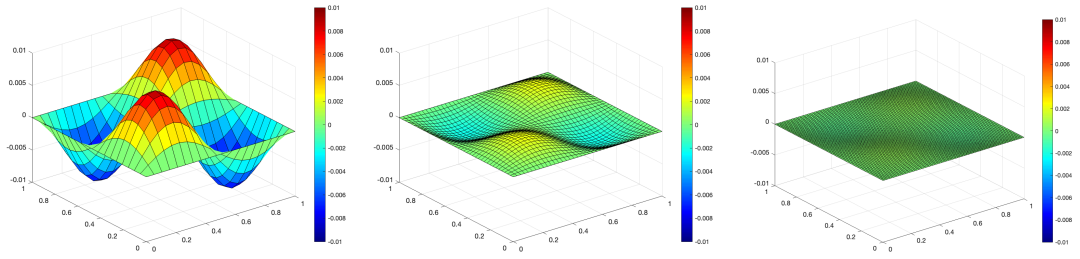


Figure 4.6: Solution of the electrostatic problem $\nabla^2 V = -8\pi^2 \cdot \sin(2\pi x) \cdot \sin(2\pi y)$. Errors with $N = 20$ (left), $N = 40$ (middle), $N = 80$ (right).

4.2 FDM in Time

4.2.1 Example 1

Solve the time-dependent heat problem described by the following PDE:

$$-\nabla^2 T + \frac{\rho c_p}{\lambda}\frac{\partial T}{\partial t} = \frac{p_{th}}{\lambda} \tag{4.3}$$

in the one-dimensional (1D) domain Ω shown in Figure 4.7, with the Dirichlet's boundary conditions:

- $T(0,t) = 0 \qquad \forall t > 0$
- $T(1,t) = 1 \qquad \forall t > 0$

and the initial condition:

- $T(x,0) = 0 \qquad x \in [0,1]$

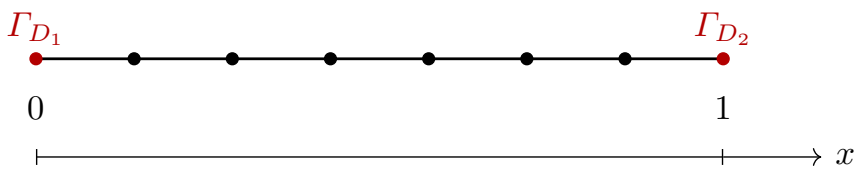


Figure 4.7: Numerical domain $[0,1]$ of the heat problem.

The material is homogeneous and the following values are assumed for the material parameters and the heat source in (4.3):

- Mass density $\rho = 1\,kg/m^3$
- Specific heat capacity at constant pressure $c_p = 20\,Jkg^{-1}K^{-1}$
- Thermal conductivity $k = 1\,Wm^{-1}K^{-1}$
- Heat source density $p_{th} = 0\,W/m^3$.

Soution with the FDM

Since in this example $p_{th} = 0$, the 1D time-dependent heat problem can be written as:

$$-\frac{\partial^2 T}{\partial x^2} + \alpha \frac{\partial T}{\partial t} = 0 \qquad \text{with} \quad \alpha = \frac{\rho c_p}{\lambda} \tag{4.4}$$

The numerical domain Ω (i.e., $x \in [0,1]$), is subdivided into $N-1$ intervals of width $a = 1/(N-1)$, where N is the number of nodes.

In this 1D problem, for any interior node $(i) \in \Omega$, we can write:

$$2\,T_{i,k} - T_{i-1,k} - T_{i+1,k} + a^2 \alpha \left.\frac{\partial T}{\partial t}\right|_{i,k} = 0 \tag{4.5}$$

where the index (k) is used to identify the "position" in time $t_k = k\Delta t$, being Δt the constant time-step used in the numerical solution of 4.5.

Writing this equation for each node, including the boundary nodes (their equations will be modified in a second step), and assembling all together, we get the following system of ordinary differential equations (ODEs):

$$[M_1]\{T_k\} + [M_2]\left\{\frac{\partial T_k}{\partial t}\right\} = 0 \tag{4.6}$$

The *stiffness matrix* $\mathbf{M_1}$ is a square matrix of size $N \times N$ with the following structure (tridiagonal matrix):

$$\mathbf{M}_1 = \begin{bmatrix} 2 & -1 & & & & & \\ -1 & 2 & -1 & & & & \\ & -1 & 2 & -1 & & & \\ & & & \ddots & & & \\ & & & -1 & 2 & -1 & \\ & & & & -1 & 2 & -1 \\ & & & & & -1 & 2 \end{bmatrix} \tag{4.7}$$

It is worth mentioning that in the first and last rows, which correspond to the boundary nodes at $x = 0$ and $x = 1$, we have only two non-zero entries, instead of three, because there are no left and right neighbors, respectively. However, the equations which involve the first and last rows will be modified later on when imposing the boundary conditions.

The *mass matrix* $\mathbf{M_2}$ is a diagonal matrix of size $N \times N$:

$$\mathbf{M}_2 = \begin{bmatrix} \alpha a^2 & & & & & & & \\ & \alpha a^2 & & & & & & \\ & & \alpha a^2 & & & & & \\ & & & \ddots & & & & \\ & & & & \alpha a^2 & & & \\ & & & & & \alpha a^2 & & \\ & & & & & & \alpha a^2 & \end{bmatrix} = \alpha a^2 \, \mathbb{I} \tag{4.8}$$

where $\mathbb{I}$ is the identity matrix of size $N \times N$. Note that not only the material parameters are included in $\mathbf{M}_2$, being $\alpha = \rho c_p/\lambda$, but also the step size a (node spacing) has a role, through the term a^2 (discretization of the spatial derivative).

θ method

Once the matrices $\mathbf{M_1}$ and $\mathbf{M_2}$ are assembled, the heat equation can be solved in the time domain by exploiting the θ method, which in this case reads:

$$\mathbf{K}_1 \mathbf{T}_{n+1} = \mathbf{K}_2 \mathbf{T}_n = \mathbf{q}_n \tag{4.9}$$

with

$$\mathbf{K}_1 = \theta \mathbf{M}_1 + \frac{1}{\Delta t}\mathbf{M}_2 \qquad \mathbf{K}_2 = -(1-\theta)\mathbf{M}_1 + \frac{1}{\Delta t}\mathbf{M}_2$$

In (4.9), we have the classical time-stepping scheme with the unknowns ($\mathbf{T}_{n+1}$ at time t_{n+1}) at the left-hand side, and the known values ($\mathbf{q}_n$ at time t_n) at the right-hand side.

Imposition of Boundary Conditions (BCs)

In this example, Dirichlet's boundary conditions are imposed in the first ($k = 1$) and last ($k = N$) nodes. A simple way to impose Dirichlet's boundary conditions is to modify the matrix $\mathbf{K}_1$ and the right-hand side $\mathbf{q}_n$ as follows:

- assign $\mathbf{K}_1(k,k) = 1$ for $k \in \{1, N\}$
- assign $\mathbf{K}_1(k, k \pm 1) = 0$ for $k \in \{1, N\}$
- assign $\mathbf{q}_n(k) = T_k^*$ for $k \in \{1, N\}$

It is important to note that $\mathbf{q}_n$ changes at any iteration, since $\mathbf{q}_n = \mathbf{K}_2 \mathbf{T}_n$ and $\mathbf{T}_n$ varies in time. Thus, we need to update $\mathbf{q}_n$ at any iteration and then set $\mathbf{q}_n(k) = T_k^*$ to keep the correct constraint on Dirichlet's boundary nodes.

Matlab coding

In this section, some hints about the practical implementation of the numerical procedure are given.

The space interval $I = [0,1]$ is discretized with $N = 20$ nodes equally spaced, while the time interval $[0,5]$ has $N_t = 100$ equally-spaced points, and so we get:

$$\Delta x = \frac{1}{N-1}, \quad \Delta t = \frac{5}{N_t - 1}$$

The tridiagonal matrix in (4.7) can be constructed in MATLAB with the algorithm 19, by exploiting the properties of the *diag* command. The same command can also be used to construct $\mathbf{M_2}$. Note that in MATLAB a single command can be used to construct a tridiagonal matrix as explained in algorithm 20.

Algorithm 19 M1 construction – Ver.1

Input: Number of discretization nodes nx of I
Output: Matrix $\mathbf{M_1}$

1: $\mathbf{M_1} \leftarrow \texttt{diag}(2*\texttt{ones}(1,N))+\texttt{diag}(-1*\texttt{ones}(1,N-1),1)+\texttt{diag}(-1*\texttt{ones}(1,N-1),-1)$

Algorithm 20 M1 construction – Ver.2

Input: Number of discretization nodes N of I
Output: Matrix $\mathbf{M_1}$

1: $v \leftarrow \texttt{ones}(N,1)$
2: $\mathbf{M_1} \leftarrow \texttt{spdiags}([-v\ 2v\ -v], -1:1, N, N)$

Algorithm 21 Theta method implementation

Input: $N, Nt, \theta, \Delta t, \mathbf{M_1}, \mathbf{M_2}, \mathbf{x}_1, \mathbf{q}$
Output: Matrix $\mathbf{U}$

1: $\mathbf{T} \leftarrow \texttt{zeros}(N, Nt)$
2: $\mathbf{T}(:,1) \leftarrow \mathbf{x}_1$
3: $\mathbf{K_1} \leftarrow \theta * \mathbf{M_1} + \mathbf{M_2}/\Delta t;$
4: $\mathbf{K_2} \leftarrow -[(1-\theta) * \mathbf{M_1} + \mathbf{M_2}/\Delta t];$
5: Modify $\mathbf{K_1}$ to impose BCs
6: **for** k=2:Nt **do**
7: $\quad \mathbf{q} \leftarrow \mathbf{K}_2 * \mathbf{T}(:,k-1);$
8: $\quad$ Modify $\mathbf{q}$ to impose BCs
9: $\quad \mathbf{T}(:,k) \leftarrow \mathbf{K}_1 \backslash \mathbf{q};$ ▷ solve with Backslash command
10: **end for**

With these matrices, we can construct $\mathbf{K_1}$ and $\mathbf{K_2}$ and begin the solution step. The initial condition states that $\mathbf{T}$ at the first time step ($k = 1$) is zero, so the iteration starts from $k = 2$, as described in algorithm 21. The solutions are stored in a matrix $\mathbf{T}$ of dimensions $N \times N_t$, where each column corresponds to the solution $\mathbf{T}_k$ at k-th time step.

The solution for each spatial and temporal discretization node can be graphically represented in MATLAB by using the commands listed in 4.1, which generates the image given in Fig. 4.8.

Listing 4.1: Surface plot

```
>> surf(T','EdgeColor','none');
>> xlabel('space nodes');
>> ylabel('time nodes');
>> zlabel('Temperature');
>> colormap hot;
>> colorbar;
>> title('Thermal behaviour');
```

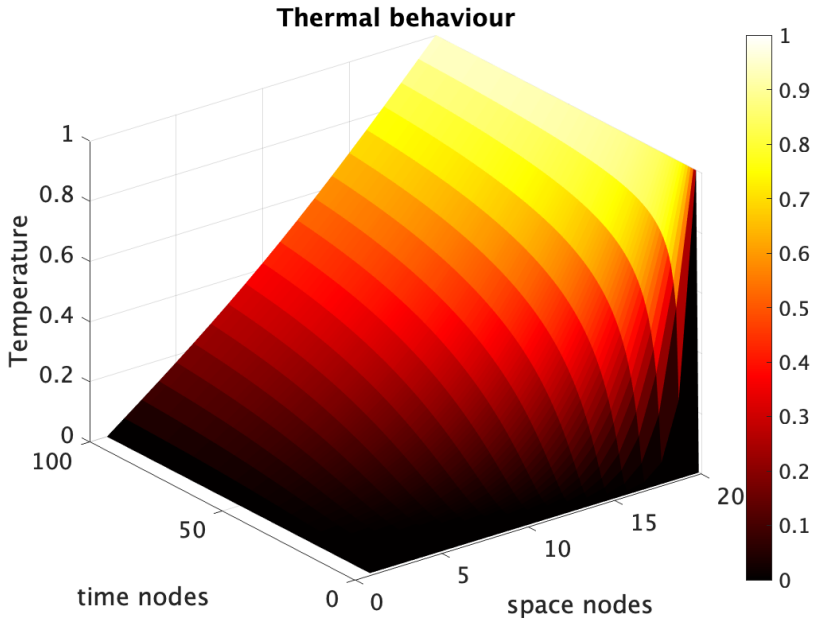


Figure 4.8: Solution $T(x,t)$ for each spatial and temporal discretization node.

Chapter 5

Finite Element Method

5.1 Electrostatic 2D problem

Solve the electrostatic problem (Laplace equation):

$$\nabla \cdot (\varepsilon \nabla V) = 0 \tag{5.1}$$

in the numerical domain Ω ($\Delta x = 1\,m$, $\Delta y = 1\,m$) shown in Fig. 5.1, with the Finite Element Method (FEM), by imposing the following boundary conditions:

- Dirichlet's condition on Γ_{D_1}: $V_1 = 1V$
- Dirichlet's condition on Γ_{D_2}: $V_2 = 0$
- Homogeneous Neumann's conditions on Γ_∞

The material in Ω is homogeneous with $\varepsilon = \varepsilon_0 = 8.854 \cdot 10^{12}\,F/m$ (vacuum/air).

5.1.1 FEM implementation

The main steps of the numerical implementation of a FEM code are described in the following. The script to be realized in MATLAB has the general structure listed in Algorithm 22.

Algorithm 22 Structure of the FEM code

1: Import the mesh data
2: Build the (stiffness) matrix **K**
3: Build the (known term) vector **b**
4: Solve the linear system **KV** = **b** for **V**
5: Plot the solution **V**

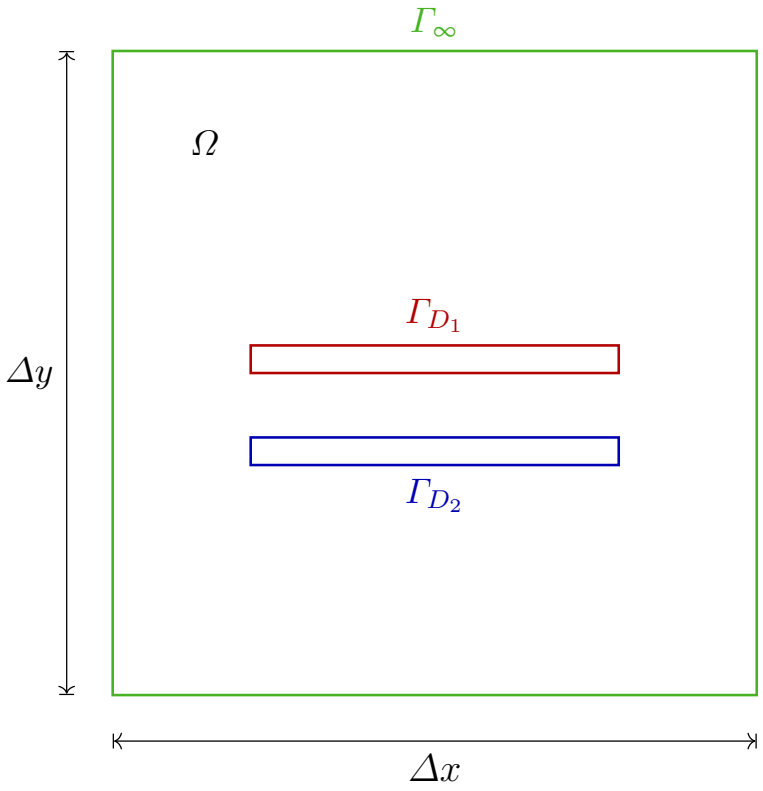


Figure 5.1: Numerical domain of the electrostatic problem $\nabla \cdot (\varepsilon \nabla V) = 0$.

5.1.2 Reading the mesh data

The first step is to import the mesh data and is performed utilizing the provided function `readmesh`:

```
[nodes,triang,mat,vD,gammaD,rhsFun,rhsId,err]=readmesh(filename)
```

- **input**
  - filename: file name containing the problem data (without any extension)
- **output**
  - nodes: $nn \times 2$ matrix of nodal coordinates
  - triang: $nt \times 3$ matrix of triangles connectivity.
    triang(i,1:3) gives the indices of the nodes forming the i-th triangle[1]

[1] It is worth mentioning that the indices are always numbered counterclockwise.

- mat: $nt \times 1$ array of materials in the domain
- vD: Dirichlet values (i.e., those imposed on Γ_{D_1} and Γ_{D_2})
- gammaD: indices of nodes in the Dirichlet boundary
- rhsFun: the value of RHS term in triangle barycenter
- rhsId: index of triangles where $RHS \neq 0$
- err: error flag

5.2 Building the stiffness matrix K

The stiffness matrix $\mathbf{K}$ is constructed with the $\mathbb{P}1$ (linear) element interpolating functions. In particular, for a generic element e_j of the mesh, the solution $V(x, y)$ is approximated as:

$$\hat{V}(x, y) = \sum_{k=1}^{3} V_k \; {}_1N_k^{e_j}(x, y) \tag{5.2}$$

where ${}_1N_k^{e_j}(x, y)$ are the nodal basis functions of the form:

$${}_1N_k^{e_j}(x, y) = (a_k + b_k x + c_k y) \quad \text{for } k \in \{1, 2, 3\}. \tag{5.3}$$

The entries of the local 3×3 system matrix $\mathbf{K}^{e_j}$ are given by:

$$K_{ij}^{e_j} = \left(b_i^{e_j} \, b_j^{e_j} + c_i^{e_j} \, c_j^{e_j}\right) \varepsilon \, \Delta_{e_j} \tag{5.4}$$

where Δ_{e_j} is the area of the triangle e_j.

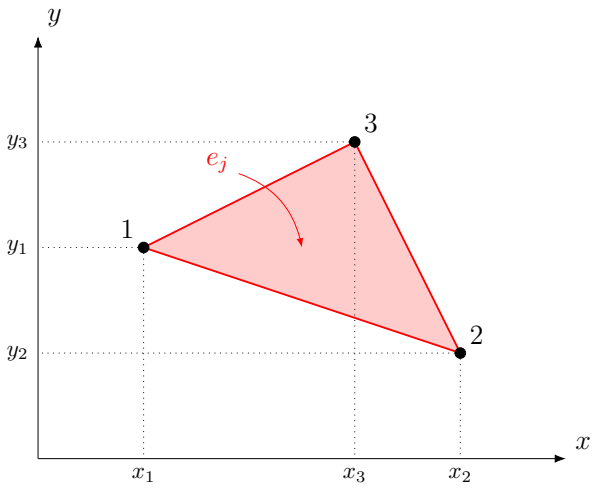


Figure 5.2: Nodes' coordinates (x_i, y_i) in local numbering.

Referring to the triangle in Figure 5.2, the coefficients b_k and c_k collected in arrays **b** and **c** are constructed as:

$$\begin{aligned} b_1 &= \frac{y_2 - y_3}{2\Delta_{e_j}} \qquad c_1 = \frac{x_3 - x_2}{2\Delta_{e_j}} \\ b_2 &= \frac{y_3 - y_1}{2\Delta_{e_j}} \qquad c_2 = \frac{x_1 - x_3}{2\Delta_{e_j}} \\ b_3 &= \frac{y_1 - y_2}{2\Delta_{e_j}} \qquad c_3 = \frac{x_2 - x_1}{2\Delta_{e_j}} \end{aligned} \tag{5.5}$$

For example, in local numbering, for $K_{12}^{e_j}$ we get:

$$K_{12}^{e_j} = \left(b_1^{e_j}\, b_2^{e_j} + c_1^{e_j}\, c_2^{e_j}\right) \varepsilon\, \Delta_{e_j} = \frac{[(x_3 - x_2)(x_1 - x_3) + (y_2 - y_3)(y_3 - y_1)]}{4\Delta_{e_j}}\, \varepsilon \tag{5.6}$$

The pseudo-code for constructing the stiffness matrix is listed in Algorithm 24. It is worth mentioning that matrix **K** is *sparse* since the number of non-zero elements is much less the dimension of the matrix (N^2).

Algorithm 23 Stifness matrix **K** assembly

Input: triangles connectivity matrix *triang*, node coordinates *nodes*, permittivity *epsilon*

Output: Matrix **K**, areas of triangles *areas*

```
nn: number of nodes
nt: number of triangles
K ← zeros(nn, nn)                                   ▷ Initialize matrix
for e=1:nt do
    for i=1:3 do
        irow ← triang(e, i)                         ▷ Gloabal row index of matrix K
        for j=1:3 do
            jcol ← triang(e, j)                     ▷ Gloabal column index of matrix K
            nodes_loc ← coordinates of the e-th triangle
            Construct b and c vectors
            Compute area of the e-th triangle
            areas(e) ← area
            K(irow, jcol) ← K(irow, jcol)+0.25*(b(i)*b(j)+c(i)*c(j))/area*
epsilon
        end for
    end for
end for
```

5.2.1 Building the RHS vector

As can be seen in (5.1) there is no source term, thus, the RHS (Right Hand Side) vector $\mathbf{b}$ is all-zero in this particular case.

5.2.2 Solution of the linear system

Once the stiffness $\mathbf{K}$ and the RHS vector $\mathbf{b}$ are built, we can proceed with the solution of the linear system $\mathbf{KV} = \mathbf{b}$ with the addition of the boundary conditions.

As far as Neumann's boundary conditions are concerned, we know from the theory that they are "naturally" imposed on boundary nodes.

As far as the Dirichlet's boundary conditions are concerned, suppose that the indices of the nodes in $\Gamma_1 \cup \Gamma_2$ are collected in the array **gammaD** and the corresponding known values of $\mathbf{V}$ collected in the array **vD**.

By splitting the array of unknowns $\mathbf{V}$ as:

$$\mathbf{V} = \left[\begin{array}{c} \mathbf{V}_f \\ \hline \mathbf{V}_d \end{array}\right], \tag{5.7}$$

where the suffix f stands for "free", and the suffix d stands for "Dirichlet", the stiffness matrix can be split accordingly as:

$$\mathbf{K} = \left[\begin{array}{c|c} \mathbf{K}_{ff} & \mathbf{K}_{fd} \\ \hline \mathbf{K}_{df} & \mathbf{K}_{dd} \end{array}\right]. \tag{5.8}$$

The same trick is done with the RHS vector:

$$\mathbf{b} = \left[\begin{array}{c} \mathbf{b}_f \\ \hline \mathbf{b}_d \end{array}\right]. \tag{5.9}$$

The system of equations is rewritten as:

$$\left[\begin{array}{c|c} \mathbf{K}_{ff} & \mathbf{K}_{fd} \\ \hline \mathbf{K}_{df} & \mathbf{K}_{dd} \end{array}\right] \left[\begin{array}{c} \mathbf{V}_f \\ \hline \mathbf{V}_d \end{array}\right] = \left[\begin{array}{c} \mathbf{b}_f \\ \hline \mathbf{b}_d \end{array}\right]. \tag{5.10}$$

We are only interested in obtaining $\mathbf{V}_f$ since $\mathbf{V}_D$ are already known.

Focusing only on the first block row of (5.10), we get:

$$\mathbf{K}_{ff}\mathbf{V}_f + \mathbf{K}_{fd}\mathbf{V}_d = \mathbf{b}_f \tag{5.11}$$

and so $\mathbf{V}_f$ are computed as the solution of:

$$\mathbf{K}_{ff}\mathbf{V}_f = \mathbf{b}_f - \mathbf{K}_{fd}\mathbf{V}_d \tag{5.12}$$

The pseudo-code for constructing the stiffness matrix is listed in Algorithm 24. It is worth mentioning that matrix $\mathbf{K}$ is *sparse*, since the the number of non-zero elements is much less the dimension of the matrix (N^2).

Algorithm 24 Stifness matrix **K** assembly

Input: triangles connectivity matrix *triang*, node coordinates *nodes*, permittivity *epsilon*

Output: Matrix **K**, areas of triangles *areas*

nn: number of nodes
nt: number of triangles
$\mathbf{K} \leftarrow \texttt{zeros}(nn, nn)$ ▷ Initialize matrix
for e=1:nt **do**
    $nodes_{loc} \leftarrow$ coordinates of the e-th triangle
    Construct **b** and **c** vectors
    Compute *area* of the e-th triangle
    $areas(e) \leftarrow area$
    for i=1:3 **do**
        $irow \leftarrow triang(e, i)$ ▷ Global row index of matrix **K**
        for j=1:3 **do**
            $jcol \leftarrow triang(e, j)$ ▷ Global column index of matrix **K**
            $\mathbf{K}(irow, jcol) \leftarrow \mathbf{K}(irow, jcol) + 0.25 * (b(i) * b(j) + c(i) * c(j))/area * epsilon$
        end for
    end for
end for

5.2.3 Visualization of results

The solution **V** of the problem can be visualized on the mesh with the `patch` command on MATLAB as listed in 5.1.

To do so in the main script the following lines are added:

Listing 5.1: Lines for visualizing solution

```
>> figure % creates a figure object
>> patch('Faces',triang,'Vertices',nodes,'CData',V,'
   Facecolor','flat')
>> axis equal
>> title('Electric potential')
>> xlabel('x')
>> ylabel('y')
>> colormap turbo % type of colorbar
>> colorbar % to visualize the colorbar
```

The reference solution of the Laplace electrostatic problem computed with the commercial FEM software COMSOL Multiphysics is given in Fig. 5.3. The electric field map evaluated as $\mathbf{E} = -\nabla V$ is illustrated in Fig. 5.4

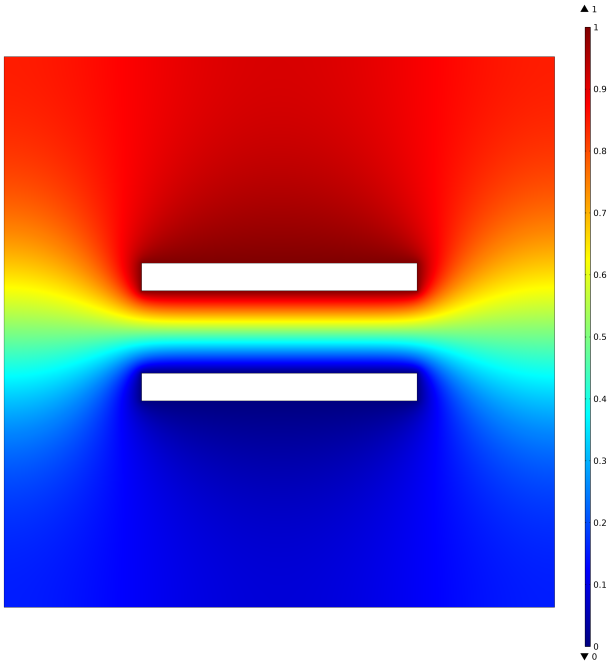


Figure 5.3: Reference solution (COMSOL): colormap of $V\,[V]$.

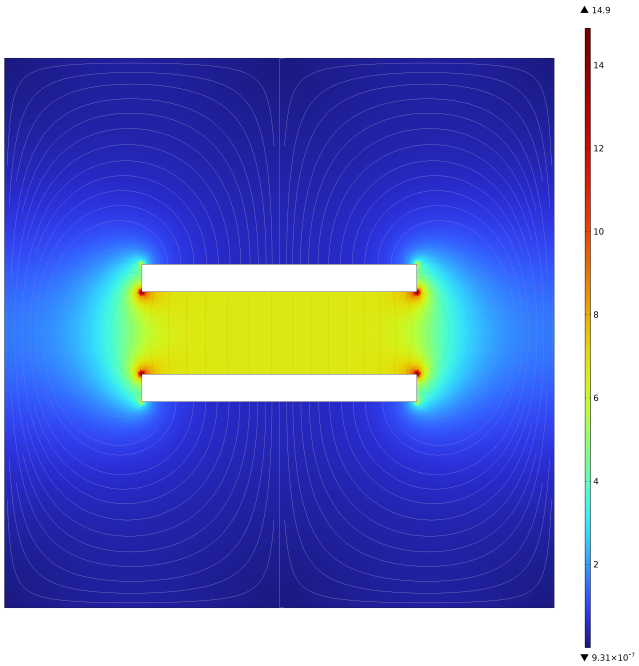


Figure 5.4: Reference solution (COMSOL): colormap of the norm of the electric field $E\,[V/m]$ and field lines.

5.3 Magnetostatic 2D problem

Solve with the Finite Element Method (FEM) the magnetostatic problem:

$$\nabla \cdot (\nu \nabla A_z) = -J_z \tag{5.13}$$

in the planar numerical domain Ω (translational symmetry) shown in Figure 5.5, with Dirichlet's Boundary Conditions on $\Gamma = \partial\Omega$: $A_z = 0\, Am$.

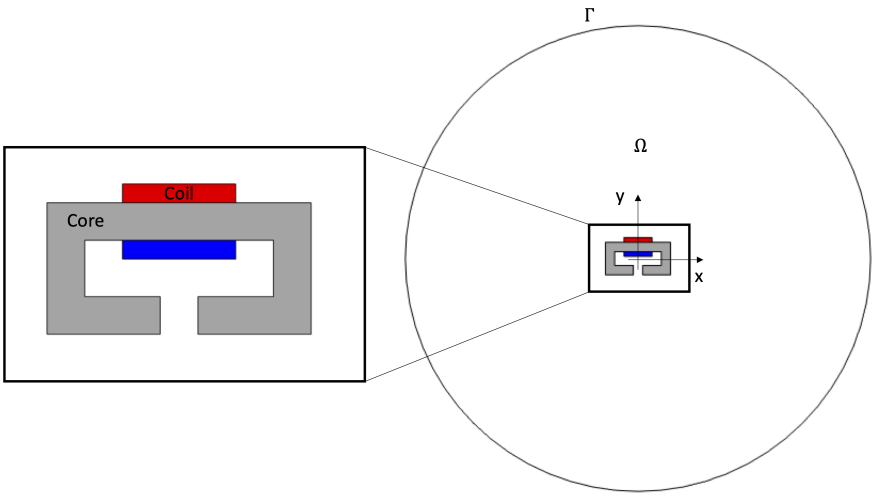


Figure 5.5: Computational domain of magnetostatic problem.

The domain examined in this example corresponds to a C-shape actuator (i.e., a coil wound around an iron core) embedded in an air circle.

The parameters used in the numerical analysis are listed in Table 5.1. $\nu = 1/\mu$ is the reluctivity and $\mu = \mu_0\mu_r$ is the permeability of the materials in Ω, with $\mu_0 = 4\pi \cdot 10^{-7}\, H/m$ (vacuum permeability).

Domain	μ_r	**Color**	**Imposed Current Density**
Air	1	white	-
Core	2024	grey	-
Coil	1	red	10^7 A/m^2
Coil	1	blue	-10^7 A/m^2

Table 5.1: Domain parameters used in the numerical analysis.

Numerical solution

The skeleton of the numerical code for the solution of (5.13) is the same as the one used for the electrostatic problem. Here the array of unknowns collects the

nodal values of A_z. The only remarkable difference arises in the construction of the right-hand side (RHS), which now is non-zero due to the presence of the current density J_z in the coil, represented by the red and blue rectangles in Figure 5.5.

Following the Galerkin testing procedure, the i-th component of the RHS can be assembled element-by-element:

$$b_i = \int_\Omega N_i J_z \, d\Omega = \sum_{k=1}^{N_e} \left[\int_{\Omega_k} N_i J_z \, d\Omega \right] \tag{5.14}$$

being Ω_k the k-th element (triangle).

Since the mesh is conformal (i.e., there are no triangles across different materials), J_z is uniform inside each triangle (i.e., either zero or a known value, as shown in Table 5.1). Thus, the contribution of the integral over Ω_k in (5.14) can be rewritten as:

$$\int_{\Omega_k} N_i J_z \, d\Omega = J_z^{e_k} \int_{\Omega_k} N_i \, d\Omega = J_z^{e_k} \frac{\Delta_{e_k}}{3} \tag{5.15}$$

where Δ_{e_k} is the area of the k-th element, and $J_z^{e_k}$ is the value of $J_z(x, y)$ in any point internal to e_k, for example in its barycenter. When assembling the RHS element-by-element, this contribution must be assigned to all three nodes that belong to the k-th triangle.

Denoting with *rhsFun* the value of the current density in the barycenter of each triangle discretizing the coils (red and blue), whose indices in the connectivity matrix *triang* are collected in the array *rhsId* the procedure to compute the RHS vector **b** is listed in Algorithm 25. Note that Algorithm 25 can also be used to solve *div-grad* electrostatic problems with non-zero charge density in the computational domain Ω.

Algorithm 25 RHS vector assembly

Input: *rhsFun*, *rhsId* *triang*, *areas*, *nn*
Output: RHS vector **b**

1: $nb \leftarrow$ `length`$(rhsId)$ ▷ number of triangles with imposed current density
2: $\mathbf{b} \leftarrow$ `zeros`$(nn, 1)$ ▷ Initialize RHS
3: **for** i=1:nb **do**
4: $Ji \leftarrow rhsFun(i)$ ▷ Get value of current in ith triangle barycenter
5: $idT \leftarrow rhsId(i)$ ▷ Index of triangle in connectivity matrix
6: $nodes_{loc} \leftarrow$ Indices of nodes of triangle idT
7: $\mathbf{b}(nodes_{loc}) \leftarrow \mathbf{b}(nodes_{loc}) + Ji * areas(idT)/3$ ▷ Integral with midpoint rule
8: **end for**

The reference solution (A_z) of the magnetostatic problem computed with the

commercial FEM software COMSOL Multiphysics is given in Fig.5.6. The magnetic flux density $\mathbf{B} = \nabla \times \mathbf{A}$ is shown in Fig.5.7.

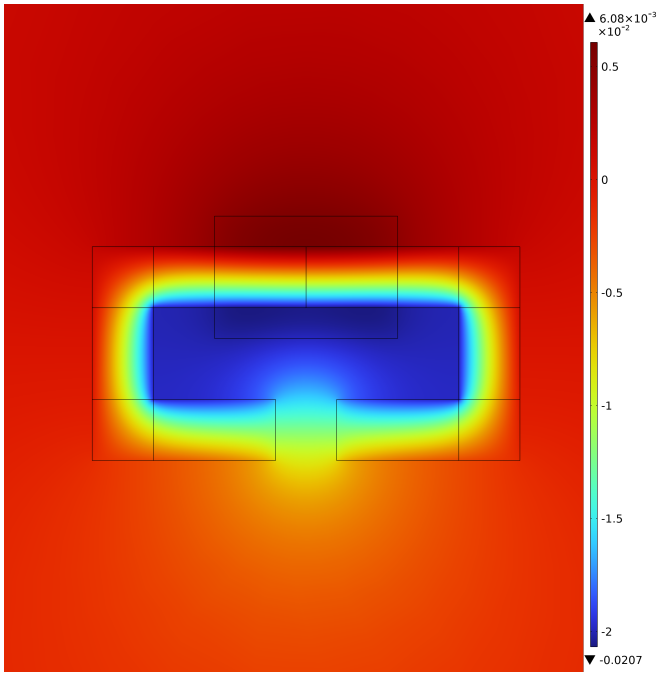


Figure 5.6: Detail of the reference solution (COMSOL): colormap of $A_z\,[A/m]$.

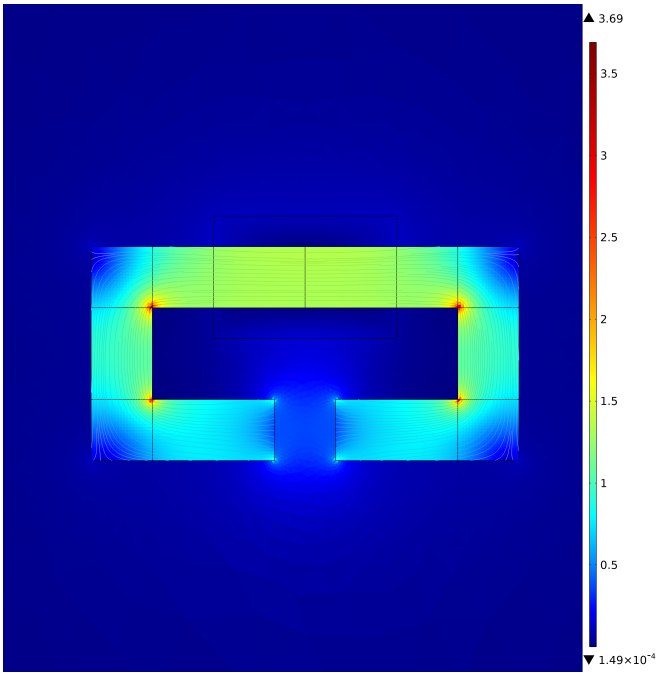


Figure 5.7: Detail of the reference solution (COMSOL): colormap of the norm of the magnetic flux density $B\,[T]$ and field lines.

Chapter 6

Finite Element with MATLAB PDE-Toolbox

6.1 General Div-Grad Equation in 2D - DC and Transient

6.1.1 Introduction

This document provides a structured explanation of a MATLAB script that solves a time-dependent partial differential equation (PDE) over a circular domain using the Finite Element Method (FEM). The PDE in question is a basic parabolic PDE with a source term, and the code also compares the numerical result with the analytical solution.

6.1.2 Initial Setup and Geometry Definition

The script starts by preparing the PDE model environment and defining the domain. In this problem, we only have a single domain (a circle with radius 1 m).

Listing 6.1: Creating the PDE model and defining geometry

```
model = createpde(); % reparing the PDE model environment
typ =1; % type of geometrical element, 1->circle
x=0;    % x-coordinate of the circle center
y=0;    % y-coordinate of the circle center
R=1;    % Radius of the circle
gd = [typ; x; y; R]; % put everthing together
[g, bt] = decsg(gd);  % function that generate the edges of
   the geometry
```

```
8 geometryFromEdges(model, g); % function that generate the
    geometry and add it to "model"
```

Here, the geometry is defined as a circle with center at (0,0) and radius 1. The function `decsg` constructs the edge structure, and `geometryFromEdges` integrates it into the PDE model.

6.1.3 Visualizing the Geometry

With this command, we can visualize the geometry, the ids of the edges and of the domain (faces since we are in 2D).

Listing 6.2: Plotting the geometry

```
figure
2 pdegplot(model,"EdgeLabels","on","FaceLabels","on");
title('geometry')
4 axis equal
```

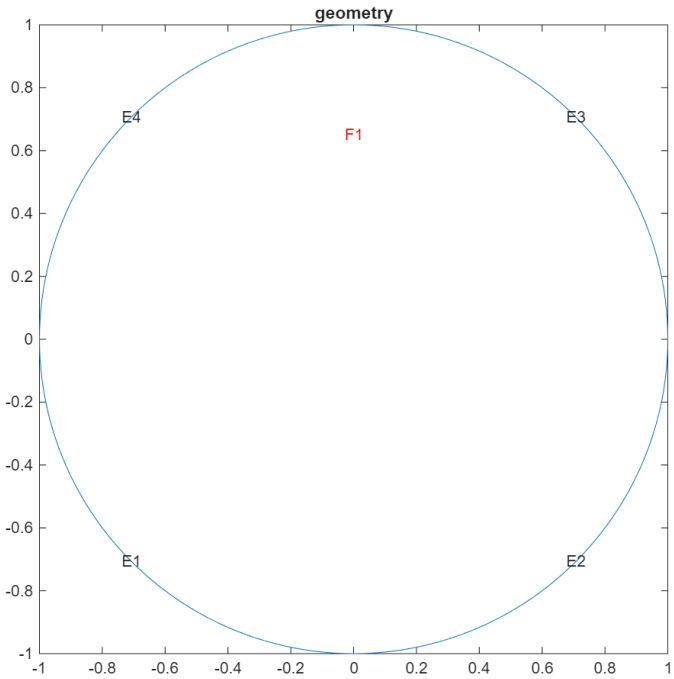


Figure 6.1: Geometry with ids of edges and faces.

This snippet labels edges and plots the defined domain.

6.1.4 Boundary Conditions

All edges are assigned a Dirichlet boundary condition with value zero.

Listing 6.3: Applying boundary conditions

```
edges_id=[1:4];
applyBoundaryCondition(model,"dirichlet", ...
"Edge",edges_id, ...
"u",0);
```

With the command, we are applying the Dirchlet condition to all the edges, i.e.,

$$u(x, y) = 0, \text{with } (x, y) \in |\Gamma. \tag{6.1}$$

With the same function, we can also apply the Neumann and generalized Neumann conditions (open the function to see how). We can also apply different boundary conditions to different edges.

6.1.5 PDE Coefficients

We then define the coefficients of the PDE (material properties). To see how, open the function. The general PDE is in the form

$$m\frac{\partial^2 u}{\partial t^2} + d\frac{\partial u}{\partial t} - \nabla \cdot (c\nabla u) + au = f. \tag{6.2}$$

We want to solve the following PDE

$$\frac{\partial u}{\partial t} - \nabla \cdot (\nabla u) = 1, \tag{6.3}$$

So, in this case, we have $m = 0$, $d = 1$, $c = 1$, $a = 0$, and $f = 1$.

We can assign these coefficients with the following command:

Listing 6.4: Specifying the PDE coefficients

```
specifyCoefficients(model,"m",0,"d",1,"c",1,"a",0,"f",1);
```

To see how these coefficients are used in the PDE, just open the function `specifyCoefficients`:

```
function coef = specifyCoefficients(self, varargin)
% specifyCoefficients - Specify all PDE coefficients over a domain or sub-domain
%     The PDE toolbox can solve equations of the form:
%
%              m*d^2u/dt^2 + d*du/dt - div(c*grad(u)) + a*u = f
```

Figure 6.2: How to assign coefficients to the PDE with `specifyCoefficients` function

In this problem, we have only one domain. In a general problem, we may have different domains with different material properties. Thus, we should apply different PDE coefficients to different domains. As we will see in other examples, we can do that by using again `specifyCoefficients(···,'face',domain_id)` and specifying the domain id.

6.1.6 Mesh Generation

Listing 6.5: Mesh creation

```
hmax = 0.1;  % mesh size
generateMesh(model,"Hmax",hmax,'GeometricOrder','linear');
mesh = model.Mesh;
```

A mesh is generated with a maximum element size of 0.1.

6.1.7 Plotting the Mesh

Listing 6.6: Plotting the mesh

```
figure
pdemesh(model);
title('mesh')
axis equal
```

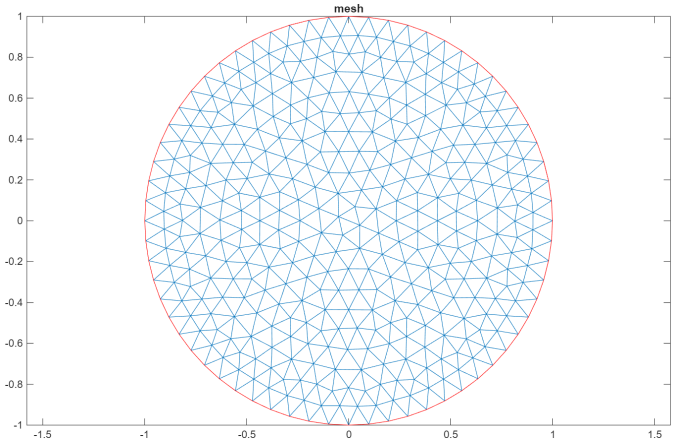


Figure 6.3: Meshed geometry.

6.1.8 Assembling the FEM Matrices

Listing 6.7: FEM assembly

```
fem = assembleFEMatrices(model,'nullspace');
```

This assembles matrices for FEM solving, eliminating boundary conditions by applying the nullspace method.

In particular, the following matrices/vectors are generated:

- `fem.Kc` is the sparse stiffness matrix already corrected to consider the Dirichlet b.c.
- `fem.M` is the sparse mass matrix already corrected to consider the Dirichlet b.c.
- `fem.F_c` is the rhs vector already corrected to consider the Dirichlet b.c.
- `fem.ud` is the Dirichlet solution vector (i.e., a vector with the values of the Dirichlet nodes in the corresponding entries and zero elsewhere)
- `B` is a sparse rectangular matrix to move the solution from the Dirichlet one (i.e., the one without the Dirichlet node) to the full one (i.e., the one including the Dirichlet nodes)

Thus, the discrete version of the PDE is written as

$$\texttt{fem.M} * \dot{\texttt{ud}} + \texttt{fem.Kc} * \texttt{ud} = \texttt{fem.Fc} \tag{6.4}$$

6.1.9 Solving the Stationary PDE

In steady state condition (i.e., $\frac{\partial}{\partial t} = 0$), the PDE reduces to

$$-\nabla \cdot (\nabla u) = 1, \tag{6.5}$$

And the corresponding discrete static problem ($\dot{\mathbf{x}} = 0$) is given by

$$\texttt{fem.Kc} * \texttt{ud} = \texttt{fem.Fc} \tag{6.6}$$

Listing 6.8: Solving the PDE

```
1 uc=fem.Kc\fem.Fc; % solving the problem
  u=fem.B*uc+fem.ud; % re-adding dirichlet nodes
```

6.1.10 Plotting the Solution

The solution u can be visualized with the following commands.

Listing 6.9: Plotting the solution

```
figure
patch('Faces', mesh.Elements.', 'Vertices', mesh.Nodes.', ...
'FaceVertexCData', u, ...
'FaceColor', 'interp', ...
'EdgeColor', 'none');
axis equal
title("Numerical Solution");
xlabel("x")
ylabel("y")
colormap jet
colorbar
```

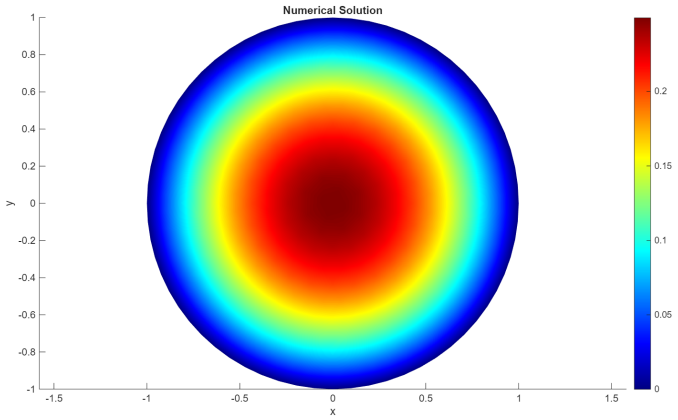


Figure 6.4: Visualizing the solution.

6.1.11 Comparing with the Exact Solution

For this simple PDE, it is possible to find the analytical solution and compare it with the numerical one.

```
p = model.Mesh.Nodes;
exact = (1 - p(1,:).^2 - p(2,:).^2)/4;
patch('Faces', mesh.Elements.', 'Vertices', mesh.Nodes.', ...
          'FaceVertexCData', u-exact.', ...
          'FaceColor', 'interp', ...
          'EdgeColor', 'none');  % optional: 'k' for visible
             edges
```

```
title("Error");
xlabel("x")
ylabel("y")
colormap jet
colorbar
axis equal
```

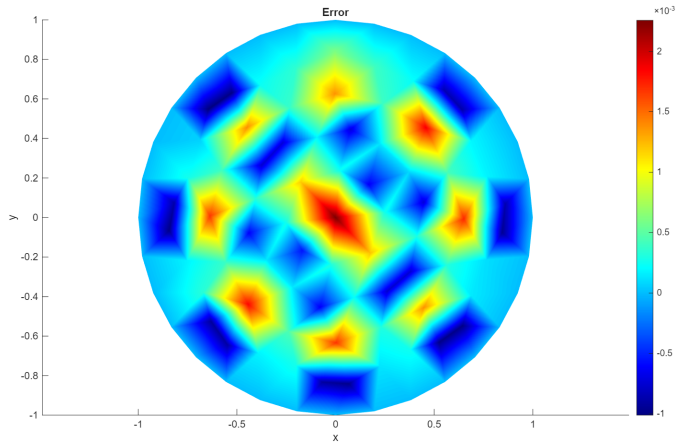


Figure 6.5: Error: Numerical solution vs Analytical solution.

We can also have an error evaluation of the numerical solution:

```
err=100*norm(u-exact.')/norm(exact)
```

Note that we can do that only for this simple academic problem, for which we know the analytical (exact) solution. For other realistic problems, we cannot have the exact solution. However, it is interesting to change the value of the mesh size and see how the error changes (do it!). So, in general, it is important to have a mesh size *sufficiently* small to have good accuracy.

6.1.12 Solving the Time-dependent Problem

With the same matrices we have already computed, we can now solve the Time-dependent problem, i.e.,

$$\frac{\partial u}{\partial t} - \nabla \cdot (\nabla u) = 1 * sin(t), \tag{6.7}$$

that is written in discrete form as

$$\texttt{fem.M} * \dot{\texttt{ud}} + \texttt{fem.Kc} * \texttt{ud} = \texttt{fem.Fc} * \texttt{sin(t)} \tag{6.8}$$

We are now imposing a sinusoidal behavior on the right-hand side.

Listing 6.10: Time domain solution

```
b=@(t) sin(t); % time domain modulation of the forcing term
Time_window=[0:0.1:10];
x0=uc*0;  % initial solution (initial condition)
[tt,xx]=ode15s(@(t,x) fem.M(-fem.Kc*x+fem.Fc*b(t)),
   Time_window,x0);
ut=fem.B*xx.'+repmat(fem.ud,1,length(tt)); % re-adding
   Dirichlet
```

6.1.13 Plotting the Time Evolution of One Node

We can then plot the solution of a specific node in the time domain (node 65 in this case):

Listing 6.11: Temporal evolution at a node

```
figure
plot(tt,(ut(65,:)))
xlabel('time [s]')
ylabel('x(1)')
drawnow
```

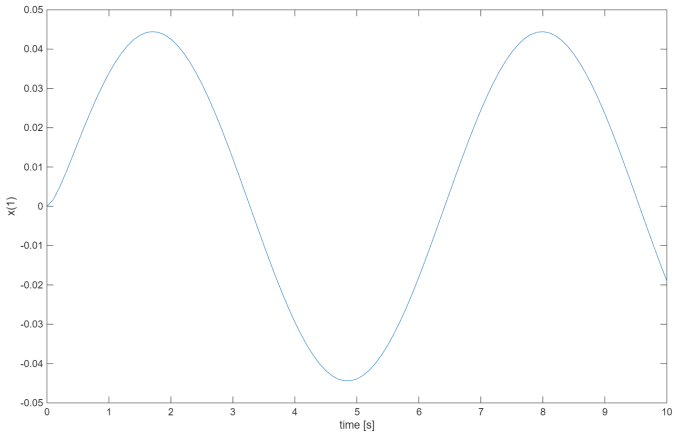


Figure 6.6: Time domain solution of node 65.

6.1.14 Animated Plot of the Full Solution in Time

Finally, we can plot the solution at each time frame using interpolated colors to show how the solution evolves.

Listing 6.12: Video plot of time evolution

```
maxut=max(ut(:));
minut=min(ut(:));
figure
for ii = 1:length(tt)
    p = model.Mesh.Nodes;
    patch('Faces', mesh.Elements.', 'Vertices', mesh.Nodes.',
        ...
        'FaceVertexCData', ut(:,ii), ...
        'FaceColor', 'interp', ...
        'EdgeColor', 'none');  % optional: 'k' for visible
           edges
    axis equal
    title(tt(ii))
    xlabel("x")
    ylabel("y")
    colormap jet
    colorbar
    clim([minut maxut])
    drawnow
    pause(0.1)
end
```

6.2 Electrostatic DC in 2D - Plane Capacitor

In this example, we solve the electrostatic equation to evaluate the capacitance of a plane capacitor. We will learn how to handle multiple domains with the PDE toolbox and how to extract the capacitance from the electrostatic energy. The capacitor we are considering is a plane capacitor. Its depth is much longer than the other dimensions (w=20 cm, h=1 cm, d=2 cm, l=1 m), so we can construct a 2D model (although this is an approximation).

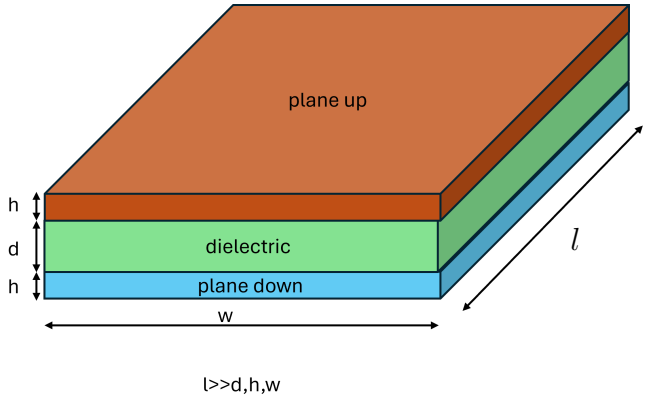


Figure 6.7: Graphical representation of the plane capacitor.

6.2.1 Geometry construction with multiple domains

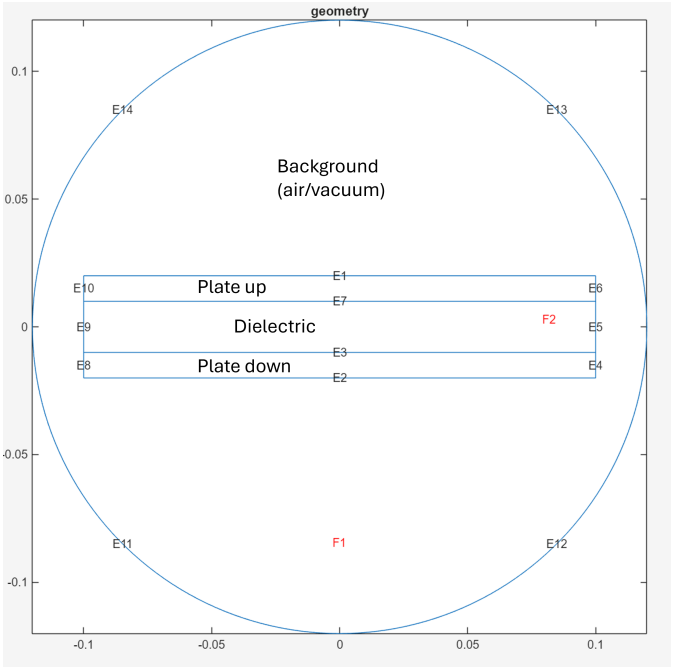


Figure 6.8: Geometry of the plane capacitor

We are considering a capacitor that is long in the longitudinal direction so that we can solve the problem in 2D. To construct the geometry, we can use the command:

```
1 [g, bt] = decsg(gd,sf,ns); % generate the edges of the
    geometry
```

```
geometryFromEdges(model, g); % generate the geometry
```

where gd is a matrix of 4 columns. Each column corresponds to one element of the geometry (background - circle, plate up - rectangle, plate down - rectangle, dielectric - rectangle):

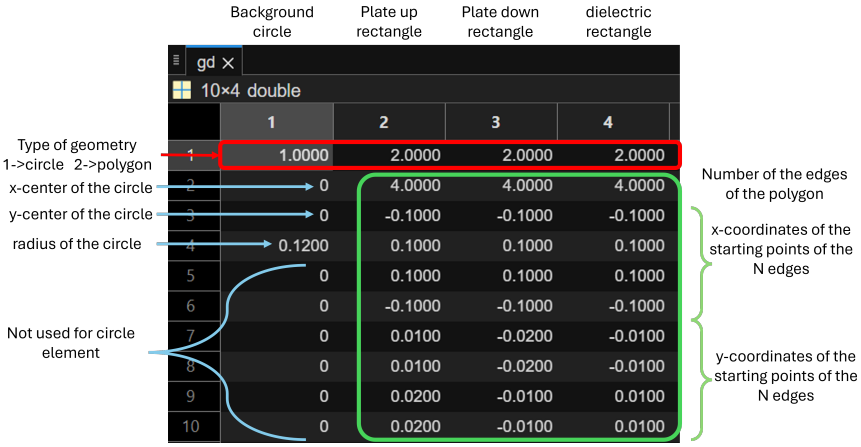


Figure 6.9: Description of the geometry with gd.

A way to construct gd is given by the following code:

```
%%
model = createpde(); % matlab command that prepares the
    workspace
%% CREATE the  geometry
% circle
typ_c =1; % type of geometrical element, 1->circle
x_c=0;    % x-coordinate of the circle center
y_c=0;    % y-coordinate of the circle center
R_c=0.12*1;    % Radius of the circle [m]
gd_c = [typ_c;
       x_c;
       y_c;
       R_c];   % put everything together for the circle
% plate up rect1
w=20e-2;% width of the rectangle [m]
h=1e-2; % height of the rectangle [m]
d=2e-2; % distance between the two plates and thickness of
    dielectric
typ_1=2; % type of geometrical element, 2->polygon
N_1=4;   % number of edges
```

```
x_1 = [-w/2;w/2;w/2;-w/2]; % starting x-coordinates of the N-
    edges
y_1 = d/2+[0;0;h;h]; % starting y-coordinates of the N-edges
gd_1 = [typ_1;
       N_1;
       x_1;
       y_1];   % put everything together for plate up
% plate down rect2
typ_2=2; % type of geometrical element, 2->polygon
N_2=4;   % number of edges
x_2 = [-w/2;w/2;w/2;-w/2]; % starting x-coordinates of the N-
    edges
y_2 = -d/2-h+[0;0;h;h]; % starting y-coordinates of the N-
    edges
gd_2 = [typ_2;
       N_2;
       x_2;
       y_2];   % put everything together
% dielectric rect3
typ_3=2; % type of geometrical element, 2->polygon
N_3=4;   % number of edges
x_3 = [-w/2;w/2;w/2;-w/2]; % starting x-coordinates of the N-
    edges
y_3 = [-d/2;-d/2;d/2;d/2]; % starting y-coordinates of the N-
    edges
gd_3 = [typ_3;
       N_3;
       x_3;
       y_3];   % put everything together  for dielectric
% put everything together
gd = zeros(length(gd_1),3); % initialize
gd(1:length(gd_c),1)=gd_c; % circle
gd(1:length(gd_1),2)=gd_1; % rectangle 1 plate top
gd(1:length(gd_2),3)=gd_2; % rectangle 2 plate bottom
gd(1:length(gd_3),4)=gd_3; % rectangle 3 plate dielectric
```

Variable `sf` assigns names to the 4 domains:

```
ns=char('background', 'plate_up','plate_down','dielectric')'
```

Finally, variable `sf` describes how the geometry is constructed from these 4 domains (we can perform boolean operations such as unions, differences, etc.) In this case, we are going to impose a uniform potential on the capacitors' plates. So, the solution (in terms of potential) is already known inside the plates and on

the plates' boundary. Thus, there is no reason to include them in the computational domain: we can remove them and then assign the potential with Dirichlet boundary conditions. We can remove them from the computational domain with the following command:

```
sf = '((background-plate_up)-plate_down)+dielectric';
```

So, for the sake of clarity, we can then construct the geometry with:

```
[g, bt] = decsg(gd,sf,ns); % generate the edges of the
    geometry
geometryFromEdges(model, g); % generate  the geometry
```

If we then plot the geometry as seen in Example 6.1, we obtain Fig. 6.9. As we can see, there are no face ids related to plate up and plate down since they have been removed from the computational domain.

6.2.2 Assign PDE coefficients to different domains

The PDE equation we are solving is

$$\nabla \cdot (\varepsilon(x,y)\nabla V(x,y)) = 0, \tag{6.9}$$

$$V(x,y) = +1 \qquad \text{with } (x,y) \in \Gamma_{\text{plate up}} \tag{6.10}$$

$$V(x,y) = -1 \qquad \text{with } (x,y) \in \Gamma_{\text{plate down}} \tag{6.11}$$

where ε is equal to the vacuum permittivity ε_0 in the background and is equal to $\varepsilon_0\varepsilon_r$ in the dielectric domain.

To assign that we can use the following command

```
eps0=8.8541878188e-12;
epsr=4;
specifyCoefficients(model,"m",0,"d",0,"c",eps0,"a",0,"f",0,'
    Face',1);
specifyCoefficients(model,"m",0,"d",0,"c",eps0*epsr,"a",0,"f
    ",0,'Face',2);
```

6.2.3 Assign different boundary conditions

We can now assign b.c.. We have to assign different Dirichlet b.c. to the boundaries of the two plates:

```
applyBoundaryCondition(model,"dirichlet", ...
                                "Edge",[1,6,7,10], ...
                                "u",1);
```

```
applyBoundaryCondition(model,"dirichlet", ...
                                "Edge",[2,3,4,8], ...
                                "u",-1);
```

For the external boundary of the circle, we can assign a homogeneous Neumann boundary condition. However, we do not need to do it since it is imposed naturally by the FEM method.

6.2.4 Extracting the Capacitance

Then we can construct the mesh and solve the problem as described in Example 6.1.

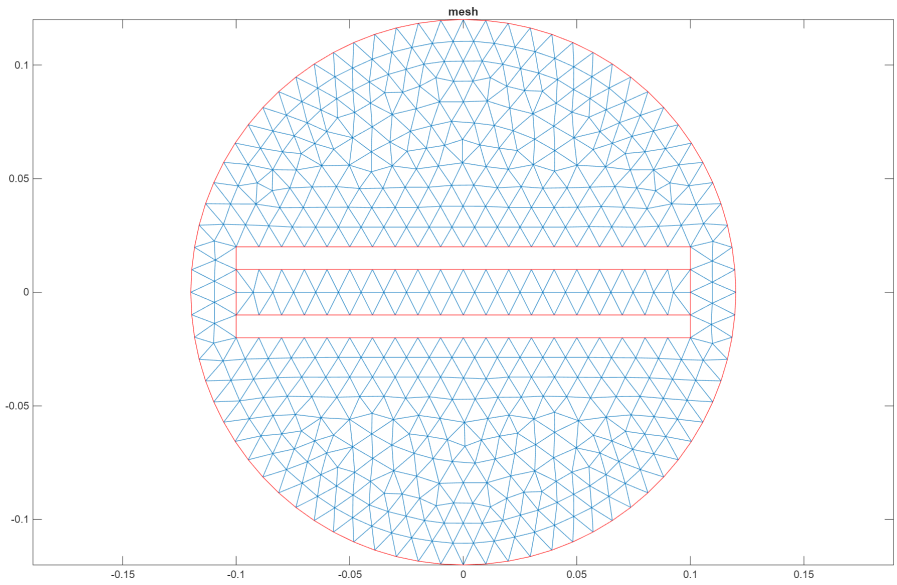


Figure 6.10: Mesh.

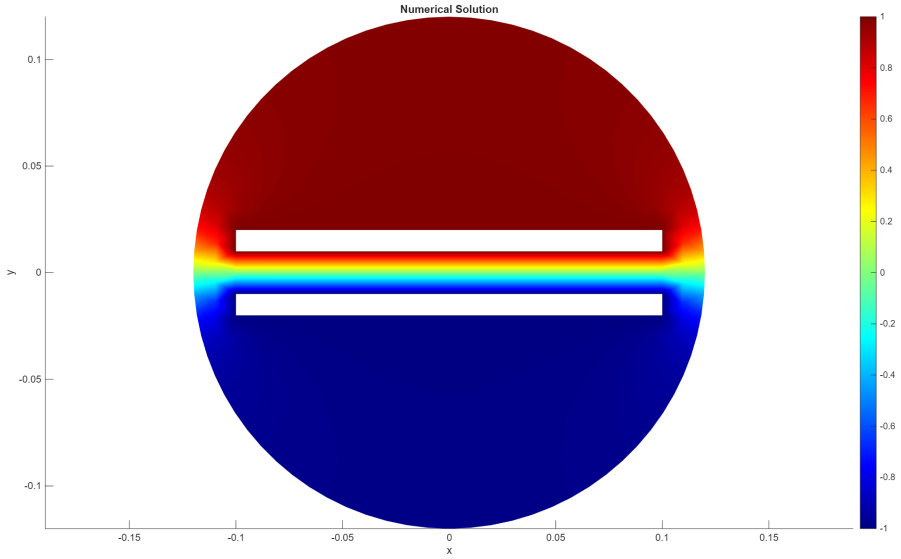


Figure 6.11: Solution (Electric Potential [V]).

Once the solution has been found, we can extract the capacitance of the capacitor from the energy evaluation. The energy can be evaluated as

```
We=u'*fem_nobc.K*u/2
```

where `fem_nobc` is obtained as

```
1 fem_nobc = assembleFEMatrices(model); % do not apply b.c.
```

From the electrostatic energy `We`, we can then evaluate the capacitance from the well known formula:

$$W_e = \frac{1}{2}C(\Delta V)^2 \tag{6.12}$$

and we can compare it with the approximate analytical formula of a plane capacitor:

$$C = \varepsilon_0 \varepsilon_r \frac{S}{d} \tag{6.13}$$

where S is the area of the plate, here equal to $0.2 * 1$.

It is suggested to compare the results also by changing the

- mesh size
- external radius of the circle
- distance between the capacitors' plates
- dielectric constant (from 1 to 10)

What happened to the value of the capacitance? Also, compare it with the analytical formula.

6.2.5 Post processing for the evaluation of the Electric field

Once the solution in terms of the nodal electric potentials is obtained, we can obtain, as post-processing, the electric field. To do that, we can use the following commands

```
[grads,centroids] = computeTriangleGradients(model.Mesh.
    Nodes.', model.Mesh.Elements.', u); % find gradent of
    the solution
E=-[grads(:,1),grads(:,2)]; % find E=-gradV field
Enorm=sqrt(dot(E,E,2)); % find norm of E
```

Function `computeTriangleGradients` is available in Moodle.

We can then plot the results:

```
figure
hold on
patch('Faces', model.Mesh.Elements.', 'Vertices', model.
    Mesh.Nodes.', ...
     'CData', (Enorm),...
     'FaceColor','flat',...
     'EdgeColor', 'none');  % optional: 'k' for visible
        edges
quiver(centroids(:,1), centroids(:,2), real(E(:,1)), real
    (E(:,2)), 1)%ngrad);
pdegplot(model, 'EdgeLabels', 'off', 'FaceLabels', 'off')
    ;
axis equal
title("E [V/m]");
xlabel("x")
ylabel("y")
colormap jet
colorbar
```

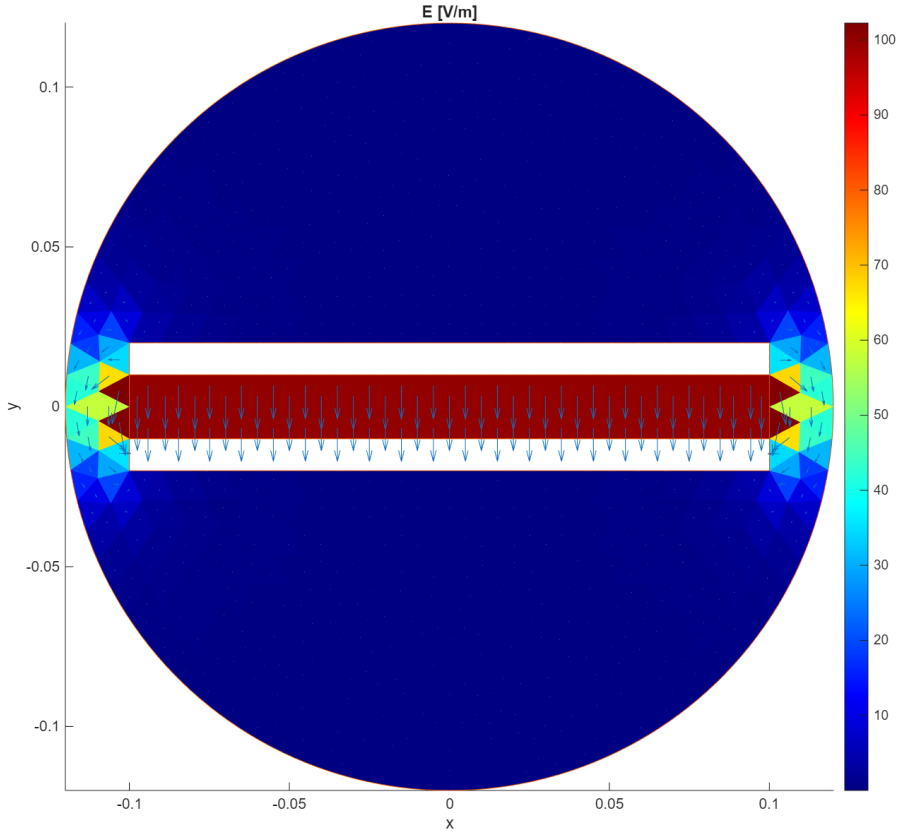


Figure 6.12: Solution (Electric Field [V/m]).

6.2.6 Quadratic elements

In the PDE toolbox, it is also possible to use quadratic order elements for the shape functions. We can do that when the mesh is generated:

```
hmax = 0.01; % mesh size
order='quadratic'; % 'linear' 'quadratic'
generateMesh(model,"Hmax",hmax,'GeometricOrder',order);
mesh = model.Mesh;
```

Then, the problem can be solved as before. However, to visualize the solution is better to solve the problem with an internal function of the PDE toolbox:

```
results = solvepde(model);
u = results.NodalSolution;
```

Finally, the solution can be visualized as

```
figure
```

```
    pdeplot(model,"XYData",u,'Contour','on')
    axis equal
    title("Numerical Solution");
    xlabel("x")
    ylabel("y")
    colormap jet
    colorbar;
```

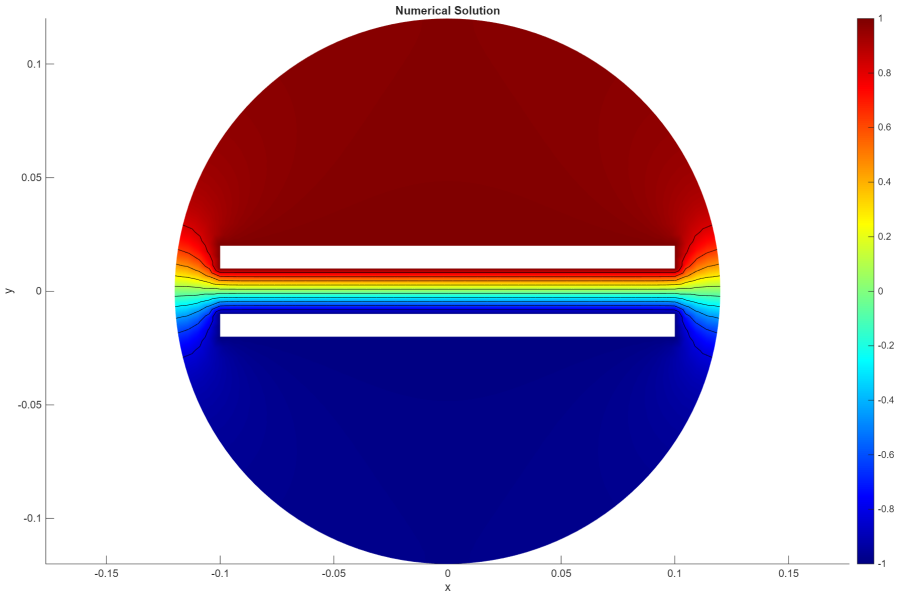


Figure 6.13: Solution (Electric Potential [V]) with second order elements.

As can be seen, by using second-order elements, the solution is much smoother. Also analyze what happens to the capacitance value.

6.3 Electrostatic DC in 2D - Plane Capacitor - Applying Symmetries

In the previous Example 6.2, we didn't exploit the symmetries of the problem. The next figure shows them.

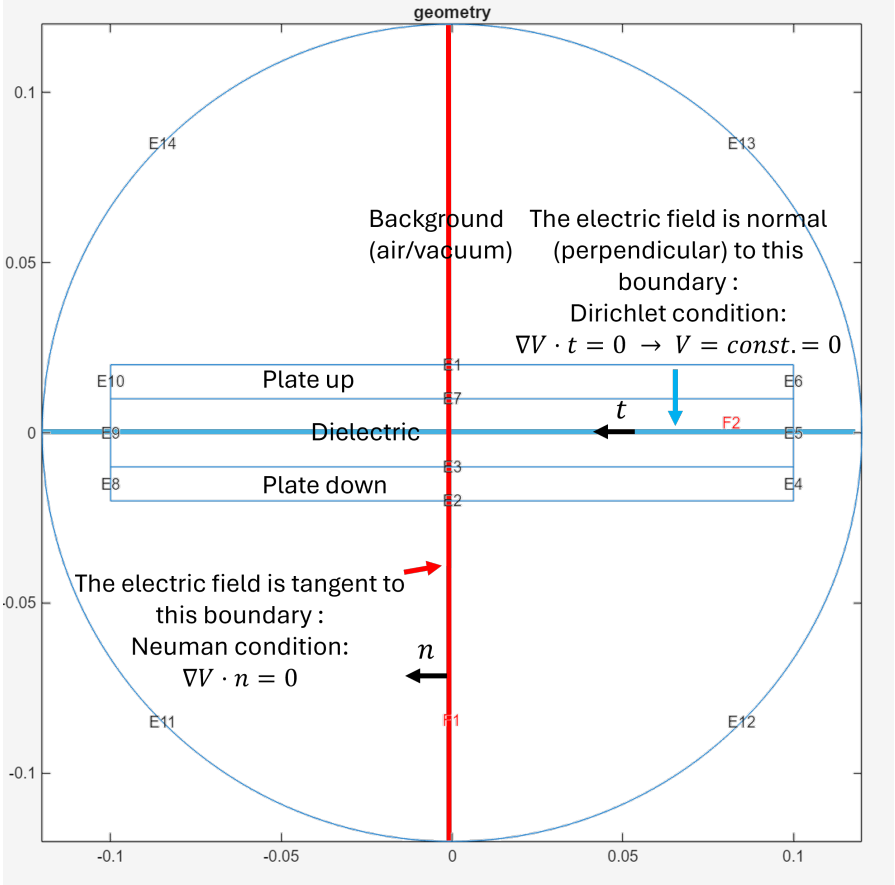


Figure 6.14: Symmetries of the Plane Capacitor.

We can then reduce the computational domain to just 1/4 of the original one by exploiting symmetries and applying the corresponding b.c..

To use just 1/4 of the geometry, we can construct a square domain and then make the intersection between the square domain and the other geometry. So, variable `gd` will have an extra column related to this square (polygon), we assign a name to it and we perform the intersection with `*` operator:

```
ns = char('background','plate_up','plate_down','dielectric','
    rect')';
sf = '(((background-plate_up)-plate_down)+dielectric)*rect';
```

We can then assign boundary conditions and PDE coefficients and solve the problem as previously discussed.

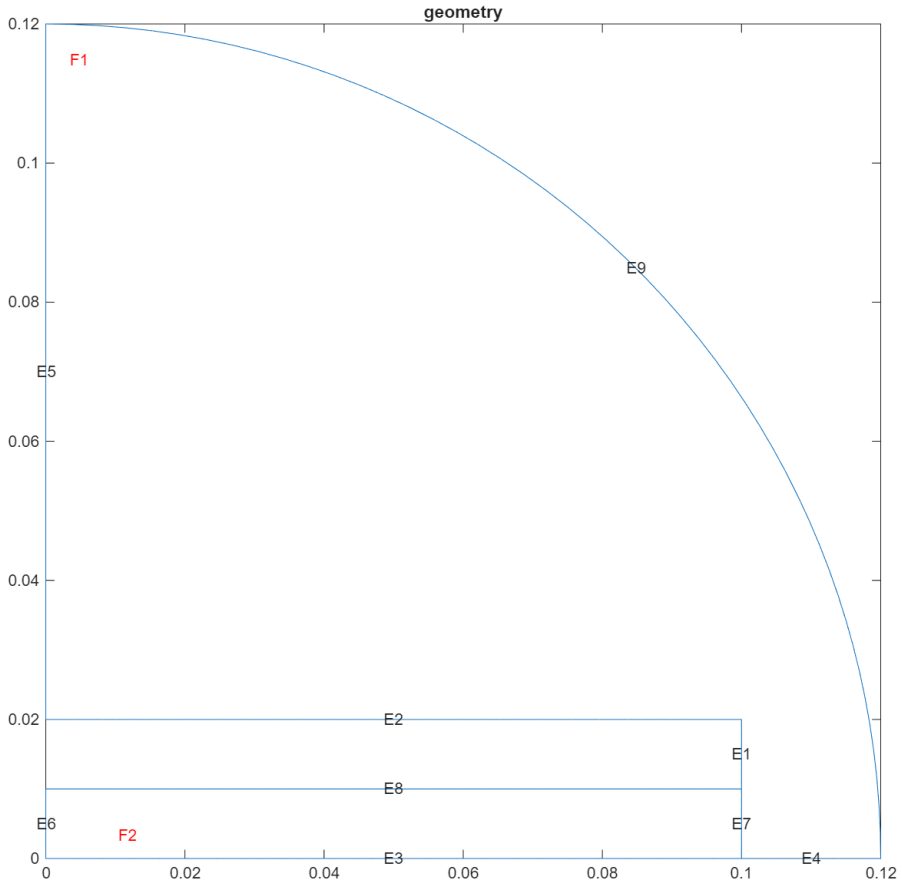


Figure 6.15: 1/4 of the geometry of the plane capacitor

Obviously, once the problem has been solved, we can also extract the capacitance value. Since we are just considering 1/4 of the domain, we need to multiply `We` by a factor of 4 to get the correct value. Check it!

6.4 Electrostatic DC in 2D-axisymmetric - Plane Capacitor

Here we consider instead the case of a capacitor with circular plates, so it is possible to solve it with a 2D-axisymmetric model (w=20 cm, h=1 cm, d=2 cm).

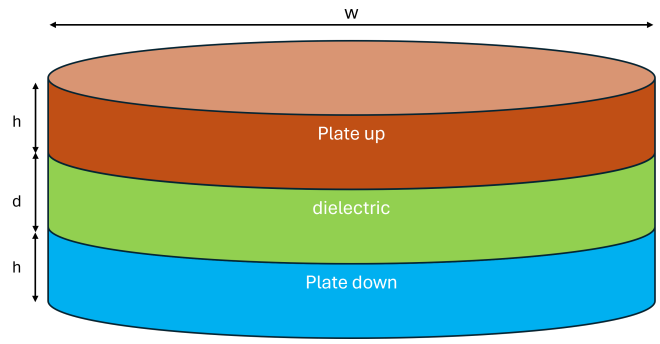


Figure 6.16: Graphical representation of the circular-plane capacitor.

The electrostatic equation in axysimmetric coordinates is:

$$\nabla \cdot (r\varepsilon \nabla V) = 0 \tag{6.14}$$

that can be written as

$$\nabla \cdot (\varepsilon' \nabla V) = 0 \tag{6.15}$$

where $\varepsilon' = r\varepsilon$.

Thus, to study it, we only have to define an equivalent permittivity that varies with r. To do that, we can use the following command:

```
eps0=8.8541878188e-12;
epsr=4;
specifyCoefficients(model,"m",0,"d",0,"c",@(location,state)
    eps0.*(location.x),"a",0,"f",0,'Face',1);
specifyCoefficients(model,"m",0,"d",0,"c",@(location,state) (
    epsr*eps0).*(location.x),"a",0,"f",0,'Face',2);
```

So, the PDE toolbox sees a permittivity as a function handle that varies with coordinate x (i.e., r).

Everything else is then the same. However, unfortunately, we cannot evaluate the capacitance as we did before. Indeed, we cannot evaluate the energy with `We=u'*fem_nobc.K*u/2` since the stiffness matrix is constructed with an equivalent permittivity.

6.5 Magnetoquasistatic in 2D-axisymmetric - Induction Heating Coil - AC

We solve a magneto-quasistatic problem in which a conductive and magnetic sphere is placed inside a coil composed of Litz wire, driven by a prescribed current density. This is a typical scenario in induction heating applications.

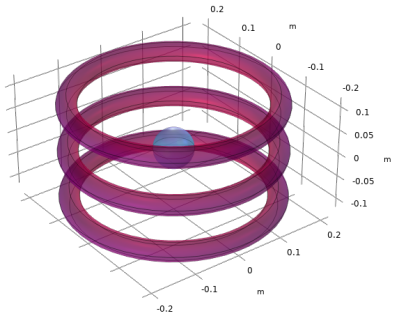


Figure 6.17: Graphical representation of the axially symmetric coil and the conductive and magnetic sphere.

The dimensions are given in the image below.

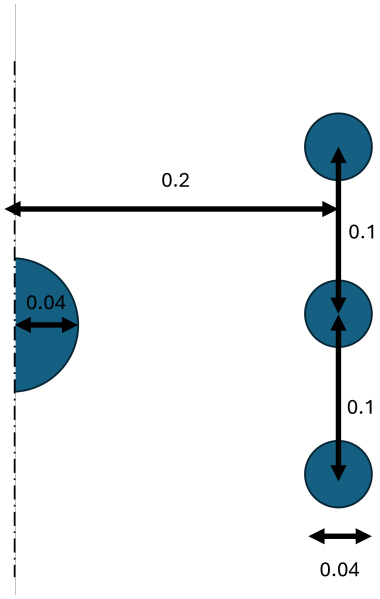


Figure 6.18: Geometry and dimensions (in m) of the problem

We are solving the magneto-quasistatic equation in axysymmetric coordinates and in the frequency domain:

$$-\nabla_a \cdot (\nu'(x,y)\nabla\Phi) + i\omega\sigma'(x,y)\Phi = J_\phi \tag{6.16}$$

where $\nu' = \frac{\nu}{r}$, $\sigma' = \frac{\sigma}{r}$, and $\Phi = rA_\phi$.

The sphere has a conductivity of 2 MS/m and a relative permeability of 10. The current in each turn is 500 A. The background has the properties of vacuum. Note that since the coils are considered as current density driven coil, they must be treated as a vacuum domain with a known current density. The working frequency is 500 Hz. Apply Dirichlet b.c. to the boundary of the external circle (suggested dimension of the external circle is 1 m radius).

6.5.1 Tips to solve the problem

Below are some tips and instructions to solve the problem

- As for the case of the Example 6.4 (the axysymmetric capacitor), here we need to assign equivalent material properties that vary with the position r. However, here r is in the denominator and this is a problem for points on the axisymmetric axis (where $r = 0$). Replace them with $\nu' = \frac{\nu}{r+\epsilon}$, $\sigma' = \frac{\sigma}{r+\epsilon}$ where ϵ is a very small number (e.g., $\epsilon = 1e-8$)

- For the magneto-quasistatic problem, a Dirichlet boundary condition must be applied on the axisymmetric axis.

- After the problem has been solved, we obtain the solution in terms of $\Phi = rA_\phi$. To obtain A_ϕ we can compute

```
Aphi=Phi./(model.Mesh.Nodes(1,:).'+seps);}
```

  where `seps` is ϵ.

- To obtain the magnetic flux density field

$$\mathbf{B} = \nabla \times \mathbf{A} = [B_r, 0, B_z] = [-\frac{1}{r}\frac{\partial \Phi}{\partial z}, 0, \frac{1}{r}\frac{\partial \Phi}{\partial r}].$$

  Thus, we can find it as

```
[grads,centroids] = computeTriangleGradients(model.Mesh.
    Nodes.',...
     model.Mesh.Elements.', Phi);
grads(:,1)=grads(:,1)./centroids(:,1);
grads(:,2)=grads(:,2)./centroids(:,1);
B=[-grads(:,2),grads(:,1)];
normB=sqrt(dot(B,B,2));
```

- It is also useful to obtain and visualize the current density induced in the sphere. It can be computed from $J_\phi = \sigma(-i\omega A_\phi)$ for each mesh element. To do that, we need to obtain the conductivity for each mesh element. This can be done with the following commands:

```
[grads,centroids] = sigma_mesh=zeros(size(model.Mesh.
    Elements,2),1);
for ii = 1:5
 reg(ii).id = model.Mesh.findElements('region','face',ii)
     ;
end
for ii = [1,3,4,5] % face ids of background and coils
     sigma_mesh(reg(ii).id)=sigma_vacuum;
end
sigma_mesh(reg(2).id)=sigma; % 2-> face id of the sphere
%%
[J, centroids] = scaledTriangleMeans(model.Mesh.Nodes
    .',...
     model.Mesh.Elements.', Aphi, 1j*w*sigma_mesh);
```

- Then, the current density can be visualized with the following command

```
figure
hold on
patch('Faces', model.Mesh.Elements.', 'Vertices', model.
    Mesh.Nodes.', ...
            'CData',abs(J),'Facecolor','flat','FaceAlpha'
                ,1.0);  % optional: 'k' for visible edges
pdegplot(model, 'EdgeLabels', 'off', 'FaceLabels', 'off')
    ;
axis equal
title("Eddy Current");
xlabel("x")
ylabel("y")
colormap jet
colorbar
```

Function `scaledTriangleMeans` is available in Moodle.

As a result, we should obtain the values in the following figures.

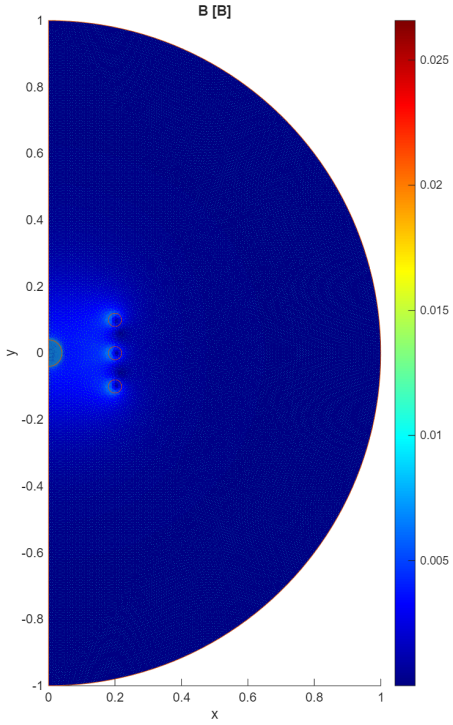


Figure 6.19: Magnetic flux density (in T).

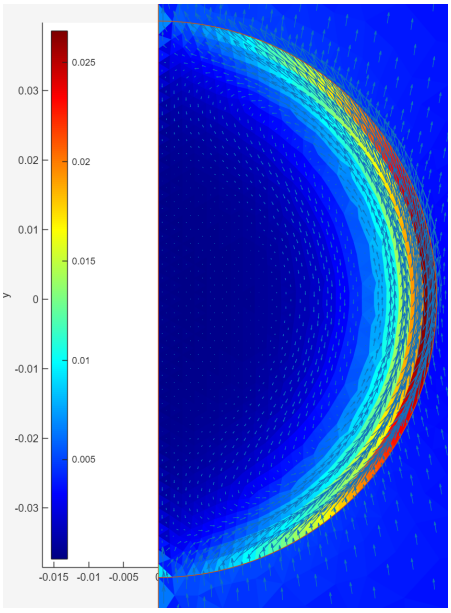


Figure 6.20: Magnetic flux density in the sphere (in T).

And the following induced current density (absolute value)

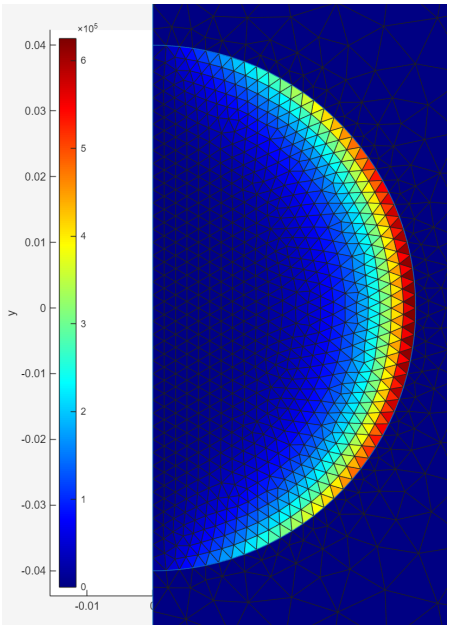


Figure 6.21: Current density (absolute value) in the sphere (in A/m^2).

6.6 Planar Inductor magnetostatic 2D-axisymmetric

We evaluate the inductance of a simple planar inductor above a ferrite core.

The geometry of the inductor is described in the following figure.

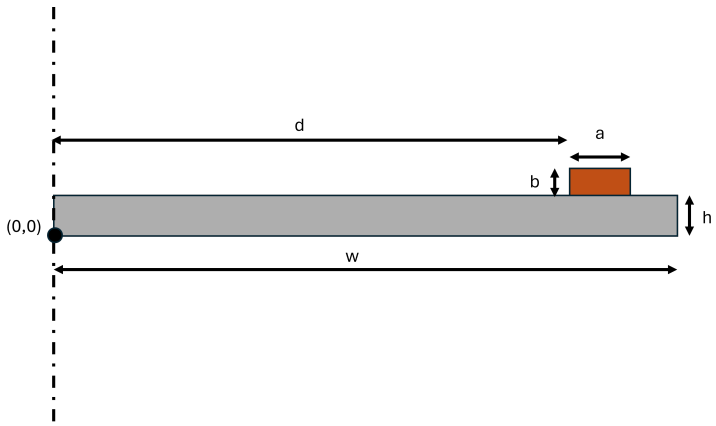


Figure 6.22: Axisymmetric model of the planar inductor. The grey block is the magnetic core. The orange rectangle is the coil cross-section. w=1.5 cm, h = 1 mm, a = 1 mm, b = 0.25 mm, d = 1.3 cm.

The magnetic core has a relative magnetic permeability of 140, and the coil is excited with a unitary current.

After solving the problem, the equivalent inductance of the planar inductor can be computed as

$$L = \frac{\int_l A_\phi dl}{I} = \frac{2\pi r_c A_\phi(r_c)}{I}$$

:

```
t=model.Mesh.Elements;
t(end+1,:)=0;
pp=[d+a/2,h+b/2];
F = pdeInterpolant(model.Mesh.Nodes,t,Aphi);
L = 2*pi*pp(1)*evaluate(F,pp(1),pp(2))
```

In this formula, we are just using the value of the magnetic vector potential at the center of the coil cross-section. However, we should take the average value in the whole cross-section, but for simplicity, we accept this approximation. The value we should obtain is about 114 nH.

6.7 Thermal problem in 2D - Chip with Heat Sink - DC and Transient

We study the thermal problem of a chip with a heat sink.

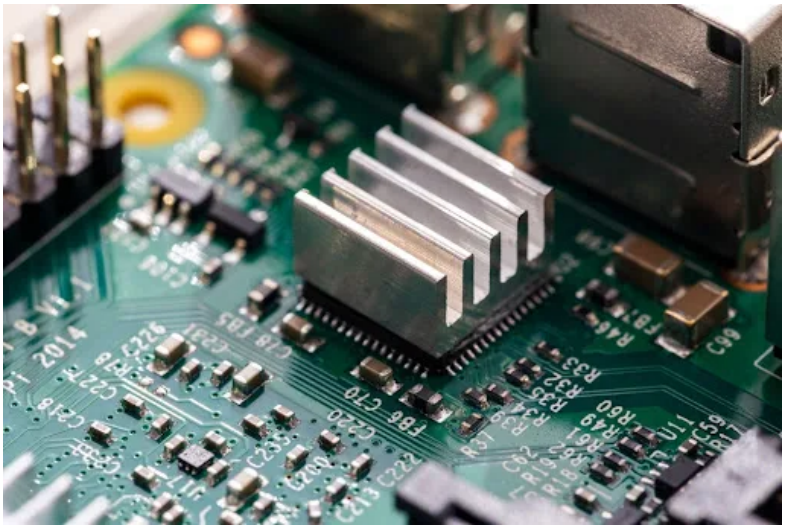


Figure 6.23: Chip and heat sink.

The geometry of the model is shown in the figure below. Also in this case, we use a 2D model, accepting an approximation.

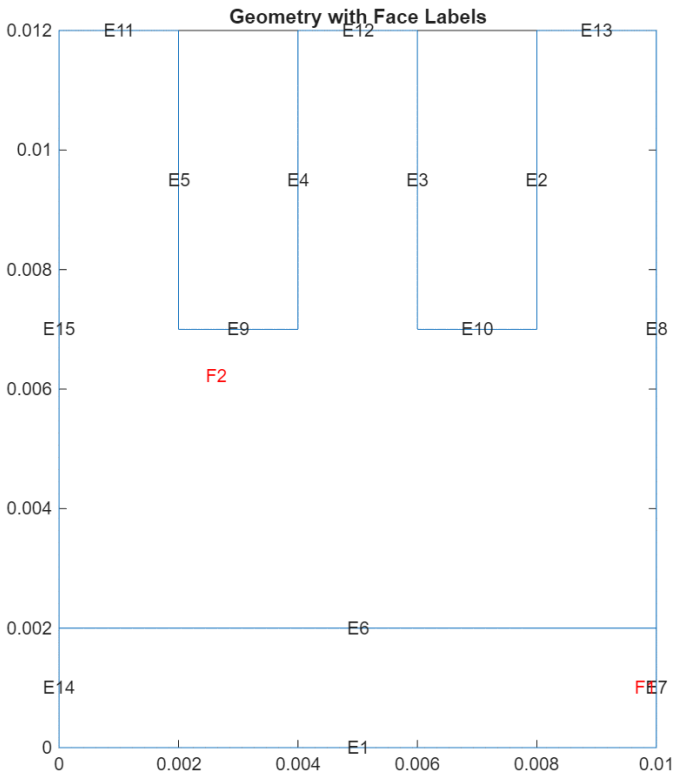


Figure 6.24: 2D Geometry of the chip and heat sink. Dimensions are in m.

The rectangle on the bottom is the silicon chip, while the other domain is the aluminum heat sink. We are supposing that the chip and the heat sink have dimensions equal to 1 cm along the z axis.

The thermal properties of the two media are:

- thermal conductivity of silicon 148 W/(m K)
- heat capacity of silicon 700 J/(kg K)
- ρ density of silicon 2330 kg/m3
- thermal conductivity of heat sink 200 W/(m K)
- heat capacity of heat sink 900 J/(kg K)
- density of heat sink 2400 kg/m3

The heat sink exchanges heat with the external environment by natural convection. The external temperature is 20 °C and the convective heat transfer coefficient is 50 W/(m2 K). The dissipated power in the chip is 2 W. For the sake of clarity, we recall here the heat equation:

$$\rho(x,y)c(x,y)\frac{\partial T}{\partial t} - \nabla\cdot(k(x,y)\nabla T) = q(x,y), \tag{6.17}$$

where ρ is the density, c is the specific heat capacity, k is the thermal conductivity, and q is the power density (W/m3).

6.7.1 Tips to solve the problem

- The convection boundary condition is

$$(-k\nabla T)\cdot\mathbf{n} = h(T - T_{ext}), \tag{6.18}$$

  which is a form of generalized Neumann b.c. This can be imposed in the PDE-toolbox

```
applyBoundaryCondition(model,"neumann", ...
                                "Edge",edges_ids,...
                                'q',coeff_h,...
                                'g',coeff_h*Text);
```

- In the PDE, we need to impose the power density (W/m3). Since we know that the power in the chip is 2 W, we can compute the power density as

$$q = \frac{2\ [\mathrm{W}]}{\text{Volume of the chip } [\mathrm{m3}]}.$$

By saving the problem for the steady state solution the following results should be obtained

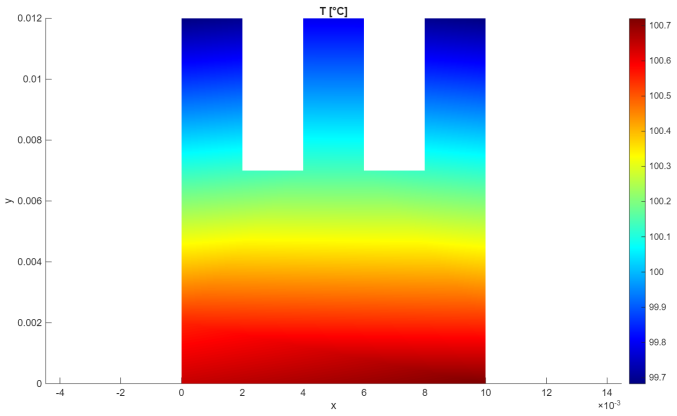


Figure 6.25: Temperature at steady state [°C].

We can also solve the problem in the time domain with a power

$$p(t) = 2 \cdot (1 + 0.2(sin(2 \cdot 2\pi ft)^2),$$

with $f = 1000$ [Hz]. By solving the problem for 200 seconds, we obtain the following results.

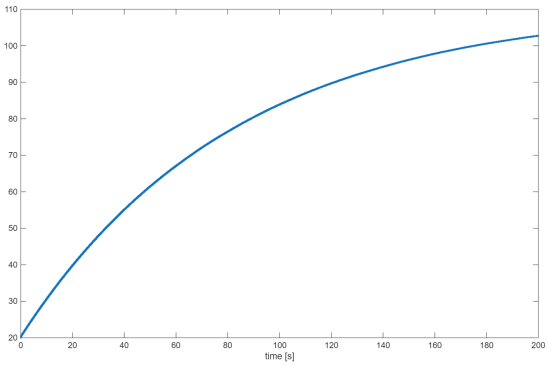


Figure 6.26: Temperature in time domain for a specific node of the mesh [°C].

6.8 Current Flow in 2D - PCB Heater

We study the current flow problem of a PCD Heater.

Figure 6.27: A PCB-Heater

The geometry of the model is shown in the figure below. Also in this case, we use a 2D model, accepting an approximation.

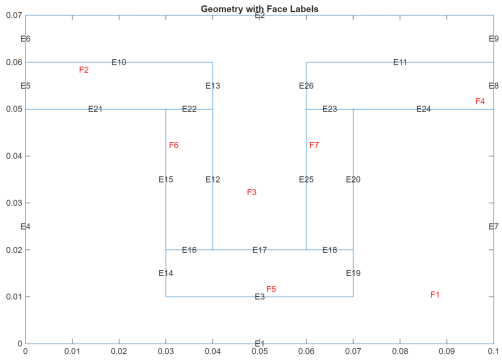


Figure 6.28: 2D Geometry of the PCB heater.

The PCB heater is made of copper (faces 2,4,5,6,7) while the background is an insulating material (faces 1 and 3). The copper has a conductivity of 20 MS/m while the background has a conductivity of 100 S/m. we suppose a 1 mm thickness of the model (for both the heater and the substrate/background).

We excite the problem by applying 1 V (edge 5) at the terminal on the left and 0 V at the terminal on the right (edge 8).

To account for the thickness of the PCB heater, which is 1 mm, we can scale the conductivity by the thickness:

$$\sigma_{2D} = \sigma \cdot th,$$

where th is the thickness. Thus, the PDE to be solved is

$$\nabla \cdot (\sigma_{2D} \nabla V) = 0.$$

After solving the problem, we can find the current density distribution as

```
sigma_mesh=zeros(size(model.Mesh.Elements,2),1);
for ii = id_substrate
    reg(ii).id = model.Mesh.findElements('region','face',ii);
    sigma_mesh(reg(ii).id)=sigma_ground;
end
for ii = id_copper
    reg(ii).id = model.Mesh.findElements('region','face',ii);
    sigma_mesh(reg(ii).id)=sigma_copper;
end
%% Post Processing, from A to B
[grads,centroids] = computeTriangleGradients(model.Mesh.Nodes
   .',...
    model.Mesh.Elements.', V);
grads(:,1)=grads(:,1);
grads(:,2)=grads(:,2);
J=-[sigma_mesh.*grads(:,1),sigma_mesh.*grads(:,2)];
normJ=sqrt(dot(J,J,2));
```

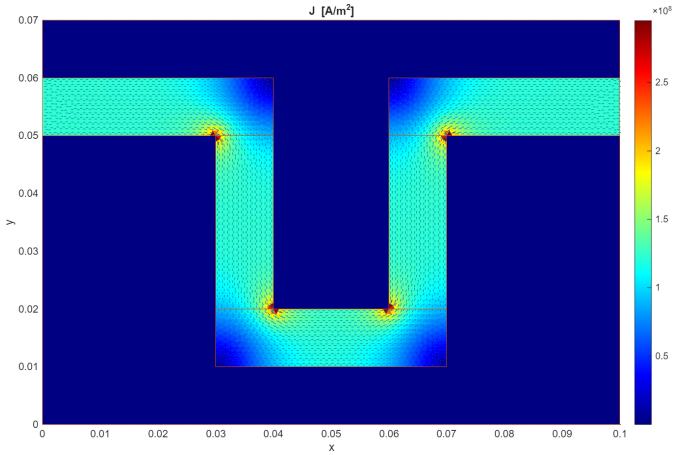


Figure 6.29: Current density distribution in the PCB Heater.

Finally, we can also find the equivalent resistance of the device by evaluating the total losses `P` and then finding the resistance as

$$R_{eq} = \frac{(\Delta V)^2}{P}$$

The commands are

```
fem_full = assembleFEMatrices(model);
P=V'*fem_full.K*V;
dV=1;
R=dV^2/P
```

The result should be $R = 8.0940e-04\ \Omega$

6.9 Assignments

6.9.1 Mandatory assignments

1. Script of the example described in Section 6.3 (2D capacitor with symmetries)

2. Script of the example described in Section 6.6 (Planar Inductor, magnetostatic, 2D-axisymmetric)

3. Script of the example described in Section 6.7 (Thermal Chip Problem)

6.9.2 Optional assignments

1. (+ 0.5 point to be checked during oral examination) Script of the example described in Section 6.5 (Magnetoquasistatic in 2D-axisymmetric - Induction Heating Coil - AC)

2. (+ 1 point to be checked during oral examination) Find the losses (power density distribution) in the billet of the problem in Section 6.5 and solve a Thermal problem with these losses.

Chapter 7

Finite Element with COMSOL

In this chapter, a brief introduction to COMSOL is given. For getting started with COMSOL, it is also suggested to check the COMSOL Application Library: The Application Gallery features COMSOL tutorials and demo app files pertinent to the electrical, structural, acoustics, fluid, heat, and chemical disciplines. You can use these examples as a starting point for your own simulation work by downloading the tutorial model file and its accompanying instructions.

7.1 Introduction

To start a new COMSOL project, open it. You should get the screen below.

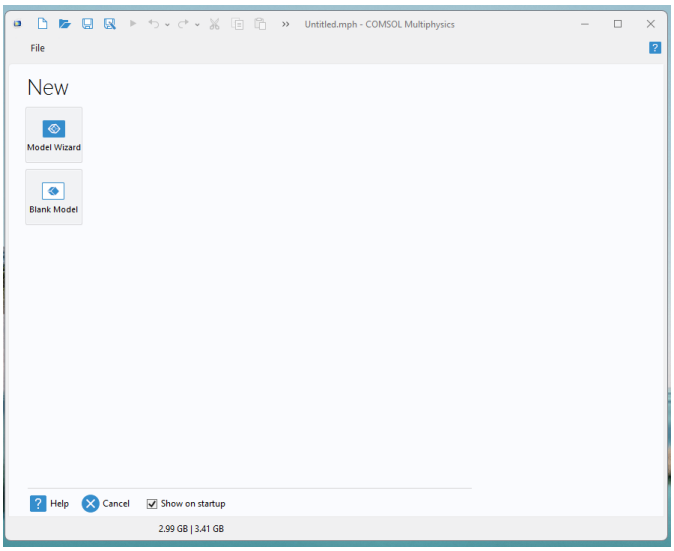


Figure 7.1: Start a new COMSOL project

Click on Model Wizard, and you will get the screen below.

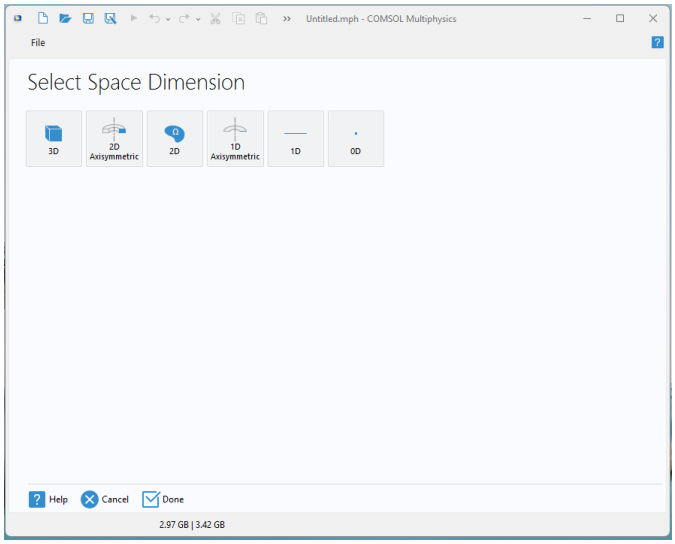


Figure 7.2: Select the dimensionality of the problem.

Here we can select the dimensionality of the problem (3D, 2D, 2D-axisymmetric, etc). We can select, for instance a 3D model and then click "Done". You will be redirected to the screen below.

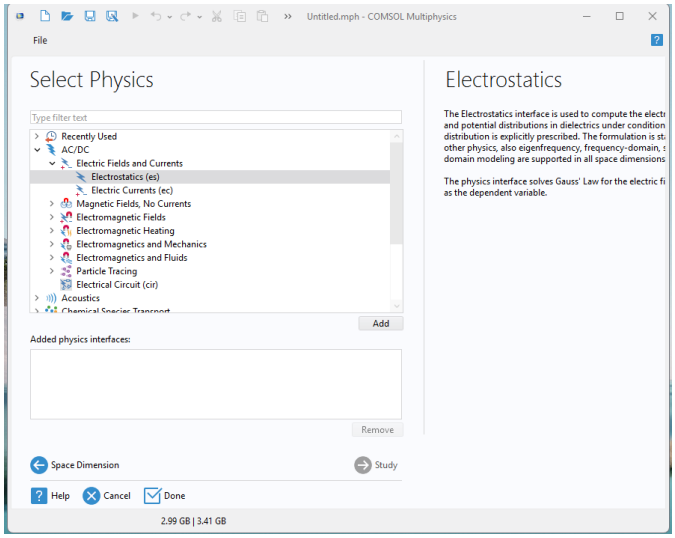


Figure 7.3: Select the formulation (also more than one for multiphysics problems).

At that point, we can select the physics and the specific formulation we want to study. The AC/DC module has many different formulations:

- **Electric Currents (ec)**: Models conduction currents in conductive media under DC or low-frequency AC; includes resistive, capacitive, and inductive effects.

- **Magnetic Fields (mf)**: Computes static and time-harmonic magnetic fields using vector and scalar potentials; includes currents and magnetization.

- **Magnetic Fields, No Currents (mfnc)**: Magnetostatic formulation without free currents; useful for modeling magnetic materials and permanent magnets.

- **Magnetic and Electric Fields (mef)**: Full-wave formulation coupling magnetic and electric fields at low frequencies (quasistatic).

and many others.

COMSOL is a multiphysics software, so other packages can be used for, e.g., thermal, mechanics, and fluid dynamics studies. We can also select more than one physics and solve a multiphysics problem (e.g., electromagnetic and thermal).

We can, for instance, select Electrostatics (ec) and then "Done". We will move to the screen below.

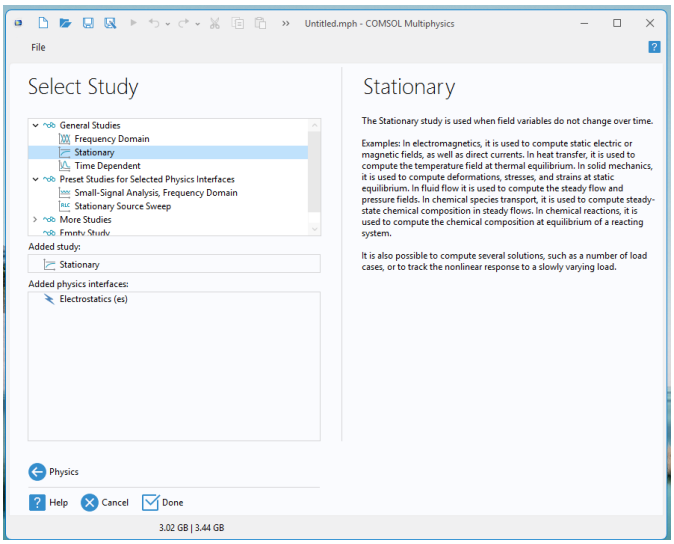


Figure 7.4: Select the study type.

Here we can select the kind of study we want to perform: stationary, frequency domain, transient, and many others. We can select stationary and then "Done". We will get the screen below.

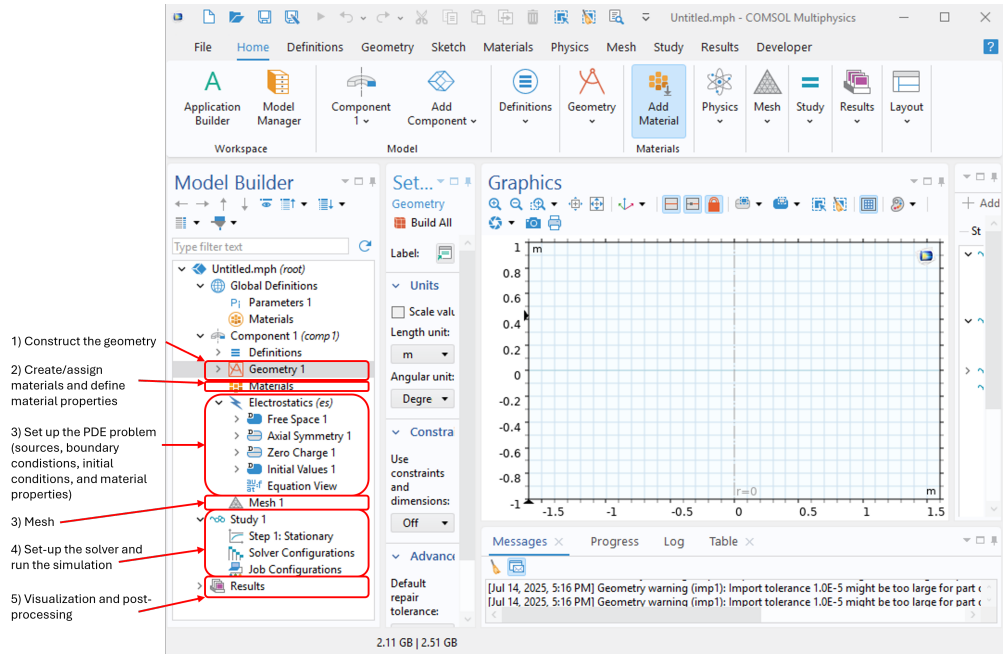


Figure 7.5: New project interface.

We are now at this main interface where we can construct the model and simulate it. In particular, in the following suggested order, we have to:

- construct the model geometry,
- assign the material parameters to the domains,
- set up the PDE problem: sources, boundary conditions, initial conditions (for transient problems), etc.,
- mesh the geometry (COMSOL does that automatically, but we can guide it; in many situations, it is suggested to do not to use its automatic meshing).
- set up the solver configuration (COMSOL does that automatically, but we can also decide which solver to use, e.g., direct or iterative, among many different methods) and finally run the simulation,
- visualize the results and do some post-processing.

COMSOL has many functionalities that require time to be learned. In general, to add a feature in any step previously mentioned, right-click on the step name (e.g., Geometry, Materials, Electrostatics, etc.) and a menu with all the possible features will open.

We will gain some expertise with the project described below.

7.2 Planar Inductor: Magnetoquasistatic and Thermal 2D-axisymmetric

We study an axisymmetric planar inductor above a magnetic core, with the mf formulation of COMSOL. We study the magnetic problem first. Then, once we obtain the electromagnetic losses, we also perform a thermal simulation. But first, we just consider the magnetic problem and we use the mf module of COMSOL.

7.2.1 Construction of the geometry

The geometry of the axisymmetric model is shown below. To create it, right-click on geometry and add rectangular objects, etc. There are many ways to create this geometry. Also, remember to create a surrounding sphere. Since we are solving an axisymmetric problem, COMSOL will then select only the right part of the geometry with respect to the axisymmetric-axis.

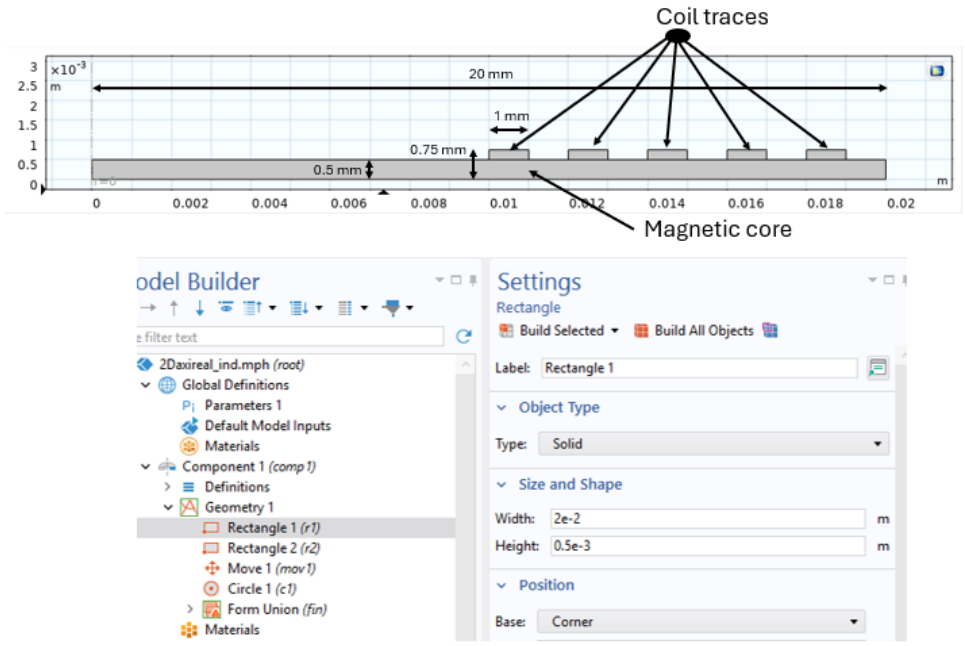


Figure 7.6: Geometry of the axisymmetric planar inductor

To create the geometry, click on the "Build All Objects" button.

7.2.2 Setting up the PDE problem

Introduction

You are probably wondering why we skipped the "Materials" step. Actually, the material properties can be assigned both in the "Materials" and "Magnetic Fields (mf)" fields. Here, we assign them directly in the "Magnetic Fields (mf)".

We can also visualize the equation that is solved by the mf module by clicking on "Magnetic Fields (mf)" and then opening "Equation". WARNING: to do that, please click on the button highlighted in red (Show more option) and activate all the voices.

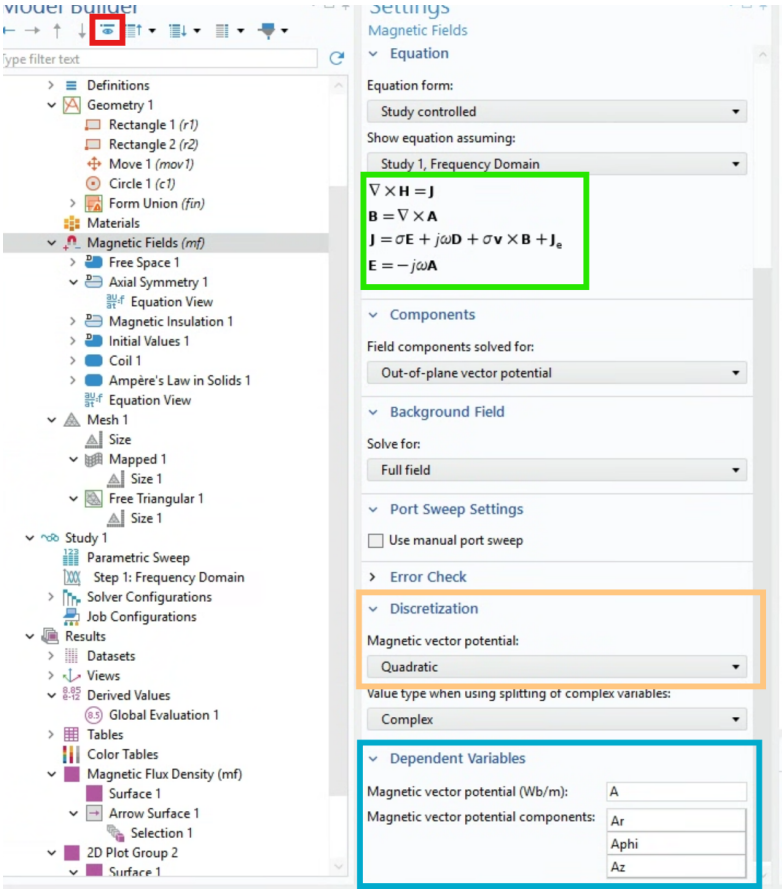


Figure 7.7: Equation solve by COMSOL mf physics

As can be seen (green block), in mf COMSOL is actually solving Full-Maxwell equations since all the effects are present. However, this formulation is numerically suitable only for low-frequency problems (it is an A-formulation). The mef formulation (A-V formulation) or the Radio-frequency package (E-formulation) are

better for high frequency problems. In Figure 7.7 we can also visualize which unknown is used as an independent variable (blue block). It is the magnetic vector potential (mf is indeed an A-formulation and only Aphi since we are dealing with an axisymmetric problem. We can also check the order of the shape functions it is using (orange block). We can also change the order.

COMSOL automatically adds some default stuff, i.e.,

- Free Space: as default, any domain in the model is considered as free space (with electromagnetic properties of vacuum $\varepsilon_r = 1$, $\mu_r = 1$, $\sigma = 0$ S/m).
- Axial Symmetry: it applies the correct boundary condition on the axisymmetric axis (Dirichlet boundary condition)
- Magnetic Insulation: Neumann boundary condition applied on the boundary of the computational domain (i.e., the surrounding sphere)
- Initial Values: not relevant for the frequency domain study we are going to perform, but for the transient, we can modify it. The default is zero everywhere.

We can now set up the problem.

Coil feature

First, we add the coil feature (right-click on mf and select "coil").

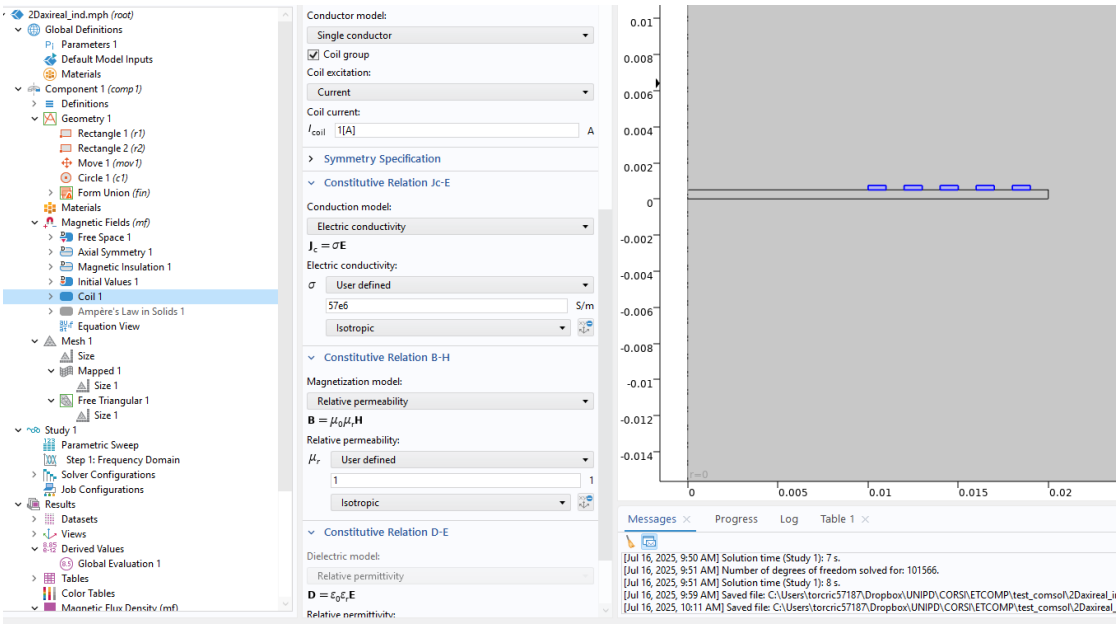


Figure 7.8: Adding the coil feature

We first select all the domains where the coil feature is applied (the 5 traces). Then, for the conductor model, we decide to treat it as a single conductor (that means it is treated like a solid conductor). Other options include Homogenized multiturns that is to consider the conductor like a litz coil.

We also select the "Coil group" box. This allows us to consider the 5 traces as they are connected in series.

For the Coil excitation, we decide to select "Current" and then 1 A.

At this point, we can select the material properties of the domain (copper): σ = 57e6 S/m, μ_r = 1, ε_r = 1.

Magnetic Core

For the magnetic core we just have to add the "Ampère Law in Solid" and select the magnetic core domain. This basically tells COMSOL to solve the same equation in that domain.

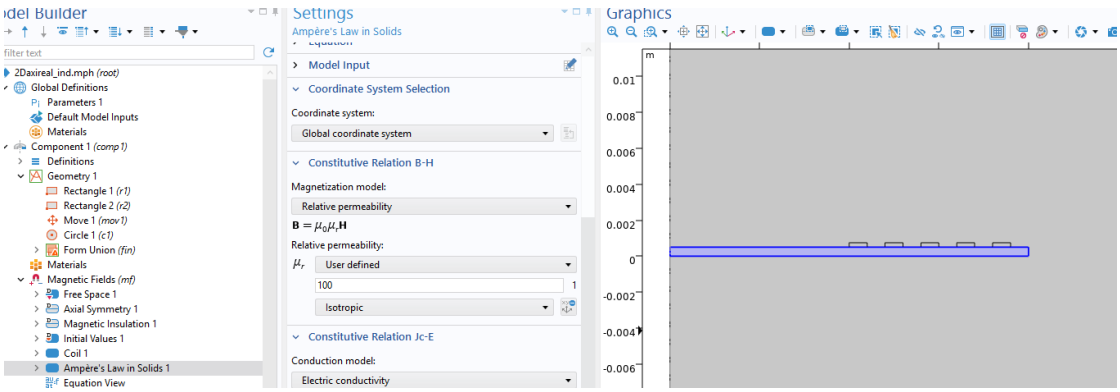


Figure 7.9: Adding the "Ampère Law in Solid" feature for the magnetic core

Then, we impose the material properties by selecting "user defined" for the Constitutive Relation Jc-E, Constitutive Relation B-H, and Constitutive Relation D-E. We assign the following values μ_r = 100, σ = 0 S/m, ε_r = 1.

7.2.3 Meshing

We can now mesh the structure. We can use the default meshing option, or we can guide the mesh by constructing a fine mapped mesh inside the coil and magnetic core domains and a free triangular mesh in the surrounding sphere.

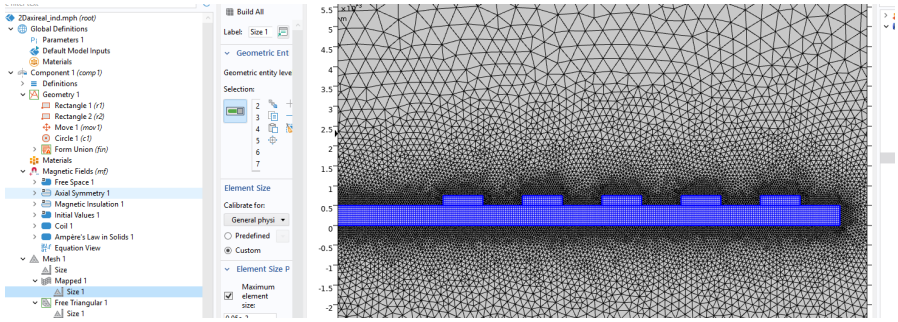


Figure 7.10: Meshing.

7.2.4 Solving the problem

We are now ready to solve the problem. Since we selected the frequency domain study, we first have to select the frequency of the problem (300 kHz).

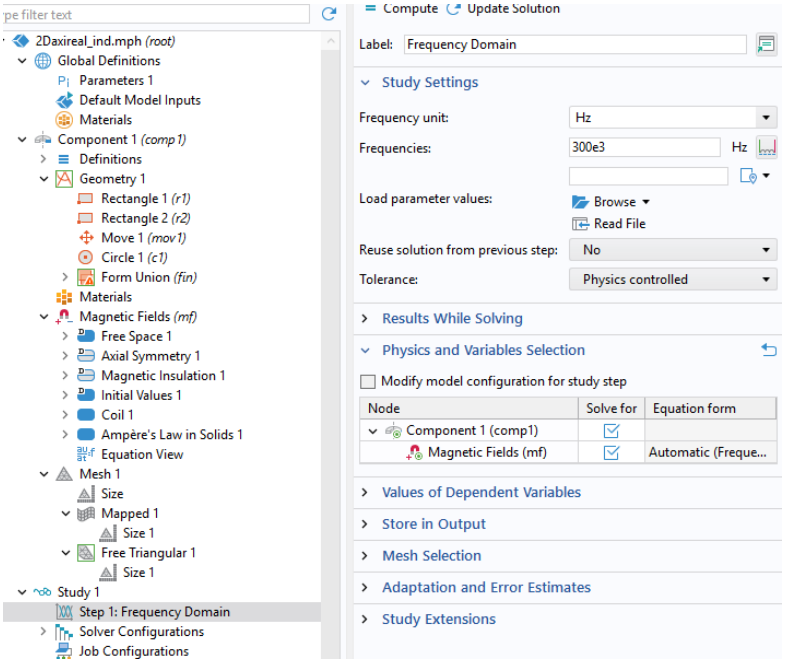


Figure 7.11: Assigning frequency and solving.

Then, we just have to click "compute". COMSOL will use a default solver strategy. We could modify that by right-clicking on the study step and then "Solver Configurations".

7.2.5 Visualization and Post-Processing

After the problem is solved, COMSOL automatically creates some plots to visualize the results. Obviously, we can also add other plots. For instance, it can also be useful to visualize the current density distribution in the coil traces.

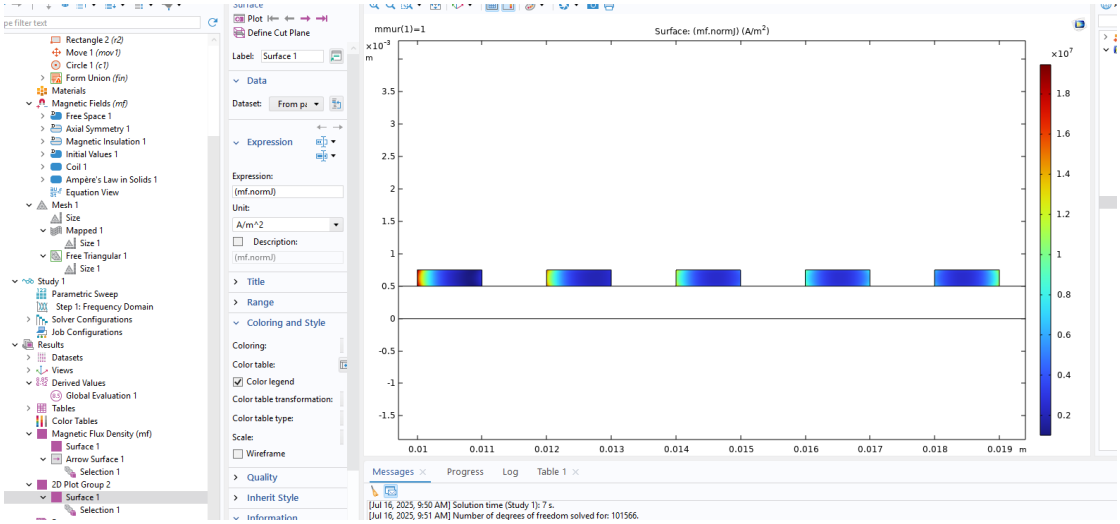


Figure 7.12: Visualize the current density distribution.

To do that, right-click on "Results" and then select 2D plot group. Then, right-click on "2D Plot Group 2" you just created and select "Surface". You can write in the expression what you want to visualize. In this case mf.normJ. We can also right-click on "Surface" and add the "Selection" filter. Then we select only the domains where we want to visualize the field.

As we can see, the skin and proximity effects are very pronounced.

7.2.6 Post Processing

Finally, we can also extract some global parameters. For instance, we can evaluate the coil equivalent resistance and inductance. To do that, just go under the "Results" section and right-click on "Derived Values" and select "Global Evaluation". You can now write what you want to extract (mf.RCoil_1 for the coil resistance and mf.LCoil_1 for the coil inductance).

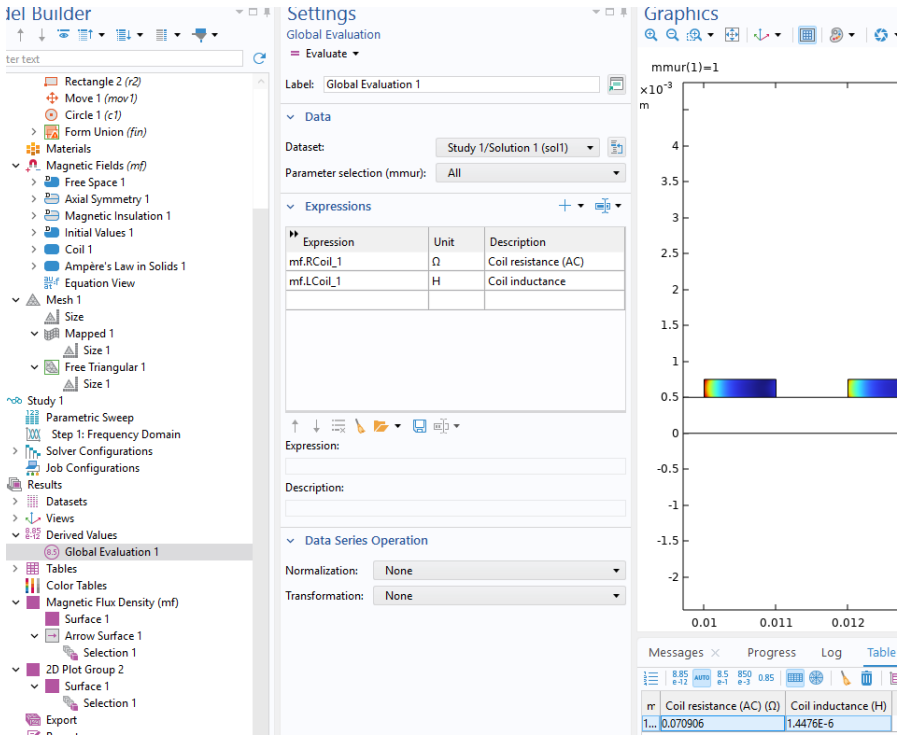


Figure 7.13: Extracting the Equivalent Resistance and Inductance.

You get the results in the table on the panel below the Graphics.

7.2.7 To do things for the Magnetic problem

- Under the "Derived values" field, use surface integration and check that the current in each trace is 1 A (integrate Jphi on one trace). Remember to deselect the "Compute Volume Integral" box on the bottom.
- Change the value of the magnetic core permeability (for instance, try μ_r = 1 → no-core). What happens to the equivalent resistance and inductance of the coil? Why? What happens to the current density distribution?
- Change the size of the surrounding sphere (make it very small). What happens to the results? why?
- Change the mesh size and order of the shape functions. What happens to the results? Why?
- Change the value of the coil current (e.g., 5 A, 100 A, 1000 A). What happens to the fields? What happens to the equivalent resistance and inductance of the coil? Why?

7.2.8 Thermal Problem

Now, we can also add the thermal problem.

Adding the thermal physics

To do that, we first have to add the Thermal physics.

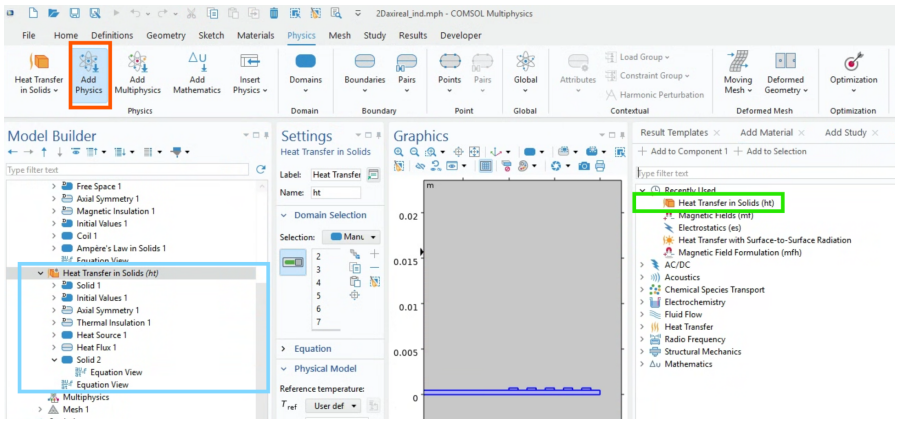


Figure 7.14: Adding the thermal physics.

Click on the "Add physics" button (in the red box) and then double click on the "Heat transfer in Solids (ht)" (orange block). Then, the "Heat transfer in Solids (ht)" physics will appear on the Model Builder block on the left (light-blue box).

Defining the computational domain

Before setting up the physics, we need to define the computational domain. For the magnetic problem, the computational domain was made by the coil, the magnetic core, and the surrounding sphere. In the thermal problem **we do not solve the heat equation in the air domain** (i.e., the surrounding sphere). Doing that would be too complex since we should simulate the convective motion of the air, and we should also add a fluid.-dynamic problem. Instead, we consider only the coil and the magnetic core in the computational domain, and the heat exchange with the surrounding ambient is managed with convective boundary conditions. So, before proceeding, we have to exclude the air domain from the computational domain.

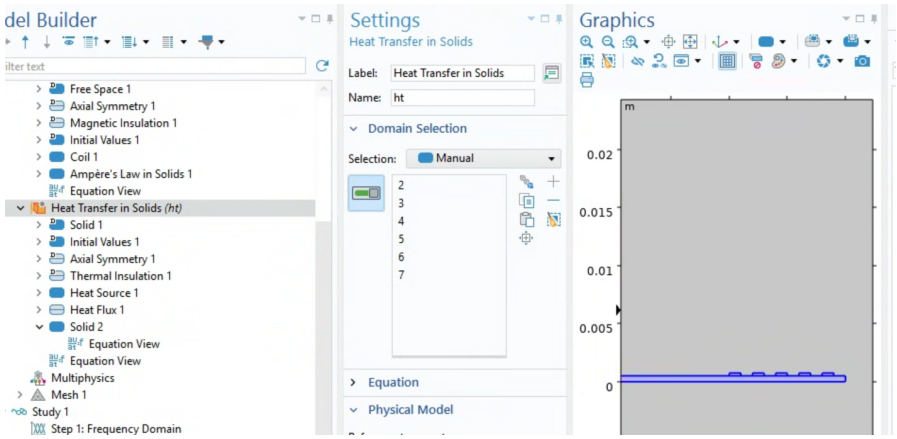


Figure 7.15: Thermal computational domain.

As shown in the image above, click on "Heat transfer in Solids (ht)" and remove the air domain from the "Domain Selection". Also, take a look at the equation you are solving under the "Equation" section.

Setting up the thermal physics

In the "Solid 1" node (added by COMSOL as default)the coil and the magnetic core domain are selected. We can use this node to assign the material properties of the coil. Then, we will generate another "Solid 2" by selecting the magnetic core only and assigning its material properties. For the copper coil:

- thermal conductivity $k = 400$ W/(m K)
- density $\rho = 9000$ kg/m3
- heat capacity $c_p = 385$ J/(kg K)

Then, we can generate "Solid 2" field (right click on "Heat transfer in Solids (ht)") and select only the magnetic core domain. This overwrites the material properties assigned in "Solid 1". For the magnetic core:

- thermal conductivity $k = 6$ W/(m K)
- density $\rho = 5000$ kg/m3
- heat capacity $c_p = 1000$ J/(kg K)

We can now assign the convective boundary condition. Right-click on "Heat transfer in Solids (ht)" and add "Heat Flux". Then, under "Flux type", select

"Convective heat flux". You can now select the heat transfer coefficient and the ambient temperature. Let's use $h = 10$ W/(m2 K) and T_amb $= 293.15$ K.

Finally, to apply the losses (i.e., the excitation of the thermal problem) right-click on "Heat transfer in Solids (ht)" and select "Heat Source". Then, inside the Heat source box, write the expression mf.Qh. This is telling COMSOL to use the Joule losses evaluated as post-processing in the magnetic simulation.

Meshing

We can skip this part and just use the mesh previously constructed for the magnetic problem.

Add Thermal Study

We can now add a study for the thermal problem.

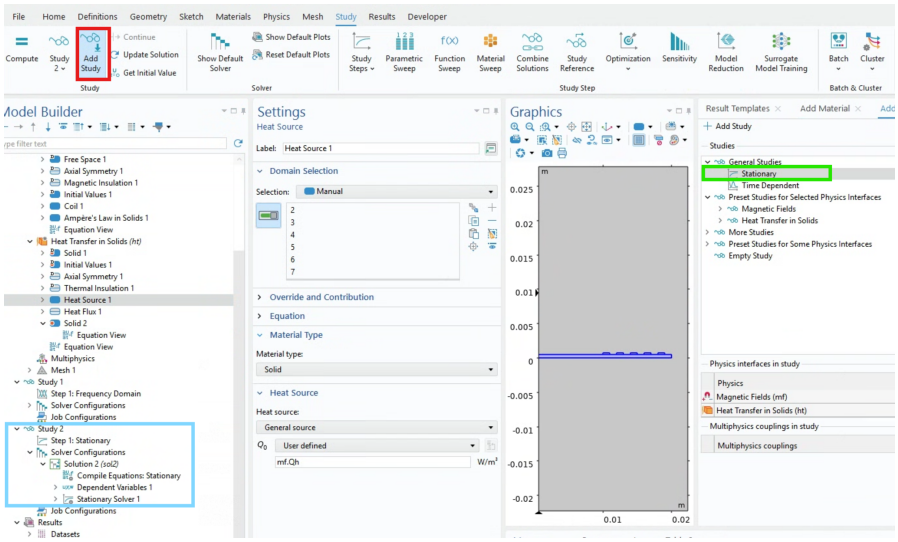


Figure 7.16: Adding the study for the thermal problem.

To do that, click on "Add Study" (red box), then select "Stationary" (green box). A new "Study 2" will appear (light-blue box). Here, we are solving a steady-state thermal problem. However, we could also study a transient problem.

Setting Up the Study

Before running the simulation, we need to fix this study. First, under the "Physics and Variable Selection", we have to deselect the "Magnetic Fields (mf)" box (we have already solved the magnetic problem!).

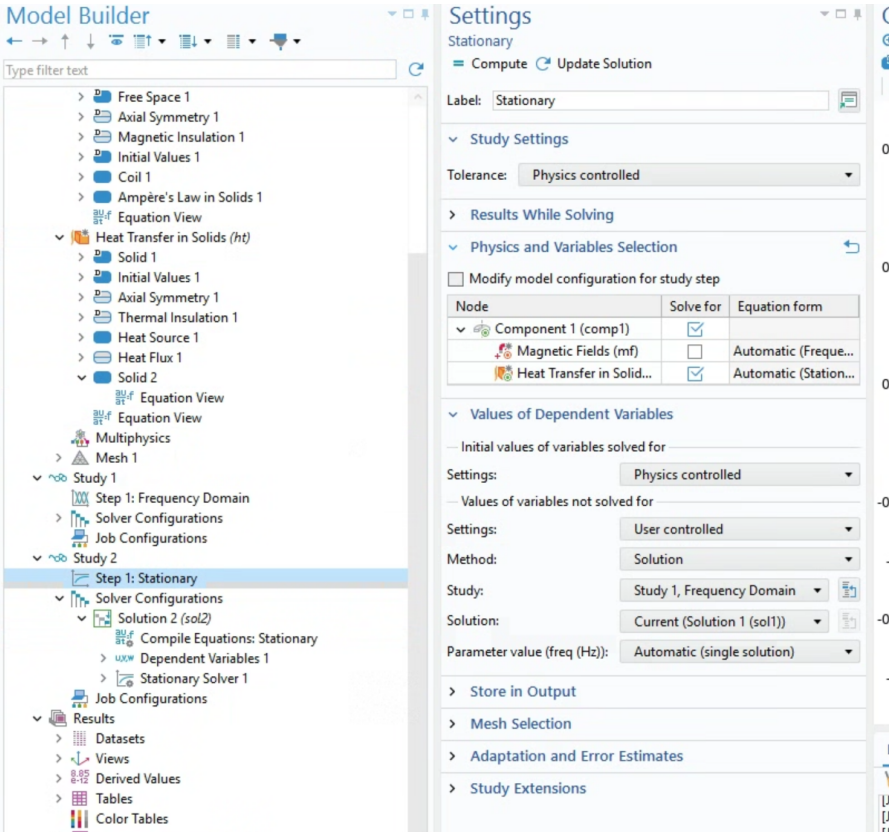


Figure 7.17: Setting Up the Study.

Then, we need to inform the study that the losses (mf.Qh) have been evaluated in "Study 1". To do that, go to the "Values of variables not solved for" and set it as in the figure above.

Then, we can finally run/simulate the problem.

Visualization

We can finally evaluate and visualize the temperature distribution. COMSOL automatically generates this result.

If you select a 5 A current as input current of the coil, you'll get that the temperature is almost 50°C everywhere.

7.3 Electrostatic DC in 2D-axisymmetric - Plane Capacitor

Try to solve the problem "Electrostatic DC in 2D-axisymmetric - Plane Capacitor" previously solved with the MATLAB PDE Toolbox (see Section 6.4) and compare the results.

7.4 Magnetoquasistatic in 2D-axisymmetric - Induction Heating Coil - AC

Try to solve the problem "Magnetoquasistatic in 2D-axisymmetric - Induction Heating Coil - AC" previously solved with the MATLAB PDE Toolbox (see Section 6.5) and compare the results.

7.5 Current Flow in 2D - PCB Heater

Try to solve the problem "Current Flow in 2D - PCB Heater" previously solved with the MATLAB PDE Toolbox (see Section 6.8) and compare the results.

7.6 3D Planar Inductor: magnetoquasistatic and thermal

This is a challenging task: Modelling a 3D planar inductor above a magnetic core. The geometry of the copper inductor is available on Moodle and can be loaded with the import feature below the Geometry field. Then, the a magnetic core can be generated by creating a block of size 1.1 cm × 1.1 cm × 0.3 mm centered in the origin so it is directly touching the copper inductor.

The magnetic core has a relative permeability of 100, and we solve the problem in the frequency domain at 300 kHz. We solve the problem with COMSOL mf module.

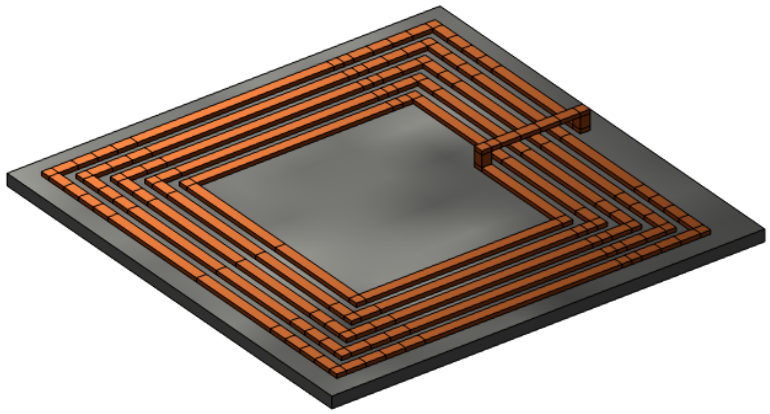


Figure 7.18: 3D COIL model.

To solve the problem, follow the same steps we did for the 2D case. Attention is needed on the following steps:

- For the "Coil" feature, you need to select an internal face of the coil in the

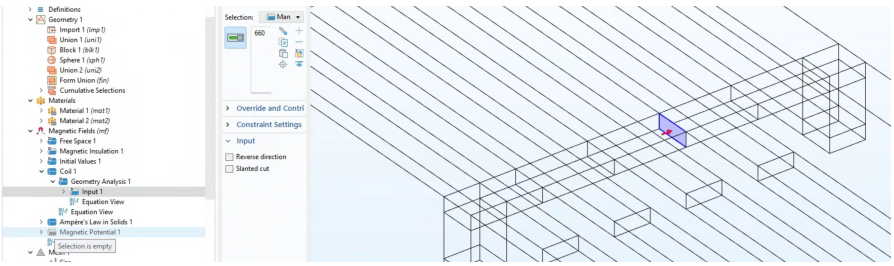


Figure 7.19: Assign the input face to the coil.

- In the Study node, add a "Coil Geometry Analysis step". This is required by COMSOL to find the current density distribution inside the coil

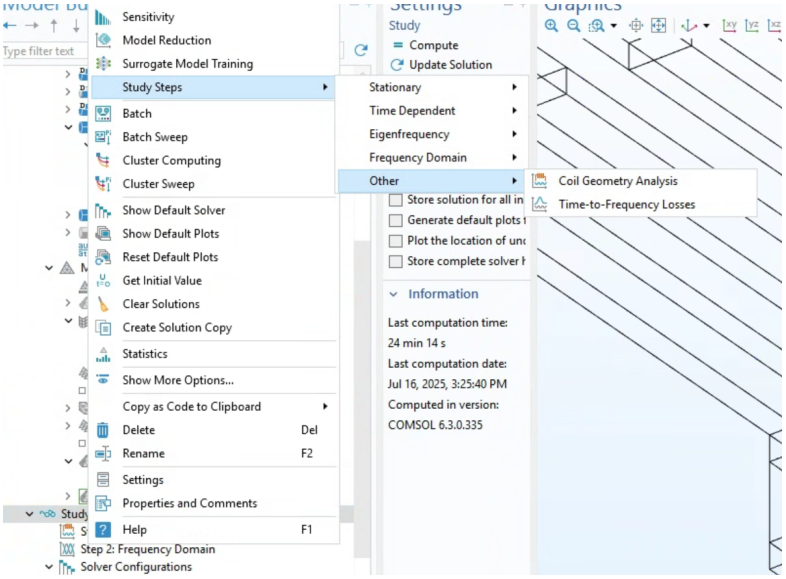


Figure 7.20: Adding "Coil Geometry Analysis step".

Results in terms of the B field are

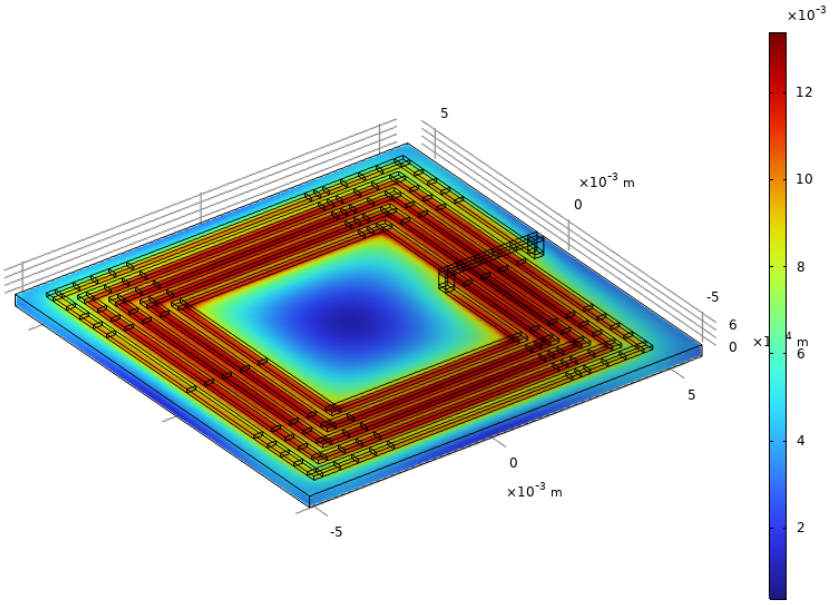


Figure 7.21: B-field visualization. Results are in [T] for a 1 A coil current excitation.

Finally, we can also extract the coil equivalent resistance and inductance. R_eq=0.66 Ω and L_eq=504 nH.

For the thermal simulation, just repeat what we have done in the 2D-axisymmetric planar inductor example.

7.7 Non-linear Inductor 2D Magnetoquasistatic

We study a non-linear magnetic coil in 2D (i.e., with a magnetic core that exhibits saturations: the constitutive B-H relation is modeled with a BH curve). The geometry is given below.

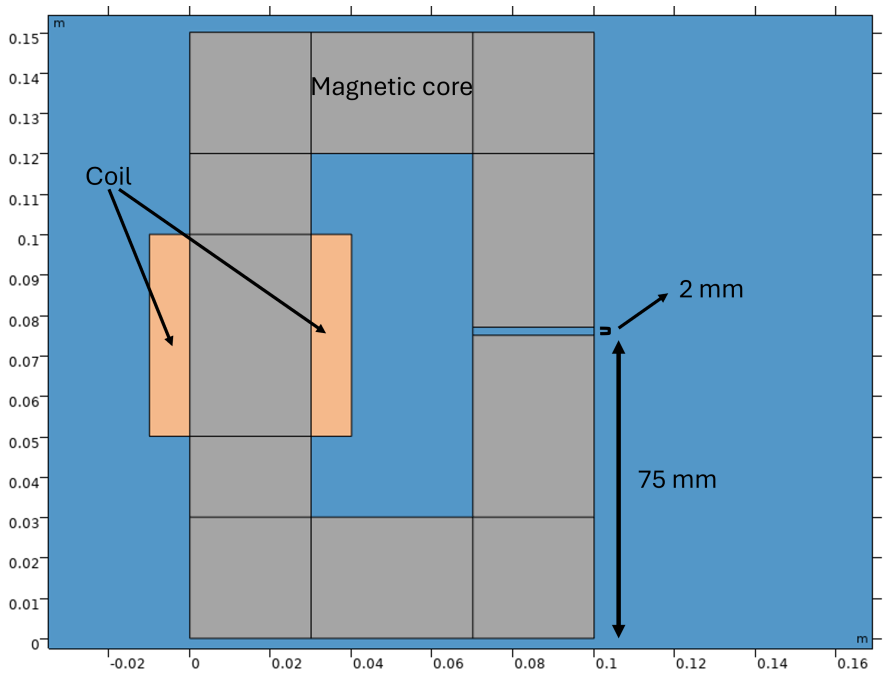


Figure 7.22: Non-linear coil model.

We use the mf module of COMSOL and a frequency domain study at 50 Hz.

Adding the Materials

In this example, we add the material in the "Materials" step. COMSOL has its own internal material library. We look for "Soft Iron (Without Losses)" and we add it.

Then we select the domains of the magnetic core.

We can visualize the BH curve of this material. Here, we are studying a frequency domain problem, so theoretically, we could not solve a nonlinear problem. However, COMSOL allows us to do that in any case by using the *effective* BH curve. Using the *effective* BH allows us to solve the problem in the frequency domain by using a first harmonic approximation (more information can be found in the related literature).

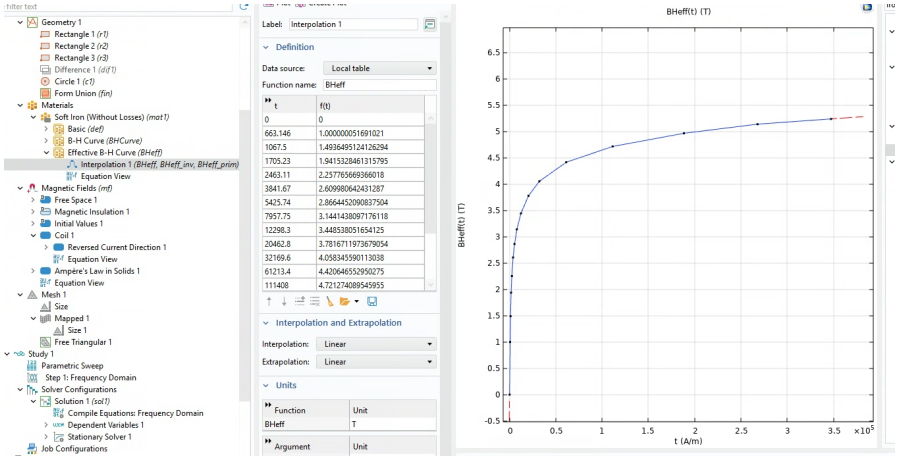


Figure 7.23: Effective BH curve.

7.7.1 Adding the Effective BH-curve to the formulation

To add the selected material to the formulation, we add an "Ampère Law in Solids" and we select the core domain. Then, in the "Constitutive Relation B-H", we select "Effective BH-curve" for the Magnetization model, and then "From material" in the other fields.

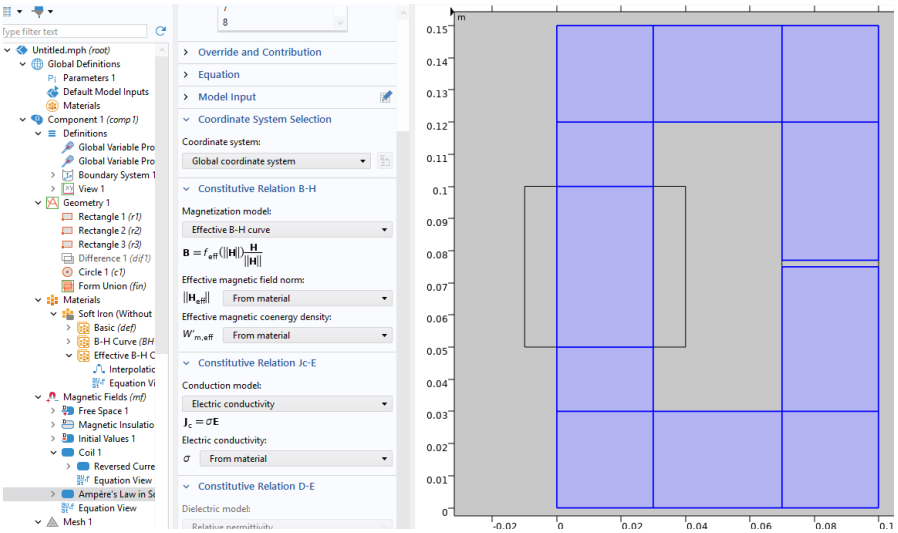


Figure 7.24: Adding BH curve in the mf formulation.

7.7.2 Adding a Parameter for the Coil Current

Also, we add a parameter for the coil currents, so we can run several simulations with different current values and check how the inductance of the device changes. To do that, do what is shown in the image and generate the parameter Icurr.

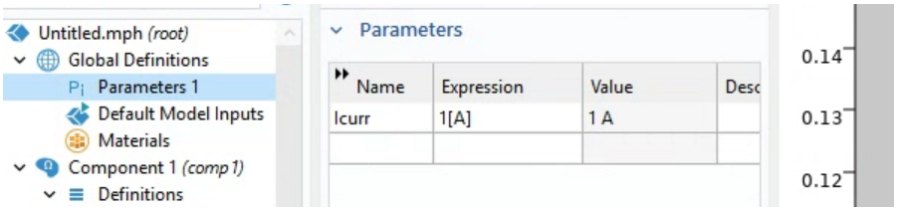


Figure 7.25: Adding a parameter.

We can put any value for Icurr, we will change it later.

7.7.3 Adding the Coil with Reverse Current in one Domain

The coil has two domains, where the current has an opposite direction. We model it as "Homogenized multiturn" with a number of turns N=5 and default parameters. Also remember to select the box "Coil Group". The coil current is Icurr, the parameter we previously defined. To select the reverse current, look at the image below.

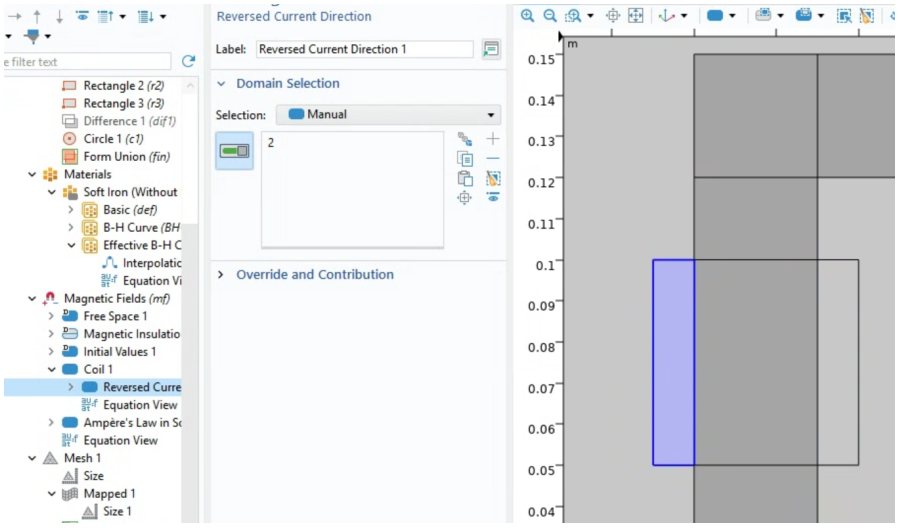


Figure 7.26: Adding the coil with opposite current in one domain.

7.7.4 Parametric Study

We can now move to the study. It is a frequency domain study with a frequency equal to 50 Hz. However, we also add a parametric step to run several simulations with different current values in the range 1 mA to 1000 A.

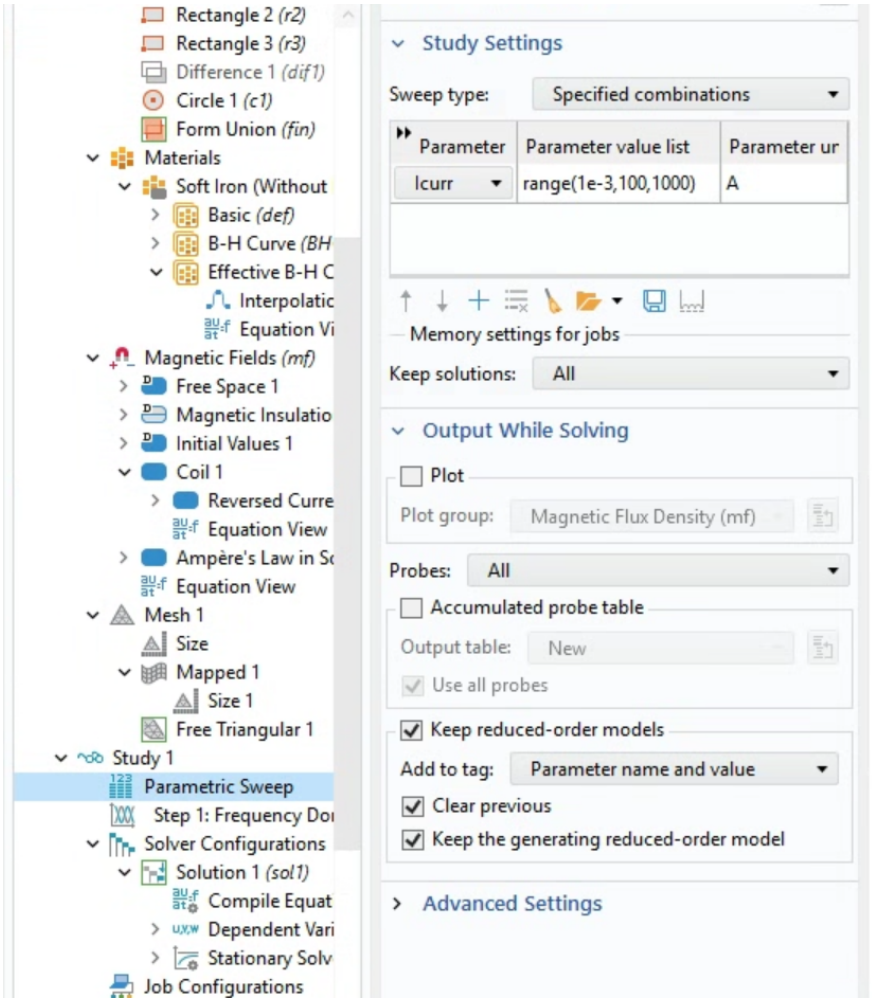


Figure 7.27: Adding the parametric simulation.

7.7.5 Post Processing: Inductance Evaluation

Add a "1D Plot Group", then add a "Global" plot, and then type mf.LCoil_1 into the expression cell to visualize the coil inductance vs the coil currents.

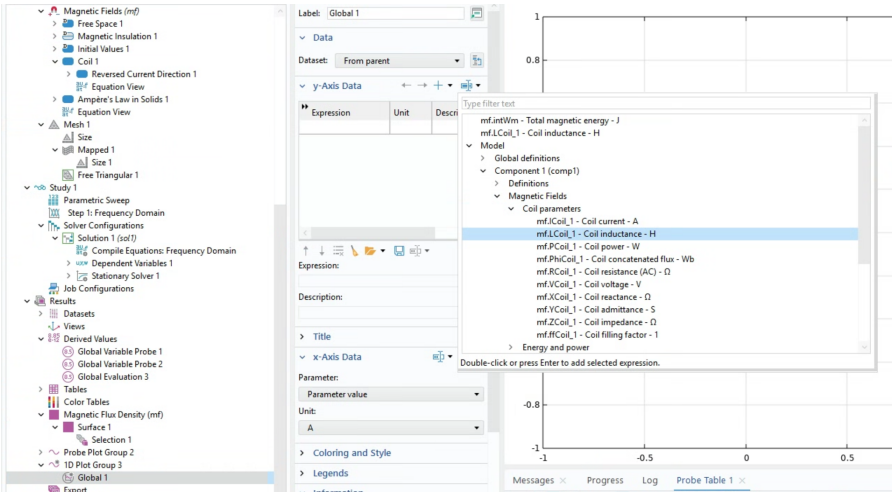


Figure 7.28: Adding the inductance evaluation in post processing.

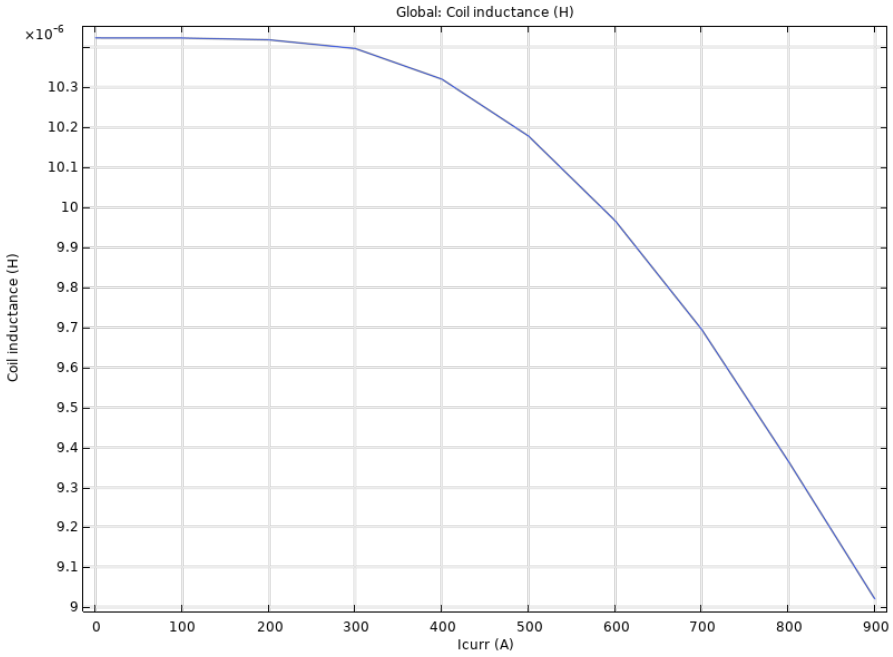


Figure 7.29: Inductance of the coil vs the coil current value.

It is suggested to compute it also from the magnetic energy formula.

7.8 Assignments

7.8.1 Mandatory assignments

- Very short report (a pdf or a power point converted into pdf, with images/screenshots) with results related to Example 7.7, including screenshots of geometry, mesh, electromagnetic field, permeability distribution at different current values, and inductance vs coil current value.

- Very short report (a pdf or a power point converted into pdf, with images/screenshots) of the problem "Electrostatic DC in 2D-axisymmetric - Plane Capacitor" previously solve with the MATLAB PDE-Toolbox, including screenshots of geometry, mesh, and electric potential distribution. Also, extract the capacitance value.

7.8.2 Optional assignments

- (+ 2 points to be checked during oral examination) Report or PowerPoint presentations of Example 7.6, including screenshots of geometry, mesh construction, electromagnetic field distributions, effect of the magnetic core on the electromagnetic fields, and equivalent parameters of the coil (resistance and inductance). Also, perform the thermal analysis.

Chapter 8

Solvers

8.1 Direct and Iterative Solver

Following the problem described in Section 6.1, compare the solution of the linear system (6.6) obtained with a direct method (\command in MATLAB) and an iterative method. For the analysis, assume $\mathbf{A}$ =`fem.Kc`, $\mathbf{x}$ =`ud`, and $\mathbf{b}$ =`fem.Fc`.

MATLAB offers a list of iterative methods that can be retrieved following the link https://it.mathworks.com/help/matlab/math/iterative-methods-for-linear-systems.html. In this example, the `pcg` and `gmres` functions must be used, and their results compared. The general structure of the function call in MATLAB is

```
[x,~,~,rv] = pcg(A,b,tol,maxit);
```

where the tolerance `tol` and the `maxit` parameters are user-defined. For example, set `tol`=1E-6 and `maxit`=200.

The previous solution strategy (similar for the GMRES case) does not consider the preconditioning. An incomplete preconditioner can be constructed using the MATLAB functions `ichol` and `ilu` (see MATLAB online helper for documentation), corresponding to the incomplete Cholesky and incomplete LU factorizations. The preconditioner CG is called by using the following command

```
1 [x,~,~,rv] = pcg(A,b,tol,maxit,U,U');
```

where U is the preconditioning matrix, specified as input. A similar approach is used for the GMRES case by using LU factorization.

The MATLAB script must produce a convergence plot for the iterative solvers in terms of relative residual versus iteration number, using the following command

```
1 semilogy(0:length(rv)-1,rv/norm(b),'-o')
```

An example of the output is shown in Fig. 8.1

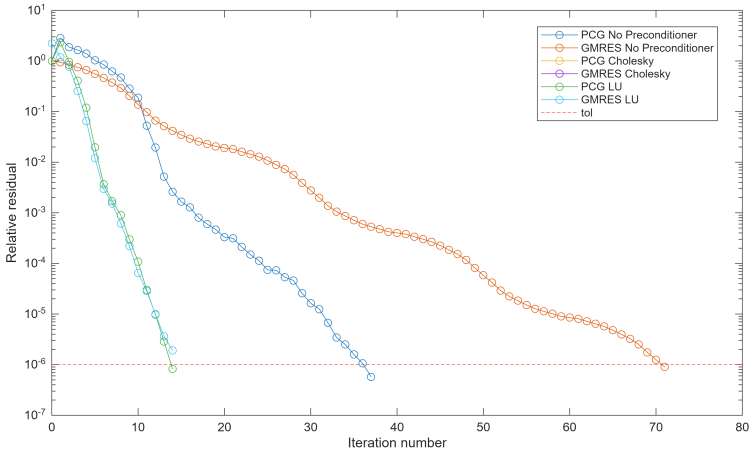


Figure 8.1: Convergence plot of iterative solvers.

In addition, a table with the relative error norm (percentage) of each solution, obtained with a different iterative method, with respect to the solution vector $\mathbf{x}$ computed with the direct method must be provided. The relative error norm can be defined as

$$err = 100\frac{\|\mathbf{x}_{iterative} - \mathbf{x}\|}{\|\mathbf{x}\|} \tag{8.1}$$

8.2 Steepest Descent Implementation

This numerical example is devoted to the MATLAB implementation of the steepest descent algorithm described in the lecture notes, here reported for clarity. The MATLAB function must accept as input, together with $\mathbf{A}, \mathbf{b}, \mathbf{x_0}$, and the tolerance factor, also the maximum number of iterations (maxit) for the iteration counter k. The solution obtained with the implemented method, in terms of relative residual, has to be compared with the results obtained in Section 8.1.

Algorithm 26 Steepest descent algorithm.

Input: $\mathbf{b}$, $\mathbf{A}$, $\mathbf{x}_0$, τ
Output: $\mathbf{x}$
$\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\epsilon \leftarrow \|\mathbf{r}_k\|$
$k \leftarrow 0$ ▷ iteration counter
while $\epsilon > \tau$ **do**
    $\mathbf{t} \leftarrow \mathbf{A}\mathbf{r}_k$
    $\alpha_k = (\mathbf{r}_k \cdot \mathbf{r}_k)/(\mathbf{r_k} \cdot \mathbf{t})$
    $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{r}_k$
    $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k - \alpha_k \mathbf{t}$
    $k \leftarrow k + 1$
    $k \leftarrow k + 1$
    $\epsilon \leftarrow \|\mathbf{r}_k\|$
    $k \leftarrow k + 1$
end while

8.3 Assignments

8.3.1 Mandatory assignments

- Very short report (a pdf or a PowerPoint converted into pdf, with images/screenshots) with results related to Example 8.1, including a screenshot of the convergence plot and a table with relative error norm.

8.3.2 Optional assignments

- (+ 1 points to be checked during oral examination) Report or PowerPoint presentations of Example 8.2, including the MATLAB script of the implemented algorithm and the convergence plot as in Example 8.1.

Chapter 9

Optimization

9.1 Comparison of Gradient-Based and Stochastic Algorithms

Given the scalar Rastrigin function with N parameters (i.e., $\mathbf{x} = [x_1, \ldots, x_N]$) defined as:

$$f(x) = 10N + \sum_{i=1}^{N} [x_i^2 - 10\cos(2\pi x_i)], \tag{9.1}$$

solve the minimization problem $\min f(\mathbf{x})$ assuming $N = 2$ and the following bounds for each design variable $x_i \in [-5.12, 5.12]$ for $i = 1, 2$. The exact solution of this problem is $\mathbf{x}_{exact} = (0, 0)$ for which $f(\mathbf{x}_{exact}) = 0$. In this exercise, the solution with two approaches must be considered:

- A deterministic (gradient-based) method (search for a suitable MATLAB function to do that **hint:** use MATLAB online helper). **Note:** the gradient-based solver requires also the evaluation of ∇f.

- A stochastic method, e.g., the Genetic Algorithm embedded in MATLAB, calling the `GA` function.

The solution of the exercise must contain:

- A plot of the objective function $f(x_1, x_2)$ (surface and contour). An example can be seen in Fig. 9.1

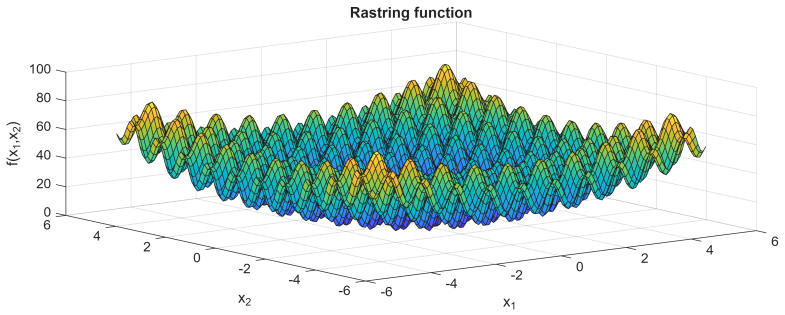


Figure 9.1: Example of surface plot of Rastrigin function.

- A table with the solutions (**x** vector) computed with the gradient-based and the stochastic approach.

- The trend of f_{best} during the iterations of the stochastic approach.

9.2 PSO Algorithm

This example deals with the implementation of the PSO algorithm following the flowchart presented in the Optimization notes. The implemented PSO must work for a general number of parameters N and a general objective function f(x). Also, the user-defined parameters w, c_1, c_2 are inputs of the algorithm, together with the lower and upper bounds for the design variables. The input `obj fun` is a function handle computing the objective for a given array of design variables, e.g.,

```
obj_fun=@(x) fun_objective(x,other_inputs_if_needed)
```

The function `fun objective` is a function with all the steps for constructing the map $\mathbf{x} \mapsto f(\mathbf{x})$ (e.g., realizes the formula of the Rastrigin function (9.1)) The function call of the PSO could looks like:

```
[x_PSO_best,f_PSO_best]=fun_PSO(obj_fun,N,lower_bounds,
    upper_bounds,w,C1,C2)
```

The algorithm is used to find the solution of the minimization problem, considering the same function (Rastrigin) of Example 9.1. The exercise must produce in output the convergence trend of $f_{PSO\ best}$ during iterations. An example can be seen in Fig. 9.2.

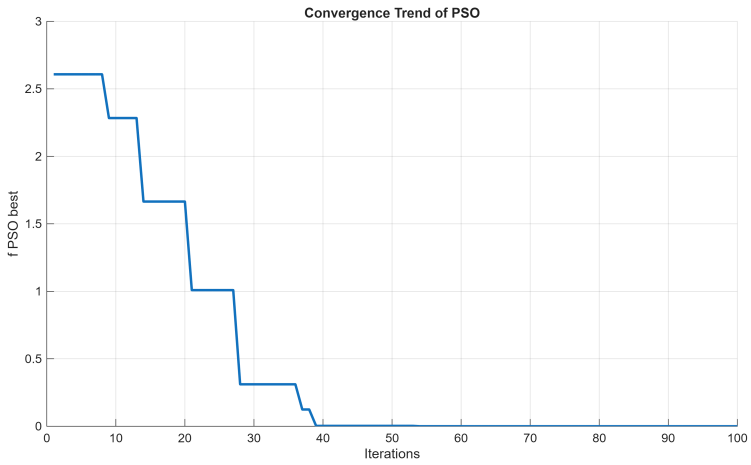


Figure 9.2: Convergence plot of PSO scheme.

9.3 Assignments

9.3.1 Mandatory assignments

- Very short report (a pdf or a PowerPoint converted into pdf, with images/screenshots) with results related to Example 9.1, including a screenshot of the optimized function, results obtained with the gradient-based and the stochastic approaches.

9.3.2 Optional assignments

- (+ 2 points to be checked during oral examination) Report or PowerPoint presentations of Example 9.2, including the MATLAB script of the implemented algorithm and the convergence plot.

- (+ 1.5 points to be checked during oral examination) Matlab Code that optimize the Planar Inductor of Example Section 6.6. The design variables and related bounds are:

  - a=[0.5 mm 1.5 mm]
  - b=[0.15 mm 0.35 mm]
  - d=[0.5 mm 1.5 mm]

The objective is to maximize the inductance. A code that utilizes a stochastic optimization method with an objective function that calls the MATLAB-PDE Toolbox must be provided.

Chapter 10

MATLAB

10.1 The MATLAB Environment

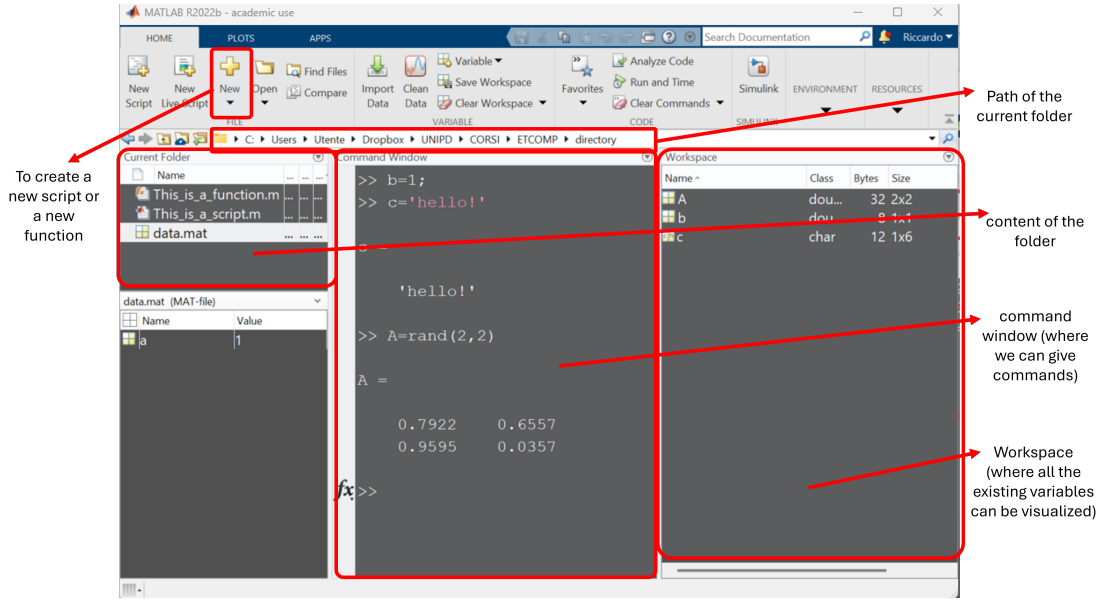


Figure 10.1: MATLAB Enviroment Version R2022b

Figures 10.1 and 10.2 provide an overview of the main components of the MATLAB development environment (as we can see, different versions may have slightly different layouts, but they are substantially the same):

- **New Script/Function Button:** Located in the top left, the `New` button allows users to create new scripts or functions. This is typically the starting point for writing code in MATLAB.

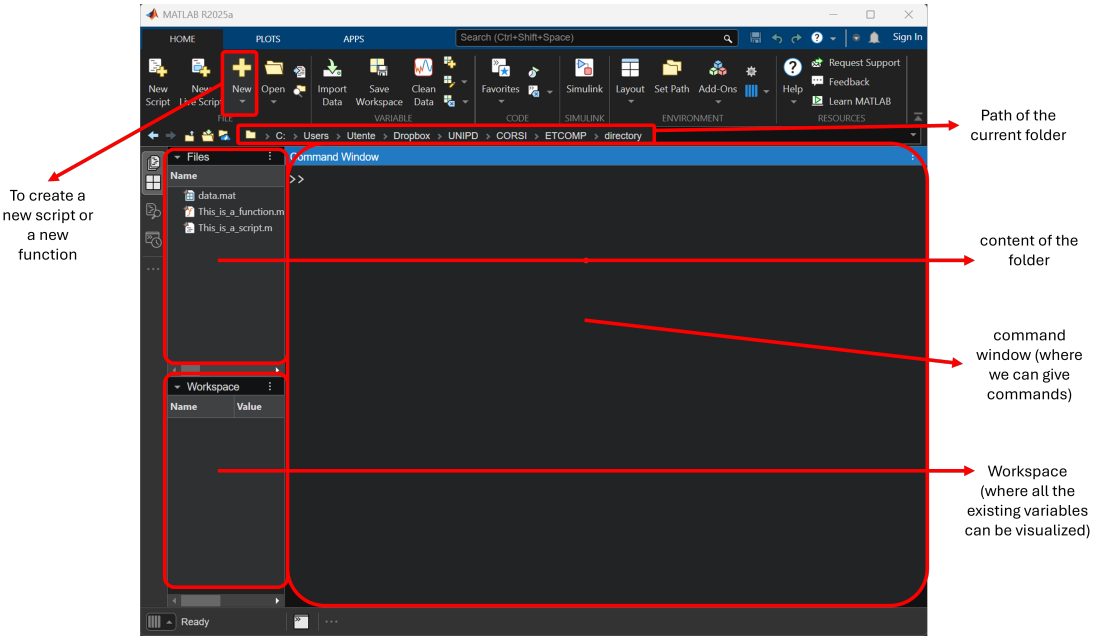


Figure 10.2: MATLAB Enviroment Version R2025b

- **Current Folder Panel:** This area displays the contents of the current working directory. Users can easily access scripts, functions, and data files saved in the folder. The file `data.mat` is an example of a saved workspace file containing variable(s).

- **Path of Current Folder:** The address bar above the folder content shows the full path to the currently active directory.

- **Command Window:** This central panel is where users can type and execute MATLAB commands interactively. As shown, variables like `b`, `c`, and matrix `A` can be defined here and their output will be displayed immediately.

- **Workspace Panel:** This panel lists all current variables in the workspace along with their type (class), size, and memory usage. It provides a quick way to monitor the data being used in the session.

- **MAT-File Content Preview:** When selecting a `.mat` file (e.g., `data.mat`), a preview of its contents is shown, displaying saved variables and their values.

10.2 Frequently used commands

A list of useful MATLAB commands, that can be used either from the command window (prompt >>) or from script (`.m`) files, is given below.

Command	Action
help	Help for functions in Command Window
clc	Clear the command window and home the cursor
who	List all the variables in the current workspace
whos	List all the variables in the current workspace (long form)
clear all	Cancel all the variables in the current workspace
clear <name>	Cancel a single variable in the current workspace
close	Close the current figure window
close <H>	Close the figure window with handle H
format <style>	Change the output display format

Note that the **format** command does not affect how MATLAB computations are done. Computations on float variables, namely single or double, are done in appropriate floating point precision, no matter how those variables are displayed. Computations on integer variables are done natively in integer. Integer variables are always displayed to the appropriate number of digits for the class, for example, 3 digits to display the INT8 range $-128:127$. For a description of possible values for <**style**> see https://mathworks.com/help/matlab/ref/format.html.

10.3 Frequently used functions

A list of the most frequently used MATLAB functions is given below. The input argument is specified between **()**. All function operate element-wise on arrays.

Function	Action
sin()	Return the sine of the argument
cos()	Return the cosine of the argument
asin()	Return the arcsine of the argument
acos()	Return the arccosine of the argument
tan()	Return the tangent of the argument
atan()	Return the arctangent of the argument
exp()	Return the exponential function of the argument
log()	Return the natural logarithm of the argument
log10()	Compute the base 10 logarithm of the argument
sqrt()	Return the square root of each element of the argument
abs()	Return the absolute value (or complex modulus) of the argument
real()	Return the real part of the elements of the complex array in the argument
imag()	Return the imaginary part of the elements of the complex array in the argument
sign()	Return an array Y the same size as th argument X , where each element of Y is 1 if the corresponding element of X is greater than zero or 0 if the corresponding element of X equals zero
factorial()	f = factorial(n) returns the product of all positive integers less than or equal to n , where n is a nonnegative integer value. If n an array, then f contains the factorial of each value of n .
round()	Round each element of the argument to the nearest integer.
floor()	Round each element of the argument to the nearest integer less than or equal to that element.
ceil()	Round each element of the argument to the nearest integer greater than or equal to that element.

10.4 Vectors and Matrices

Whereas other programming languages work mainly on one single number at a time, MATLAB is designed to work mainly on entire arrays (matrices and vectors).

All MATLAB variables are multidimensional arrays, regardless of the data type. A matrix is a two-dimensional array frequently used in linear algebra. Vectors are mathematical entities made by more elements: they can be row-vectors or column-vectors.

10.4.1 Vectors

To generate a **row-vector** put the elements between "**[ ]**" and separate them with a comma "**,**" or a blank space.

```
>> A=[1,2,3]

A =

1      2      3
```

To generate a **column-vector** put the elements between "**[ ]**" and separate them with a semicolon "**;**".

```
>> A=[1;2;3]

A =

1
2
3
```

Generate regularly spaced vectors

The following functions can be used to generate regularly spaced vectors.

Function	Action
x=linspace(a,b,n)	Generates a row-vector with n elements linearly spaced. The spacing between the points is $(b-a)/(n-1)$
x=logspace(a,b,n)	Generates row-vector with n elements between 10^a and 10^b.
x=[a:h:b]	Generates a row-vector with elements evenly distributed between a and b with step-size h

Functions to manipulate vectors

A list of the most frequently used functions to manipulate vectors is given below.

Function	Action
x'	Return the transposed vector (conjugate if complex)
x=[ ]	Return the empty vector
sort(x)	Reorders the components of vector x in ascending order
issorted(x)	Returns the logical scalar 1 (true) when the elements of x are listed in ascending order and 0 (false) otherwise
max(x)	Returns the maximum of x
min(x)	Returns the minimum of x
x(k)	Return the k-th component of the vector
x(k)=z	Assign at the k-th component the z value
x(k)=[ ]	Remove the k-th component
x([h k])=x([k h])	Invert the position of the components h and k

10.4.2 Matrices

To create a matrix (i.e., an array with multiple rows) separate the elements in each row with a comma "**,**" or a blank space, and separate rows with semicolons "**;**".

```
>> A=[1,2,3;4,5,6;7,8,9]

A =

     1     2     3
     4     5     6
     7     8     9
```

Select matrix entries

We can operate on a single entry of a matrix or on several entries all at once.

For example, if. we consider the matrix **A** of the previous example, the command **a=A(2,3)** selects a single entry, namely the element in the second row and third column of the matrix **A**, and assigns it to **a**.

```
>> a=A(2,3)

a =

6
```

On the other hand, by using the special symbol "**:**", we can select an entire row or an entire column. For example, the command **b=A(:,3)** selects the entire third column of the matrix **A** and copies it into **b**.

```
>> b=A(:,3)

b =

3
6
9
```

Finally, the command **c=A(2,1:2)**, selects all elements in the second row which belong to columns 1 and 2 of the matrix **A** and copies them into **c**.

```
>> c=A(2,1:2)

c =

4     5
```

Generate special matrices

The following functions can be used to generate special matrices.

Function	Action
A=eye(n)	Generates the identity matrix with dimension equal to n
A=zeros(n,m)	Generates a matrix with n raws and m columns with all elements equal to 0
A=ones(n,m)	Generates a matrix with n raws and m columns with all elements equal to 1

Functions to manipulate matrices

In MATLAB there are several commands to deal with matrices. A list of the most frequently used is given below.

Function	Action
det(A)	Returns the determinant of the square matrix A
size(A)	[m,n] = size(A) returns the size of matrix A in separate variables m (number of rows) and n (number of columns)
norm(A)	Returns the scalar value $\|\|A\|\|$
x=sum(A)	Returns the raw vector $x = (x_j)_{j=1,\ldots,n}$ with $x_j = \sum_{i=1}^{m} A_{(m,j)}$
x=max(A)	Returns the raw vector $x = (x_j)_{j=1,\ldots,n}$ with $x_j = \max_m A_{(m,j)}$
x=min(A)	Returns the raw vector $x = (x_j)_{j=1,\ldots,n}$ with $x_j = \min_m A_{(m,j)}$
x=diag(A)	Returns the column vector $x = (x_i)_{i=1,\ldots,n}$ with $x_i = A_{(i,i)}$
B=abs(A)	Returns the B matrix where $B(i,j) = \|A(i,j)\|$
B=tril(A)	Returns the lower triangular matrix of A
B=triu(A)	Returns the upper triangular matrix of A
A=[ ]	Returns the empty matrix A
A'	Returns the transposed matrix A
A(r,c)	Extract from the A matrix the elements that belong to the intersection between the raws and the columns specified in **r** and **c**
A(r,c)=C	Assign to the elements specified in **r** and **c** the values specified in C
A(r,c)=[ ]	Removes the elements in A specified in **r** and **c**
A([h k],c)=A([k h],c)	Swaps the elements in the rows **h** and **h**
A(r,[h k])=A(r,[h k])	Swaps the elements in the columns **h** and **k**

10.4.3 Mathematical operations on matrices and vectors

MATLAB allows all values in an array to be processed using a single operator or arithmetic function. When working with arrays we have to distinguish if we are operating in a scalar or vectorial way.

Operation	Meaning
C=A*B	**Matrix product of A and B** Any scalar (a 1-by-1 matrix) may multiply anything. Otherwise, the number of columns of A must equal the number of rows of B.
C=A^ m	**Computes A to the m power**
C=A.*B	**Element-by-element multiplication** A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar.
C = A./B	**Element-by-element division** A and B must have compatible sizes. In the simplest cases, they can be the same size or A be a scalar.
C=A.^ m	**Element-by-element m power** Raises each element of A to the m power.
X = A\B	**Solve systems of linear equations X = A\B for** X The matrices A and B must have the same number of rows.

10.4.4 Sparse matrices

Sparse matrices provide efficient storage of double or logical data that has a large percentage of zeros. While full (or dense) matrices store every single element in memory regardless of value, sparse matrices store only the nonzero elements and their row indices. For this reason, using sparse matrices can significantly reduce the amount of memory required for data storage. For more information, see Computational Advantages of Sparse Matrices.

All MATLAB built-in arithmetic, logical, and indexing operations can be applied to sparse matrices, or to mixtures of sparse and full matrices. Operations on sparse matrices return sparse matrices and operations on full matrices return full matrices. MATLAB never creates sparse matrices automatically. Instead, you must determine if a matrix contains a large enough percentage of zeros to benefit from sparse techniques. Matrices with very low density[1] are often good candidates for use of the sparse format.

[1]The density of a matrix is the number of nonzero elements divided by the total number of matrix elements: `nnz(M)/numel(M);`

Converting Full to Sparse

You can convert a full matrix to sparse storage using the sparse function with a single argument:

```
S = sparse(A)
```

```
>> A = [ 0    0    0    5
         0    2    0    0
         1    3    0    0
         0    0    4    0];
S = sparse(A)

S =

   (3,1)          1
   (2,2)          2
   (3,2)          3
   (4,3)          4
   (1,4)          5
```

The printed output lists the nonzero elements of S, together with their row and column indices between brackets. The elements are sorted by columns, reflecting the internal data structure.

Converting a full matrix to sparse storage is not the most frequent way of generating sparse matrices. If the order of a matrix is small enough that full storage is possible, then conversion to sparse storage rarely offers significant savings.

Converting Sparse to Full

You can also go the other way round and convert a sparse matrix to full storage using the full function, provided the matrix order is not too large. For example, we can reverse the example conversion with:

```
A = full(S)
```

Generating Sparse Matrices

You can generate a sparse matrix directly from a list of nonzero elements using the sparse function with five arguments:

```
S = sparse(i,j,s,m,n)
```

- `i` and `j` are vectors of row and column indices for the nonzero elements of the matrix and `s` is a vector of nonzero values whose indices are specified by the corresponding `(i,j)` pairs

- `m` and `n` are the rows and columns dimension of the resulting matrix

For example, the matrix S of the previous example can be generated with:

```
>>  i=[3 2 3 4 1];
 j=[1 2 2 3 4];
 s=[1 2 3 4 5];
 m=4;
 n=4;
 S=sparse(i,j,s,m,n)

S =

   (3,1)         1
   (2,2)         2
   (3,2)         3
   (4,3)         4
   (1,4)         5
```

10.5 Loops and conditional execution

10.5.1 IF (conditional execution)

A set of instructions (statements 1) are executed when the expression is true. Otherwise an alternative set of instructions (statements 2) are executed[2].

```
if expression    % condition
statements 1     % instructions
else
statements 2     % alternative instructions
end
```

Example

[2]Note that an expression is true when its result is nonempty and contains only nonzero elements (logical or real numeric). Otherwise, the expression is false

```
>> i=2;
if i > 3
        a=5
else
        a=10
end

a =

    10
```

10.5.2 For loop

A group of instructions (statements) are executed in a `for` loop for a specified number of times.

```
for index = values       % condition
statements               %  instructions
end
```

Example 1: Double loop to assign matrix values

```
>> s = 3;
H = zeros(s);

for c = 1:s
    for r = 1:s
        H(r,c) = 1/(r+c-1);
    end
end
H

H =

    1.0000    0.5000    0.3333
    0.5000    0.3333    0.2500
    0.3333    0.2500    0.2000
```

Example 2: Step by increments of +0.5

```
>> for v = 2.0:+0.5:4.0
   disp(v)
```

```
3 end
      2
5
     2.5000
7
      3
9
     3.5000
11
      4
```

Example 3: Execute Statements for Specified Values

```
>> for v = [1 5 8 17]
    disp(v)
end
     1

     5

     8

    17
```

10.5.3 While loop

A group of instructions (statements) are executed in a loop while a condition (expression) is true.

```
while expression          % condition
statements                % group of instructions
end
```

Example: use a while loop to calculate factorial(10)

```
>> n = 10;
f = n;
while n > 1
    n = n-1;
    f = f*n;
end
disp(['n! = ' num2str(f)])
n! = 3628800
```

10.6 Graphic functions

There are various graphic functions that you can use to plot data in MATLAB.The most frequently used graphics functions are illustrated below. Further information are available on line: https://it.mathworks.com/help/releases/R2023b/matlab/creating_plots/types-of-matlab-plots.html?lang=en.

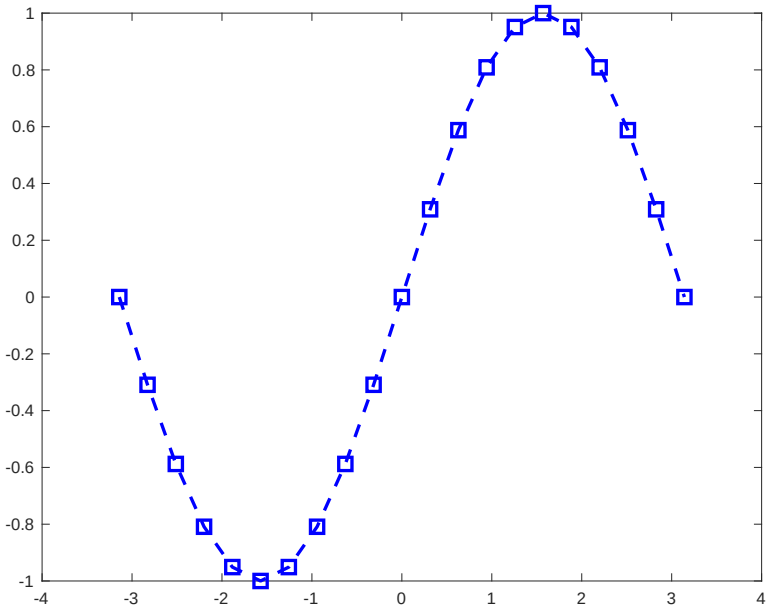
10.6.1 Line Plots

Line plots are a useful way to compare sets of data or track changes over time. You can plot the data in a 2-D or 3-D view using either a linear or a logarithmic scale. The most frequently used are summarised in the following table.

Function	Description	Syntax	URL
plot	2-D Line plot	`plot(x,y,LineSpec)`	plot
plot3	3-D point or line plot	`plot3(x,y,z,LineSpec)`	plot3
errorbar	Line plot with error bars	`errorbar(x,y,err)`	errorbar
loglog	Log-log scale plot	`loglog(x,y,LineSpec)`	loglog
semilogx	Semilog plot (x-axis)	`semilogx(x,y,LineSpec)`	semilogx
semilogy	Semilog plot (y-axis)	`semilogy(x,y,LineSpec)`	semilogy

Example
Create a line plot and use the LineSpec option to specify a dashed blue line with square markers. Use Name,Value pairs to specify the line width, and marker size.

```
x = -pi:pi/10:pi;
y = sin(x);
figure
plot(x,y,'--bs','LineWidth',2,'MarkerSize',10)
```



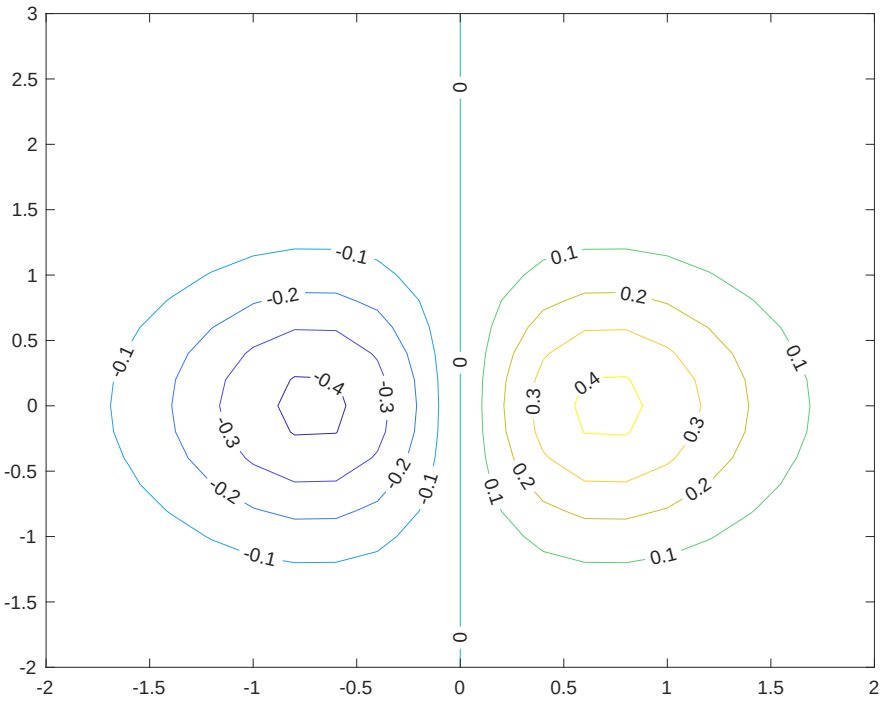
10.6.2 Contour Plots

A contour plot represents a 3-D surface by plotting lines that connect points with common z-values along a slice. For example, you can use a contour plot to visualize the height of a surface in two or three dimensions. The most frequently used commands are summarised in the following table.

Function	Description	Syntax	URL
contour	Contour plot of matrix	`contour(X,Y,Z)`	contour
clabel	Label contour plot elevation	`clabel(C,h)`	clabel

Example
Define Z as a function of two variables, X and Y. Then create a contour plot of that function, and display the labels by setting the ShowText property to 'on'

```
x = -2:0.2:2;
y = -2:0.2:3;
[X,Y] = meshgrid(x,y);
Z = X.*exp(-X.^2-Y.^2);
contour(X,Y,Z,'ShowText','on')
```



10.7 Solving differential equations

An ordinary differential equation (ODE) contains one or more derivatives of a dependent variable, y, with respect to a single independent variable, t, usually referred to as time. The order of the ODE is equal to the highest-order derivative of y that appears in the equation.

The notation used here for representing derivatives of y with respect to t is y' for a first derivative, y'' for a second derivative, and so on. For example, this is a second order ODE:

$$y'' + 3y' + 5y = 0$$

In an initial value problem, the ODE is solved by starting from an initial state. Using the initial condition, y_0 as well as a period of time over which the answer is to be obtained, $[t_0, t_f]$, the solution is obtained iteratively. At each step the solver applies a particular algorithm to the results of previous steps. At the first such step, the initial condition provides the necessary information that allows the integration to proceed. The final result is that the ODE solver returns a vector of time steps $t = [t_0, t_1, t_2, \ldots, t_f]$ as well as the corresponding solution at each step $y = [y_0, y_1, y_2, \ldots, y_f]$

10.7.1 Types of ODEs

The ODE solvers in MATLAB solve these types of first-order ODEs:

- **Explicit ODEs of the form:** $\mathbf{y' = f(t, y)}$
- **Linearly implicit ODEs of the form:** $\mathbf{M(t, y)y' = f(t, y)}$[3]
  where $M(t, y)$ is a nonsingular mass matrix. The mass matrix can be time- or state-dependent, or it can be a constant matrix. Linearly implicit ODEs involve linear combinations of the first derivative of y, which are encoded in the mass matrix.
- **Differential Algebraic Equations (DAEs)**
  If some components of y' are missing, then the equations are called differential algebraic equations, or DAEs, and the system of DAEs contains some algebraic variables. Algebraic variables are dependent variables whose derivatives do not appear in the equations. A system of DAEs can be rewritten as an equivalent system of first-order ODEs by taking derivatives of the equations to eliminate the algebraic variables. The number of derivatives

[3]Linearly implicit ODEs can always be transformed to an explicit form, $y' = M^{-1}(t, y)f(t, y)$. However, specifying the mass matrix directly to the ODE solver avoids this transformation, which is inconvenient and can be computationally expensive.

needed to rewrite a DAE as an ODE is called the differential index. The `ode15s` and `ode23t` solvers can solve index-1 DAEs.

- **Fully implicit ODEs of the form: $\mathbf{f}(\mathbf{t}, \mathbf{y}, \mathbf{y}') = \mathbf{0}$**
  Fully implicit ODEs cannot be rewritten in an explicit form, and might also contain some algebraic variables. The `ode15i` solver is designed for fully implicit problems, including index-1 DAEs.

10.7.2 Basic Solver Selection

`ode45` performs well with most ODE problems and should generally be your first choice of solver. However, `ode23`, `ode78`, `ode89` and `ode113` can be more efficient than `ode45` for problems with looser or tighter accuracy requirements. Some ODE problems exhibit stiffness, or difficulty in evaluation. If you observe that a nonstiff solver is very slow, try using a stiff solver such as `ode15s` instead.

If you are not sure which solver to use, then the following tables provide general guidelines on when to use each solver.

For more information, see Choose an ODE Solver.

Solver	Problem Type	Accuracy	When to Use
ode45	Nonstiff	Medium	Most of the time. ode45 should be the first solver you try
ode23	Nonstiff	Low	Can be more efficient than ode45 at problems with crude tolerances, or in the presence of moderate stiffness
ode113	Nonstiff	Low	Can be more efficient than ode45 at problemswith stringent error tolerances, or whenthe ODE function is expensive to evaluate
ode78	Nonstiff	High	Can be more efficient than ode45 at problems with smooth solutions that have high accuracy requirements
ode89	Nonstiff	High	Can be more efficient than ode78 on very smooth problems, when integrating over long time intervals, or when tolerances are especially tight

Solver	Problem Type	Accuracy	When to Use
ode15s	Stiff	Low to Medium	Try ode15s when ode45 fails or is inefficient and you suspect that the problem is stiff. Also use ode15s when solving DAEs
ode23s	Stiff	Low	Can be more efficient than ode15s at problems with crude error tolerances. It can solve some stiff problems for which ode15s is not effective. ode23s computes the Jacobian in each step, so it is beneficial to provide the Jacobian via odeset to maximize efficiency and accuracy. If there is a mass matrix, it must be constant
ode23t	Stiff	Low	Use ode23t if the problem is only moderately stiff and you need a solution without numerical damping. ode23t can solve DAEs.
ode23tb	Stiff	Low	Like ode23s, the ode23tb solver might be more efficient than ode15s at problems with crude error tolerances.
ode15i	Fully implicit	Low	Use ode15i for fully implicit problems $f(t, y, y') = 0$ and for DAEs of index 1

10.7.3 Function handles

A function handle is a MATLAB data type that stores an association to a function. Indirectly calling a function enables you to invoke the function regardless of where you call it from. Typical uses of function handles include:

- Passing a function to another function (often called function functions). For example, passing a function to integration and optimization functions, such as `integral` and `fzero`.
- Specifying callback functions (for example, a callback that responds to a UI event or interacts with data acquisition hardware).
- Constructing handles to functions defined inline instead of stored in a program file (anonymous functions).
- Calling local functions from outside the main function.

You can see if a variable, h, is a function handle using `isa(h,'function_handle')`

Creating Function Handles

To create a handle for a function, precede the function name with an "@" sign.

For example, if you have a function called `myfunction`, create a handle named `f` as follows: `f = @myfunction;`

You call a function using a handle the same way you call the function directly. For example, suppose that you have a function named `computeSquare`, defined as:

```
function y = computeSquare(x)
y = x.^2;
end
```

Create a handle and call the function to compute the square of four.

```
f = @computeSquare;
a = 4;
b = f(a)

b =

    16
```

You can also create handles to "anonymous functions". An anonymous function is a one-line expression-based MATLAB function that does not require a program file. Construct a handle to an anonymous function by defining the body of the function, `anonymous_function`, and a comma-separated list of input arguments to the anonymous function `arglist`.

The syntax is: `h = @(arglist)anonymous_function`

For example, create a handle, `sqr`, to an anonymous function that computes the square of a number, and call the anonymous function using its handle.

```
>> sqr = @(n) n.^2;
x = sqr(3)

x =

     9
```

10.7.4 Example: solution of a series RLC circuit with ode45

Solve the series RLC circuit shown in figure 10.3, in the time interval $[0, 0.1]$ seconds, with ode45: circuit's parameters and initial conditions are given in the

table below.

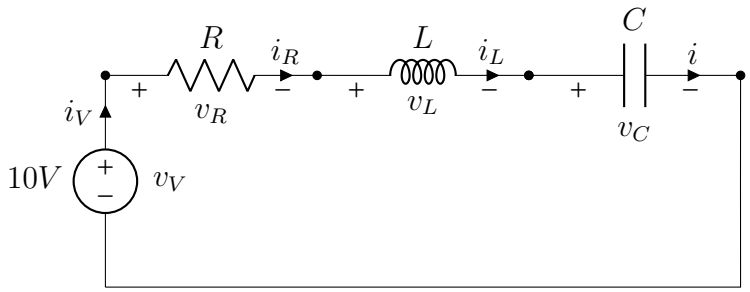


Figure 10.3: Series RLC circuit

Circuit's parameters and initial conditions
$L = 100\,mH \quad C = 10\,\mu F \quad R = 10\,\Omega \quad i_0 = 0\,A \quad v_0 = 0\,V \quad e = 10\,V\,(const)$

To set any problem in a form suitable for solving with ode45, the system of ordinary differential equations has to be in the form $\mathbf{y'} = \mathbf{f(t, y)}$.

Therefore, in our case we need to derive a system of two first-order differential equations in the unknowns v and i, using LKT, LKC and components' equations. From LKT and components' equations we have:

$$i_C = C\,\frac{dv_C}{dt}$$
$$L\,\frac{di_L}{dt} + R\,i_R + v_C = e(t)$$

Then, by LKC ($i_L = i_R = i_C$), and sorting:

$$\frac{dv_C}{dt} = \frac{1}{C}\,i_L$$
$$\frac{di_L}{dt} = -\frac{1}{L}\,v_C - \frac{R}{L}\,i_L + \frac{1}{L}e(t)$$

or, in a more compact form, with matrix notation:

$$\begin{bmatrix} \frac{dv_C}{dt} \\ \frac{di_L}{dt} \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{C} \\ -\frac{1}{L} & -\frac{R}{L} \end{bmatrix} \begin{bmatrix} v_C \\ i_L \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} e(t) \tag{10.1}$$

which has the required form:

$$\mathbf{y'} = \mathbf{A\,y(t)} + \mathbf{B\,u(t)}$$

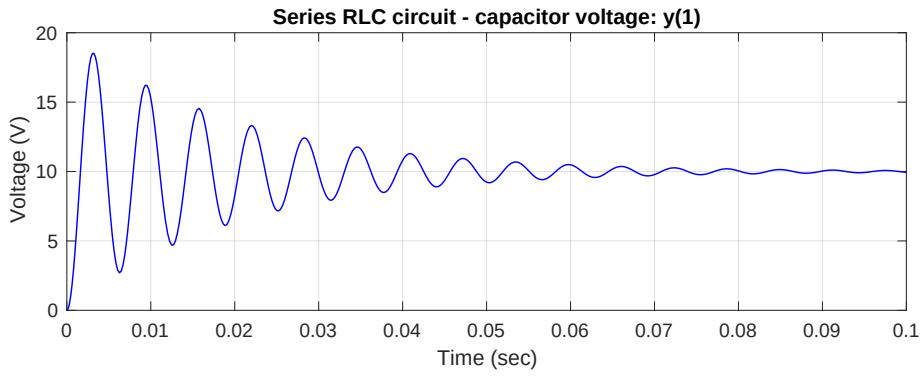
$$y = \begin{bmatrix} v_C \\ i_L \end{bmatrix} \qquad A = \begin{bmatrix} 0 & \frac{1}{C} \\ -\frac{1}{L} & -\frac{R}{L} \end{bmatrix} \qquad B = \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} \qquad u(t) = e(t)$$

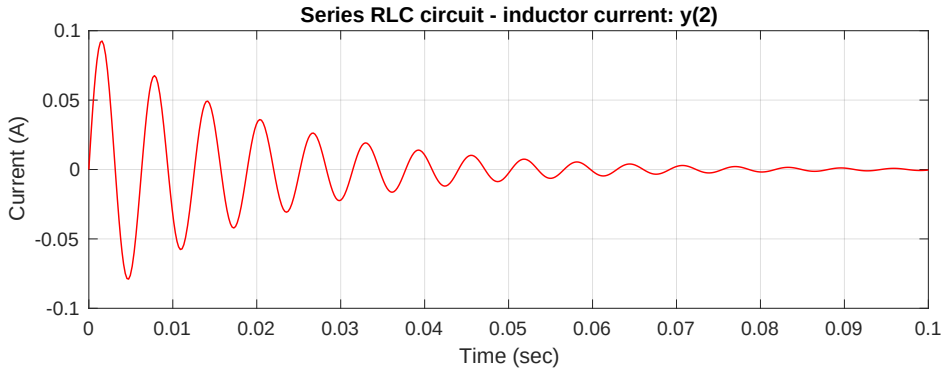
The MATLAB code to solve it with ode45 is the following:

```
tspan = [0, 0.1];          % Simulation Time..
y0 =    [0; 0];            % Initial Conditions

[t, y] = ode45(@(t,y) myfun(t,y), tspan, y0);

%-------------------------------------------------------
function dydt = myfun(t,y)
        R = 10;            % Resistance [Ohm]
        L = 100e-3;        % Inductance [H]
        C = 10e-6;         % Capacitor [F]
        u = 10;            % Input Voltage
        dydt = [0   1/C; -1/L -R/L]*y + [0 ;1/L]*u;
end
```





10.8 Other functions

10.8.1 Stopwatch timer

In MATLAB we can be interested in how long does a program take to compute all the instructions. The commands we need are `tic` and `toc`. The `tic` function records the current time, and the `toc` function uses the recorded value to calculate the elapsed time.

For example, we can measure the time required to create two random matrices.

```
>> tic
A = rand(12000,4400);
B = rand(12000,4400);
toc
Elapsed time is 0.491956 seconds.
```