

MotorXP-PM

Design and Analysis of Permanent Magnet Machines

Version 1.2

User Manual

Veeco Technologies Inc

February 2024

Contents

PART I. User Interface

1. INTRODUCTION	8
2. BRIEF THEORETICAL BACKGROUND.....	9
2.1. General description.....	9
2.2. Finite element mesh.....	11
2.3. Calculation of electromagnetic torque and force.	12
2.4. Multi-slice FEM.	15
2.5. D-Q model of a permanent magnet machine.....	16
2.6. Iron loss calculation.....	18
2.7. Demagnetization of permanent magnets.	18
2.8. Calculation of inverter power losses.	20
2.8.1. Dynamic D-Q analysis inverter power losses calculation (method 1).	20
2.8.2. Steady-state D-Q analysis inverter power losses calculation (method 2).....	23
3. GETTING STARTED	26
3.1. Using MotorXP-PM MATLAB version.	26
3.2. Using MotorXP-PM Standalone version.....	26
3.3. MotorXP-PM basics.....	27
3.4. Working with project files.....	29
3.4.1. Protected project files.	30
3.4.2. Import from MotorAnalysis-PM.....	31
3.5. Licensing options.	31
3.6. Activating the license.	32
4. CREATING DESIGN PROTOTYPES	36
4.1. Geometry Editor.....	36
4.1.1. Importing rotor geometry from DXF-file.....	40
4.1.2. Importing stator geometry from DXF-file.....	44
4.1.3. Using other tools of Geometry Editor.	47

4.2.	Assigning materials.....	48
4.2.1.	Adding new materials.....	49
4.3.	Winding Editor.....	53
4.4.	Mesh Editor.....	57
4.5.	Drive Settings.....	59
4.5.1.	Adding user defined transistors.....	62
5.	MAGNETOSTATIC FINITE ELEMENT ANALYSIS.....	78
5.1.	Running Magnetostatic FE Analysis.....	78
5.2.	Viewing Magnetostatic FE Analysis results.....	82
5.2.1.	Time plots.....	82
5.2.2.	Air gap distribution plots.....	86
5.2.3.	Cross-section distribution plots.....	89
5.2.4.	Animation.....	91
6.	STEADY STATE D-Q ANALYSIS.....	93
6.1.	Building a D-Q model.....	93
6.2.	Viewing Steady State D-Q Analysis results.....	93
6.3.	Efficiency map and other performance maps.....	100
6.4.	Extracting Id and Iq current components tables from the efficiency map data.....	103
6.5.	Extracting D-Q model parameters.....	103
7.	DYNAMIC D-Q ANALYSIS.....	106
7.1.	Running Dynamic D-Q Analysis.....	106
7.2.	Viewing Dynamic D-Q Analysis results.....	108
8.	DYNAMIC FINITE ELEMENT ANALYSIS	113
8.1.	Running Dynamic FE Analysis.....	113
8.2.	Viewing Dynamic FE Analysis results.....	117
8.2.1.	Time-averaged quantities.....	117
8.2.2.	Plot wizard.....	117
8.2.3.	Time plots.....	117
8.2.4.	Air gap distribution plots.....	121

8.2.5.	Cross-section distribution plots.	124
8.2.6.	Animation.	126
9.	DYNAMIC FE ANALYSIS SIMULATION SCRIPT FUNCTIONS	128
9.1.	General information.	128
9.2.	Writing a simulation script function.....	128
9.3.	Main data structures.	130
9.4.	Simple example of simulation script function.....	135
10.	USING ELECTRICAL CIRCUITS.....	138
10.1.	Writing an electrical circuit function.....	138
10.1.1.	Electrical circuit branches.....	139
10.1.2.	Adding circuit components.	139
10.2.	Controlling power sources and electronic switches using simulation script functions.	142
10.3.	Default three-phase inverter circuit function.....	143
11.	WRITING GEOMETRY SCRIPTS	147
11.1.	General information and geometry script structure.....	147
11.2.	Working with console.....	151
11.3.	Generating geometry.	152
11.3.1.	Adding line segments.....	152
11.3.2.	Adding arcs.	152
11.3.3.	Rounding corners.....	154
11.3.4.	<i>sDraw</i> objects and methods.	156
11.3.5.	Simple stator geometry script example using <i>sDraw</i> objects and methods.....	163
11.4.	Global object <i>motor</i>	165
11.5.	Assigning material types.....	166
11.5.1.	Assigning ‘General’ material type.	166
11.5.2.	Assigning ‘Iron’ material type.	166
11.5.3.	Assigning ‘Winding’ material type.....	167
11.5.4.	Assigning ‘Conductor’ material type.....	167
11.5.5.	Assigning ‘Magnet’ material type.....	167

11.6.	Files <i>.rotors.json</i> and <i>.stators.json</i> and geometry template attributes.	170
11.6.1.	Rotor geometry script example.	171
12.	MOTORXP-PM SETTINGS	176
APPENDIX		178
Appendix A. Power balance, discretization error and accuracy of the results.....		178
Appendix B. Magnetostatic Analysis results.		179
Appendix C. Out of memory errors handling.		180

PART II. MotorXP-PM MATLAB scripting API

1.	Scripting API initialization	183
2.	Functions for working with mxp-files	183
2.1.	<i>openMXP</i> function.....	183
2.2.	<i>saveDesign2MXPfile</i> function.....	184
3.	Functions for setting and getting parameters	184
3.1.	<i>setParampm</i> function	184
3.1.5.	Dependencies on json-files	185
3.1.6.	Stator and rotor geometry template json-files	185
3.2.	<i>motorProps</i> structure.....	186
3.2.1.	Main <i>paramName</i> parameter description and <i>paramValue</i> values.....	186
3.2.2.	Examples.....	190
3.2.3.	Some limitations for <i>paramValue</i> imposed by the machine topology.	190
3.2.4.	The <i>paramName</i> and <i>paramValue</i> variable values for default rotor and stator geometries.	190
3.3.	<i>settingsMagnetostatic</i> structure.....	197
3.3.1.	Main <i>paramName</i> parameter description and <i>paramValue</i> values.....	197
3.3.2.	Examples.....	199
3.4.	<i>getParampm</i> function.....	200
3.4.5.	Dependencies on json-files	201
3.4.6.	Examples.....	201
4.	Serial processing functions	202
4.1.	Assembling designs in series using the <i>assembleMXP</i> function.....	202

4.2.	Running magnetostatic FE simulations in series using the <i>runMStimestepping</i> function	203
4.3.	Parametric sweep script example	208
5.	Parallel processing functions	211
5.1.	Assembling designs in parallel.....	211
5.1.1.	<i>initAssembleMXP_par</i> function.....	211
5.1.2.	<i>runAssembleMXP_par</i> function.....	211
5.1.3.	<i>getAssembleMXP_par</i> function	211
5.2.	Running magnetostatic FE simulations in parallel.....	212
5.2.1.	<i>initMStimestepping_par</i> function	212
5.2.2.	<i>runMStimestepping_par</i> function	212
5.2.3.	<i>getMStimestepping_par</i> function.....	213
5.3.	Parametric sweep script example with parallel processing	214
6.	Automatic optimization workflows	217
6.1.	Overview	217
6.2.	Functions for automatic optimization workflow development	217
6.2.1.	Suggested optimization workflow structure	218
6.2.2.	<i>runTSEMOptimizationpm</i> function.....	218
6.2.3.	Design function (on an example of the <i>evalDesigns_SMPMSM</i> function)	220
6.2.4.	<i>plotParetoFromFile</i> function.....	221
6.3.	Example of the automatic optimization workflow implementation.....	222
6.4.	Example of the MATLAB code for automation of the motor and generator design	224
7.	MATLAB code debugging tips.....	231

PART I

User Interface

1. INTRODUCTION

Thank you for your interest in MotorXP-PM.

MotorXP-PM is software for design and analysis of permanent magnet machines including brushless DC and permanent magnet synchronous machines with surface-mounted and interior magnets. MotorXP-PM is based on automated finite element analysis (FEA) simulations combined with analytical methods and establishes a complete set of tools for design and analysis of permanent magnet machines. We believe that our work will help to save our environment supporting green technologies like electric vehicles or wind turbine energy generation as well as enhancing the efficiency and effectiveness of electric machines.

2. BRIEF THEORETICAL BACKGROUND

The purpose of this chapter is to give the user a brief description of the problem examined. This chapter contains the formulation of the problem, short description of the nonlinear solver organization, rotation of the finite element mesh, torque and force calculation and etc. This information is critical for proper application adjustment and getting desired results.

2.1. General description.

The magnetic field problems for a permanent magnet machine can be solved using a two-dimensional approximation, which is based on the assumption that the magnetic field does not depend on z -coordinate (z -axis being parallel to the axis of the rotor shaft). Thus, the magnetic field is solved in the plane of the machine's cross-section (x - y plane). The current density and magnetic vector potential in two-dimensional problems only have z -components and can be expressed as follows:

$$\nabla \times \left(\frac{1}{\mu_r} \cdot \nabla \times A \right) = J \quad (2.1)$$

$$J = \sigma \cdot \frac{U}{l} - \sigma \cdot \frac{\partial A}{\partial t}, \quad (2.2)$$

where A – magnetic vector potential ($B = \nabla \times A$, B – magnetic flux density), μ_r – relative magnetic permeability, J – current density, σ – conductivity, U – voltage applied to the FE area, l – length in z direction.

There are two methods used in MotorXP-PM to solve the problem defined by Exp. (2.1) and (2.2). First method is called a *magnetostatic finite element method* (FEM) which is used in the Magnetostatic Analysis. For the magnetostatic FEM the current density J is assumed to be predefined so only Exp. (2.1) is used to define the magnetostatic problem. Note that the magnetostatic FEM can only be used for analysis of steady-state regimes.

Another method used in MotorXP-PM is called a *transient finite element method* which is used in the Dynamic FE Analysis. According to Exp. (2.2) the current density consists of two components, the first one is caused by external voltage and the second one is induced by the varying magnetic field. The idea behind the transient FEM is a representation of the time derivative of the magnetic vector potential as follows:

$$\frac{\partial A}{\partial t} = \frac{A_n - A_{n-1}}{\Delta t},$$

where Δt – time step, A_n and A_{n-1} – magnetic vector potentials on n -th and $(n-1)$ -th time steps.

Determining the initial magnetic vector potentials with magnetostatic FEM and assuming zero initial stator currents the iterative procedure can be used to find the magnetic vector potentials for each time

step. This method allows to take into account induced eddy currents, nonlinearity of iron and the rotation of the rotor and can be used for analysis of both transient and steady-state regimes.

In both methods the discretization of the problem over the plane of the machine's cross-section using FEM results in the system of linear equations written in the matrix form as follows:

$$K \cdot X = F, \quad (2.3)$$

where K – stiffness matrix, X – vector of unknowns, F – right side vector of the problem. For the magnetostatic FEM the unknowns are the magnetic vector potential values in mesh nodes. For the transient FEM the vector of unknowns also includes values of stator currents and voltages.

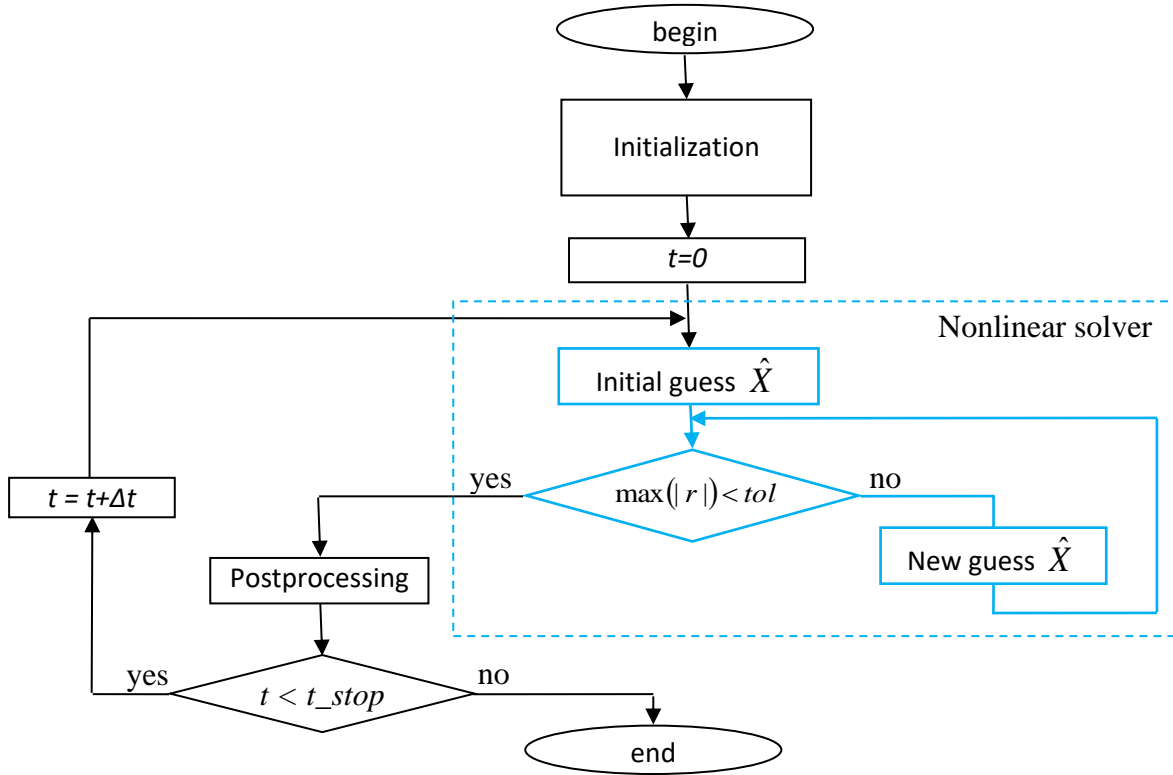


Figure 2.1. Generalized algorithm of solving a nonlinear problem with time-stepping FEM.

To solve the problem defined by Exp. (2.1) and (2.2) with a nonlinear B-H relationship the Gauss-Newton method is used. Assuming that we have a guess \hat{X} of the solution and stiffness matrix \hat{K} corresponding to the guess, then the residual vector of the guess \hat{X} will be defined as:

$$r = \hat{K} \cdot \hat{X} - F \quad (2.4)$$

Solving system (2.3) using Gauss-Newton iteration tends to be the minimization of the residual. The problem is said to be solved if the certain condition is satisfied. In MotorXP-PM this condition is defined by the maximum absolute value of the residual vector:

$$\max(|r|) < tol \quad (2.5)$$

where tol – convergence tolerance defining the desired accuracy of the solution.

Both methods described above use an iterative time-stepping procedure to solve a time-varying magnetic field problem. The generalized algorithm used in MotorXP-PM for a nonlinear case of the time stepping method is shown in Figure 2.1. An outer loop of the algorithm represents integration over time with step Δt . The magnetic field problem is solved on every integration step in the nonlinear solver loop (marked with blue color). Gauss-Newton iteration continues until the inequality (2.5) is satisfied.

2.2. Finite element mesh.

In MotorXP-PM the first-order triangular finite element mesh is used.

Every time while the time-stepping FE simulation is in progress, the rotor position is changed the mesh should be also changed. It is implemented by rotating the rotor mesh connected to the fixed stator mesh via a *sliding layer* located in the air gap. In MotorXP-PM the air gap always has an odd number of mesh layers (3, 5, 7 or 9) and the sliding layer is always located at the center of the air gap (see Figure 2.2).

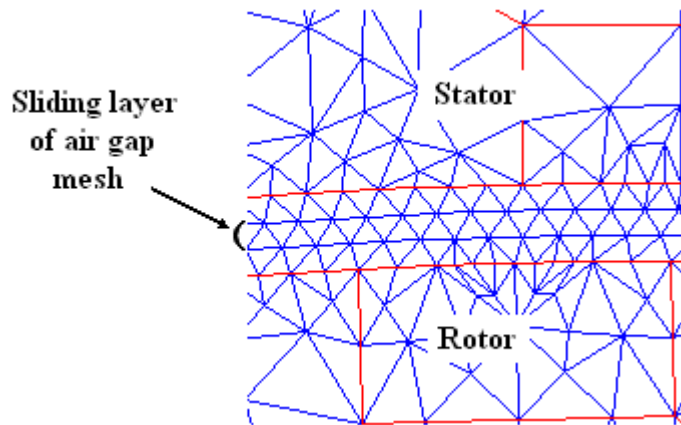


Figure 2.2. Three mesh layers in the air gap and the sliding layer at the center of the air gap.

To reduce the number of finite elements and thus to reduce the computation time, periodic/antiperiodic boundary conditions can be applied. Using this type of boundary conditions is based on periodicity of the magnetic field of the machine.

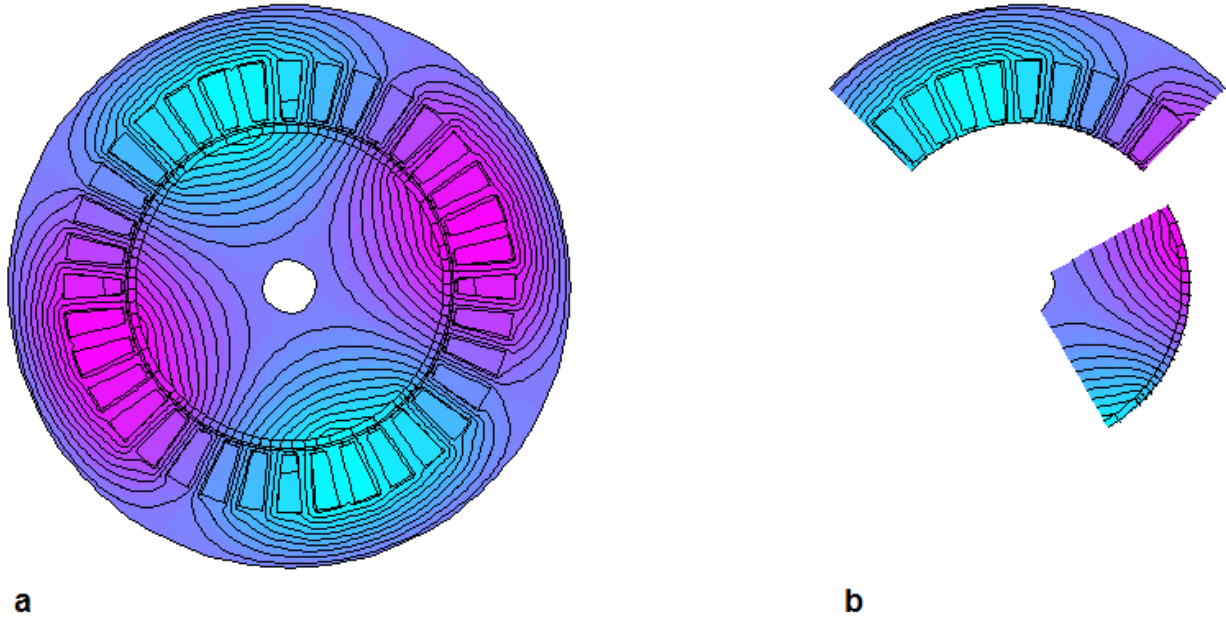


Figure 2.3. Using boundary conditions.

As it is seen from the example of four pole surface-mounted permanent magnet motor shown in Figure 2.3(a), the magnetic vector potentials repeat two times along the circle. Moreover, absolute values of the magnetic potential values repeat four times. Using this property of the field it is possible compute magnetic potentials for one quarter and then find the magnetic field for the whole cross-section as shown in Figure 2.3(b). In this case we use *antiperiodic* boundary conditions. If the magnetic potentials would be computed for the half, when *periodic* boundary conditions were applied. The number of periodicities of the magnetic field in MotorXP-PM is called *periodicity factor*. The periodicity factor basically equals to a number of pole pairs; in the presented example the periodicity factor is 2. Note that using periodic or antiperiodic boundary conditions is only possible if geometry of the cross-section repeats together with the field. In practice it means that to apply periodic boundary conditions the number of stator slots should be divided by number of pole pairs, while for antiperiodic boundary conditions the number of stator slots should be divided by number of poles.

2.3. Calculation of electromagnetic torque and force.

There are two methods used in MotorXP-PM for calculation of the torque. These are the Maxwell stress tensor method and virtual work method. For calculation of the radial force acting between the stator and rotor the virtual work method is used.

According to the Maxwell stress tensor method the electromagnetic torque for the inner rotor can be expressed as the following:

$$T = -\frac{l \cdot (D_r + l_\delta)^2}{4\mu_0} \cdot \int_0^{2\pi} B_n B_t d\phi \quad (2.6)$$

where T – electromagnetic torque, l – lamination length in z direction, D_r – rotor diameter, l_δ – air gap length, μ_0 – permeability of the free space, B_n и B_t – normal and tangential components of the magnetic flux density in the air gap, as shown in Figure 2.4. This expression is used in MotorXP-PM for calculation of the torque with the Maxwell stress tensor method.

The integration contour used in Exp. (2.6) always lies inside the sliding layer of the air gap mesh, as shown in Figure 2.4.

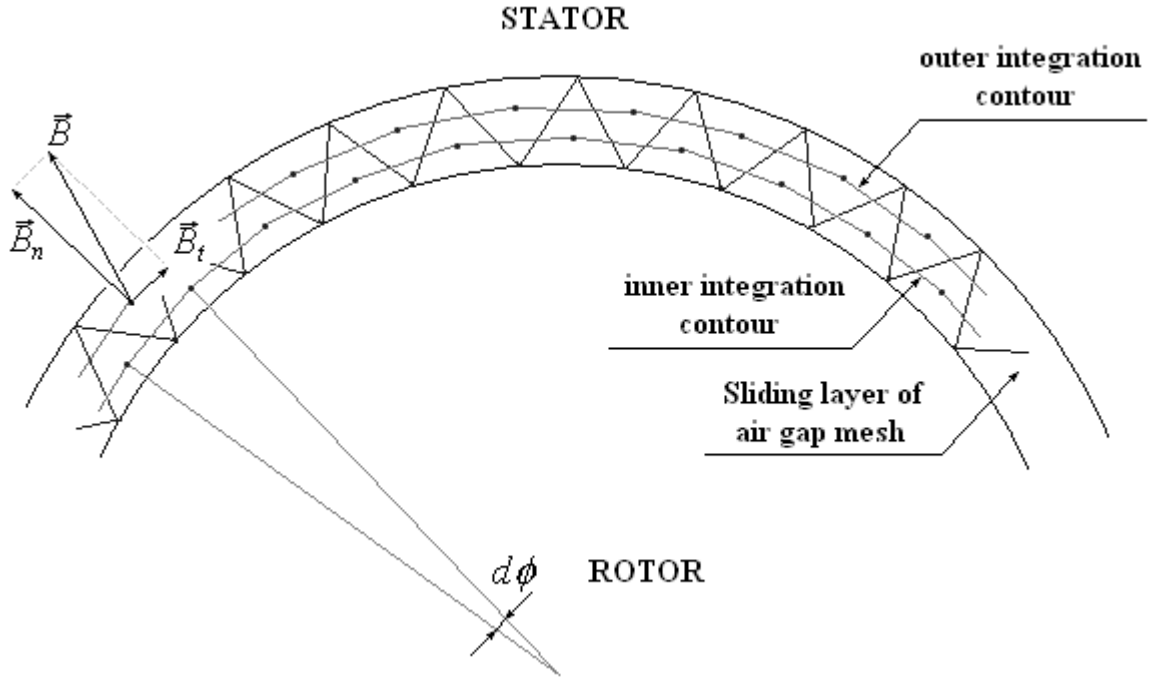


Figure 2.4. Calculation of the electromagnetic torque with the Maxwell stress tensor method.

Only the sliding layer of the air gap mesh is shown. Two integration contours are used - the inner contour passing through midpoints of the rotor-oriented triangles and the outer contour passing through midpoints of the stator-oriented triangles. The actual value of the torque is resulted as the average of two values calculated over the inner and outer integrated contours.

Source code for calculation of the electromagnetic torque with the Maxwell stress tensor method is presented in file *MaxwellTorque.m*.

According to the virtual work principle, the electromagnetic force is given by the derivation of the magnetic energy with respect to the virtual displacement with constant flux linkage:

$$F_u = - \left. \frac{\partial W}{\partial e} \right|_{\psi = \text{const}},$$

where ∂e - virtual displacement, ∂W - change of the magnetic energy between initial and final positions of the rotor.

Similarly, the expression for the torque calculation is defined as:

$$M = - \left. \frac{\partial W}{\partial \phi} \right|_{\psi = \text{const}},$$

where $\partial \phi$ - virtual angular displacement.

When the virtual work principle is applied for the torque and force calculation, the accuracy significantly depends on the choice of the virtual displacement value. On the one hand the displacement should be small enough not to distort the finite element mesh. On the other hand, the round-off error arises when the displacement value is too small. The optimal value of the virtual displacement is not provided automatically and should be defined by the user in menu **File -> Settings**.

It is considered that the virtual work method is more accurate comparing with the Maxwell stress tensor one since the first one implies the integration over the whole machine's cross-section while using the Maxwell stress tensor method involves only finite elements of the air gap. Though in most cases the difference between the two methods is not significant and the Maxwell stress tensor method is usually used.

When the time-stepping FEM is applied the calculated torque value is used to determine the current rotor position. The rotor rotation is defined as follows:

$$T = T_{load} + J \frac{d\omega}{dt}$$

$$\omega = \frac{d\varphi}{dt},$$

where T_{load} – load torque on the motor shaft, J – combined rotor and load moment of inertia, ω - rotor angular speed, φ – rotor angular position.

2.4. Multi-slice FEM.

As it was previously assumed the magnetic field of a permanent magnet machine does not depend on z -coordinate. In reality this assumption is not quite correct because of the end-winding leakage flux and rotor or stator skewing. In 2D FEM the end-winding leakage flux can be taken into consideration with satisfactory accuracy using end-winding inductance. But the two-dimensional approximation is of no help when it comes to skewed geometries. Rotor and stator skewing can be accurately simulated with 3D FEM. But these simulations are usually long. As an alternative, multi-slice FEM is used in MotorXP-PM. Figure 2.5 illustrates the multi-slice approach.

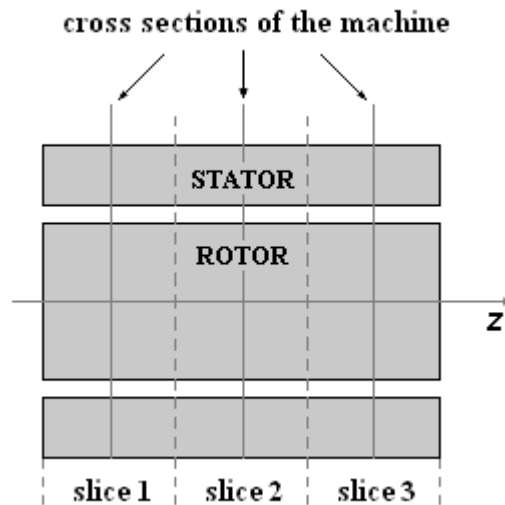


Figure 2.5. The multi-slice FEM principle.

The multi-slice FEM principle consists in dividing the machine along the z -coordinate into several slices. The cross-section geometry changes from one slice to another according to the applied skew angle. Then the magnetic field problem is solved in every slice all at the same time with 2D FEM. The electromagnetic torque is calculated for every slice and the actual torque is obtained as a summation of all slice values. In MotorXP-PM the multi-slice approach is used for both magnetostatic FEA and transient FEA (Dynamic FE Analysis).

Note that the calculation time is proportional to the number of slices. Set the number of slices to one to get the 2D FEM simulation. If there is no rotor or stator skewing, using several slices is not reasonable, since the result will be the same as with one slice used.

2.5. D-Q model of a permanent magnet machine.

The dq-axes reference frame is a convenient way to represent sinusoidal quantities as constants. The dq-axes reference frame is fixed to the rotor and turning with the same speed as the rotor. As shown in Figure 2.6 on the example of a two-pole surface-mounted permanent magnet rotor, the d-axis is always lying in the direction of the magnetic field of the permanent magnets. The q-axis is turned by 90 electrical degrees, i.e. it always lies between two magnetic poles of the machine.

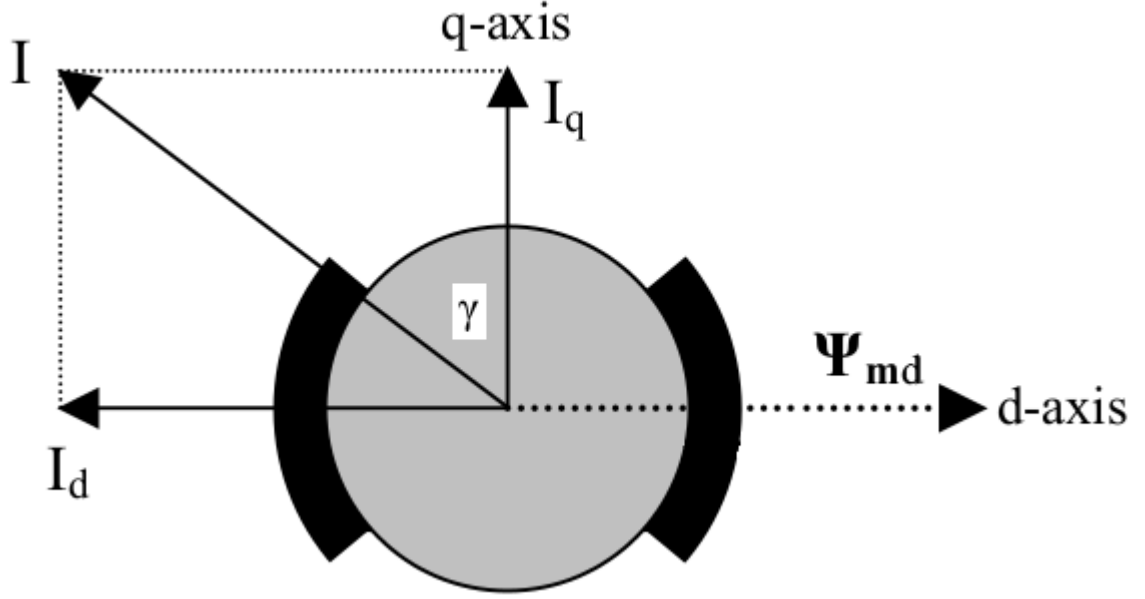


Figure 2.6. Definition of dq-axes reference frame in permanent magnet machines. ¹

As shown in Figure 2.6, the q-axis current I_q creates a magnetic field in the q-axis that interacts with the magnetic field of the permanent magnets Ψ_{md} lying in the d-axis to generate torque. The d-axis current I_d creates a magnetic field in the d-axis, which can be used to weaken the magnetic field of the permanent magnets when chosen negative (field weakening operation above base speed).

As shown in Figure 2.6, the current vector is defined by its peak value I and the advance angle γ towards the q-axis. The relationship between the amplitude and advance angle of the current vector and the dq-axes components is given by the following equations:

$$\begin{aligned}
 I_q &= I \cos \gamma \\
 I_d &= -I \sin \gamma \\
 I &= \sqrt{I_d^2 + I_q^2} \\
 \gamma &= \arctan \left(-\frac{I_d}{I_q} \right)
 \end{aligned} \tag{2.7}$$

¹ <https://www.emotor.com/blog/post/blac-machine-simulations/>

To take into account the effect of cross-saturation the cross-saturation inductance L_{dq} and cross saturation magnet flux linkage Ψ_{mqd} lying in the q-axis are included. With the cross-saturation terms, the d-axis and q-axis flux linkages are given as follows:

$$\begin{aligned}\Psi_d &= \Psi_{md} + L_d I_d + L_{dq} I_q \\ \Psi_q &= \Psi_{mqd} + L_q I_q + L_{dq} I_d\end{aligned}\tag{2.8}$$

Where L_d , L_q , L_{dq} , Ψ_{md} and Ψ_{mqd} values depend on I_d and I_q to take into account saturation.

The steady state equations after resolving voltages into d and q components can be written in the general form:

$$\begin{aligned}V_d &= R_s I_d - \omega \Psi_q - \omega L_{sew} I_q \\ V_q &= R_s I_q + \omega \Psi_d + \omega L_{sew} I_d \\ V &= \sqrt{V_d^2 + V_q^2}\end{aligned}\tag{2.9}$$

The electromagnetic torque is calculated using the well-known equation:

$$T = \frac{3}{2} p (\Psi_d I_q - \Psi_q I_d)\tag{2.10}$$

Where R_s – stator winding phase resistance, L_{sew} – stator end winding inductance, ω – electrical operating speed, p – number of pole pairs.

The calculation of D-Q model parameters L_d , L_q , L_{dq} , Ψ_{md} and Ψ_{mqd} in MotorXP-PM is based on finite element method with «permeance freezing» which allows using superposition to extract individual parameters of the machine under load and preserve information about saturation while taking into account the effect of cross-saturation as well.²

And finally, the dynamic equations written for the voltage components are given by the following expressions:

$$\begin{aligned}V_d &= \frac{d\Psi_d}{dt} + L_{sew} \frac{dI_d}{dt} + R_s I_d - \omega \Psi_q - \omega L_{sew} I_q \\ V_q &= \frac{d\Psi_q}{dt} + L_{sew} \frac{dI_q}{dt} + R_s I_q + \omega \Psi_d + \omega L_{sew} I_d\end{aligned}\tag{2.11}$$

² Žarko, Damir, Ban, Drago, Klarić, Ratko. (2006). Finite Element Approach to Calculation of Parameters of an Interior Permanent Magnet Motor. AUTOMATIKA: Journal for Control, Measurement, Electronics, Computing and Communications; Vol.46, No.3-4.

2.6. Iron loss calculation

Iron core losses in MotorXP-PM are calculated during the post processing so it assumes that the iron losses do not influence the magnetic field distribution. There is a well-known expression for the iron loss estimation based on the Steinmetz equation:

$$P = K_h f^\alpha B_m^\beta + K_e (f \cdot B_m)^2 \quad (2.12)$$

First term in Exp. 2.12 represents hysteresis losses, the second one – eddy current loss term. Coefficients K_h , α , β and K_e are determined by fitting equation 2.12 to the measured iron loss data from the manufacturer at different frequencies.

Exp. 2.12 allows to estimate the iron losses at a single frequency, which is usually a supply frequency (fundamental harmonic iron losses). Estimation of the iron losses for other flux density harmonics with Exp. 2.12 would require FFT procedure, which is associated with long simulation times to provide sufficient frequency resolution. Instead, MotorXP-PM uses more advanced dynamic iron loss model³. This model uses the same coefficients as in Exp. 2.12 and takes into consideration the non-sinusoidal shape of the flux density waveform without using FFT. It allows one to include into analysis the iron loss contribution from higher harmonics of the flux density such as slot harmonics and PWM produced harmonics for both stator and rotor cores as well as minor hysteresis loops.

The same iron loss model is used for both magnetostatic FEA and transient FEA (Dynamic FE Analysis). Note that D-Q analysis in MotorXP-PM takes into account only the fundamental harmonic iron losses.

2.7. Demagnetization of permanent magnets.

Figure 2.9 shows an example of demagnetization curves for permanent magnet material N52. Irreversible demagnetization occurs if the working point in any part of the permanent magnet exceeds the linear range, i.e. falls below the "knee" of the demagnetization curve, as shown in Figure 2.9. It means that the magnet irreversibly changes its properties (remanence flux density is irreversibly reduced) and the new demagnetization curve shown by dotted line emerges.

An important parameter which characterizes the ability of permanent magnet to resist the irreversible demagnetization is intrinsic coercivity H_{cj} . Intrinsic coercivity is the applied demagnetizing field required to fully demagnetize the permanent magnet material so when the demagnetizing field is removed the material does not act like a magnet anymore. The higher intrinsic coercivity the higher demagnetizing field can be sustained by the permanent magnet without risk of demagnetization.

It is a good practice to always check your simulations for the possible risk of demagnetization at worst operating conditions (maximum current, advance angle and temperature of permanent magnet) including

³ D. Lin, P. Zhou, W. N. Fu, Z. Badics and Z. J. Cendes, "A dynamic core loss model for soft ferromagnetic and power ferrite materials in transient finite element analysis," in IEEE Transactions on Magnetics, vol. 40, no. 2, pp. 1318-1321, March 2004.

possible fault conditions. In MotorXP-PM the risk of demagnetization can be estimated from the maximum demagnetizing field intensity experienced by the permanent magnet during simulation. The maximum demagnetizing field intensity is available measured in A/m or in percentage of intrinsic coercivity H_{cj} . If the demagnetization curve for the given permanent magnet material is available, make sure that the maximum demagnetizing field value is to the right of the "knee" of the demagnetization curve. For neodymium magnets there is usually no risk of demagnetization if the maximum demagnetizing field is less than 70% of intrinsic coercivity.

As seen from Figure 2.9 the demagnetization curve changes significantly with the temperature and the risk of demagnetization increases at higher temperatures of the permanent magnet. MotorXP-PM varies properties of the permanent magnet depending on the temperature so to better estimate the risk of demagnetization it is recommended to check simulation results with the maximum expected temperature of the permanent magnet. You should also keep in mind that the material data and demagnetization curves represent typical properties that may vary due to magnet shape and size with tolerances of up to $\pm 10\%$.

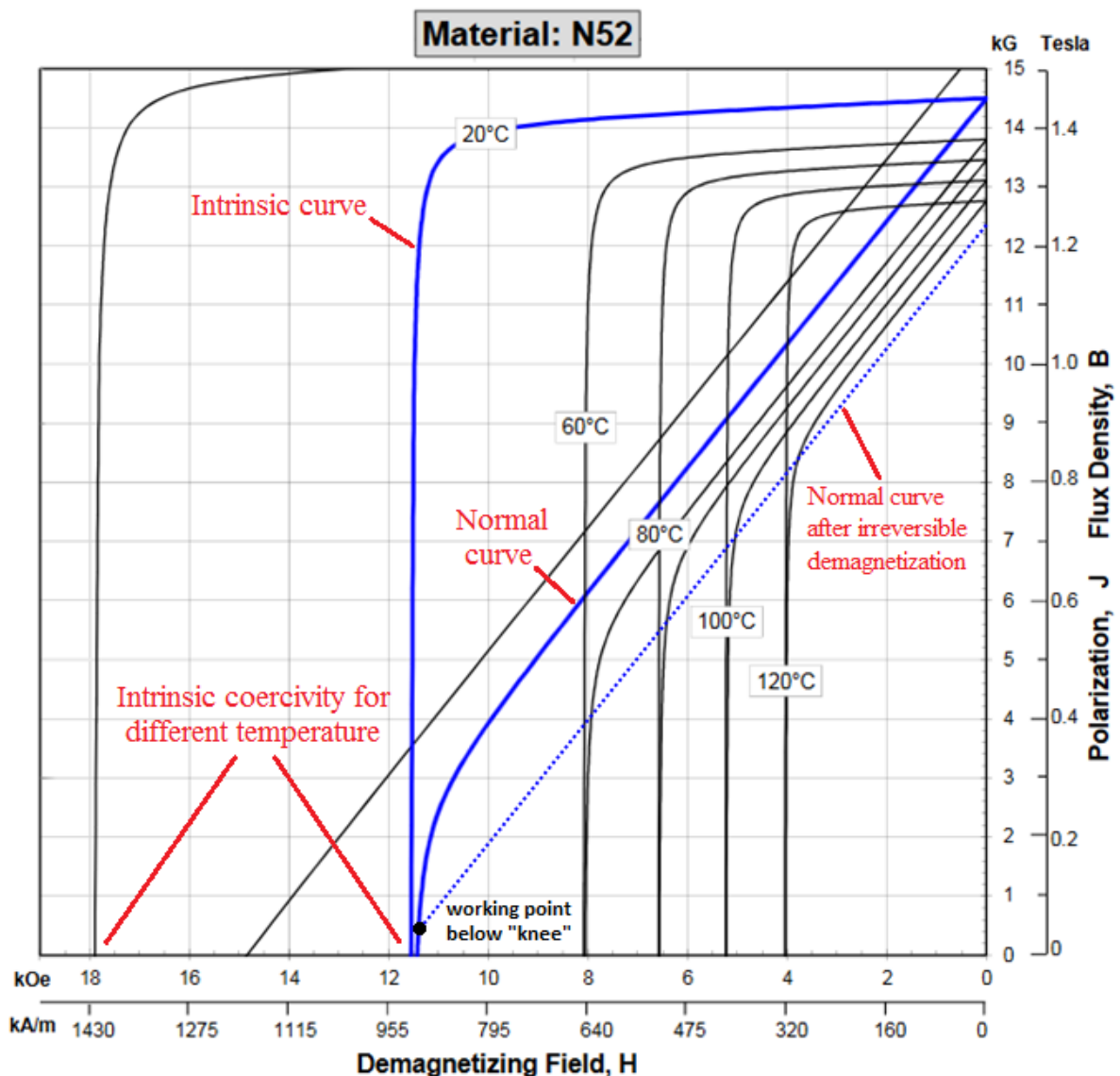


Figure 2.9. Demagnetization curves for permanent magnet material N52.

2.8. Calculation of inverter power losses.

Power losses in any semiconductor component, such as IGBT and MOSFET transistor or diode, operating in the switch-mode can be divided into conduction losses and switching losses. Conduction losses occur during on-state operation because of the finite value of the on-state resistance. Switching losses occur when the semiconductor component changes its state (from on-state to off-state or from off-state to on-state) because of finite switching time.

There are two methods in MotorXP-PM used for the inverter power losses calculation⁴. The first method is used with **Dynamic D-Q Analysis** (see chapter 7) simulation and consists of calculating the instantaneous conduction and switching losses for every transistor and diode of the inverter for every time step and every switching event. This method is more accurate, but the computational speed is limited by the speed of the dynamic D-Q simulation. The second method is used with **Steady State D-Q Analysis** (see chapter 6) only for the space-vector PWM supply and based on the analytical expressions with the use of the peak inverter output current value. This method is very fast, but the accuracy is less, and the applicability is limited only by the space-vector PWM power supply.

Refer to section 4.5.1 to learn which parameters of the transistor are required for the inverter power losses calculation and how they can be extracted from the datasheet data.

2.8.1. Dynamic D-Q analysis inverter power losses calculation (method 1).

This method is used with **Dynamic D-Q Analysis** and suitable for all power supply types. The inverter power loss calculation is considered for two cases when transistors of IGBT and MOSFET type are used.

2.8.1.1. Transistor of MOSFET type.

Instantaneous conduction power losses of the MOSFET:

$$P_c^{MOSFET} = R_{DSon}(i_D) \cdot i_D^2,$$

where R_{DSon} – drain-to-source on-state resistance, i_D – instantaneous drain current.

Instantaneous conduction power losses of the freewheeling diode:

$$P_c^{diode} = v_f \cdot i_f$$

$$v_f = V_{f0} + R_f(i_f) \cdot i_f,$$

where v_f – instantaneous voltage across the diode, i_f – instantaneous diode current, R_f – diode on-state resistance, V_{f0} – diode on-state zero-current voltage.

⁴ D. Graovac, M. Pürschel, "IGBT Power Losses Calculation Using the Data-Sheet Parameters", Application Note – Infineon, Jan. 2009.

D. Graovac, M. Pürschel, A. Kiep, "MOSFET Power Losses Calculation Using the Data-Sheet Parameters", Application Note – Infineon, July 2006.

Instantaneous off-state to on-state switching energy losses:

$$E_{on}^{MOSFET} = V_{DC} \cdot i_{Don} \cdot \frac{t_{ri} + t_{fu}}{2} + Q_{rr} \cdot V_{DC}$$

$$E_{on}^{diode} = \frac{1}{4} \cdot Q_{rr} \cdot V_{DC},$$

where V_{DC} – DC supply voltage, i_{Don} – drain current after the switching event, t_{ri} – drain current rise time from the datasheet, t_{fu} – voltage fall time, Q_{rr} - diode reverse recovery charge from the datasheet.

The voltage fall time is defined as follows:

$$t_{fu} = \frac{t_{fu1} + t_{fu2}}{2}$$

$$t_{fu1} = (V_{DC} - R_{DSon}(i_D) \cdot i_{Don}) \cdot R_{Dr} \cdot \frac{C_{GD1}}{V_{Dr} - V_{plateau}}$$

$$t_{fu2} = (V_{DC} - R_{DSon}(i_D) \cdot i_{Don}) \cdot R_{Dr} \cdot \frac{C_{GD2}}{V_{Dr} - V_{plateau}}$$

$$C_{GD1} = C_{rss}(V_{DS} = V_{DC})$$

$$C_{GD2} = C_{rss}(V_{DS} = R_{DSon}(i_D) \cdot i_{Don}),$$

where R_{Dr} – gate driver circuit resistance, V_{Dr} – gate driver circuit output voltage, $V_{plateau}$ – gate Muller plateau voltage from the datasheet, C_{GD1} and C_{GD2} - reverse transfer (gate-to-drain) capacitance vs. drain-to-source voltage $C_{rss} = f(V_{DS})$ at different V_{DS} levels.

Instantaneous on-state to off-state switching energy loss:

$$E_{off}^{MOSFET} = V_{DC} \cdot i_{Doff} \cdot \frac{t_{ru} + t_{fi}}{2}$$

where i_{Doff} – drain current before the switching event, t_{fi} – drain current fall time from the datasheet, t_{ru} – voltage rise time. The switch-off losses in the diode are neglected ($E_{off}^{diode} = 0$).

The voltage rise time is defined as follows:

$$t_{ru} = \frac{t_{ru1} + t_{ru2}}{2}$$

$$t_{ru1} = (V_{DC} - R_{DSon}(i_D) \cdot i_{Doff}) \cdot R_{Dr} \cdot \frac{C_{GD1}}{V_{plateau}}$$

$$t_{ru2} = (V_{DC} - R_{DSon}(i_D) \cdot i_{Doff}) \cdot R_{Dr} \cdot \frac{C_{GD2}}{V_{plateau}}$$

$$C_{GD1} = C_{rss}(V_{DS} = V_{DC})$$

$$C_{GD2} = C_{rss}(V_{DS} = R_{DSon}(i_D) \cdot i_{Doff}),$$

where C_{GD1} and C_{GD2} - reverse transfer (gate-to-drain) capacitance vs. drain-to-source voltage $C_{rss} = f(V_{DS})$ at different V_{DS} levels.

The instantaneous switching power losses are calculated as a sum of the switching energy losses over all switching events divided by the considered time interval t :

$$P_{sw}^{MOSFET} = \frac{\Sigma(E_{on}^{MOSFET}) + \Sigma(E_{off}^{MOSFET})}{t}$$

$$P_{sw}^{diode} = \frac{\Sigma(E_{on}^{diode})}{t}$$

2.8.1.2. Transistor of IGBT type.

Instantaneous conduction power losses of the IGBT:

$$P_c^{IGBT} = v_{CE}(i_c) \cdot i_c$$

$$v_{CE} = v_{CE0}(i_c) + R_C(i_c) \cdot i_c,$$

where v_{CE} – instantaneous collector-emitter on-state voltage, i_c – instantaneous collector current, v_{CE0} – instantaneous zero-current collector-emitter on-state voltage, R_C – collector-emitter on-state resistance. Instantaneous conduction power losses of the freewheeling diode:

$$P_c^{diode} = v_f(i_f) \cdot i_f$$

$$v_f = v_{f0}(i_f) + R_f(i_f) \cdot i_f,$$

where v_f – instantaneous voltage across the diode, i_f – instantaneous diode current, R_f – diode on-state resistance, v_{f0} – diode on-state zero-current voltage.

Instantaneous turn-on and turn-off IGBT switching energy losses E_{on}^{IGBT} and E_{off}^{IGBT} are calculated from the corresponding turn-on and turn-off energy vs. collector current diagrams from the datasheet (see section 4.5.1).

Instantaneous diode turn-on switching energy losses consist mostly of the reverse-recovery energy:

$$E_{on}^{diode} = \frac{1}{4} \cdot Q_{rr} \cdot V_{DC},$$

where V_{DC} – DC supply voltage, Q_{rr} – diode reverse recovery charge from the datasheet.

The turn-off losses in the diode are neglected ($E_{off}^{diode} = 0$).

The instantaneous switching power losses are calculated as a sum of the switching energy losses over all switching events divided by the considered time interval t :

$$P_{sw}^{IGBT} = \frac{\Sigma(E_{on}^{IGBT}) + \Sigma(E_{off}^{IGBT})}{t}$$

$$P_{sw}^{diode} = \frac{\Sigma(E_{on}^{diode})}{t}$$

2.8.2. Steady-state D-Q analysis inverter power losses calculation (method 2).

This method is used with **Steady State D-Q Analysis** and suitable only for the space-vector PWM power supply. The inverter power losses calculation is considered for two cases when transistors of IGBT and MOSFET type are used.

2.8.2.1. Transistor of MOSFET type.

It is assumed that the freewheeling diodes do not open with the space-vector PWM since MOSFET conducts current in both directions and the on-state voltage across the MOSFET is very low (much less than the diode on-state zero-current voltage), so only MOSFET losses are taken into account.

Forward direction conduction power losses of the MOSFET:

$$P_{CF}^{MOSFET} = R_{DSon} \cdot I_{Om}^2 \cdot \left(\frac{1}{8} + \frac{m_a \cdot PF}{3 \cdot \pi} \right)$$

Reverse direction conduction power losses of the MOSFET:

$$P_{CR}^{MOSFET} = R_{DSon} \cdot I_{Om}^2 \cdot \left(\frac{1}{8} - \frac{m_a \cdot PF}{3 \cdot \pi} \right),$$

where R_{DSon} – drain-to-source on-state resistance, I_{Om} – peak inverter output current, m_a – inverter amplitude modulation index (0 - 0.8660 for space-vector PWM), PF – motor power factor.

Off-state to on-state switching energy losses:

$$E_{on}^{MOSFET} = V_{DC} \cdot \left(\frac{I_{Om}}{\pi} \right) \cdot \frac{t_{ri} + t_{fu}}{2} + Q_{rr} \cdot V_{DC},$$

where V_{DC} – DC supply voltage, t_{ri} – drain current rise time from the datasheet, t_{fu} – voltage fall time, Q_{rr} – diode reverse recovery charge from the datasheet.

The voltage fall time is defined as follows:

$$\begin{aligned} t_{fu} &= \frac{t_{fu1} + t_{fu2}}{2} \\ t_{fu1} &= \left(V_{DC} - R_{DSon} \cdot \left(\frac{I_{Om}}{\pi} \right) \right) \cdot R_{Dr} \cdot \frac{C_{GD1}}{V_{Dr} - V_{plateau}} \\ t_{fu2} &= \left(V_{DC} - R_{DSon} \cdot \left(\frac{I_{Om}}{\pi} \right) \right) \cdot R_{Dr} \cdot \frac{C_{GD2}}{V_{Dr} - V_{plateau}} \\ C_{GD1} &= C_{rss}(V_{DS} = V_{DC}) \\ C_{GD2} &= C_{rss} \left(V_{DS} = R_{DSon} \cdot \left(\frac{I_{Om}}{\pi} \right) \right), \end{aligned}$$

where R_{Dr} – gate driver circuit resistance, V_{Dr} – gate driver circuit output voltage, $V_{plateau}$ – gate Muller plateau voltage from the datasheet, C_{GD1} and C_{GD2} – reverse transfer (gate-to-drain) capacitance vs. drain-to-source voltage $C_{rss} = f(V_{DS})$ at different V_{DS} levels.

On-state to off-state switching energy losses:

$$E_{off}^{MOSFET} = V_{DC} \cdot \left(\frac{I_{Om}}{\pi} \right) \cdot \frac{t_{ru} + t_{fi}}{2}$$

where t_{fi} – drain current fall time from the datasheet, t_{ru} – voltage rise time.

The voltage rise time is defined as follows:

$$\begin{aligned} t_{ru} &= \frac{t_{ru1} + t_{ru2}}{2} \\ t_{ru1} &= \left(V_{DC} - R_{DSon} \cdot \left(\frac{I_{Om}}{\pi} \right) \right) \cdot R_{Dr} \cdot \frac{C_{GD1}}{V_{plateau}} \\ t_{ru2} &= \left(V_{DC} - R_{DSon} \cdot \left(\frac{I_{Om}}{\pi} \right) \right) \cdot R_{Dr} \cdot \frac{C_{GD2}}{V_{plateau}} \\ C_{GD1} &= C_{rss}(V_{DS} = V_{DC}) \\ C_{GD2} &= C_{rss} \left(V_{DS} = R_{DSon} \cdot \left(\frac{I_{Om}}{\pi} \right) \right), \end{aligned}$$

where C_{GD1} and C_{GD2} – reverse transfer (gate-to-drain) capacitance vs. drain-to-source voltage $C_{rss} = f(V_{DS})$ at different V_{DS} levels.

The switching losses are the product of switching energies and the switching frequency (f_s):

$$P_{sw}^{MOSFET} = 2 \cdot (E_{on}^{MOSFET} + E_{off}^{MOSFET}) \cdot f_s$$

2.8.2.2. Transistor of IGBT type.

Conduction power losses of the IGBT:

$$P_C^{IGBT} = V_{CE0} \cdot I_{Om} \cdot \left(\frac{1}{2\pi} + \frac{m_a \cdot PF}{8} \right) + R_C \cdot I_{Om}^2 \cdot \left(\frac{1}{8} + \frac{m_a \cdot PF}{3\pi} \right)$$

Conduction power losses of the freewheeling diode:

$$P_C^{diode} = V_{f0} \cdot I_{Om} \cdot \left(\frac{1}{2\pi} - \frac{m_a \cdot PF}{8} \right) + R_f \cdot I_{Om}^2 \cdot \left(\frac{1}{8} - \frac{m_a \cdot PF}{3\pi} \right),$$

where V_{CE0} – zero-current collector-emitter on-state voltage, R_C – collector-emitter on-state resistance, V_{f0} – diode on-state zero-current voltage, R_f – diode on-state resistance, I_{Om} – peak inverter output current, m_a – inverter amplitude modulation index (0 - 0.8660 for space-vector PWM), PF – motor power factor. Turn-on and turn-off IGBT switching energy losses E_{on}^{IGBT} and E_{off}^{IGBT} are calculated from the corresponding turn-on and turn-off energy vs. collector current diagrams from the datasheet (see section 4.5.1).

Diode turn-on switching energy losses consist mostly of the reverse-recovery energy:

$$E_{on}^{diode} = \frac{1}{4} \cdot Q_{rr} \cdot V_{DC},$$

where V_{DC} – DC supply voltage, Q_{rr} – diode reverse recovery charge from the datasheet.

The turn-off losses in the diode are neglected ($E_{off}^{diode} = 0$).

The switching losses are the product of switching energies and the switching frequency (f_s):

$$P_{sw}^{IGBT} = (E_{on}^{IGBT} + E_{off}^{IGBT} + E_{on}^{diode}) \cdot f_s$$

3. GETTING STARTED

3.1. Using MotorXP-PM MATLAB version.

If you do not have MATLAB® refer to the next section to learn how to get started with MotorXP-PM Standalone version.

MotorXP-PM MATLAB version is a MATLAB-application and to use it you need to have MATLAB® installed on your computer. MotorXP-PM was currently tested with MATLAB versions between MATLAB R2017a and later.

You do not need to install the MATLAB version of MotorXP-PM – just copy all the files on your hard drive. To start MotorXP-PM enter `motorxp` in the MATLAB Command Window. Note that the MATLAB Current Folder should be changed to the folder with the application files. MotorXP-PM main window shown in Figure 3.1 will appear.

3.2. Using MotorXP-PM Standalone version.

MotorXP-PM Standalone version works like any Windows program and does not require MATLAB®. Use installation file *MotorXPPMv1.2Installer.exe* to install MotorXP-PM Standalone version. Internet connection is required during installation since the installation package does not include MATLAB Compiler Runtime components.

If you receive a message stating "Error finding installer class" when trying to install MotorXP-PM Standalone version, most likely, illegal characters are present in the path to the installation folder, or in the path to the Windows TEMP folder, or in the path to the folder containing the MotorXP-PM installer. Visit <https://www.mathworks.com/matlabcentral/answers/92499-why-do-i-receive-a-message-stating-error-finding-installer-class-when-trying-to-install-matlab-on> for more details.

Important notice: *MotorXP-PM Standalone version has several limitations* comparing with MotorXP-PM MATLAB version. These limitations are related to the fact that the MCR application is not able to execute m-files which were not included while the application was compiled. Note that several m-files are provided together with the program. These m-files were wrapped into `motorxp.exe` file while the application was being compiled. If you change any of these m-files or add your own m-file it will not have any effect.

Limitations of MotorXP-PM Standalone version:

1. Dynamic FE Analysis simulation script functions (see chapter 8 and 9) cannot be used except those provided together with the program (*simscript_hystpwm.m*, *simscript_spacevecpwm.m*, *simscript_sixstep.m*);

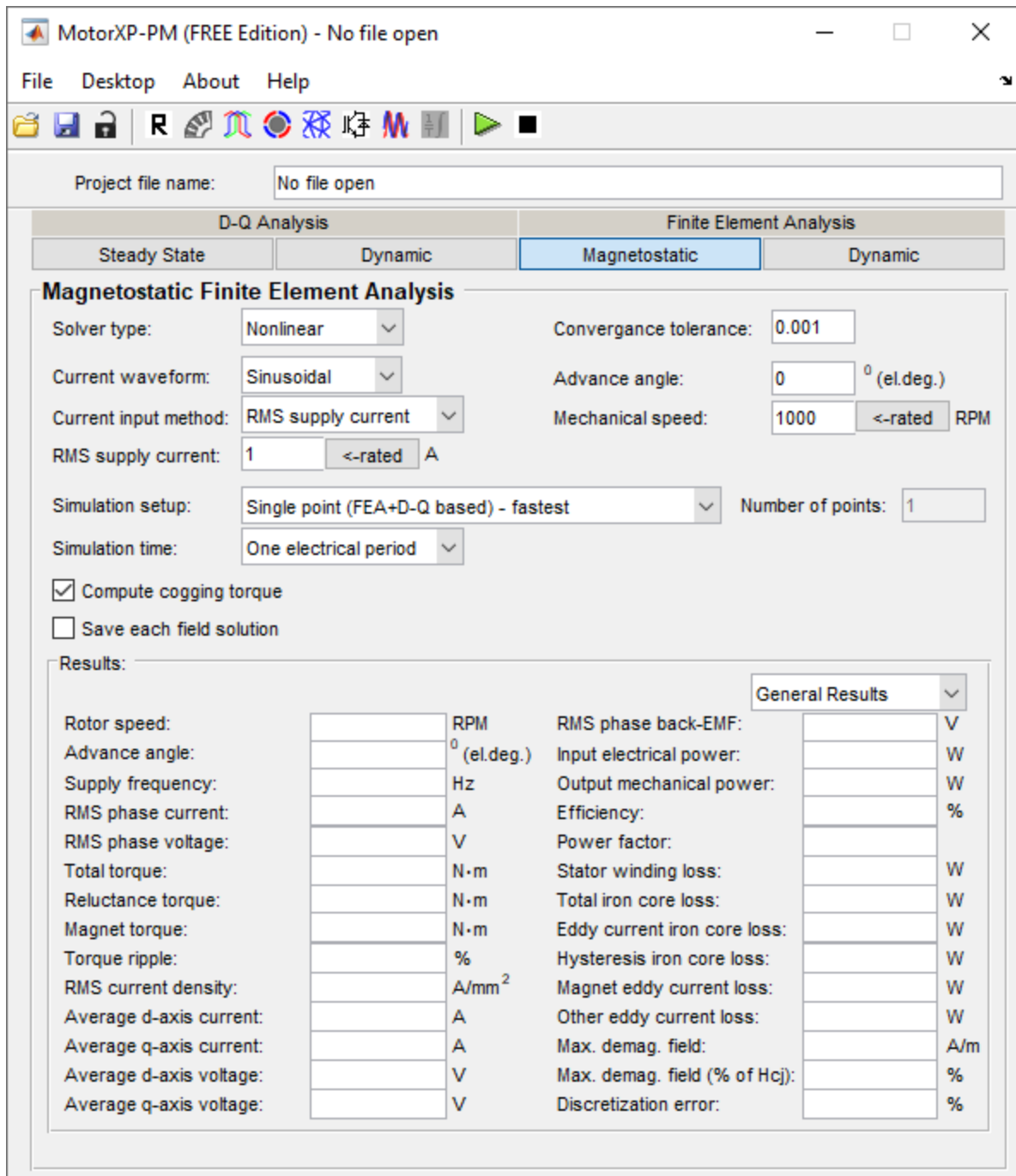


Figure 3.1. MotorXP-PM main window.

2. Electrical circuit functions (see chapter 10) cannot be used except those provided together with the program (*StarConnection.m*, *DeltaConnection.m*, *InverterCircuit.m*);

3. Parametric analysis and automatic optimization API is not available in the Standalone version.

If you use MotorXP-PM Standalone version, keep in mind these limitations while reading this manual since some features described in the manual are available only in MotorXP-PM MATLAB version.

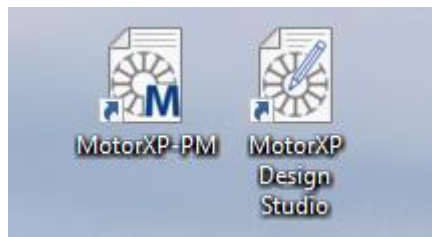
3.3. MotorXP-PM basics.

When you start MotorXP-PM, the main window appears as shown in Figure 3.1.

Using toolbar buttons, you can get access to all MotorXP-PM tools to create and analyze the design prototypes such as MotorXP Design Studio (consisting of **Geometry Editor / Materials, Winding Editor** and **Mesh Editor** tabs), **Drive Settings**, **Rated Data** and **Plot Wizard**. These are also available from the **Desktop** menu. Next chapters provide the detailed information on using these tools.

There are four analysis types available in MotorXP-PM: **Magnetostatic Finite Element Analysis**, **Steady State D-Q Analysis**, **Dynamic D-Q Analysis** and **Dynamic Finite Element Analysis**. To choose the analysis type, use the corresponding button under the toolbar. Table 3.1 presents a summary of MotorXP-PM analysis types with a short description of each analysis type. Refer to the corresponding chapter of this manual to learn more about each analysis type.

After installation of MotorXP-PM Standalone version, two shortcuts appear on the desktop for MotorXP-PM and for MotorXP Design Studio:



For MotorXP-PM MATLAB version the MotorXP Design Studio shortcut can be found in the folder with the application files.

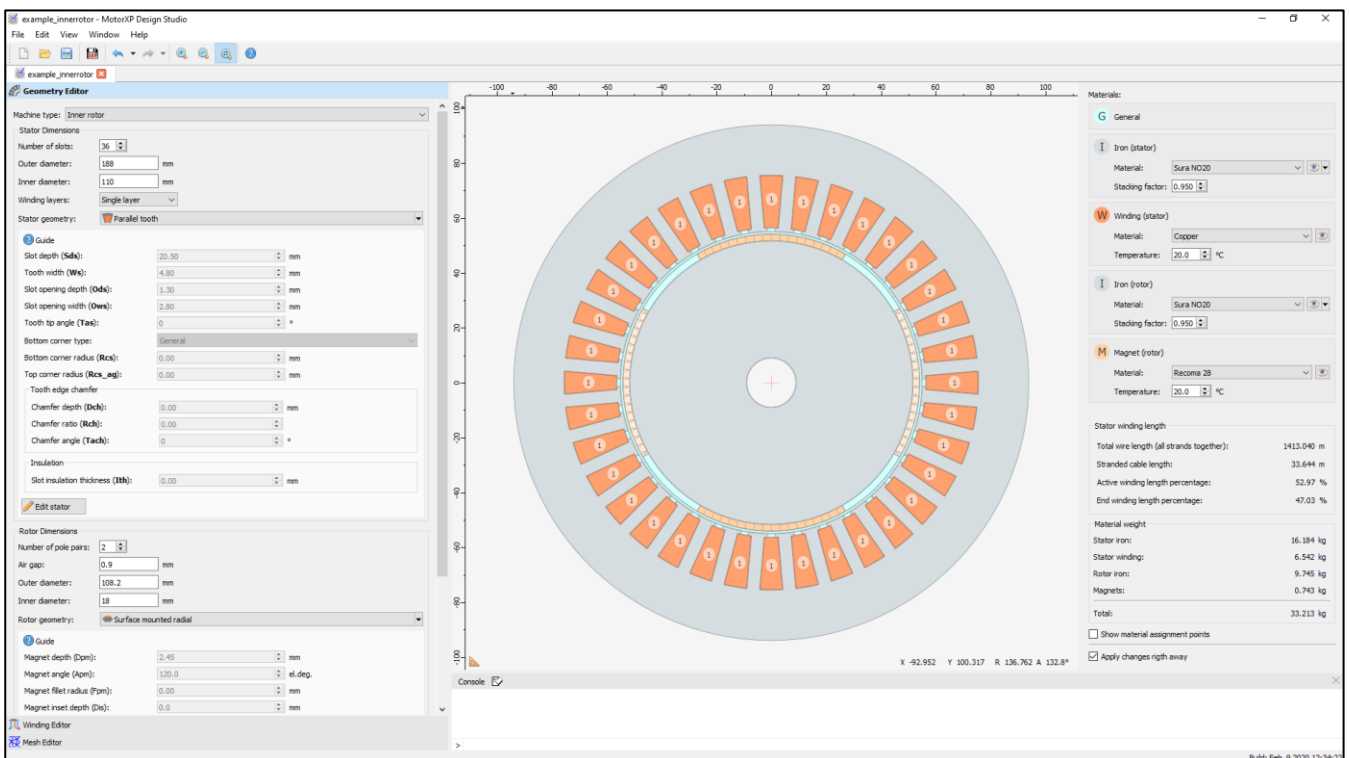



Figure 3.2. MotorXP Design Studio standalone.

MotorXP Design Studio is a tool used to input geometry, material, winding and mesh parameters (see chapter 4). There are two versions of MotorXP Design Studio: as an integral part of MotorXP-PM (accessible from the MotorXP-PM toolbar and the **Desktop** menu), and as a standalone application. MotorXP Design Studio when used as a standalone application is shown in Figure 3.2. Figures 4.1, 4.13 and 4.16 demonstrate MotorXP Design Studio when used as a part of MotorXP-PM. For example, it is more convenient to use the standalone version of MotorXP Design Studio when working with geometry scripts (see chapter 11). You can also use the standalone version of MotorXP Design Studio to create a prototype of the machine and then open it in MotorXP-PM for analysis. MotorXP Design Studio standalone and MotorXP-XP use the same format of project files and are fully compatible with each other – files created using a standalone version of MotorXP Design Studio can be open in MotorXP-XP and vice versa.

3.4. Working with project files.

All machine parameters, simulation settings, finite element mesh and simulation results are stored in a *project file*. The project file has .mxp extension.

Standard file commands of creating, opening and saving project files are available from the **File** menu or using corresponding toolbar buttons.

Machine parameters, materials and finite element mesh presented in the **Geometry Editor**, **Winding Editor** and **Mesh Editor** tabs of MotorXP Design Studio should be defined before any analysis is started and cannot be changed afterwards in order not to confuse simulation results for different design prototypes. Once you started any analysis the project file gets locked (**Geometry Editor**, **Winding Editor** and **Mesh Editor** tabs of MotorXP Design Studio become not available for editing). If you want to change any parameter in **Geometry Editor**, **Winding Editor** and **Mesh Editor** tabs of MotorXP Design Studio after the project file got locked, use the **Unlock** toolbar button  (see Figure 3.1) to unlock the project file. Be aware that all simulation data of all analysis types will be deleted.

Open As New (Blank) Project item of the **File** menu allows you to create a new project based on another project. All machine parameters, materials, finite element mesh and analysis setting will be copied in the new project file while the analysis results will not be included. This is a convenient way to create several design prototypes (each design prototype corresponding to a separate project file) changing one or several parameters to find an optimal design.

Export results to MAT-file item of the **File** menu allows you to create a MATLAB data file with all the machine parameters and simulation results for further processing using MATLAB.

3.4.1. Protected project files.

In some cases, there may be a need to hide the design details of the project, i.e. geometry, materials and winding configuration while **keeping the possibility to run all the analyses and view all the results**. It can be a situation when you are presenting the project to your customer for evaluation but do not want to disclose all the details about the design at this stage. In this case the **Save As Protected Project** option of the **File** menu can be used. The new project file with the extension **.mxpp** will be created while all the information about the geometry, materials and winding configuration will be excluded from the file. When the protected file is open in MotorXP Design Studio, it will look like shown in Figure 3.3.

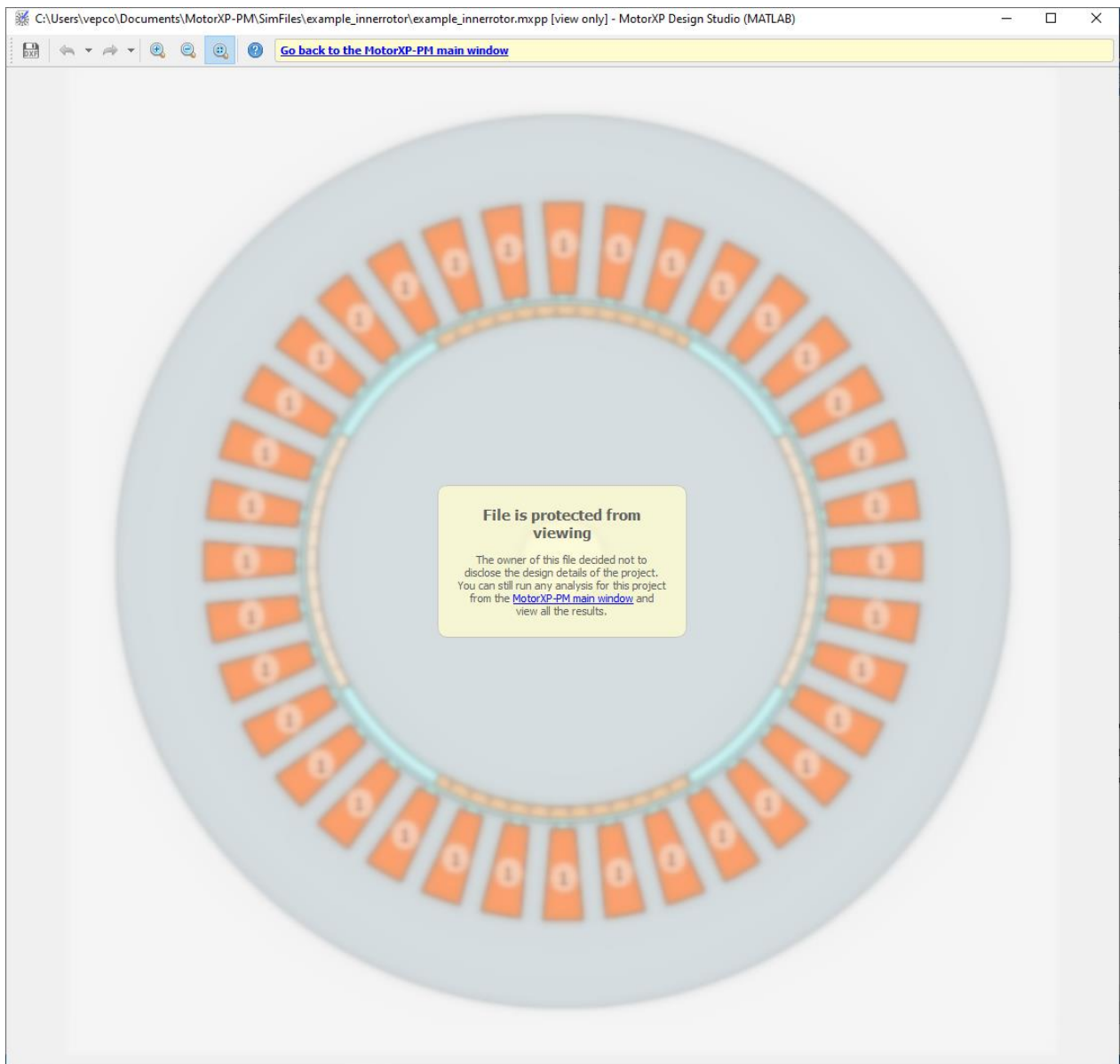


Figure 3.3. Protected project view in MotorXP Design Studio.

3.4.2. Import from MotorAnalysis-PM.

To import design data from the simulation file with a .mat extension created with MotorAnalysis-PM the **Import from MotorAnalysis-PM** item of the **File** menu of the MotorXP Design Studio standalone version should be used as shown in Figure 3.4.

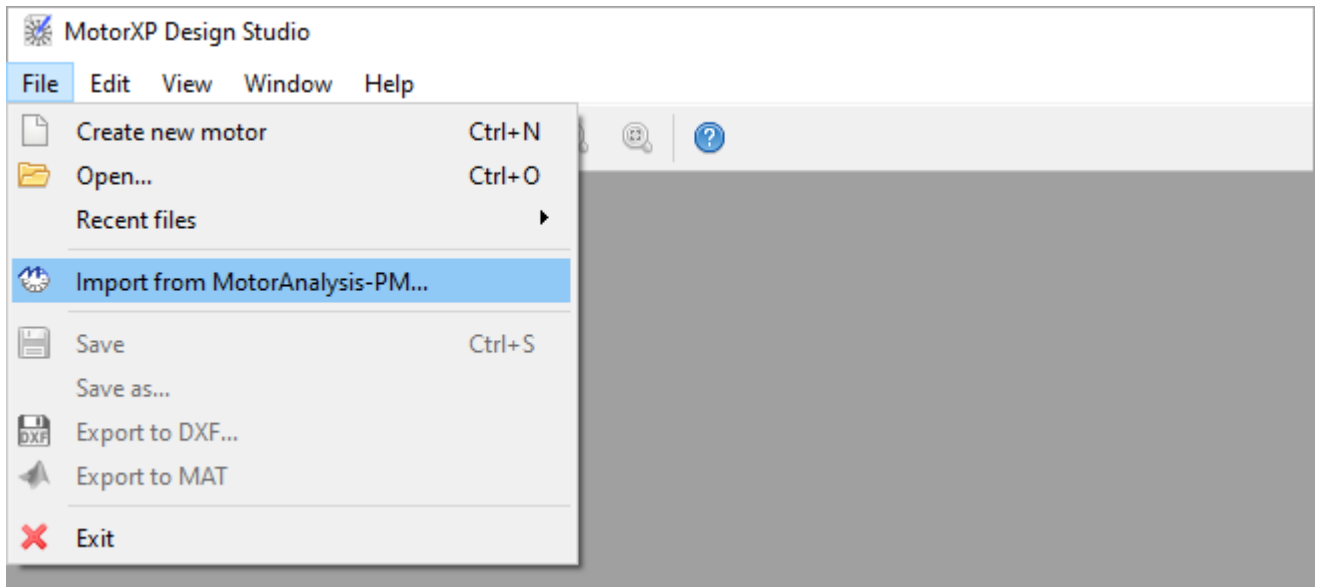


Figure 3.4. Import from MotorAnalysis-PM mat-file.

After the mat-file is selected, it will be automatically converted into mxp-file format. Note that parameters stored in **Drive Settings**, **Rated Data** and all the simulation settings will not be imported.

3.5. Licensing options.

By default, MotorXP-PM is provided free of charge under the **FREE Edition** license – MotorXP-PM (FREE Edition). MotorXP-PM (FREE Edition) can be upgraded to **MotorXP-PM (Pro)** by purchasing a **License**. MotorXP-PM (FREE Edition) has all the functionality of MotorXP-PM (Pro) except that the FREE Edition of the program does not activate project files. It means that all the project files created or modified in MotorXP-PM (FREE Edition) will not be available for any analysis and simulation using MotorXP-PM (FREE Edition). After the project file has been activated in MotorXP-PM (Pro), it also becomes available for analysis using MotorXP-PM (FREE Edition). The project file can be activated in MotorXP-PM (Pro) by running any analysis or by using the **Activate Project** option of the **File** menu. The sign to the right of the project file name (as shown in Figure 3.5) indicates whether the project has been activated or not, i.e. whether the project is available for analysis using MotorXP-PM (FREE Edition) or not.

*If you need to send your project file to someone else who is using MotorXP-PM (FREE Edition) for analysis, make sure that there is the green **Activated** sign to the right of the project file name as shown in Figure 3.5. It enables all the analysis options in MotorXP-PM (FREE Edition) for this project file.*

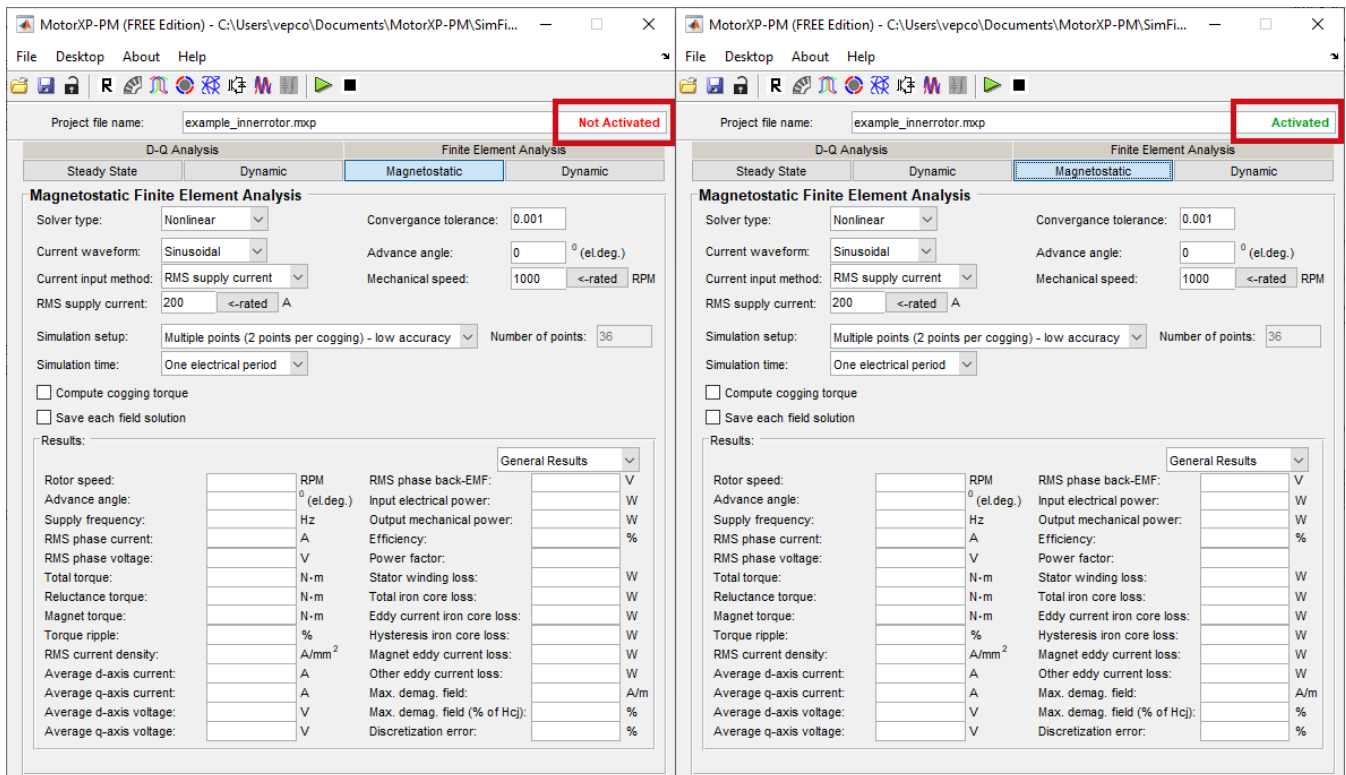
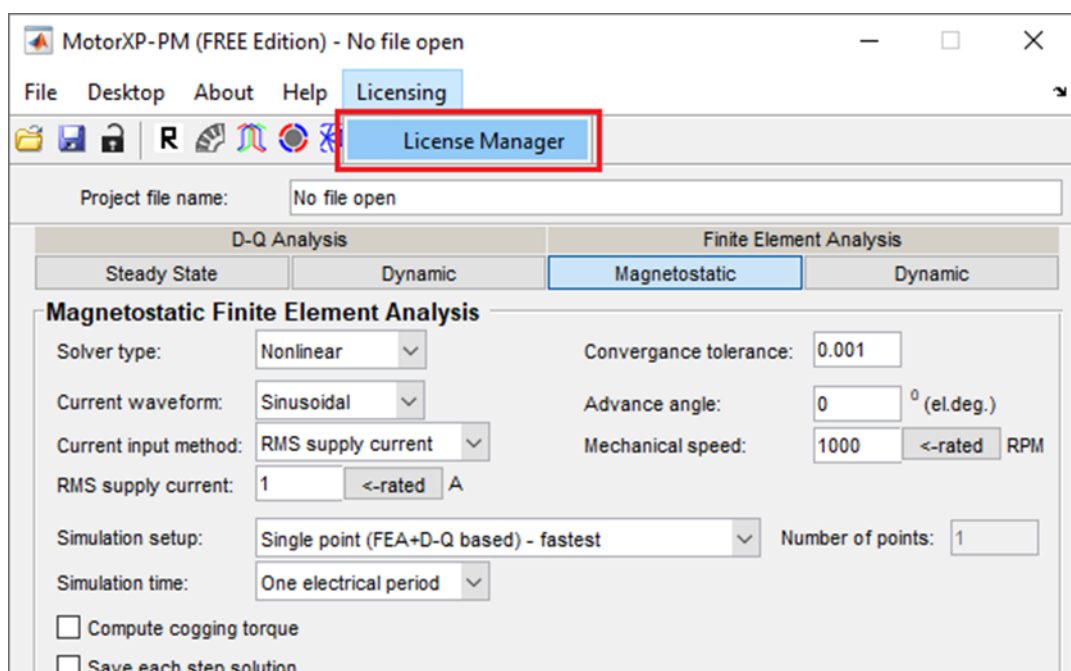


Figure 3.5. Not Activated and Activated project files open in MotorXP-PM (FREE Edition).

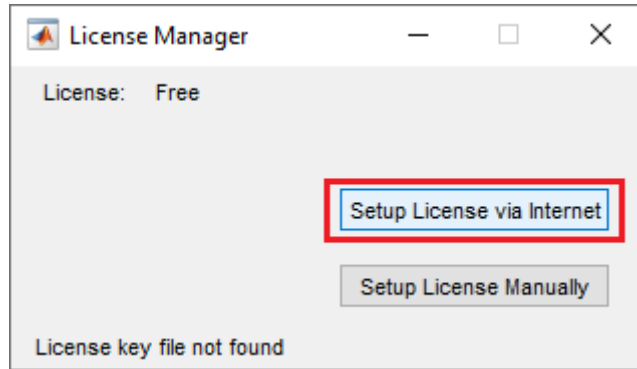
3.6. Activating the license.

In most cases the MotorXP-PM license is linked to the MotorXP account and activated via Internet following the steps given below.

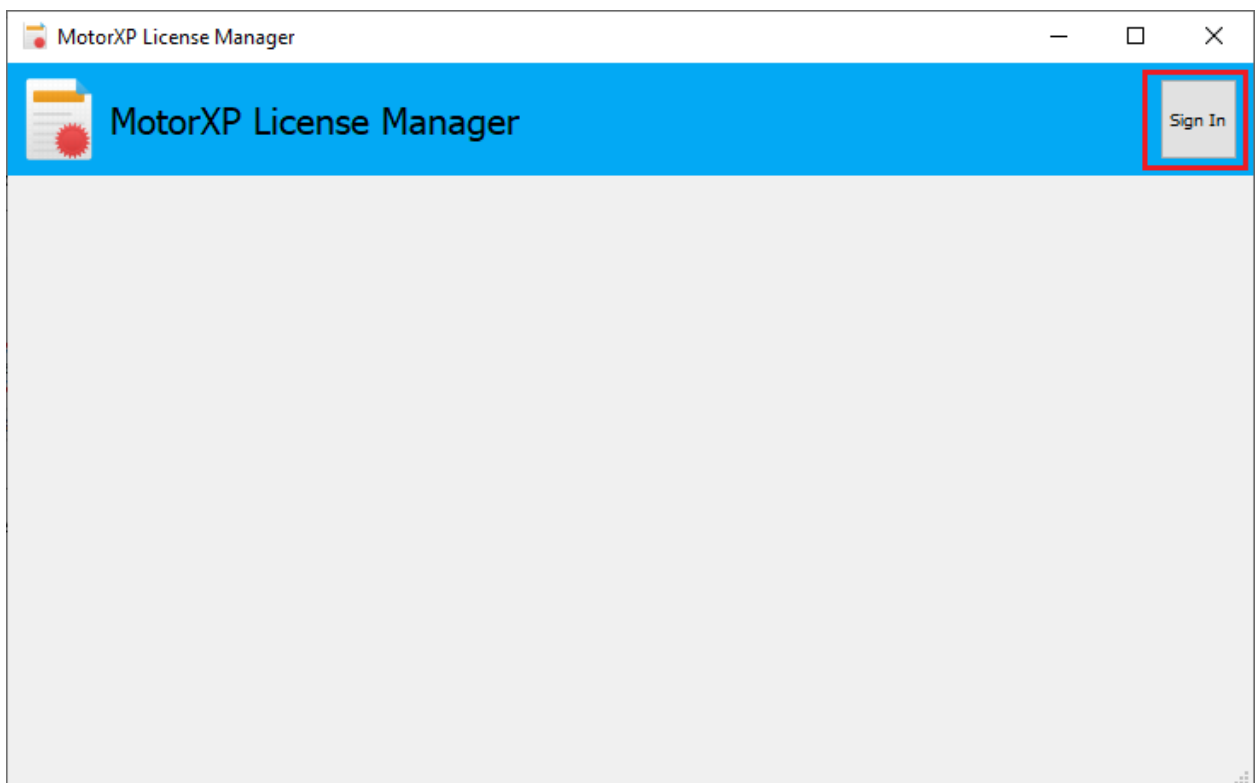
1. Open License Manager



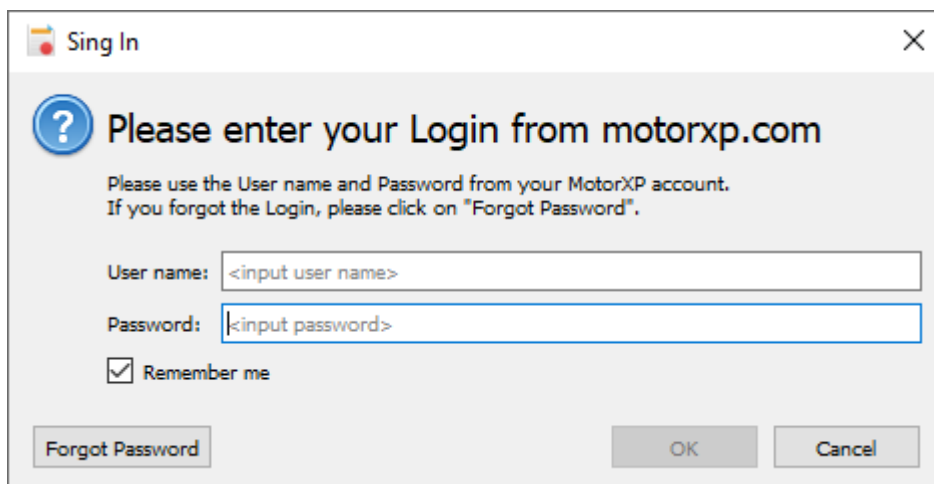
2. In the **License Manager** window click the *Setup License via Internet* button



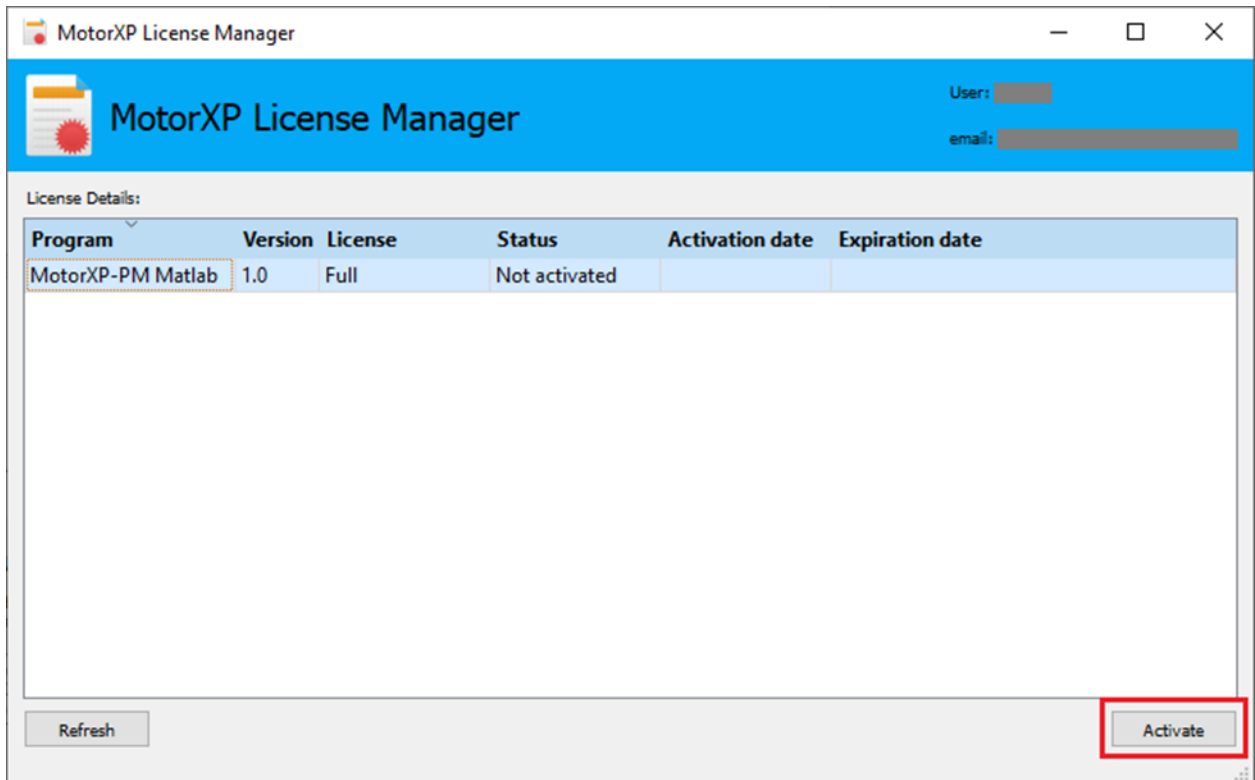
3. In the next window click the *Sign In* button:



4. Enter the MotorXP account user name and password then click the **OK** button:



5. Select the license you want to activate and click the **Activate** button



Close **License Manager**.

After successful activation the licensing status in the **License Manager** window will change from **Free** to **Pro**.

Table 3.1. Summary of MotorXP-PM analysis types.

	Analysis method	Simulation models used for the analysis	What is included into analysis	Application	Results and visualization	Comments
Magnetostatic Analysis	Single point magnetostatic analysis	One step magnetostatic FE model with single rotor position and D-Q representation of the machine.	General electromagnetic performance; Demagnetization.	Simulation of steady state performance for single operating point with ideal sinusoidal current waveform.	Time-averaged parameters; Air gap distribution plots; Cross-section field plots.	Determines motor performance from single run of magnetostatic FE simulation for one rotor position and D-Q representation of the machine. Medium accuracy since rotor rotation is not considered, medium computation speed.
	Multiple points (time-stepping) magnetostatic analysis with sinusoidal and trapezoidal current waveform	Time-stepping magnetostatic FE model with multiple rotor positions.	General electromagnetic performance; Cogging torque and torque ripple; Back-EMF harmonics; Iron losses (including higher harmonics and minor hysteresis loops); Eddy current magnet and sleeve losses; Demagnetization.	Simulation of steady state performance for single operating point with ideal sinusoidal or trapezoidal current waveform.	Time-averaged parameters; Time plots; Air gap distribution plots; Cross-section field plots; Animation.	Current driven simulation (current waveform is predefined in advance). Rotor rotation is considered by running simulations for several rotor positions. High accuracy and low computation speed.
Steady State D-Q Analysis	Steady state D-Q analysis with sinusoidal supply	Steady state D-Q model. Simplified inverter losses model (see section 2.8.2).	General electromagnetic performance; Sinusoidal back-EMF assumed; Fundamental frequency iron losses; Field weakening strategy; Mechanical losses; Inverter losses (only space-vector PWM).	Simulation of steady state performance for the range of operating conditions with ideal sinusoidal current waveform.	Motor characteristic curves as a function of speed, current or advance angle; Efficiency map and other performance maps.	Requires preliminary parameterization of D-Q model *. Medium accuracy and high computation speed.
	Steady state D-Q analysis with “FEA based” model type	One step magnetostatic FE model with single rotor position and D-Q representation of the machine.	General electromagnetic performance; Field weakening strategy.		Motor characteristic curves as a function of speed, current or advance angle.	Does not require preliminary parameterization of D-Q model *. Medium accuracy without iron losses taken into account and medium computation speed.
	Steady state D-Q analysis with PWM supply	Dynamic D-Q model.	General electromagnetic performance; Sinusoidal back-EMF assumed; Fundamental frequency iron losses; Field weakening strategy; PWM switching; Inverter voltage drop.	Simulation of steady state performance for the range of operating conditions with PWM sine wave voltage.	Motor characteristic curves as a function of speed, current or advance angle.	Requires preliminary parameterization of D-Q model *. Medium accuracy with influence of PWM switching taken into account. Medium computation speed.
Dynamic D-Q Analysis	Dynamic D-Q analysis with linear, linearized, and nonlinear solvers	Dynamic D-Q model. Inverter losses model (see section 2.8.1)	General electromagnetic performance; Sinusoidal back-EMF assumed; Fundamental frequency iron losses; PWM switching; Inverter voltage drop; Inverter losses.	Simulation of transient and steady state performance for single operating point with PWM sine wave or six-step voltage.	Time-averaged parameters; Time plots	Requires preliminary parameterization of D-Q model *. Voltage driven simulation – current is determined by applied DC voltage and switching sequence. Medium accuracy and high computational speed.
Dynamic FE Analysis	Dynamic FE analysis	Transient finite element model coupled with electrical circuit; Dynamic D-Q model to calculate initial conditions and speed up simulation.	General electromagnetic performance; Induced eddy currents; Cogging torque and torque ripple; Back-EMF harmonics; Iron losses (including higher harmonics and minor hysteresis loops); Demagnetization; PWM switching; User defined electrical circuits; User defined control algorithms.	Simulation of transient and steady state performance for single operating point with PWM sine wave or six-step voltage. Simulation of specific operating conditions which cannot be simulated with other analysis methods.	Time-averaged parameters; Time plots; Air gap distribution plots; Cross-section field plots; Animation.	Very high accuracy but very low computational speed since the simulation should go through the initial transient until the steady state is reached.

* Preliminary parameterization of D-Q model is a process of extraction of D-Q model parameters using the finite element model of the machine. It should be done only once and takes from several minutes to several hours. Refer to section 6.1 for more information.


4. CREATING DESIGN PROTOTYPES

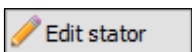
Design prototypes of an electric machine are created using a tool called MotorXP Design Studio which is a part of MotorXP-PM. MotorXP Design Studio is accessible from the MotorXP-PM main window toolbar shown in Figure 3.1. MotorXP Design Studio is also available as a standalone application as described in section 3.3. MotorXP Design Studio is divided into three tabs to input machine's geometry and materials (**Geometry Editor**), winding parameters (**Winding Editor**) and mesh parameters (**Mesh Editor**). Four toolbar buttons of the MotorXP-PM main window can be used to open the corresponding tab of MotorXP Design Studio, or the required tab can be accessed from the navigation panel. The view of the MotorXP Design Studio window when the **Geometry Editor** tab is open is shown in Figure 4.1.

4.1. Geometry Editor.


Geometry Editor tab of MotorXP Design Studio allows you to set up dimensions of the machine and specify materials and their characteristics. **Geometry Editor** tab of MotorXP Design Studio is shown in Figure 4.1.

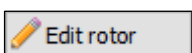
Machine type specifies either the machine with *Inner rotor* or *Outer rotor* is used. Stator geometry is defined either by the geometry template or the stator geometry can be imported from DXF-file. There are three standard stator geometry templates available which can be chosen from the **Stator geometry** pop-up menu: *Parallel tooth*, *Parallel slot* and *General slot*. You can also create your own geometry templates for stator or modify existing geometry templates (see chapter 11 for more details). **Winding layers** pop-up menu allows you to choose either *Single layer* or *Double layer* stator winding is used. If the double layer stator winding is chosen, the **Layer orientation** field (*Upper / Lower* or *Left / Right*) specifies either the stator slot will be divided horizontally or vertically into two parts with equal areas.

Stator geometry parameters placed under the **Stator geometry** pop-up menu correspond to the specific parameters of the selected geometry template. Use button  right under the **Stator geometry** pop-up menu to see all the dimensions corresponding to the selected geometry template. Button



opens the stator geometry in editing mode as shown in Figure 4.2.

Rotor geometry is defined either by the geometry template or the rotor geometry can be imported from DXF-file. There are several standard rotor geometry templates available which can be chosen from the **Rotor geometry** pop-up menu: *Surface mounted radial*, *Surface mounted parallel*, *Halbach array rotor*, *Bread loaf*, *Straight buried*, *Spoke* and *V-shaped*. You can also create your own geometry templates for rotor or modify existing geometry templates (see chapter 11 for more details). Rotor geometry parameters placed under the **Rotor geometry** pop-up menu correspond to the specific parameters of the selected geometry template. Use button  right under the **Rotor geometry** pop-up menu to see all the dimensions corresponding to the selected geometry template. Button



opens the rotor geometry in editing mode.

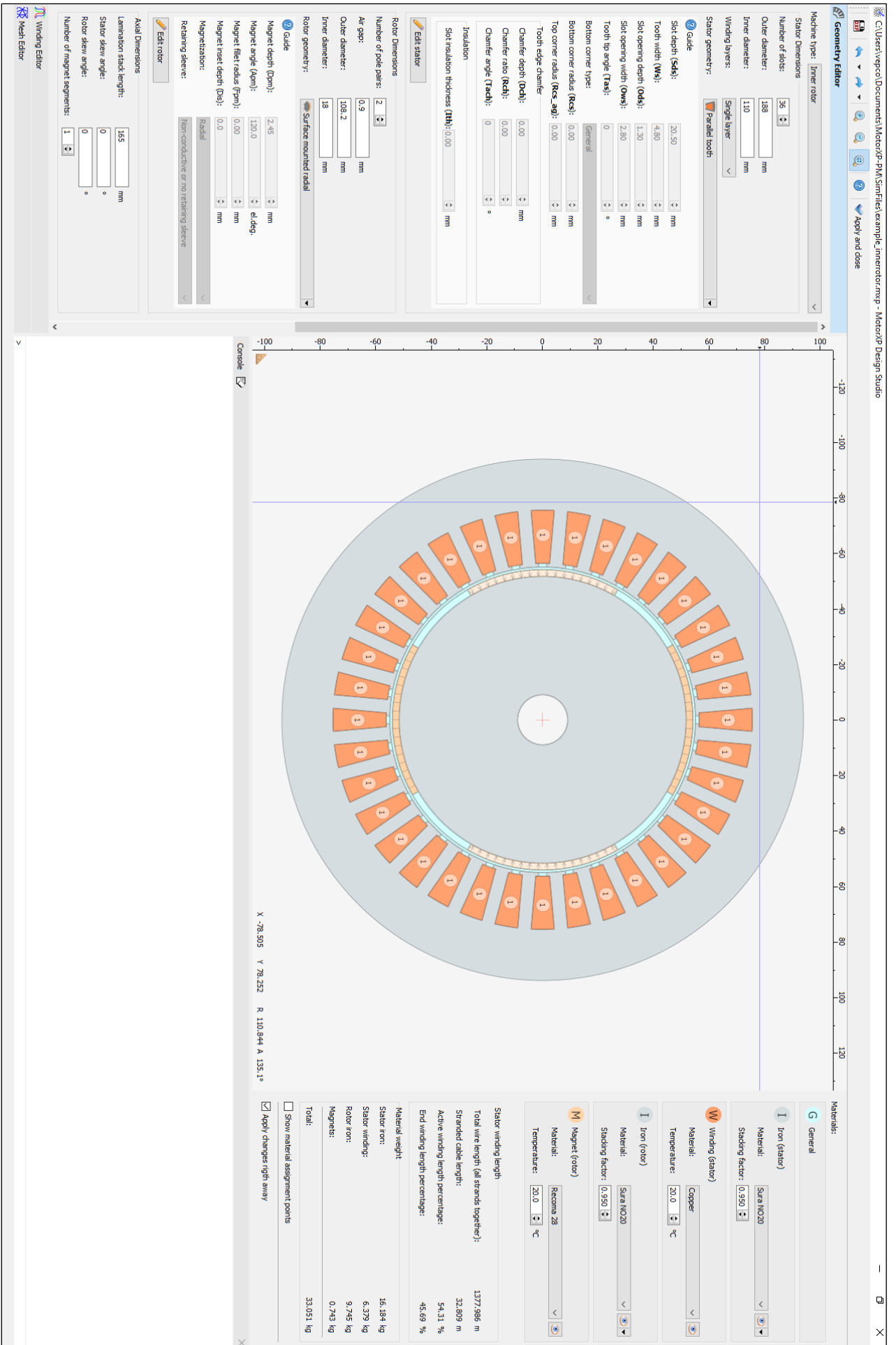


Figure 4.1. Geometry Editor of MotorXP Design Studio.

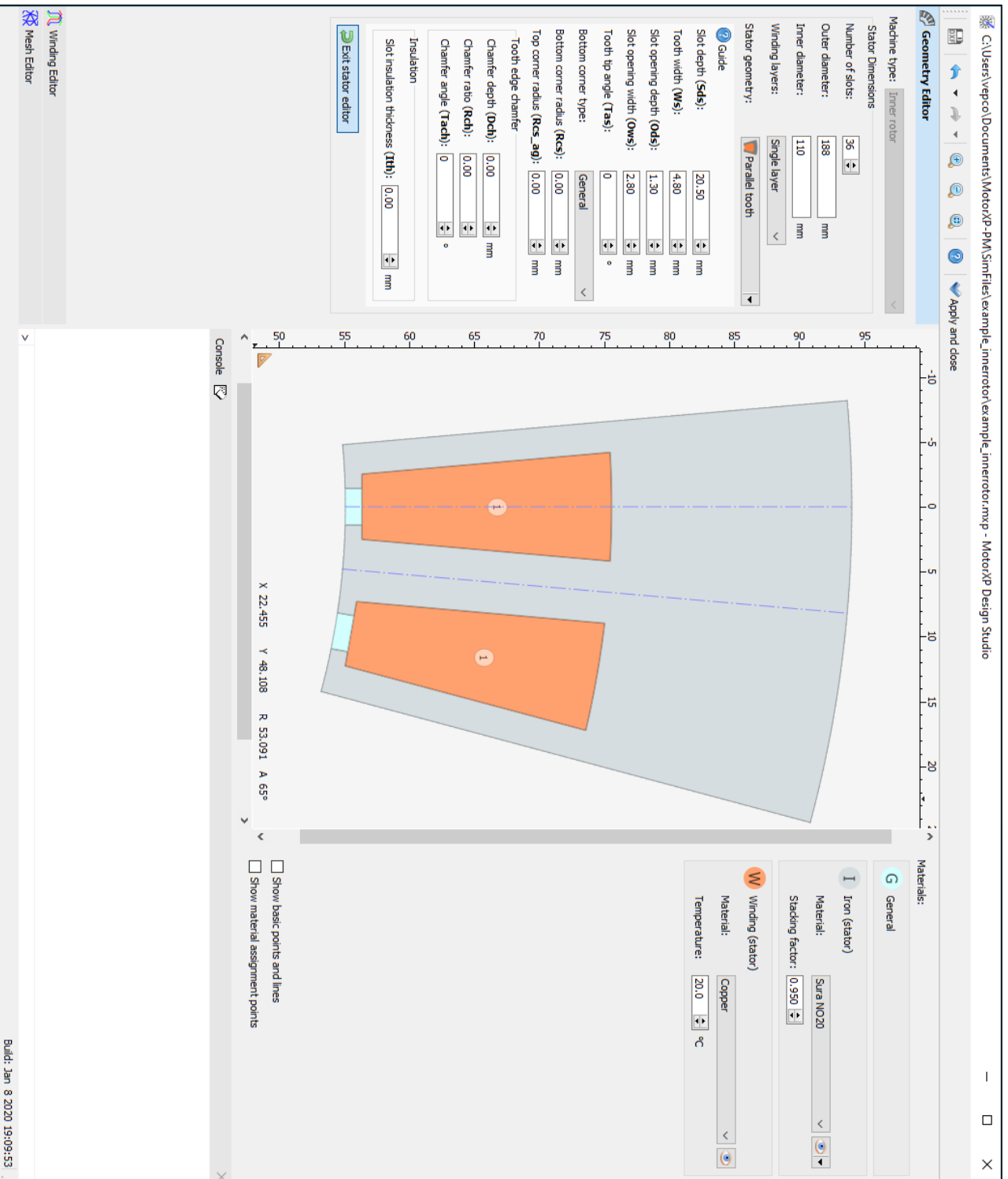


Figure 4.2. Geometry Editor of MotorXP Design Studio, stator geometry editing mode.

There is also an option of stator tooth shape optimization in MotorXP-PM available for all standard stator geometry templates. The stator tooth shape optimization can lead to reduction of the torque ripple and the noise radiation providing the same average torque. Changing stator tooth shape can be achieved by setting up tooth edge chamfers in the **Tooth edge chamfer** panel of the **Geometry Editor** tab.

Axial dimensions of the machine include length of the lamination stack, radial stator and rotor skew angles and **Number of magnet segments** along the shaft. **Lamination stack length** is the length of the stator and rotor iron cores as shown in Figure 4.3. **Stator skew angle** or **Rotor skew angle** may be used to reduce the torque ripple. Note that in case of stator or rotor skewing the multi-slice FEM (see section 2.4) is used to simulate the skewed geometry and **Number of slices** in **Mesh Editor** should be more than 1 (see section 4.4).

Segmentation of magnets along the shaft as shown in Figure 4.4 helps to reduce eddy currents induced in the magnet and thus reduce the magnet losses and magnet temperature. If the number of magnet segments is more than one, it is assumed that each magnet is divided in a given number of pieces along z-axis electrically isolated from each other. Segmented magnets are also used to apply rotor magnet skewing.

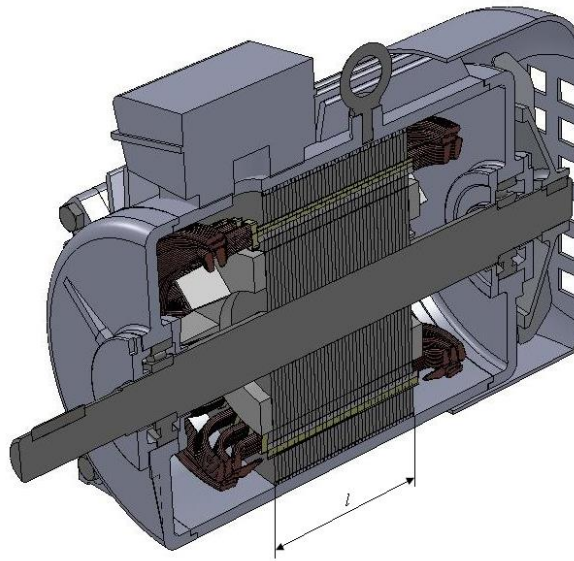


Figure 4.3. Lamination stack length.

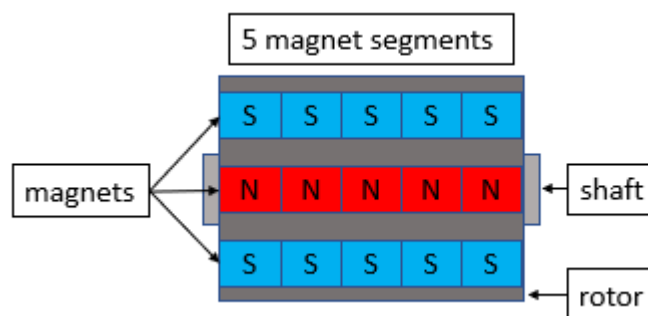
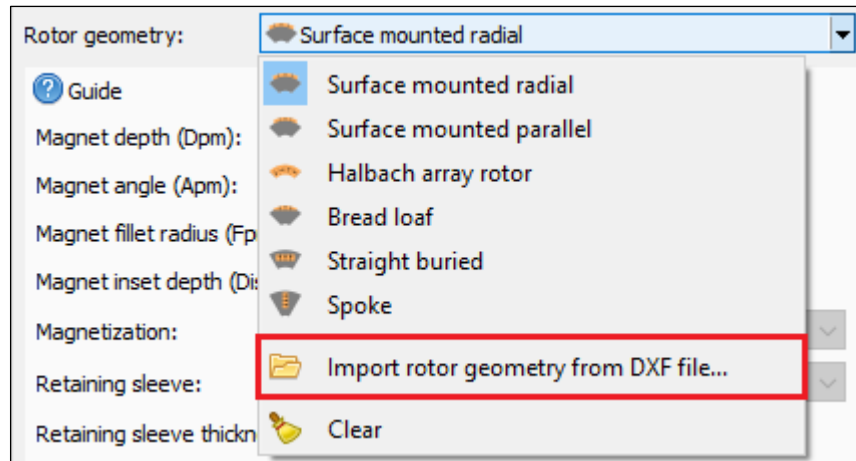


Figure 4.4. Segmentation of magnets along the shaft.

4.1.1. Importing rotor geometry from DXF-file.

To import rotor geometry from DXF-file select the corresponding item from the **Rotor geometry** pop-up menu then open DXF-file with the rotor geometry:



The DXF rotor geometry requirements are shown in Figure 4.5. The rotor geometry being imported must include one half of a pole pitch with a center line of the pole being aligned with the y-axis. The center of the rotor must coincide with the origin (0, 0). Inner and outer boundaries of the rotor geometry can be of arbitrary shape.

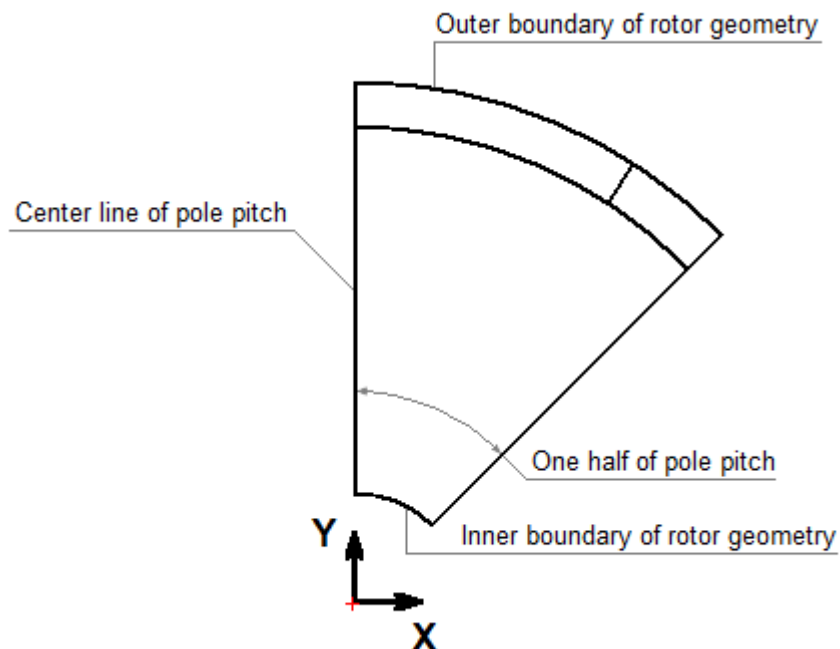


Figure 4.5. DXF rotor geometry requirements.

Once the DXF-file is open the window for importing rotor geometry will appear as shown in Figure 4.6. Choose the units of DXF-file. Then assign subdomain type for each part of the geometry. Only **General** (for air, insulation, etc.), **Iron** (for iron core), **Magnet** (for permanent magnets) and **Conductor** subdomain types are allowed for the rotor. Use **Conductor** subdomain type for any conductive parts of the rotor other than **Magnet** or **Iron** (for example, retaining sleeve). Eddy current losses in all **Conductor** subdomains are considered during analysis and appear in the **Other eddy current loss** parameter of **Magnetostatic Analysis** (see chapter 5). Use **Not assigned** for subdomains you want to exclude. For example, if you have the drawing of the whole machine, the **Not assigned** subdomain type should be used for the stator subdomains so only the rotor geometry will be imported.

The **Division factor** should be applied if the full cross-section drawing of the machine is used to meet the DXF rotor geometry requirements shown in Figure 4.5. The division factor for the rotor should be twice of the number of poles as shown in Figure 4.7 on the example of a four-pole surface-mounted permanent magnet rotor drawing with division factor 8.

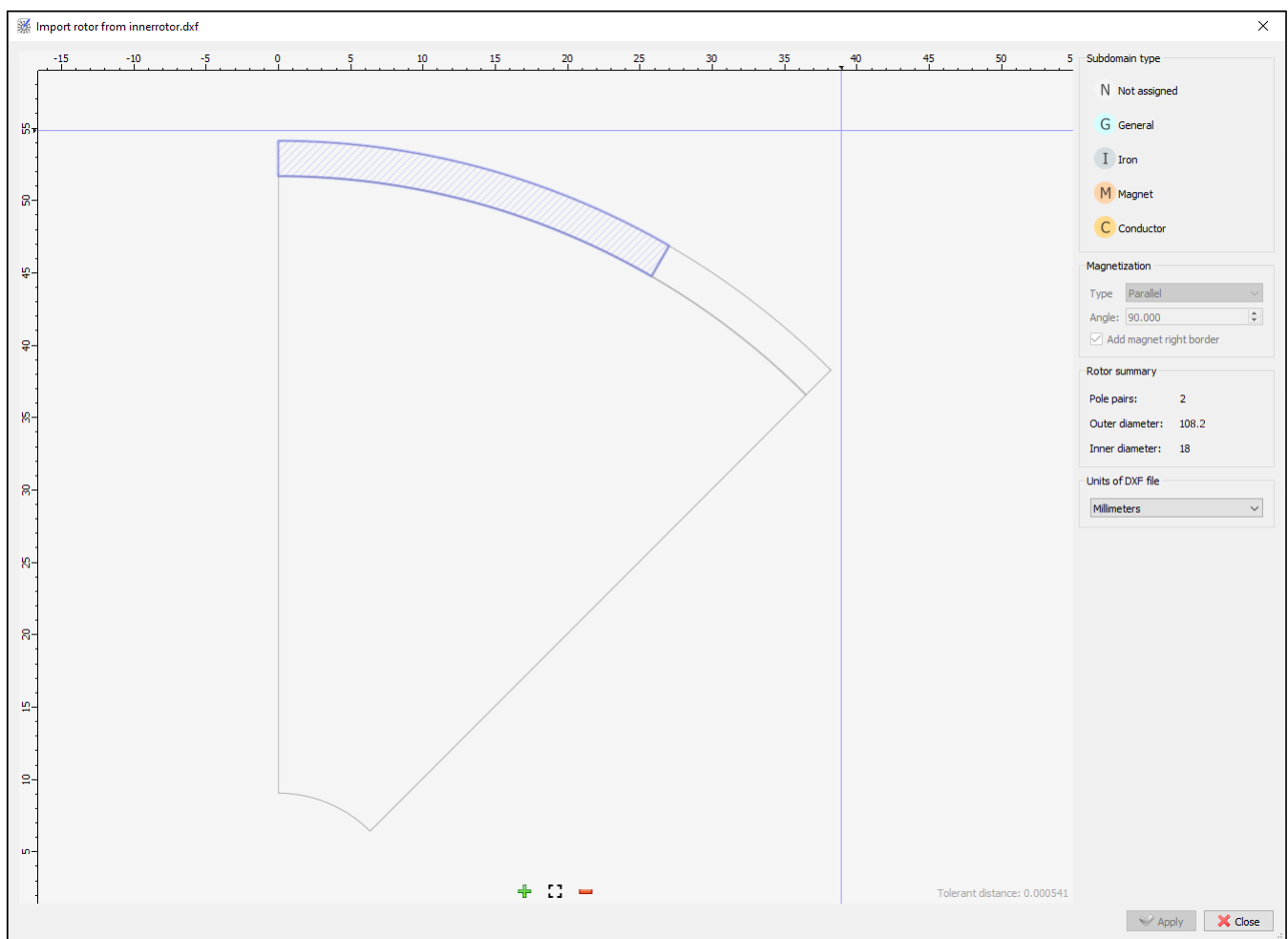


Figure 4.6. DXF rotor geometry importing window.

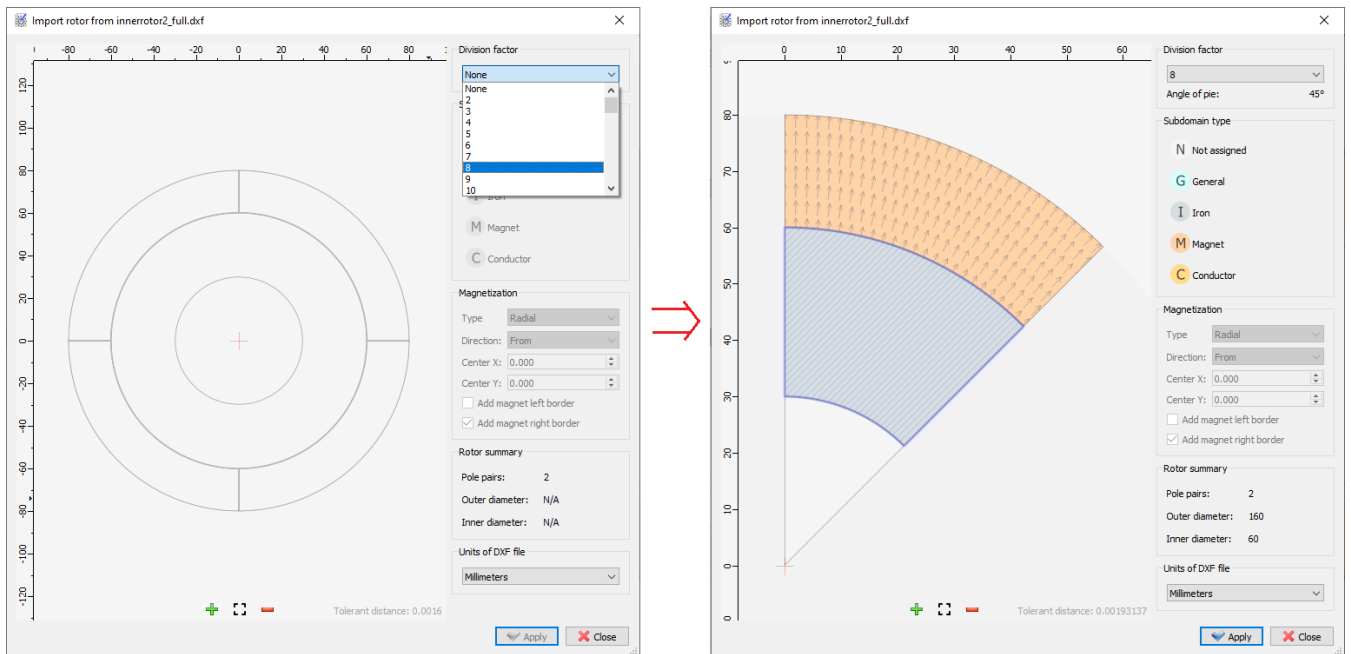


Figure 4.7. DXF rotor geometry with division factor applied.

For the permanent magnet the magnetization properties should be specified. The magnet can have either **Radial** or **Parallel** magnetization. For the radial magnetization the direction (from the center, towards the center, clockwise or counterclockwise with respect to the center of magnetization) and center of magnetization should be defined. The parallel magnetization is defined by the angle of magnetization. While importing the rotor geometry from DXF-file the magnetization is defined only for magnets of the right half of the pole pitch (DXF-file area in Figures 4.8 and 4.9), the magnetization for the rest of the geometry is automatically determined as shown in Figure 4.8 for parallel magnetization and in Figure 4.9 for radial magnetization. As seen from Figure 4.8 the magnetization angle assigned to the right half of pole pitch is mirrored for the left half of pole pitch except when the magnetization angle is equal to 0 (Figure 4.8(b)). For example, the magnetization angle of 107.5° will be mirrored to 72.5° in the left half of pole pitch (Figure 4.8(a)). Note that the magnetization angles of 90° and 180° are mirrored to angles of the same values of 90° and 180° . In case of radial magnetization, as shown in Figure 4.9, the counterclockwise magnetization assigned to the right half of pole pitch is mirrored for the left half of pole pitch (becomes clockwise), while three other magnetization directions are not mirrored.

Magnet left or right borders can be assigned to add the electrical insulation between corresponding adjacent magnets. Insulation between magnets is taken into account while eddy current losses in magnets are calculated. Also, if the magnet in the rotor geometry drawing is divided in several segments (several subdomains), it is assumed that these segments are electrically isolated from each other which is also taken into account while eddy current losses in magnets are calculated. Magnet segmentation can be used to reduce eddy currents induced in magnet and thus reduce the magnet losses and its temperature.

The rotor geometries shown in Figures 4.6 and 4.7 will be imported and appear in **Geometry Editor** after clicking the **Apply** button.

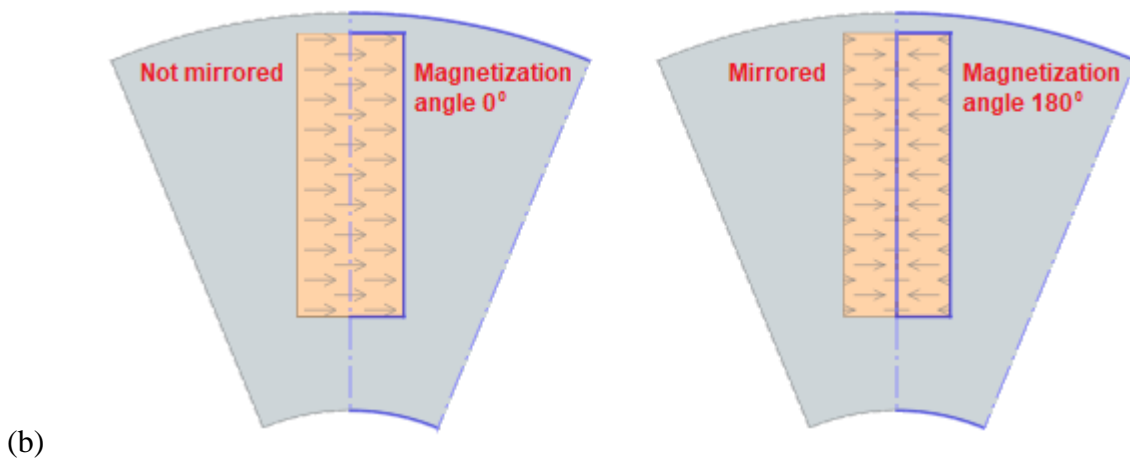
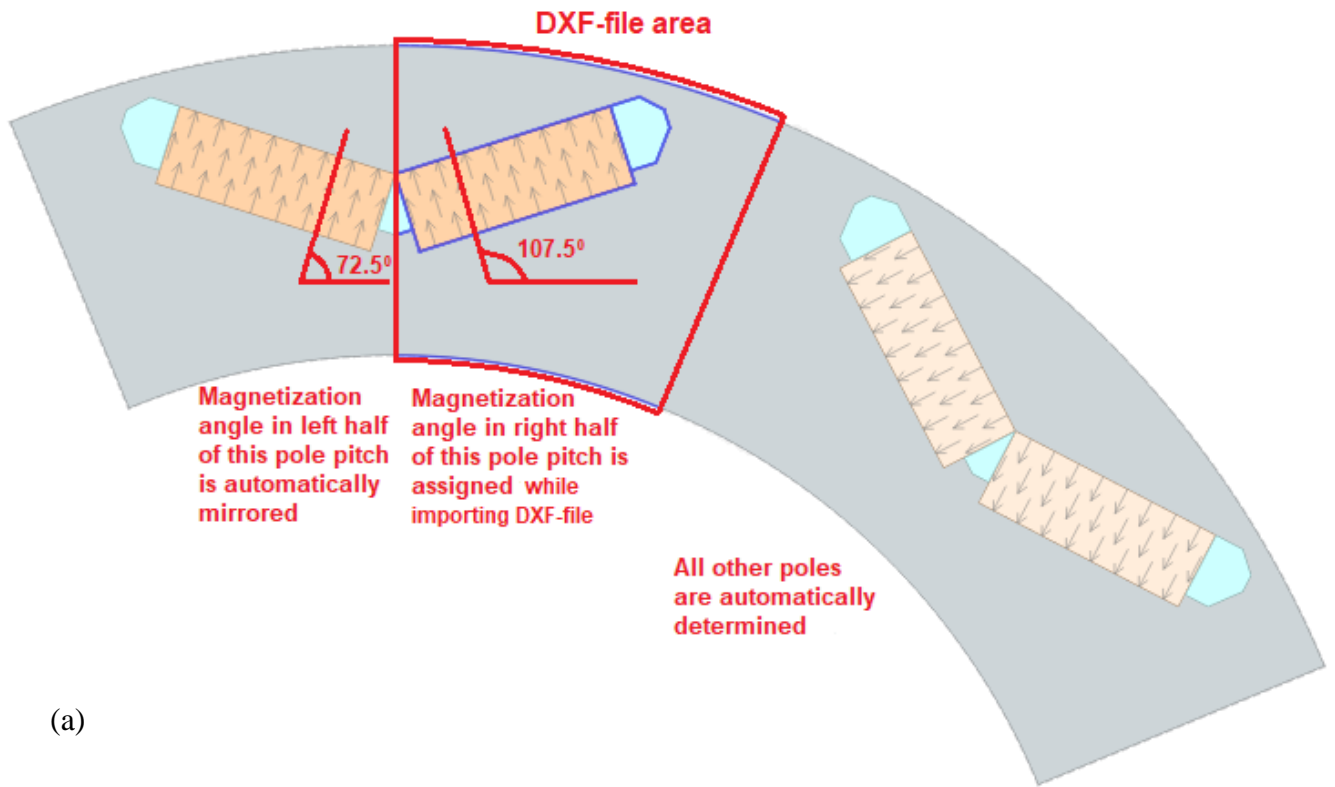


Figure 4.8. (a) Explanation on magnetization angle assignment; (b) Magnetization angle of 0° is not mirrored in left half of pole pitch.

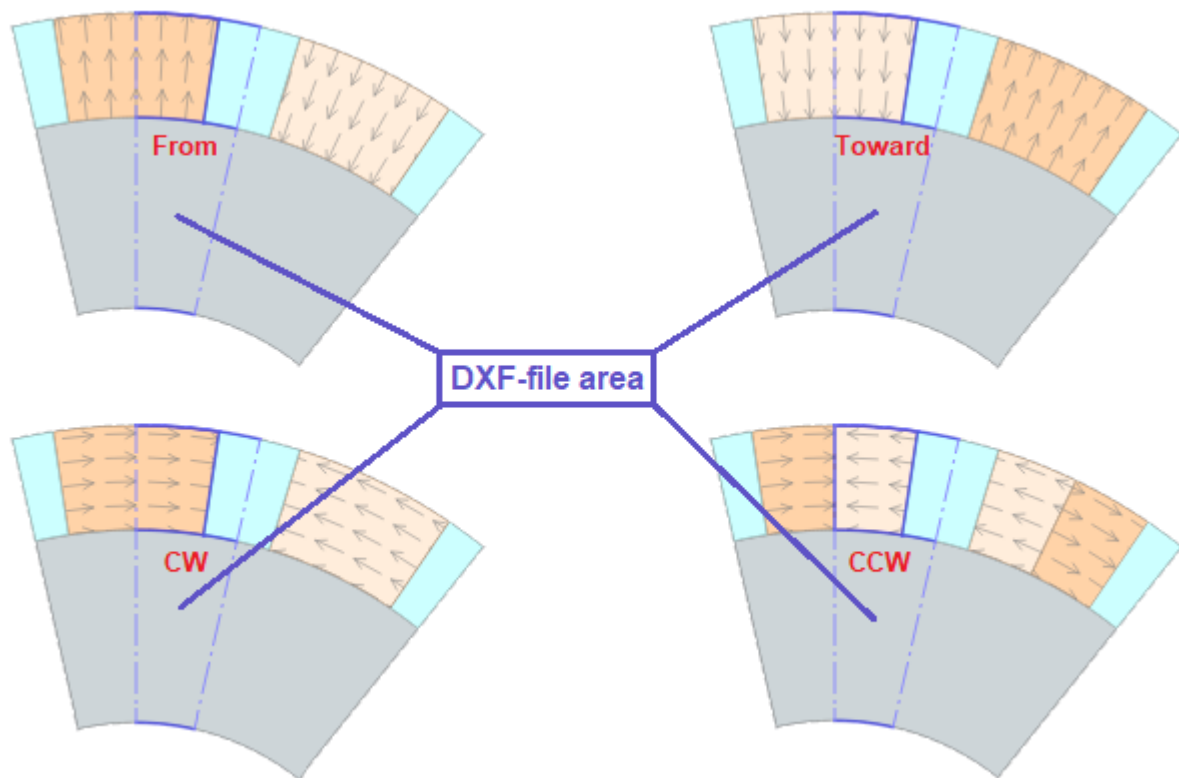
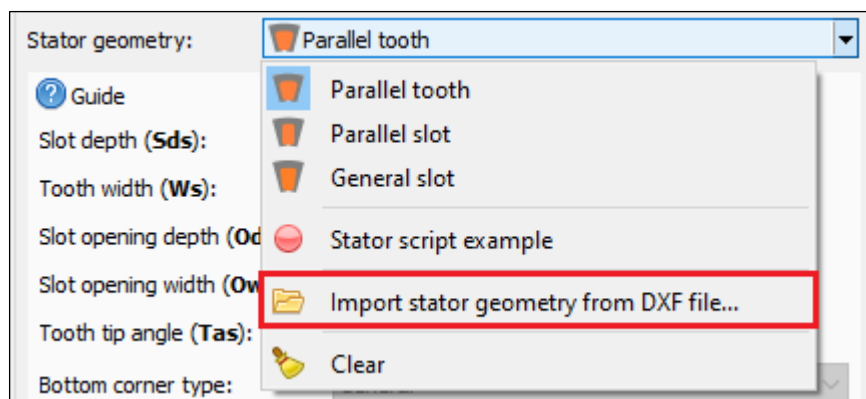


Figure 4.9. Radial magnetization examples with different directions (from center, toward center, clockwise (CW), counterclockwise (CCW)); clockwise magnetization is not mirrored in the left half of pole.

4.1.2. Importing stator geometry from DXF-file.

To import stator geometry from DXF-file select the corresponding item from the **Stator geometry** pop-up menu then open DXF-file with the stator geometry:



The DXF stator geometry requirements are shown in Figure 4.10. The stator geometry being imported must include one half of a slot pitch with a center line of the slot being aligned with the y-axis. The center of the stator must coincide with the origin (0, 0). Inner and outer boundaries of the stator geometry can be of arbitrary shape.

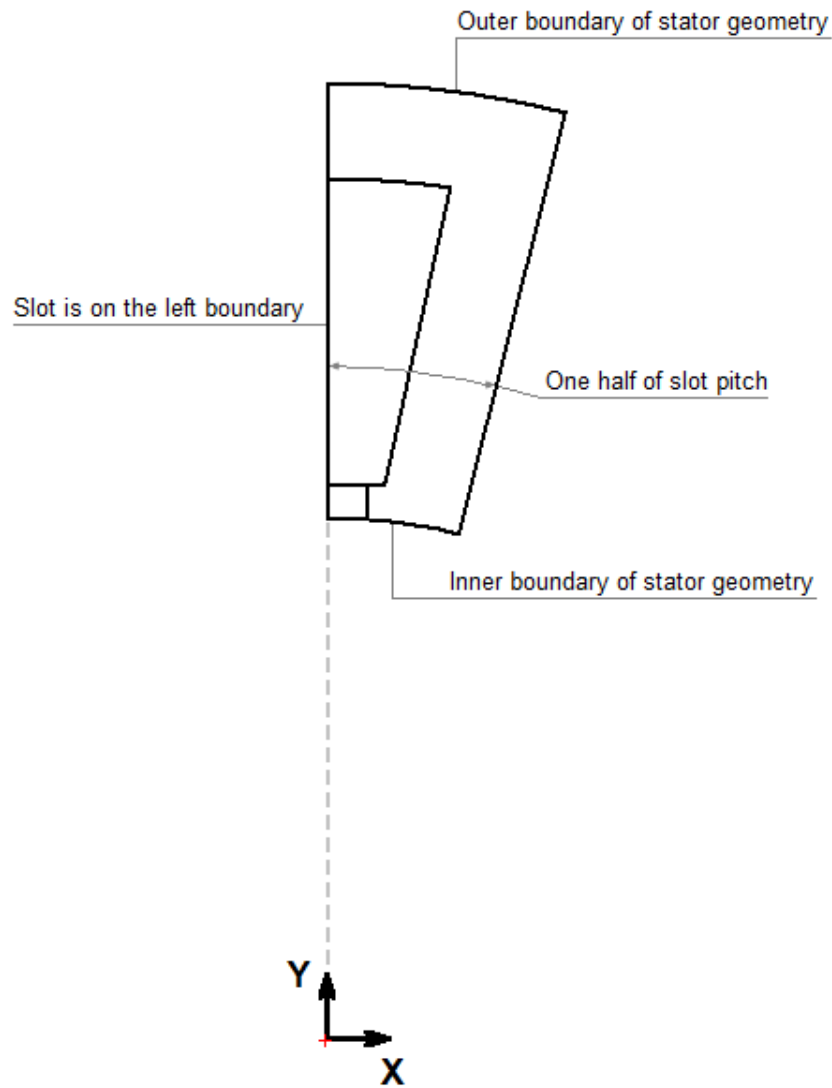


Figure 4.10. DXF stator geometry requirements.

Once the DXF-file is open the window for importing stator geometry will appear as shown in Figure 4.11. Choose the units of DXF-file. Then assign subdomain type for each part of the geometry. Only **General** (for air, insulation, etc.), **Iron** (for iron core) and **Winding** (for stator winding) subdomain types are allowed for the stator. Use **Not assigned** for subdomains you want to exclude. For example, if you have the drawing of the whole machine, the **Not assigned** subdomain type should be used for the rotor subdomains so only the stator geometry will be imported.

Check the **Vertical slot divider** checkbox if the left/right winding layers position is used. Assign the **Winding layer number** for each visible layer (subdomain of 'Winding' type) starting from '1' as shown in Figure 4.11 (bottom) on the example of a double layer slot with the upper/lower winding layers position. Use **Division factor** if the full cross-section drawing of the machine is used similarly to the rotor geometry importing described in the previous section. The division factor for the stator should be twice the number of slots. The stator geometry will be imported and appear in **Geometry Editor** after clicking the **Apply** button.

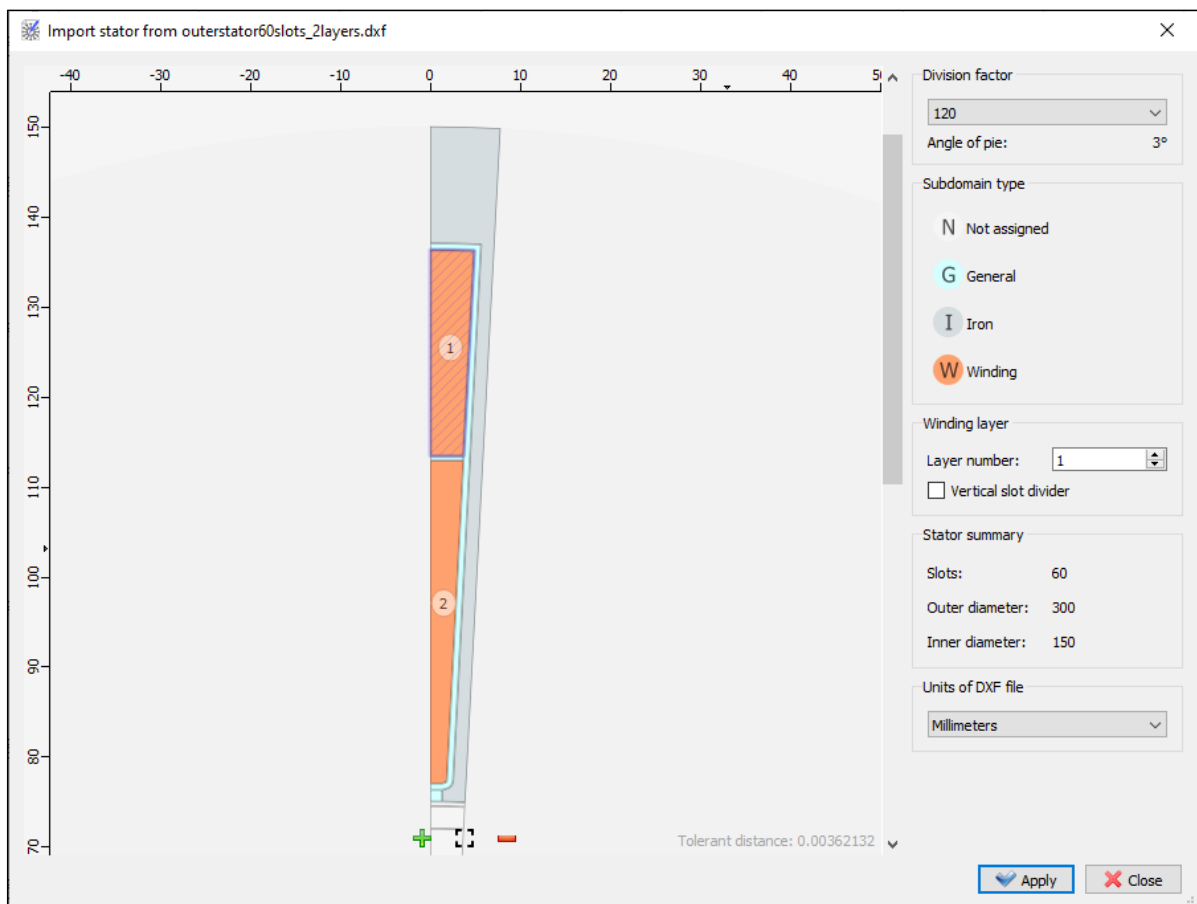
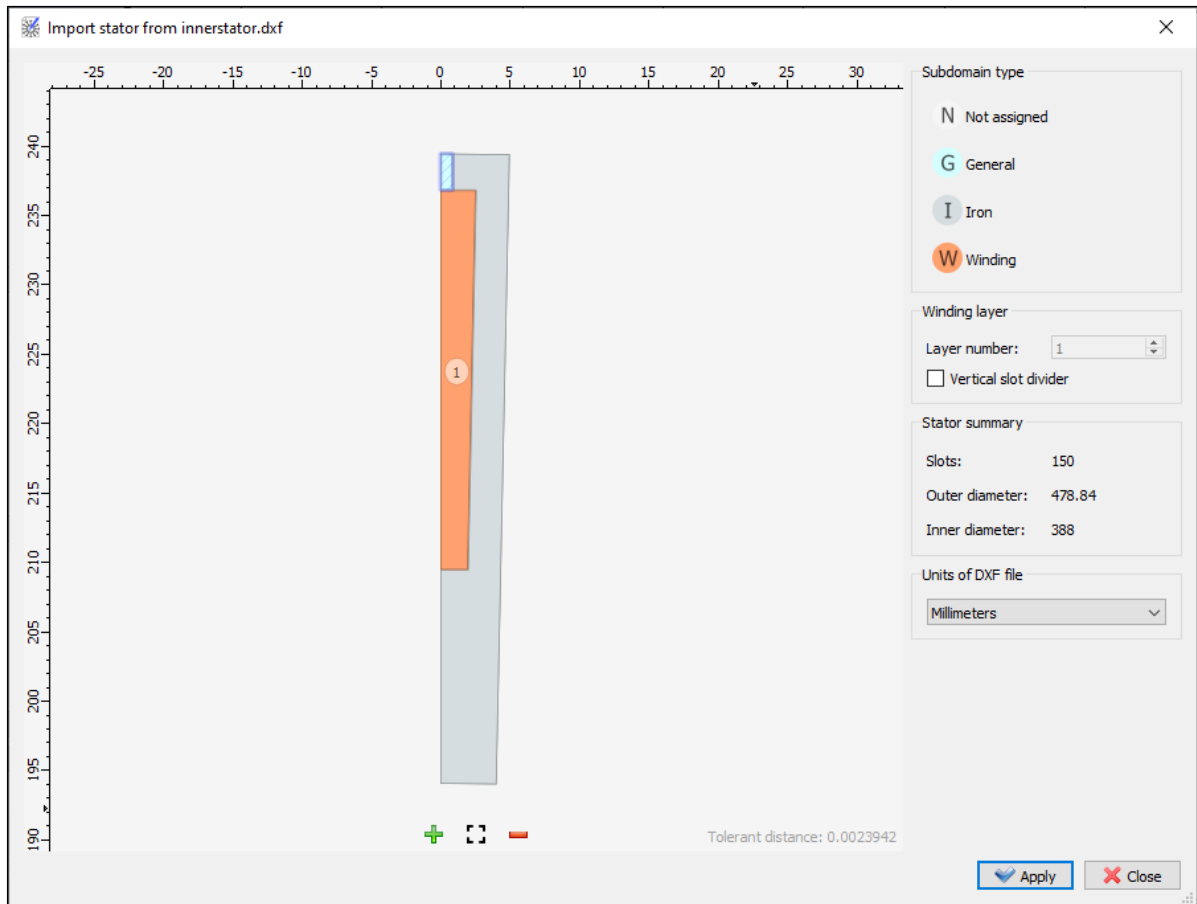
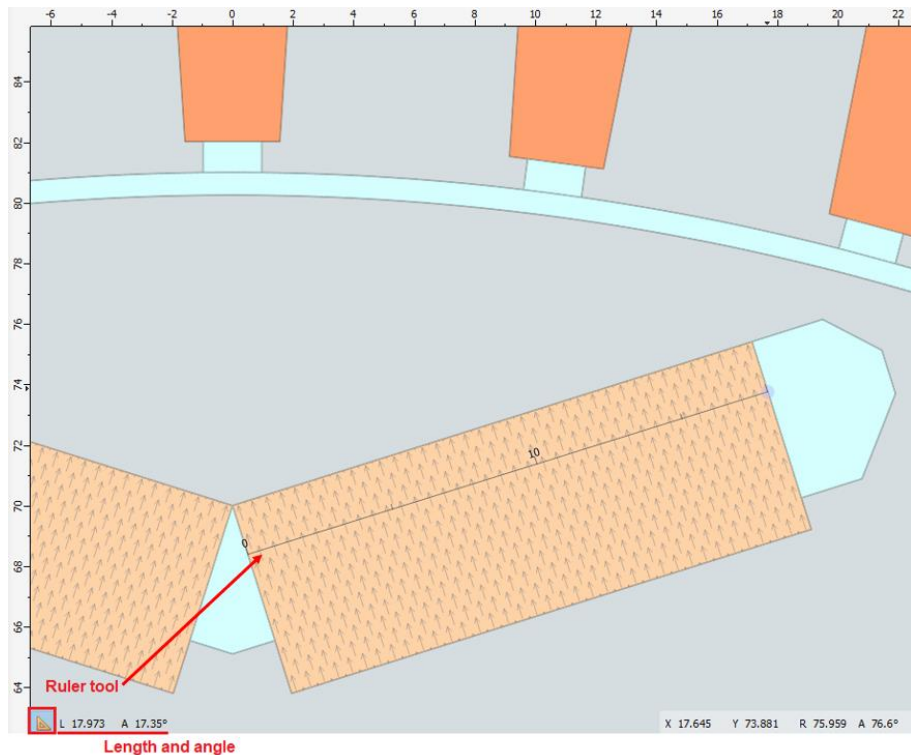


Figure 4.11. DXF stator geometry importing window with inner stator and single layer slot example (top) and outer stator and double layer slot example (bottom).

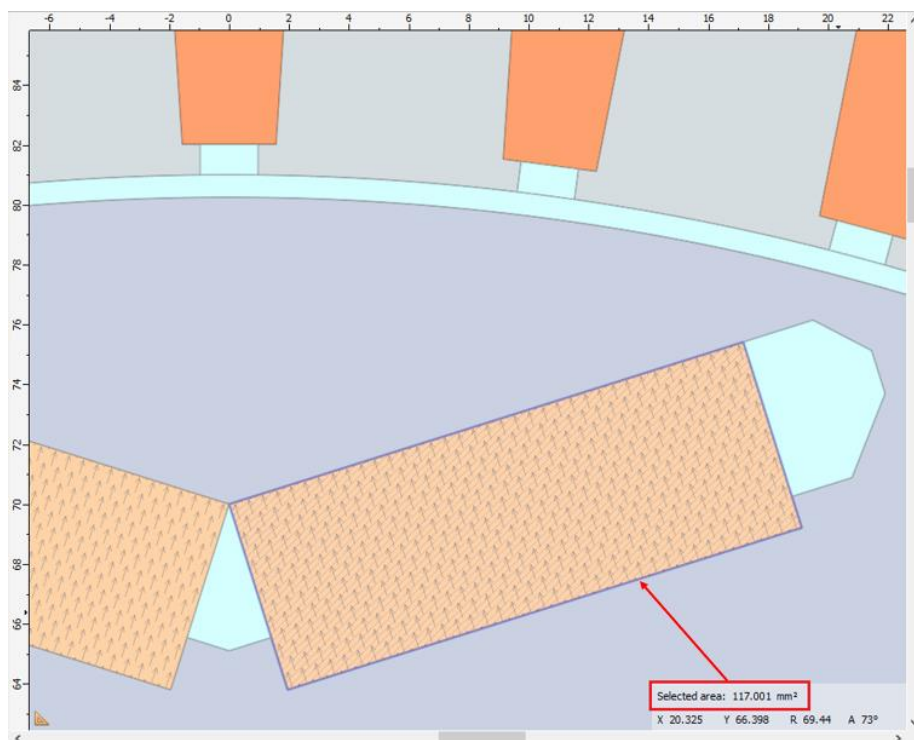
4.1.3. Using other tools of Geometry Editor.

The Ruler tool allows you to measure distances and angles. To access the Ruler tool, click the button in the left bottom angle of the geometry drawing:



Use the mouse to place the ruler over the geometry.

Select any subdomain of the geometry to see its area (use the Ctrl key to select several subdomains):



Use button  to export the geometry of the machine into DXF-file.

4.2. Assigning materials.

The panel for assigning materials is shown on the right when the **Geometry Editor** tab of MotorXP Design Studio is open (see Figure 4.12).

Materials:

G General

I Iron (stator)

Material: M-15 29 Ga

Stacking factor: 0.900

W Winding (stator)

Material: Copper

Temperature: 20.0 °C

I Iron (rotor)

Material: M-15 29 Ga

Stacking factor: 0.900

M Magnet (rotor)

Material: N28AH

Temperature: 20.0 °C

C Conductor (rotor)

Material: Aluminum

Temperature: 20.0 °C


Stator winding length

Total wire length (all strands together):	1566.446 m
Stranded cable length:	15.664 m
Active winding length percentage:	45.96 %
End winding length percentage:	54.04 %

Material weight

Stator iron:	15.261 kg
Stator winding:	7.252 kg
Rotor iron:	9.663 kg
Magnets:	1.118 kg
Conductors:	0.126 kg
Total:	33.420 kg

Figure 4.12. The panel for assigning materials in MotorXP-PM Design Studio.

There are five types of materials (**General**, **Iron**, **Winding**, **Magnet** and **Conductor**) assigned to different parts of the machine. Clicking button  opens the file with material properties.

Stacking factor field specifies the ratio of the volume filled by electrical steel to the total volume of the iron core. The total volume of the iron core consists of the volume of lamination steel sheets and the volume of the coating between the sheets. Stacking factor reduces the flux carried by the iron core which is taken into account by MotorXP-PM through the stacking factor value.

Temperature of stator winding specifies the temperature of the winding conductors. Winding phase resistance is automatically adjusted for the given temperature according to the winding material temperature coefficient of resistance. The expression for temperature effect on resistance is as follows:

$$R = R_{20^{\circ}\text{C}} * [1 + \alpha * (T - 20^{\circ}\text{C})],$$

where R – resistance at temperature T ; $R_{20^{\circ}\text{C}}$ – resistance at 20°C ; α – temperature coefficient of resistance; T – winding temperature.

Temperature of magnet has its effect on the permanent magnet properties such as remanence flux density and intrinsic coercivity. The remanence flux density used by MotorXP-PM is defined as follows:

$$Br = Br_{20^{\circ}\text{C}} * [1 + \alpha_{Br} * (T_{pm} - 20^{\circ}\text{C}) / 100],$$

where Br – remanence flux density at temperature T_{pm} ; $Br_{20^{\circ}\text{C}}$ – remanence flux density at 20°C ; α_{Br} – temperature coefficient of remanence flux density in $\% / ^{\circ}\text{C}$; T_{pm} – permanent magnet temperature.

Similarly, the intrinsic coercivity is defined as follows:

$$Hcj = Hcj_{20^{\circ}\text{C}} * [1 + \alpha_{Hcj} * (T_{pm} - 20^{\circ}\text{C}) / 100],$$

where Hcj – intrinsic coercivity at temperature T_{pm} ; $Hcj_{20^{\circ}\text{C}}$ – intrinsic coercivity at 20°C ; α_{Hcj} – temperature coefficient of intrinsic coercivity in $\% / ^{\circ}\text{C}$; T_{pm} – permanent magnet temperature.

Temperature of the **Conductor** material affects the resistance of the conductor according to its temperature coefficient of resistance:

$$R = R_{20^{\circ}\text{C}} * [1 + \alpha * (T - 20^{\circ}\text{C})],$$

where R – resistance of the conductor at temperature T ; $R_{20^{\circ}\text{C}}$ – resistance at 20°C ; α – temperature coefficient of resistance; T – conductor temperature.

4.2.1. Adding new materials.

You can add your own materials to the Materials Library or modify properties of the existing materials. All material files are stored in *[User data directory]\Materials*. Each group of materials is stored in a separate folder (*Conductor*, *Iron* and *Magnet*) inside the *Materials* folder. The materials can also be sorted by subcategories, for example, ferrite magnets are stored in *[User data directory]\Materials\Magnet\Ferrite Magnets*, while neodymium magnets are stored in

[User data directory]\Materials\Magnet\Neodymium Magnets. Each material has the following properties:

Conductor:

- Electric resistivity at 20⁰C
- Temperature coefficient of resistance
- Mass Density

Iron:

- B-H curve
- Iron loss
- Electric resistivity at 200C (only for soft magnetic composites)
- Mass Density

Magnet:

- Electric resistivity
- Relative permeability
- Coercivity
- Remanence flux density
- Intrinsic coercivity
- Temperature coefficient of remanence flux density
- Temperature coefficient of intrinsic coercivity
- Mass Density

File with material properties is a txt-file of a special format. The example of the file with material properties is shown below (only part of the file is shown, full text can be found in [User data directory]\Materials\Iron\Non-oriented Silicon Steels\Sura NO20.txt):

<Description>

Sura NO20
Non-oriented silicon steel

<B-H curve> at 20°C

% H(A/m)	B(T)
0	0
10	0.0190
20	0.0530
30	0.1100
37	0.2000

...

...

...

30000	1.9730
50000	2.0590
100000	2.1830
130000	2.2310

<Iron loss>

% frequency(Hz)	flux density(T)	iron loss(w/kg)
-----------------	-----------------	-----------------

50	0.1000	0.0200
50	0.2000	0.0700
50	0.3000	0.1400
50	0.4000	0.2300
50	0.5000	0.3200
50	0.6000	0.4200
50	0.7000	0.5400
50	0.8000	0.6600
50	0.9000	0.8000
50	1.0000	0.9500
50	1.1000	1.1400
50	1.2000	1.3600
50	1.3000	1.6500
50	1.4000	2.0000
50	1.5000	2.4000
50	1.6000	2.7500
50	1.7000	3.0600
50	1.8000	3.3200
400	0.1000	0.1700
400	0.2000	0.7200
...		
...		
...		
10000	0.1000	27.0000
10000	0.2000	95.6000
10000	0.3000	191.0000
10000	0.4000	315.0000

<Mass Density> (kg/m³)
7650

Each property is designated by a *property tag*, the name of the tag is enclosed inside the angle brackets: <B-H curve>, <Iron loss> and etc. The tag is followed by the *property content* up to the next property tag or the end of the file. If the property is not specified it means that the corresponding property tag is followed by empty line(s) up to the next property tag or the end of the file. The content of a line following after the percent sign “%” and after the property tag is ignored and treated as a comment.

Refer to the corresponding files in the Materials Library to check the structure and units of each property content while adding new material files. There are properties defined by a single value such as <Mass Density>, <Electric resistivity>, <Coercivity> and others are defined by arrays of values (<Iron loss>, <B-H curve>).

The name of the material is following right after the <Description> tag. Each property file must have a unique material name. The name of the property file may be different from the name of the material.

The magnetization properties of permanent magnets are defined by the following property tags: <Relative Permeability>, <Coercivity> and <Remanence flux density>. It is allowed to specify either all of them or any two of them and leave another one empty.

The iron loss of the iron can be defined either by iron loss data or by iron loss Steinmetz equation coefficients (see chapter 2.6). The iron loss defined by the iron loss data are shown in the example above. The first and second columns are frequency and peak flux density values respectively in increasing order exactly as shown above and the third column is the corresponding iron loss values. There are the following requirements for the flux density values:

<Iron loss>		
% frequency(Hz)	flux density(T)	iron loss(W/kg)
50	0.1	0.018371768
50	0.2	0.06912301
50	0.3	0.138519028
50	0.4	0.228291933
50	0.5	0.337521845
50	0.6	0.447654955
50	0.7	0.593724411
50	0.8	0.721922481
50	0.9	0.858938885
50	1.0	1.00136478
100	0.1	0.04015814
100	0.2	0.154411359
100	0.3	0.323172176
100	0.4	0.532617086
100	0.5	0.787456215
100	0.6	1.044402552
100	0.7	1.355424938
100	0.8	1.721270226
100	0.9	2.092930066
100	1.0	2.490155197
200	0.1	0.102196015
200	0.2	0.384508248
200	0.3	0.804748873
200	0.4	1.326299205
200	0.5	1.960888187
200	0.6	2.600724444
200	0.7	3.449338764
200	0.8	4.380356263
200	0.9	5.443134384
200	1.0	6.33704502
400	0.1	0.277585148
400	0.2	1.05364468
400	0.3	2.185861503
400	0.4	3.602498209
400	0.5	5.326170862
400	0.6	7.219224808
400	0.7	9.574852126
400	0.8	12.15921843
1000	0.1	1.067337791
1000	0.2	3.929521078
1000	0.3	8.22421281

Example of the iron loss defined by the Steinmetz equation coefficients can be found in M-15 29 Ga.txt:

```
<Iron loss>
%Steinmetz coefficients
0.0227288148583325    % kh
1                      % alfa
1.77364043109264      % beta
2.11492029537421e-05  % ke
```


If the iron loss property is not specified (property tag <Iron loss> is followed by empty line(s) up to the next property tag or the end of the file) it is assumed that the iron losses are zero.


4.3. Winding Editor.

Winding Editor tab of MotorXP Design Studio allows you to set up parameters of the stator winding. **Winding Editor** is shown in Figure 4.13.

Parameters of **Winding Editor** such as **Number of Slots**, **Number of pole pairs**, **Winding layers** and **Layers orientation** are the same as in **Geometry Editor** and described in section 4.1.

The **Wire size method** pop-up menu specifies how the **Strand diameter** value is determined. **Strand diameter** specifies the diameter of one strand (wire) of the winding. There are four wire size methods used in MotorXP-PM: **Wire diameter** (the wire diameter should be specified by the user), **AWG** (the wire diameter is specified by the AWG number according to the American wire gauge standard), **SWG** (the wire diameter is specified by the SWG number according to the Standard wire gauge), **Fill factor** (whether the **Slot fill factor** value or the **Coil fill factor** value should be specified by the user).

The **winding connection** pop-up menu specifies either the winding is star-connected or delta-connected. **Number of parallel paths** specifies the number of groups of coils per phase connected in parallel. **Number of turns** specifies the number of turns per one coil, the same as the number of turns per one winding layer. **Number of strands in hand** specifies a number of smaller wires (strands) bundled together to form a larger conductor to reduce the skin-effect. If the conductor consists of one piece of metal wire the number of strands is one. The **slot fill factor** field specifies the ratio of the area of all conductors (i.e. pure copper or aluminum) of a slot to the total slot area (including slot insulation). The **coil fill factor** is the ratio of the area of all conductors (i.e. pure copper or aluminum) of a slot to the slot area occupied by the conductors NOT including slot insulation. Use button  on the right to highlight the area used for calculation of coil or slot fill factors.

There are three methods to specify the stator winding layout: **Automatic**, **Manual** and **From file**. When you choose the **Layout method** as **Automatic** you should also specify **Coil span** in number of slots. The winding layout is displayed as three tables corresponding to each phase. The first column of each table is a coil number. The second column indicates the starting slot number, and third column indicates the starting layer number if the winding has more than one layer. Whereas the fourth column indicates the ending slot number, and fifth column indicates the ending layer of each coil if the winding has more than one layer. There is also an additional sixth column for the parallel path number appearing if the number of parallel paths is more than 1. The same parallel path number corresponds to coils of the same phase connected in series. The winding layout can be copied or saved to file by clicking the  button at the top-right corner of the winding layout table. Colored representation of the winding layout is displayed in the center part of **Winding Editor**, where the sign "+/-" specifies the forward and return direction of the conductors within a slot, the letter following the sign specifies the phase and the phase is followed by the parallel path number.

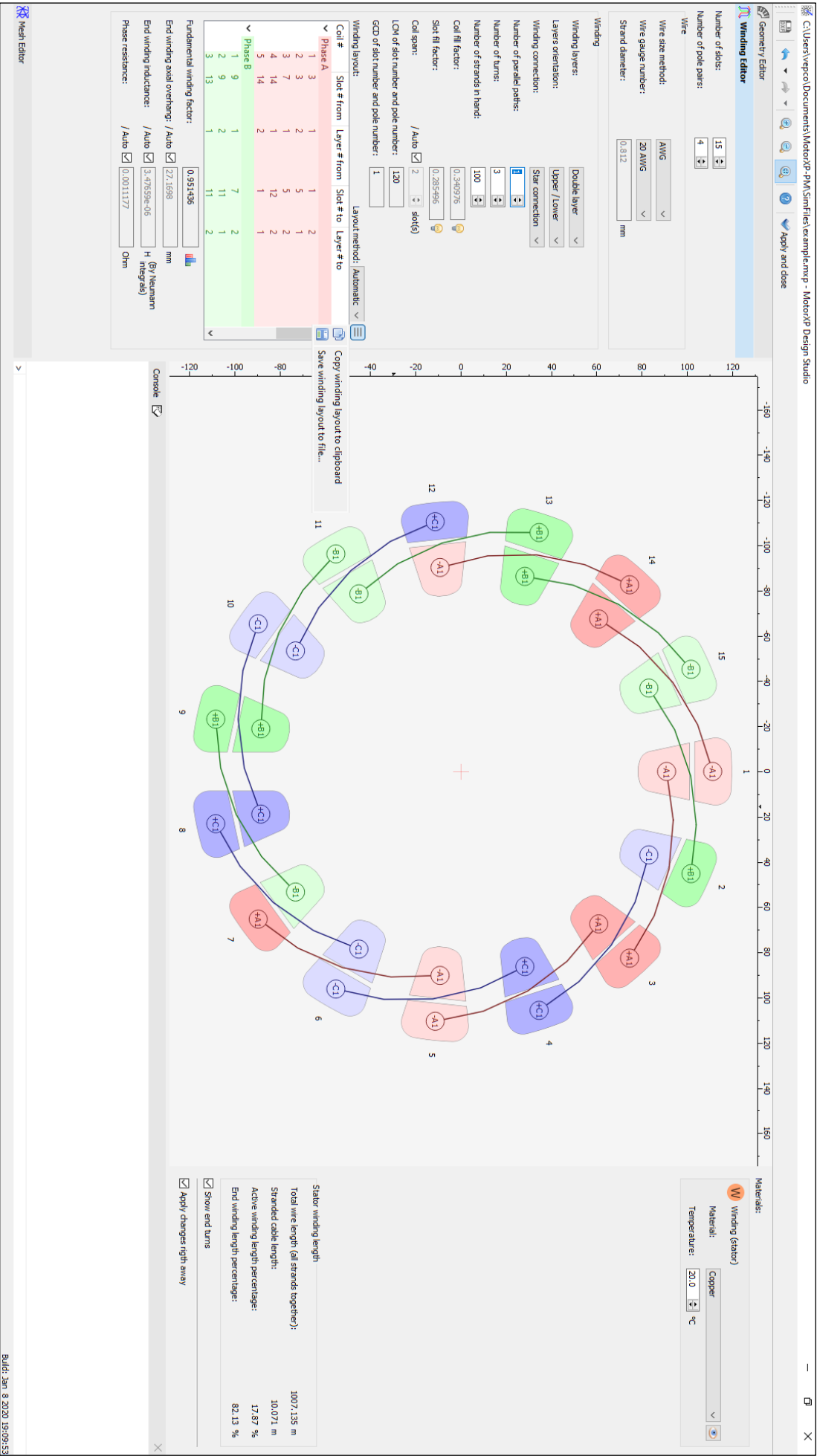


Figure 4.13. Winding Editor of MotorXP Design Studio.

1	1	2	13	1	1
2	2	2	14	1	1
3	3	2	15	1	1
4	4	2	16	1	1
5	5	2	17	1	1
6	28	1	16	2	2
7	29	1	17	2	2
8	30	1	18	2	2
9	31	1	19	2	2
10	32	1	20	2	2
11	31	2	43	1	1
12	32	2	44	1	1
13	33	2	45	1	1
14	34	2	46	1	1
15	35	2	47	1	1
16	58	1	46	2	2
17	59	1	47	2	2
18	60	1	48	2	2
19	1	1	49	2	2
20	2	1	50	2	2

1	18	1	6	2	1
2	19	1	7	2	1
3	20	1	8	2	1
4	21	1	9	2	1
5	22	1	10	2	1
6	21	2	33	1	2
7	22	2	34	1	2
8	23	2	35	1	2
9	24	2	36	1	2
10	25	2	37	1	2
11	48	1	36	2	1
12	49	1	37	2	1
13	50	1	38	2	1
14	51	1	39	2	1
15	52	1	40	2	1
16	51	2	3	1	2
17	52	2	4	1	2
18	53	2	5	1	2
19	54	2	6	1	2
20	55	2	7	1	2

1	11	2	23	1	1
2	12	2	24	1	1
3	13	2	25	1	1
4	14	2	26	1	1
5	15	2	27	1	1
6	38	1	26	2	2
7	39	1	27	2	2
8	40	1	28	2	2
9	41	1	29	2	2
10	42	1	30	2	2
11	41	2	53	1	1
12	42	2	54	1	1
13	43	2	55	1	1
14	44	2	56	1	1
15	45	2	57	1	1
16	8	1	56	2	2
17	9	1	57	2	2
18	10	1	58	2	2
19	11	1	59	2	2
20	12	1	60	2	2

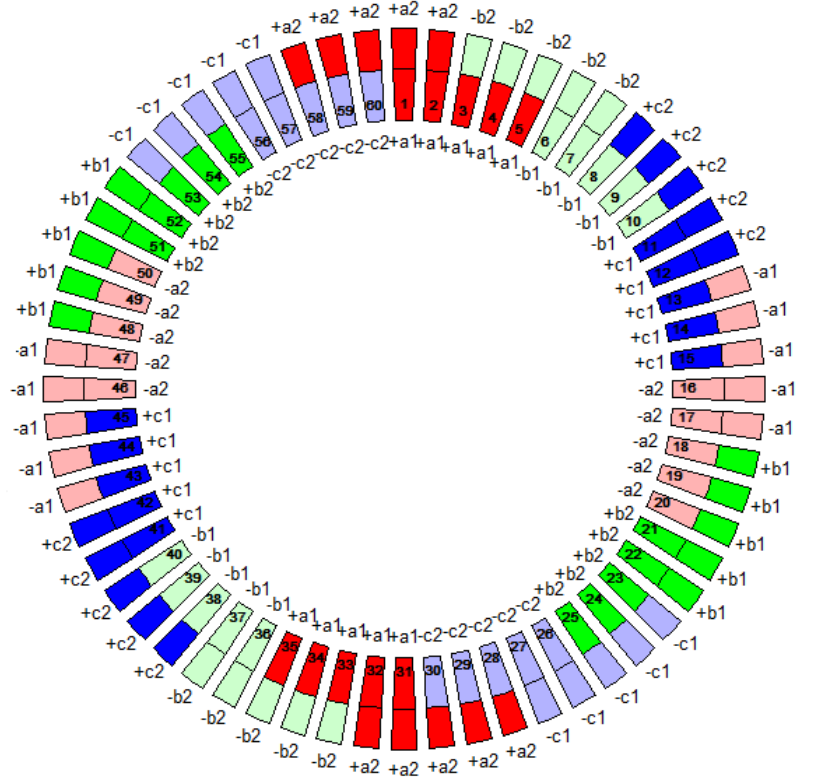



Figure 4.14. Example of the winding layout file and corresponding layout picture.

The example of the winding layout loaded from file `layoutfile_example.txt` is shown in Figure 4.14. The layout file should include layout for each phase. First column of the layout file is a coil number as shown in Figure 4.14, second column is a 'from' slot number, third column – winding layer number of the slot 'from', forth column - 'to' slot number, fifth column – winding layer number of the slot 'to', and the sixth column is the parallel path number the coil belongs to. By default, the first winding layer is placed at the bottom of the slot as shown in Figure 4.11 (bottom of the slot is placed farther from the air gap). When the stator geometry is imported from the DXF-file the winding layer numbers are specified by the user i.e. layers can have any position within a slot. Windings having more than two layers are also allowed. To use the winding with more than two layers the stator geometry should be imported from DXF-file and the winding layout should be loaded from file.

LCM of slot number and pole number specifies the least common multiple (LCM) between the number of slots and number of poles. It is equal to the number of periods of cogging torque waveform per rotor revolution. LCM should be as high as possible to minimize cogging torque.

GCD of slot number and pole number specifies the greatest common divisor (GCD) between the number of slots and number of poles. It should be an even number and as high as possible to ensure a better balance of radial forces with reduced noise and motor vibrations.

Fundamental winding factor is a ratio of flux linked by the winding compared to flux linked by a single-layer full-pitch non-skewed integer-slot winding with the same number of turns and one single slot per pole per phase. Fundamental winding factor should be as high as possible (close to 1) to maximize torque.

Use button  on the right to see harmonic winding factors.

End winding axial overhang specifies the distance between the lamination stack and the outside of the windings as shown in Figure 4.15. This parameter can be calculated automatically (if / **Auto** is chosen) or defined manually.

End winding inductance specifies the leakage inductance of the stator winding end-turns per phase. End winding inductance can be calculated automatically (if / **Auto** is chosen) based on the winding configuration and machine dimensions or defined manually.

Phase resistance specifies the active DC resistance of the stator winding per phase for the temperature specified in the **Materials** panel. Winding phase resistance is automatically adjusted for the given temperature according to the conductor material temperature coefficient of resistance (see section 4.2). Phase resistance can be calculated automatically (if / **Auto** is chosen) based on the winding configuration and machine dimensions or defined manually.

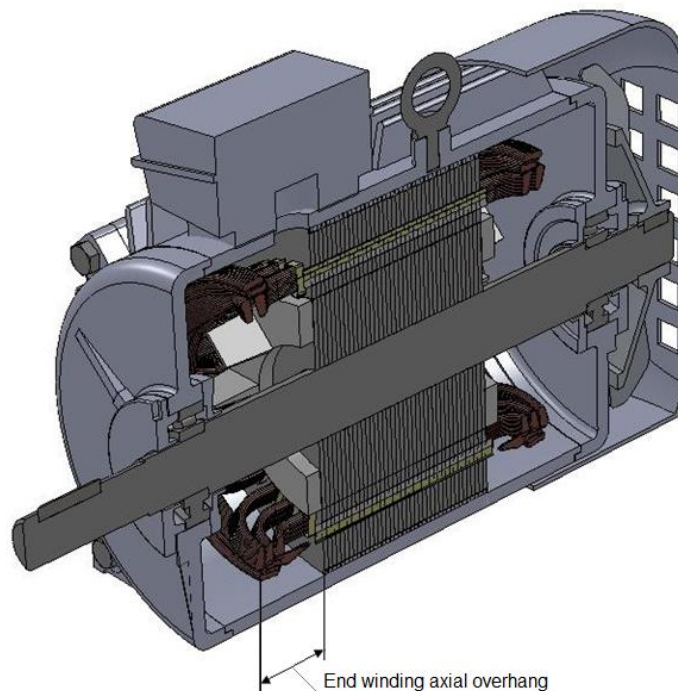


Figure 4.15. End winding axial overhang.

4.4. Mesh Editor.

Mesh Editor tab of MotorXP Design Studio is shown in Figure 4.16. It is recommended to examine the finite element mesh before starting the analysis. Quality of the mesh influences the simulation result accuracy. To enhance the accuracy of the simulation results the triangles of the mesh should be close to an equilateral triangle as much as possible.

Maximum triangle side limits the length of the longest triangle side to the specified value. If / **Auto** is chosen, the **Maximum triangle side** value is set up to 10% of the outer diameter of the machine.

Number of slices field specifies the number of slices used by the multi-slice FEM, i.e. the number of the machine's cross-sections in which the magnetic field is simultaneously calculated. For more details on the multi-slice FEM refer to section 2.4 of this manual.

Number of layers in air gap specifies the total number of finite element layers placed in the air gap. Only odd numbers of finite element layers from 1 to 9 are allowed (refer to section 2.2 of this manual for more details).

Air gap mesh quality specifies the quality of the mesh in the air gap region (*Low*, *Medium* or *High*). The mesh of the highest quality would consist of equilateral triangles which is practically not possible for complex geometry. The lower air gap mesh quality usually leads to the smaller number of mesh triangles and faster computational speed sacrificing the accuracy.

Refine mesh in air gap is recommended to be always selected unless very coarse mesh is required to speed up the simulation.

You can apply periodic or antiperiodic boundary conditions choosing corresponding item from the **Boundary conditions** pop-up menu. If *None* is chosen from the **Boundary conditions** pop-up menu, the boundary conditions are not used. If / **Auto** is chosen, the best possible boundary conditions are automatically assigned. For more details on boundary conditions refer to section 2.2 of this manual.

The **Mesh statistics** parameters such as the number of mesh vertices, edges and triangles are calculated for one slice. To calculate the total number of mesh vertices, edges, and triangles one should multiply these values by the number of slices.

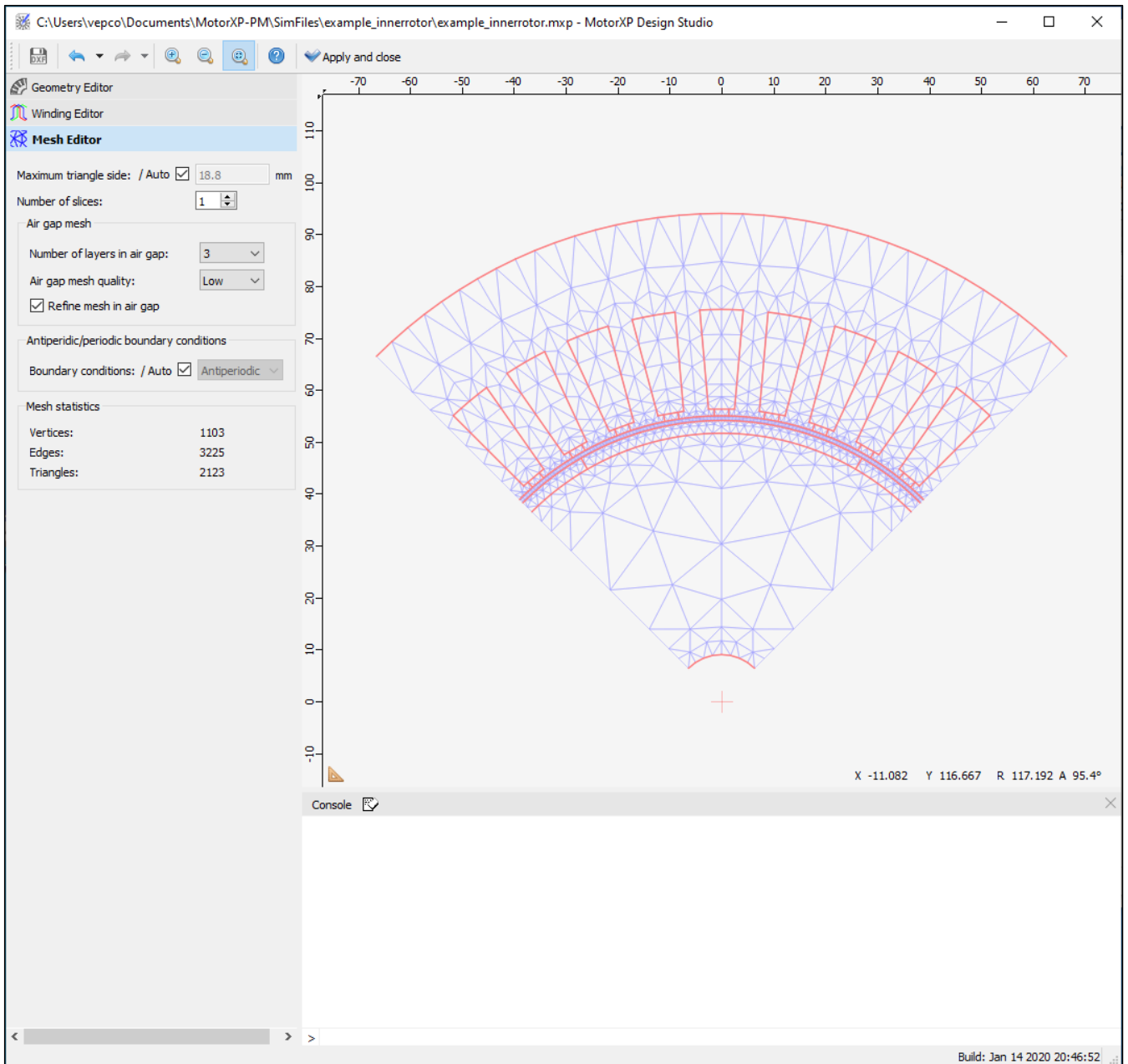


Figure 4.16. Mesh Editor of MotorXP Design Studio.

4.5. Drive Settings.

There are three drive types available in MotorXP-PM (*Current hysteresis PWM*, *Space vector PWM* and *Six-step*) which can be chosen in the **Drive Settings** window as shown in Figures 4.18 – 4.22. It is assumed that the machine is supplied from the three-phase inverter shown in Figure 4.17 (for **Dynamic FE Analysis** the inverter circuit also includes diodes as shown in Figure 10.1):

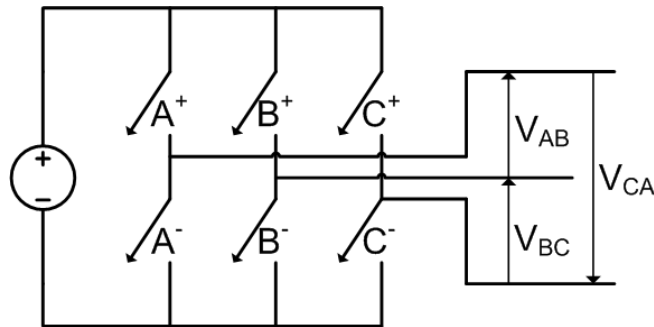


Figure 4.17. Three-phase inverter.

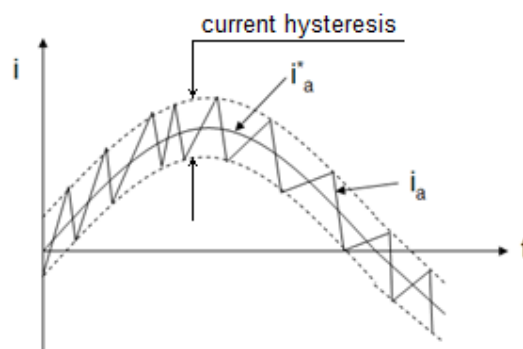
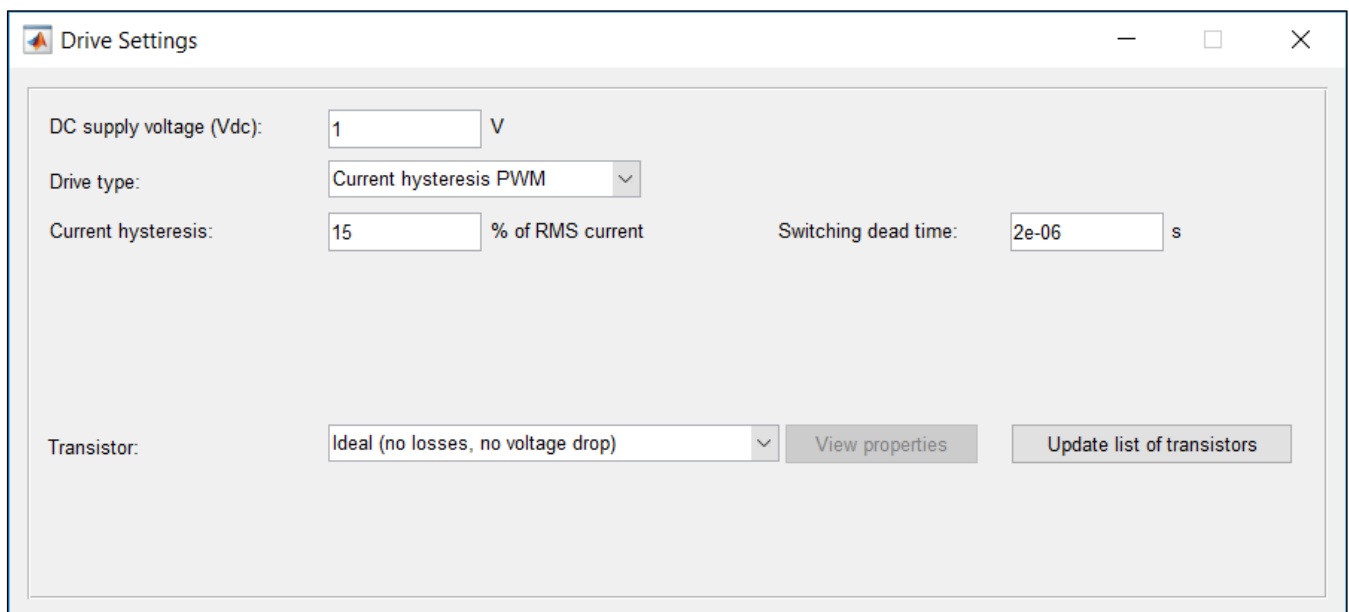


Figure 4.18. Current hysteresis PWM with the ideal transistor chosen.

The *Current hysteresis PWM* drive type is shown in Figure 4.18. The phase current is kept within the upper and lower bounds defined by the **Current hysteresis** field value.

The **Switching dead time** value normally varies from 0 to 5 microsecond. The switching dead time defines the small amount of time between switching edges for top and bottom switches of the same phase so during this time both switches are off. In the real inverter the switching dead time helps to avoid “short through” current between top and bottom switches because of the switch-off lag of the transistor. In MotorXP-PM the switching dead time is taken into account in the dynamic D-Q simulations (see chapter 7 for more information on **Dynamic D-Q Analysis**); no dead time compensation methods are applied. The switching dead time is not included into standard simulation scripts of **Dynamic FE Analysis** (*simscrip_hystpwm.m*, *simscrip_spacevecpwm.m*, *simscrip_sixstep.m*), though the dead time can be implemented in the user defined simulation script (see chapters 8 and 9 for more information on **Dynamic FE Analysis** and simulation scripts).

The **Drive Settings** window when the *Space vector PWM* drive type is chosen is shown in Figure 4.19.

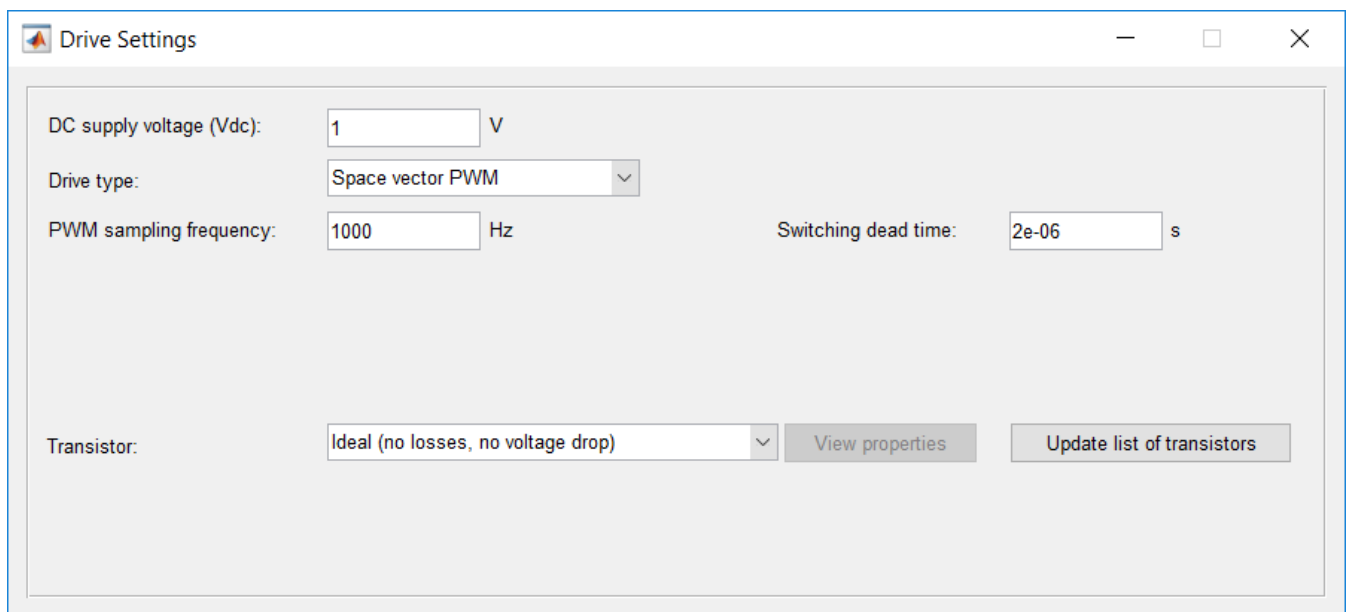
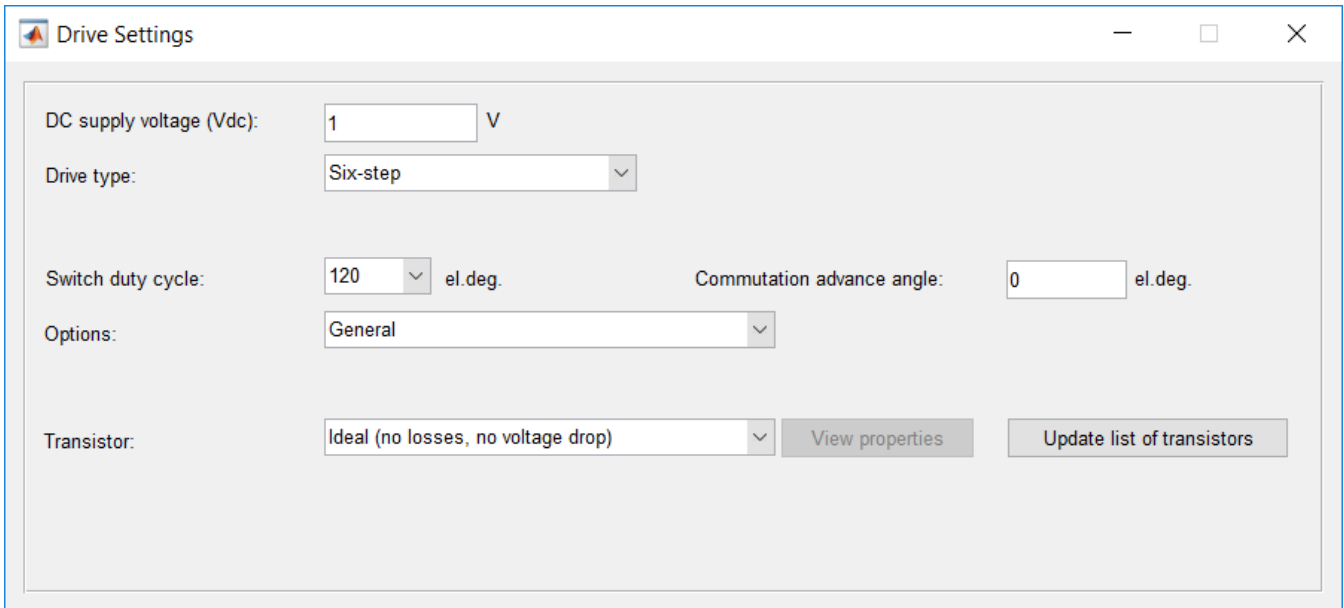


Figure 4.19. Space vector PWM with the ideal transistor chosen.

The **Drive Settings** window when the *Six step* drive type is chosen is shown in Figures 4.20 – 4.22.

Switch duty cycle specifies the time in electrical degrees when the switch is turned on. **120** degrees switch duty cycle corresponds to the inverter operation with two switches turned on at the same time; with **180** degrees switch duty cycle there will be three switches turned on at the same time.

In practice, the phase inductances and the non-ideal current commutations will cause a phase lag of the current with respect to the voltage. The lag of the current can be compensated by imposing the **Commutation advance angle** shifting forward the commutation timing.



Drive Settings

DC supply voltage (Vdc): 1 V

Drive type: Six-step

Switch duty cycle: 120 el.deg. Commutation advance angle: 0 el.deg.

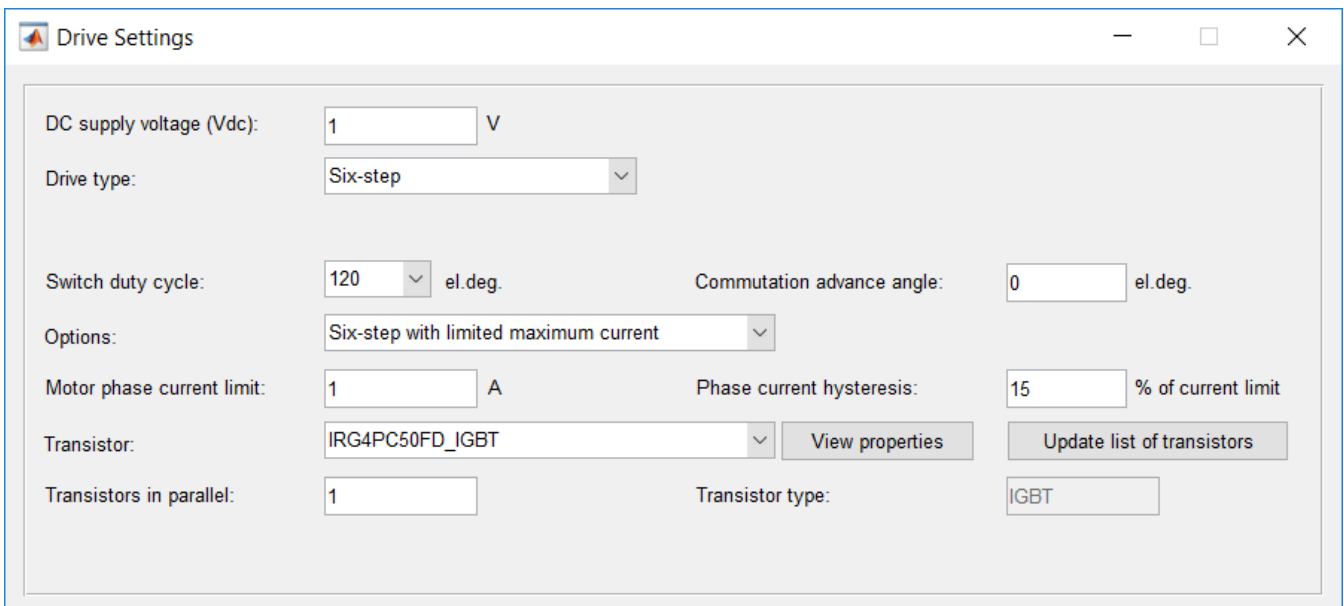
Options: General

Transistor: Ideal (no losses, no voltage drop) View properties Update list of transistors

Figure 4.20. Drive Settings window with six-step drive type and ideal transistor chosen.

There are two additional options for the six-step drive: *Six-step with limited maximum current* and *Six-step with variable DC voltage*.

When the six-step with limited maximum current is used (Figure 4.21), all switches are turned off once the current of any phase reaches the **Motor phase current limit**. The corresponding switches are turned on again when the maximum phase current becomes lower than the current limit minus the value specified by the **Phase current hysteresis** field.



Drive Settings

DC supply voltage (Vdc): 1 V

Drive type: Six-step

Switch duty cycle: 120 el.deg. Commutation advance angle: 0 el.deg.

Options: Six-step with limited maximum current

Motor phase current limit: 1 A Phase current hysteresis: 15 % of current limit

Transistor: IRG4PC50FD_IGBT View properties Update list of transistors

Transistors in parallel: 1 Transistor type: IGBT

Figure 4.21. Six-step drive with limited maximum current and IGBT transistor chosen.

When the six-step with variable DC voltage is used (Figure 4.22), the input DC voltage of the inverter is modulated so only the part of the battery voltage is used specified by the **Vdc usage percentage** field value. Make sure that the **PWM sampling frequency** is high enough not to interfere with the six-step commutation frequency.

The screenshot shows the 'Drive Settings' window with the following parameters:

- DC supply voltage (Vdc): 1 V
- Drive type: Six-step
- PWM sampling frequency: 1000 Hz
- Switch duty cycle: 120 el.deg.
- Commutation advance angle: 0 el.deg.
- Options: Six-step with variable DC voltage
- Vdc usage percentage: 100 % of Vdc
- Transistor: IAUT300N08S5N012_MOSFET
- Transistors in parallel: 1
- MOSFET driver resistor: 3.5 Ohm
- Buttons: View properties, Update list of transistors

Figure 4.22. Six-step drive with variable DC voltage and MOSFET transistor chosen.

When the *ideal (no losses, no voltage drop)* item is chosen from the **Transistor** pop-up menu (as shown in Figures 4.18 – 4.20), no inverter losses are calculated, and no transistor on-state voltage drop is included in simulation. There are two transistor types in MotorXP-PM supported at the moment for the inverter losses calculation: MOSFET and IGBT. Figure 4.21 shows the **Drive Settings** window when the transistor of IGBT type is chosen from the **Transistor** pop-up menu. The **Transistors in parallel** field, when specified greater than 1, allows to distribute the load current across the several transistors connected in parallel so the current of each individual transistor does not exceed the transistor limit, which also taken into account while the transistor losses are calculated. The **Drive Settings** window when the transistor of MOSFET type is chosen is shown in Figure 4.22. The **MOSFET driver resistor** field specifies the gate drive circuit external resistor value which influences the MOSFET switching losses. The inverter losses are only calculated with **Steady State D-Q Analysis** (see chapter 6) and **Dynamic D-Q Analysis** (see chapter 7). General information about the inverter losses calculation can be found in section 2.8.

4.5.1. Adding user defined transistors.

The properties of each transistor are specified in the separate txt-file of a special format stored in *[User data directory]\Materials\Transistor*. You can add your own transistor or modify properties of the

existing one. The format of the transistor property file is the same as the format of the material property file described in section 4.3.1.

Each property is designated by a *property tag*, the name of the tag is enclosed inside the angle brackets: <Drain-source resistance>, <Diode characteristic> and etc. The full list of properties of the transistor is given in Table 4.1 for MOSFET and in Table 4.2 for the transistor of IGBT type. The tag is followed by the *property content* up to the next property tag or the end of the file. If the property is not specified it means that the corresponding property tag is followed by empty line(s) up to the next property tag or the end of the file. The content of a line following the percent sign “%” and after the property tag is ignored and treated as a comment.

4.5.1.1. Adding transistor of MOSFET type.

The parameters of the MOSFET specified in the property file are listed in Table 4.1.

Table 4.1. MOSFET parameters.

MOSFET parameter		Symbol [units]	Property file tag
Drain-to-source on-state resistance vs. drain current		R_{DSon} [mOhm] vs. I_D [A]	<Drain-source resistance>
Gate drive circuit output voltage		V_{Dr} [V]	<Driver output voltage>
Option 1	Gate Muller plateau voltage	$V_{plateau}$ [V]	<Plateau voltage>
	Reverse transfer (gate-to-drain) capacitance vs. drain-to-source voltage	C_{rss} [pF] vs. V_{DS} [V]	<Gate-drain capacitance>
	MOSFET drain current rise time	t_{ri} [ns]	<Rise time>
	MOSFET drain current fall time	t_{fi} [ns]	<Fall time>
Option 2	Turn-on MOSFET switching losses vs. on-state drain current	E_{on} [mJ] vs. I_D [V]	<Switch-on loss>
	Turn-off MOSFET switching losses vs. on-state drain current	E_{off} [mJ] vs. I_D [V]	<Switch-off loss>
Freewheeling diode reverse recovery charge		Q_{rr} [nC]	<Diode reverse recovery charge>
Freewheeling diode current vs. diode voltage		I_f [A] vs. V_f [V]	<Diode characteristic>

The process of creating the MOSFET property file is demonstrated on the examples of the IAUT300N08S5N012 transistor and SiC MOSFET transistor FF11MR12W1M1_B11, the corresponding property files can be found in [User data directory]\Materials\Transistor\ (files IAUT300N08S5N012_MOSFET.txt and FF11MR12W1M1_B11_MOSFET.txt) and the corresponding datasheet files can be found in [User data directory]\Materials\Transistor\datasheet\ (files IAUT300N08S5N012.pdf and Infineon-FF11MR12W1M1_B11-DS-v02_02-EN.pdf).

Drain-to-source on-state resistance vs. drain current characteristic $R_{DSon} = f(I_D)$ is read from the datasheet diagram as shown in Figure 4.23. At least two points are required. Web-based tool <https://automeris.io/WebPlotDigitizer> can be used to extract data from the diagram. As can be seen from Figure 4.23 the $R_{DSon} = f(I_D)$ characteristic used for the property file corresponds to the gate-to-source

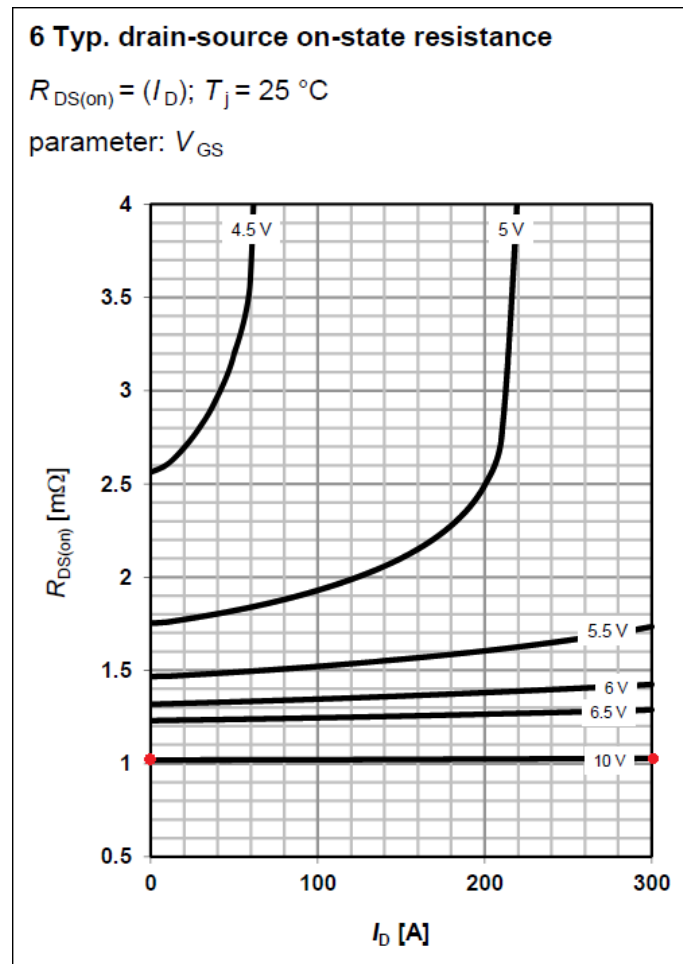
voltage $V_{GS} = 10V$ equal to the MOSFET gate driver output voltage. The R_{DSon} diagram shown in Figure 4.23 refers to the junction temperature $T_j = 25^{\circ}C$. While this should be sufficient for the majority of applications, most datasheets also provide the R_{DSon} variation diagram versus temperature so the R_{DSon} values for the expected junction temperature can be calculated.

There are two options for adding MOSFET transistor switching losses to the property file depending on the data provided by the manufacturer in the datasheet (see Table 4.1). Option 1 (on the example of IAUT300N08S5N012 transistor): using Gate Muller plateau voltage, reverse transfer capacitance vs. drain-to-source voltage diagram and MOSFET drain current rise and fall time; Option 2 (on the example of FF11MR12W1M1_B11 transistor): using turn-on and turn-off MOSFET switching losses vs. on-state drain current diagrams (see Table 4.1). Either of these two groups of parameters can be presented in the MOSFET transistor property file.

The gate Muller plateau voltage $V_{plateau}$ can be read from the MOSFET datasheet table or extracted from the typical gate charge vs. gate-to-source voltage diagram (whichever of them is available; see Figure 4.24). The plateau voltage $V_{plateau} = 4.5V$ appears in the MOSFET property file as shown in Figure 4.25. Reverse transfer (gate-to-drain) capacitance vs. drain-to-source voltage characteristic $C_{rss} = f(V_{DS})$ is read from the datasheet diagram as shown in Figure 4.25. At least two points are required. Figure 4.26 and Figure 4.27 show reading the current rise time and fall time and the diode reverse recovery charge from the datasheet.

Turn-on and turn-off MOSFET switching energy losses vs. on-state drain current can be read from the datasheet diagrams as shown in Figure 4.28 for the FF11MR12W1M1_B11 transistor.

Reading the freewheeling diode current vs. diode voltage characteristic $I_f = f(V_f)$ is shown in Figure 4.29.



```

<Drain-source resistance> Drain-to-source on-state resistance vs. drain current
% Id[A]      Rds_on[mOhm]          @ Tj=25*C, Vgs=10V
0            1
300          1

<Driver output voltage>
10  % [V]

```

Figure 4.23. Drain-to-source on-state resistance vs. drain current diagram from the datasheet (IAUT300N08S5N012) and the corresponding part of the MOSFET property file.

Gate Charge Characteristics²⁾

Gate to source charge	Q_{gs}	$V_{DD}=40\text{ V}, I_D=100\text{ A},$ $V_{GS}=0\text{ to }10\text{ V}$	-	56	73	nC
Gate to drain charge	Q_{gd}		-	37	56	
Gate charge total	Q_g		-	178	231	
Gate plateau voltage	$V_{plateau}$		-	4.5	-	V

15 Typ. gate charge
 $V_{GS} = f(Q_{gate}); I_D = 300\text{ A pulsed}$

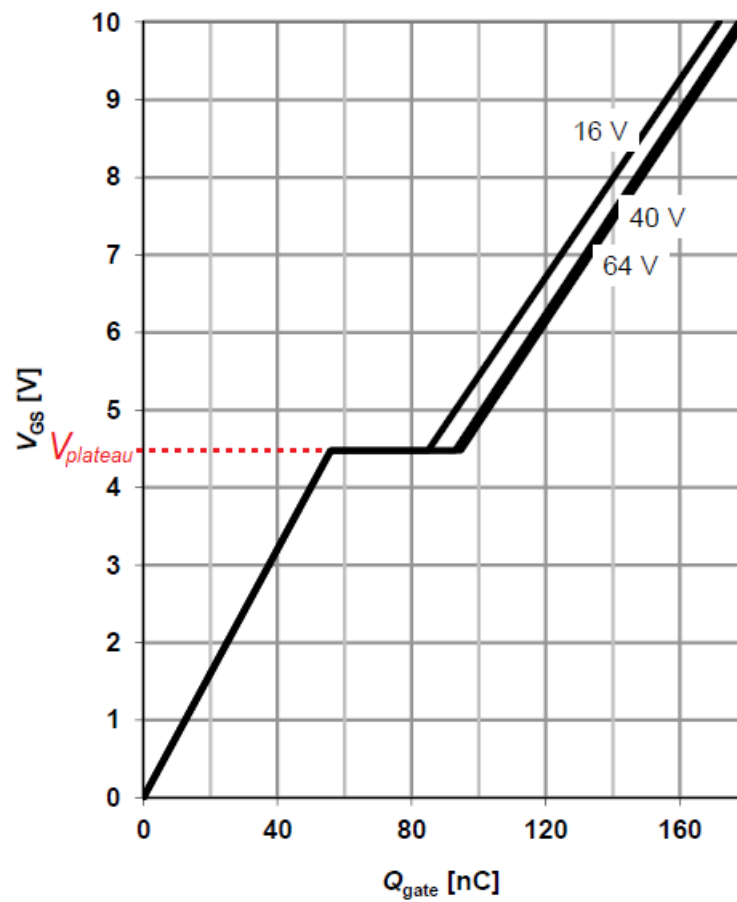
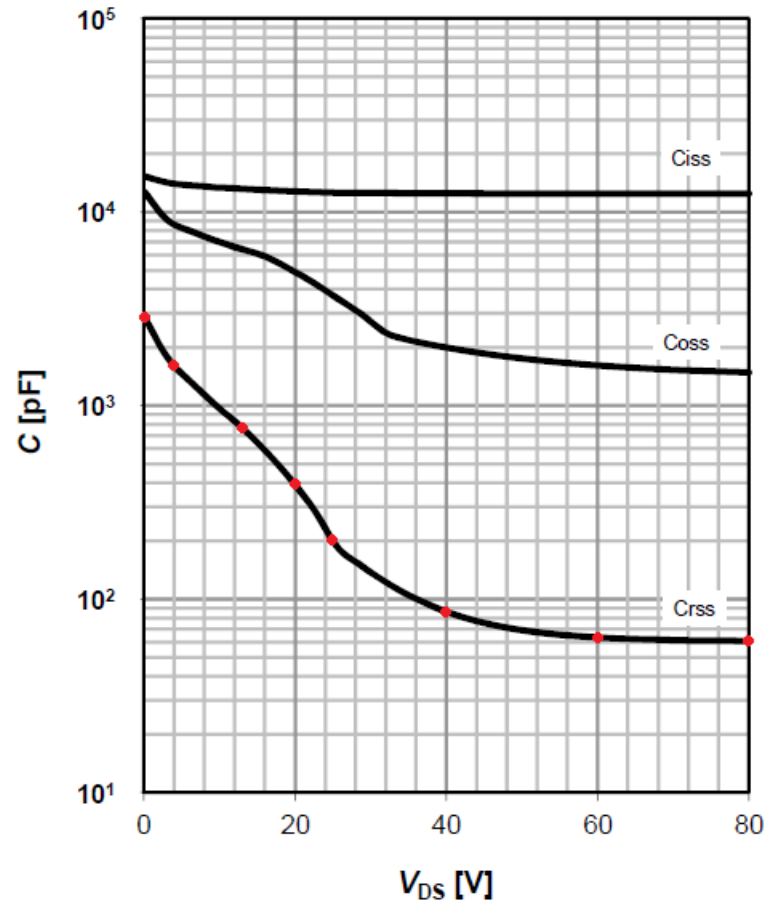
 parameter: V_{DD}


Figure 4.24. Two ways to determine the gate Muller plateau voltage $V_{plateau}$ from the MOSFET datasheet (IAUT300N08S5N012).

10 Typ. capacitances

$C = f(V_{DS}); V_{GS} = 0 \text{ V}; f = 1 \text{ MHz}$



<Plateau voltage>

4.5 % [V]

<Gate-drain capacitance>

% Reverse transfer capacitance vs. drain-source voltage

% vds[V] Crss[pF]

0 2832

4 1610

13 765

20 398

25 202

40 88

60 64

80 62

Figure 4.25. Reverse transfer (gate-to-drain) capacitance vs. drain-to-source voltage diagram from the datasheet (IAUT300N08S5N012) and the corresponding part of the MOSFET property file.

Dynamic characteristics ²⁾						
Input capacitance	C_{iss}	$V_{GS}=0\text{ V}, V_{DS}=40\text{ V},$ $f=1\text{ MHz}$	-	12500	16250	pF
Output capacitance	C_{oss}		-	2000	2600	
Reverse transfer capacitance	C_{rss}		-	86	130	
Turn-on delay time	$t_{d(on)}$	$V_{DD}=40\text{ V}, V_{GS}=10\text{ V},$ $I_D=100\text{ A}, R_G=3.5\ \Omega$	-	31	-	ns
Rise time	t_r		-	19	-	
Turn-off delay time	$t_{d(off)}$		-	69	-	
Fall time	t_f		-	55	-	

```
<Rise time>
19      % [ns]
```

```
<Fall time>
55      % [ns]
```

Figure 4.26. Reading the current rise time and fall time from the datasheet (IAUT300N08S5N012) and the corresponding part of the MOSFET property file.

Reverse Diode						
Diode continuous forward current ²⁾	I_S	$T_C=25\text{ }^\circ\text{C}$	-	-	300	A
Diode pulse current ²⁾	$I_{S,pulse}$		-	-	1200	
Diode forward voltage	V_{SD}	$V_{GS}=0\text{ V}, I_F=100\text{ A},$ $T_J=25\text{ }^\circ\text{C}$	-	0.9	1.2	V
Reverse recovery time ²⁾	t_{rr}	$V_R=40\text{ V}, I_F=50\text{ A},$ $di_F/dt=100\text{ A}/\mu\text{s}$	-	86	-	ns
Reverse recovery charge ²⁾	Q_{rr}		-	177	-	nC

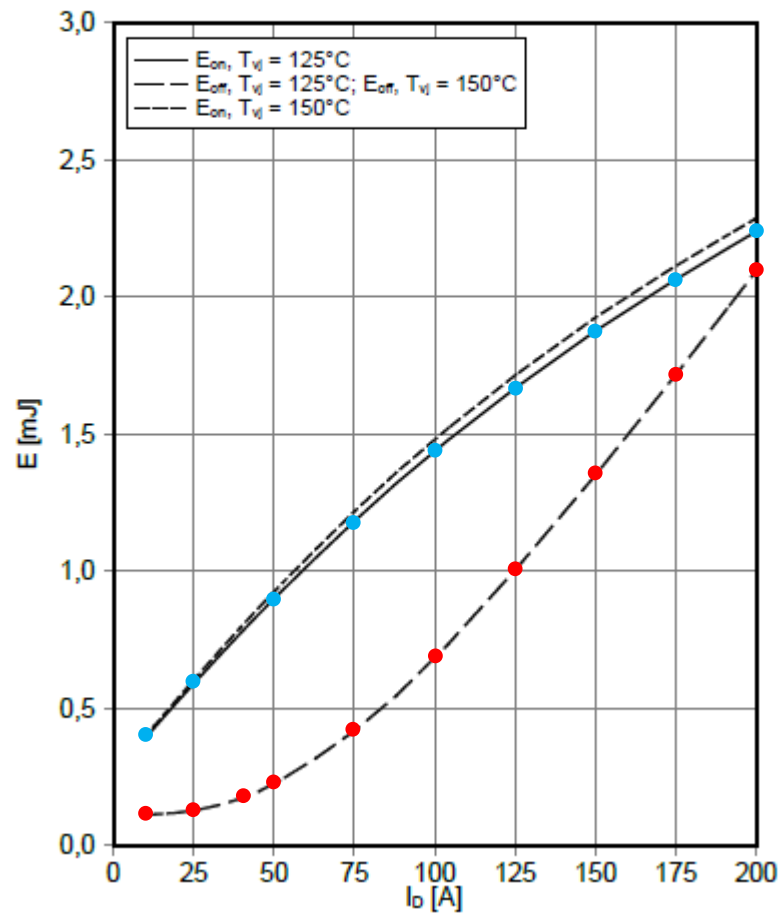
```
<Diode reverse recovery charge>
177      % [nC]
```

Figure 4.27. Reading the diode reverse recovery charge from the datasheet (IAUT300N08S5N012) and the corresponding part of the MOSFET property file.

Schaltverluste MOSFET (typisch)
switching losses MOSFET (typical)

$E_{on} = f(I_D)$, $E_{off} = f(I_D)$

$V_{GS} = -5\text{ V} / 15\text{ V}$, $R_{Gon} = 3,9\ \Omega$, $R_{Goff} = 3,9\ \Omega$, $V_{DS} = 600\text{ V}$



<Switch-on loss> @ Tj=125°C	
% Id[A]	Eon[mJ]
0	0
10.174	0.39770
25.000	0.59207
49.855	0.89898
74.709	1.1803
100.00	1.4361
124.85	1.6662
150.15	1.8760
175.00	2.0652
199.85	2.2391

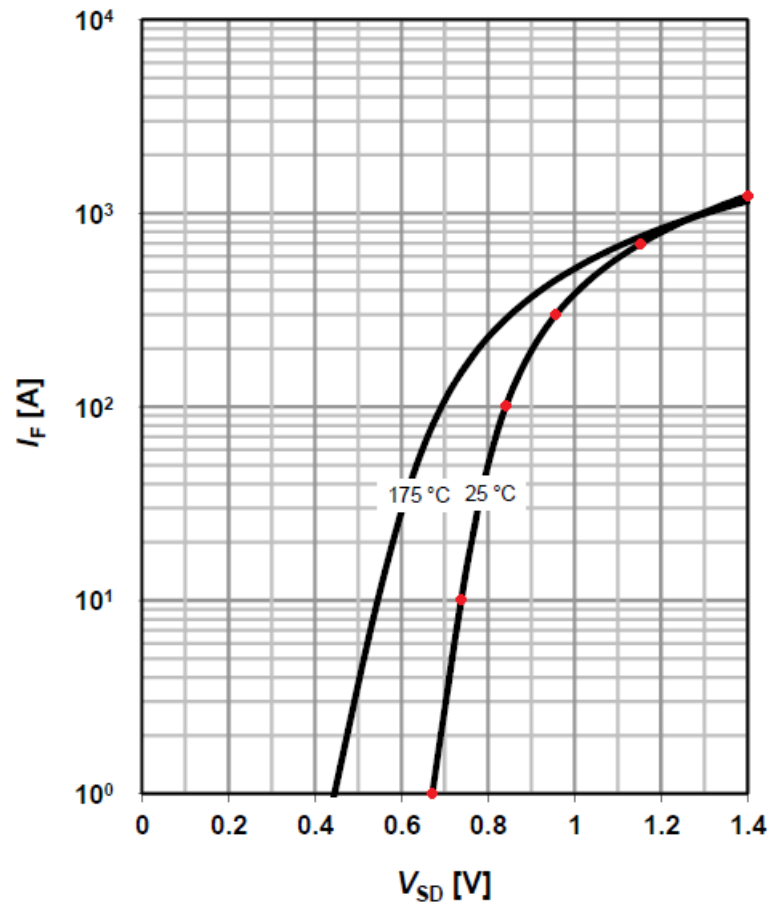
<Switch-off loss> @ Tj=125°C	
% Id[A]	Eoff[mJ]
0	0
10.174	0.10614
25.000	0.12660
39.390	0.16752
49.855	0.22379
75.145	0.41816
100.00	0.68414
124.85	1.0013
149.71	1.3491
175.00	1.7174
199.85	2.0959

Figure 4.28. Turn-on and turn-off MOSFET switching energy losses vs. drain current diagrams from the datasheet (FF11MR12W1M1_B11) and the corresponding part of the MOSFET property file.

11 Typical forward diode characteristics

$$I_F = f(V_{SD})$$

parameter: T_j



<Diode characteristic> @ $T_j=25^{\circ}\text{C}$	
% Vf[V]	If[A]
0.67	1
0.74	10
0.84	100
0.96	298
1.16	698
1.4	1241

Figure 4.29. Freewheeling diode current vs. diode voltage diagram from the datasheet (IAUT300N08S5N012) and the corresponding part of the MOSFET property file.

4.5.1.2. Adding transistor of IGBT type.

The parameters of the IGBT specified in the property file are listed in Table 4.2.

Table 4.2. IGBT parameters.

IGBT parameter	Symbol [units]	Property file tag
On-state collector current vs. collector-to-emitter voltage	I_C [A] vs. V_{CE} [V]	<Output characteristic>
Freewheeling diode current vs. diode voltage	I_f [A] vs. V_f [V]	<Diode characteristic>
Turn-on IGBT switching energy losses vs. on-state collector current	E_{on} [mJ] vs. I_C [V]	<Switch-on loss>
Turn-off IGBT switching energy losses vs. on-state collector current	E_{off} [mJ] vs. I_C [V]	<Switch-off loss>
Freewheeling diode switching energy losses vs. diode current *	E_{rec} [mJ] vs. I_f [V]	<Diode switch-on loss>
Freewheeling diode reverse recovery charge *	Q_{rr} [nC]	<Diode reverse recovery charge>

* These parameters are not necessarily required for the property file. Only one of these parameters <Diode switch-on loss> or <Diode reverse recovery charge> can be specified in the property file to calculate the freewheeling diode switching losses. If none of these parameters are specified in the property file, the freewheeling diode switching losses will not be included into the inverter losses calculation (diode switching losses are usually much less compared with the transistor switching losses).

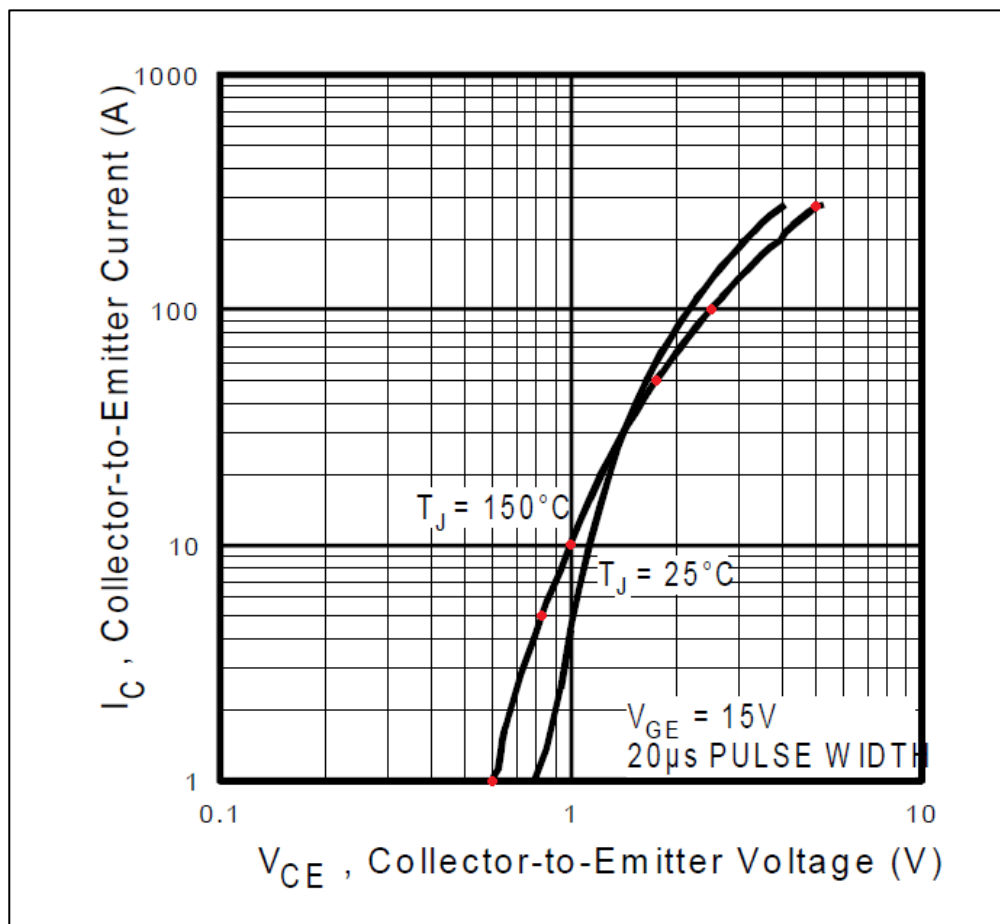
The process of creating the IGBT property file is demonstrated on the example of the IRG4PC50FD and IXGH24N60AU1 transistors, the corresponding property files can be found in [User data directory] \Materials\Transistor\ and the corresponding datasheets can be found in [User data directory] \Materials\Transistor\datasheet\.

On-state collector current vs. collector-to-emitter voltage characteristic $I_C = f(V_{CE})$ and the freewheeling diode current vs. diode voltage characteristic $I_f = f(V_f)$ are read from the datasheet diagrams as shown in Figure 4.30 and Figure 4.31, respectively. At least two points are required. Web-based tool <https://automeris.io/WebPlotDigitizer> can be used to extract data from the diagram.

Turn-on and turn-off IGBT switching energy losses can also be read from the datasheet diagrams as shown in Figure 4.32 for the FF450R12KT4 transistor. Some manufacturers do not provide turn-on and turn-off losses diagrams in the datasheet though the required information can still be extracted from the datasheet data. One example is shown in Figure 4.33 for the IRG4PC50FD transistor where only the total IGBT switching losses vs. collector current diagram is provided in the datasheet. Assuming that the correlation between the turn-on and turn-off energy losses (E_{on}/E_{off} ratio) does not depend on the collector current and the junction temperature and using the specific turn-on and turn-off switching losses values from the datasheet (see Figure 4.33) the turn-on and turn-off IGBT losses vs. collector current characteristics have been restored and recorded in the property file. Another example is shown in Figure 4.34 for the IXGH24N60AU1 transistor where only the turn-off losses E_{off} diagram is available

from the datasheet. The turn-on IGBT losses E_{on} vs. collector current characteristic has been restored using the $E_{on}/E_{off} = 0.8/2.3$ ratio from the datasheet data (see Figure 4.34).

If the freewheeling diode switching energy losses vs. diode current diagram is provided in the datasheet (see Figure 4.35 for the FF450R12KT4 transistor) when this characteristic can be directly specified in the IGBT property file for the diode switching calculation. Otherwise, the diode reverse recovery charge from the datasheet can be used instead as shown in Figure 4.36 for the IRG4PC50FD transistor. If none of these parameters are specified in the property file, the freewheeling diode switching losses will not be included into the inverter losses calculation (diode switching losses are usually much less comparing with the transistor switching losses).



<Output characteristic> @ Tj=150°C, Vge=15V	
% Vce[V]	Ic[A]
0.60	1
0.83	5
0.99	10
1.75	50
2.50	103
5.00	277

Figure 4.30. On-state collector current vs. collector-to-emitter voltage diagram from the datasheet (IRG4PC50FD) and the corresponding part of the IGBT property file.

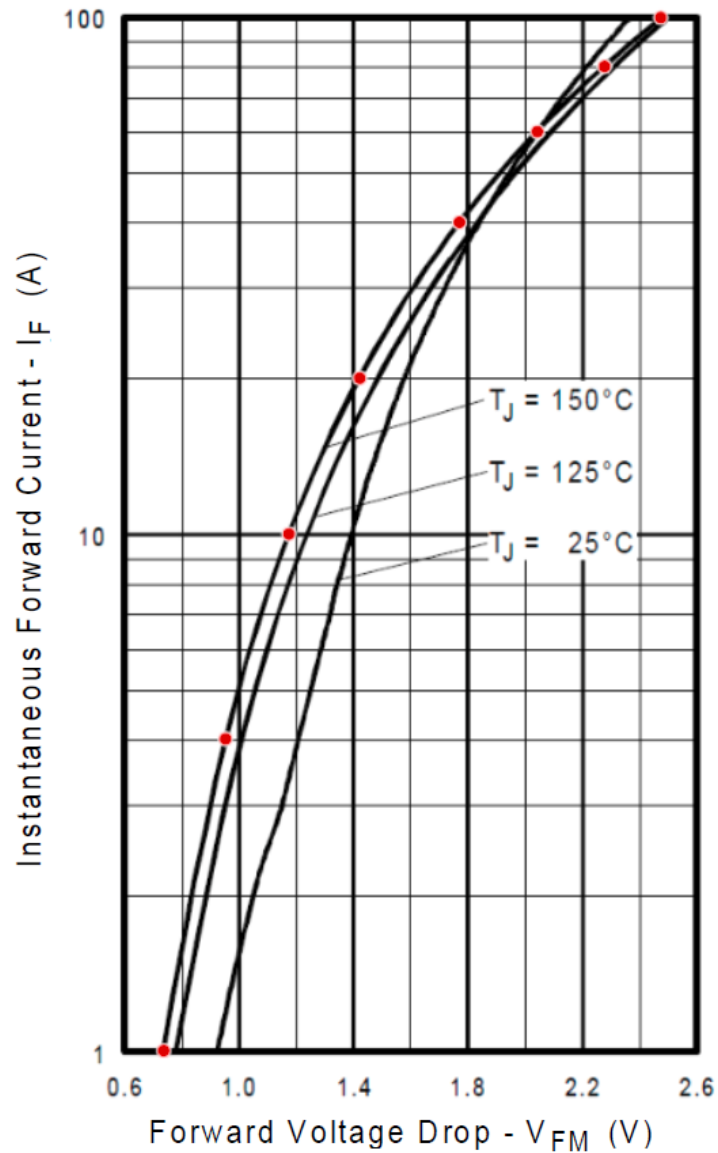


Fig. 13 - Maximum Forward Voltage Drop vs. Instantaneous Forward Current

<Diode characteristic>		@ $T_J = 150^\circ\text{C}$
% V_f [V]	I_f [A]	
0.73	1	
0.95	4	
1.17	10	
1.42	20	
1.77	40	
2.04	60	
2.28	80	
2.47	100	

Figure 4.31. Freewheeling diode current vs. diode voltage diagram from the datasheet (IRG4PC50FD) and the corresponding part of the IGBT property file.

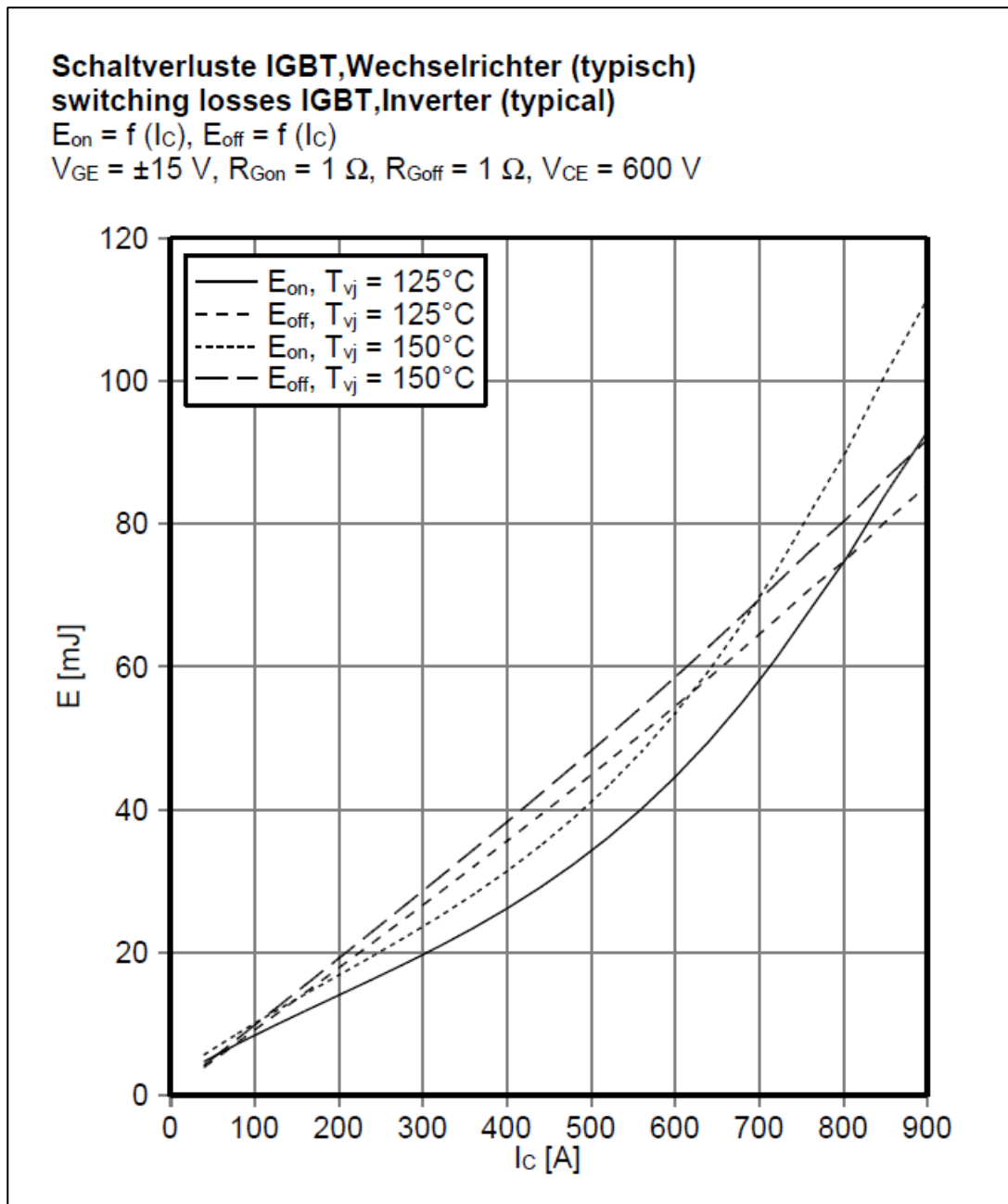
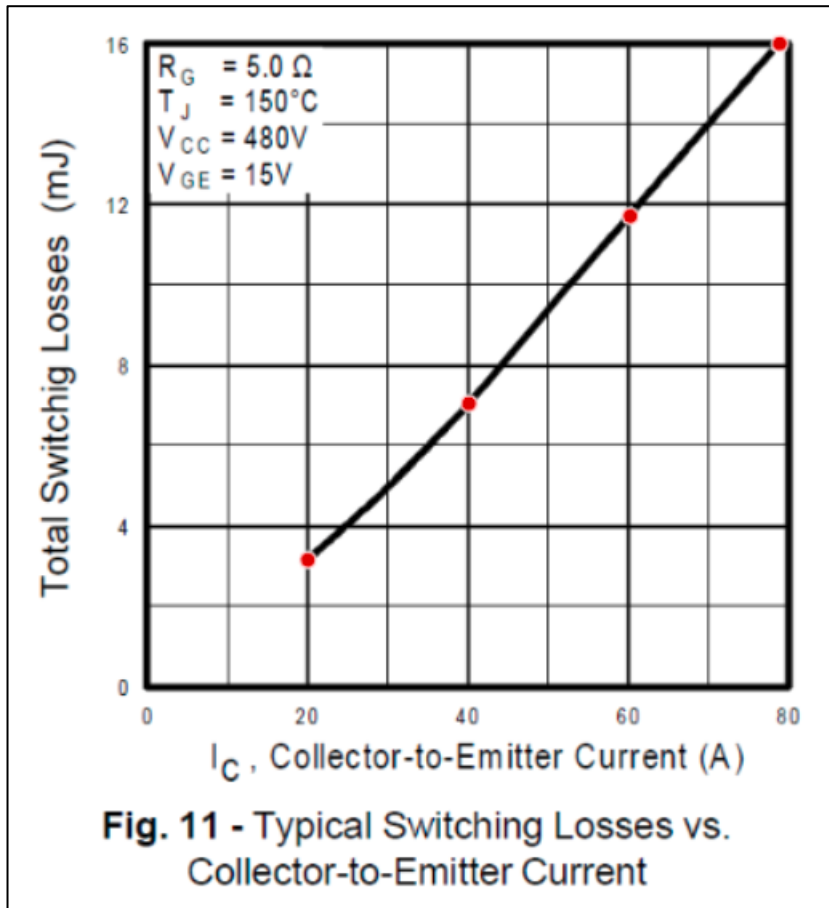


Figure 4.32. Turn-on and turn-off IGBT switching energy losses vs. collector current diagrams from the datasheet (FF450R12KT4).


Switching Characteristics @ $T_J = 25^\circ\text{C}$ (unless otherwise specified)

	Parameter	Min.	Typ.	Max.	Units	Conditions
Q_g	Total Gate Charge (turn-on)	----	190	290	nC	$I_C = 39\text{A}$ $V_{CC} = 400\text{V}$ $V_{GE} = 15\text{V}$ See Fig. 8
Q_{ge}	Gate - Emitter Charge (turn-on)	----	28	42		
Q_{gc}	Gate - Collector Charge (turn-on)	----	65	97		
$t_{d(on)}$	Turn-On Delay Time	----	55	----	ns	$T_J = 25^\circ\text{C}$ $I_C = 39\text{A}$, $V_{CC} = 480\text{V}$ $V_{GE} = 15\text{V}$, $R_G = 5.0\Omega$ Energy losses include "tail" and diode reverse recovery. See Fig. 9, 10, 11, 18
t_r	Rise Time	----	25	----		
$t_{d(off)}$	Turn-Off Delay Time	----	240	360		
t_f	Fall Time	----	140	210		
E_{on}	Turn-On Switching Loss	----	1.5	----	mJ	
E_{off}	Turn-Off Switching Loss	----	2.4	----		
E_{ts}	Total Switching Loss	----	3.9	5.0		

<Switch-on loss> @ $T_J=150^\circ\text{C}$

% I_C [A] E_{on} [mJ]

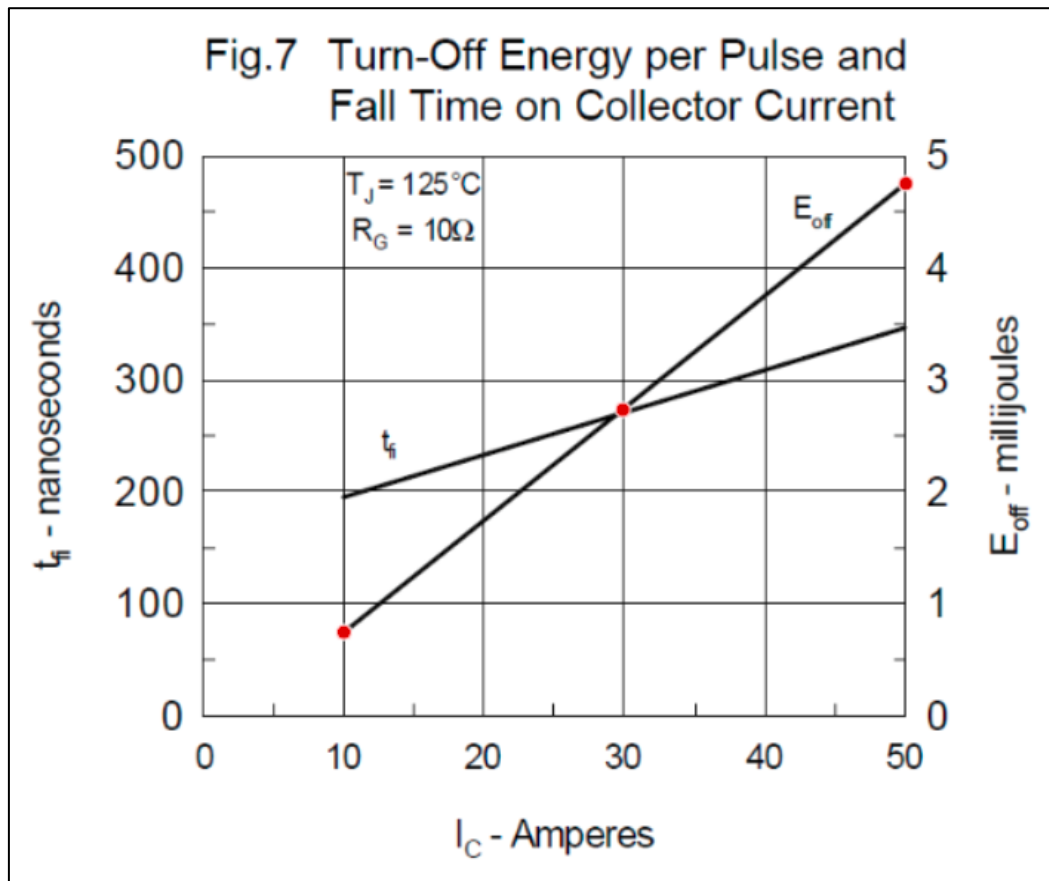
0	0
19.9	1.2
40.0	2.7
60.0	4.5
78.7	6.2

<Switch-off loss> @ $T_J=150^\circ\text{C}$

% I_C [A] E_{off} [mJ]

0	0
19.9	1.9
40.0	4.3
60.0	7.2
78.7	9.8

Figure 4.33. Example (1) of extracting the turn-on and turn-off IGBT switching energy losses vs. collector current from the datasheet (IRG4PC50FD) and corresponding part of the IGBT property file.



$t_{d(on)}$ t_{ri} E_{on} $t_{d(off)}$ t_{fi} E_{off}	Inductive load, $T_J = 125^\circ\text{C}$		
	$I_C = I_{C90}, V_{GE} = 15\text{ V}, L = 100\text{ }\mu\text{H}$		
	$V_{CE} = 0.8 V_{CES}, R_G = R_{off} = 10\text{ }\Omega$		
	Remarks: Switching times may increase for V_{CE} (Clamp) $> 0.8 \cdot V_{CES}$, higher T_J or increased R_G		
		25	ns
		15	ns
		0.8	mJ
		250	ns
		400	ns
		2.3	mJ

```

<Switch-on loss>      @ Tj=150*C
% Ic[A]      Eon[mJ]
10           0.26
30           0.95
50           1.66

<Switch-off loss>     @ Tj=150*C
% Ic[A]      Eoff[mJ]
10           0.75
30           2.74
50           4.76

```

Figure 4.34. Example (2) of extracting the turn-on and turn-off IGBT switching energy losses vs. collector current from the datasheet (IXGH24N60AU1) and corresponding part of the IGBT property file.

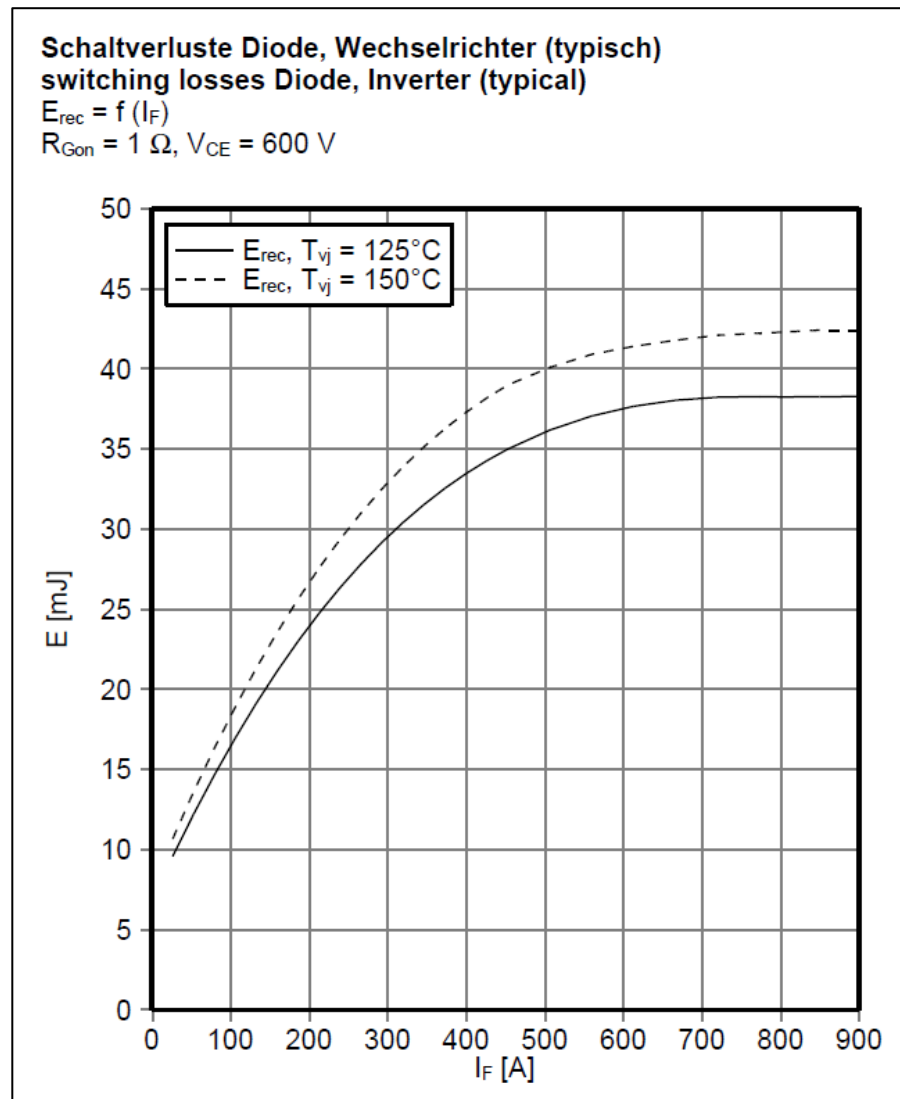


Figure 4.35. Freewheeling diode switching energy losses vs. diode current diagrams from the datasheet (FF450R12KT4).

t_{rr}	Diode Reverse Recovery Time	----	50	75	ns	$T_J = 25^\circ C$	See Fig.	$I_F = 25A$
		----	105	160		$T_J = 125^\circ C$	14	
I_{rr}	Diode Peak Reverse Recovery Current	----	4.5	10	A	$T_J = 25^\circ C$	See Fig.	$V_R = 200V$
		----	8.0	15		$T_J = 125^\circ C$	15	
Q_{rr}	Diode Reverse Recovery Charge	----	112	375	nC	$T_J = 25^\circ C$	See Fig.	$di/dt \ 200A/\mu s$
		----	420	1200		$T_J = 125^\circ C$	16	
$di_{(rec)M}/dt$	Diode Peak Rate of Fall of Recovery During t_b	----	250	----	A/ μs	$T_J = 25^\circ C$	See Fig.	
		----	160	----		$T_J = 125^\circ C$	17	

<Diode reverse recovery charge>
 1200 % [nC]

Figure 4.36. Reading the diode reverse recovery charge from the datasheet (IRG4PC50FD) and the corresponding part of the IGBT property file. Maximum value of the reverse recovery charge is used to represent the worst case of the diode switching losses.

5. MAGNETOSTATIC FINITE ELEMENT ANALYSIS

Magnetostatic FE Analysis allows the user to estimate motor parameters like voltage, current, back-EMF, power, torque, power factor, efficiency, losses and their waveforms as well as air gap and cross-section distribution plots of various quantities such as flux density, permeability, current density and etc. assuming an ideal sinusoidal or trapezoidal current waveforms.

5.1. Running Magnetostatic FE Analysis.

The view of the main window when **Magnetostatic FE Analysis** is chosen is shown in Figure 5.1. **Magnetostatic FE Analysis** consists of a series of finite element simulations (series of time steps) over one electrical period (or specified time interval) with predefined currents and rotor positions defined by the rotor speed. The current waveform is defined by the **Current waveform** field and can be either sinusoidal or trapezoidal. There are several current input methods defined by the **Current input method** field which also depend on the current waveform. For the sinusoidal current waveform, the following current input methods are available: ***RMS supply current***, ***Peak supply current*** and ***RMS current density***. When the ***RMS supply current*** input method is chosen the phase current will depend on the **RMS supply current** field value and on the stator winding connection. Since the supply current is a line current, for star connection the phase current is equal to the supply current, for delta connection the phase current is equal to the supply current divided by $\sqrt{3}$. ***Peak supply current*** input method is similar to the previous one, but peak value of supply current is used instead. When the ***RMS current density*** input method is chosen the RMS current density in stator slots can be directly defined.

For the trapezoidal current waveform, the following current input methods are available: ***RMS phase current***, ***DC supply current*** and ***RMS current density***. When the ***RMS phase current*** input method is chosen the phase current can be directly defined. When the ***DC supply current*** input method is chosen the phase current will depend on the **DC supply current** field value, stator winding connection and switch duty cycle. Switch duty cycle (120 or 180 electrical degrees) is defined in the **Drive Settings** window when six-step drive type is chosen (see section 4.4). Figure 5.2 shows an example of trapezoidal current waveforms for star and delta connection and 120 degrees switch duty cycle. Be aware that the real current waveform for the six-step drive can be significantly different from those used in **Magnetostatic FE Analysis**. When the ***RMS current density*** input method is chosen the RMS current density in stator slots can be directly defined.

The **Advance angle** field value defines the timing of the current commutation in relation to the rotor position assuming ideal current commutation and varies between 0 and 360 electrical degrees. The magnetostatic FEA simulation is performed for fixed rotor speed defined by the **Mechanical speed** field. **Number of points** defines the number of time steps per one electrical period or per the selected time interval (depending on the **Simulation time** pop-up menu item) the FEA simulation is run for. Several options for the number of points are available from the **Simulation setup** pop-up menu depending on the

required accuracy. The number of points can be automatically determined based on the number of points (time steps) per one period of cogging torque (choose between **Multiple points (2 points per cogging) - low accuracy**, **Multiple points (4 points per cogging) - medium accuracy**, **Multiple points (8 points per cogging) - high accuracy**). To consider the effect of the cogging torque on the motor performance, at least two points per cogging torque period are required. Number of periods of cogging torque waveform per rotor revolution equals to the least common multiple (LCM) of number of slots (N_s) and number of poles (N_p) so the cogging torque period can be determined as $T_{cogging} = N_p / (2f_s \cdot LCM(N_s, N_p))$, where f_s is supply frequency.



Besides the multiple points simulation setup, for the sinusoidal current waveform the **Single point (FEA + D-Q based) – fastest** item from the **Simulation setup** pop-up menu is also available which means the machine parameters are determined from the magnetostatic FEA simulation for one rotor position with the help of the D-Q representation of the machine (see section 2.5). In this case the iron losses and magnet losses are not calculated, and time plots are not available. Finally, the number of points can be directly specified if the corresponding item is chosen from the **Simulation setup** pop-up menu.

By default, the magnetostatic FEA simulation is run for one electrical period, however the specific time interval can be specified if **Define time interval** is chosen from the **Simulation time** pop-up menu.

Button ‘<-rated’ to the right of some fields allows you to set up the corresponding parameter to its rated value specified in the **Rated Data** window.

Solver type specifies either linear or nonlinear FEA is used. Using linear solver is recommended only for testing purposes. **Convergence tolerance** specifies the accuracy of FEA solution as defined by Exp. 2.5. The cogging torque is computed with zero stator current and if **RMS supply current (DC supply current)** field value is not zero when it will require one additional FEA simulation for each time step to compute cogging torque. Be aware of this fact when checking **Compute cogging torque**.

Save each field solution checkbox enables (if checked) to store additional data of each FEA solution such as magnetic vector potential distribution and permeability distribution values. This data are not stored by default because of the large amount of hard drive space required. Data for each time step is saved in a separate file so the number of files saved is equal to the **Number of points** field value.

To start the analysis, click the **Start magnetostatic simulation** button  on the MotorXP-PM main window toolbar. To stop the analysis, click the **Stop simulation** button  which is to the right of the **Start magnetostatic simulation** button. If the analysis is stopped all the simulation data of the running analysis will be lost.

Once the simulation is complete the results are displayed at the bottom of the MotorXP-PM main window. The results are divided into four groups: **General Results**, **Machine Constants**, **Flux Density Levels** and **Inductances**, and available from the corresponding pup-up menu as shown in Figure 5.1.

MotorXP-PM (Pro) - C:\Users\vepco\Documents\MotorXP-PM\SimFiles\exa... — □ ×

File Desktop About Help

Project file name: example_innerrotor.mxp Activated

D-Q Analysis **Finite Element Analysis**

Steady State Dynamic **Magnetostatic** Dynamic

Magnetostatic Finite Element Analysis

Solver type: Nonlinear Convergence tolerance: 0.001

Current waveform: Sinusoidal Advance angle: 0 (el.deg.)

Current input method: RMS supply current Mechanical speed: 1500 <-rated RPM

RMS supply current: 208 <-rated A

Simulation setup: Multiple points (8 points per cogging) - high accuracy Number of points: 144

Simulation time: One electrical period

☒ Compute cogging torque

☒ Save each field solution in folder: C:\Users\vepco\Documents\MotorXP-PM\SimFiles\example...

Results:

General Results

Rotor speed:	1500	RPM	RMS phase back-EMF:	26.3264	V
Advance angle:	0	(el.deg.)	Input electrical power:	9737.65	W
Supply frequency:	50	Hz	Output mechanical power:	9325	W
RMS phase current:	120.089	A	Efficiency:	96.3926	%
RMS phase voltage:	32.1968	V	Power factor:	0.829254	
Total torque:	59.3648	N·m	Stator winding loss:	293.869	W
Reluctance torque:	-1.13572	N·m	Total iron core loss:	41.3941	W
Magnet torque:	60.5005	N·m	Eddy current iron core loss:	1.98987	W
Torque ripple:	31.2904	%	Hysteresis iron core loss:	39.4043	W
RMS current density:	5.52143	A/mm ²	Magnet eddy current loss:	13.7148	W
Average d-axis current:	2.08602e-16	A	Other eddy current loss:	0	W
Average q-axis current:	169.831	A	Max. demag. field:	674140	A/m
Average d-axis voltage:	-24.7139	V	Max. demag. field (% of H _{cj}):	56.1783	%
Average q-axis voltage:	38.2248	V	Discretization error:	1.21985	%

Results:

Rotor speed: 1500 RPM
Advance angle: 0 (el.deg.)
Supply frequency: 50 Hz
RMS phase current: 120.089 A

Velocity constant (Kv) in [RPM/V]: 38.0779 RPM/V
Velocity constant (Kv) in [rad/(V·s)]: 3.98751 rad/(V·s)
Back-EMF constant (Ke) in [V/RPM]: 0.0262619 V/RPM
Back-EMF constant (Ke) in [V·s/rad]: 0.250783 V·s/rad
Torque constant: 0.494341 N·m/A
Motor constant (Km): 3.463 N·m/sqrt(W)

Results:

Rotor speed: 1500 RPM
Advance angle: 0 (el.deg.)
Supply frequency: 50 Hz
RMS phase current: 120.089 A

Average airgap flux density: 0.610876 T
Maximum airgap flux density: 1.23192 T
Peak of stator tooth average flux density: 1.75181 T
Peak of stator back iron average flux density: 1.40734 T
Peak of rotor back iron average flux density: 0.649914 T

Results:

Rotor speed: 1500 RPM
Advance angle: 0 (el.deg.)
Supply frequency: 50 Hz
RMS phase current: 120.089 A

D-axis inductance (Ld): 0.0757647 mH
Q-axis inductance (Lq): 0.0772449 mH
Phase self inductance (Lself): 0.0699321 mH
Phase mutual inductance (Lmutual): -0.00657265 mH

Figure 5.1. MotorXP-PM main window for Magnetostatic FE Analysis with different results shown.

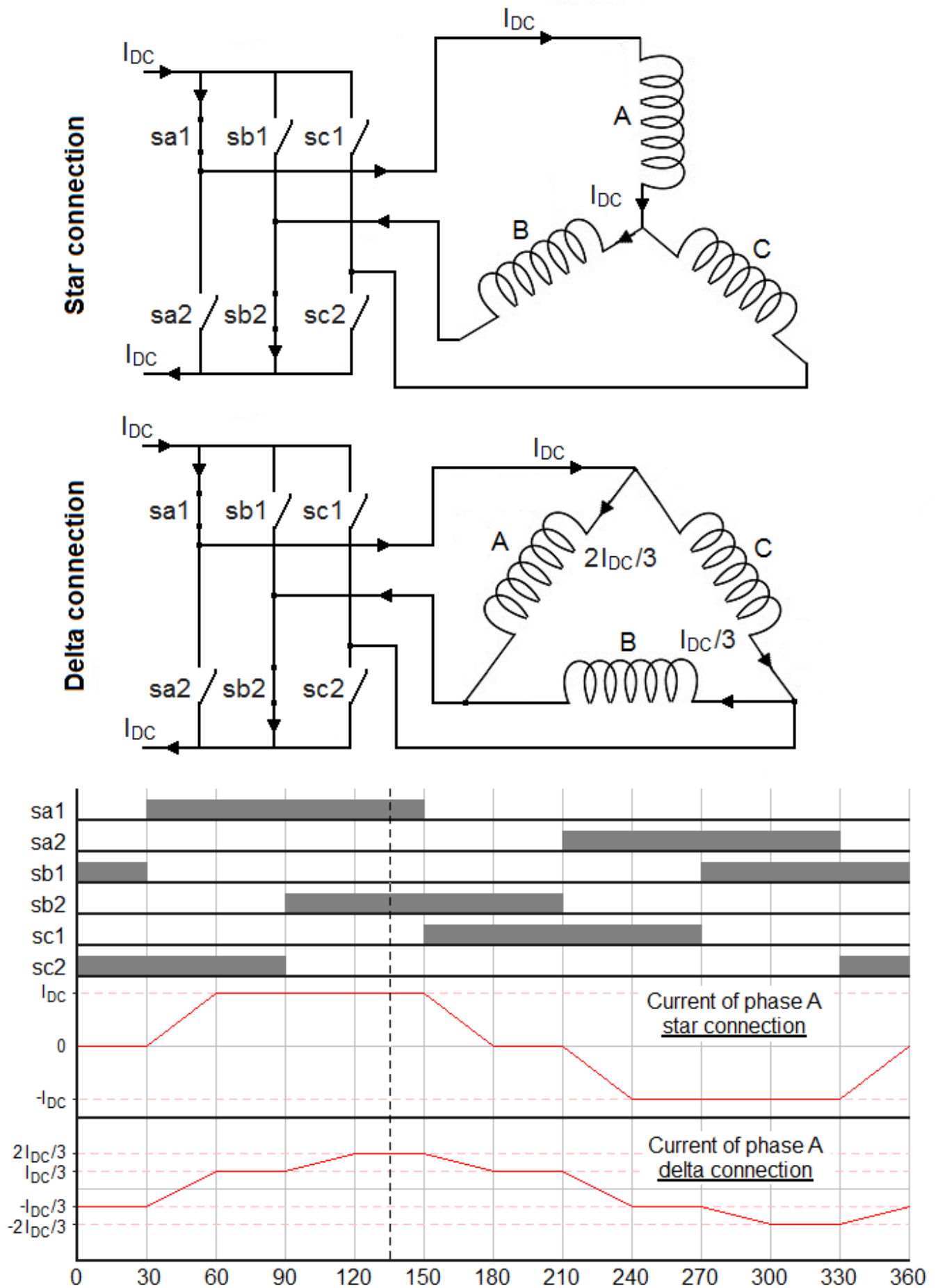


Figure 5.2. Example of trapezoidal current waveforms and distribution for star and delta connection.

5.2. Viewing Magnetostatic FE Analysis results.

The view of the **Plot Wizard** window when **Magnetostatic FE Analysis** is chosen is shown in Figure 5.3.

Several plot types are available for **Magnetostatic FE Analysis**:

- time-series data plot and frequency spectrum of the time-series data;
- air gap distribution plot and frequency spectrum of the air gap distribution;
- cross-section distribution plot;
- animation.

Plot Wizard also allows saving waveforms to a CSV-file (text file with comma-separated values) or to a Microsoft® Excel® spreadsheet file (Figure 5.3).

5.2.1. Time plots.

Time-series data plots are available from the **Time plot** panel of the **Plot Wizard** window. It is organized as a standard MATLAB plotting construction consisting of a set of `subplot` and `plot` functions. **Subplot** checkboxes allow you to control the number of axes or rectangular panes displayed within a current figure window. The corresponding axes are activated or deactivated by a mouse click within a cell of the **Subplot** column. When the subplot is activated, the **change** button allows you to choose quantities to be plotted into the selected axes. Quantities can be chosen from the dialog shown in Figure 5.4. Use Ctrl key to select several quantities. The list of available quantities is given below:

- Stator phase current (phase A, B, C);
- Phase current (d-axis, q-axis);
- Phase voltage (phase A, B, C);
- Phase voltage (d-axis, q-axis);
- Effective advance angle;
- Back-EMF (phase A, B, C);
- Back-EMF (d-axis, q-axis);
- Input electrical power;
- Mechanical power on the rotor shaft;
- Stator winding losses;
- Eddy current magnet losses;
- Electromagnetic torque by Maxwell stress tensor;
- Electromagnetic torque by virtual work method;
- Electromagnetic torque by flux linkage and current;
- Magnet torque (by Maxwell stress tensor);
- Reluctance torque (by Maxwell stress tensor);

- Cogging torque (by Maxwell stress tensor);
- Flux linkage (phase A, B, C);
- Flux linkage, (d-axis, q-axis);

Plot Wizard: Magnetostatic Analysis

Time plot

Subplot	Plot function or Matlab expression	Plot type	X-axis limits	Y-axis limits
<input checked="" type="checkbox"/> 1	plot(time, Ia, time, BackEMFa); legend('Ia, A', 'BackEMFa, V');	change waveform		
<input checked="" type="checkbox"/> 2	plot(time, Torque_maxwell); legend('Torque maxwell, N*m');	change waveform		
<input checked="" type="checkbox"/> 3	plot(time, Torque_maxwell); legend('Torque maxwell, N*m');	change spectrum		0 5
<input checked="" type="checkbox"/> 4	plot(time, Fluxlinkage_a, time, Fluxlinkage_b, time, Fluxlinkage_c); legend('Fluxlinkage_a, B, C');	change waveform		
<input type="checkbox"/>		change waveform		

☒ Synchronize subplot time-axis limits

Plot

Air gap distribution plot

Subplot	Variables	Time (s)	Slice	Plot type	X-axis limits	Y-axis limits
<input checked="" type="checkbox"/> 1	flux	+ < 0 > >> 1	1	distribution		
<input checked="" type="checkbox"/> 2	flux	+ < 0 > >> 1	1	spectrum	50	
<input type="checkbox"/>		+ < 0 > >> 1	1	distribution		
<input type="checkbox"/>		+ < 0 > >> 1	1	distribution		
<input type="checkbox"/>		+ < 0 > >> 1	1	distribution		

☒ Show graph legend

Plot

Cross-section distribution plot

Figure	Plotted quantity	Time (s)	Slice	Options	X-axis limits	Y-axis limits	Z-axis limits
<input checked="" type="checkbox"/> 1	Magnetic flux density, [T]	< 0 > >> 1	1	flux lines			
<input checked="" type="checkbox"/> 2	Relative permeability	< 0 > >> 1	1	flux lines			
<input checked="" type="checkbox"/> 3	Stator current density, [A/m^2]	< 0 > >> 1	1	flux lines			
<input type="checkbox"/>	None	< 0 > >> 1	1	none			
<input type="checkbox"/>	None	< 0 > >> 1	1	none			

Number of flux line levels: 20

☒ Full cross-section view

Plot

Animated plot

☒ Animate air gap distribution subplots

Skip: 0 files

Animation start time: 0 s

☒ Animate cross-section distribution figures

Frame display time: 0 ms

Animation stop time: 0.0198888 s

☒ Position figures

Pause Start animation

Data source folder: SimFiles\example_innerrorator_MSdata

Plot

Plot and save waveforms to a CSV-file

Plot and save waveforms to Excel

Figure 5.3. Magnetostatic FE Analysis Plot Wizard window and export data to file options.

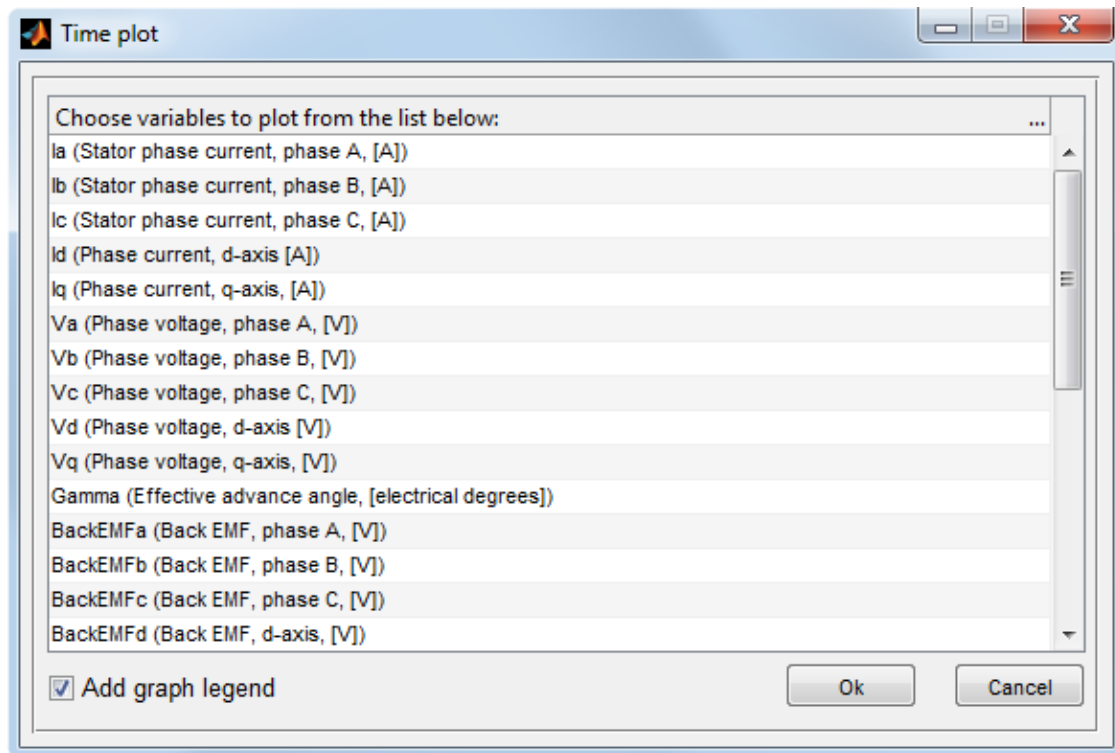


Figure 5.4. Choosing quantities to be plotted.

By clicking the **OK** button of the dialog (Figure 5.4), the MATLAB-expression is constructed to plot the selected quantities appearing in the corresponding line as shown in Figure 5.5 for plotting phase A stator current and back-EMF waveforms (first line), torque waveform (second line) and torque spectrum (third line). The fourth line is not active and, therefore, will not be plotted.

When the **Plot** button is clicked, the MATLAB figure window with plots of the selected quantities will appear (see Figure 5.6).

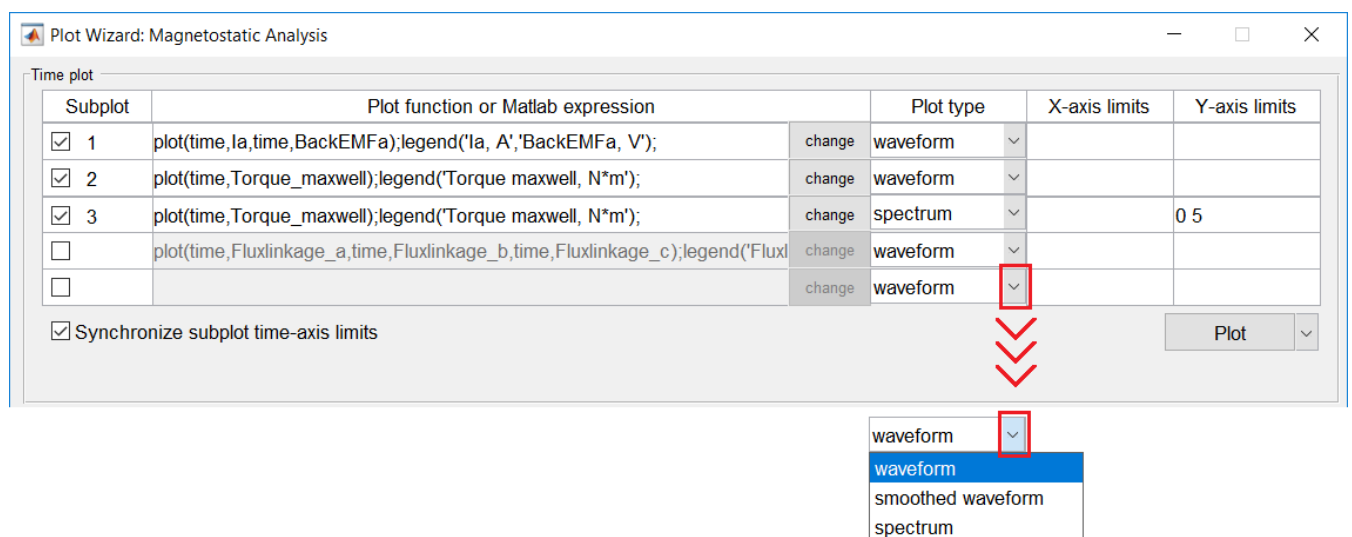


Figure 5.5. Example of using **Plot Wizard** for creating time plots.

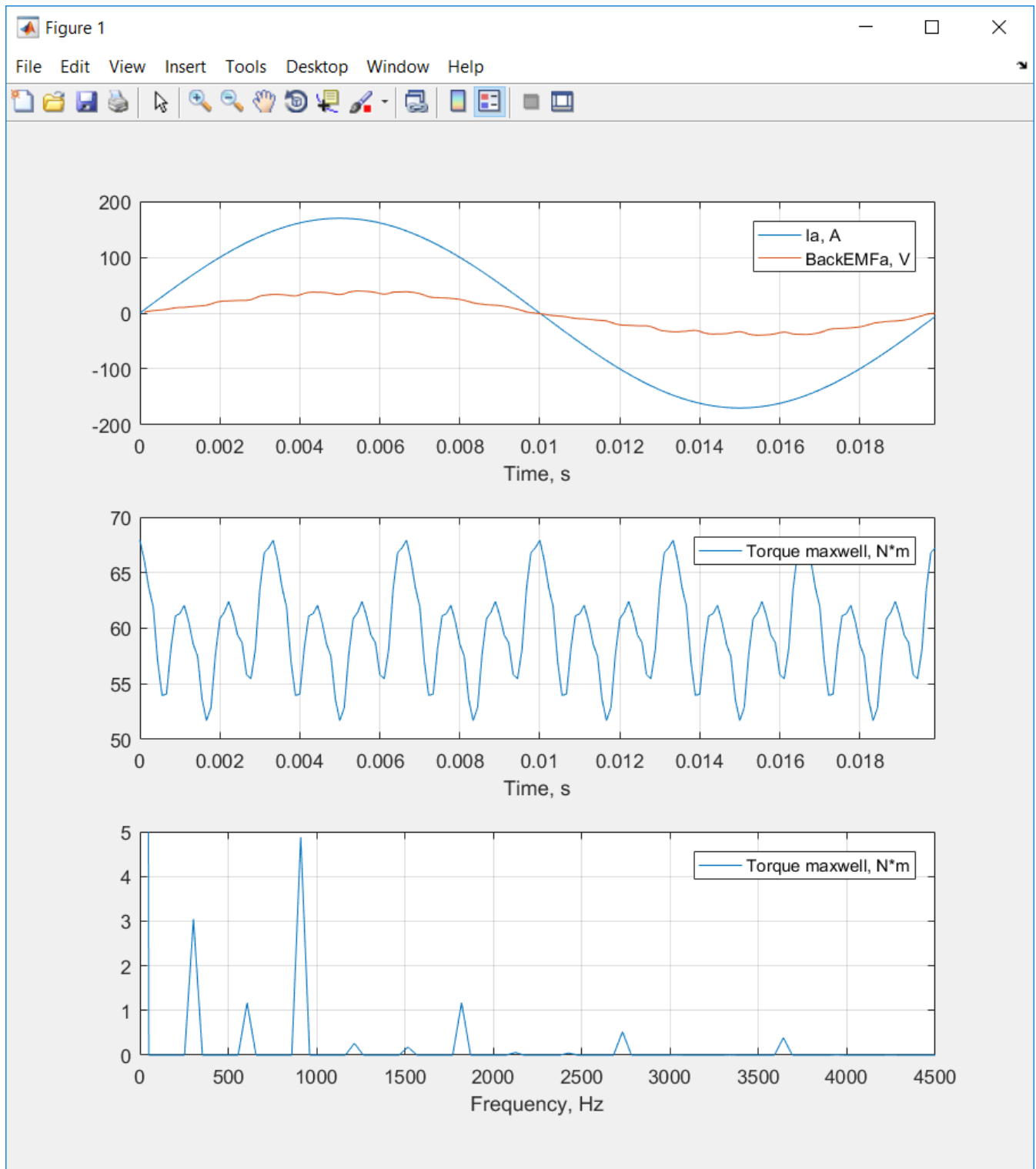


Figure 5.6. MATLAB figure window with plots of the selected quantities.

As it is seen, the plotting expression consists of the `plot` function with selected variables used as input arguments. If the **Add graph legend** checkbox of the dialog is checked, the `legend` function is added to the plotting expression so the graph legend of the corresponding axes will be shown. All plotting expressions are editable, so you can use any MATLAB plotting options and functions to change the way the plots appear on the screen. You can also change the `time` variable to any other variable you would

like to use as an input argument of the `plot` function. There is also a list of additional variables which can be used within a plotting expression (see section 9.3).

There are several plotting options available from the **Plot type** pop-up menu: *waveform*, *smoothed waveform* and *spectrum* (see Figure 5.5). Frequency spectrum is calculated over one electrical period.

X-axis limits and **Y-axis limits** fields of the **Time plot** panel allow you to set the x-axis and y-axis limits, respectively, to the specified values. Two limit values within a cell are separated by a space, comma ‘,’ or semicolon ‘;’. If the cell is empty, the limits of the corresponding axis will be chosen automatically. If the **Synchronize subplot time-axis limits** checkbox is checked, all subplots of the figure will have identical limits along the time-axis when you zoom or pan one of the subplots of the figure.

5.2.2. Air gap distribution plots.

Air gap distribution plots allow you to view the distribution of the particular quantity over the machine’s air gap as well as its frequency spectrum. It is available from the **Air gap distribution plot** panel. **Subplot** checkboxes allow you to control the number of axes or rectangular panes displayed within a current figure window. The corresponding axes are activated or deactivated by a mouse click within a cell of the **Subplot** column. When the subplot is activated, the “+” button allows you to choose quantities to be plotted from the dialog shown in Figure 5.7. Use Ctrl key to select several quantities.

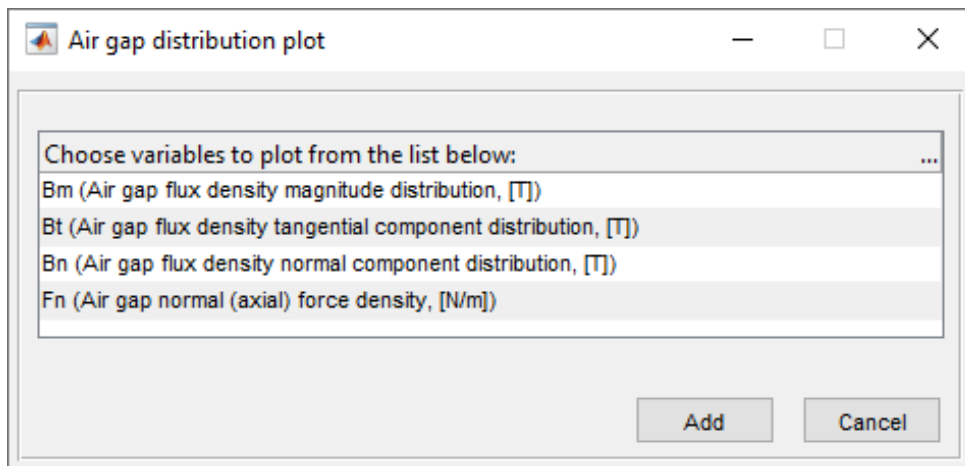


Figure 5.7. Choosing quantities to be plotted.

The following quantities are listed in the dialog of Figure 5.7:

Quantity	Comments
Bm (Air gap flux density magnitude distribution, [T])	Defined as $B_m = \sqrt{B_t^2 + B_n^2}$ where B_n and B_t – normal and tangential components of the magnetic flux density in the center of the air gap, as shown in Figure 2.4.
Bt (Air gap flux density tangential component distribution, [T])	B_n and B_t – normal and tangential components of the magnetic flux density in the center of the air gap, as shown in Figure 2.4.
Bn (Air gap flux density normal component distribution, [T])	
Fn (Air gap normal (radial) force distribution, [N])	According to Maxwell stress tensor method the normal force in each point of the round path can be expressed as the following: $F_n = -\frac{B_n^2 - B_t^2}{2\mu_0} d \cdot l$, where l – lamination length in z direction, d – length of the path in the center of the air gap between two consecutive points, μ_0 – permeability of the free space, B_n and B_t – normal and tangential components of the magnetic flux density in the center of the air gap, as shown in Figure 2.4.

By clicking the **Add** button of the dialog (Figure 5.7), the selected quantities are added to the corresponding line separated by commas as shown in Figure 5.8. Each line of the **Variables** column is editable, so you can use MATLAB arithmetic operations to obtain desired plots. For example, the second and third lines in Figure 5.8 produce plots of the tangential air gap force density distribution and the radial air gap force density distribution, respectively, where μ_0 – variable corresponding to the permeability of free space. According to Exp. 2.6 the tangential air gap force produces the electromagnetic torque so the plot of the electromagnetic torque distribution over an air gap can be obtained.

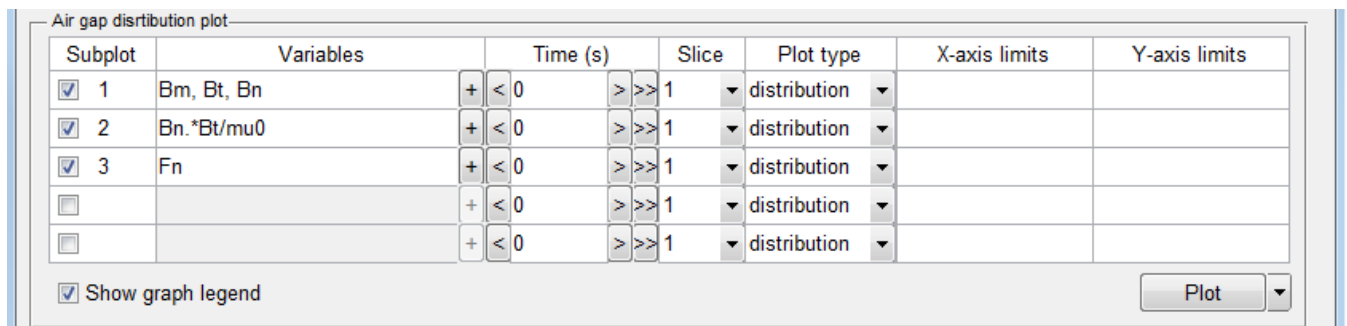


Figure 5.8. Example of using **Plot Wizard** for creating air gap plots.

Time fields of the **Air gap distribution plot** panel allow you to specify the time point which the selected quantities are plotted for. By default, zero time point is set up. Plotting for the time points other than zero is only possible, if the file containing data for the desired time point was previously saved. These data-files are being saved when **Save each field solution** is checked in the MotorXP-PM main window. So, if you are interested in viewing the air gap distribution plots for different time points make sure that the corresponding data-files are being saved. The folder used as a source of data-files is specified in the **Data source folder** field located at the bottom of the **Plot Wizard** window (Figure 5.3). You can change it by clicking the button to the right, if needed.

X-axis limits and **Y-axis limits** fields allow you to set the x-axis and y-axis limits, respectively, to the specified values in the same way as for the **Time plot** panel. **Slice** fields allow you to choose the machine's cross-section which the selected quantities are plotted for, if several slices are used (see section 2.4 for more details on multi-slice simulations).

Besides the air gap distribution, it is also possible to calculate the air gap harmonic components of the selected quantities. To obtain the frequency spectrum, select the *spectrum* item in the corresponding **Plot type** pop-up menu. An example of plotting the air gap distribution of the normal flux density component and its spectrum is shown in Figure 5.9. Only the first 100 harmonics are shown, since the value specified in the corresponding **X-axis limits** field is 100 (if the minimum limit equals to 0, it can be omitted).

The **Show graph legend** field allows you to choose whether the graph legend will be displayed.

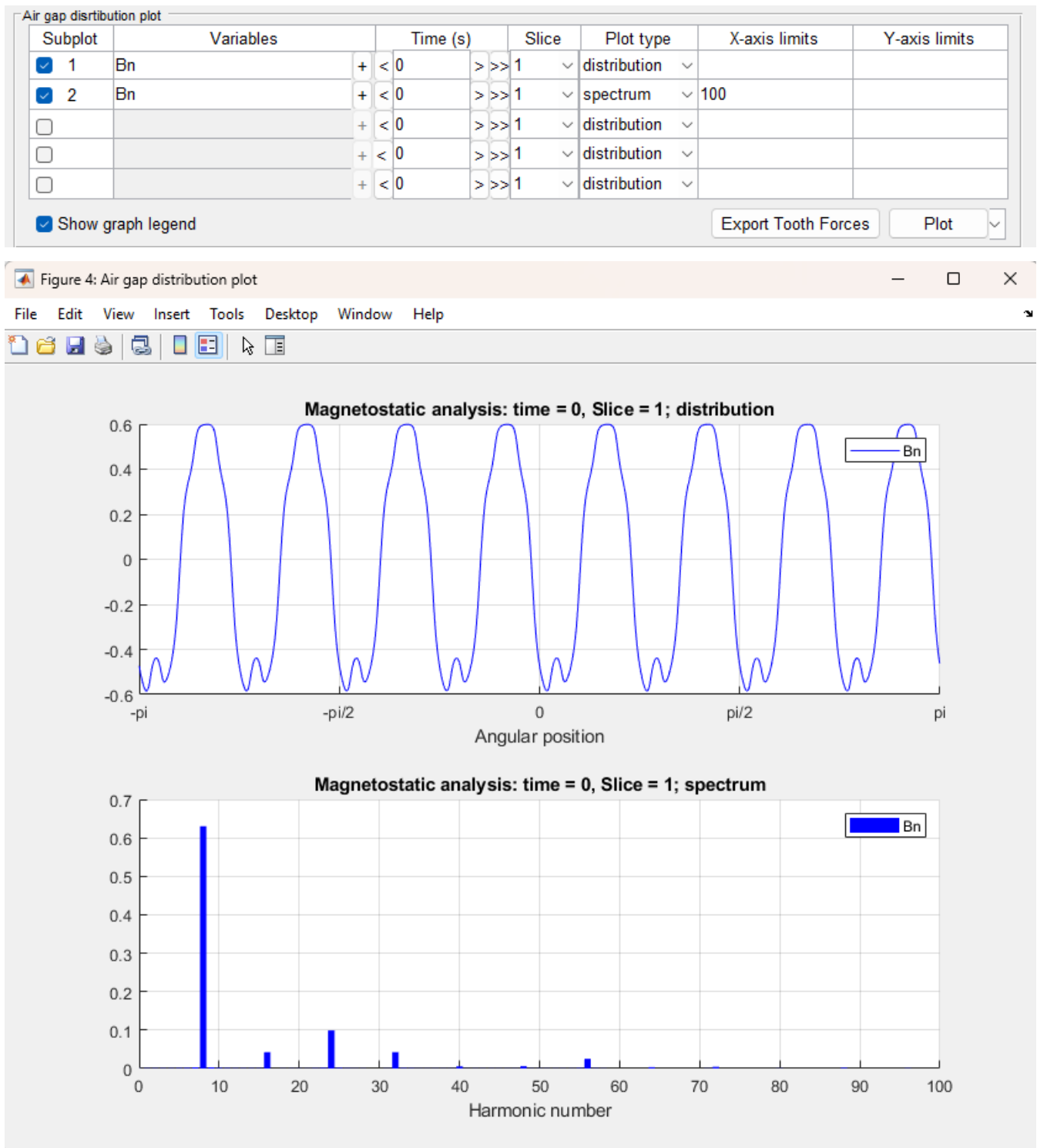


Figure 5.9. Plotting of the air gap distribution of the normal flux density component and its spectrum.

5.2.3. Cross-section distribution plots.

Cross-section distribution plots are available from the **Cross-section distribution plot** panel of the **Plot Wizard** window. As opposed to the two previous plot types, the cross-section distribution for each quantity is plotted in a separated window which contains only one axes. **Figure** checkboxes allow you to control the number of windows appearing when the **Plot** button is clicked. The corresponding figure is activated or deactivated by a mouse click within a cell of the **Figure** column. When the figure is activated,

the corresponding **Plotted quantity** pop-up menu allows you to choose the quantity to be plotted. If *None* is chosen, the machine's cross-section geometry will be plotted.

The following items are available from the **Plotted quantity** pop-up menu:

- Magnetic vector potential, [T*m];
- Magnetic flux density, [T];
- Magnetic field intensity, [A/m];
- Relative permeability;
- Stator current density, [A/m²];
- Squared flux density, [T];
- Magnetic field energy density, [J/m³];
- Stator loss density, [W/m³];
- Total loss density (stator+iron), [W/m³];
- Iron loss density, [W/m³];
- Demagnetizing field, [A/m];
- Demagnetizing field, [% of H_{cj}];
- Finite element mesh.

Time fields of the **Cross-section distribution plot** panel allow you to specify the time point which the selected quantity is plotted for. By default, zero time point is set up. Plotting for the time point other than zero is only possible, if the file containing data for the desired time point was previously saved. These data-files are being saved when **Save each field solution** is checked in the MotorXP-PM main window. So, if you are interested in viewing the cross-section distribution plots for different time points make sure that the corresponding data-files are being saved. The folder used as a source of data-files is specified in the **Data source folder** field located at the bottom of the **Plot Wizard** window (Figure 5.3). You can change it by clicking the button to the right, if needed.

Slice fields allow you to choose the machine's cross-section which the selected quantity is plotted for, if several slices are used (see section 2.4 for more details on multi-slice simulations).

The **Options** field allows you to show the magnetic flux lines (if *flux lines* is chosen) or magnetic flux arrows (if *flux arrows* is chosen) on the corresponding plot. Flux arrows are plotted such that the direction of the arrow indicates the direction of the flux, and the size of the arrow indicates the magnitude of the flux density. The **number of flux line levels** field allows you to alter the flux lines density. If periodic/antiperiodic boundary conditions are used, by default, only part of the cross-section (as it appears in **Mesh Editor**) will be plotted. Check the **Full cross-section view** checkbox if you want to plot the whole cross-section.

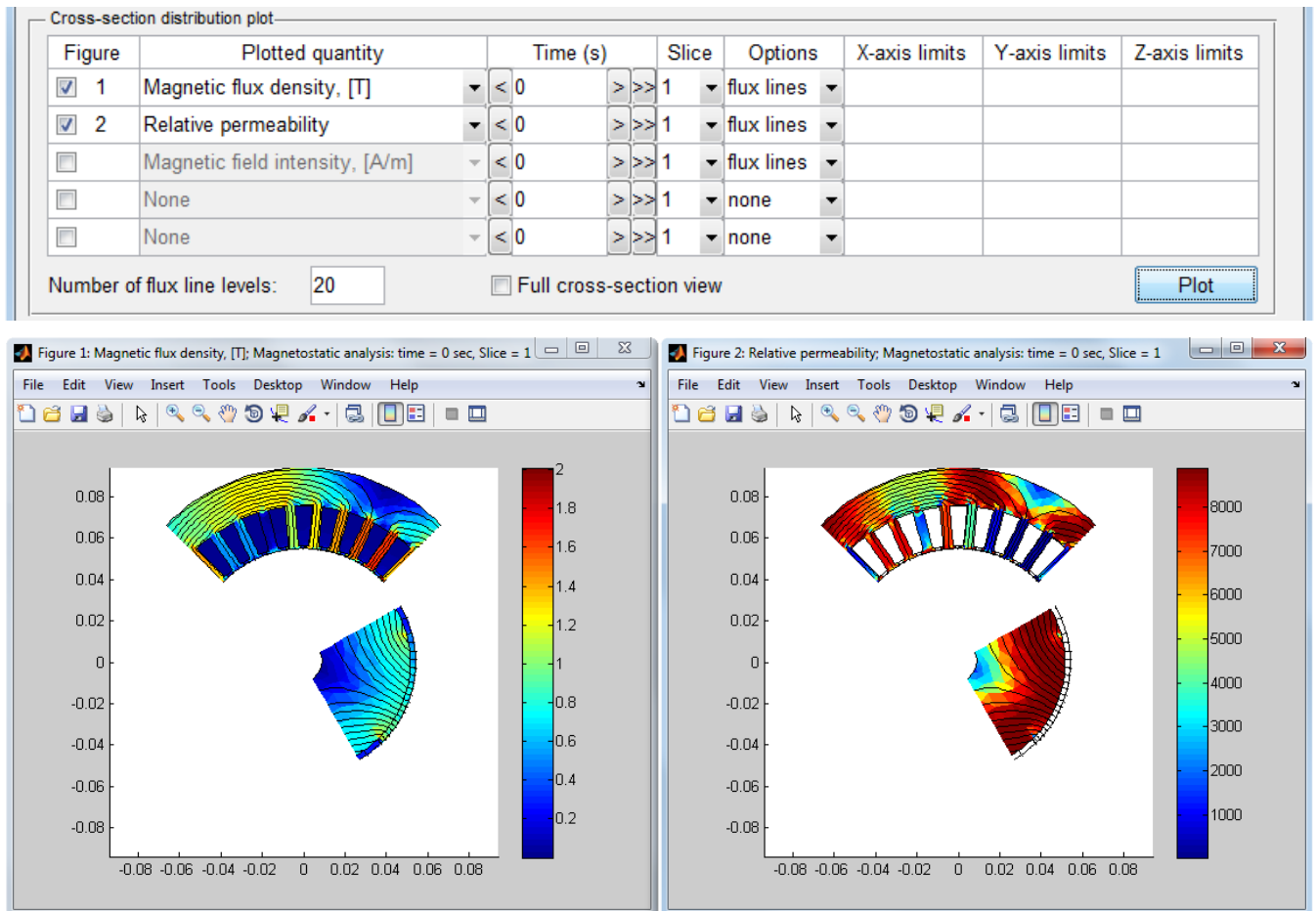


Figure 5.10. Example of using **Plot Wizard** for creating cross-section distribution plots.

X-axis limits and **Y-axis limits** fields allow you to set the x-axis and y-axis limits, respectively. If x-axis and y-axis limits are not specified, their values will be determined by the size of the machine's cross-section. **Z-axis limits** field sets the color scale limits to specified minimum and maximum values separated by a space, comma ',' or semicolon ';'. Values between z-axis limits are linearly mapped to the used color scale (colormap). Data values smaller or greater than specified z-axis limits are mapped to the minimum limit or to the maximum limit, respectively.

An example of plotting the cross-section distribution of the magnetic flux density and relative permeability is shown in Figure 5.10. When the **Plot** button is clicked, two windows will appear. As it is seen, the **flux lines** option is chosen for both figures, so the flux lines are additionally shown. Since the third figure is not active, the Magnetic field intensity is not plotted. Note that only part of the machine's cross-section is shown, since **Full cross-section view** is not checked.

5.2.4. Animation.

Animated plot panel is used to create animated sequences of the air gap distribution plots and/or the cross-section distribution plots (Figure 5.3). **Animate air gap distribution subplots** and **Animate cross-section distribution figures** checkboxes allow you to choose plot types to be animated. If **Animate air**

gap distribution subplots is checked, all selected subplots of the **Air gap distribution plot** panel will be involved in the animation. Similarly, if **Animate cross-section distribution figures** is checked, all selected figures of the **Cross-section distribution plot** panel will be involved in the animation. If **Position figures** control is checked, the size and location on the screen for all figure windows will be determined automatically so that the windows fit best the screen size.

Skip and **Frame display time** fields allow you to specify the way frames change each other while the animation is in progress. **Skip** field specifies the number of data-files or frames to be skipped. So, if **Skip** field is set to 0, data for each time step will be shown on the screen, if **Skip** field is set to 1, data for each second time step will be shown if **Skip** field is set to 2 – for each third time step and so on. The **Frame display time** field specifies the time each frame is displayed on the screen. Note that the least time the frame is displayed is limited by the animation function runtime. So, if the **Frame display time** field is 0, the actual time the frame is displayed on the screen will be the least possible in this case.

Animation start time and **Animation stop time** fields set up the time interval for the animation. The **data source folder** field specifies the folder with data-files to be animated. The same folder is used for animations and for air gap distribution and cross-section distribution plots. Using animation is only possible if the files containing data for the time interval to be animated were previously saved.

To start the animation, click the **Start animation** button. The current time point, animated quantity and other information is displayed at the top of each window involved in the animation. The animation continues until the time specified in the **Animation stop time** field is reached or until all windows involved in the animation are closed. You can also pause the animation using the **Pause** button. While the animation is in progress or paused, you can use all MATLAB figure tools, for example, changing the scale and position of the plots.

6. STEADY STATE D-Q ANALYSIS

As it was already pointed out, the D-Q analysis group consists of steady state and dynamic analysis types. **Steady State D-Q Analysis** allows to estimate steady-state machine parameters and characteristics such as torque – speed curve, torque – advance angle curve, etc. as well as to compute the efficiency map and other performance maps. Since the variation of D-Q model parameters with rotor positions is not taken into account and the back-EMF waveform is assumed to be sinusoidal the accuracy of **Steady State D-Q Analysis** can be lower compared with the finite element analysis.

6.1. Building a D-Q model.

The view of the main window when **Steady State D-Q Analysis** is chosen is shown in Figure 6.1. In order to use **Steady State D-Q Analysis** the D-Q model of the machine should be previously built, i.e. parameterized using FEA simulations. Use the **Build D-Q model** button to compute the D-Q model parameters of the machine. The short description of the d-q representation of a permanent magnet machine is presented in section 2.5. While the D-Q model is being built the model parameters L_d , L_q , L_{dq} , Ψ_{md} and Ψ_{mqd} are derived from the FEA solution for each pair of supply current and advance angle values to include saturation and cross-saturation effects into account. The supply current values are defined by the **Maximum RMS current** and **Current step** field values. The advance angle values are in the range between 0° and 360° electrical degrees with the step defined by the **Advance angle step** field value. **Number of rotor positions** field specifies the number of rotor positions for which the D-Q model parameters are averaged. Rotor positions are spaced at intervals corresponding to half of cogging torque period. At least 2 rotor positions are recommended to compensate variation of the D-Q model parameters with the rotor rotation due to cogging torque. Iron losses can be included into the D-Q model using the **Include iron losses** checkbox. If iron losses are included, the D-Q model parameterization takes significantly longer time. Once the D-Q model is built, different machine characteristics can be obtained.

6.2. Viewing Steady State D-Q Analysis results.

The view of the **Plot Wizard** window when **Steady State D-Q Analysis** is chosen is shown in Figure 6.2. Use the **Quantities to plot** button to see the list of all available parameters. The plotted quantities can be displayed as a function of *Speed*, *Supply current* or *Advance angle* depending on the chosen item in the **X-axis quantity** pop-up menu. Speed, current and advance angle values should be entered in the corresponding fields. If a list of values is used, the values can be separated by space, comma or semicolon, alternatively the statement ‘start:step:stop’ can be used to enter linearly spaced values between ‘start’ and ‘stop’.

There are three **Model types** to choose for **Steady State D-Q Analysis**: *D-Q with sinusoidal supply*, *D-Q with PWM supply* and *FEA based. D-Q with sinusoidal supply* assumes ideal sinusoidal current and voltage waveforms, *D-Q with PWM supply* uses the dynamic D-Q simulation (refer to chapter 7) and corresponding drive type (*Current hysteresis PWM* or *Space vector PWM* specified in the **Drive Settings**

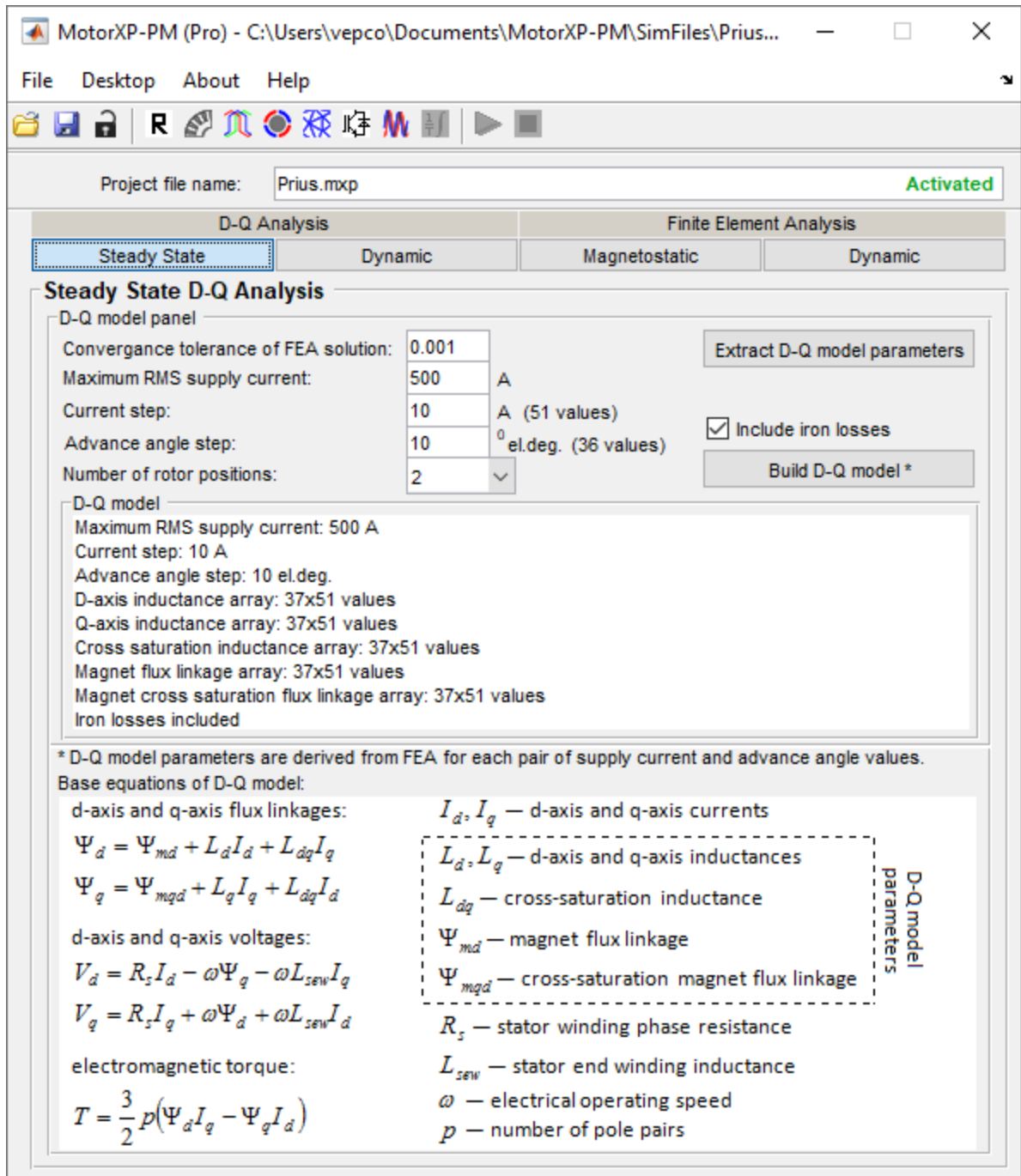


Figure 6.1. MotorXP-PM main window for Steady State D-Q Analysis.

window as described in section 4.5) to obtain the result. **FEA based** is the same as **D-Q with sinusoidal supply** but the result is determined directly from FEA solution (only for one rotor position) so the analysis can be done without the need to build the D-Q model.

Max. phase voltage selection defines either the ideal current source with no voltage limit is used (if **unlimited voltage** is chosen) or the limited supply voltage is used and how the **Max. RMS phase voltage** available from the inverter is determined. If **by Vdc and PWM method** is chosen, the **Max. RMS phase voltage** is determined based on the DC supply voltage V_{dc} and drive type

Plot Wizard: Steady State D-Q Analysis

D-Q analysis plot

Quantities to plot <<

X-axis quantity: Speed

Rotor speed value(s): 0:50:7000 RPM

RMS supply current value(s): 230 A

Advance angle value(s): 50 el.deg.

Max. phase voltage selection: by Vdc and PWM method

Max. RMS phase voltage: 204.124 ? V

Field-weakening control: max-torque-limited-current

Model setup

Model type: D-Q with sinusoidal supply

D-Q parameters interpolation: Nonlinear

☒ Multiple y-axes

Plot

Quantities to plot

☒ RMS phase current

☒ RMS phase voltage

☐ Average d-axis current

☐ Average q-axis current

☐ Average d-axis voltage

☐ Average q-axis voltage

☒ Advance angle

☒ Total torque

☐ Magnet torque

☐ Reluctance torque

☒ Input electrical power

☒ Output mechanical power

☐ Reactive power

☒ Efficiency

☒ Power factor

☐ Stator winding loss

☐ Iron loss

☐ RMS phase back-EMF

☐ D-axis inductance Ld

☐ Q-axis inductance Lq

☐ Mutual inductance Ldq

☐ D-axis magnet flux linkage

☐ Q-axis magnet flux linkage

☐ Lq/Ld ratio

Performance maps plot

Maps to plot >>

Control method: Max. efficiency

Advance angle values: 0:0.1:90 el.deg.

Maximum RMS phase voltage: 204.124 <-rated V

Maximum RMS phase current: 200 <-rated A

Maximum input power: <-rated W

Maximum speed: 7000 <-rated RPM

Mechanical loss coefficients: 2e-5 3.5e-3 0

Map resolution:

Torque step: 5 N*m

Speed step: 50 RPM

Model setup

Model type: D-Q with sinusoidal supply

D-Q parameters interpolation: Linearized

Plot Maps

Create Maps

Additional plotting options

Plot

Plot and save to a CSV-file

Plot and save to Excel

Plot and save to a MAT-file

Plot from a MAT-file

Figure 6.2. Steady State D-Q Analysis Plot Wizard window.

(**Current hysteresis PWM** or **Space vector PWM**) specified in the **Drive Setting** window as well as stator winding connection (star or delta). For the space vector PWM the maximum inverter phase peak voltage

is $\frac{V_{DC}}{\sqrt{3}}$. For the current hysteresis PWM in order to keep the voltage and current sinusoidal, the inverter

phase peak voltage should not exceed $\frac{V_{DC}}{2}$. Note that for delta-connected winding the phase voltage of the machine is the line-to-line voltage of the inverter.

If the motor operates with the limited voltage the field-weakening operation above the base speed can be analyzed. There are two field-weakening strategies available: **max-torque-limited-current** and **const-output-power**. If the field-weakening is applied the current and advance angle are adjusted to meet the corresponding requirements of the field-weakening control algorithm. If **const-advance-angle** is chosen, the current and advance angle are no longer controlled - current starts to naturally decrease above the base speed.

D-Q parameters interpolation specifies how D-Q model parameters are interpolated. **Linearized** interpolation is sufficient in most cases provided that the D-Q model has been built with enough number of supply current and advance angle pairs.

When **Model type** is set to **D-Q with PWM supply** the simulation speed/accuracy setup can be additionally adjusted by clicking the **Simulation speed/accuracy** button as shown in Figure 6.3.

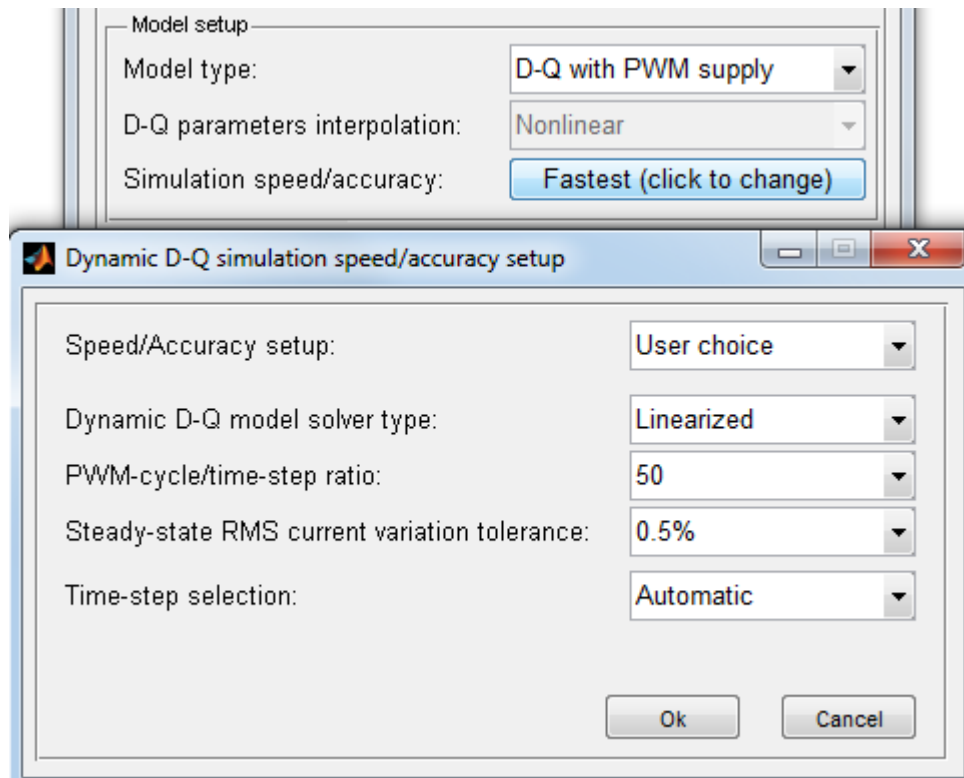


Figure 6.3. Dynamic D-Q simulation speed/accuracy setup.

You can choose one of the predefined setups (***Fastest***, ***Balanced*** or ***Accurate***) from the **Speed/Accuracy setup** pop-up menu or manually adjust the simulation speed/accuracy setup choosing the ***User choice*** item.

Dynamic D-Q model solver type set to ***Linearized*** is sufficient in most cases. Be careful when using ***Linear*** solver type since in this case the fixed D-Q model parameters are used – linear solver can only be used if the current waveform is very close to sinusoidal.

PWM-cycle/time-step ratio determines the choice of the time step when **Time-step selection** is set to ***Automatic***. Good practice is to manually specify the time-step since in some cases the optimal time-step cannot be determined by automatic selection.

Steady-state RMS current variation tolerance defines the acceptable variation of the RMS current from one period to the next to determine whether the steady-state is reached. The simulation stops when the steady state is reached i.e. when the variation of the RMS current from one period to the next does not exceed the specified variation tolerance.

Several examples of **Steady State D-Q Analysis** plots are shown in Figures 6.4-6.5. Figure 6.4 shows the plot corresponding to the **Steady State D-Q Analysis** setup shown in Figure 6.2 with ideal sinusoidal supply and “max-torque-limited-current” field-weakening control above the base speed. Figure 6.5 compares plots obtained with ideal sinusoidal supply with no voltage limit (***D-Q with sinusoidal supply***) and with space vector PWM inverter supply (***D-Q with PWM supply***) for 1500RMP rotor speed. Supply current is varied in the range between 0 and 300A in both cases. As shown for the ideal sinusoidal supply (top) the phase current changes from 0 to 173A (delta-connected winding), the phase voltage is freely increasing, and the advance angle is strictly 0 electrical degrees. For the PWM supply (bottom) after some point the inverter is no longer able to maintain the desired current since the phase voltage reaches its limit. Note that the effective advance angle is shown for PWM supply.

There are additional plotting options as shown in Figure 6.2. These allow saving the plotted quantities to a CSV-file (text file with comma-separated values), to a Microsoft® Excel® spreadsheet file or to a MAT-file or plot the selected quantities from the MAT-file previously saved.

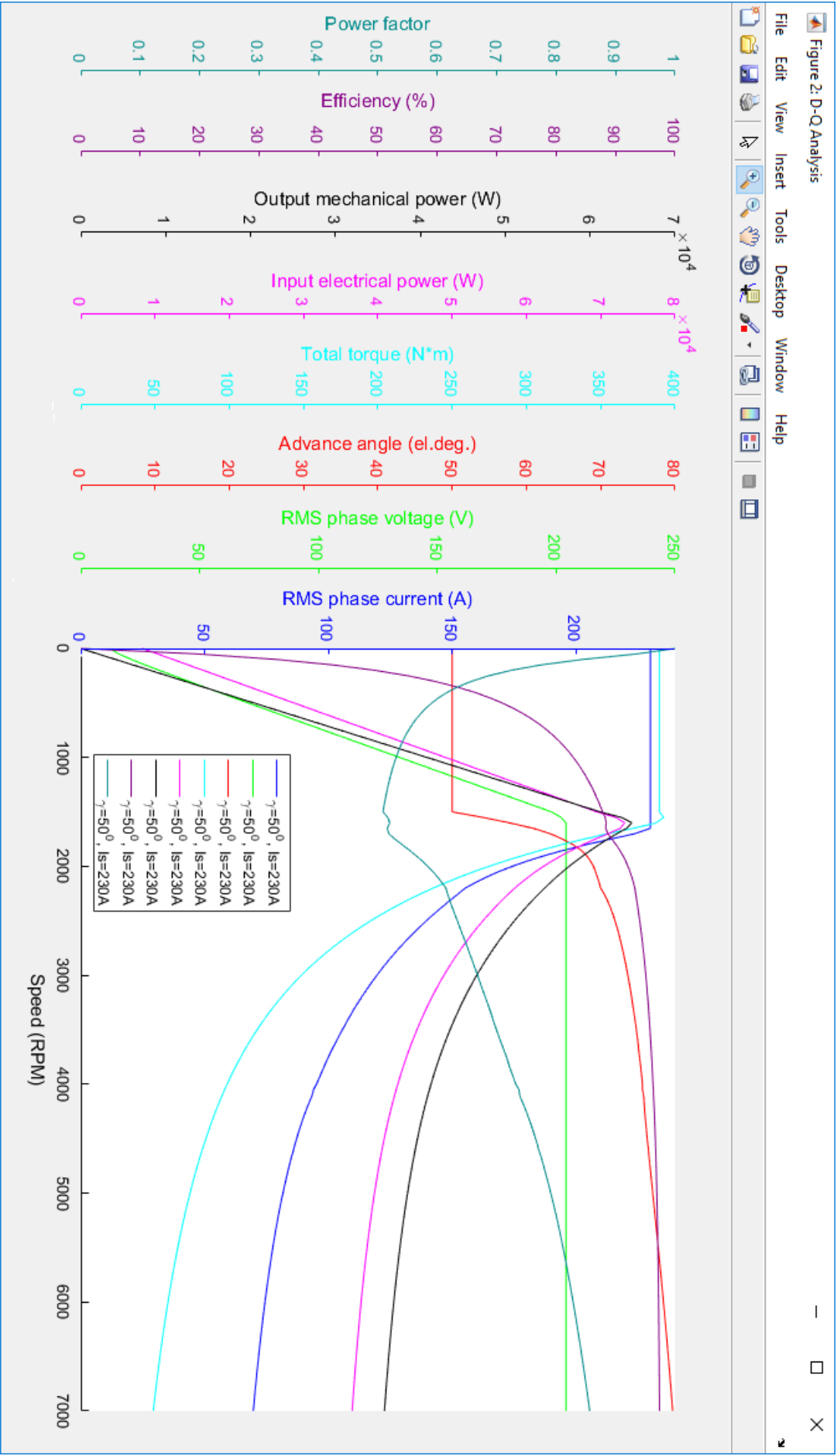


Figure 6.4. D-Q Analysis plot with ideal sinusoidal supply and *max-torque-limited-current* field-weakening control above the base speed (*P_{rius.max}*).

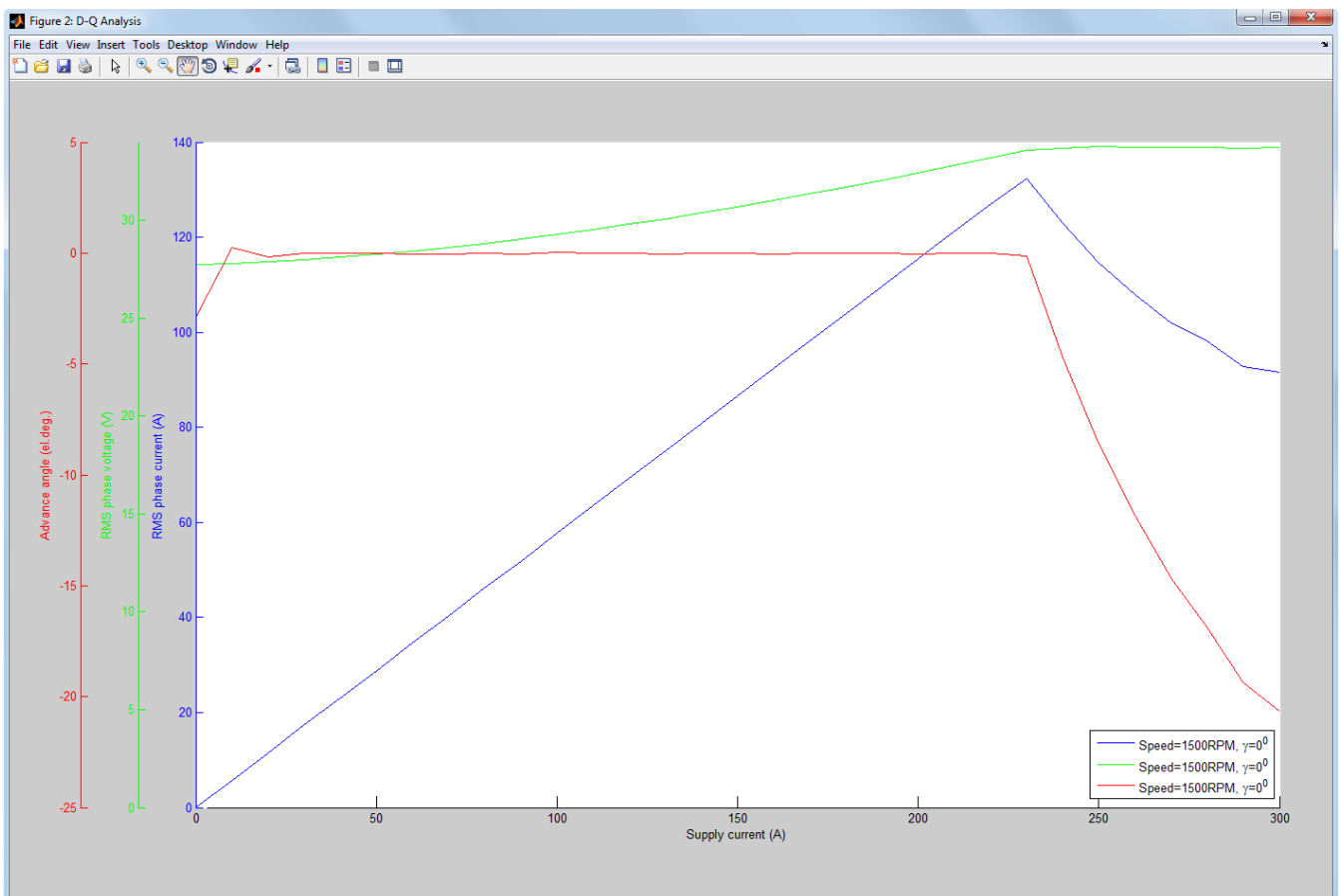
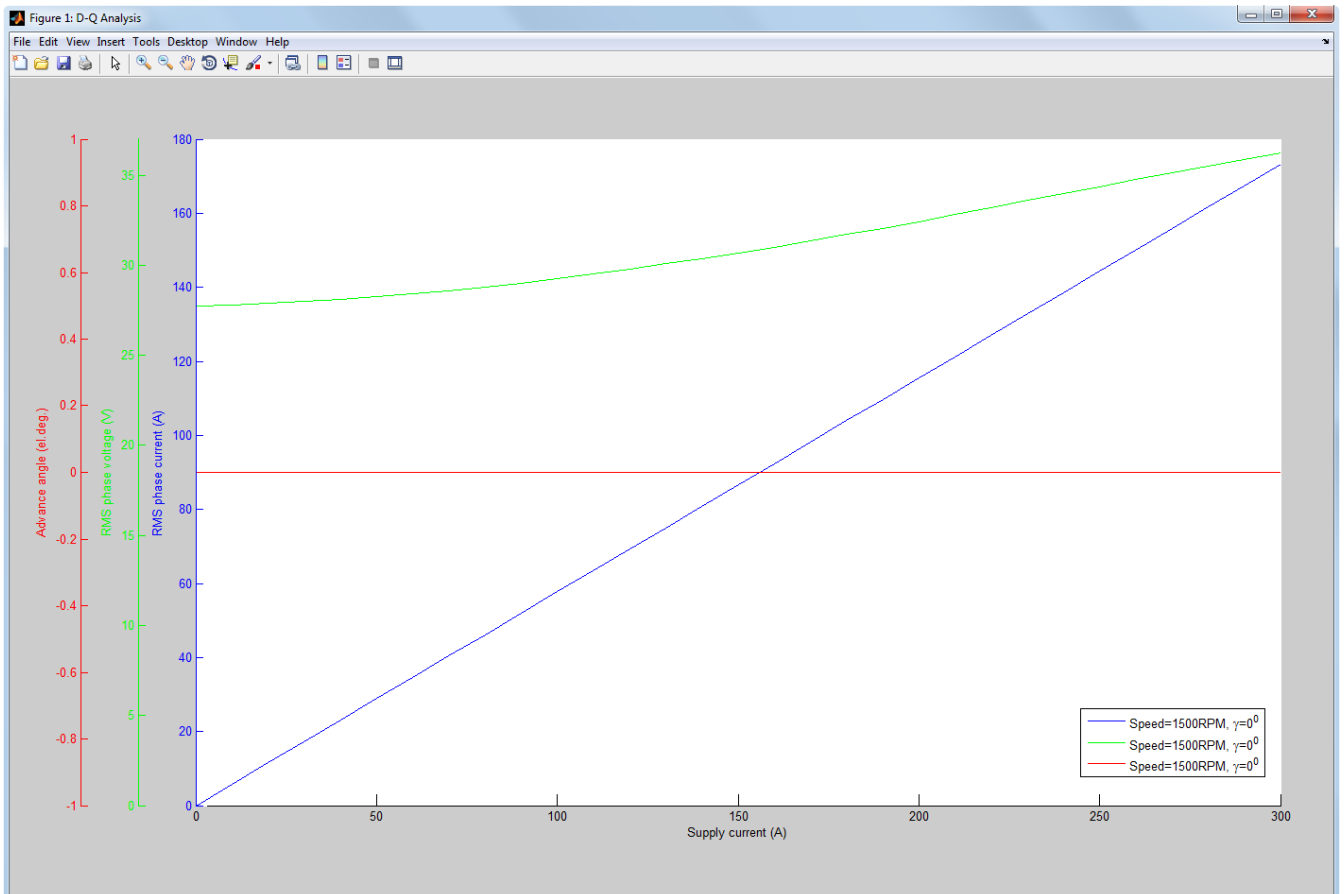


Figure 6.5. Comparison between *D-Q with sinusoidal supply* and *D-Q with PWM supply* model types (*example_innerrotor.mxp*).

6.3. Efficiency map and other performance maps.

Another option of **Steady State D-Q Analysis** is the generation of the efficiency map and other performance maps such as current map, voltage map, power map, loss map and etc. Use the **Maps to plot** button to see the full list of available performance maps as shown in Figure 6.6. There are two control strategies available to generate the performance maps of the machine specified by the **Control method** pop-up menu: *Max. efficiency* and *MTPA/MTPV*. If *Max. efficiency* is chosen, the calculation of the performance maps is based on the machine operation with maximum possible efficiency, i.e. the iterative procedure is applied to search for the RMS current and advance angle values which provide the maximum efficiency for the specific values of speed and load torque taking into account the current and voltage limits. If *MTPA/MTPV* is chosen, the calculation of the performance maps is based on the maximum torque per ampere (MTPA) and maximum torque per volt (MTPV) operation with the current and voltage limits taken into account. The example of the efficiency maps for motor and inverter and some other performance maps are shown in Figure 6.7. Note that the part of the efficiency map below 80% is not shown as the **Hide efficiency less than** field is set to 80% in the **Plot Wizard** window.

Advance angle values field specifies the range of advance angles the performance maps are computed for. Use the statement ‘start:step:stop’ to enter linearly spaced values between ‘start’ and ‘stop’, where ‘step’ determines the accuracy of the calculated performance maps in terms of the advance angle tolerance. Advance angle values in the range between 0° and 90° electrical degrees correspond to the motor operation mode, the range of advance angle values between 90° and 180° degrees correspond to the generator mode. **Maximum RMS phase voltage** and **Maximum RMS phase current** define the supply voltage and current limits applied while the performance maps are computed. **Maximum input power** does influence the calculation of performance maps and only affects the way how the maps are displayed: the part of the map beyond the input power limit specified by the **Maximum input power** field will not be shown. If the **Maximum input power** field is left empty when the input power is not limited (limited only by maximum voltage and current) no part of the map will be hidden.

Button ‘<-rated’ to the right of some fields allows you to set up the corresponding parameter to its rated value specified in the **Rated Data** window.

Use the **Mechanical loss coefficients** field to apply the mechanical losses while the efficiency map of the machine is plotted. Mechanical loss coefficients do not influence the calculation of performance maps and they are applied after the performance maps are calculated. Mechanical loss coefficients are entered as a vector whose elements are the coefficients in descending powers of the mechanical loss equation. For the mechanical loss coefficients [2e-5 3.5e-3 0] shown in Figure 6.6 the mechanical loss equation is as follows:

$$W_{mech.loss} = 0.00002(rpm)^2 + 0.0035(rpm) + 0,$$

where *rpm* – rotor speed in RPM.

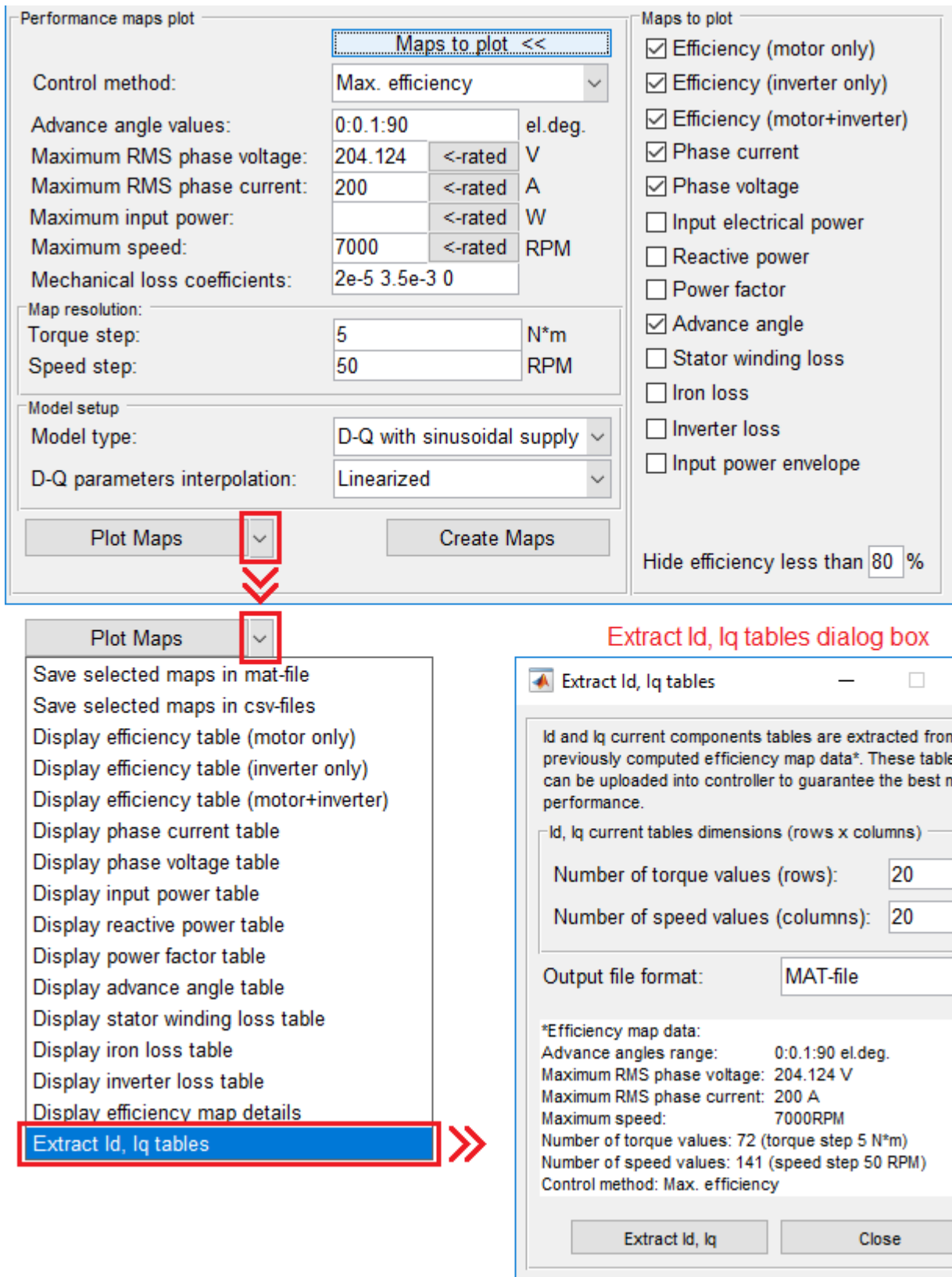


Figure 6.6. Performance maps plot panel of Plot Wizard with the Extract Id, Iq tables dialog box.

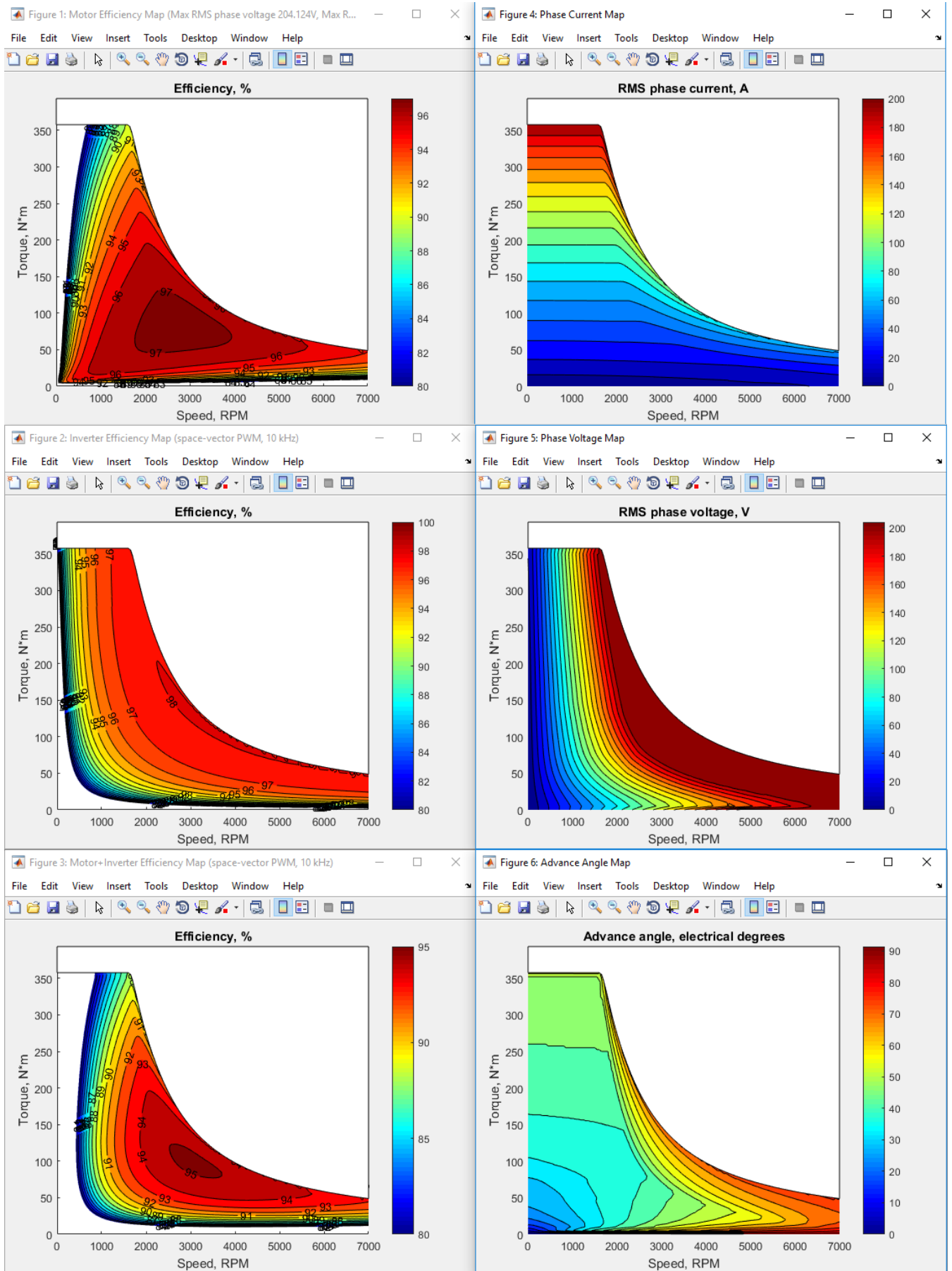


Figure 6.7. Some performance maps generated by Steady State D-Q Analysis Plot Wizard.

The number of points (speed-torque pairs) at which the performance maps are calculated is defined by the **Map resolution** panel.

D-Q parameters interpolation specifies how D-Q model parameters are interpolated. *Linearized* interpolation is sufficient in most cases provided that the D-Q model has been built with enough number of supply current and advance angle pairs. Performance maps are calculated assuming ideal sinusoidal current and voltage waveforms.

To generate the performance maps, click the **Create Maps** button. All the performance maps will be calculated no matter which ones are selected in the Plot Wizard window; however, only selected maps will be displayed. Once the performance maps are computed for the first time, they will be saved within the project mxp-file. Every time you click the **Create Maps** button once again you will be asked whether you want to replace previously computed maps with new ones. Use the **Plot Maps** button to plot the performance map previously saved. There are also some additional options available to the right of the **Plot Maps** button as shown in Figure 6.6, such as displaying the performance map as a table choosing the corresponding *Display ... table* item or saving selected maps in MAT-file or CSV-files.

6.4. Extracting Id and Iq current components tables from the efficiency map data.

Calculation of the performance maps corresponds to determining the optimum motor operating conditions with maximum motor efficiency or MTPA/MTPV operation depending on the selected **Control method** pop-up menu item. This is valuable information to design the control system. Id and Iq current components can be extracted from the efficiency map data and then uploaded into controller. Figure 6.6 shows the **Extract Id, Iq tables** dialog box. The output data produced by the **Extract Id, Iq tables** dialog box are 2-D lookup table representations of the Id and Iq current components with rows corresponding to the torque values and columns corresponding to the speed values. Number of torque and speed value can be different from those used for the calculation of the performance maps (defined by the **Torque step** and **Speed step** fields of the **Map resolution** panel). The output tables can be saved as a MAT-file (all tables in one file) or CSV-files (each table in separate CSV-file) depending on the selected **Output file format** pop-up menu item.

6.5. Extracting D-Q model parameters.

Parameters of the D-Q model such as the d-axis and q-axis inductances L_d and L_q , magnet flux linkage Ψ_{md} , cross-saturation inductance L_{dq} and cross saturation magnet flux linkage Ψ_{mqd} , can be extracted and saved in a separate file (files) to use the generated D-Q model in another application. The **Extract D-Q model parameters** window shown in Figure 6.8 is available by clicking the **Extract D-Q model parameters** button of the **Steady State D-Q Analysis** main window (Figure 6.1). The output data are available in two formats: as a *function of phase RMS current and advance angle* (see Figure 6.8(a)) and as a *function of D-axis and Q-axis currents Id and Iq* (see Figure 6.8(b)), depending on the selected **Output data format** pop-up menu item.

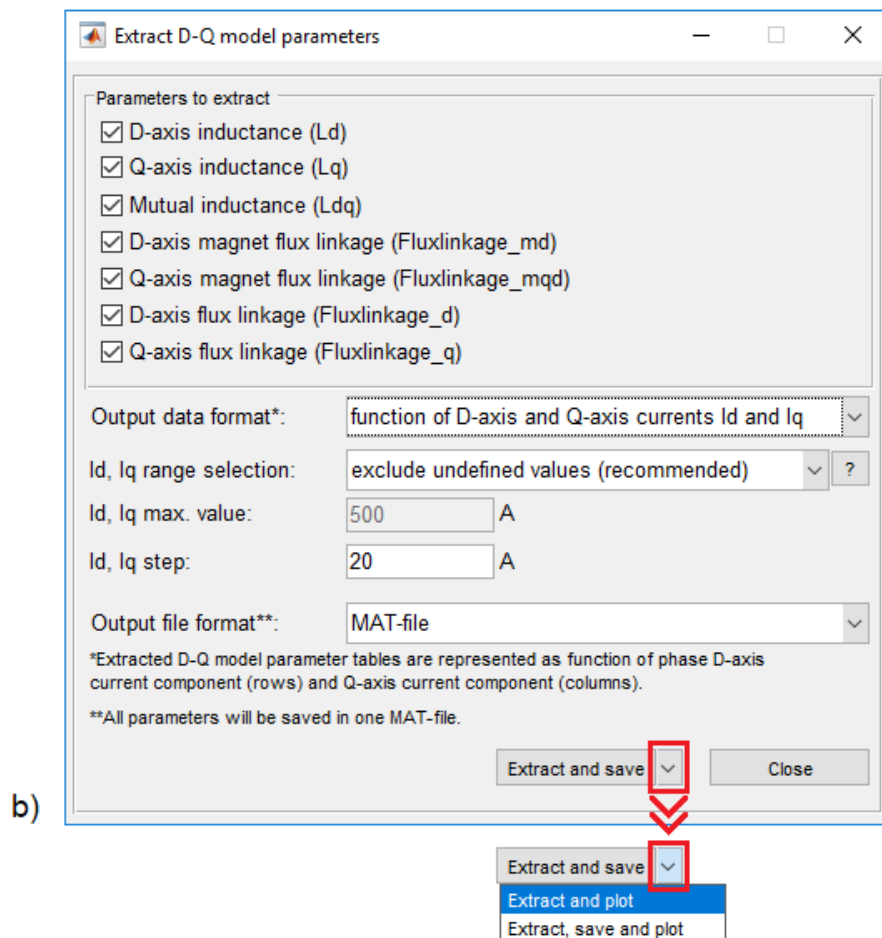
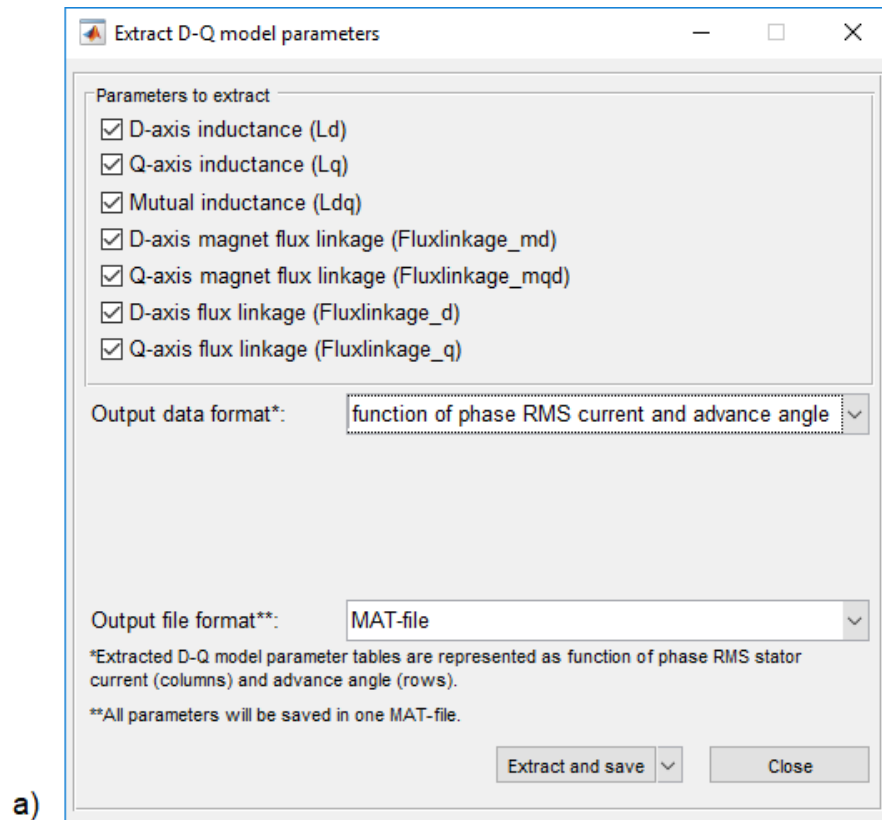


Figure 6.8. **Extract D-Q model parameters** window for different output data formats: (a) *function of phase RMS current and advance angle*, (b) *function of D-axis and Q-axis currents I_d and I_q* .

When the *function of D-axis and Q-axis currents I_d and I_q* output data format is used the **Id, Iq range selection** pop-up menu determines the maximum value of I_d and I_q (I_d and I_q has the same maximum value which is displayed in the **Id, Iq max. value** field). Figure 6.9 explains the I_d , I_q range selection principle. Since initially the D-Q model parameters are determined as a function of phase RMS current and advance angle, when the *maximum possible range* item is selected from the **Id, Iq range selection** pop-up menu, there will be some I_d and I_q pairs which the D-Q model parameters are not defined for (“undefined values” in Figure 6.9). Undefined values are replaced with “NaN” (not a number) in the output tables. I_m in Figure 6.9 corresponds to the maximum phase current amplitude for which the D-Q model parameters are determined ($I_m/\sqrt{2}$ is the RMS phase current corresponding to the **Maximum RMS supply current** value of the **Steady State D-Q Analysis** main window (Figure 6.1)). To exclude undefined values of the D-Q model parameters from the output tables the *exclude undefined values* item should be selected from the **Id, Iq range selection** pop-up menu. **Id, Iq step** determines the size of the output tables (number of rows is equal to the number of columns). The output tables can be saved as a MAT-file (all tables in one file) or CSV-files (each table in separate CSV-file) depending on the selected **Output file format** pop-up menu item.

Additional options to the right of the **Extract and save** button (see Figure 6.8) allow also to plot the selected parameters versus I_d and I_q or versus RMS phase current and advance angle depending on the selected **Output data format** pop-up menu item.

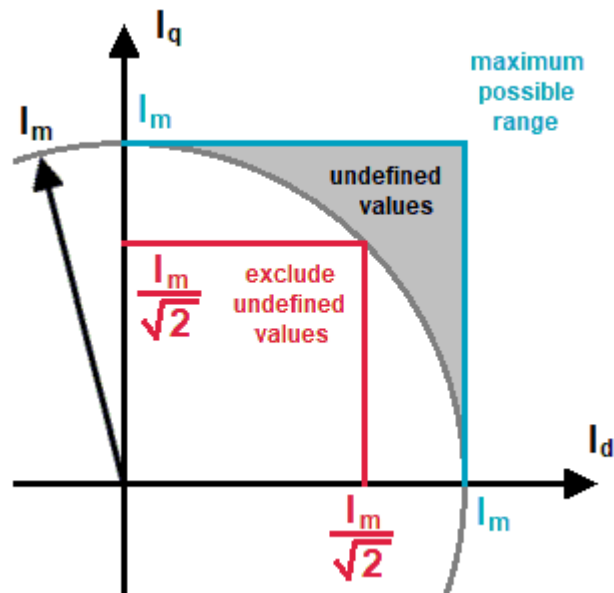




Figure 6.9. Explanation on the I_d and I_q current components range selection.

7. DYNAMIC D-Q ANALYSIS

Dynamic D-Q Analysis is based on Exp. 2.11 and allows the user to simulate the dynamic behavior of the machine taking into account non-sinusoidal voltage and current waveforms imposed by inverter switching. To avoid misunderstanding of the results you should be aware of the following limitations of **Dynamic D-Q Analysis**:

- The back-EMF waveform is assumed to be sinusoidal. To examine the Back-EMF waveform, use **Magnetostatic FE Analysis** or **Dynamic FE Analysis** instead.
- Variation of the D-Q model parameters with rotor position is not taken into account and the torque is defined by Exp. 2.10, therefore the torque ripple is not properly determined. To determine the torque ripple, use **Magnetostatic FE Analysis** or **Dynamic FE Analysis** instead.

7.1. Running Dynamic D-Q Analysis.

The view of the main window when **Dynamic D-Q Analysis** is chosen is shown in Figure 7.1. To use **Dynamic D-Q Analysis** the D-Q model of the machine should be built as described in the previous chapter. Use toolbar buttons  and  to start and stop the simulation. If the simulation is stopped all the data of the running simulation will be lost.

Solver type specifies how D-Q model parameters are interpolated. *Linearized* solver type is sufficient in most cases provided that the D-Q model has been built with enough number of supply current and advance angle pairs. Be careful when using *Linear* solver since in this case fixed D-Q model parameters are used – linear solver can only be used if the current waveform is very close to sinusoidal, i.e. the switching ripple in the current is considerably small comparing with the current amplitude.

Be aware that **Time step** should be small enough comparing with the PWM sampling period to get accurate results. Control the *Discretization error* in the **Time Average Quantities** window shown in Figure 7.3 to adjust the time step. It is recommended that the discretization error does not exceed 1%.

There are two ways to stop the dynamic D-Q simulation in MotorXP-PM: by specifying the **Simulation stop time value** or by reaching the steady state, i.e. the simulation will be automatically stopped when the steady state is reached. **Steady-state RMS current variation tolerance** defines the acceptable variation of RMS current from one period to the next to determine whether the steady-state is reached.

The phase current of the machine depends on the **Target RMS supply current** field value and on the stator winding connection. Since the supply current is a line current, for star connection the phase current is equal to the supply current, for delta connection the phase current is equal to the supply current divided by $\sqrt{3}$. **Target advance angle** determines an angle between the current phasor and q-axis of the rotor as shown in Figure 2.6. For the current hysteresis and space vector PWM drive types, the current control algorithm is applied to adjust the inverter voltage in order to maintain d-axis and q-axis current components defined by the **Target RMS supply current** and **Target advance angle** field values. For the space vector PWM the field oriented current control is used, for the current hysteresis

PWM the hysteresis (bang-bang) current control is used. For the six-step drive the current is not controlled – the current is determined by the DC voltage, rotor speed, switch duty cycle (120 or 180 electrical degrees) and commutation advance angle.

Rotor speed dependency – specifies whether the rotor speed is fixed (when *Fixed speed simulation* is chosen) or varies depending on the load and electromagnetic torque of the motor (when *Variable speed simulation* is chosen). If *Fixed speed simulation* is chosen, you should also specify the speed in the field appearing to the right. If *Variable speed simulation* is chosen, the initial speed should be specified.

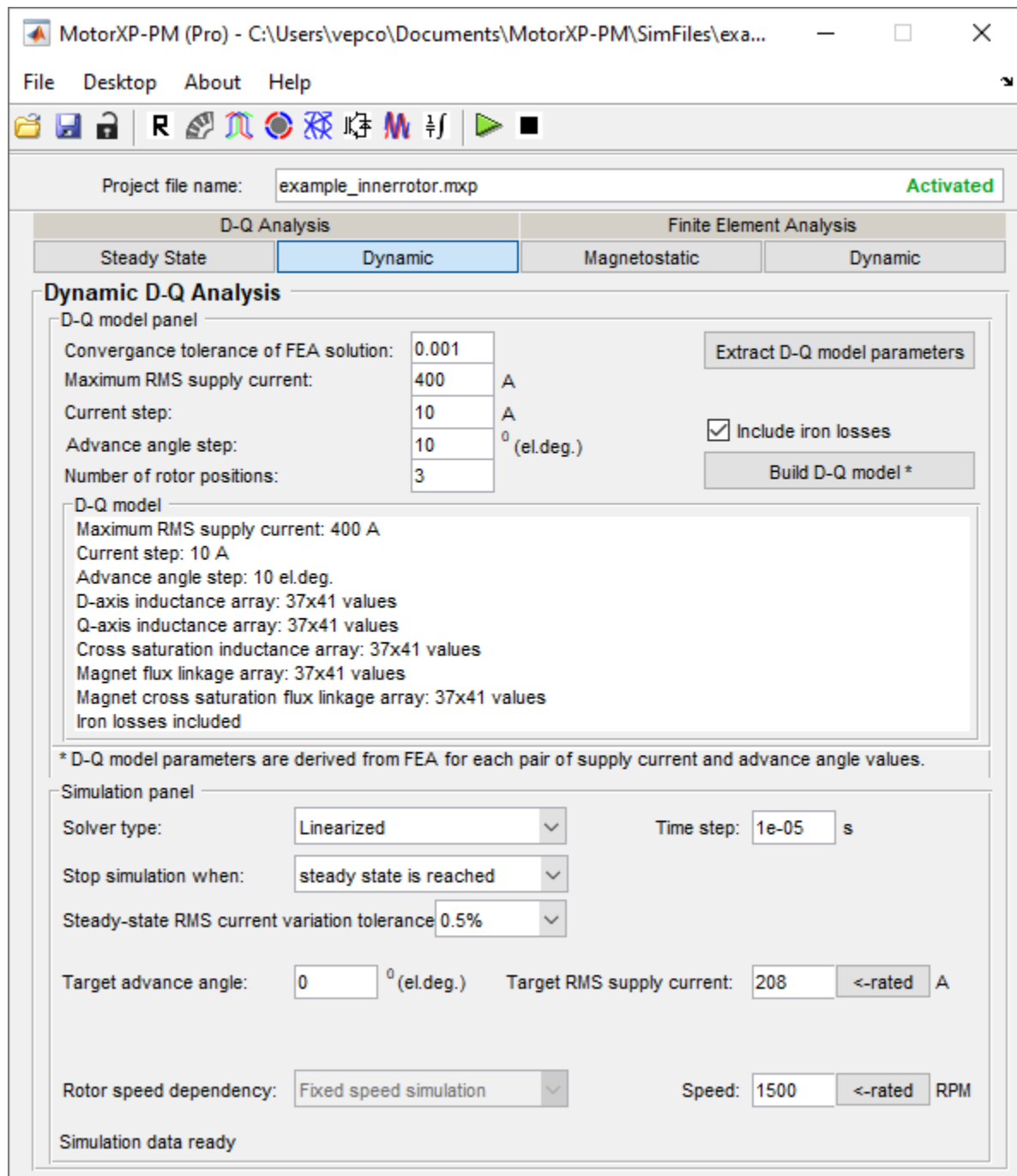


Figure 7.1. MotorXP-PM main window for Dynamic D-Q Analysis.

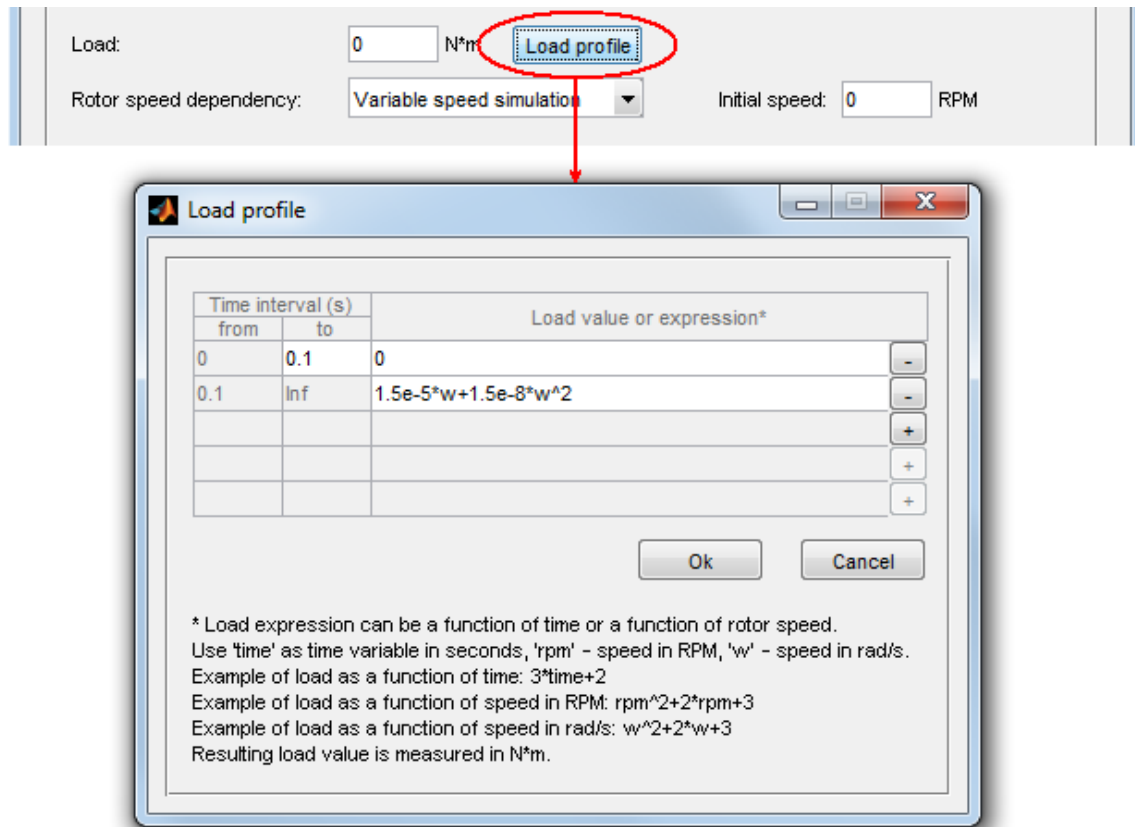


Figure 7.2. Specifying the load for the dynamic D-Q simulation.

Button '**<-rated**' to the right of some fields allows you to set up the corresponding parameter to its rated value specified in the **Rated Data** window.

If simulation with variable speed is used the load should be specified. The load can be specified either by a single value entered in the **Load** field or different load values can be specified for each time interval using the **Load profile** window as shown in Figure 7.2. Load can be a function of time or speed. Speed can be measured in rad/s or in RPM. Use +/- buttons to add or delete the time interval.

7.2. Viewing Dynamic D-Q Analysis results.

It is possible to view the **Dynamic D-Q Analysis** results as time-averaged quantities or waveform and spectrum plots. Use $\frac{1}{T} \int$ toolbar button to view time-averaged quantities as shown in Figure 7.3. The displayed quantities can be averaged over one, two or three electrical periods or the averaging time can be defined directly choosing *User choice* from the **Averaging time selection** pop-up menu.

The view of the **Plot Wizard** window when **Dynamic D-Q Analysis** is chosen is shown in Figure 7.4. It is organized as a standard MATLAB plotting construction consisting of a set of subplot and plot functions. **Subplot** checkboxes allow you to control the number of axes or rectangular panes displayed within a current figure window. The corresponding axes are activated or deactivated by a mouse click within a cell of the **Subplot** column. When the subplot is activated, the **change** button allows you to choose quantities to be plotted into the selected axes.

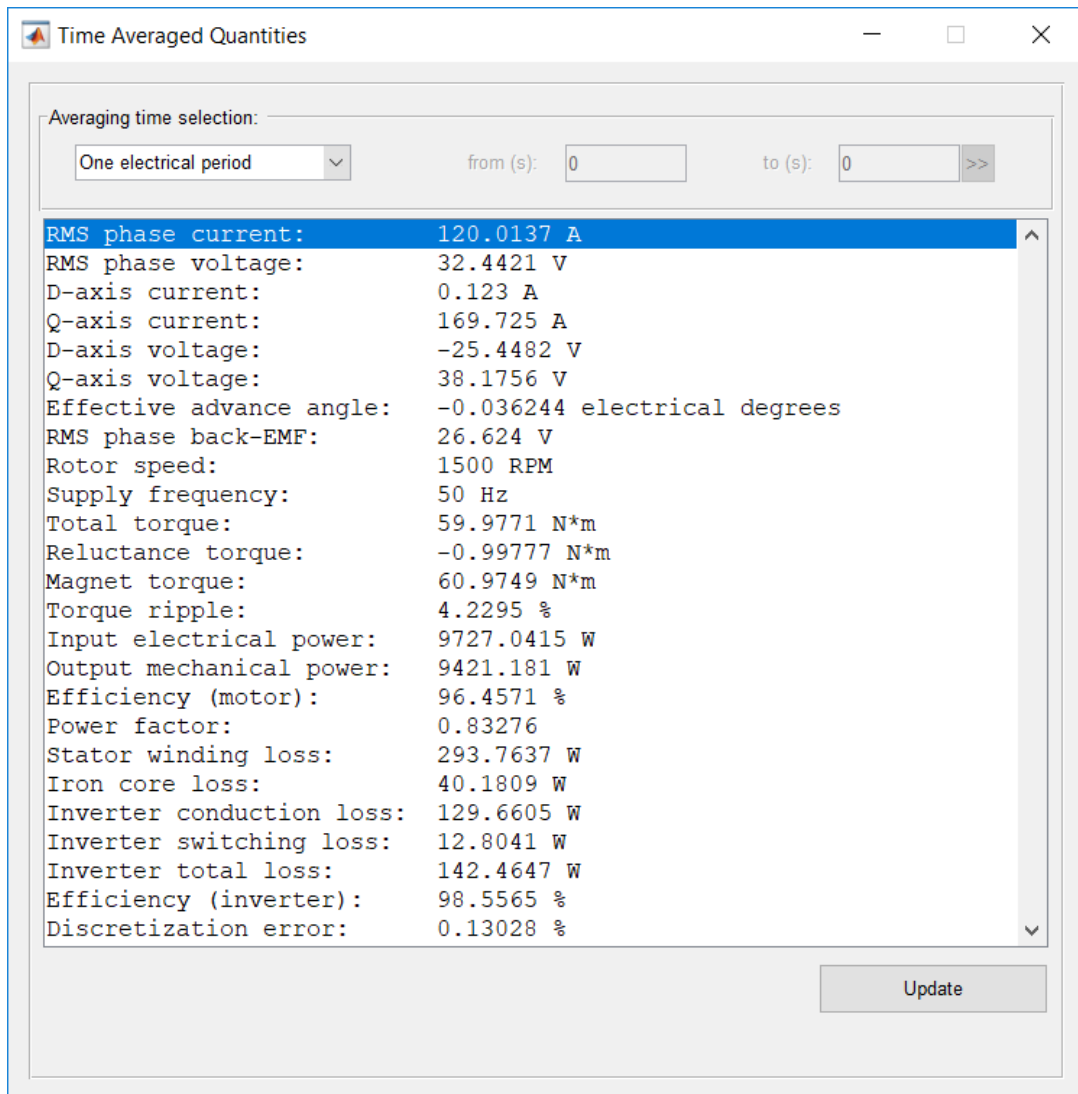


Figure 7.3. Time-averaged quantities.

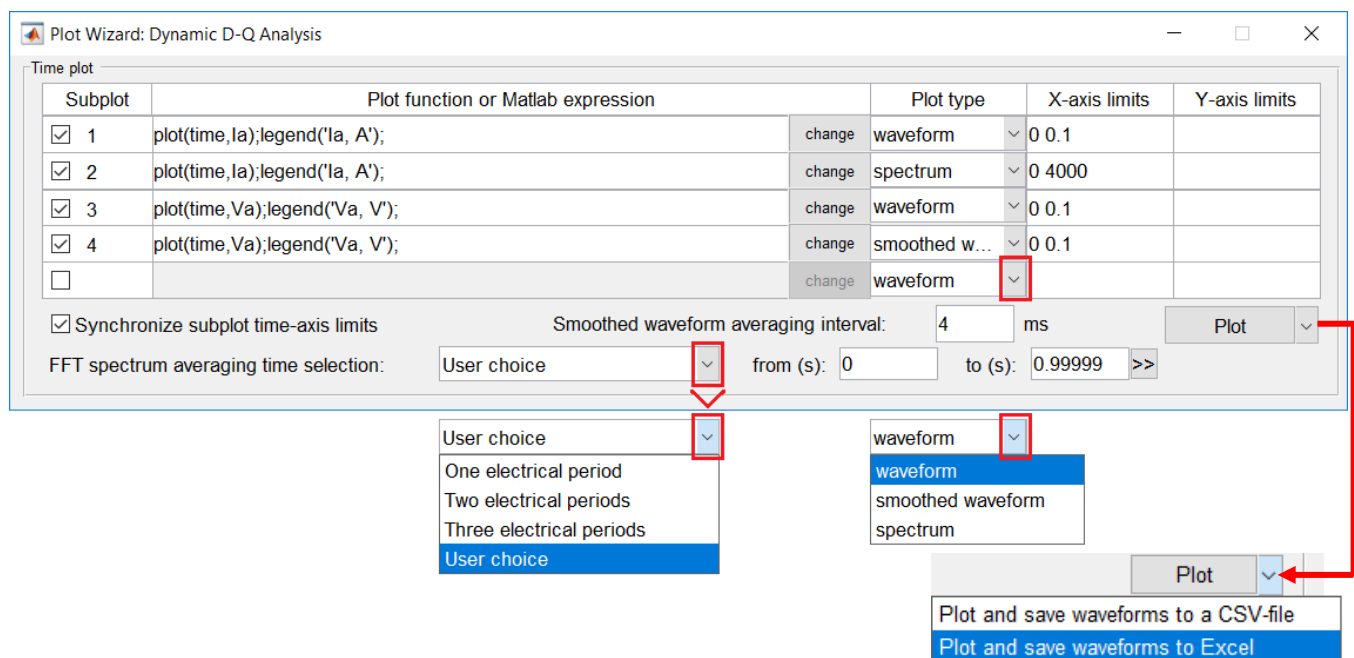


Figure 7.4. Dynamic D-Q Analysis Plot Wizard window.

The quantity to plot can be chosen from the dialog shown in Figure 7.5. Use Ctrl key to select several quantities. The list of available quantities is given below:

- Stator phase current (phase A, B, C);
- Phase current, d-axis;
- Phase current, q-axis;
- Phase voltage (phase A, B, C);
- Phase voltage, d-axis;
- Phase voltage, q-axis;
- Effective advance angle;
- Back-EMF (phase A, B, C);
- Back-EMF, d-axis;
- Back-EMF, q-axis;
- Electromagnetic torque (by flux linkage and current);
- Load torque on the motor shaft;
- Rotor speed;
- Rotor angular position;
- Input active electrical power;

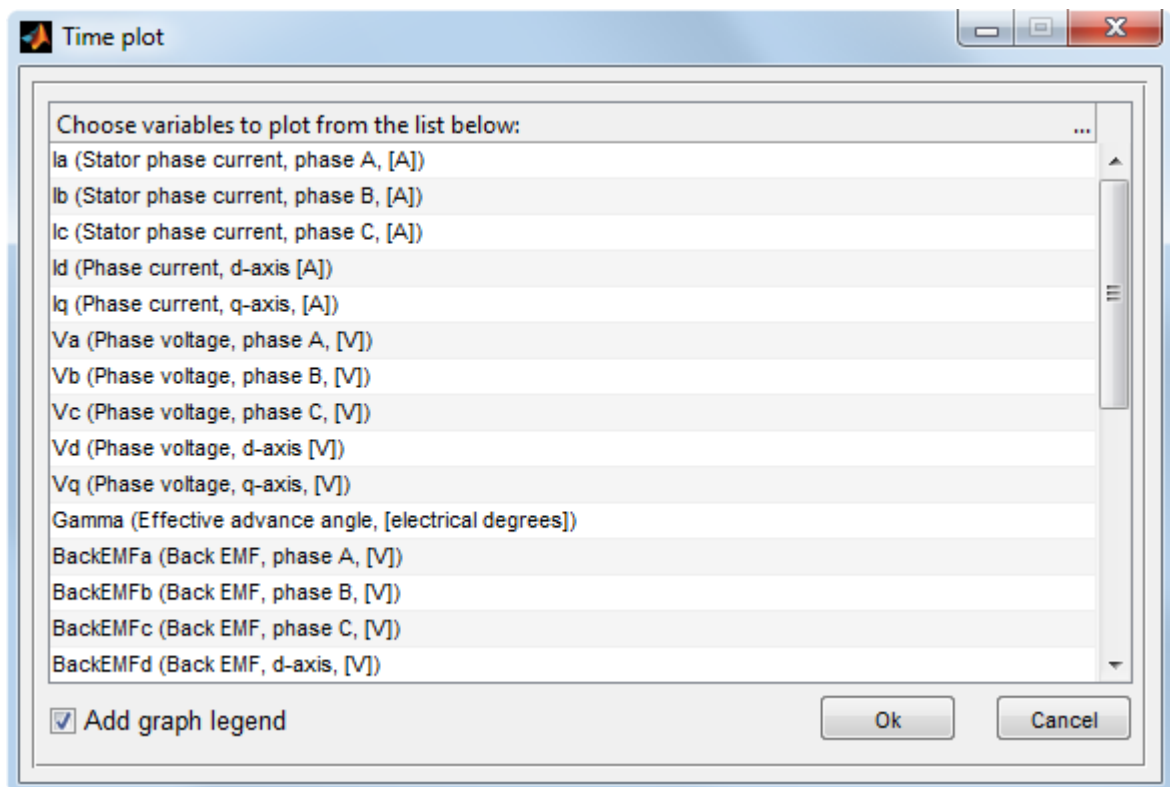


Figure 7.5. Choosing quantities to be plotted.

- Reactive electrical power;
- Mechanical power on the rotor shaft;
- Stator winding losses;
- Time derivative of the magnetic field energy;
- Flux linkage (phase A, B, C);
- Flux linkage, d-axis;
- Flux linkage, q-axis;
- Magnet torque (by flux linkage and current);
- Reluctance torque (by flux linkage and current);
- D-axis inductance;
- Q-axis inductance;
- Cross-saturation inductance;
- D-axis magnet flux linkage;
- Q-axis cross-saturation magnet flux linkage;

By clicking the **OK** button of the dialog (Figure 7.5), the MATLAB-expression is constructed to plot the selected quantities appearing in the corresponding line as shown in Figure 7.4 for plotting the phase A stator current waveform (first line), phase A stator current spectrum (second line), phase A stator voltage waveform (third line) and phase A stator voltage waveform after smoothing (fourth line). When the **Plot** button is clicked, the MATLAB figure window with corresponding plots of the selected quantities will appear (see Figure 7.6).

As it is seen, the plotting expression consists of the `plot` function with selected variables used as input arguments. If the **Add graph legend** checkbox of the dialog is checked, the `legend` function is added to the plotting expression so the graph legend of the corresponding axes will be shown. All plotting expressions are editable, so you can use any MATLAB plotting options and functions to change the way the plots appear on the screen. You can also change the `time` variable to any other variable you would like to use as an input argument of the `plot` function. There is also a list of additional variables which can be used within a plotting expression (see section 9.3).

X-axis limits and **Y-axis limits** fields allow you to set the x-axis and y-axis limits, respectively, to the specified values. Two limit values within a cell are separated by a space, comma ‘,’ or semicolon ‘;’. If the cell is empty, the limits of the corresponding axis will be chosen automatically. If the **Synchronize subplot time-axis limits** checkbox is checked, all subplots of the figure will have identical limits along the time-axis when you zoom or pan one of the subplots of the figure.

There are several plotting options available from the **Plot type** pop-up menu: *waveform*, *smoothed waveform* and *spectrum* (see Figure 7.4). Smoothed waveform plot type can be useful when plotting the

pulse width modulated waveforms to filter out the PWM carrier frequency and obtain the smooth waveform of the signal (compare subplots 3 and 4 of Figure 7.6). The **Smoothed waveform averaging interval** field specifies the sliding window width of the smoothing algorithm. The resolution of the frequency spectrum, plotted when *spectrum* is chosen from the **Plot type** pop-up menu, is controlled by the **FFT spectrum averaging time selection** pop-up menu. The spectrum can be calculated over one, two or three electrical periods or the time can be defined directly choosing *User choice* from the **FFT spectrum averaging time selection** pop-up menu (see Figure 7.4).

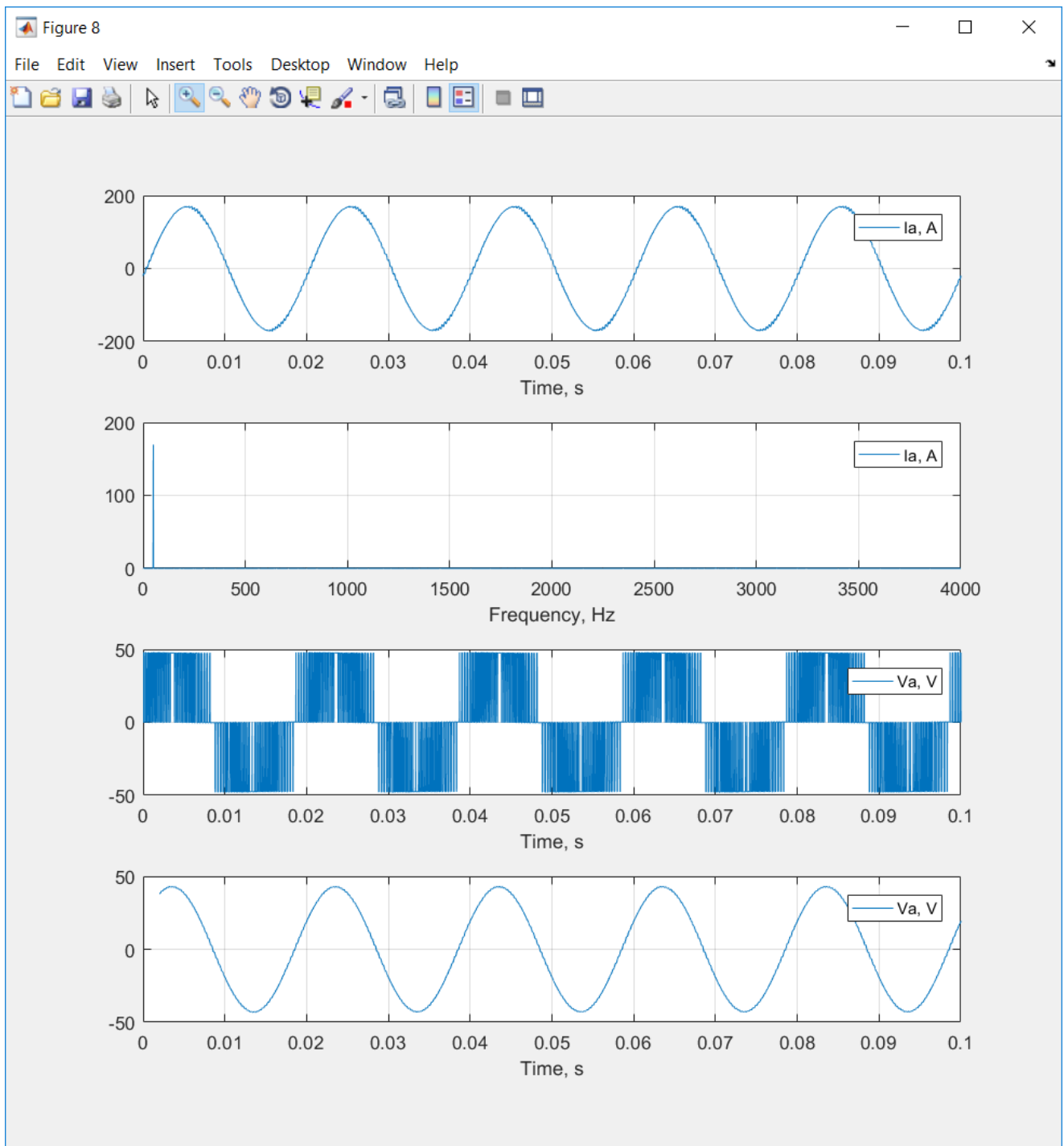




Figure 7.6. MATLAB figure window with plots of the selected quantities.

8. DYNAMIC FINITE ELEMENT ANALYSIS

Dynamic FE Analysis is the most powerful and most accurate of all the analysis methods taking into account rotation of the rotor, eddy currents, higher harmonics, PWM switching and etc. It simulates the machine connected to a power supply electronic circuit with the time-stepping transient finite element method. By default, the electrical circuit is a three-phase inverter as shown in Figure 10.1, but any power supply circuit can be used (see chapter 10). However, this type of analysis takes considerably more time compared with other analysis types.

8.1. Running Dynamic FE Analysis.

The view of the main window when **Dynamic FE Analysis** is chosen is shown in Figure 8.1. To start the simulation, click the **Start dynamic FE simulation** button  on the MotorXP-PM main window toolbar. The first run of the simulation begins at zero time. Every subsequent run of the simulation resumes from the time where it was previously paused. The simulation continues, until you pause the simulation, until the simulation reaches **Simulation stop time** or until an error occurs. The first run of the simulation starts with the initialization which may take a few minutes. To pause the simulation, click the **Pause simulation** button  which is to the right of the **Start dynamic FE simulation** button. The simulation will be paused when the calculation of the current time step is completed. It is not recommended to interrupt the simulation using Ctrl+C shortcut otherwise the project file may be corrupted, and all data will be lost.

The **solver type** specifies whether linear or nonlinear FEA is used. Using linear solver type is recommended only for testing purposes. **Convergence tolerance** specifies the accuracy of FEA solution as defined by Exp. 2.5.

Simulation settings field specifies either the default **Simulation script file** and the default **Stator electrical circuit file** are used (if *General* is chosen) or the simulation script and the stator circuit are defined by the user (if *Advanced* is chosen). **Simulation script file** is a MATLAB-function which is called on each simulation time step and allows you to change all general simulation settings, compute and store user defined variables, control power supply sources and electronic switches, implement user defined motor control algorithms. Refer to chapter 9 for more details on using simulation script files. **Stator electrical circuit file** is a MATLAB-function defining the coupled electrical circuit which is solved simultaneously with the magnetic field. Refer to chapter 10 for more details on using stator electrical circuits.

Be aware that the **Time step** value should be small enough compared with the PWM sampling period to get accurate results. Control the **Discretization error** in the **Time Average Quantities** window shown in Figure 8.3 to adjust the time step. It is recommended that the discretization error does not exceed 1%.

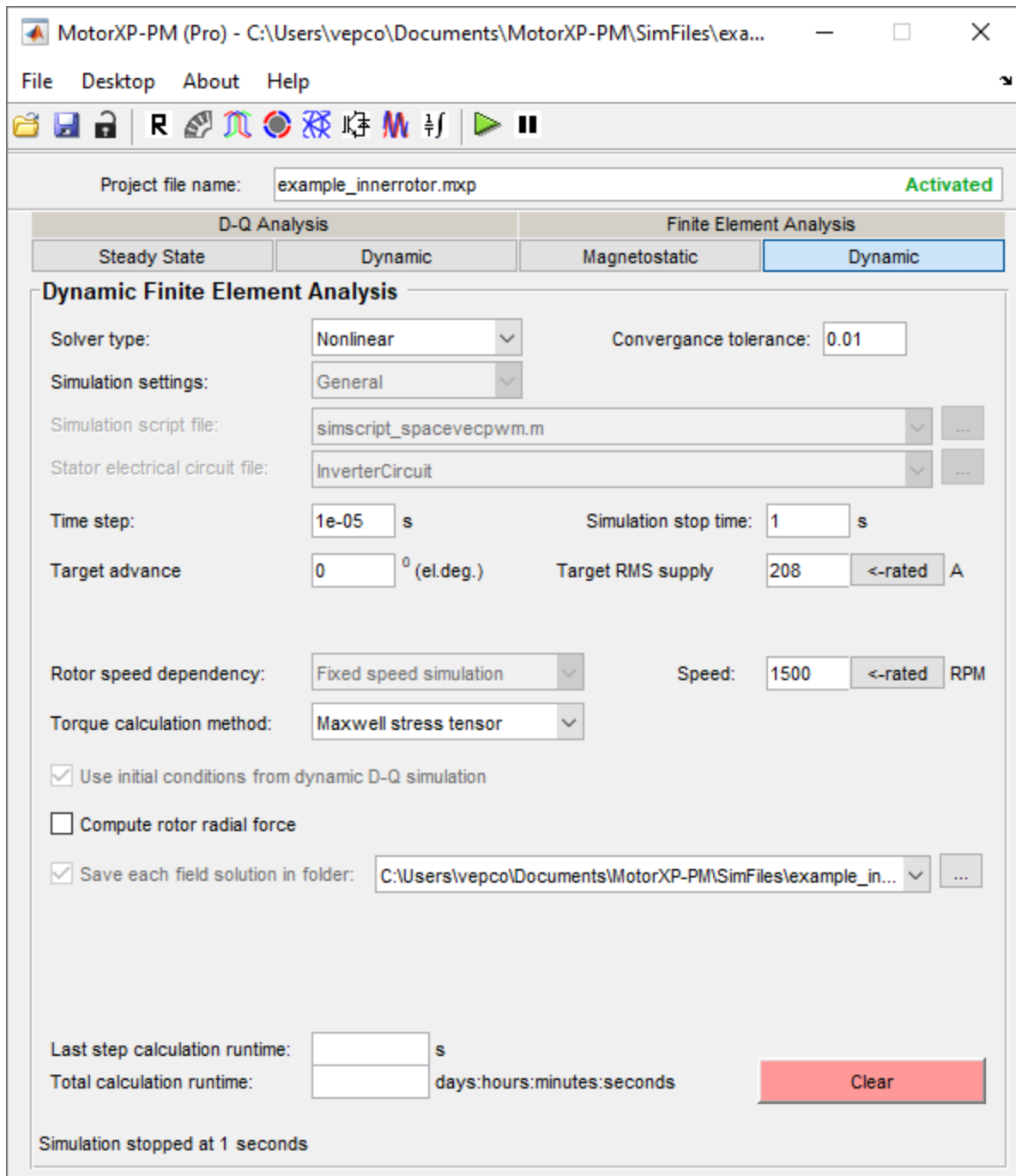


Figure 8.1. MotorXP-PM main window for Dynamic FE Analysis.

The phase current of the machine depends on the **Target RMS supply current** field value and on the stator winding connection. Since the supply current is a line current, for star connection the phase current is equal to the supply current, for delta connection the phase current is equal to the supply current divided by $\sqrt{3}$. **Target advance angle** determines an angle between the current phasor and q-axis of the rotor as shown in Figure 2.6. For the current hysteresis and space vector PWM drive types, the current control algorithm is applied to adjust the inverter voltage to maintain d-axis and q-axis current components defined by the **Target RMS supply current** and **Target advance angle** field values. For the space vector PWM the field oriented current control is used, for the current hysteresis PWM the hysteresis (bang-bang)

current control is used. For the six-step drive the current is not controlled – the current is determined by the DC voltage, rotor speed, switch duty cycle (120 or 180 electrical degrees) and commutation advance angle.

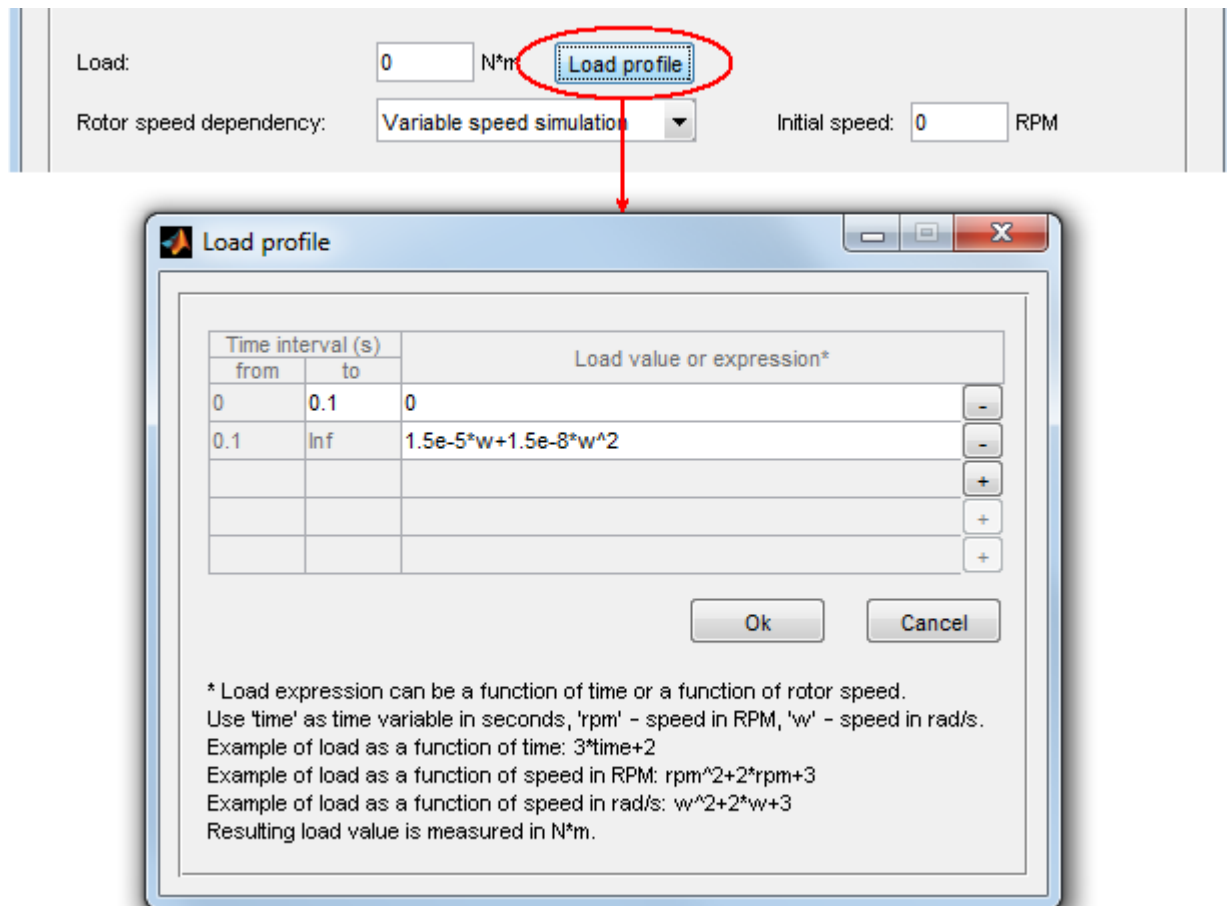


Figure 8.2. Specifying the load for the dynamic FE simulation.

Rotor speed dependency – specifies whether the rotor speed is fixed (when *Fixed speed simulation* is chosen) or varies depending on the load and electromagnetic torque of the motor (when *Variable speed simulation* is chosen). If *Fixed speed simulation* is chosen, you should also specify the speed in the field appearing to the right. If *Variable speed simulation* is chosen, the initial speed should be specified.

If simulation with variable speed is used the load should be specified. The load can be specified either by a single value entered in the **Load** field or different load values can be specified for each time interval using the **Load profile** window as shown in Figure 8.2. Load can be a function of time or speed. Speed can be measured in rad/s or in RPM. Use +/- buttons to add or delete the time interval.

Button '<-rated' to the right of some fields allows you to set up the corresponding parameter to its rated value specified in the **Rated Data** window.

Torque calculation method – two torque calculation methods are available: *Maxwell stress tensor* and *Virtual work*. The Maxwell stress tensor method is usually used. Refer to section 2.3 for more details.

If **Use initial conditions from dynamic D-Q simulation** is checked the FE simulation is started with the initial state (initial values of magnetic field, currents and voltages) computed using the dynamic D-Q model so the dynamic FE simulation reaches the steady-state in less number of time steps.

Compute rotor radial force checkbox enables (if checked) to compute the x-axis and y-axis radial force components between the stator and rotor (integrated along the airgap center line).

Save each field solution checkbox enables (if checked) to store additional data of each FEA solution such as magnetic vector potential distribution and permeability distribution values. This data is not saved by default because of the large amount of hard drive space required. Data for each time step is saved in a separate file so the number of files saved is equal to the number of computed time steps.

Refer to section 8.3 to learn about **Multiple harmonics iron loss analysis**.

Use **Clear** button to delete all dynamic FEA simulation data and start the simulation from zero time.

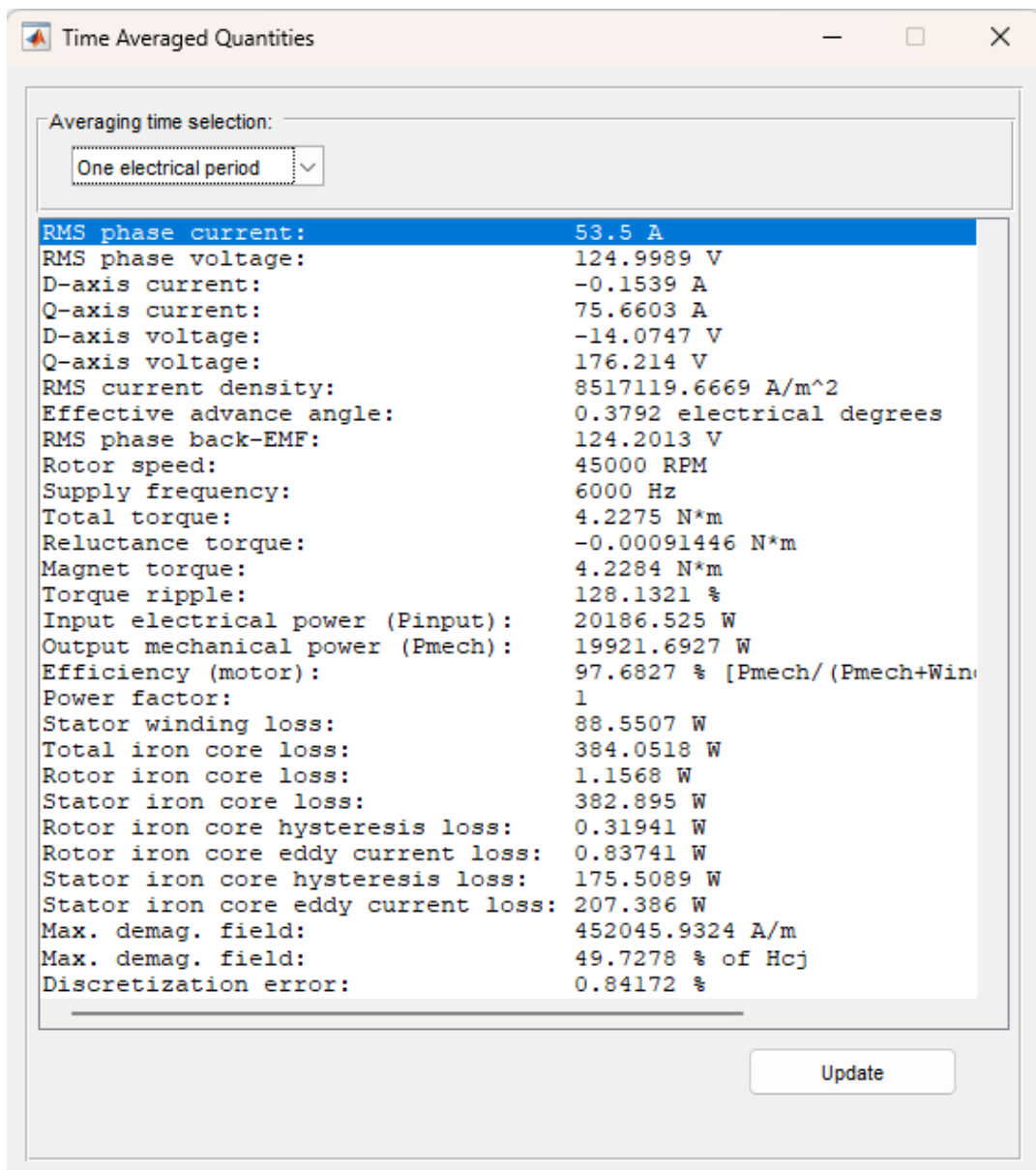



Figure 8.3. Time averaged quantities.

8.2. Viewing Dynamic FE Analysis results.

It is possible to view the **Dynamic FE Analysis** results as time-averaged quantities or plots.

8.2.1. Time-averaged quantities.

Use  toolbar button to view time-averaged quantities as shown in Figure 8.3. The displayed quantities can be averaged over one, two or three electrical periods or the averaging time can be defined directly choosing *User choice* from the **Averaging time selection** pop-up menu.

For more details on the interpretation of the maximum demagnetizing field results see section 2.7.

8.2.2. Plot wizard.

The view of the **Plot Wizard** window when **Dynamic FE Analysis** is chosen is shown in Figure 8.4. Several plot types are available for **Dynamic FE Analysis**:

- time-series data plot and frequency spectrum of the time-series data;
- air gap distribution plot and frequency spectrum of the air gap distribution;
- cross-section distribution plot;
- animation.

Plot Wizard also allows saving waveforms to a CSV-file (text file with comma-separated values) or to a Microsoft® Excel® spreadsheet file (Figure 8.4).

8.2.3. Time plots.

Time-series data plots are available from the **Time plot** panel of the **Plot Wizard** window. It is organized as a standard MATLAB plotting construction consisting of a set of `subplot` and `plot` functions. **Subplot** checkboxes allow you to control the number of axes or rectangular panes displayed within a current figure window. The corresponding axes are activated or deactivated by a mouse click within a cell of the **Subplot** column. When the subplot is activated, the **change** button allows you to choose quantities to be plotted into the selected axes. Quantities can be chosen from the dialog shown in Figure 8.5. Use Ctrl key to select several quantities. The list of available quantities is given below:

- Stator phase current (phase A, B, C, d-axis, q-axis);
- Phase voltage (phase A, B, C, d-axis, q-axis);
- Effective advance angle;
- Back-EMF (phase A, B, C, d-axis, q-axis);
- Input apparent electrical power;
- Consumed apparent power;
- Apparent power (real and reactive) consumed by a stator circuit;
- Mechanical power on the rotor shaft;
- Time derivative of the magnetic field energy;
- Electromagnetic torque;

- Load torque on the motor shaft;
- Rotor speed;
- Rotor angular position;
- Electromagnetic torque by Maxwell stress tensor;
- Electromagnetic torque by virtual work method;
- Electromagnetic torque by flux linkage and current;
- Magnet torque (by Maxwell stress tensor);
- Reluctance torque (by Maxwell stress tensor);
- Cogging torque (by Maxwell stress tensor);
- Flux linkage (phase A, B, C, d-axis, q-axis);
- Radial electromagnetic force between stator and rotor along x-direction;
- Radial electromagnetic force between stator and rotor along y-direction;

Plot Wizard: Dynamic FE Analysis

Time plot

Subplot	Plot function or Matlab expression	Plot type	X-axis limits	Y-axis limits
<input checked="" type="checkbox"/> 1	plot(time, Ia); legend('Ia, A'); change	waveform	0.1	-250 250
<input checked="" type="checkbox"/> 2	plot(time, Ia); legend('Ia, A'); change	spectrum	0 500	
<input checked="" type="checkbox"/> 3	plot(time, Va); legend('Va, V'); change	waveform	0.1	
<input checked="" type="checkbox"/> 4	plot(time, Va); legend('Va, V'); change	smoothed...	0.1	
<input type="checkbox"/>	change	waveform		

☒ Synchronize subplot time-axis limits Smoothed waveform averaging interval: 1 ms [Plot](#)

FFT spectrum averaging time selection: User choice from (s): 0 to (s): 1 >>

Air gap distribution plot

Subplot	Variables	Time (s)	Slice	Plot type	X-axis limits	Y-axis limits
<input checked="" type="checkbox"/> 1	flux	+ < 1.00000 > >> 1	1	distribution		
<input checked="" type="checkbox"/> 2	flux	+ < 1.00000 > >> 1	1	spectrum	50	
<input type="checkbox"/>		+ < 1.00000 > >> 1	1	distribution		
<input type="checkbox"/>		+ < 1.00000 > >> 1	1	distribution		
<input type="checkbox"/>		+ < 1.00000 > >> 1	1	distribution		

☒ Show graph legend [Plot](#)

Cross-section distribution plot

Figure	Plotted quantity	Time (s)	Slice	Options	X-axis limits	Y-axis limits	Z-axis limits
<input checked="" type="checkbox"/> 1	Magnetic flux density, [T]	< 1.00000 > >> 1	1	flux lines			
<input checked="" type="checkbox"/> 2	Relative permeability	< 1.00000 > >> 1	1	flux lines			
<input checked="" type="checkbox"/> 3	Magnetic field intensity, [A/m]	< 1.00000 > >> 1	1	flux lines			
<input type="checkbox"/>	None	< 1.00000 > >> 1	1	none			
<input type="checkbox"/>	None	< 1.00000 > >> 1	1	none			

Number of flux line levels: 20 ☐ Full cross-section view [Plot](#)

Animated plot

☒ Animate air gap distribution subplots Skip: 0 files Animation start time: 0.00001 s

☒ Animate cross-section distribution figures Frame display time: 0 ms Animation stop time: 1.00000 s

☒ Position figures [Pause](#) [Start animation](#)

Data source folder: SimFiles\example_innerrtor_dynFEdata

[Plot](#)

[Plot and save waveforms to a CSV-file](#)

[Plot and save waveforms to Excel](#)

Figure 8.4. Dynamic FE Analysis Plot Wizard window and export data to file options.

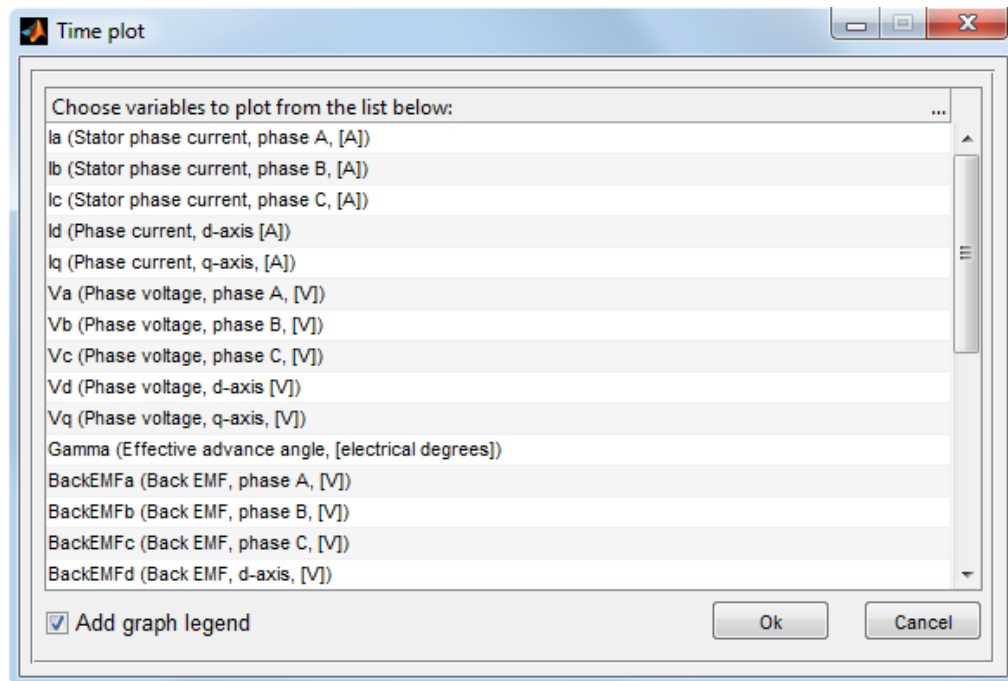


Figure 8.5. Choosing quantities to be plotted.

All user defined variables stored in the `Userdata` structure appear at the end of the list shown in Figure 8.5. For more details on saving your own variables and using the `Userdata` structure refer to section 9.4.

By clicking the **OK** button of the dialog (Figure 8.5), the MATLAB-expression is constructed to plot the selected quantities appearing in the corresponding line as shown in Figure 8.6 for plotting phase A stator current (first line), phase A stator current spectrum (second line), phase A stator voltage waveform (third line) and phase A stator voltage waveform after smoothing (fourth line).

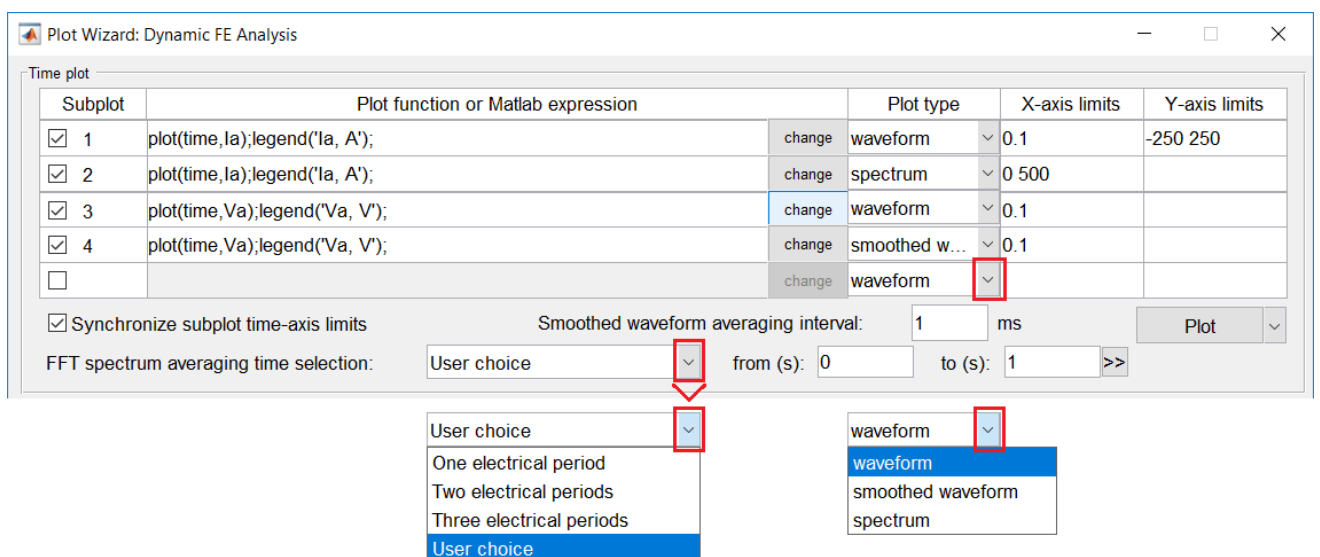


Figure 8.6. Example of using **Plot Wizard** for creating time plots.

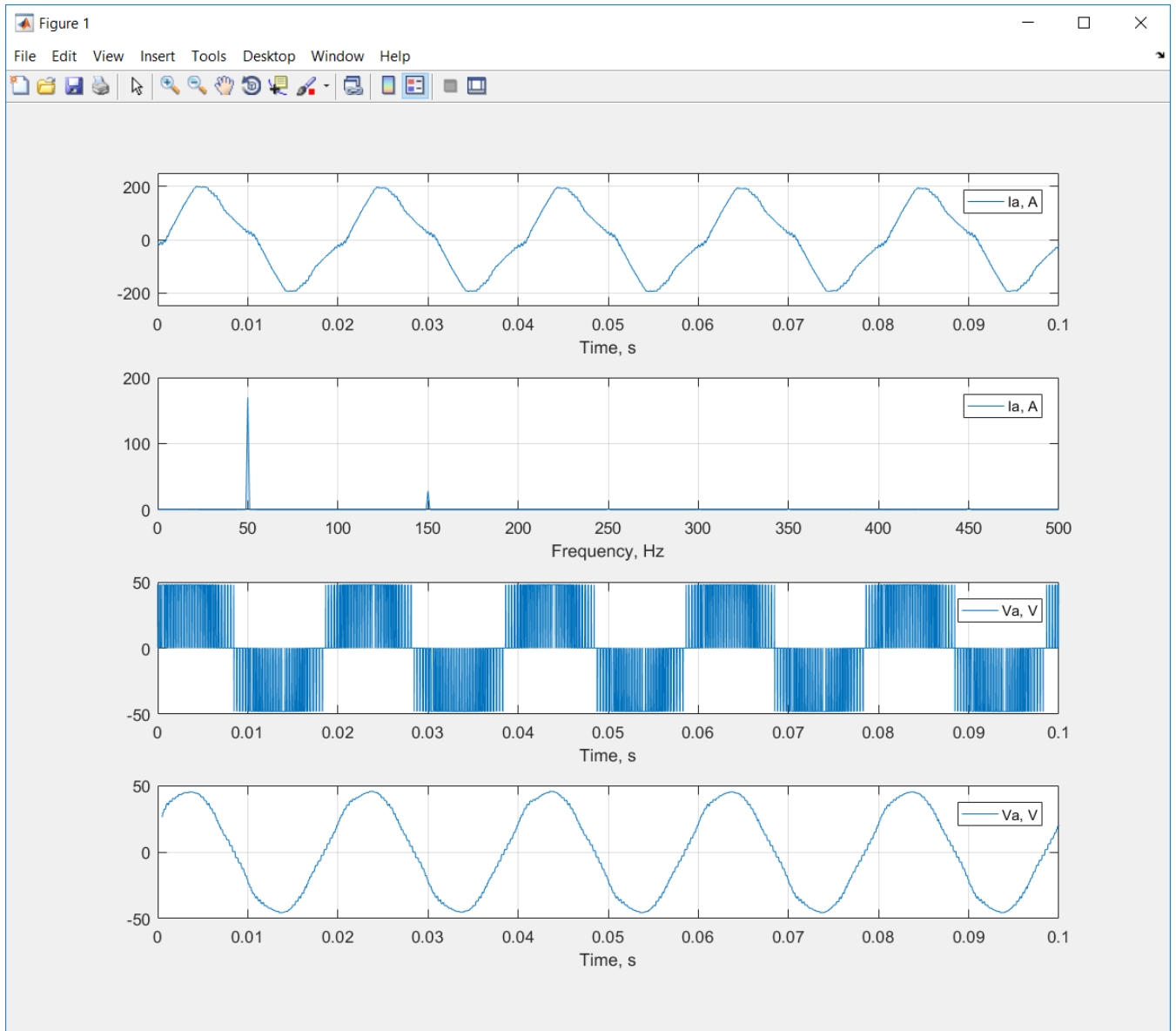


Figure 8.7. MATLAB figure window with time plots of the selected quantities.

When the **Plot** button is clicked, the MATLAB figure window with plots of the selected quantities will appear (see Figure 8.7).

As it is seen, the plotting expression consists of the `plot` function with selected variables used as input arguments. If the **Add graph legend** checkbox of the dialog is checked, the `legend` function is added to the plotting expression so the graph legend of the corresponding axes will be shown. All plotting expressions are editable, so you can use any MATLAB plotting options and functions to change the way the plots appear on the screen. You can also change the `time` variable to any other variable you would like to use as an input argument of the `plot` function. There is also a list of additional variables which can be used within a plotting expression (see section 9.3).

X-axis limits and **Y-axis limits** fields of the **Time plot** panel allow you to set the x-axis and y-axis limits, respectively, to the specified values. Two limit values within a cell are separated by a space, comma ‘,’

or semicolon ‘;’. If a cell is empty, the limits of the corresponding axis will be chosen automatically. If the **Synchronize subplot time-axis limits** checkbox is checked, all subplots of the figure will have identical limits along the time-axis when you zoom or pan one of the subplots of the figure.

There are several plotting options available from the **Plot type** pop-up menu: *waveform*, *smoothed waveform* and *spectrum* (see Figure 8.6). Smoothed waveform plot type can be useful when plotting the pulse width modulated waveforms to filter out the PWM carrier frequency and obtain the original waveform of the signal (compare subplots 3 and 4 of Figure 8.7). The **Smoothed waveform averaging interval** field specifies the sliding window width of the smoothing algorithm. The resolution of the frequency spectrum, plotted when *spectrum* is chosen from the **Plot type** pop-up menu, is controlled by the **FFT spectrum averaging time selection** pop-up menu. The spectrum can be calculated over one, two or three electrical periods or the time can be defined directly choosing *User choice* from the **FFT spectrum averaging time selection** pop-up menu (see Figure 8.6).

8.2.4. Air gap distribution plots.

Air gap distribution plots allow you to view the distribution of the particular quantity over the machine’s air gap as well as its frequency spectrum. It is available from the **Air gap distribution plot** panel. **Subplot** checkboxes allow you to control the number of axes or rectangular panes displayed within a current figure window. The corresponding axes are activated or deactivated by a mouse click within a cell of the **Subplot** column. When the subplot is activated, the “+” button allows you to choose quantities to be plotted from the dialog shown in Figure 8.8. Use Ctrl key to select several quantities.

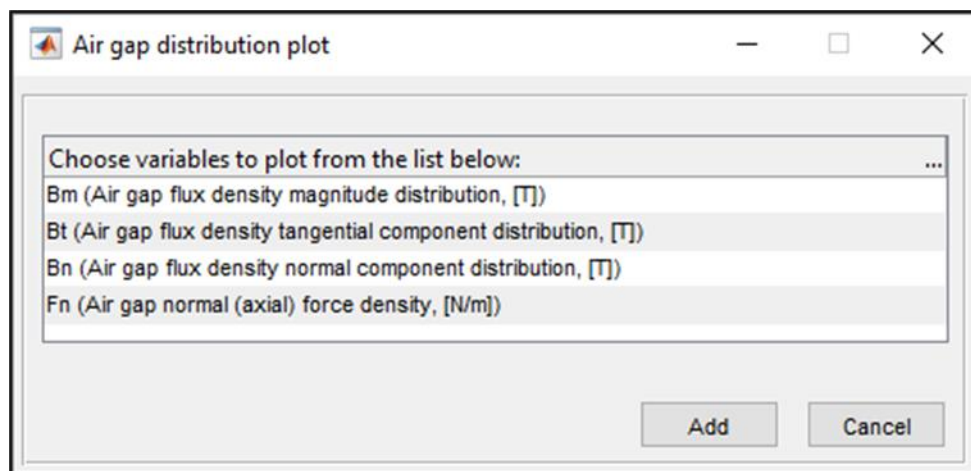


Figure 8.8. Choosing quantities to be plotted.

The following quantities are listed in the dialog of Figure 8.8:

Quantity	Comments
Bm (Air gap flux density magnitude distribution, [T])	Defined as $B_m = \sqrt{B_t^2 + B_n^2}$ where B_n and B_t – normal and tangential components of the magnetic flux density in the center of the air gap, as shown in Figure 2.4.
Bt (Air gap flux density tangential component distribution, [T])	B_n and B_t – normal and tangential components of the magnetic flux density in the center of the air gap, as shown in Figure 2.4.
Bn (Air gap flux density normal component distribution, [T])	
Fn (Air gap normal (radial) force distribution, [N])	According to Maxwell stress tensor method the normal force in each point of the round path can be expressed as the following: $F_n = -\frac{B_n^2 - B_t^2}{2\mu_0} d \cdot l$, where l – lamination length in z direction, d – length of the path in the center of the air gap between two consecutive points, μ_0 – permeability of the free space, B_n and B_t – normal and tangential components of the magnetic flux density in the center of the air gap, as shown in Figure 2.4.

By clicking the **Add** button of the dialog (Figure 8.8), the selected quantities are added to the corresponding line separated by commas as shown in Figure 8.9. Each line of the **Variables** column is editable, so you can use MATLAB arithmetic operations to obtain desired plots. For example, the second and third lines in Figure 8.9 produce plots of the tangential air gap force density distribution and the radial air gap force density distribution, respectively, where μ_0 – variable corresponding to the permeability of free space. According to Exp. 2.6 the tangential air gap force produces the electromagnetic torque so the plot of the electromagnetic torque distribution over an air gap can be obtained.

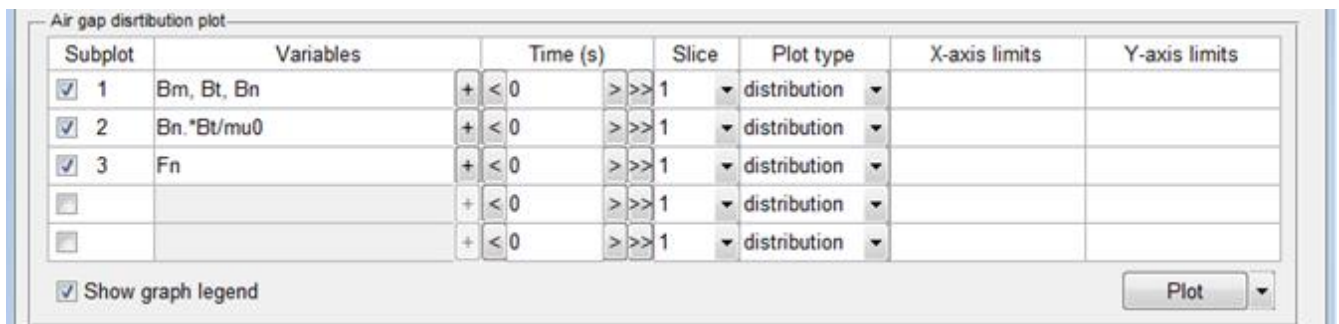


Figure 8.9. Example of using **Plot Wizard** for creating air gap plots.

Time fields of the **Air gap distribution plot** panel allow you to specify the time point which the selected quantities are plotted for. By default, the last computed time point is set up. Plotting for the time points

other than the last computed time point is only possible if the file containing data for the desired time point was previously saved. These data-files are being saved when **Save each field solution** is checked in the MotorXP-PM main window (Figure 8.1). So, if you are interested in viewing the air gap distribution plots for different time points make sure that the corresponding data-files are being saved. The folder used as a source of data-files is specified in the **Data source folder** field located at the bottom of the **Plot Wizard** window (Figure 8.4). You can change it by clicking the button to the right, if needed.

X-axis limits and **Y-axis limits** fields allow you to set the x-axis and y-axis limits, respectively, to the specified values in the same way as for the **Time plot** panel. **Slice** fields allow you to choose the machine's cross-section which the selected quantities are plotted for, if several slices are used (see section 2.4 for more details on multi-slice simulations).

Besides the air gap distribution, it is also possible to calculate the air gap harmonic components of the selected quantities. To obtain the frequency spectrum, select the *spectrum* item in the corresponding **Plot type** pop-up menu. An example of plotting the air gap distribution of the normal flux density component and its spectrum is shown in Figure 8.10. Only the first 100 harmonics are shown, since the value specified in the corresponding **X-axis limits** field is 100 (if the minimum limit equals to 0, it can be omitted).

The **Show graph legend** field allows you to choose whether the graph legend will be displayed.

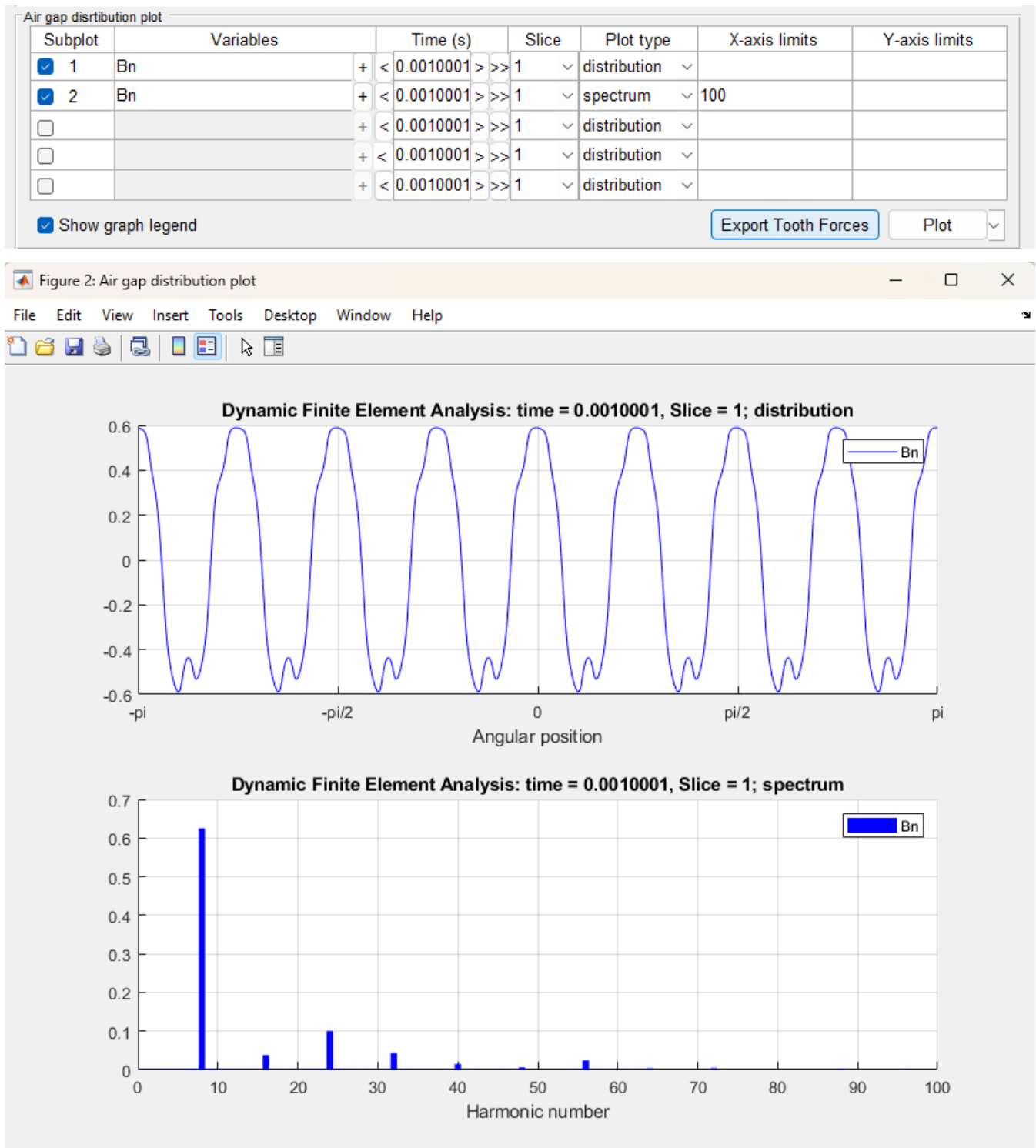


Figure 8.10. Plotting of the air gap distribution of the normal flux density component and its spectrum.

8.2.5. Cross-section distribution plots.

Cross-section distribution plots are available from the **Cross-section distribution plot** panel of the **Plot Wizard** window (Figure 8.4). As opposed to two previous plot types, the cross-section distribution for each quantity is plotted in a separated window which contains only one axes. **Figure** checkboxes allow you to control the number of windows appearing when the **Plot** button is clicked. The corresponding figure is activated or deactivated by a mouse click within a cell of the **Figure** column. When the figure is

activated, the corresponding **Plotted quantity** pop-up menu allows you to choose the quantity to be plotted. If *None* is chosen, the machine's cross-section geometry will be plotted.

The following items are available from the **Plotted quantity** pop-up menu:

- Magnetic vector potential, [T*m];
- Magnetic flux density, [T];
- Magnetic field intensity, [A/m];
- Relative permeability;
- Stator current density, [A/m²];
- Squared flux density, [T];
- Magnetic field energy density, [J/m³];
- Stator loss density, [W/m³];
- Iron loss density, [W/m³];
- Total loss density (stator+iron), [W/m³];
- Demagnetizing field, [A/m];
- Demagnetizing field, [% of H_{cj}];
- Finite element mesh.

Time fields of the **Cross-section distribution plot** panel allow you to specify the time point which the selected quantity is plotted for. By default, the last computed time point is set up. Plotting for the time points other than the last computed time point is only possible if the file containing data for the desired time point was previously saved. These data-files are saved when **Save each field solution** is checked in the MotorXP-PM main window (Figure 8.1). So, if you are interested in viewing the cross-section distribution plots for different time points make sure that the corresponding data-files are being saved. The folder used as a source of data-files is specified in the **Data source folder** field located at the bottom of the **Plot Wizard** window (Figure 8.4). You can change it by clicking the button to the right, if needed.

Slice fields allow you to choose the machine's cross-section which the selected quantity is plotted for, if several slices are used (see section 2.4 for more details on multi-slice simulations).

The **Options** field allows you to show the magnetic flux lines (if *flux lines* is chosen) or magnetic flux arrows (if *flux arrows* is chosen) on the corresponding plot. Flux arrows are plotted such that the direction of the arrow indicates the direction of the flux, and the size of the arrow indicates the magnitude of the flux density. The **number of flux line levels** field allows you to alter the flux lines density. If periodic/antiperiodic boundary conditions are used, by default, only part of the cross-section (as it appears in **Mesh Editor**) will be plotted. Check the **Full cross-section view** checkbox if you want to plot the whole cross-section.

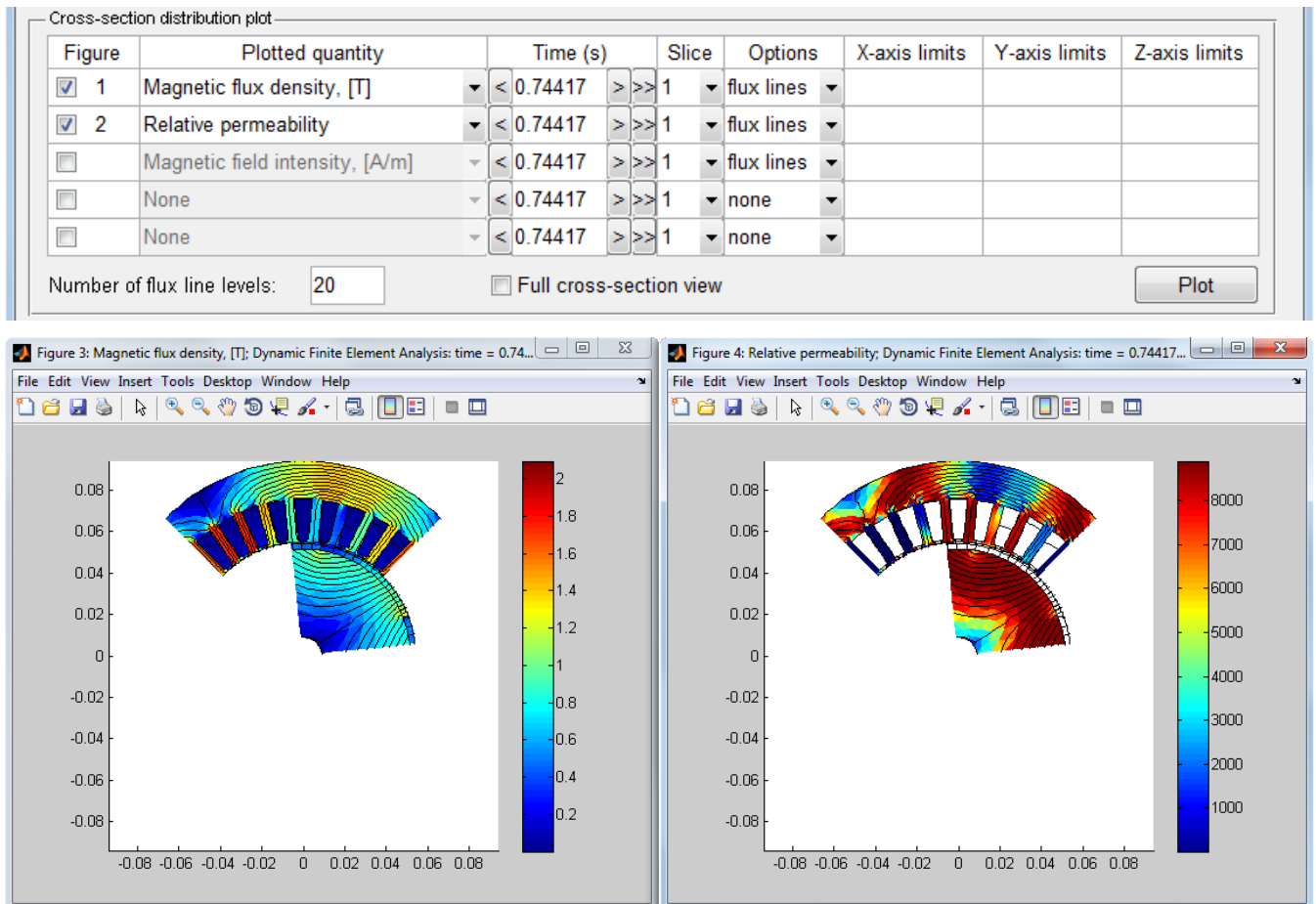


Figure 8.11. Example of using **Plot Wizard** for creating cross-section distribution plots.

X-axis limits and **Y-axis limits** fields allow you to set the x-axis and y-axis limits, respectively. If x-axis and y-axis limits are not specified, their values will be determined by the size of the machine's cross-section. The **Z-axis limits** field sets the color scale limits to specified minimum and maximum values separated by a space, comma ',' or semicolon ';'. Values between z-axis limits are linearly mapped to the used color scale (colormap). Data values smaller or greater than specified z-axis limits are mapped to the minimum limit or to the maximum limit, respectively.

An example of plotting the cross-section distribution of the magnetic flux density and relative permeability is shown in Figure 8.11. When the **Plot** button is clicked, two windows appear. As it is seen, the **flux lines** option is chosen for both figures, so the flux lines are additionally shown. Since the third figure is not active, the Magnetic field intensity is not plotted. Note that only part of the machine's cross-section is shown, since **Full cross-section view** is not checked.

8.2.6. Animation.

Animated plot panel is used to create animated sequences of the air gap distribution plots and/or the cross-section distribution plots (Figure 8.4). **Animate air gap distribution subplots** and **Animate cross-section distribution figures** checkboxes allow you to choose plot types to be animated. If **Animate air**

gap distribution subplots is checked, all selected subplots of the **Air gap distribution plot** panel will be involved in the animation. Similarly, if **Animate cross-section distribution figures** is checked, all selected figures of the **Cross-section distribution plot** panel will be involved in the animation. If **Position figures** control is checked, the size and location on the screen for all figure windows will be determined automatically so that the windows fit best the screen size.

Skip and **Frame display time** fields allow you to specify the way frames change each other while the animation is in progress. **Skip** field specifies the number of data-files or frames to be skipped. So, if **Skip** field is set to 0, data for each time step will be shown on the screen, if **Skip** field is set to 1, data for each second time step will be shown if **Skip** field is set to 2 – for each third time step and so on. The **Frame display time** field specifies the time each frame is displayed on the screen. Note that the least time the frame is displayed is limited by the animation function runtime. So, if the **Frame display time** field is 0, the actual time the frame is displayed on the screen will be the least possible in this case.

Animation start time and **Animation stop time** fields set up the time interval for the animation. The **data source folder** field specifies the folder with data-files to be animated. The same folder is used for animations and for air gap distribution and cross-section distribution plots. Using animation is only possible if the files containing data for the time interval to be animated were previously saved.

To start the animation, click the **Start animation** button. The current time point, animated quantity and other information is displayed at the top of each window involved in the animation. The animation continues until the time specified in the **Animation stop time** field is reached or until all windows involved in the animation are closed. You can also pause the animation using the **Pause** button. While the animation is in progress or paused, you can use all MATLAB figure tools, for example, changing the scale and position of the plots.

9. DYNAMIC FE ANALYSIS SIMULATION SCRIPT FUNCTIONS

A simulation script is a file with .m extension containing MATLAB-function of a specific format. While the dynamic FEA simulation is running this function is called on each simulation time step and allows to automatically change all simulation settings, compute, and store your own variables, control power supply sources and electronic switches, implement motor control algorithms and etc. For example, the simulation script function is used to control electronic switches (IGBT, etc.) to implement PWM switching sequence. To understand this chapter, some familiarity with MATLAB programming is required. Refer to MATLAB help documentation for more details on MATLAB programming.

9.1. General information.

MotorXP-PM offers you a variety of options to properly set up your simulation and in most cases, there is no need to use simulation scripts. On the other hand, simulation scripts give you much more flexibility in using MotorXP-PM. Note that simulation scripts are used only for **Dynamic FE Analysis** and are available only in MotorXP-PM MATLAB version. Refer to section 3.2 for more details on limitations of MotorXP-PM Standalone version.

The chosen simulation script file is displayed in the **Simulation script file** field of the MotorXP-PM main window when **Dynamic FE Analysis** is chosen (see Figure 8.1). There are three simulation scripts used by default: *simscript_hystpwm.m*, *simscript_spacevecpwm.m*, *simscript_sixstep.m* which can be found in *[MotorXP-PM installation directory]\SimScripts*. If the **Simulation settings** field is set to **General** (see Figure 8.1), the corresponding simulation script is chosen automatically depending on the drive type. If the **Simulation settings** field is set to **Advanced**, the simulation script should be specified by the user. Click the button to the right of the **Simulation script file** field to choose your own simulation script.

If the simulation script file is used all simulation settings displayed in the MotorXP-PM main window can be overwritten with those specified in the simulation script function. For example, you can specify any RMS supply current or advance angle values regardless of those specified in the **Target RMS supply current** and **Target Advance angle** fields of the MotorXP-PM main window.

9.2. Writing a simulation script function.

The definition of the simulation script function `simscript` and a minimum amount of code appearing in file *simscript.m* is as follows:

```
function [ShaftPosition,CircuitControl,Settings,Enforce,Userdata] ...
= simscript(Outputs,Settings,Userdata,Circuit, Drive,Geometry,Mesh,...
            Windings,A,cell_p,cell_t,cell_Nu,path)
ShaftPosition = [];
CircuitControl = [];
Enforce = [];
```

The first line of the function always starts with the keyword `function`. The names of the M-file and of the function should be the same. Refer to MATLAB help documentation for more details on writing MATLAB-functions.

Be aware that the presented example of the simulation script function will not work with stator electrical circuit file *InverterCircuit.m* since states of switches are not defined.

The function's output arguments:

`ShaftPosition` – rotor shaft position; 1×2 array, containing rotor shaft coordinates $[x; y]$ in meters. This variable allows you to apply static or dynamic eccentricity to the rotor. If array `ShaftPosition` = [] or `ShaftPosition` = [0; 0], there is no rotor eccentricity.

`CircuitControl` – structure containing current and voltage values which are supplied by current and voltage courses as well as states of electronic switches. See chapter 10 for more details on using current and voltage sources and electronic switches.

`Settings` – structure of simulation settings, the same as input argument `Settings`.

`Enforce` – structure to control rotor speed. If `Enforce`=[], when rotor speed is computed depending on the load and electromagnetic torque (variable speed simulation). To set the rotor speed to some fixed value use the statement `Enforce.speed=fixedspeed`, where `fixedspeed` is the rotor speed in rad/s.

`Userdata` – structure to store user data, the same as input argument `Userdata`. By default, `Userdata` is empty. Refer to section 9.4 to figure out how to use the `Userdata` structure to store your own data using simulation script functions.

The function's input arguments:

`Outputs` – structure containing the dynamic FEA simulation results.

`Settings` – structure containing the dynamic FEA simulation settings.

`Userdata` – structure to store user data.

`Circuit` – structure containing stator electrical circuit parameters.

`Drive` – structure containing drive parameters.

`Geometry` – structure containing the machine's dimensions data.

`Mesh` – structure containing mesh data.

`Windings` – structure containing the machine's windings data.

`A` – array of magnetic vector potential node values for the current time step; `A[1:np]` – first slice values, `A[np+1:2*np]` – second slice values (if several slices are used; see section 2.4), `A[2*np+1:3*np]` – third slice values and etc., where `np` – number of mesh nodes in one slice.

`cell_p` – cell-array containing x and y coordinates of finite element mesh nodes for every slice, as it is described in MATLAB PDE Toolbox help documentation (see help on `initmesh` function for more details).

`cell_t` – cell-array containing finite elements data for every slice, as it is described in MATLAB PDE Toolbox help documentation (see help on `initmesh` function for more details).

`cell_Nu` – cell-array containing midpoint magnetic reluctivity values for every slice. Magnetic reluctivity is $\nu = \frac{1}{\mu_0 \mu_r}$, where μ_0 – permeability of the free space, μ_r - relative permeability.

`path` – directory with application files.

9.3. Main data structures.

Outputs structure fields:

Field	Size, type	Units	Description
<code>Outputs.CurrentTime</code>	1×1, double	second	Simulation current time point.
<code>Outputs.nPolePairs</code>	1×1, double		Number of pole pairs.
<code>Outputs.gamma0</code>	1×1, double	Electrical degree	Angle between a-phase axis and rotor q-axis for initial rotor position.
<code>Outputs.Va</code>	1×n, double	Volt	Instantaneous values of voltages measured at stator winding terminals; n – number of computed time steps.
<code>Outputs.Vb</code>			
<code>Outputs.Vc</code>			
<code>Outputs.Vd</code>	1×n, double	Volt	Instantaneous values of d-axis and q-axis voltage components.
<code>Outputs.Vq</code>			
<code>Outputs.Ia</code>	Npp×n, double	Ampere	Current flowing in each parallel path of each phase; number of array rows corresponds to number of parallel paths. Npp – number of parallel paths.
<code>Outputs.Ib</code>			
<code>Outputs.Ic</code>			
<code>Outputs.Id</code>	1×n, double	Ampere	Instantaneous values of d-axis and q-axis current components.
<code>Outputs.Iq</code>			
<code>Outputs.time</code>	1×n, double	second	Time points.
<code>Outputs.Gamma</code>	1×n, double	Electrical degree	Instantaneous values of advance angle.
<code>Outputs.BackEMFa</code>	1×n, double	Volt	Instantaneous values of phase back-EMF.
<code>Outputs.BackEMFb</code>			
<code>Outputs.BackEMFc</code>			
<code>Outputs.BackEMFd</code>	1×n, double	Volt	Instantaneous values of d-axis and q-axis back-EMF components.
<code>Outputs.BackEMFq</code>			
<code>Outputs.Psi</code>	nCircuitNodes×n, double	Volt	Stator electrical circuit potentials. nCircuitNodes – total number of nodes of stator electrical circuits.
<code>Outputs.Torque</code>	1×n, double	N*m	Electromagnetic torque; equals to one of the variables <code>Torque_maxwell</code> or <code>Torque_vwork</code> depending on the selected torque calculation method.

Outputs.Load	1×n, double	N*m	Load torque on the motor shaft.
Outputs.Speed	1×n, double	rad/s	Rotor angular speed.
Outputs.Rotang	1×n, double	rad	Rotor angular position.
Outputs.Pinput	1×n, double	Watt	Input apparent power delivered by all current and voltage sources.
Outputs.Pcons	1×n, double	Watt	Consumed apparent power.
Outputs.Ps	1×n, double	Watt	Apparent power (real and reactive) consumed by the stator electrical circuit.
Outputs.Pmf	1×n, double	Watt	Magnetic energy time derivative.
Outputs.Pmech	1×n, double	Watt	Mechanical power on the shaft.
Outputs.Piron.stator	1×1, double	Watt	Stator iron loss.
Outputs.Piron.rotor	1×1, double	Watt	Rotor iron loss.
Outputs.Torque_maxwell	1×n, double	N*m	Electromagnetic torque calculated with the Maxwell stress tensor method.
Outputs.Torque_vwork	1×n, double	N*m	Electromagnetic torque calculated with the Virtual work method.
Outputs.Torque_fluxlinkage	1×n, double	N*m	Electromagnetic torque calculated by currents and flux linkages using Exp. 2.10.
Outputs.Torque_magnet	1×n, double	N*m	Magnet component of electromagnetic torque.
Outputs.Torque_reluctance	1×n, double	N*m	Reluctance component of electromagnetic torque.
Outputs.Fluxlinkage_a	1×n, double	Weber	Instantaneous values of flux linkages.
Outputs.Fluxlinkage_b			
Outputs.Fluxlinkage_c			
Outputs.Fluxlinkage_d	1×n, double	Weber	Instantaneous values of d-axis and q-axis flux linkage components.
Outputs.Fluxlinkage_q			
Outputs.Fx	1×n, double	N	Radial electromagnetic force between stator and rotor along x -direction.
Outputs.Fy	1×n, double	N	Radial electromagnetic force between stator and rotor along y -direction.

Settings structure fields:

Field	Value	Description
Settings.DF_script	Directory and/or file name	Corresponds to the Simulation script file field of the main window.
Settings.DF_solvertype	'Nonlinear'	Corresponds to the Solver type field of the main window.
	'Linear'	
Settings.DF_tol	Number > 0	Corresponds to the Convergence tolerance field of the main window.
Settings.DF_statorcircuit	Function name	Name of the function specifying the stator electrical circuit; corresponds to the Stator electrical circuit file field of the main window.

Settings.DF_timestep	Number > 0	Simulation time step; corresponds to the Time step field of the main window.
Settings.DF_stoptime	Number > 0	Corresponds to the Simulation stop time field of the main window.
Settings.DF_Isrms	Number >= 0	RMS supply current to be supplied by inverter; corresponds to the Target RMS supply current field of the main window.
Settings.DF_Gamma	Number >= 0	Corresponds to the Target advance angle field of the main window.
Settings.DF_motorload	Number >= 0	Load torque on the motor shaft, corresponds to the Load field of the main window.
Settings.DF_InitSpeed	Number	Initial speed of the rotor; corresponds to the Initial Speed field of the main window when variable speed simulation is chosen.
Settings.DF_saveeachsolution	0	Corresponds to the Save each filed solution checkbox value of the main window.
	1	
Settings.DF_saveeachsolutionfolder	Directory	Folder to store data-files when Save each filed solution is checked.
Settings.DF_torquemethod	'Maxwell stress tensor'	Electromagnetic torque calculation method; corresponds to the Torque calculation method field of the main window.
	'Virtual work'	
Settings.DF_force	0	Enables calculation of the radial force acting between stator and rotor; corresponds to the Compute rotor radial force field of the main window.
	1	

Geometry structure fields:

Field	Value	Description
Geometry.lag	Number > 0	Air gap length; corresponds to the Air gap field of Geometry Editor .
Geometry.motortype	'InnerRotor'	Corresponds to the Machine type field of Geometry Editor .
	'OuterRotor'	
Geometry.slotlayertype	'Single layer'	Determined either single layer or double layer winding is used; corresponds to the Winding layers field of Geometry Editor .
	'Double layer'	
Geometry.Ns	Number > 0	Corresponds to the Number of slots field of Geometry Editor .
Geometry.layerpos	Upper/Lower'	Determined either winding layers are oriented horizontally or vertically inside the slot; corresponds to the Layers orientation field of Geometry Editor .
	'Left/Right'	
Geometry.l	Number > 0	Corresponds to the Lamination stack length field of Geometry Editor .
Geometry.rotorskew	Number >= 0	Corresponds to the Stator skew angle field of Geometry Editor .
Geometry.statorskew	Number >= 0	Corresponds to the Rotor skew angle field of Geometry Editor .

Windings structure fields:

Field	Value	Description
Windings.Npp	Number > 0	Number of parallel paths of the stator winding per phase; corresponds to the Number of parallel paths field of Winding Editor .
Windings.Lsew	Number > = 0	Leakage inductance of the stator winding end-turns per phase; corresponds to the End winding inductance field of Winding Editor .
Windings.Rs	Number > = 0	Active DC resistance of the stator winding per phase depending on the winding temperature; corresponds to the Phase resistance field of Winding Editor .
Windings.layout1	Arrays	Stator winding layout for each phase; corresponds to the Winding layout tables of Winding Editor .
Windings.layout2		
Windings.layout3		
Windings.W	Number > 0	Total number of conductors in one stator slot (for double layer winding – number of conductors in both layers of the slot).
Windings.nstrands	Number > 0	Corresponds to the Number of strands in hand field of Winding Editor .
Windings.fillfactor	$0 < \text{Number} \leq 1$	Stator coil fill factor; corresponds to the Coil fill factor field of Winding Editor .
Windings.ks	Number > 0	Stator winding material conductivity depending on the winding temperature.
Windings.statorcircuit	'StarConnection'	Determines the stator winding connection; corresponds to the Winding connection field of Winding Editor .
	'DeltaConnection'	
Windings.Hsew	Number > 0	Corresponds to the End winding axial overhang field of Winding Editor .
Windings.layoutmethod	'Automatic'	Stator winding layout input method; corresponds to the Layout method field of Winding Editor .
	'Manual'	
	'From file'	
Windings.nPolePairs	Number > 0	Corresponds to the Number of pole pairs field of Winding Editor .

Mesh structure fields:

Field	Value	Description
Mesh.nAirGapLayers	3, 5, 7, 9	Number of mesh layers in the air gap; corresponds to the Number of layers in air gap field of Mesh Editor .
Mesh.nSlices	Integer number ≥ 1	Number of machine's cross-sections used by multi-slice FEM; corresponds to the Number of slices field of Mesh Editor .
Mesh.perbndcnd	'None'	Periodic/antiperiodic boundary conditions; corresponds to the Boundary conditions field of Mesh Editor .
	'Periodic'	
	'Antiperiodic'	
Mesh.p	2 \times np, double	Initial mesh data.
Mesh.t	4 \times nt, double	
Mesh.e	7 \times ne, double	
Mesh.b	1 \times np, double	Array of boundary conditions.
Mesh.Rotate	Structure of double arrays	Variable for internal use.
Mesh.nper	Integer number ≥ 1	Number of periodicities for periodic/antiperiodic boundary conditions; if equals to 1 – no periodic/antiperiodic boundary conditions applied.
Mesh.periodicity	1	Equals to 1 for periodic boundary conditions; equals to -1 for antiperiodic boundary conditions.
	-1	

Drive structure fields:

Field	Value	Description
Drive.Vdc	Number ≥ 0	Corresponds to the DC supply voltage (Vdc) field of Drive Settings .
Drive.DriveType	'Current hysteresis PWM'	Corresponds to the Drive type field of Drive Settings .
	'Space vector PWM'	
	'Six-step'	
Drive.Is_hyst_perc	Number > 0	Current hysteresis in percentage of RMS current; corresponds to the Current hysteresis field of Drive Settings .
Drive.fspwm	Number > 0	Corresponds to the PWM sampling frequency field of Drive Settings .
Drive.SwitchDutyCycle_sixstep	'120'	Corresponds to the Switch duty cycle field of Drive Settings .
	'180'	

Drive. CommutationAdvanceAngle _sixstep	Number > 0	Corresponds to the Commutation advance angle field of Drive Settings .
Drive. SixStepOptions	'General'	Corresponds to the Options field of Drive Settings .
	'Six-step with limited maximum current'	
	'Six-step with variable DC voltage'	
Drive. Is_max_sixstep	Number > 0	Corresponds to the Motor phase current limit field of Drive Settings .
Drive. Is_hyst_perc_sixstep	Number > 0	Phase current hysteresis in percentage of current limit; corresponds to the Phase current hysteresis field of Drive Settings .
Drive. Vdc_perc_sixstep	$0 \leq \text{Number} \leq 100$	DC voltage usage percentage in % of Vdc; corresponds to the Vdc usage percentage field of Drive Settings .
Drive.Transistor	String	Corresponds to the Transistor field of Drive Settings .
Drive.TransistorType	'MOSFET'	Corresponds to the Transistor type field of Drive Settings .
	'IGBT'	
Drive.nTransistors	Integer number ≥ 1	Corresponds to the Transistors in parallel field of Drive Settings .
Drive.Rg	Number > 0	Corresponds to the MOSFET driver resistor field of Drive Settings .
Drive.deadtime	Number ≥ 0	Corresponds to the Switching dead time field of Drive Settings .

9.4. Simple example of simulation script function.

This example shows how to write a simple simulation script function which changes simulation time step and stores some user defined variables in the `Userdata` structure. The source code of the function presented below can be found in *[MotorXP-PM installation directory]\SimScripts\simscript_example.m*.

```
function [ShaftPosition, CircuitControl, Settings, Enforce, Userdata] = ...
simscript_example(Outputs, Settings, Userdata, Circuit, Drive, Geometry, Mesh, ...
    Windings, A, cell_p, cell_t, cell_Nu, path)
% Example of simulation script function (should be used with circuit function
InvertedCircuit.m)

% call default simulation script function for space vector PWM
[ShaftPosition, CircuitControl, Settings, Enforce, Userdata] = ...
simscript_spacevecpwm(Outputs, Settings, Userdata, Circuit, Drive, Geometry, Mesh, ...
    Windings, A, cell_p, cell_t, cell_Nu, path);

% Current simulation time
if isempty(Outputs.time)
```

```

    CurrentTime = 0;
else
    CurrentTime = Outputs.time(end);
end
% Change time step if simulation time reaches 0.1 sec
if CurrentTime>0.1
    Settings.DF_timestep=10^-6;
else
    Settings.DF_timestep=10^-5;
end

```

At the beginning of the code the default simulation script function for the space vector PWM `simscript_spacevecpwm` is called so there is no need to worry about assigning `ShaftPosition`, `CircuitControl` and `Enforce` structures since they are assigned by `simscript_spacevecpwm`.

In this example the simulation time step is changed when the simulation reaches 0.1 sec. You can define your own condition to change any simulation parameter in the `Settings` structure.

To store user defined variables (switching functions for this example) the following piece of code is used:

```

% Save switching function for each phase in Userdata structure
if ~isfield(Userdata, 'Switch_a')
    % Create fields 'Switch_a', 'Switch_b', 'Switch_c' in Userdata when
    % simscript_example is called for the first time
    Userdata.Switch_a=[];
    Userdata.Switch_b=[];
    Userdata.Switch_c=[];
else
    % CircuitControl.switch_x1 - upper switch state
    if CircuitControl.switch_a1==1
        sA=1;
    else
        sA=-1;
    end
    if CircuitControl.switch_b1==1
        sB=1;
    else
        sB=-1;
    end
    if CircuitControl.switch_c1==1
        sC=1;
    else
        sC=-1;
    end
    % Collect switching function values for each time step
    Userdata.Switch_a=[Userdata.Switch_a sA];
    Userdata.Switch_b=[Userdata.Switch_b sB];
    Userdata.Switch_c=[Userdata.Switch_c sC];
end

```

When the function is called for the first time, fields `Switch_a`, `Switch_b` and `Switch_c` are created in the `Userdata` structure. The switching function values are stored in the corresponding fields of the `Userdata` structure on each subsequent call of the function. Variable saved in the `Userdata` structure can be plotted as a function of time using the **Time plot** panel of **Plot Wizard** as discussed in section

8.2.3. Be aware that in order to plot the variable, the length of the variable should be the same as number of time steps, i.e. the variable should be an array and its elements should be saved on each time step as shown in the example above.

Fields of the `CircuitControl` structure corresponding to states of electronic switches used in default simulation scripts are discussed in section 10.3.

10. USING ELECTRICAL CIRCUITS

MotorXP-PM allows you to connect the stator winding to the electrical circuit consisting of the following components:

- resistors;
- capacitors;
- inductances;
- ideal diodes;
- electronic switches;
- voltage sources;
- current sources.

Electrical circuits are generated programmatically using a library of special functions. User defined electrical circuits can be used only for **Dynamic FE Analysis** and are available only in MotorXP-PM MATLAB version. Refer to section 3.2 for more details on limitations of MotorXP-PM Standalone version.

The electrical circuit is specified through the function which retrieves the electrical circuit object. The name of the function appears in the **Stator electrical circuit file** pop-up menu of the main window when **Dynamic FE Analysis** is chosen. Clicking the button to the right allows you to specify your own electrical circuit choosing corresponding M-file from the list of files. Function used for the default electrical circuits of the three-phase inverter (shown in Figure 10.1) is implemented in *[MotorXP-PM installation directory]\Circuits\InverterCircuit.m*. You can use this file as an example to write your own electrical circuit functions.

To understand this chapter some familiarity with MATLAB programming is required. Refer to MATLAB help documentation for more details on MATLAB programming.

10.1. Writing an electrical circuit function.

The definition of the electrical circuit function `InverterCircuit` appears in file *InverterCircuit.m* as follows:

```
function Schematic=InverterCircuit(p,t,Subdomains,Circuit,l,nper,~,~)
```

The function retrieves one output argument `Schematic` which contains the electrical circuit description.

The input arguments of the function:

`p` – variable taken from `Mesh.p`.

`t` – variable taken from `Mesh.t`,

Refer to section 9.3 with the `Mesh` structure description for more details on `Mesh.p` and `Mesh.t`.

`Subdomains` – array of subdomains.

`Circuit` – structure containing stator electrical circuit parameters.

`l` – lamination length; this variable is taken from the **Lamination stack length** field of **Geometry Editor**.

`nper` – number of periodicities to define periodic/antiperiodic boundary conditions.

The following fields of the `Circuit` structure can be used:

`Circuit.Npp` – number of parallel paths of the stator winding per phase; this variable is taken from the **Number of parallel paths** field of **Winding Editor**.

`Circuit.Rs` – active resistance of the stator winding per phase for the specified stator winding temperature; this variable is taken from the **Phase resistance** field of **Winding Editor**.

`Circuit.Lsew` – leakage inductance of the stator winding end-turns per phase; this variable is taken from the **End winding inductance** field of **Winding Editor**.

`Circuit.layout` – stator winding layout.

Circuit construction procedure always begins with the following function:

```
Schematic = CircuitCreateSchematic();
```

This function does not have input arguments and retrieves the empty circuit object `Schematic`.

The procedure of circuit construction consists of building circuit branches and adding branches to the circuit object.

10.1.1. Electrical circuit branches.

The following function creates an electrical circuit branch:

```
Branch = CircuitCreateBranch(node1,node2);
```

All circuit nodes should be previously numbered beginning with zero node. The function receives start and end node numbers of the circuit branch, respectively, and retrieves an empty branch object `Branch`.

When all circuit components are added to the branch object (see section 10.1.2), the branch should be added to the circuit object using the following function:

```
Schematic = CircuitAddBranch(Schematic,Branch);
```

The function receives the circuit and branch objects and retrieves the circuit objects with the new branch added to the circuit.

10.1.2. Adding circuit components.

10.1.2.1. To add a resistor to the branch, the following function is used:

```
Branch = CircuitAddR(Branch,name,value);
```

Branch – variable corresponding to the branch which the resistor is added to, name – unique circuit component name, value – resistance value in Ohm.

Example of adding resistor R1 of 1 kOhm to the branch defined by variable Branch:

```
Branch = CircuitAddR(Branch, 'R1', 1000);
```

10.1.2.2. To add an inductance to the branch, the following function is used:

```
Branch = CircuitAddL(Branch, name, value);
```

Branch – variable corresponding to the branch which the inductance is added to, name – unique circuit component name, value – inductance value in Henries.

Example of adding inductance L1 of 100 mH to the branch defined by variable Branch:

```
Branch = CircuitAddL(Branch, 'L1', 0.1);
```

10.1.2.3. To add a capacitor to the branch, one of the following sentences can be used:

```
Branch = CircuitAddC(Branch, name, value);
```

```
Branch = CircuitAddC(Branch, name, value, Vinit)
```

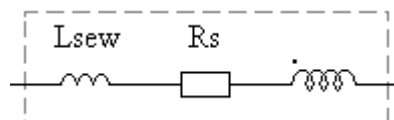
Branch – variable corresponding to the branch which the capacitor is added to, name – unique circuit component name, value – capacitance value in Farads, Vinit – initial voltage on the capacitor. In the first case the initial voltage is zero.

Example of adding capacitor C1 of 100 μF to the branch defined by variable Branch:

```
Branch = CircuitAddC(Branch, 'C1', 100*10^-6);
```

Since in this case the input variable Vinit is not used the initial voltage is zero.

10.1.2.4. Each stator winding coil or a group of coils can be treated as an independent circuit component consisting of an active resistance, stator end winding inductance and conductors within stator slots designated as a spiral with a dot at the input terminal:



A coil or a group of coils is herein referred to as a coil object or a coil component. Each coil object combines all stator slots or stator slot layers associated with the same phase and the same parallel path.

To add a coil object to the branch, the following function is used:

```
Branch = CircuitAddCoil(Branch, name, phase, Subdomains, R, Lsew, ...
                        npath, Npp, coilorientation, p, t, l, nper);
```

Branch – variable corresponding to the branch which the coil object is added to, name – unique circuit component name, phase – phase accepts one of the following values: 'a', 'b', 'c'; Subdomains – array of subdomains; R and Lsew – active resistance and end winding inductance of the coil component, respectively, npath – parallel path number (npath=1 if there are no parallel paths), Npp – number of parallel paths of the stator winding per phase; coilorientation = 1 if the coil component is oriented concordantly with the branch which the coil component is added to, otherwise coilorientation = -1. The branch orientation is defined by node1 and node2 arguments of CircuitAddBranch function and the coil component orientation is defined by a dot at the input terminal. Input arguments p, t, l, nper are the same as described in section 10.1.

Note that in order to prevent incorrect results **each branch should include only one coil component**, otherwise coil components must be divided by additional nodes. For example, if you have two coils connected in series you should put additional note between coils, so each coil has its own branch object.

Example of adding the coil component to the branch defined by variable Branch:

```
Branch = CircuitAddCoil(Branch, 'coil_c2', 'c', Subdomains, ...
                        Rs, Lsew, 2, Npp, 1, p, t, l, nper);
```

In this case the coil component corresponds to the second group (parallel path 2) of coils of phase “c” named as coil_c2, oriented concordantly with the branch, with the active resistance Rs and end winding inductance Lsew.

10.1.2.5. To add a diode to the branch, the following function is used:

```
Branch = CircuitAddDiode(Branch, name, Roff, Ron, direction);
```

Branch – variable corresponding to the branch which the diode is added to, name – unique circuit component name, Roff and Ron – backward and forward resistance of the diode in Ohm; direction = 1 if the diode forward direction is oriented from node1 to node2 of the branch (see description of CircuitCreateBranch in section 10.1.1), otherwise direction = -1.

Note that backward and forward resistances should be as follows:

$$0 < R_{off} < \infty$$

$$0 < R_{on} < \infty$$

Example of adding diode D1 to the branch defined by variable Branch:

```
Branch = CircuitAddDiode(Branch, 'D1', 10^8, 10^-6, 1);
```

10.1.2.6. To add a voltage source to the branch, the following function is used:

```
Branch = CircuitAddE(Branch, name, data);
```

Branch – variable corresponding to the branch which the voltage source is added to, name – unique circuit component name, data – field of the `CircuitControl` structure used by a simulation script function to control the voltage source (see section 10.2).

10.1.2.7. To add a current source to the branch, the following function is used:

```
Branch = CircuitAddJ(Branch, name, data);
```

Branch – variable corresponding to the branch which the current source is added to, name – unique circuit component name, data – field of the `CircuitControl` structure used by a simulation script function to control the current source (see section 10.2).

10.1.2.8. To add a switch to the branch, the following function is used:

```
Branch = CircuitAddSwitch(Branch, name, data, Roff, Ron);
```

Branch – variable corresponding to the branch which the switch is added to, name – unique circuit component name, data – field of the `CircuitControl` structure used by a simulation script function to control the switch state (see section 10.2), Roff and Ron – ‘off’ and ‘on’ resistance of the switch in Ohm.

Note that ‘off’ and ‘on’ resistances should be as follows:

$$0 < R_{off} < \infty$$

$$0 < R_{on} < \infty$$

10.2. Controlling power sources and electronic switches using simulation script functions.

While the dynamic FE simulation is running, the voltage and current values supplied by the voltage and current sources as well as the state of each electronic switch can be specified at each time step by the simulation script function. For this purpose, the `CircuitControl` structure, the output argument of the simulation script function, is used. Each voltage or current source and switch is associated with its field of the `CircuitControl` structure through the input argument data, when the `CircuitAddE`, `CircuitAddJ` or `CircuitAddSwitch` function is called. For the power sources, the value assigned to the corresponding field of the `CircuitControl` structure at a specific time step is the voltage or

current output value of the voltage or current source associated with this field. Similarly, the state of the switch is determined by the value assigned to the corresponding field of the `CircuitControl` structure at a specific time step: 1 for state 'on', 0 – for state 'off' (see section 10.3 for more details on using electronic switches).

The following example provides an explanation for the power source control principle. Assume that the voltage source named V1 was added to the stator circuit calling the function

```
Branch = CircuitAddE(Branch, 'V1', 'CircuitControl.v1');
```

In this case the voltage source V1 is associated with the field `v1` of the `CircuitControl` structure. The following piece of code can be used in the simulation script function to specify the voltage value supplied by the voltage source V1 at a given simulation time step:

```
CurrentTime = Outputs.CurrentTime;
CircuitControl.v1 = 220*sqrt(2)*sin(2*pi*50*CurrentTime);
```

In this case we have the source of sinusoidal voltage with frequency 50Hz and RMS value 220V. Note that power sources can have any desired voltage or current waveform.

Similarly, for the switch S1 created by calling the following function:

```
Branch = CircuitAddSwitch(Branch, 'S1', 'CircuitControl.s1', Roff, Ron);
```

Writing simulation script function, you can use the following command to set the switch S1 to 'on' state:

```
CircuitControl.s1 = 1;
```

The following command will set the switch S1 to 'off' state:

```
CircuitControl.s1 = 0;
```

10.3. Default three-phase inverter circuit function.

As mentioned before the three-phase inverter circuit function `InverterCircuit` is used by default for **Dynamic FE Analysis**. The source code of the function can be found in *[MotorXP-PM installation directory]\Circuits\InverterCircuit.m*. An example of the electrical circuit with the star-connected stator winding and two parallel paths is shown in Figure 10.1. The electrical circuit is modified automatically depending on the winding connection and number of parallel paths. The following piece of code constructs the inverter circuit:

```
Schematic = CircuitCreateSchematic();
Branch = CircuitCreateBranch(0,1);
Branch = CircuitAddE(Branch, 'Vdc', 'CircuitControl.vdc');
Schematic = CircuitAddBranch(Schematic, Branch);
```

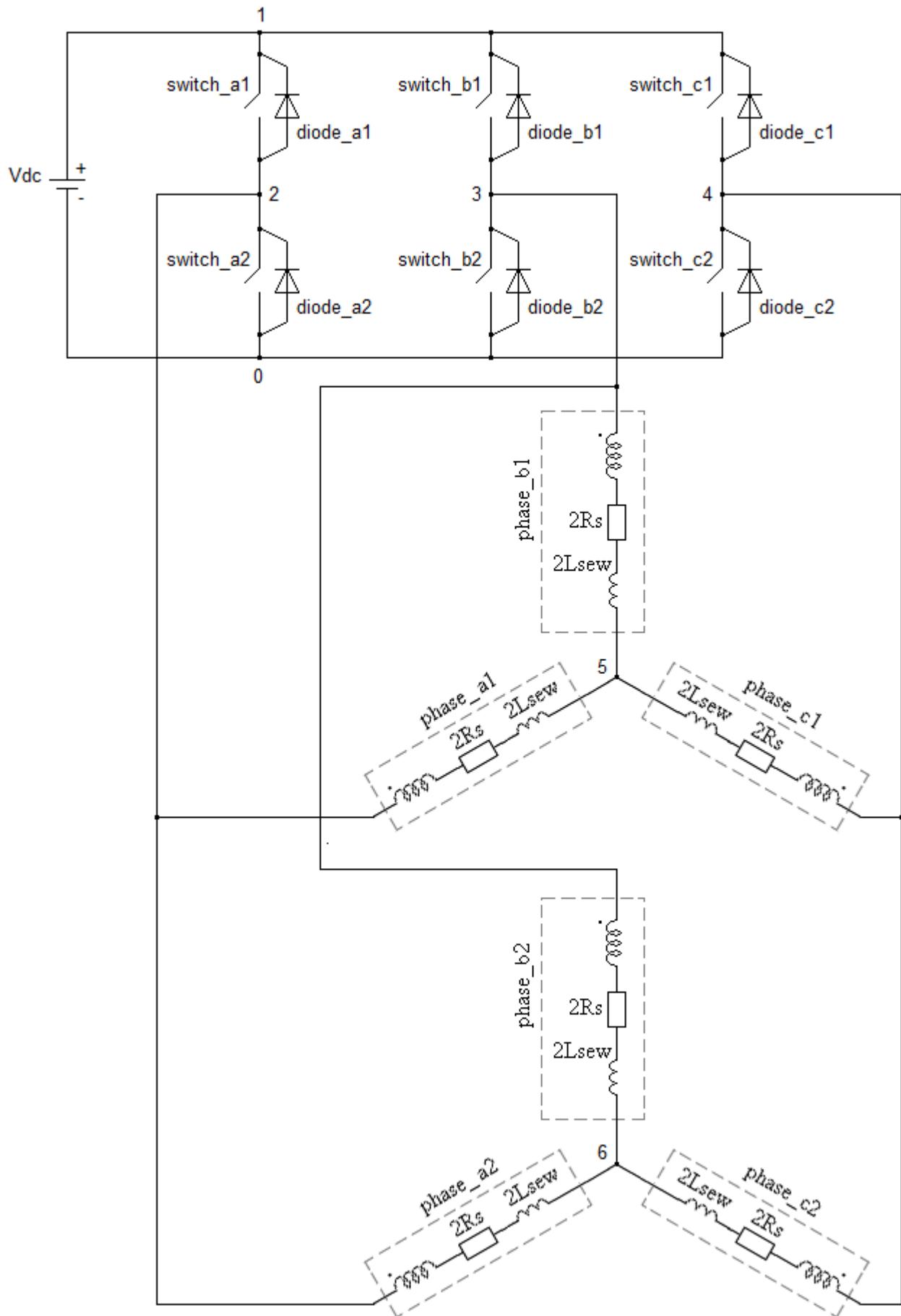


Figure 10.1. Three-phase inverter electrical circuit with star-connected winding and two parallel paths.

```

Branch = CircuitCreateBranch(2,1);
Branch = CircuitAddSwitch(Branch, 'switch_a1', 'CircuitControl.switch_a1', Roff, Ron);
Schematic = CircuitAddBranch(Schematic, Branch);
Branch = CircuitCreateBranch(2,1);
Branch = CircuitAddDiode(Branch, 'diode_a1', Roff, Ron, 1);
Schematic = CircuitAddBranch(Schematic, Branch);

Branch = CircuitCreateBranch(3,1);
Branch = CircuitAddSwitch(Branch, 'switch_b1', 'CircuitControl.switch_b1', Roff, Ron);
Schematic = CircuitAddBranch(Schematic, Branch);
Branch = CircuitCreateBranch(3,1);
Branch = CircuitAddDiode(Branch, 'diode_b1', Roff, Ron, 1);
Schematic = CircuitAddBranch(Schematic, Branch);

Branch = CircuitCreateBranch(4,1);
Branch = CircuitAddSwitch(Branch, 'switch_c1', 'CircuitControl.switch_c1', Roff, Ron);
Schematic = CircuitAddBranch(Schematic, Branch);
Branch = CircuitCreateBranch(4,1);
Branch = CircuitAddDiode(Branch, 'diode_c1', Roff, Ron, 1);
Schematic = CircuitAddBranch(Schematic, Branch);

Branch = CircuitCreateBranch(0,2);
Branch = CircuitAddSwitch(Branch, 'switch_a2', 'CircuitControl.switch_a2', Roff, Ron);
Schematic = CircuitAddBranch(Schematic, Branch);
Branch = CircuitCreateBranch(0,2);
Branch = CircuitAddDiode(Branch, 'diode_a2', Roff, Ron, 1);
Schematic = CircuitAddBranch(Schematic, Branch);

Branch = CircuitCreateBranch(0,3);
Branch = CircuitAddSwitch(Branch, 'switch_b2', 'CircuitControl.switch_b2', Roff, Ron);
Schematic = CircuitAddBranch(Schematic, Branch);
Branch = CircuitCreateBranch(0,3);
Branch = CircuitAddDiode(Branch, 'diode_b2', Roff, Ron, 1);
Schematic = CircuitAddBranch(Schematic, Branch);

Branch = CircuitCreateBranch(0,4);
Branch = CircuitAddSwitch(Branch, 'switch_c2', 'CircuitControl.switch_c2', Roff, Ron);
Schematic = CircuitAddBranch(Schematic, Branch);
Branch = CircuitCreateBranch(0,4);
Branch = CircuitAddDiode(Branch, 'diode_c2', Roff, Ron, 1);
Schematic = CircuitAddBranch(Schematic, Branch);

```

Some part of the code connecting stator winding to the inverter is not shown.

As described in the previous section the state of each switch is controlled by the corresponding field of the `CircuitControl` structure. There are three default simulation script functions used together with the `InverterCircuit` function: `simscript_hystpwm`, `simscript_spacevecpwm` and `simscript_sixstep`. The following piece of code is used to control switch states in the `simscript_spacevecpwm` function:

```

[sA,sB,sC] = spacevecpwm(Vai_ref,Vbi_ref,Vci_ref,fspwm,CurrentTime,Vdc);
if sA==1
    CircuitControl.switch_a1=1;
    CircuitControl.switch_a2=0;
else % sA== -1
    CircuitControl.switch_a1=0;
    CircuitControl.switch_a2=1;
end

```

```

if sB==1
    CircuitControl.switch_b1=1;
    CircuitControl.switch_b2=0;
else % sB== -1
    CircuitControl.switch_b1=0;
    CircuitControl.switch_b2=1;
end
if sC==1
    CircuitControl.switch_c1=1;
    CircuitControl.switch_c2=0;
else % sC== -1
    CircuitControl.switch_c1=0;
    CircuitControl.switch_c2=1;
end

```

Use the source code of *InverterCircuit.m*, *simscrip_hystpwm.m*, *simscrip_spacevecpwm.m* and *simscrip_sixstep.m* as an example to write your own electrical circuit functions and simulation scripts.

11. WRITING GEOMETRY SCRIPTS

11.1. General information and geometry script structure.

In MotorXP Design Studio the geometry of both stator and rotor is determined by the geometry template, which is defined by the corresponding geometry script. Geometry scripts allow to create fully parameterized geometry templates for stator and rotor. It is more convenient to use a standalone version of MotorXP Design Studio (see section 3.3) when working with geometry scripts. Each geometry script is a JavaScript file with .js extension so some familiarity with the JavaScript language is required. Geometry script includes UI widgets definition, geometry drawing algorithms and material type assignments. Every geometry script can be modified, or new geometry script can be created to define new stator or rotor geometry template.

Geometry scripts use C++ Qt objects; the corresponding information can be found on the official Qt documentation website <https://doc.qt.io/>. The name of Qt classes usually starts with 'Q' letter. The general structure of geometry script is demonstrated on the simple stator template example as shown in Figure 11.1. Full source code of the script can be found in *[MotorXP-PM installation directory]\bin\assets\scripts\Stator\stator_script_example.js* (use right click context menu to open folder with the script as shown in Figure 11.2). Figure 11.1 shows the stator geometry generated by the script. Only one right half of slot pitch (half of pole pitch for rotor) is generated by the script, the rest of the geometry is automatically drawn by **Geometry Editor**. The corresponding user interface in the stator editor mode is shown in Figure 11.2. Only slot depth and slot width are available for the user to modify from the user interface for this script example.

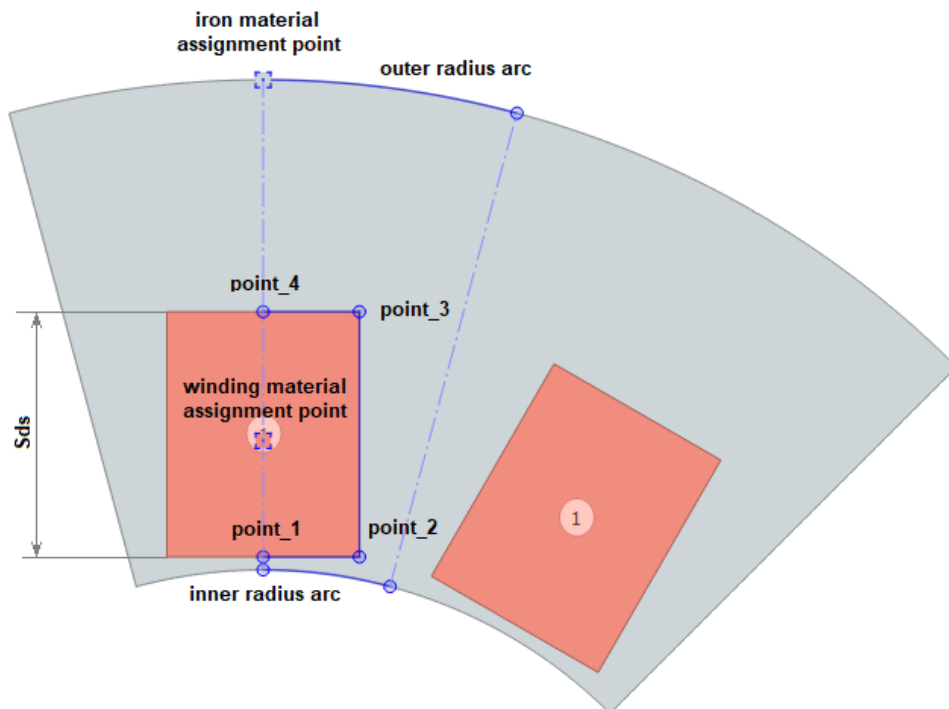


Figure.11.1. Stator geometry generated by the *stator_script_example.js* script.

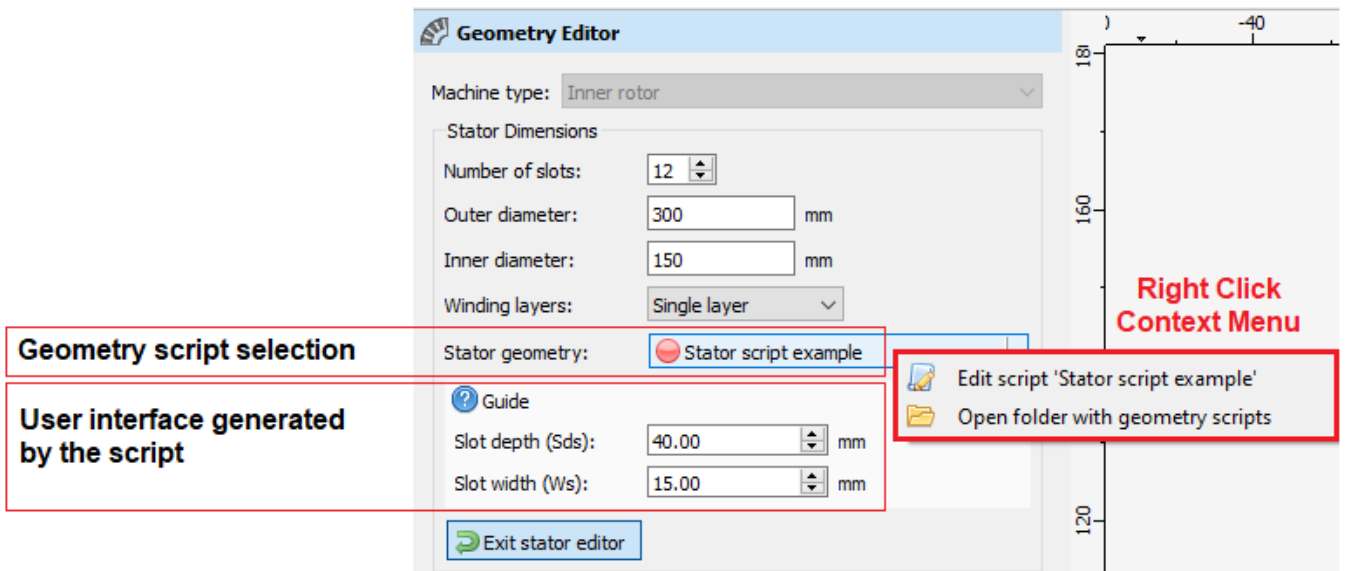


Figure 11.2. User interface corresponding to the *stator_script_example.js* script.

The structure of the stator geometry script looks as following (for rotor geometry script see section 11.7):

```
include('filename.js');

function Stator()
{
    // initialization code
};

Stator.prototype.getGeometry = function () {
    // generate stator geometry
};

Stator.prototype.getMaterials = function() {
    // assign materials
};

var stator = new Stator();

function StatorUI(ui)
{
    // stator widgets constructor
};

StatorUI.prototype.updateUI = function()
{
    // update widgets (control widgets visibility and etc.)
}

// call this function from c++ code
createStatorUI = function(ui)
{
    statorUI = new StatorUI(ui);
    statorUI.updateUI();
}
```


Rotor geometry script has similar structure with different names of some functions (see section 11.7). Function ***include('filename.js')*** allows to call functions defined in *filename.js* within the current script. In *stator_script_example.js* the ***include*** function is used to be able to use functions defined in *sDraw.js*. The following function defines the initial values of the slot depth and slot width variables ***Sds*** and ***Ws***:

```
function Stator()
{
    var Rsinner = motor.statorInnerRadius;    // Stator inner radius
    var Rsouter = motor.statorOuterRadius;    // Stator outer radius
    var Ns = motor.statorNumberSlots;         // Number of stator slots
    // Default stator dimensions:
    this.Sds = 0.5*Math.abs(Rsouter - Rsinner); // Slot depth
    this.Ws = Math.PI*(Rsinner+Rsouter)/Ns/4;  // Slot width
};
```

Function ***Stator*** uses global object ***motor*** containing information about motor parameters. Detailed description of the ***motor*** object can be found in section 11.4. Variables ***Sds*** and ***Ws*** are accessed using ***this*** keyword, so they are available inside the ***getGeometry*** and ***getMaterial*** functions. Call of the ***getGeometry*** function generates the geometry of the stator:

```
// generate stator geometry
Stator.prototype.getGeometry = function () {
    var Rsinner = motor.statorInnerRadius;    // Stator inner radius
    var Rsouter = motor.statorOuterRadius;    // Stator outer radius
    var Ns = motor.statorNumberSlots;         // Number of stator slots
    if (motor.isInnerRotor()) { //Inner rotor
        var point_1 = new QPointF(0, Rsinner+0.05*this.Sds);
        var point_2 = new QPointF(this.Ws, Rsinner+0.05*this.Sds);
        var point_3 = new QPointF(this.Ws, Rsinner+this.Sds);
        var point_4 = new QPointF(0, Rsinner+this.Sds);

        var geoms = new Array; // stator geometry array
        geoms.push(new Segment(point_1, point_2)); // line 1-2
        geoms.push(new Segment(point_2, point_3)); // line 2-3
        geoms.push(new Segment(point_3, point_4)); // line 3-4
        geoms.push(new Arc(90-360/Ns/2,90,Rsinner,0,0)); // inner radius arc
        geoms.push(new Arc(90-360/Ns/2,90,Rsouter,0,0)); // outer radius arc
    }
    else { //Outer rotor
        console.error("This script does not support outer rotor geometry");
    }
    return geoms;
};
```

As seen from the code above and from Figure 11.1 the geometry of the slot is determined by four points defined in the script by objects ***point_1***, ***point_2***, ***point_3***, ***point_4***. The geometry must be drawn inside the sector restricted by dash-and-dot blue lines in Figure 11.1 (between angles of $90-180/Ns$ and 90 degrees, where Ns – number of slots). The geometry of the stator is stored in the ***geoms*** array and defined by objects of type ***Segment*** and ***Arc***. Detailed information on the ***Segment*** and ***Arc*** object types can be found in sections 11.3.1 and 11.3.2. The geometry script *stator_script_example.js* is defined only for a

machine with inner rotor and *motor.isInnerRotor()* method (see section 11.4) is used to determine whether the inner rotor machine type is used. If the machine with outer rotor is used the error message will be displayed in the Console window.

Material types are assigned by the function *getMaterials*:

```
// assign materials
Stator.prototype.getMaterials = function() {
    var delta = 0.1;
    var materials = new Array(); // array of materials
    // stator iron material
    materials.push(new IronMaterial(0, motor.statorOuterRadius - delta) );
    // stator winding material
    var matWinding = new WindingMaterial(0, motor.statorInnerRadius + this.Sds/2);
    matWinding.layer = 1;
    materials.push(matWinding);
    return materials;
};
```

It can be seen that the material assignments are stored in the *materials* array. To assign the material type to the subdomain the point coordinates within this subdomain should be defined. In the above example ‘Iron’ and ‘Winding’ material types are assigned to corresponding subdomains using *IronMaterial* and *WindingMaterial* type objects inserted into the *materials* array. More information on assigning material types and corresponding objects and methods can be found in section 11.5.

User interface generated by the *stator_script_example.js* script is shown in Figure 11.2. There are only two fields called ‘Slot depth’ and ‘Slot width’. The script user interface is generated by the following function:

```
// stator widgets constructor
function StatorUI(ui)
{
    this.ui = ui;
    // Slot depth (Sds) controls:
    this.labelSds = new QLabel("Slot depth (Sds):"); // display widget name
    this.editSds = new QDoubleSpinBox; // create a spin box widget to enter Sds value
    this.editSds.setObjectName = "Sds"; // name of this object
    var Sds_max = Math.abs(motor.statorOuterRadius-motor.statorInnerRadius);
    this.editSds.setRange(0.0, Sds_max); // set min and max values of Sds
    this.editSds.decimals = 2; // precision of the Sds value in decimals
    this.editSds.singleStep = 1; // set the step value
    this.editSds.value = stator.Sds; // link the widget with variable
    this.unitsSds = new QLabel("mm"); // display units
    // Slot width (Ws) controls:
    this.labelWs = new QLabel("Slot width (Ws):");
    this.editWs = new QDoubleSpinBox;
    this.editWs.setObjectName = "Ws";
    var Ws_max = Math.PI*motor.statorInnerRadius/motor.statorNumberSlots;
    this.editWs.setRange(0.0, Ws_max);
```

```

this.editWs.decimals = 2;
this.editWs.singleStep = 1;
this.editWs.value = stator.Ws;
this.unitsWs = new QLabel("mm");

this.layout = new QGridLayout(ui);
// Add slot depth controls to UI:
this.layout.addWidget(this.labelSds, 0, 0);
this.layout.addWidget(this.editSds, 0, 1);
this.layout.addWidget(this.unitsSds, 0, 2);
// Add slot width controls to UI:
this.layout.addWidget(this.labelWs, 1, 0);
this.layout.addWidget(this.editWs, 1, 1);
this.layout.addWidget(this.unitsWs, 1, 2);
};

```

There are three widgets created for every motor parameter in the example: widget name text (*QLabel* object type), spin box widget (*QDoubleSpinBox* object type) and units text (*QLabel* object type). The *layout* object of *QGridLayout* type lays out widgets in a grid, i.e. divides them up into rows and columns, and puts each widget into the correct cell. Rows and columns are defined by the *addWidget* function. More information on creating user interfaces and working with widgets can be found in corresponding Qt documentation, an example of creating a group of widgets and controlling the widget visibility can be found in section 11.7.

11.2. Working with console.

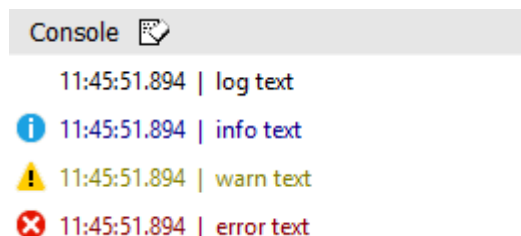
Console functions allow displaying the information in the Console window while the script is being executed. The following functions can be used to display general information, warnings and error messages:

```

console.log('log text');
console.info('info text');
console.warn('warn text');
console.error('error text');

```

The result of this code will be shown in the Console window as follows:



Function *console.clear()* deletes all messages from the Console window.

Function *console.dir(object)* displays information stored in the *object*.

11.3. Generating geometry.

As it was already shown in the example of section 11.1 the geometry is programmatically generated inside the *getGeometry* function and defined as an array of geometry objects (*geoms* array in the example of section 11.1). There are two geometry object types: *Segment* and *Arc*.

11.3.1. Adding line segments.

Segment object is a line segment that is a part of a line that is bounded by two distinct end points. There are several ways to create a *Segment* object:

```
// example 1
var segment1 = new Segment;
segment1.x1 = 1; segment1.y1 = 1; segment1.x2 = 1; segment1.y2 = 2;
// example 2
var segment2 = new Segment;
segment2.p1 = new QPointF(1, 1); segment2.p2 = new QPointF(1, 2);
// example 3
var x1 = 1; var y1 = 1; var x2 = 1; var y2 = 2;
var segment3 = new Segment(x1, y1, x2, y2);
// example 4
var p1 = new QPointF(1, 1); var p2 = new QPointF(1, 2);
var segment4 = new Segment(p1, p2); // p1 and p2 are objects of QPointF type
```

In all four examples a straight line connecting two points (1,1) and (1,2) is created. In the first two examples the empty *Segment* object is created first. Then its properties are assigned to define the start and end point coordinates. As seen from the examples 1 and example 2 there are six properties in a *Segment* object which can be used: *x1*, *y1*, *x2*, *y2* which are the start and end point coordinates (see example 1) and *p1*, *p2* which are the start and end point objects of *QPointF* type (see example 2).

Two other *Segment* properties can be used to determine the segment length and angle:

```
var l = segment1.length; // segment length
var a = segment1.angle;  // segment angle in degrees from the X-axis
```

11.3.2. Adding arcs.

An *Arc* object is a part of a circle that is bounded by two points. There are several ways to create an *Arc* object:

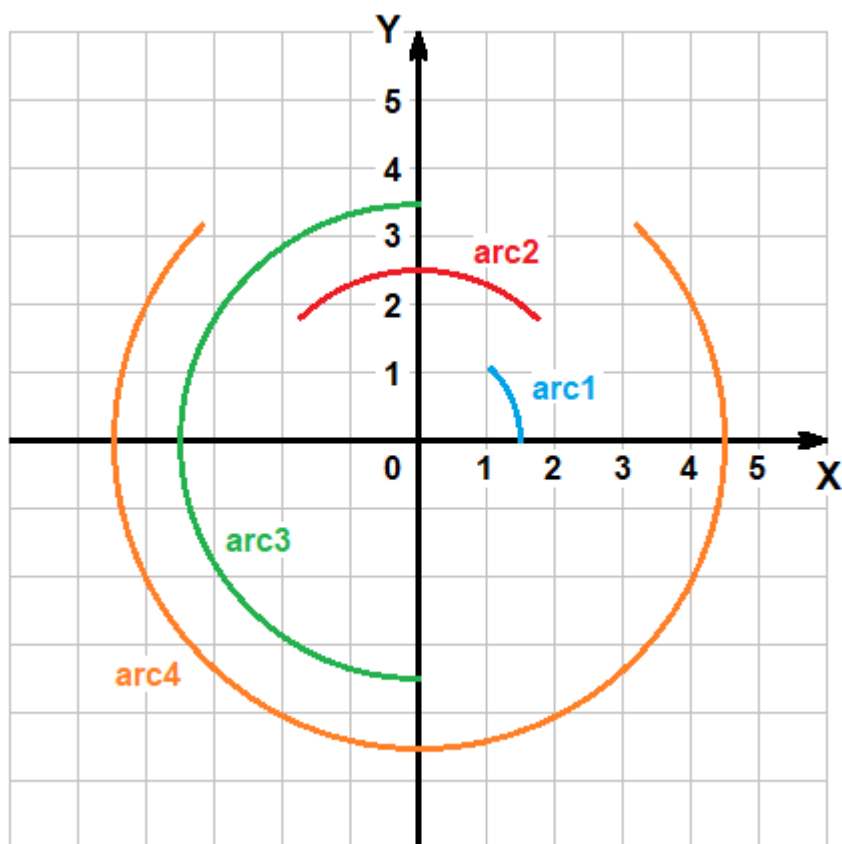
```
// example 1
var arc1 = new Arc;
arc1.angle1 = 0;      // arc start angle (counterclockwise direction)
arc1.angle2 = 45;     // arc end angle (counterclockwise direction)
arc1.center = new QPointF(0, 0); // arc center
arc1.radius = 1.5;    // arc radius
// example 2
var arc2 = new Arc;
arc2.angle1 = 45;     // arc start angle (counterclockwise direction)
arc2.angle2 = 135;    // arc end angle (counterclockwise direction)
arc2.radius = 2.5;    // arc radius
arc2.centerX = 0;     // arc center x coordinate
arc2.centerY = 0;     // arc center y coordinate
```

```

// example 3
var a1 = 90;           // arc start angle (counterclockwise direction)
var a2 = 270;          // arc end angle (counterclockwise direction)
var c = new QPointF(0, 0); // arc center
var r = 3.5;           // arc radius
var arc3 = new Arc(a1, a2, r, c);
// example 4
var a1 = 135;          // arc start angle (counterclockwise direction)
var a2 = 45;           // arc end angle (counterclockwise direction)
var cx = 0;            // arc center x coordinate
var cy = 0;            // arc center y coordinate
var r = 4.5;           // arc radius
var arc4 = new Arc(a1, a2, r, cx, cy);

```

The result of this code is the four arcs as shown below:



In the first two examples the empty *Arc* object is created first. Then its properties are assigned to define the start and end angles, arc radius and arc center coordinates. As seen from the examples 1 and 2 there are six properties in an *Arc* object which can be used: *angle1* and *angle2* - start and end angles in degrees, *radius* - arc radius, *center* - arc center point object of *QPointF* type, *centerX* and *centerY* - arc center coordinates. Start and end angles are defined so that the direction of the arc is counterclockwise (compare example 2 and example 3).

Three other *Arc* properties can be used to determine the arc start and end points and arc length:

```

var p1 = arc.p1;       // arc start point, object of QPointF type
var p2 = arc.p2;       // arc end point, object of QPointF type
var l = arc.length;    // arc length

```

To determine the type of the geometry object the *isSegment* and *isArc* functions can be used:

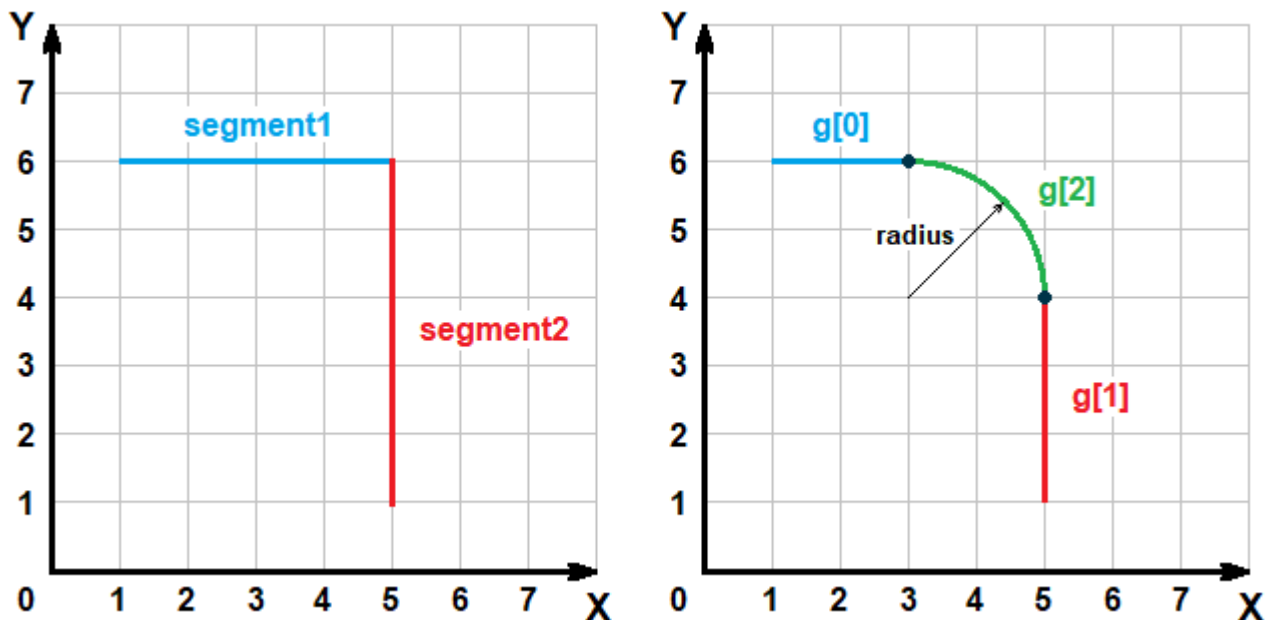
```
var arc = new Arc(0, 90, 1, 0, 0);
var issegment = arc.isSegment(); // false
var isarc = arc.isArc(); // true
var segment = new Segment(0, 0, 1, 1);
var issegment = segment.isSegment(); // true
var isarc = segment.isArc(); // false
```

11.3.3. Rounding corners.

Function *couplingSegments* allows to round the corner of two line segments or a line segment and an arc defined by corresponding *Segment* and *Arc* objects. The corner is rounded by an arc smoothly coupling two geometry elements.

The following example demonstrates rounding the corner of two line segments:

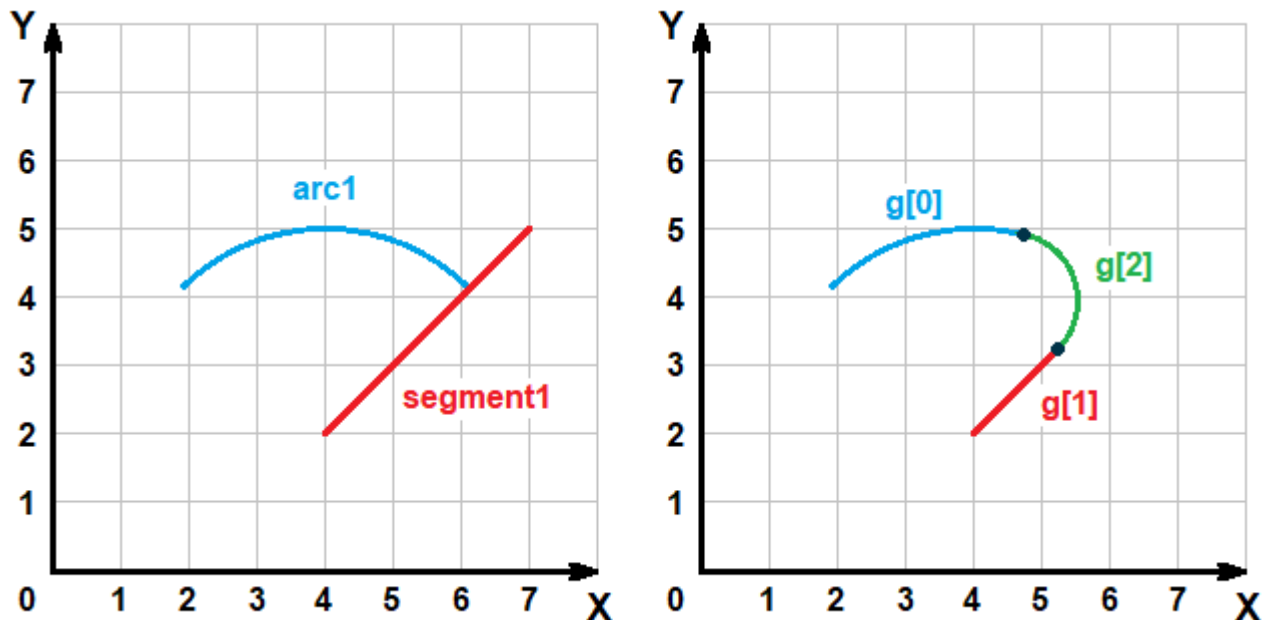
```
var geom = new Array; // geometry array
var segment1 = new Segment(1, 6, 5, 6);
var segment2 = new Segment(5, 1, 5, 6);
var radius = 2; // coupling arc radius
var g = couplingSegments(segment1, segment2, radius);
geom.push(g[0]); // g[0] - segment1 with adjusted length
geom.push(g[1]); // g[1] - segment2 with adjusted length
geom.push(g[2]); // g[2] - coupling arc
```



The picture on the left shows two initial line segments defined by the *segment1* and *segment2* objects. According to the above example and the picture on the right the *couplingSegments* function returns an array *g* consisting of the line segment objects with adjusted lengths *g[0]* and *g[1]* and the coupling arc object *g[2]*. These objects are inserted into the *geom* array with the *push* function.

Another example demonstrates rounding the corner of a line segment and an arc:

```
var geom = new Array; // geometry array
var arc1 = new Arc(45, 135, 3, 4, 2);
var segment1 = new Segment(4, 2, 7, 5);
var radius = 1; // coupling arc radius
var g = couplingSegments(arc1, segment1, radius);
geom.push(g[0]); // g[0] - arc1 with adjusted length
geom.push(g[1]); // g[1] - segment1 with adjusted length
geom.push(g[2]); // g[2] - coupling arc
```



The picture on the left shows the initial arc and the line segment defined by the *arc1* and *segment1* objects. According to the above example and the picture on the right the *couplingSegments* function returns an array *g* consisting of the arc segment object with adjusted length *g[0]*, the line segment object with adjusted length *g[1]* and the coupling arc object *g[2]*. These objects are inserted into the *geom* array with the *push* function.

In some cases when it is not possible to couple two elements the *couplingSegments* function returns coupling arc of zero length (empty *Arc* object: *angle1* = 0, *angle2* = 0, *radius* = 0, *centerX* = 0, *centerY* = 0). In the following example two parallel line segments are being coupled resulting in an empty coupling *Arc* object *g[2]* returned by the *couplingSegments* function:

```
var segment1 = new Segment(4, 4, 7, 4);
var segment2 = new Segment(4, 6, 7, 6);
var g = couplingSegments(segment1, segment2, 3);
console.info(g[2]); // g[2] - empty coupling arc
```

Since the line segments defined by the *segment1* and *segment2* objects are parallel the coupling arc object is empty and the result of this code will be shown in the Console window as follows:

Console 

 21:05:24.254 | Arc [A1:0° A2:0° R:0 C:(0, 0)]

11.3.4. *sDraw* objects and methods.

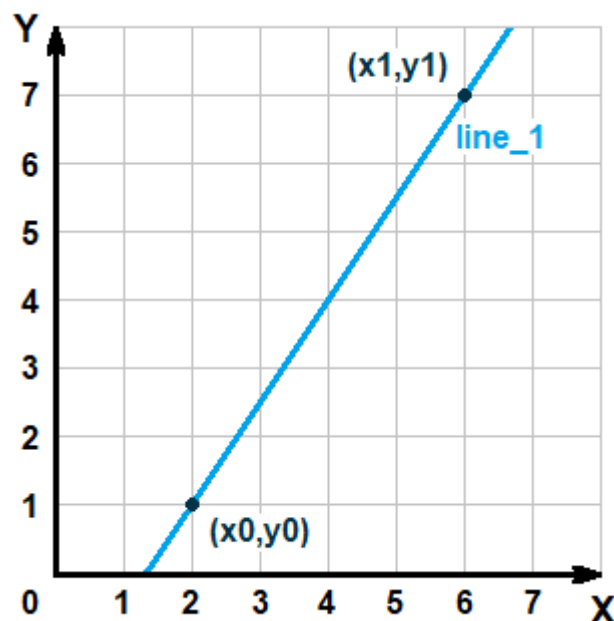
sDraw objects and methods are used to make the process of geometry drawing easier and allow to draw complex geometries without using complicated mathematical expressions. Unlike the previously described *Segment* and *Arc* objects the *sDraw* objects are not intended to be visible. There are two main *sDraw* object types: *Line* and *Circle*.

11.3.4.1. Creating *Line* objects.

There are three ways (functions) to create *Line* objects:

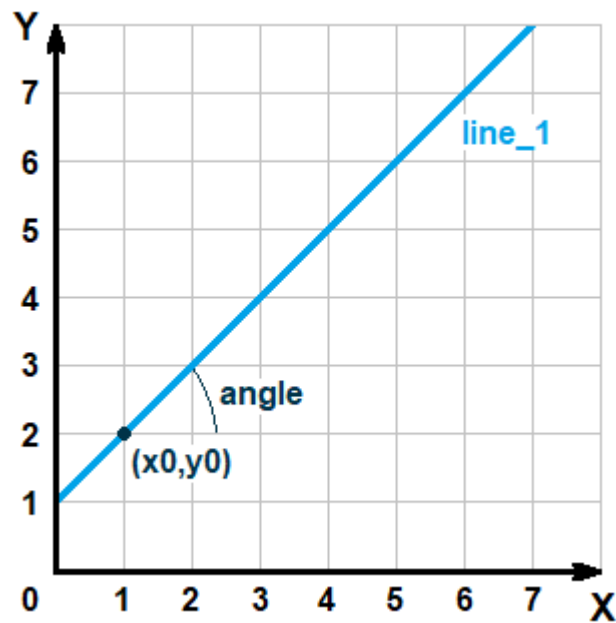
1. Line passing through two points (*createLineBy2Points* function);
2. Line passing through a point at a given angle (*createLineByPointAngle* function);
3. Line parallel to the given line and located a given distance from it (*createLineParallel* function);

Using *createLineBy2Points* function.



```
var x0 = 2; var y0 = 1;
var x1 = 6; var y1 = 7;
var line_1 = sDraw.createLineBy2Points(x0,y0,x1,y1);
```

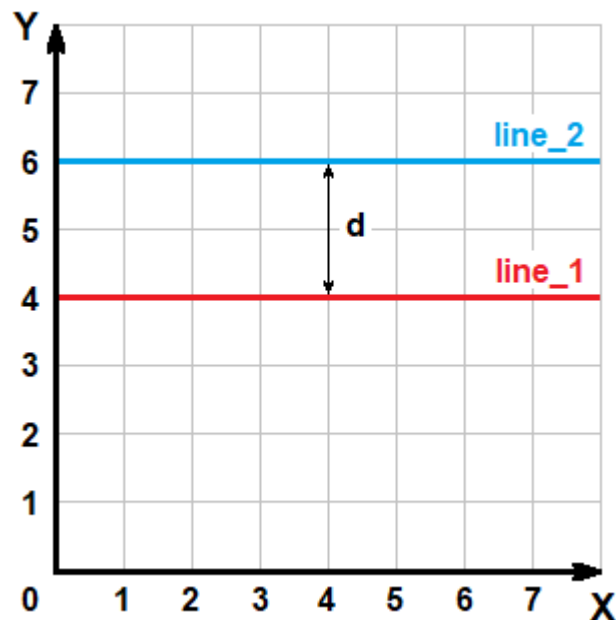

Using *createLineByPointAngle* function.



```
var x0 = 1; var y0 = 2;
var angle = 45; // angle in degrees
var line_1 = sDraw.createLineByPointAngle(x0,y0,angle);
```

Input argument *angle* is an angle in degrees from the X-axis in the range [-360; 360].

Using *createLineParallel* function.



```
var x0 = 1; var y0 = 4;
var x1 = 7; var y1 = 4;
var line_1 = sDraw.createLineBy2Points(x0,y0,x1,y1);
var d = 2;
var line_2 = sDraw.createLineParallel(line_1,d,'higher');
```

In the example shown above *line_1* is first created using function *createLineBy2Points*. Then *line_2* is created using the function *createLineParallel*. Input argument *d* defines the distance between two lines.

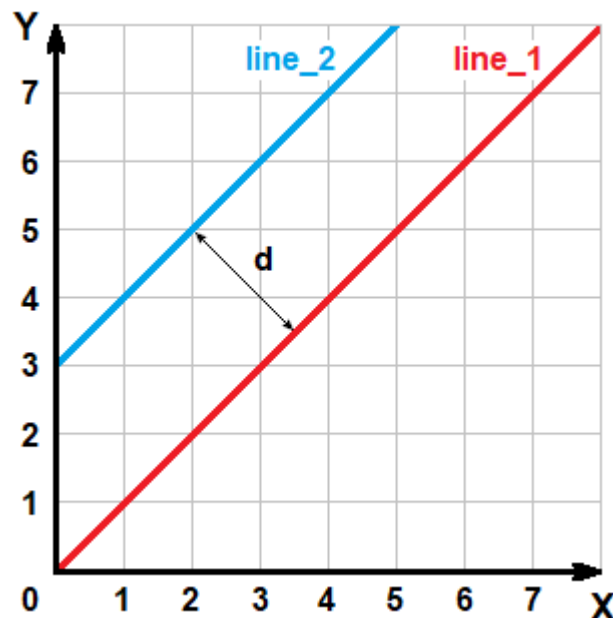
The third input argument of the function *createLineParallel* determines the position of *line_2* related to *line_1*. The position can be either 'higher', 'lower', 'left', 'right'.

Avoid using 'left' or 'right' position if lines are parallel to the *x*-axis and using 'higher' or 'lower' position if lines are parallel to the *y*-axis, it will lead to unpredictable results. The following example demonstrates **incorrect** using of the *createLineParallel* function (see previous picture):

```
var x0 = 1; var y0 = 4;
var x1 = 7; var y1 = 4;
var line_1 = sDraw.createLineBy2Points(x0,y0,x1,y1);
var d = 2;
var line_2 = sDraw.createLineParallel(line_1,d,'left');
```

Position 'left' is incorrect in this case since *line_1* is parallel to the *x*-axis, 'higher' or 'lower' must be used instead.

Another example of using *createLineParallel* function:



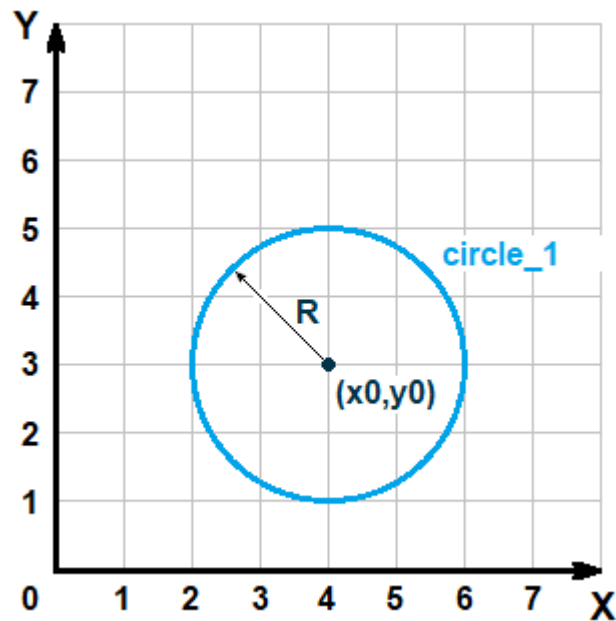
```
var d = Math.sqrt(2*1.5*1.5);
var line_2 = sDraw.createLineParallel(line_1,d,'higher');
```

Another variant with the position defined as 'left' is also correct:

```
var line_2 = sDraw.createLineParallel(line_1,d,'left');
```

11.3.4.2. Creating *Circle* objects.

There is only one function *createCircle* to create *Circle* objects:

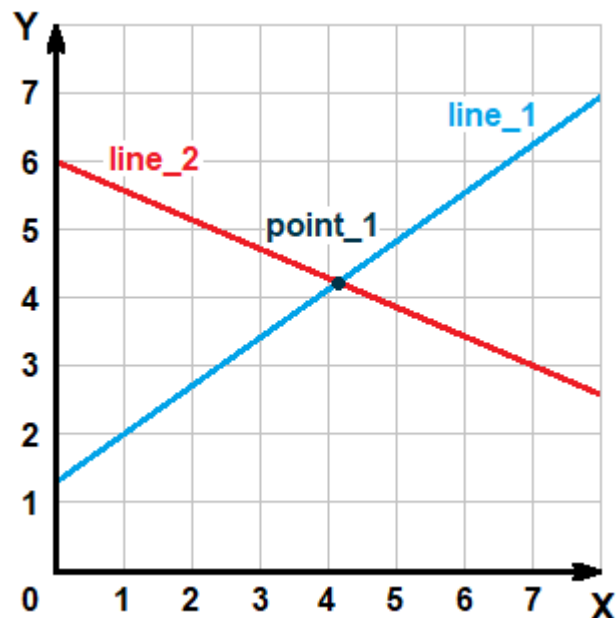


```
var x0 = 4; var y0 = 3; var R = 2;
var circle_1 = sDraw.createCircle(x0,y0,R);
```

11.3.4.3. Using *sDraw.intersection* method.

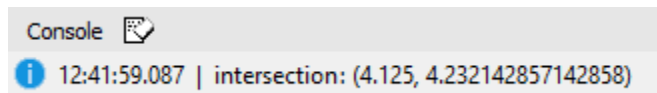
sDraw.intersection function allows to find coordinates of an intersection point between two lines or between a line and a circle or between two circles.

1. Intersection of two lines.



```
var line_1 = sDraw.createLineBy2Points(1,2,8,7);
var line_2 = sDraw.createLineBy2Points(0,6,7,3);
var point_1 = sDraw.intersection(line_1, line_2);
console.info("intersection: (" + point_1.x() + ", " + point_1.y() + ")");
```

The result will be shown in the Console window as follows:

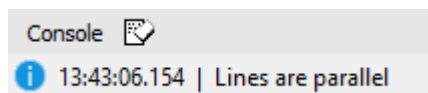


sDraw.intersection function returns an object of type *QPointF* (see Qt documentation for more information); *x()* and *y()* functions of *QPointF* class were used in the above example to retrieve *x* and *y* coordinates of *point_1* object.

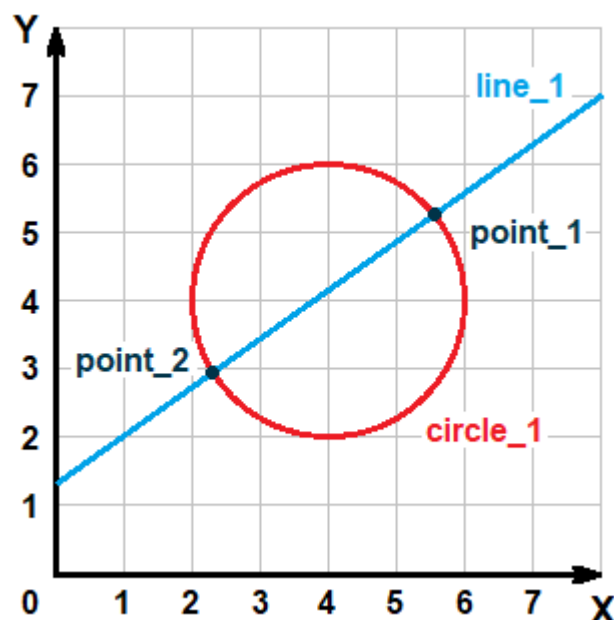
If *sDraw.intersection* function returns *null* when lines are parallel:

```
var line_1 = sDraw.createLineByPointAngle(1,2,30);
var line_2 = sDraw.createLineByPointAngle(3,4,30);
var point_1 = sDraw.intersection(line_1, line_2);
if (point_1 == null) { console.info("Lines are parallel"); }
```

The result shown in the Console window:



2. Intersection of a line and a circle.



```
var line_1 = sDraw.createLineBy2Points(1,2,8,7);
var circle_1 = sDraw.createCircle(4,4,2);
var position = 'higher';
var point_1 = sDraw.intersection(line_1, circle_1, position);
position = 'left';
var point_2 = sDraw.intersection(circle_1, line_1, position);
```

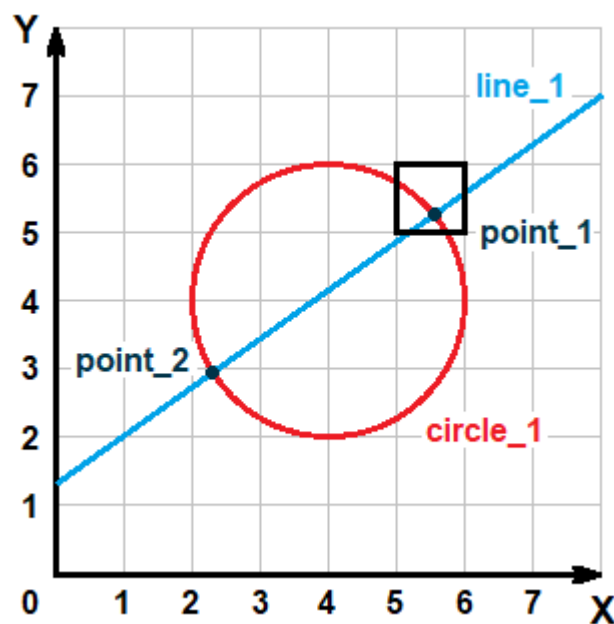
The example above demonstrates how to use the *sDraw.intersection* function to find the intersection point of *line_1* and *circle_1*. The *position* variable can be either 'higher', 'lower', 'left', 'right' or an array defining rectangle area containing intersection point. In the above example both intersection points *point_1* and *point_2* are determined. The following code will show the result in the Console window:

```
console.info("point_1: (" + point_1.x() + ", " + point_1.y() + ")");
console.info("point_2: (" + point_2.x() + ", " + point_2.y() + ")");
```

Console

```
11:25:30.184 | point_1: (5.557147949475995, 5.25510567819714)
11:25:30.186 | point_2: (2.3077169153888692, 2.9340835109920493)
```

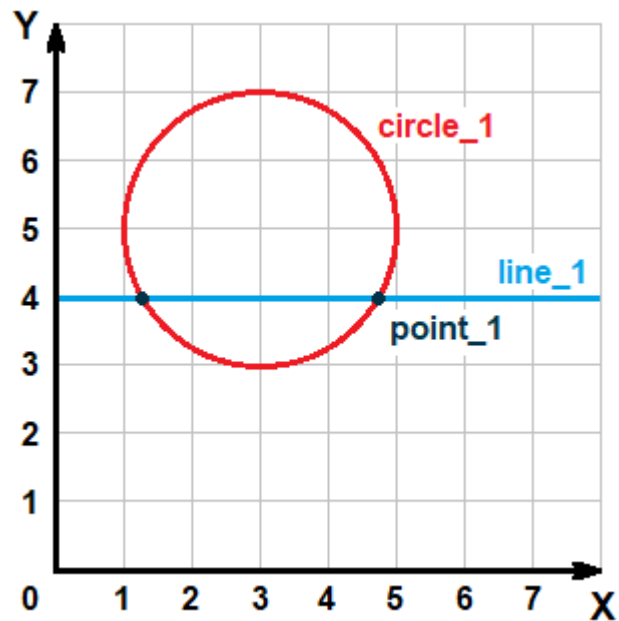
The following example demonstrates how to use the *sDraw.intersection* function if the position of the intersection point is defined by the rectangle area:



```
var line_1 = sDraw.createLineBy2Points(1,2,8,7);
var circle_1 = sDraw.createCircle(4,4,2);
var position = [new QPointF(5, 5), new QPointF(6, 6)];
var point_1 = sDraw.intersection(line_1, circle_1, position);
```

In this case the rectangle with the intersection point is defined by an array consisting of two points (5, 5) and (6, 6).

Avoid using 'left' or 'right' position if the line is parallel to the y-axis and using 'higher' or 'lower' position if the line is parallel to the x-axis, it will lead to unpredictable results. The following example demonstrates **incorrect** using of the *sDraw.intersection* function:

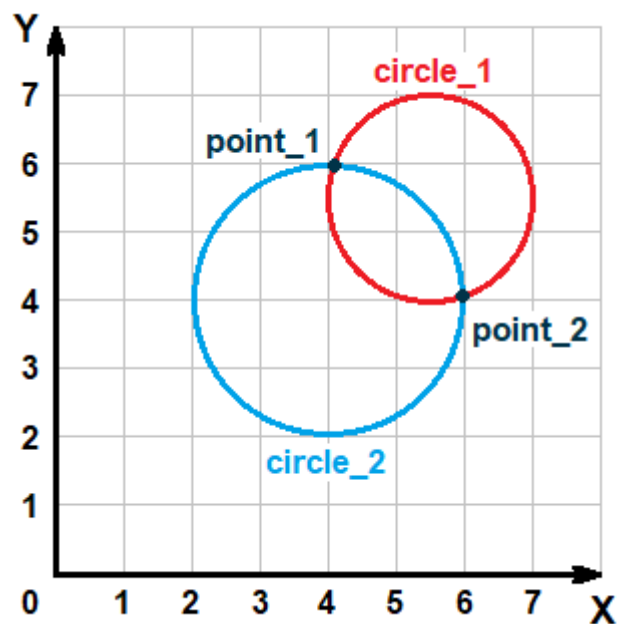


```
var line_1 = sDraw.createLineBy2Points(1,4,7,4);
var circle_1 = sDraw.createCircle(3,5,2);
var point_1 = sDraw.intersection(line_1, circle_1, 'higher');
```

Position `'higher'` is incorrect in this case since *line_1* is parallel to the x-axis, `'right'` or `'left'` must be used instead.

If the *sDraw.intersection* function returns *null* when the line and the circle do not intersect.

3. Intersection of two circles.



```

var circle_1 = sDraw.createCircle(5.5,5.5,1.5);
var circle_2 = sDraw.createCircle(4,4,2);
var position = 'higher';
var point_1 = sDraw.intersection(circle_1, circle_2, position);
position = 'right';
var point_2 = sDraw.intersection(circle_1, circle_2, position);

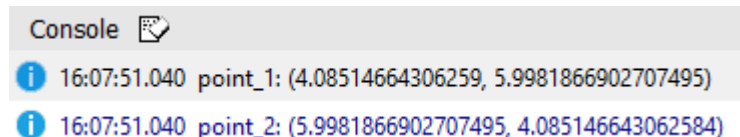
```

The example above demonstrates how to use the *sDraw.intersection* function to find the intersection point of two circles defined by objects *circle_1* and *circle_2*. The *position* variable can be either 'higher', 'lower', 'left', 'right' or an array defining rectangle area containing intersection point. In the above example both intersection points *point_1* and *point_2* are determined. The following code will show the result in the Console window:

```

console.info("point_1: (" + point_1.x() + ", " + point_1.y() + ")");
console.info("point_2: (" + point_2.x() + ", " + point_2.y() + ")");

```



```

Console
16:07:51.040 point_1: (4.08514664306259, 5.9981866902707495)
16:07:51.040 point_2: (5.9981866902707495, 4.085146643062584)

```

The definition of the position of the intersection point by rectangle area is similar to the previous example for intersection of a line and a circle.

Using 'left' or 'right' position must be avoided if centers of two circles have the same y coordinates – 'higher' or 'lower' position must be used in this case. Similarly, 'left' or 'right' position must be used if centers of two circles have the same x coordinates.

11.3.5. Simple stator geometry script example using sDraw objects and methods.

In this section the example of stator geometry script *stator_script_example.js* previously considered in section 11.1 is used with some modifications. Full source code of the script can be found in *[MotorXP-PM installation directory]\bin\assets\scripts\Stator\stator_script_example.js*. The only changes are made in the *getGeometry* function which after modification looks as follows:

```

// generate stator geometry
Stator.prototype.getGeometry = function () {
    var Rsinner = motor.statorInnerRadius;    // Stator inner radius
    var Rsouter = motor.statorOuterRadius;    // Stator outer radius
    var Ns = motor.statorNumberSlots;        // Number of stator slots
    if (motor.isInnerRotor()) { //Inner rotor
        var point_1 = new QPointF(0, Rsinner+0.05*this.Sds);
        var point_2 = new QPointF(this.Ws, Rsinner+0.05*this.Sds);
        var point_3 = new QPointF(this.Ws, Rsinner+this.Sds);
        //var point_4 = new QPointF(0, Rsinner+this.Sds);
        var line_1_4 = sDraw.createLineByPointAngle(point_1.x(),point_1.y(),90);
        var line_3_4 = sDraw.createLineByPointAngle(point_3.x(),point_3.y(),135);
        var point_4 = sDraw.intersection(line_1_4, line_3_4);
    }
}

```

```

var geoms = new Array; // stator geometry array
geoms.push(new Segment(point_1, point_2)); // line 1-2
geoms.push(new Segment(point_2, point_3)); // line 2-3
geoms.push(new Segment(point_3, point_4)); // line 3-4
geoms.push(new Arc(90-360/Ns/2,90,Rsinner,0,0)); // inner radius arc
geoms.push(new Arc(90-360/Ns/2,90,Rsouter,0,0)); // outer radius arc
}
else { //Outer rotor
  console.error("This script does not support outer rotor geometry");
}
return geoms;
};

```

As can be seen the *point_4* is now defined as an intersection of two *Line* objects named *line_1_4* and *line_3_4*. Figure 11.3 shows the changed stator slot geometry generated by the modified script (compare with Figure 11.1).

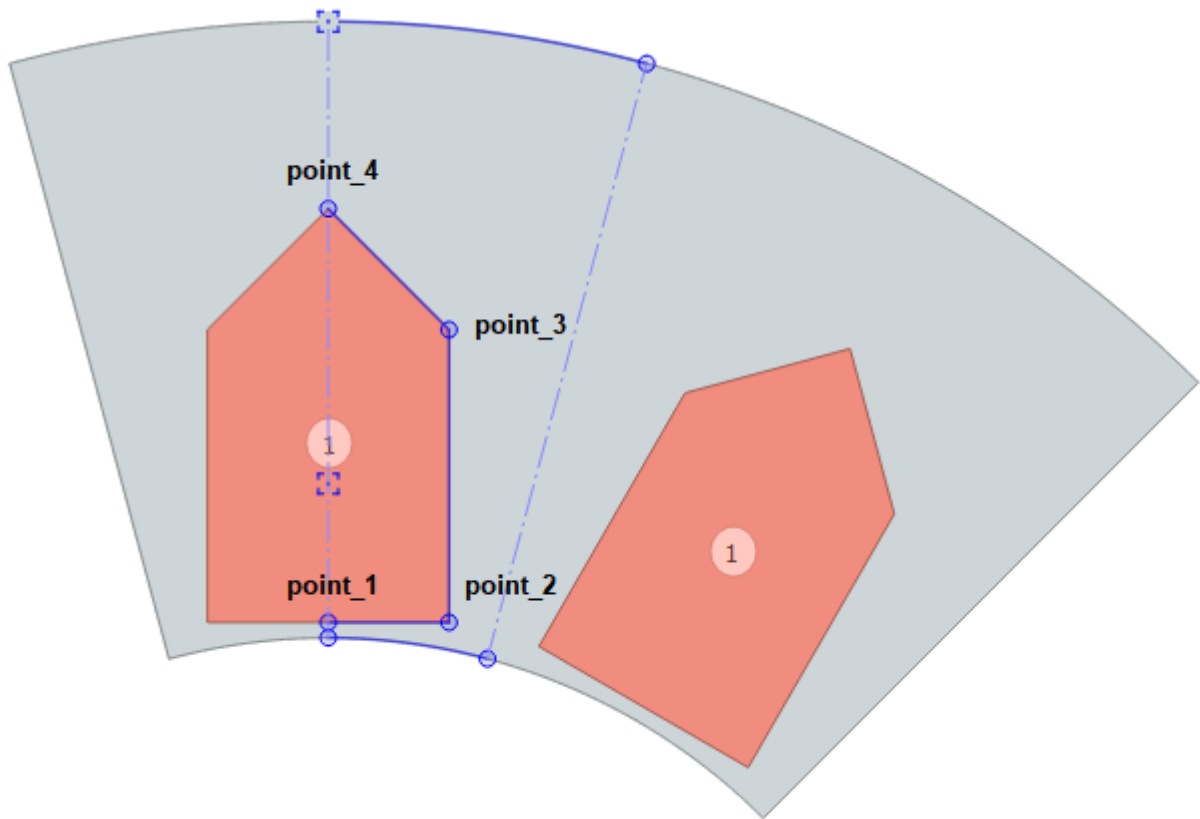


Figure 11.3. Stator geometry generated by modified *stator_script_example.js*.

11.4. Global object *motor*.

The general parameters of the machine geometry, such as machine type (inner or outer rotor), number of stator slots, stator outer and inner diameter, number of slot layers, slot layers orientation (upper/lower or left/right), stator and rotor geometry script names, air gap length, number of pole pairs, rotor outer and inner diameter, are stored in the global object named *motor*. The *motor* object was already used in the example of section 11.1. The list of all machine parameters stored in the *motor* object:

```
// Machine type
if (motor.machineType == 'InnerRotor') {}
if (motor.machineType == 'OuterRotor') {}
// Functions to check the machine type
var isinnerRotor = motor.isInnerRotor(); // return true | false
var isouterRotor = motor.isOuterRotor(); // return true | false

// Number of stator slots
var Ns = motor.statorNumberSlots;

// Stator outer diameter (mm)
var Dso = motor.statorOuterDiameter;
// Stator outer radius (mm)
var Rso = motor.statorOuterRadius; // = motor.statorOuterDiameter/2;
// Stator inner diameter (mm)
var Dsi = motor.statorInnerDiameter;
// Stator inner radius (mm)
var Rsi = motor.statorInnerRadius; // = motor.statorInnerDiameter/2;

// Number of stator slot layers
var Nsl = motor.statorNumberSlotLayers; // return 1 or 2
// Functions to check the number of stator slot layers
var issingleLayer = motor.isStatorNumberSlotLayersSingle(); // return true | false
var isdoubleLayer = motor.isStatorNumberSlotLayersDouble(); // return true | false

// Stator slot layers orientation
if (motor.statorSlotLayersOrientation == 'UpperLower') {}
if (motor.statorSlotLayersOrientation == 'LeftRight') {}
// Functions to check the stator slot layers orientation
var isul = motor.isStatorSlotLayersOrientationUpperLower(); // return true | false
var islr = motor.isStatorSlotLayersOrientationLeftRight(); // return true | false

// Stator geometry script name
var scriptname_stator = motor.statorScript;

// Rotor geometry script name
var scriptname_rotor = motor.rotorScript;

// Air gap (mm)
var Ga = motor.airGap;

// Rotor outer diameter (mm)
var Dro = motor.rotorOuterDiameter;
// Rotor outer radius (mm)
var Rro = motor.rotorOuterRadius; // = motor.rotorOuterDiameter/2;
// Rotor inner diameter (mm)
var Dri = motor.rotorInnerDiameter;
// Rotor inner radius (mm)
var Rsi = motor.statorInnerRadius; // = motor.rotorInnerDiameter/2;

// Number of pole pairs
var nPolePairs = motor.rotorNumberPairPoles;
```

11.5. Assigning material types.

As it was already shown in the example of section 11.1 the material types are programmatically assigned to each subdomain inside the *getMaterial* function which should return an array of material assignment objects. To assign the material type to a subdomain the point within this subdomain should be defined.

There are five material types:

1. ‘General’: nonconductive nonmagnetic materials (air, insulation, etc.);
2. ‘Iron’: ferromagnetic material with nonlinear B-H curve (laminated steel core);
3. ‘Winding’: stranded conductor material (winding coils);
4. ‘Conductor’: solid conductor material (for example, retaining sleeve, see chapter 4).
5. ‘Magnet’: permanent magnet material;

11.5.1. Assigning ‘General’ material type.

Two examples of how to assign the material type ‘General’ to the subdomain:

```
var materials = new Array();           // array of materials
var matGeneral_1 = new GeneralMaterial(1, 5); // point (1, 5) belongs to subdomain
                                           // the 'General' is assigned to
materials.push(matGeneral_1);           // insert material assignment object
var matGeneral_2 = new GeneralMaterial;
matGeneral_2.pos = new QPointF(10, 10); // point (10, 10) belongs to subdomain
                                           // the 'General' is assigned to
materials.push(matGeneral_2);           // insert material assignment object
```

In the above examples the material type ‘General’ is assigned to two subdomains and two material assignment objects of the *GeneralMaterial* type are created. In each case the point within the subdomain is defined to assign the ‘General’ material type. In the first case (*matGeneral_1* object) the material assignment point is directly defined during call of the *GeneralMaterial* function creating the material assignment object. In the second case (*matGeneral_2* object) the material assignment point is defined by assigning the *pos* property of the *matGeneral_2* object. Both material assignment objects *matGeneral_1* and *matGeneral_2* are inserted into the *materials* array which is then returned by the *getMaterial* function.

11.5.2. Assigning ‘Iron’ material type.

Two examples of how to assign the material type ‘Iron’ to the subdomain:

```
var materials = new Array();           // array of materials
var matIron_1 = new IronMaterial(1, 3); // point (1, 3) belongs to subdomain
                                           // the 'Iron' is assigned to
materials.push(matIron_1);             // insert material assignment object
var matIron_2 = new IronMaterial;
matIron_2.pos = new QPointF(2, 6);     // point (2, 6) belongs to subdomain
                                           // the 'Iron' is assigned to
materials.push(matIron_2);             // insert material assignment object
```

In the above examples the material type ‘Iron’ is assigned to two subdomains and two material assignment objects of the *IronMaterial* type are created. In each case the point within the subdomain is defined to assign the ‘Iron’ material type. In the first case (*matIron_1* object) the material assignment point is directly defined during call of the *IronMaterial* function creating the material assignment object. In the second case (*matIron_2* object) the material assignment point is defined by assigning the *pos* property of the *matIron_2* object. Both material assignment objects *matIron_1* and *matIron_2* are inserted into the *materials* array which is then returned by the *getMaterial* function.

11.5.3. Assigning ‘Winding’ material type.

The ‘Winding’ material type is assigned to the subdomain using the *WindingMaterial* function in a similar way as it was already shown for ‘General’ and ‘Iron’ material types in two previous sections. An example of assigning the ‘Winding’ material type can be found in section 11.1 in the description of the *getMaterial* function. The *layer* property of the material assignment objects of the *WindingMaterial* type defines the layer number to link the geometry with stator windings. Each subdomain assigned with the ‘Winding’ material type should have a unique layer number starting from ‘1’ even if there is only one layer in the slot. Note that if there are two layers in the slot but slot layers orientation is ‘Left / Right’ when there should be only one subdomain with the ‘Winding’ material type defined in the script and its layer number should be ‘1’. Second layer will be automatically added by **Geometry Editor**. See implementation of the *getMaterial* function in *[MotorXP-PM installation directory]\bin\assets\scripts\Stator\Parallel_tooth.js* for more examples on assigning the ‘Winding’ material type for different stator slot configurations.

11.5.4. Assigning ‘Conductor’ material type.

The ‘Conductor’ material type is assigned to the subdomain using the *ConductorMaterial* function in a similar way as it was already shown for ‘General’ and ‘Iron’ material types in sections 11.5.1 and 11.5.2. See implementation of the *getMaterial* function in *[MotorXP-PM installation directory]\bin\assets\scripts\Rotor\halback_array_rotor.js* for more examples on assigning the ‘Conductor’ material type.

11.5.5. Assigning ‘Magnet’ material type.

There are two magnetization types used in MotorXP-PM: ‘Radial’ and ‘Parallel’ and two corresponding functions used to assign the ‘Magnet’ material type to the subdomain depending on the magnetization: *MagnetRadialMaterial* and *MagnetParallelMaterial*. Example of how to assign the material type ‘Magnet’ with parallel magnetization to the subdomain:

```
var materials = new Array(); // array of materials
var matMagnet = new MagnetParallelMaterial;
matMagnet.pos = new QPointF(0.1, 7); // point (0.1, 7) belongs to subdomain
// the 'Magnet' is assigned to
matMagnet.angle = 90; // magnetization angle
materials.push(matMagnet); // insert material assignment object
```

In general, the process of assigning the ‘Magnet’ material type is similar to other material types considered in sections 11.5.1 – 11.5.4. The **angle** property of the material assignment object (**matMagnet.angle** in the above example) defines the magnetization angle of the permanent magnet in degrees from the X-axis in the range [0; 360]. The magnetization angle is assigned through the script only for magnets in the right half of pole pitch and the magnetization angles for the rest of the geometry are automatically determined (Figure 11.4(a)). The magnetization angle assigned in the script for the right half of pole pitch is mirrored for the left half of pole pitch except when the magnetization angle is equal to 0 (Figure 11.4(b)). For example, the magnetization angle of 107.5° will be mirrored to 72.5° in the left half of pole pitch (Figure 11.4(a)). Note that the magnetization angles of 90° and 180° are mirrored to angles of the same values of 90° and 180° .

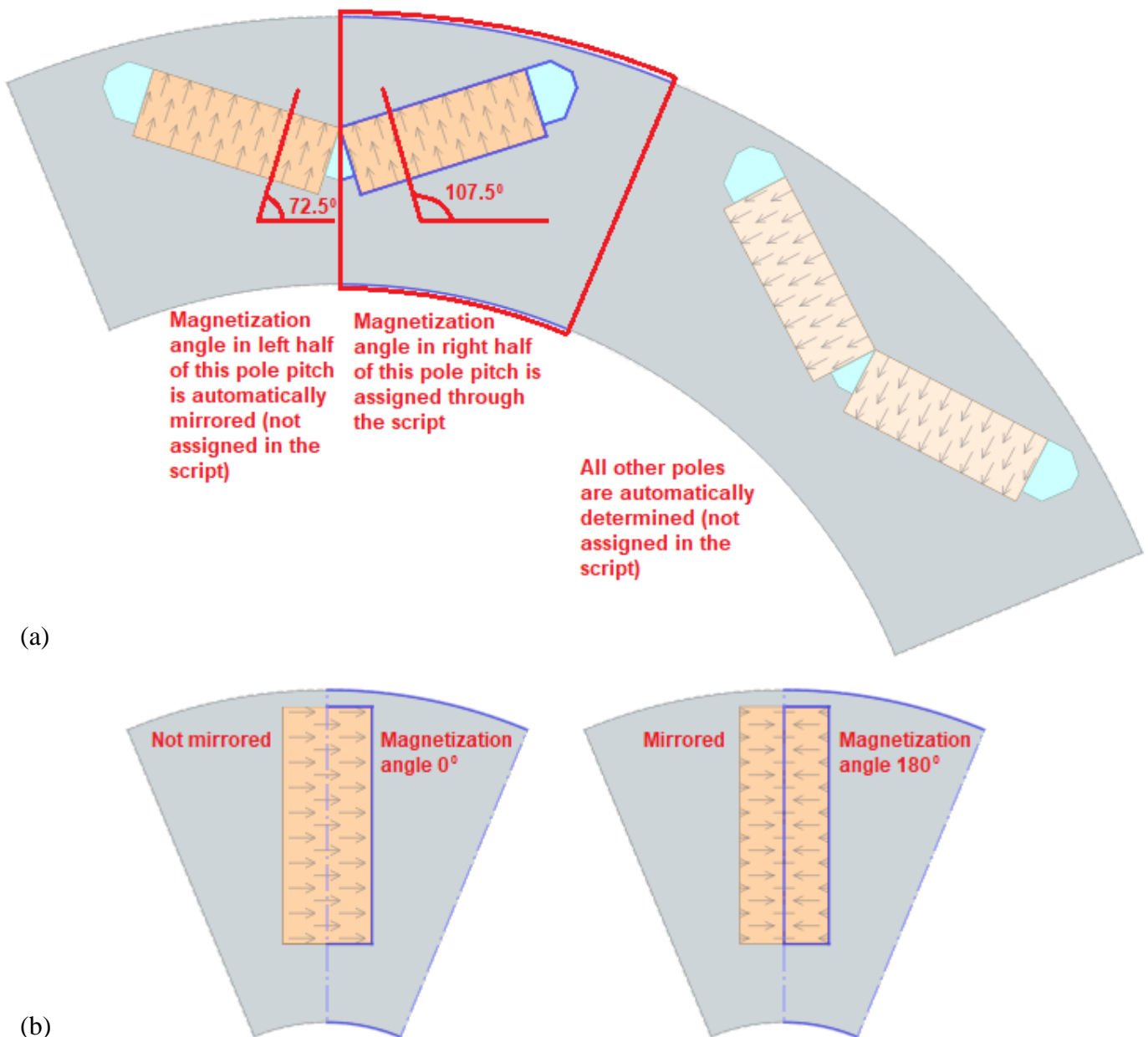


Figure 11.4. (a) Explanation on magnetization angle assignment; (b) Magnetization angle of 0° is not mirrored.

The following example demonstrates how to assign the material type ‘Magnet’ with radial magnetization to the subdomain using the *MagnetRadialMaterial* function:

```
var materials = new Array(); // array of materials
var matMagnet = new MagnetRadialMaterial;
matMagnet.pos = new QPointF(0.3, 10); // point (0.3, 10) belongs to subdomain
// the 'Magnet' is assigned to
matMagnet.center = new QPointF(0, 0); // center of magnetization
matMagnet.direction = Direction.Toward; // magnetization is toward the center
materials.push(matMagnet); // insert material assignment object
```

The *center* property of the material assignment object (*matMagnet.center* in the above example) defines the center of magnetization of the permanent magnet and the *direction* property defines the direction of magnetization. There are four possible magnetization directions for radially magnetized magnet: toward the center of magnetization, from the center of magnetization, clockwise and counterclockwise; see the example below:

```
matMagnet.direction = Direction.From; // radial, from the center
matMagnet.direction = Direction.Toward; // radial, toward the center
matMagnet.direction = Direction.CW; // radial, clockwise
matMagnet.direction = Direction.CCW; // radial, counterclockwise
```

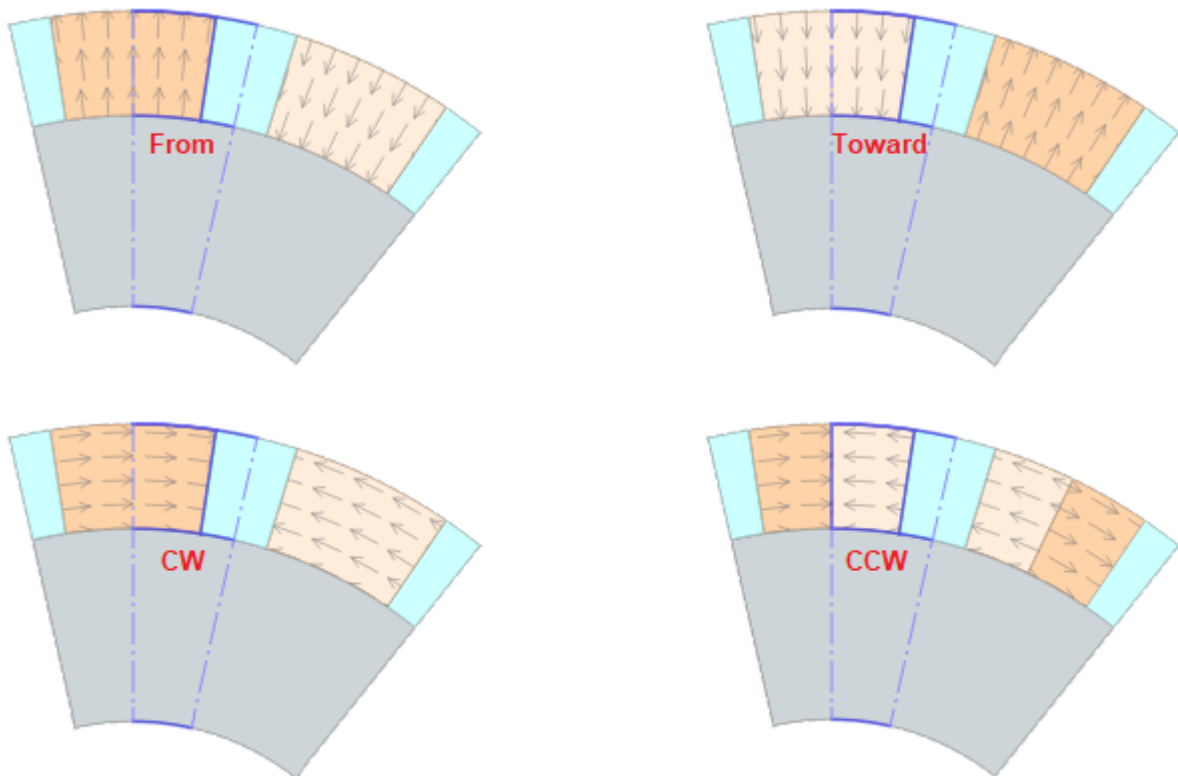


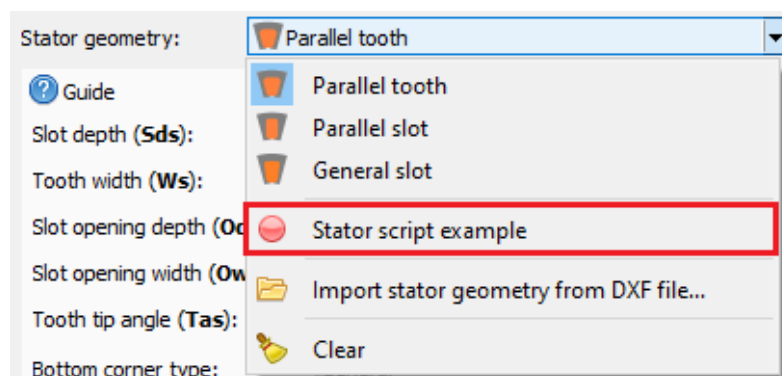
Figure 11.5. Radial magnetization examples with different directions (from center, toward center, clockwise, counterclockwise).

Similarly with the parallel magnetization, the radial magnetization direction is assigned through the script only for magnets in the right half of pole pitch and the magnetization for the rest of the geometry is automatically determined (Figure 11.5). As can be seen from Figure 11.5 the counterclockwise magnetization assigned in the script for the right half of pole pitch is mirrored for the left half of pole pitch (becomes clockwise), while three other magnetization directions are not mirrored.

11.6. Files *.rotors.json* and *.stators.json* and geometry template attributes.

In order for the geometry script to appear in the **Rotor geometry** or **Stator geometry** pup-up menu the script should be registered in the *.rotors.json* file (rotor geometry script) or *.stators.json* file (stator geometry script). File *.rotors.json* is located in *[MotorXP-PM installation directory]\bin\assets\scripts\Rotor* and file *.stators.json* is located in *[MotorXP-PM installation directory]\bin\assets\scripts\Stator*. The following lines of code added to the *.stators.json* file demonstrate how to add a new geometry script to the **Stator geometry** pup-up menu on the example of the *stator_script_example.js* geometry script considered in section 11.1:

```
"name" : "Stator script example",
"picture_inner" : "../../images/red.png",
"picture_outer" : "../../images/red.png",
"script" : "stator_script_example.js",
"applicability" : "innerRotor"
```



The “*name*” property defines the text of the item appearing in the **Stator geometry** pup-up menu corresponding to the *stator_script_example.js* geometry script. The “*picture_inner*” and “*picture_outer*” properties define the picture displayed to the left of the text in the **Stator geometry** pup-up menu as shown above; “*picture_inner*” corresponds to **Inner rotor** machine type and “*picture_outer*” corresponds to **Outer rotor**. The “*script*” property defines the name of the script file. The script should be in the same folder as the *.stators.json* file. The “*applicability*” property determines the machine type (**Inner rotor** or **Outer rotor**) which the script is applicable to. The “*applicability*” property can be one of the following: “innerRotor”, “outerRotor” and “anyRotor” defining the script applicable for both machine types. In the above example the geometry script is applicable only for the **Inner rotor** machine type so when the **Outer rotor** machine type is chosen the **Stator script example** item will not be visible in the **Stator geometry** pup-up menu.

11.6.1. Rotor geometry script example.

This section describes the example of the real rotor geometry script *halbach_array_rotor.js* used for the Halbach array rotor type. The full source code of the script can be found in *[MotorXP-PM installation directory]\bin\assets\scripts\Rotor\halbach_array_rotor.js*. This section mostly focuses on some practical aspects of using *sDraw* objects and methods described in section 11.3.4 to draw the geometry and on the user interface programming tips such as widgets visibility control, widgets grouping, etc. Similarly with the stator geometry script (see section 11.1), the structure of the rotor geometry script looks as following:

```
include('../sDraw.js');

function Rotor()
{
    // initialization code
};

Rotor.prototype.getGeometry = function () {
    // generate rotor geometry
};

Rotor.prototype.getMaterials = function() {
    // assign materials
};

var rotor = new Rotor();

function RotorUI(ui)
{
    // rotor widgets constructor
};

RotorUI.prototype.updateUI = function()
{
    // update widgets (control widgets visibility and etc.)
};

// call this function from c++ code
createRotorUI = function(ui)
{
    rotorUI = new RotorUI(ui);
    rotorUI.updateUI();
}
```

Figure 11.6 shows the user interface generated by the script. Depending on the user interface setup the geometry changes (inner or outer rotor, curved or flat magnet shape, presence, or absence of the retaining sleeve). The geometry considered in this section corresponds to the case shown in Figure 11.6, i.e. inner rotor, curved magnet shape and no retaining sleeve. The corresponding drawing explaining how the geometry algorithm works in this case is shown in Figure 11.7. The geometry algorithms are implemented in function *calcGeometry* (see source code of the *halbach_array_rotor.js* script).

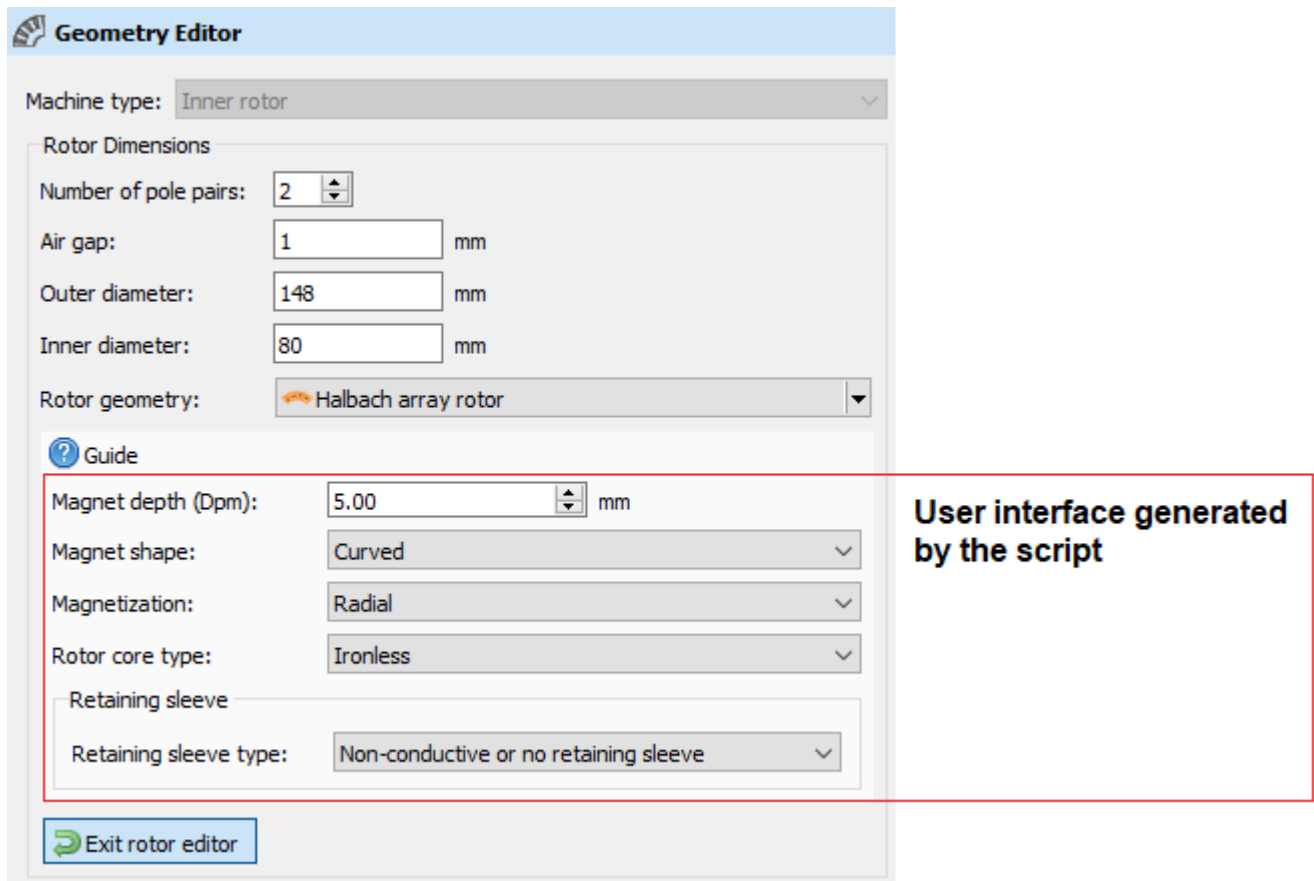


Figure 11.6. User interface generated by the *halbach_array_rotor.js* script.

According to Figure 11.7, the geometry of the rotor corresponding to the user interface setup shown in Figure 11.6 consists of three objects of type *Arc* (*arc_5_1*, *arc_6_2*, *arc_Rrinner*) and one object of type *Segment* (*segment_4_3*). These objects are then inserted into the *geomr* array which is returned by the *calcGeometry* function. Refer to sections 11.3.1 and 11.3.2 for more information on using the *Arc* and *Segment* geometry objects.

As it was already shown in section 11.3.4, using *sDraw* objects and methods is a convenient way of drawing complex geometries. In this example an intersection of the *line_0_3* line object and the *circle_Rpm* circle object is used to define the *point_4* point:

```
// PM inner radius circle
this.circle_Rpm = sDraw.createCircle(0, 0, this.Rrouter-this.Trs_real-this.Dpm_real);
this.line_0_3 = sDraw.createLineByPointAngle(0, 0, 90 - this.alfa / 2);
this.point_4 = sDraw.intersection(this.line_0_3, this.circle_Rpm, 'higher');
```

Similarly, the *point_3* point is defined as an intersection of the *line_0_3* line object and the *circle_Rsleeve* circle object. *sDraw* objects are not visible in **Geometry Editor** and shown in Figure 11.7 only for explanation purposes.

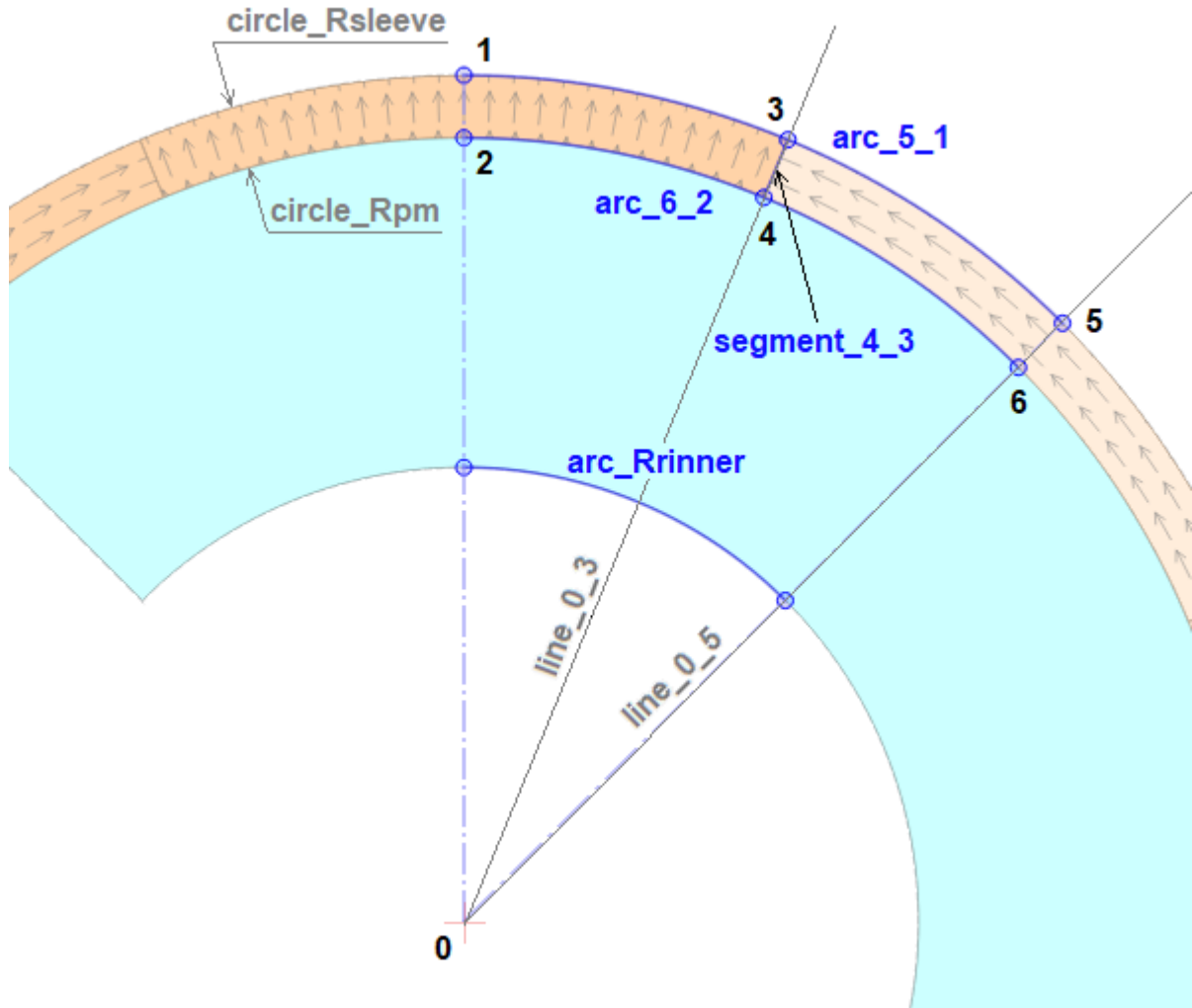


Figure 11.7. Explanation on the geometry generation for the *halbach_array_rotor.js* script.

The user interface of the *halbach_array_rotor.js* geometry script is generated by the rotor widget constructor function **RotorUI** and consists of several spin box widgets (*QDoubleSpinBox* type objects), several pop-up menu widgets (*QComboBox* type objects) as well as several text boxes (*QLabel* type objects). The **RotorUI** function works in the same way as the **StatorUI** function previously considered in the example of section 11.1. Check the *halbach_array_rotor.js* script source code and Qt documentation to see how the user interface is organized and for more information on the user interface objects.

Function **updateUI** can be used to control the visibility of widgets. In this example the **updateUI** function is used to control the visibility of the retaining sleeve widgets. When the **Outer rotor** machine type is chosen the retaining sleeve group of widgets is not visible. When the **Inner rotor** machine type is chosen the retaining sleeve group of widgets becomes visible and the visibility of the retaining sleeve thickness widgets will depend on whether the user selects to include the retaining sleeve into analysis or not. When **Non-conductive or no retaining sleeve** item is selected from the **Retaining sleeve type** pop-up menu the retaining sleeve thickness widgets are not visible as it is shown in Figure 11.6. Figure 11.8 shows the user interface when the **Conductive retaining sleeve** item is selected from the **Retaining sleeve type** pop-up

menu and the retaining sleeve thickness widgets are visible allowing the user to enter the thickness value. The source code of the *updateUI* function from the *halbach_array_rotor.js* script looks as following:

```
// update rotor widgets
RotorUI.prototype.updateUI = function()
{
    //console.log("rotorUI.updateUI");

    // Visibility of retaining sleeve widgets
    if (motor.isInnerRotor()) { // inner rotor
        var visibleRetainingSleeveType = true;
        var visibleTrs = rotor.RetainingSleeveType == "Conductive retaining sleeve";
    }
    else { // outer rotor
        var visibleRetainingSleeveType = false;
        var visibleTrs = false;
    }
    this.labelRetainingSleeveType.setVisible(visibleRetainingSleeveType);
    this.comboboxRetainingSleeveType.setVisible(visibleRetainingSleeveType);
    this.groupRetainingSleeve.setVisible(visibleRetainingSleeveType);
    this.labelTrs.setVisible(visibleTrs);
    this.editTrs.setVisible(visibleTrs);
    this.unitsTrs.setVisible(visibleTrs);
}
```

The screenshot shows a user interface for configuring a rotor. It includes several input fields and dropdown menus. A red rectangular box highlights the 'Retaining sleeve' section, which contains a dropdown menu for 'Retaining sleeve type' (currently set to 'Conductive retaining sleeve') and a numeric input for 'Retaining sleeve thickness (Trs)' (currently set to 1.0 mm). Above this section, there are other configuration options: 'Magnet depth (Dpm)' (5.00 mm), 'Magnet shape' (Curved), 'Magnetization' (Radial), and 'Rotor core type' (Ironless). At the bottom of the interface is a button labeled 'Exit rotor editor'.

Figure 11.8. Part of the user interface when the retaining sleeve thickness widgets are visible.

The visibility of the retaining sleeve type widgets is set up by the *visibleRetainingSleeveType* variable and visibility of the retaining sleeve thickness widgets is set up by the *visibleTrs* variable using the *setVisible* function. The *visibleRetainingSleeveType* and *visibleTrs* variable values change depending on the machine type (*Inner rotor* or *Outer rotor*) and on the selected **Retaining sleeve type** pop-up menu item (*Non-conductive or no retaining sleeve* or *Conductive retaining sleeve*) as it was described above. The widget groups allow to organize several widgets, so they are kept inside one frame with a title (title is optional). For example, in Figure 11.8 all widgets related to the retaining sleeve are organized in the group of widgets titled 'Retaining sleeve'. The widget groups are defined inside the *RotorUI* function

(inside the *StatorUI* function for stator geometry scripts); the fragment of the code of the *RotorUI* function creating the ‘Retaining sleeve’ widget group looks as following:

```
// create a group for retaining sleeve widgets:
this.groupRetainingSleeve = new QGroupBox("Retaining sleeve");
// Arranging group of widgets:
var layoutGroup_RetainingSleeve = new QGridLayout(this.groupRetainingSleeve);

layoutGroup_RetainingSleeve.addWidget(this.labelRetainingSleeveType, 0, 0);
layoutGroup_RetainingSleeve.addWidget(this.comboboxRetainingSleeveType, 0, 1, 1, 2);

layoutGroup_RetainingSleeve.addWidget(this.labelTrs, 1, 0);
layoutGroup_RetainingSleeve.addWidget(this.editTrs, 1, 1);
layoutGroup_RetainingSleeve.addWidget(this.unitsTrs, 1, 2);

this.layout.addWidget(this.groupRetainingSleeve, 4, 0, 1, 3); // insert group
```

The group contains and manages two widgets corresponding to the retaining sleeve type (*labelRatainingSleeveType* and *comboboxRatainingSleeveType*) and three widgets corresponding to the retaining sleeve thickness (*labelTrs*, *editTrs* and *unitsTrs*). The group has its own grid layout object *layoutGroup_RatainingSleeve* of *QGridLayout* type the widgets are added to using the *addWidget* function. Finally, the group is inserted into the main grid layout object *layout*.

12. MOTORXP-PM SETTINGS

MotorXP-PM settings are available from menu **File -> Settings** and shown in Figure 12.1.

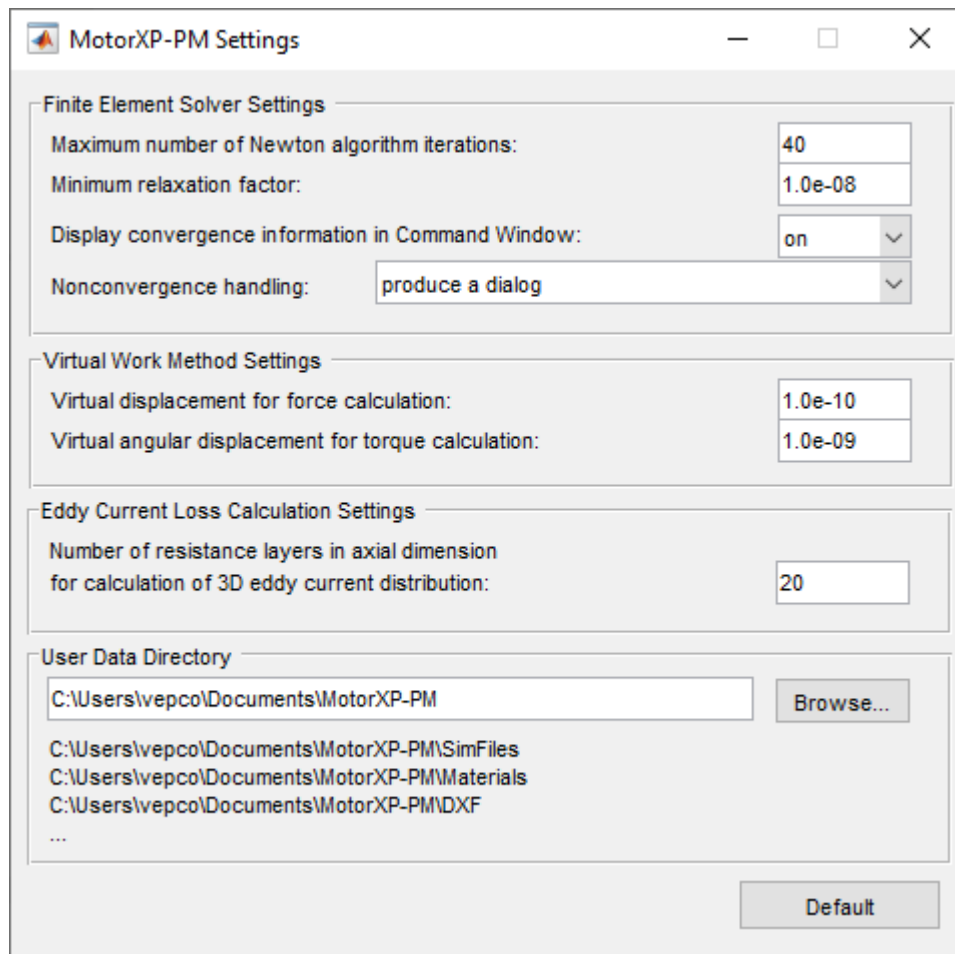


Figure 12.1. MotorXP-PM settings.

Finite Element Solver Settings.

Maximum number of Newton algorithm iterations – if the number of Newton algorithm iterations exceeds the value specified by this field it is said that the solution does not converge. The subsequent action will depend on the **Nonconvergence handling** field value.

Minimum relaxation factor – if the solution does not converge on the current Newton algorithm iteration, the nonlinear solver damps down the search step with relaxation factor $0 < \alpha < 1$. The relaxation factor iteratively decreases until convergence occurs. The smaller relaxation factor, the worse convergence. If the relaxation factor becomes less than **Minimum relaxation factor**, it is said that the solution does not converge. The subsequent action will depend on the **Nonconvergence handling** field value.

Display convergence information in Command Window – determines whether the convergence information (i.e. Newton algorithm iteration number, the maximum and average residual, and the relaxation factor) is displayed in the Command Window.

Nonconvergence handling – determines an action the program takes when the solution does not converge. If *ignore nonconvergence and continue simulation* is chosen, the simulation is continued with the accuracy reached regardless the value specified in the **Convergence tolerance** field of the main window; if *produce a dialog* is chosen, the dialog with the convergence information is provided to allow the user to decide either to continue or interrupt the simulation; if *interrupt simulation and report an error* is chosen, the simulation is interrupted with an error report.

Virtual Work Method Settings.

Fields of this panel define the virtual displacement for torque and force calculation with the virtual work method as discussed in section 2.3.

Eddy Current Loss Calculation Settings.

This panel influences the accuracy of the magnet loss calculation and eddy current loss calculation. By default, the number of the resistance layers is 20.

User Data Directory.

Specifies the directory where all the user files are stored such as projects and simulation data files (*SimFiles* folder), material library files (*Materials* folder), DXF-files (*DXF* folder) and winding layout files (*layoutfiles* folder).

The **Default** button resets all the MotorXP-PM settings shown in Figure 12.1 to its default values except **User Data Directory**.

APPENDIX

Appendix A. Power balance, discretization error and accuracy of the results.

If the power balance is satisfied it means that the input apparent power delivered by all current and voltage sources equals to the consumed apparent power:

$$P_{input} = P_{cons} \quad (13.1)$$

The input power is calculated from voltages and currents of all power sources:

$$P_{input} = \sum_{n=1}^m i_n v_n \quad (13.2)$$

The consumed power is defined as follows:

$$P_{cons} = P_s + P_{mech} + \frac{\Delta W_{mf}}{\Delta t}, \quad (13.3)$$

where P_s – apparent power (real and reactive) of the stator electrical circuit, P_{mech} – mechanical power on the shaft, $\Delta W_{mf}/\Delta t$ – magnetic field energy time derivative.

The discretization error of the simulation is a percentage difference between the input power and consumed power. The discretization error is mostly influenced by the simulation time step and can be used to assess the accuracy of the simulation results.

Appendix B. Magnetostatic Analysis results.

In the Magnetostatic Analysis the instantaneous values of the input power are calculated as the following:

$$p_{input} = i_a v_a + i_b v_b + i_c v_c \quad (13.4)$$

And the input power value displayed in the **Results** panel of the Magnetostatic Analysis is the average of the instantaneous input power defined by the Exp. (13.4). Note that the iron loss, magnet loss and other eddy current loss are calculated during the post processing and not included in the input power.

The efficiency is calculated as the following:

$$Efficiency = \frac{P_{mech}}{P_{mech} + P_s + P_{iron} + P_{magnet} + P_{other}} \quad (13.5)$$

where P_{mech} – mechanical power on the shaft, P_s – stator winding loss, P_{iron} – total iron core loss, P_{magnet} – magnet eddy current loss, P_{other} – other eddy current loss.

The power factor is calculated as the following:

$$PowerFactor = \frac{P_{mech} + P_s}{3 \cdot I_{RMS} \cdot V_{RMS}} \quad (13.6)$$

where I_{RMS} and V_{RMS} – RMS values of the phase current and phase voltage.

The reason why the input power is not used for the efficiency and power factor calculation is that the Exp. (13.5) and (13.6) are more stable to the discretization error as the simulation time step increases.

The discretization error shows how the input power matches with the mechanical power and stator winding loss and defined as follows:

$$Error = 100 \cdot abs \left[\frac{P_{input} - (P_{mech} + P_s)}{P_{input}} \right] \quad (13.7)$$

The discretization error decreases as the simulation time step decreases and the mismatch between the input power, mechanical power, losses, efficiency and power factor also decrease.

Appendix C. Out of memory errors handling.

“Out of memory” errors can occur when the size of the mesh is too large. According to the MATLAB help documentation, to avoid “out of memory” errors when using MotorXP-PM, you can increase the size of the swap file or add more memory to the system. Restarting MATLAB when an “out of memory” error occurs may also help in some cases.

PART II

MotorXP-PM MATLAB scripting API

Overview

MotorXP-PM MATLAB scripting API allows changing the geometry, winding and mesh parameters of the radial flux machine design as well as simulation settings and running magnetostatic finite element (FE) simulations directly from the MATLAB scripts and functions without using the user interfaces. MotorXP-PM MATLAB scripting API enables parallel processing, i.e. running several simulations at the same time fully utilizing the capabilities of the modern multi-core and multi-CPU computers. MotorXP-PM MATLAB scripting API does not require any additional MATLAB toolboxes.

Note that MotorXP-PM scripting API requires MATLAB Runtime R2020b (9.9). Download and install it from the MathWorks official website <https://www.mathworks.com/products/compiler/matlab-runtime.html>.

1. Scripting API initialization

[initMXPScriptingAPI](#) – function initializing the MotorXP-PM MATLAB scripting API environment, i.e., adding all the necessary paths and loading necessary libraries.

2. Functions for working with mxp-files

[openMXP](#) – function that opens the design from a mxp-file creating the [motorProps](#) and [settingsMagnetostatic](#) structures.

[saveDesign2MXPfile](#) – function that saves the design defined by the [motorProps](#) and [settingsMagnetostatic](#) structures to a mxp-file.

3. Functions for setting and getting parameters

[setParampm](#) – function used to write geometry, winding and mesh data parameters to the [motorProps](#) structure as well as simulation settings to the [settingsMagnetostatic](#) structure.

[getParampm](#) – function used to retrieve geometry, winding and mesh data parameters from the [motorProps](#) structure as well as simulation settings from the [settingsMagnetostatic](#) structure.

4. Serial processing functions

4.1. Assembling designs in series

[assembleMXP](#) – function preparing all the necessary structures for a specific design for running simulations.

4.2. Running magnetostatic FE simulations in series

[runMStimestepping](#) – run a magnetostatic FE simulation for a specific design.

5. Parallel processing functions

5.1. Assembling designs in parallel

[initAssembleMXP_par](#) – initializes assembling multiple designs in parallel.

[runAssembleMXP_par](#) – starts assembling multiple designs in parallel.

[getAssembleMXP_par](#) – retrieves structures for assembled designs.

5.2. Running magnetostatic FE simulations in parallel

[initMStimestepping_par](#) – initializes running multiple magnetostatic FE simulations in parallel.

[runMStimestepping_par](#) – starts running multiple magnetostatic FE simulations in parallel.

[getMStimestepping_par](#) – retrieves results for finished magnetostatic FE simulations.

6. Functions for automatic optimization workflow development

[runTSEMOptimizationpm](#) – function that executes the optimization algorithm.

[evalDesigns](#) – example of the design function for evaluating generated designs.

[plotPareto](#) – displays the optimization algorithm results as the Pareto plot.

[plotParetoFromFile](#) – displays the Pareto plot for the optimization algorithm results previously saved to a mat-file.

1. Scripting API initialization

initMXPscriptingAPI

The *initMXPscriptingAPI* function initializes the necessary path folders that will be used during the analysis and load necessary libraries. It is recommended to start the script calling this function.

Syntax

```
initMXPscriptingAPI
```

This function does not have inputs and outputs.

2. Functions for working with mxp-files

2.1. *openMXP* function

2.1.1. Syntax

```
[motorProps, settingsMagnetostatic] = openMXP(file)
```

2.1.2. Description

The *openMXP* function is used to create two fundamental structures: [*motorProps*](#) and [*settingsMagnetostatic*](#). To create these structures, initial design data stored in a mxp-file are needed. Once *openMXP* is called, the script can access the model data and create the necessary structures for the simulation.

2.1.3. Input and outputs

To input the initial mxp-file for the analysis, the user should provide its path. When the design is saved using the MotorXP-PM user interface, a mxp-file is automatically created. To simplify the process of providing a path, the *getpmpath* function can be used to return the path of the mxp-file. This allows for relative paths to be used, as shown in the example:

```
file = getpmpath('SimFiles\Prius\Prius.mxp');
```

The outputs of *openMXP* are two structures: *motorProps* and *settingsMagnetostatic*, as described below.

2.1.4. *motorProps* structure

This is a structure that contains the parameters of the motor model, such as the geometry of the stator and rotor, winding, materials, and mesh. All the parameters defining the design are stored within this structure. The detailed description for all the fields of *motorProps* is provided in [section 3.2](#).

2.1.5. settingsMagnetostatic structure

This is a structure that contains the magnetostatic FE simulation settings such as solver type, convergence tolerance, mechanical speed, stator current, etc. The detailed description for all the fields of *settingsMagnetostatic* is provided in [section 3.3](#).

2.2. *saveDesign2MXPfile* function

2.2.1. Syntax

```
saveDesign2MXPfile(initdesignfile, motorProps, MXPfile)
```

2.2.2. Description

The *saveDesign2MXPfile* function is used to save the specific design to a mxp-file so it can then be opened using MotorXP-PM user interface for detailed analysis.

2.2.3. Input and outputs

initdesignfile – initial mxp-file storing the basic parameters of the design.

motorProps – [motorProps](#) structure containing desired parameters of the design which will overwrite the basic parameters from *initdesignfile*.

MXPfile – path and mxp-file name for the created new mxp-file.

3. Functions for setting and getting parameters

3.1. *setParampm* function

3.1.1. Syntax

```
paramStruct = setParampm(paramStruct,paramName,paramValue)
```

3.1.2. Description

The *setParampm* function is used to modify values in structures [motorProps](#) and [settingsMagnetostatic](#).

3.1.3. Inputs

The input *paramStruct* is either [motorProps](#) or [settingsMagnetostatic](#) structure. If you try to use another structure, an error message will be displayed: “*Incorrect input structure, only motorProps and settingsMagnetostatics are valid*”.

The possible values for inputs *paramName* and *paramValue* are defined by json-files as described in sections [3.1.5](#) and [3.1.6](#) and in Tables 1-7.

3.1.4. Output

The output structure *paramStruct* is the same as the input structure *paramStruct* with the modified field value defined by *paramName*.

3.1.5. *Dependencies on json-files*

The *setParampm* function depends on some json-files which define the possible *paramName* values for the [motorProps](#) and [settingsMagnetostatic](#) structures. There are several default json-files as listed below.

In `\bin\assets\scripts`:

- *motorProps.json* – general [motorProps](#) parameters.
- *settingsMagnetostatic.json* – general [settingsMagnetostatic](#) parameters.

In `\bin\assets\scripts\Rotor`:

- *bread_loaf.json* – geometry parameters for the **Bread Loaf** rotor geometry template.
- *halbach_array_rotor.json* – geometry parameters for the **Halbach array** rotor geometry template.
- *spoke.json* – geometry parameters for the **Spoke** rotor geometry template.
- *straight_buried.json* – geometry parameters for the **Straight Buried** rotor geometry template.
- *sufacemounted_parallel.json* – geometry parameters for the **Surface Mounted Parallel** rotor geometry template.
- *Surfacemounted_radial.json* – geometry parameters for the **Surface Mounted Radial** rotor geometry template.
- *vshape.json* – geometry parameters for the **V-Shape** rotor geometry template.

In `\bin\assets\scripts\Stator`:

- *Parallel_tooth.json* – geometry parameters for the **Parallel tooth** stator geometry template.
- *Parallel_slot.json* – geometry parameters for the **Parallel slot** stator geometry template.
- *General_slot.json* – geometry parameters for the **General slot** stator geometry template.

3.1.6. *Stator and rotor geometry template json-files*

As can be noticed from the json-file names listed above, each stator and rotor geometry template js-file have the corresponding json-file with the same name. This is how the *setParampm* and *getParampm* functions recognize which geometry parameter defined by *paramName* corresponds to each field name in *motorProps.stator.scriptProps* (for stator) or in *motorProps.rotor.scriptProps* (for rotor). For example, in file *Parallel_slot.json* there are the follow lines:

```
{
  ...
  "Slot width": "stator.scriptProps.Ws",
```

"Slot opening depth": "stator.scriptProps.Ods",

...

}

For example, to set the stator diameter to 10, use the following

```
motorProps = setParampm(motorProps, 'Stator outer diameter', 10);
```

3.2. *motorProps* structure

3.2.1. Main *paramName* parameter description and *paramValue* values

The possible *paramName* and their respective description and values are shown in Table 1.

Table 1. The *paramName* and *paramValue* variable values used for the *motorProps* structure.

Name (<i>paramName</i>)	Description	Values (<i>paramValue</i>)
Stator inner diameter	Corresponds to the Stator inner diameter (mm) field of Geometry Editor (Stator).	Number > 0
Stator outer diameter	Corresponds to the Stator outer diameter (mm) field of Geometry Editor (Stator).	Number > 0
Number of slots	Corresponds to the Number of slots field of Geometry Editor (Stator).	Number >= 3
Rotor inner diameter	Corresponds to the Rotor inner diameter (mm) field of Geometry Editor (Rotor).	Number > 0
Number of pole pairs	Corresponds to the Number of pore pairs field of Geometry Editor (Rotor).	Number >= 1
Rotor outer diameter	Corresponds to the Rotor outer diameter (mm) field of Geometry Editor (Rotor).	Number > 0
End winding axial overhang	Corresponds to the End winding axial overhang field of Winding Editor .	'Auto' or Number > 0
End winding inductance	Leakage inductance of the stator winding end-turns per phase; corresponds to the End winding inductance field of Winding Editor .	'Auto' or Number >= 0

	If set to 'Auto', the value will be calculated automatically based on the winding configuration and machine dimensions.	
Number of parallel paths	Number of parallel paths of the stator winding per phase; corresponds to the Number of parallel paths field of Winding Editor .	Number ≥ 1
Phase resistance	Active DC resistance of the stator winding per phase for one layout; corresponds to the Phase resistance field of Winding Editor . If set to 'Auto', the value will be calculated automatically based on the winding configuration, machine dimensions and winding temperature.	'Auto' or Number ≥ 0
Coil span	Corresponds to the Coil span field of Winding Editor . If set to 'Auto', the value will be calculated automatically.	'Auto' or $1 \leq \text{Number} \leq \text{max_value}$
Coil fill factor	Stator coil fill factor; corresponds to the Coil fill factor field of Winding Editor . This parameter is required only if 'Wire size method' is set to 'Fill factor'.	$0 < \text{Number} \leq 1$
Winding layers orientation	Determined either winding layers are oriented horizontally or vertically inside the slot; corresponds to the Winding layers orientation field of Winding Editor .	'Upper / Lower' 'Left / Right'
Winding layout method	Corresponds to Winding layout method field of Winding Editor .	'Automatic' 'Manual' 'From file'
Number of strands in hand	Corresponds to the Number of strands in hand field of Winding Editor .	Number ≥ 1

Winding layers	Determines the number of winding layers per slot; corresponds to the Winding layers field of Winding Editor .	'Single layer' 'Double layer' or $3 \leq \text{Number} \leq 20$
Number of turns	Specifies the number of turns per one coil, the same as the number of turns per one winding layer; corresponds to the Number of turns field of Winding Editor .	Number ≥ 1
Winding connection	Determines the stator winding connection; corresponds to the Winding connection field of Winding editor .	'StarConnection' 'DeltaConnection'
Strand diameter	Corresponds the Strand diameter field of Winding editor . Strand diameter, by default, is calculated based on coil fill factor, number of turns and number of strands in hand. This parameter is required only if 'Wire size method' is set to 'Wire diameter'.	Number > 0
Wire gauge number	Wire diameter according to the Standard wire gauge (SWG) or American wire gauge (AWG) standard; corresponds the Wire gauge number field of Winding Editor . This parameter is required only if 'Wire size method' is set to 'AWG' or 'SWG'.	AWG Table value, e.g., '21 AWG', or SWG Table value, e.g., '21 SWG',
Wire size method	Wire size method specifies how the Strand diameter value is determined; corresponds the Wire size method field of Winding editor .	'Wire diameter' 'AWG' 'SWG' 'Fill factor'
Air gap	Air gap length; corresponds to the Air gap field of Geometry Editor .	Number > 0

Air gap mesh quality	Air gap mesh quality modifies the quality of the mesh in the air gap region; corresponds the Air gap mesh quality field of Mesh Editor .	'Low' 'Medium' 'High'
Maximum triangle side	The upper bound of length of the longest mesh triangle side; corresponds the Maximum triangle side (mm) field of Mesh Editor . If set to 'Auto', the value will be calculated automatically.	'Auto' or Number > 0.1
Number of layers in air gap	Number of mesh layers in the air gap; corresponds to the Number of layers in air gap field of Mesh Editor .	3, 5, 7 or 9
Number of slices	Number of machine's cross-sections along the shaft used by the multi-slice FEM; corresponds to the Number of slices field of Mesh Editor .	Integer number from 1 to 20
Boundary conditions	Periodic/antiperiodic boundary conditions; corresponds to the Boundary conditions field of Mesh Editor . If set to 'Auto', best possible boundary conditions will be used.	'Auto' 'None' 'Periodic' 'Antiperiodic'
Lamination stack length	Corresponds to the Lamination stack length field of Geometry Editor .	Number > 0
Stator skew angle	Corresponds to the Stator skew angle field of Geometry Editor .	Number >= 0
Rotor skew angle	Corresponds to the Rotor skew angle field of Geometry Editor .	Number >= 0
Number of magnet segments	Corresponds to the Number of magnet segments field of Geometry Editor .	Number >= 1

3.2.2. Examples.

Example to change the air gap length passing 'Air gap' as **paramName** and '1.2' as **paramValue**:

```
motorProps = setParampm(motorProps, 'Air gap', 1.2);
```

Example to change the number of mesh layers in air gap passing 'Number of layers in air gap' as **paramName** and '9' as **paramValue**:

```
motorProps = setParampm(motorProps, 'Number of layers in air gap', 9);
```

Example to change the number of slots passing 'Number of slots' as **paramName** and 48 as **paramValue**:

```
motorProps = setParampm(motorProps, 'Number of slots', 48);
```

Example to change phase resistance passing 'Phase resistance' as **paramName** and 0.346 as **paramValue**:

```
motorProps = setParampm(motorProps, 'Phase resistance', 0.346);
```

3.2.3. Some limitations for paramValue imposed by the machine topology.

If 'Winding layout Method' is set to 'From file' and 'Winding number of layers' is greater than 2, then 'End winding inductance' and 'Phase Resistance' cannot be set as 'Auto' and 'End winding axial overhang' will not be considered (can be of any value).

If 'Winding layers' is greater than two the 'Winding layout method' can be only 'From file'.

Some rotor and stator configurations are not possible for some topologies with a few examples given below.

An error message will be displayed when using 'Winding Layout Method' as 'From file' and trying to change 'End winding inductance' to 'Auto'

```
motorProps = setParampm(motorProps, 'End winding inductance', 'Auto');
```

An error message will be displayed: “The parameter – End winding inductance – cannot be set as Auto.”.

3.2.4. The paramName and paramValue variable values for default rotor and stator geometries.

There are three default stator geometries (*Parallel tooth*, *Parallel slot* and *General slot*) and seven default rotor geometries (*Surface mounted radial*, *Surface mounted parallel*, *Halbach array rotor*, *Bread loaf*, *Straight buried*, *Spoke*, *V-shaped magnet*). The possible **paramName** and **paramValue** variable values of the **setParampm** function for the default rotor and stator geometries are listed in Tables 2-11.

Table 2. Stator geometry *Parallel tooth*: *paramName* and *paramValue* variable values.

<i>paramName</i>	Values (<i>paramValue</i>)
Slot depth	Number > 0
Tooth width	Number > 0
Slot opening depth	Number >= 0
Slot opening width	Number >= 0
Tooth tip angle	Number >= 0
Slot bottom corner type	'General' 'Round'
Bottom corner radius	Number >= 0
Top corner radius	Number >= 0
Tooth edge chamfer depth	Number >= 0
Tooth edge chamfer ratio	Number >= 0
Tooth edge chamfer angle	Number >= 0
Slot insulation thickness	Number >= 0
Between layers insulation thickness	Number >= 0

Table 3. Stator geometry *Parallel slot*: *paramName* and *paramValue* variable values.

<i>paramName</i>	Values (<i>paramValue</i>)
Slot depth	Number > 0
Slot width	Number > 0
Slot opening depth	Number >= 0
Slot opening width	Number >= 0

Tooth tip angle	Number ≥ 0
Slot bottom corner type	'General' 'Round'
Bottom corner radius	Number ≥ 0
Top corner radius	Number ≥ 0
Tooth edge chamfer depth	Number ≥ 0
Tooth edge chamfer ratio	Number ≥ 0
Tooth edge chamfer angle	Number ≥ 0
Slot insulation thickness	Number ≥ 0
Between layers insulation thickness	Number ≥ 0

Table 4. Stator geometry *General slot*: *paramName* and *paramValue* variable values.

<i>paramName</i>	Values (<i>paramValue</i>)
Slot depth	Number > 0
Slot bottom width	Number > 0
Slot top width	Number > 0
Slot opening depth	Number ≥ 0
Slot opening width	Number ≥ 0
Tooth tip angle	Number ≥ 0
Slot bottom corner type	'General' 'Round'
Bottom corner radius	Number ≥ 0
Top corner radius	Number ≥ 0
Tooth edge chamfer depth	Number ≥ 0

Tooth edge chamfer ratio	Number ≥ 0
Tooth edge chamfer angle	Number ≥ 0
Slot insulation thickness	Number ≥ 0
Between layers insulation thickness	Number ≥ 0

Table 5. Rotor geometry *Surface mounted radial*: *paramName* and *paramValue* variable values.

<i>paramName</i>	Values (<i>paramValue</i>)
Magnet depth	Number > 0
Magnet angle	Number > 0
Magnet filet radius	Number ≥ 0
Magnet inset depth	Number ≥ 0
Magnet magnetization	'Radial' 'Parallel'
Retaining sleeve type	'Non-conductive or no retaining sleeve' 'Conductive retaining sleeve'
Retaining sleeve thickness	Number ≥ 0

Table 6. Rotor geometry *Surface mounted parallel*: *paramName* and *paramValue* variable values.

<i>paramName</i>	Values (<i>paramValue</i>)
Magnet depth	Number > 0
Magnet width	Number > 0
Magnet filet radius	Number ≥ 0
Magnet inset depth	Number ≥ 0

Magnet magnetization	'Radial' 'Parallel'
Retaining sleeve type	'Non-conductive or no retaining sleeve' 'Conductive retaining sleeve'
Retaining sleeve thickness	Number ≥ 0

Table 7. Rotor geometry *Halbach array rotor*: *paramName* and *paramValue* variable values.

<i>paramName</i>	Values (<i>paramValue</i>)
Magnet depth	Number > 0
Magnet shape	'Curved' 'Flat'
Magnet magnetization	'Radial' 'Parallel'
Rotor core type	'Ironless' 'Iron'
Retaining sleeve type	'Non-conductive or no retaining sleeve' 'Conductive retaining sleeve'
Retaining sleeve thickness	Number ≥ 0

Table 8. Rotor geometry *Bread loaf*: *paramName* and *paramValue* variable values.

<i>paramName</i>	Values (<i>paramValue</i>)
Magnet depth	Number > 0
Magnet width	Number > 0
Magnet radius	Number > 0
Magnet fillet radius	Number >= 0
Magnet inset depth	Number >= 0

Table 9. Rotor geometry *Straight buried*: *paramName* and *paramValue* variable values.

<i>paramName</i>	Values (<i>paramValue</i>)
Magnet depth	Number > 0
Magnet width	Number > 0
Barrier width	Number >= 0
Magnet inset depth	Number > 0

Table 10. Rotor geometry *Spoke*: *paramName* and *paramValue* variable values.

<i>paramName</i>	Values (<i>paramValue</i>)
Magnet width	Number > 0
Magnet thickness	Number > 0
Magnet inset depth	Number > 0
Bridge depth	Number > 0
Bridge slit width	Number >= 0

Table 11. Rotor geometry *V-shaped magnet*: *paramName* and *paramValue* variable values.

<i>paramName</i>	Values (<i>paramValue</i>)
Magnet width	Number > 0
Magnet length	Number > 0
Magnet inset depth	Number > 0
Distance between magnets	Number >= 0
Magnet angle	Number >= 0
Outer barrier upper length	Number >= 0
Outer barrier lower length	Number >= 0
Outer barrier upper fillet radius	Number >= 0
Outer barrier lower fillet radius	Number >= 0
Inner barrier upper length	Number >= 0
Inner barrier lower length	Number >= 0
Inner barrier upper fillet radius	Number >= 0
Inner barrier lower fillet radius	Number >= 0
Groove radius	Number >= 0
Groove depth	Number >= 0
Groove position angle	Number >= 0

3.3. settingsMagnetostatic structure

This structure contains settings which are used for running magnetostatic FE simulations. The settings are the same as displayed in the MotorXP-PM main window for Magnetostatic FE Analysis, see Figure 1.

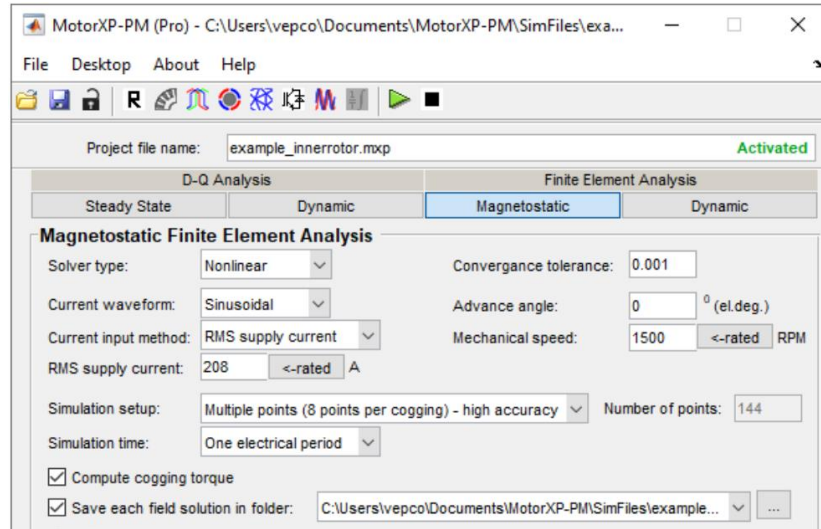


Figure 1. MotorXP-PM main window with Magnetostatic FEA settings.

3.3.1. Main paramName parameter description and paramValue values

The possible *paramName* and their respective description and values for the *settingsMagnetostatic* structure are shown in Table 12. A more detailed description of the magnetostatic FE simulation settings can be found in [section 5](#) of Part I of this user manual.

Table 12: The *paramName* and *paramValue* variable values used for the *settingsMagnetostatic* structure.

<i>paramName</i>	Description	Values (<i>paramValue</i>)
Solver type	Corresponds to the Solver type pup-up menu of the main window, see Figure 1.	'Nonlinear' 'Linear'
Convergence tolerance	Corresponds to the Convergence tolerance field of the main window, see Figure 1.	Number > 0
Number of points	Defines the number of time steps (rotor positions) per one electrical period or per selected time interval.	Number >= 1

	This can be calculated with another external function <code>setNumberOfPoints('High accuracy',nPolePairs,Ns)</code> . See section 3.3.2 .	
Mechanical speed	The mechanical speed of the rotor in RPM; corresponds to the Mechanical Speed field of the main window, see Figure 1.	Number =0 Units: RPM
Advance angle	Corresponds to the Advance angle field of the main window, see Figure 1.	Number >= 0 Units: electrical degrees
Compute cogging torque	Enables calculation of the cogging torque, corresponds to the Compute cogging torque checkbox value of the main window, see Figure 1. Note that calculation of the cogging torque slows down the simulation.	0 1
Save each step solution	Corresponds to the Save each step solution checkbox value of the main window, see Figure 1.	0 1
Solution folder	Folder to store data-files when 'Save each field solution' is set 1.	Directory path e.g., 'C:\example'
Simulation time	Predefined simulation time interval. Simulation can be run either for one electrical period starting from zero or for the time interval specified by 'Time from' and 'Time to' (see below).	'One electrical period' 'Define time interval'
Time from	Simulation start time when 'Simulation time' is set to 'Define time interval'.	Number > 0 Units: seconds
Time to	Simulation stop time when 'Simulation time' is set to 'Define time interval'.	Number > 0 Units: seconds

Current waveform	Defines the stator current waveform used for the simulation.	'Sinusoidal' 'Trapezoidal'
RMS supply current Peak supply current RMS current density RMS phase current DC supply current	<p>The Stator current input method which also depends on the current waveform. For the sinusoidal current waveform, the following current input methods are available: <i>RMS supply current</i>, <i>Peak supply current</i>, and <i>RMS current density</i>.</p> <p>For the trapezoidal current waveform, the following current input methods are available: <i>RMS phase current</i>, <i>DC supply current</i>, and <i>RMS current density</i>.</p>	<p>Number ≥ 0</p> <p>Units:</p> <p>for current: Amperes</p> <p>for current density: Amperes per square millimeter</p>

3.3.2. Examples

Example to change the solver type passing 'Solver type' as *paramName* and 'Linear' as *paramValue*:

```
settingsMagnetostatic = setParampm(settingsMagnetostatic, 'Solver
type', 'Linear');
```

Example to change the rotor speed passing 'Mechanical speed' as *paramName* and '1500' as *paramValue*:

```
settingsMagnetostatic = setParampm(settingsMagnetostatic, 'Mechanical
speed', 1500);
```

There are some particularities to change the values of the parameters related to the current input method and current density. Two fields are assigned at the same time, which are shown in the example below, where 'RMS supply current' is a current input method and 200 is the RMS supply current value. The current waveform must be separately assigned.

```
settingsMagnetostatic = setParampm(settingsMagnetostatic, 'Current
waveform', 'Sinusoidal');

settingsMagnetostatic = setParampm(settingsMagnetostatic, 'RMS supply
current', 200);
```

Another case is shown below when a current input method is wrongly used:

```
settingsMagnetostatic = setParampm(settingsMagnetostatic, 'Current
                                waveform', 'Trapezoidal');

settingsMagnetostatic = setParampm(settingsMagnetostatic, 'RMS supply
                                current', 200);
```

In this case an error message will be displayed “*The parameter – RMS supply current – cannot be assigned with this current waveform.*” since the 'RMS supply current' option is not available for the trapezoidal current waveform.

Example to change the number of points passing 'Number of points' as *paramName* and the result of the *setNumberOfPoints* function as *paramValue*:

```
settingsMagnetostatic = setParampm(settingsMagnetostatic, 'Number of
                                points', setNumberOfPoints('High accuracy', nPolePairs, Ns));
```

Note that the *setNumberOfPoints* function returns the number of points related to desired accuracy. The first input variable of *setNumberOfPoints* defines the number of points (time-steps) per one period of cogging torque which determines the accuracy of the simulation results. The following options are available:

- 'High accuracy' – 8 time-steps per one period of cogging torque
- 'Medium accuracy' – 4 time-steps per one period of cogging torque
- 'Low accuracy' – 2 time-steps per one period of cogging torque

The number of points is then calculated using the number of pole pairs (nPolePairs) and number of stator slots (Ns) input variables.

If 'Number of points' is set to 1, it will correspond to the *Single point (FEA + D-Q based)* – fastest item of the **Simulation setup** pop-up menu of the main window, see Figure 1. Refer to [section 5](#) of Part I of this user manual for more details.

3.4. *getParampm function*

3.4.1. *Syntax*

```
paramValue = getParampm(paramStruct,paramName)
```

3.4.2. *Description*

The *getParampm* function is used to query the values from the structures [motorProps](#) and [settingsMagnetostatic](#).

3.4.3. Inputs

The input ***paramStruct*** is either [motorProps](#) or [settingsMagnetostatic](#) structure. If you try to use another structure instead of [motorProps](#) or [settingsMagnetostatic](#), an error message will be displayed: “*Incorrect input structure, only motorProps and settingsMagnetostatic are valid*”.

The possible values for input ***paramName*** are the same as for ***setParampm*** and listed in Tables 1-12. Additionally, 'all' can be used to display all the values from [motorProps](#) or [settingsMagnetostatic](#), as shown below:

```
getParampm(motorProps, 'all')
getParampm(settingsMagnetostatic, 'all')
```

3.4.4. Output

The output ***paramValue*** is the corresponding value from [motorProps](#) or [settingsMagnetostatic](#) as listed in Tables 1-12.

3.4.5. Dependencies on json-files

The ***getParampm*** function depends on the same json-files as ***setParampm***. Refer to [section 3](#) for more details.

3.4.6. Examples

Example to query the value of 'Air gap mesh quality' from the [motorProps](#) structure:

```
value = getParampm(motorProps, 'Air gap mesh quality')
```

Example to query the value of 'Rotor outer diameter' from the [motorProps](#) structure:

```
value = getParampm(motorProps, 'Rotor outer diameter')
```

Example to query the value of 'Convergence tolerance' from the [settingsMagnetostatic](#) structure:

```
value = getParampm(settingsMagnetostatic, 'Convergence tolerance')
```

4. Serial processing functions

4.1. Assembling designs in series using the *assembleMXP* function

4.1.1. *Syntax*

```
[Geometry, Windings, Mesh, Materials, SubdomainProperty, Error,
motorProps_output, Weight] = assembleMXP(initMXPfile,motorProps)
```

4.1.2. *Description*

The *assembleMXP* function is used to prepare the design structures necessary for running magnetostatic FE simulations. This process is called design assembling. During assembling the program creates the geometry of the machine, builds the finite element mesh, assigns subdomain properties, etc.

4.1.3. *Inputs*

There are two inputs, [*motorProps*](#) structure and the file path of the initial mxp-file project.

4.1.4. *Outputs*

The outputs are listed below with a brief description:

Geometry – structure containing information about the machine’s geometry and topology.

Windings – structure with windings information, such as the number of turns, stator connection, number of pole pairs, etc. There are the following field which can be useful:

- **Winding.Rs** – active DC resistance of the stator winding per phase considering the winding temperature specified in *Materials.Tsw*.
- **Winding.Lsew** – the leakage inductance of the stator winding end turns per phase.

Mesh – structure with mesh data.

Materials – structure with materials data.

SubdomainProperty – structure containing information about subdomain properties.

Error – if the design failed assembling, the error message will be stored in this structure.

motorProps_output – output [*motorProps*](#) structure. In some cases, there can be some inconsistencies and improperly assigned values in the input [*motorProps*](#) structure. So, the *assembleMXP* function corrects it and returns the corrected values in the *motorProps_output* structure. The user can compare the corresponding fields of the [*motorProps*](#) and *motorProps_output* structures to make sure the resulting design looks as intended.

Weight - return the mass in kilograms for each motor active material. This parameter has 5 fields for each active material of the machine:

- **Weight.statoriron** – stator iron core mass.
- **Weight.rotoriron** – rotor iron core mass.
- **Weight.statorwinding** – stator winding mass.
- **Weight.magnet** – magnet mass.
- **Weight.conductor** – other conductive parts (rotor retaining sleeve, etc.,) mass.

4.2. Running magnetostatic FE simulations in series using the *runMStimestepping* function

4.2.1. Syntax

```
[Results, Timesteppingdata, field, ~, ~, Private] =
runMStimestepping(Private, Geometry, Mesh, Windings,
settingsMagnetostatic, Materials, SubdomainProperty,
SwitchDutyCycle)
```

4.2.2. Description

This function runs the magnetostatic finite element simulation.

4.2.3. Inputs

Private – structure with internal data of the FE solver. If **Private** is empty, it will be calculated inside the *runMStimestepping* function before the simulation is started. **Private** depends on the design parameters (i.e. input arguments **Geometry**, **Windings**, **Mesh**, **Materials**, **SubdomainProperty**), but does not depend on the simulation settings (i.e. input arguments **settingsMagnetostatic** and **SwitchDutyCycle**). It means that **Private** should be recalculated (i.e. should be empty while calling *runMStimestepping*) for each new design. If only simulation settings are changed (current, advance angle, speed, number of rotor positions, convergence tolerance or any other parameter in **settingsMagnetostatic**), there is no need to recalculate **Private** – the one can use **Private** calculated with the previous call of *runMStimestepping* – refer to output argument **Private** below.

Geometry, **Windings**, **Mesh**, **Materials**, **SubdomainProperty** – structures which define the design.

settingsMagnetostatic – magnetostatic FE simulation settings. Refer to [section 3.3](#) for more details.

SwitchDutyCycle – switch duty cycle (120 or 180 degrees) for trapezoidal current waveform. This parameter can be empty for sinusoidal current waveform. Refer to [section 9.3](#) of Part I of this user manual for more details.

4.2.4. Outputs

Results –structure with average magnetostatic FE simulation results. Refer to [Table 13](#) for more details on the **Results** structure fields.

Timesteppingdata –structure with time-stepping magnetostatic FE simulation results. Refer to [Table 14](#) for more details on the **Timesteppingdata** structure fields.

field – structure with field data results.

Private – structure with internal data of the FE solver. Note that function **runMStimestepping** also has an input argument **Private** (see above). If the input argument **Private** is empty while calling **runMStimestepping**, it will be calculated inside the **runMStimestepping** function and returned as the output argument **Private**. Since calculation of the **Private** structure may be time consuming, the output argument **Private** can be used for subsequent calls of **runMStimestepping** for the same design. If the input argument **Private** is not empty while calling **runMStimestepping**, the input and output **Private** structures will be equal.

Table 13. The **Results** structure fields.

Field	Description
SolutionConverged	SolutionConverged = 1 indicates that all points of the simulation converged. If SolutionConverged = 0, there was a problem with the solution convergence.
speed	Rotor speed (RPM).
f1	Supply frequency (Hz).
gamma	Advance angle °(el.deg).
torque	Total torque (N*m).
torque_reluctance	Reluctance torque (N*m).
torque_magnet	Magnet torque (N*m).
Is	RMS phase current (A).
Id	Average d-axis current (A).

Iq	Average q-axis current (A).
Vs	RMS phase voltage (V).
Vd	Average d-axis voltage (V).
Vq	Average q-axis voltage (V).
backEMF	RMS phase back-EMF (V).
Pinput	Input electrical power (W).
Pmech	Output mechanical power (W).
Ps	Stator winding loss (W).
Piron	Total iron core loss (W).
Piron_hyst	Hysteresis iron core loss (W).
Piron_eddy	Iron eddy current loss (W).
MagnetLoss	Magnet loss (W).
OtherECloss	Eddy current losses (W) in parts of the machine assigned with a material type 'Conductor'(retaining sleeve, etc.).
Piron_tooltip	Decomposition of the stator and rotor eddy current and hysteresis iron losses.
PowerFactor	Power factor.
Efficiency	Efficiency (%).
TorqueRipple	Torque ripple (%).
Error	Discretization error (%).
CurrentDensity	Current Density (A/mm ²)
DemagH_max	Max. demagnetization field (A/m).
DemagHpercent_max	Max. demagnetization field (% of Hcj).

Ke_rms	Back-EMF constant (V/RPM).
Ke_rad	Back-EMF constant (V*s/rad).
Kv_rms	Velocity constant (RPM/V).
Kv_rad	Velocity constant (rad/(V*s)).
Kt	Torque constant (N*m/A).
Km	Motor constant (N*m/sqrt(W)).
Bav_max_tooth	Peak of stator tooth average flux density (T).
Bav_max_backiron_stator	Peak of stator back iron average flux density (T).
Bav_max_backiron_rotor	Peak of rotor back iron average flux density (T).
Bav_airgap	Average airgap flux density (T).
Bmax_airgap	Maximum airgap flux density (T).
Lself	Phase self-inductance (mH).
Lmutual	Phase mutual inductance (mH).
Ld	D-axis inductance (mH).
Lq	Q-axis inductance (mH).

Table 14. The *Timesteppingdata* structure fields.

Field	Description
time	Time of each calculated point (s).
BackEMFa	Phase A RMS back-EMF (V).
BackEMFb	Phase B RMS back-EMF (V).
BackEFMc	Phase C RMS back-EMF (V).
BackEMFd	D-axis back-EMF (V).

BackEMFq	Q-axis back-EMF (V).
Gamma	Advance angle °(el.deg).
Torque_maxwell	Electromagnetic torque by Maxwell stress tensor (N*m).
Torque_work	Electromagnetic torque by virtual work method (N*m).
Torque_fluxlinkage	Electromagnetic torque by flux linkage and current (N*m).
Torque_magnet	Magnet torque (by Maxwell stress tensor) (N*m).
Torque_reluctance	Reluctance torque (by Maxwell stress tensor) (N*m).
Torque_cogging	Cogging torque (by Maxwell stress tensor) (N*m).
Ia	Stator phase A RMS current (A).
Ib	Stator phase B RMS current (A).
Ic	Stator phase C RMS current (A).
Id	D-axis stator current (A).
Iq	Q-axis stator current (A).
Va	Stator phase A RMS voltage (V).
Vb	Stator phase B RMS voltage (V).
Vc	Stator phase C RMS voltage (V).
Vd	D-axis stator phase voltage (V).
Vq	Q-axis stator phase voltage (V).
Fluxlinkage_a	Flux linkage phase A.
Fluxlinkage_b	Flux linkage phase B.
Fluxlinkage_c	Flux linkage phase C.

Fluxlinkage_d	Flux linkage d-axis.
Fluxlinkage_q	Flux linkage q-axis.
Pinput	Electrical input power (W).
Pmech	Output mechanical power (W).
Ps	Stator winding losses (W).
MagnetLoss	Magnet losses (W).
OtherECloss	Eddy current losses (W) in parts of the machine assigned with a material type 'Conductor'(retaining sleeve, etc.).
Convergence.converged	Indicates whether the solution has converged (1) or not (0) for each time step for each mesh slice.
Convergence.residual	Residual of the FEA solution for each time step for each mesh slice.
Convergence.message	Message that may indicate the reason for non-convergence.

4.3. Parametric sweep script example

In this section, it is explained how to run a parametric sweep analysis using an example of a motor with inner rotor which can be found in *SimFiles\example_innerrotor\example_innerrotor.mxp*. The analysis involves running multiple simulations to examine how the motor's efficiency is impacted by variations of the **ToothWidth** and **SlotDepth** parameters. To provide an overview of the process, a code diagram is shown in Figure 2, and the complete code can be found in *CustomScripts/ExampleParametricSweep.m*.

To start the parametric sweep analysis, the script is initialized by calling [**initMXPscriptingAPI**](#), and the initial mxp-file project is opened. Two arrays are then created, each containing six possible values for the parameters being swept: **ToothWidth** and **SlotDepth**. This generates a total of 36 possible combinations of parameter variations, with each combination resulting in a simulation run.

To set the initial values for the magnetostatic FE simulations and define the model geometry, the function [**setParampm**](#) is used. With the updated structures, they are then used to assemble each design using the [**assembleMXP**](#) function.

To vary the *ToothWidth* and *SlotDepth* parameters, two for-loops are created. To ensure that all the designs are compared under the same conditions, another loop is used to adjust the input stator current until the torque is approximately equal to the assigned target torque. It's worth noting that the initial magnetostatic simulations are performed using only one rotor position. During this stage, an additional structure is calculated and stored to improve the total simulation time: *Private*. The *Private* structure only needs to be redefined when the geometry changes.

After the first stage, once the stator current for each design is determined, the simulation with multiple points is run using [*runMStimestepping*](#). If all time steps converge, the results are saved, and the process is repeated for the new combination of *ToothWidth* and *SlotDepth* values until all possible combinations have been processed.

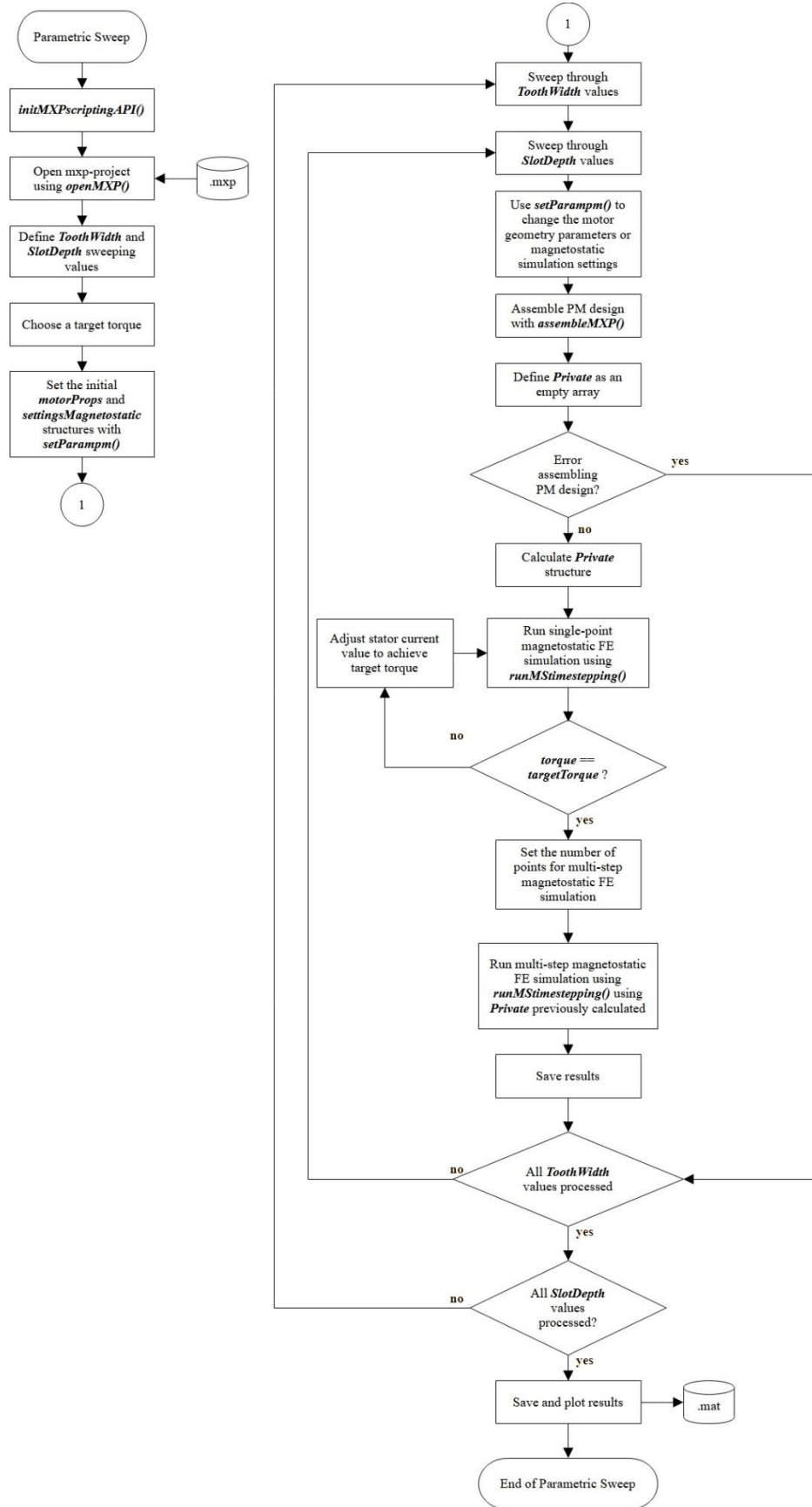


Figure 2. Parametric sweep code diagram.

5. Parallel processing functions

5.1. Assembling designs in parallel

5.1.1. *initAssembleMXP_par* function

The *initAssembleMXP_par* function initializes assembling designs in parallel.

Syntax

```
initAssembleMXP_par()
```

5.1.2. *runAssembleMXP_par* function

Syntax

```
runAssembleMXP_par(initMXPfile, motorProps)
```

Description

Function *runAssembleMXP_par* is used to assemble several designs in parallel.

The *runAssembleMXP_par* function is similar to the [assembleMXP](#) function and has the same input arguments. Each time *runAssembleMXP_par* is called, it initiates a new MotorXP-PM process for assembling one design which enables parallel processing, i.e. assembling several designs at the same time.

Inputs

There are two inputs: [motorProps](#) structure and the file path to the initial mxp-file project.

Outputs

There are no outputs; *getAssembleMXP_par* needs to be called to get the resulting cell arrays containing data for each design.

5.1.3. *getAssembleMXP_par* function

Syntax

```
[cell_Geometry, cell_Windings, cell_Mesh, cell_Materials,  
cell_SubdomainProperty, cell_Error, cell_motorProps_output,  
cell_Weight, numberOfProcesses] = getAssembleMXP_par()
```

Description

The ***getAssembleMXP_par*** function should be called after calling ***runAssembleMXP_par*** to get the cell arrays with assembled designs. The number of cells in each output cell array is equal to the number of the ***runAssembleMXP_par*** function calls.

Outputs

The output arguments of ***getAssembleMXP_par*** are similar to the output arguments of the ***assembleMXP*** function. The difference is that each output of ***getAssembleMXP_par*** is a cell array where each cell contains data equivalent to outputs of ***assembleMXP***.

cell_Geometry, ***cell_Windings***, ***cell_Mesh***, ***cell_Materials***, ***cell_SubdomainProperty*** - cell arrays of the structures containing each assembled design data. Refer to [section 4.1.4](#) for more details.

cell_Error - cell array of the error messages if the corresponding design failed assembling. Refer to description of ***Error*** in [section 4.1.4](#) for more details.

cell_motorProps_output - cell array of the output ***motorProps*** structures for each design. Refer to description of ***motorProps_output*** in [section 4.1.4](#) for more details.

cell_Weight - cell array of the active material weights for each design. Refer to description of ***Weight*** in [section 4.1.4](#) for more details.

numberOfProcesses - number of the MotorXP-PM processes initiated by the ***runAssembleMXP_par*** function calls currently running. Once the design assembling is finished, the corresponding process is terminated. The ***numberOfProcesses*** variable shows how many designs are currently being assembled.

5.2. Running magnetostatic FE simulations in parallel

5.2.1. *initMStimestepping_par* function

The ***initMStimestepping_par*** function initializes running magnetostatic FE simulations in parallel.

Syntax

```
initMStimesteppingpm_par()
```

5.2.2. *runMStimestepping_par* function

Syntax

```
runMStimestepping_par(Private,Geometry,Mesh,Windings,  
    settingsMagnetostatic,Materials,SubdomainProperty,  
    SwitchDutyCycle)
```


Description

Function ***runMStimestepping_par*** is used to run several magnetostatic FE simulations in parallel.

The ***runMStimestepping_par*** function is similar to the [runMStimestepping](#) function and has the same input arguments. Each time ***runMStimestepping_par*** is called, it initiates a new MotorXP-PM process for running one magnetostatic FE simulation which enables parallel processing, i.e. running several magnetostatic FE simulations at the same time.

Inputs

Private - structure with internal data of the FE solver. Same as described in [section 4.2.3](#).

Geometry, Windings, Mesh, Materials, SubdomainProperty - structures defining the design. Same as described in [section 4.2.3](#).

settingsMagnetostatic - magnetostatic FE simulation settings. Refer to [section 3.3](#) for more details.

SwitchDutyCycle - switch duty cycle (120 or 180 degrees) for trapezoidal current waveform. This parameter can be empty for sinusoidal current waveform. Refer to [section 9.3](#) of Part I of this user manual for more details.

5.2.3. ***getMStimestepping_par*** function

Syntax

```
[cell_Results, cell_Timesteppingdata, cell_field, cell_Private,
    numberOfProcesses] = getMStimestepping_par()
```

Description

This function should be called after calling function ***runMStimestepping_par*** to get the cell arrays with magnetostatic FE simulation results.

Outputs

cell_Results – cell array of the structures with average magnetostatic FE simulation results. Refer to description of ***Results*** in [section 4.2.4](#) for more details.

cell_Timesteppingdata – cell array of the structures with time-stepping magnetostatic FE simulation results. Refer to description of ***Timesteppingdata*** in [section 4.2.4](#) for more details.

cell_field – cell array of the structures with field data results.

cell_Private – cell array of the structures with internal data of the FE solver for each design. Refer to description of ***Private*** in [section 4.2.4](#) for more details.

numberOfProcesses – number of the MotorXP-PM processes initiated by the ***runMStimestepping_par*** function calls currently running. Once one of the magnetostatic FE simulations is finished, the corresponding process is terminated. The ***numberOfProcesses*** variable shows how many magnetostatic FE simulations are currently running.

5.3. Parametric sweep script example with parallel processing

This section explains how to run a parametric sweep with parallel processing to speed up the analysis. This example implements the same process as presented in [section 4.3](#), but, instead of the serial processing with functions [assembleMXP](#) and [runMStimestepping](#), their parallel processing analogs described above are used. To provide an overview of the process, a code diagram is shown in Figure 4, and the complete code can be found in *CustomScripts/ExampleParametricSweep_par.m*.

To perform the parametric sweep analysis, the script first calls the [initMXPs scripting API](#) function to initialize the scripting API and then opens the initial mxp-file project. Two arrays are created (***ToothWidth*** and ***SlotDepth***), each containing six values to be swept. This generates a total of 36 possible combinations of parameter variations, with each combination resulting in a simulation run and the results are saved in corresponding data arrays. Additionally, the maximum number of MotorXP-PM processes that can be run in parallel (variable ***maxNumberOfProcesses***) is specified at this stage. When defining the ***maxNumberOfProcesses*** value, it is recommended to consider the number of physical CPU cores and the amount of RAM. Figure 3 shows the case when the number of running MotorXP-PM processes is equal to four.

The [runAssembleMXP_par](#) function is used to start assembling the designs in parallel, while the [getAssembleMXP_par](#) function is used to retrieve the designs data until all designs are ready. Afterward, the designs are verified to ensure that each was created correctly. The parallel processing helps to reduce the overall analysis time, as the designs can be assembled simultaneously instead of sequentially.

The subsequent stage focuses on adjusting the stator currents for each design through parallel simulations, using the [runMStimestepping_par](#) and [getMStimestepping_par](#) functions. This process retains the maximum number of parallel processes until all designs achieve the target torque. During this stage, the ***Private*** structure is calculated with the first call of ***runMStimestepping_par*** and then this structure is used for all the subsequent calls to reduce the overall simulation time. Finally, [runMStimestepping_par](#) is executed with the desired accuracy for multi-step FE simulation and the results are kept in corresponding cell arrays.

Process Explorer - Sysinternals: www.sysinternals.com [HOFFMANN-PC\kleyt]

File Options View Process Find Users Help

<Filter by name>

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
Registry		16.288 K	61.820 K	168		
System Idle Process	84.19	60 K	8 K	0		
System	0.18	64 K	4.732 K	4		
Interrupts	0.55	0 K	0 K	n/a	Hardware Interrupts and DPCs	
smss.exe		1.164 K	1.168 K	676		
Memory Compression	< 0.01	2.772 K	898.052 K	3400		
csrss.exe		2.384 K	5.292 K	992		
wininit.exe		1.560 K	6.104 K	968		
csrss.exe	< 0.01	3.876 K	6.676 K	164		
winlogon.exe		3.068 K	14.424 K	1160		
explorer.exe	< 0.01	288.800 K	334.988 K	1808	Windows Explorer	Microsoft Corporation
SecurityHealthSystray.exe		1.920 K	10.024 K	10544	Windows Security notification ...	Microsoft Corporation
MaximAudioService64.exe		68.184 K	55.684 K	12696	Maxim(R) Audio Service	Maxim Integrated
AdskAccessCore.exe	< 0.01	36.656 K	84.404 K	4656	Autodesk Access Core	Autodesk, Inc.
OneDrive.exe		727.468 K	486.224 K	17072	Microsoft OneDrive	Microsoft Corporation
MATLAB.exe	0.73	1.699.244 K	1.262.180 K	9820	MATLAB R2021a	The MathWorks Inc.
cef_helper.exe	< 0.01	39.292 K	41.540 K	10584	Java Chromium Embedded F...	
cef_helper.exe	< 0.01	10.228 K	23.880 K	13136	Java Chromium Embedded F...	
cef_helper.exe	< 0.01	14.584 K	31.004 K	12684	Java Chromium Embedded F...	
cef_helper.exe	< 0.01	90.596 K	112.560 K	12224	Java Chromium Embedded F...	
cef_helper.exe	< 0.01	11.400 K	21.288 K	13228	Java Chromium Embedded F...	
MotorXP-PM_DS.exe	< 0.01	10.292 K	23.528 K	6940	MotorXP-PM Design Studio	Vepco
MotorXP-PM_DS.exe	< 0.01	9.956 K	23.352 K	4480	MotorXP-PM Design Studio	Vepco
MotorXP-PM_DS.exe	< 0.01	10.740 K	23.780 K	21472	MotorXP-PM Design Studio	Vepco
MotorXP-PM_DS.exe	0.18	10.740 K	23.788 K	23280	MotorXP-PM Design Studio	Vepco

Figure 3. Four MotorXP-PM processes running under the host MATLAB process.

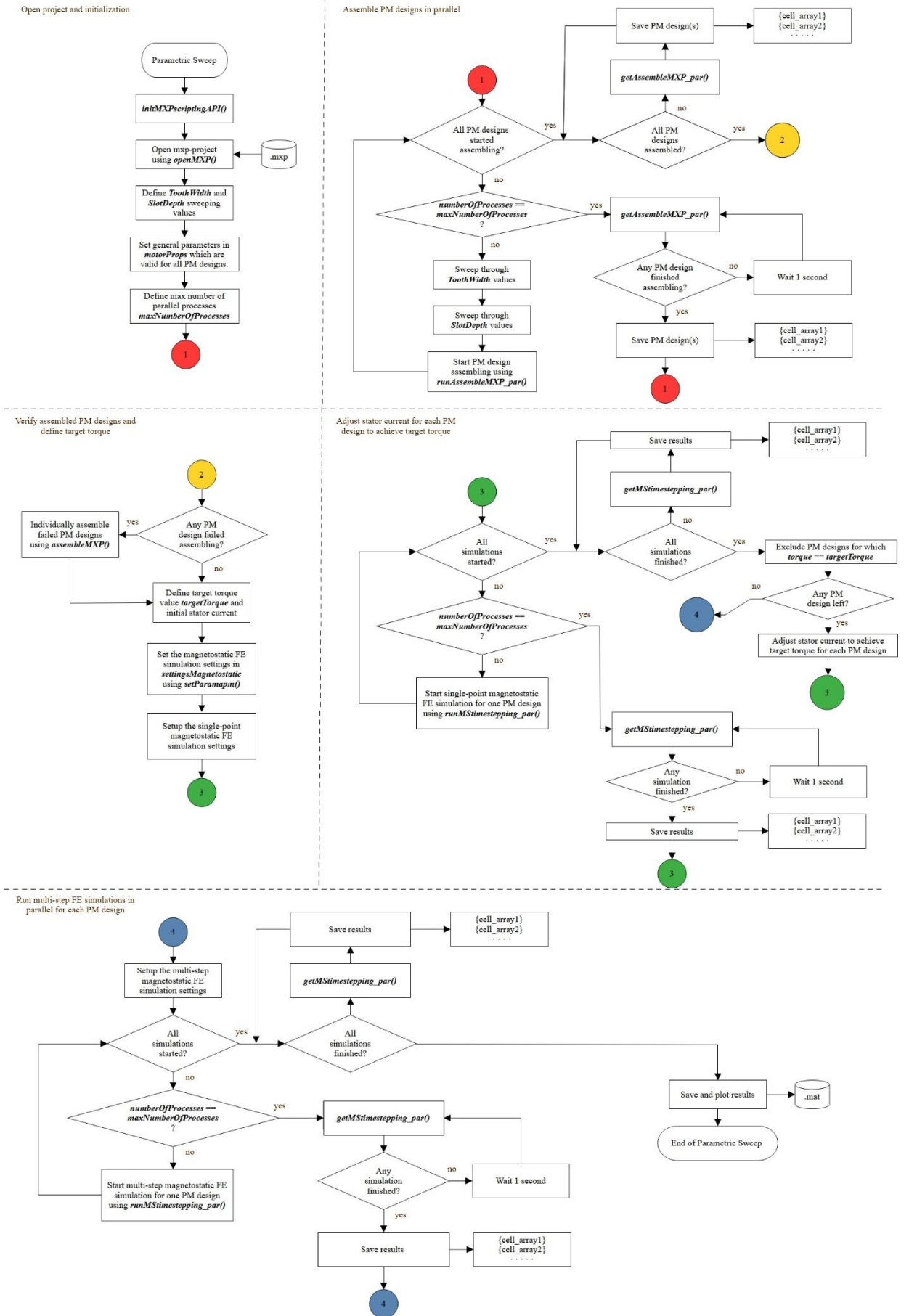


Figure 4. Parametric sweep code diagram with parallel processing.

6. Automatic optimization workflows

6.1. Overview

Electric machine automatic optimization is a process that involves using advanced software and algorithms to optimize the design of electric machines for maximum efficiency and performance. The idea behind this process is to modify the machine's geometry (and maybe other parameters) to achieve the desired performance metrics, such as torque, power, and efficiency, while also reducing weight, costs and environmental impact. The concept of electric machine automatic optimization is based on the principle that small changes to the machine's geometry can have a significant impact on its performance.

By optimizing the design of the machine, it is possible to achieve higher levels of efficiency and performance, while also reducing energy consumption and weight as well minimizing cost and environmental impact. The process of electric machine automatic optimization involves several steps. The first step is to define the performance metrics that the machine should achieve. This information is used to guide the optimization process and ensure that the machine meets the desired performance targets.

The next step is to identify the design variables that can be modified to achieve the desired performance metrics. These variables may include the shape and size of the rotor and stator, the number of poles, the winding configuration, and other factors. Once the design variables have been identified, a model of the machine is created using simulation and analysis software. Optimization algorithms are then used to find the optimal combination of design variables that will achieve the desired performance metrics. After the optimization process is complete, the optimized machine design is verified and validated using simulation and analysis software. This step involves simulating the machine's performance under various operating conditions to ensure that it meets the desired performance criteria.

Electric machine automatic optimization has numerous benefits, including increased efficiency, improved performance, reduced mass and cost. This process also leads to a positive impact on the environment, making it an essential tool in the push towards sustainability and energy efficiency. Overall, electric machine automatic optimization is a powerful process that can help to optimize the performance and efficiency of electric machines, leading to significant benefits for both manufacturers and end-users alike.

6.2. Functions for automatic optimization workflow development

There are several functions in the MotorXP-PM MATLAB scripting API for developing the custom automatic optimization workflows and visualizing the optimization results. By default, the “Thompson sampling efficient multiobjective optimization” (TSEMO) algorithm is used adapted for the electric machine optimization tasks with parallel processing. The original version of the algorithm can be found at <https://www.mathworks.com/matlabcentral/fileexchange/66588-multi-objective-optimization-algorithm-for-expensive-to-evaluate-function>. The adapted version of the optimization algorithm function can be found in `\m\scriptapi\TSEMO\TSEMO_V4.m`.

6.2.1. Suggested optimization workflow structure

The suggested optimization workflow structure includes the main script defining basic parameters such as number of processes (FE simulations) running in parallel, number of optimization algorithm iterations, total number of designs to be evaluated with FEA, initial or basic design which needs to be optimized, design variable bounds and optimization objectives. The main script also specifies the function implementing the FEA evaluation of the generated designs defined as a MATLAB function handle *@designFunction*. The *designFunction* is called on each iteration of the optimization algorithm receiving the design variable values and returning the objective values for each design. Finally, the main script calls the *runTSEMOptimizationpm* function executing the optimization algorithm.

6.2.2. runTSEMOptimizationpm function

Syntax

```
runTSEMOptimizationpm(designFunction, initdesignfile, ub, lb,
no_outputs, init_dataset_size, maxeval, niter, maxNumberOfProcesses,
save2filename, y_refdesign, labelObjective, Userdata)
```

Description

This function executes the optimization algorithm and displays the results as the Pareto plot. The corresponding optimization algorithm diagram is shown in Figure 5. The optimization algorithm is a combination of Gaussian process surrogate models with NSGA-II genetic algorithm which demonstrates very good optimization performance. The optimization results are saved to a mat-file on each iteration of the algorithm. It allows to continue the interrupted optimization from the point where it previously left off. If, however, this is the first run, the Latin hypercube sampling is employed to produce a random collection of initial designs. Then, the *@designFunction* function is invoked to evaluate designs generated on each iteration of the optimization algorithm.

Inputs

designFunction – MATLAB function handle processing generated designs.

initdesignfile – initial mxp-file (base design).

ub and *lb* – lower and upper bounds on design variables.

no_outputs – number of the optimization objectives.

init_dataset – number of initial designs randomly generated using Latin hypercube sampling.

maxeval – total number of generated designs to be processed with *designFunction*.

niter – number of optimization algorithm iterations.

maxNumberOfProcesses – maximum number of MotorXP-PM processes that can be run in parallel; the same as described in [section 5.3](#).

save2filename – save optimization algorithm results to this mat-file.

y_refdesign – (can be empty if not used) reference design objectives to be displayed on the Pareto plot as a red **x**.

labelObjective – (can be empty if not used) objective names to be displayed on the Pareto plot.

Userdata – structure that store user-defined variables and settings to ease the customization of the optimization workflows.

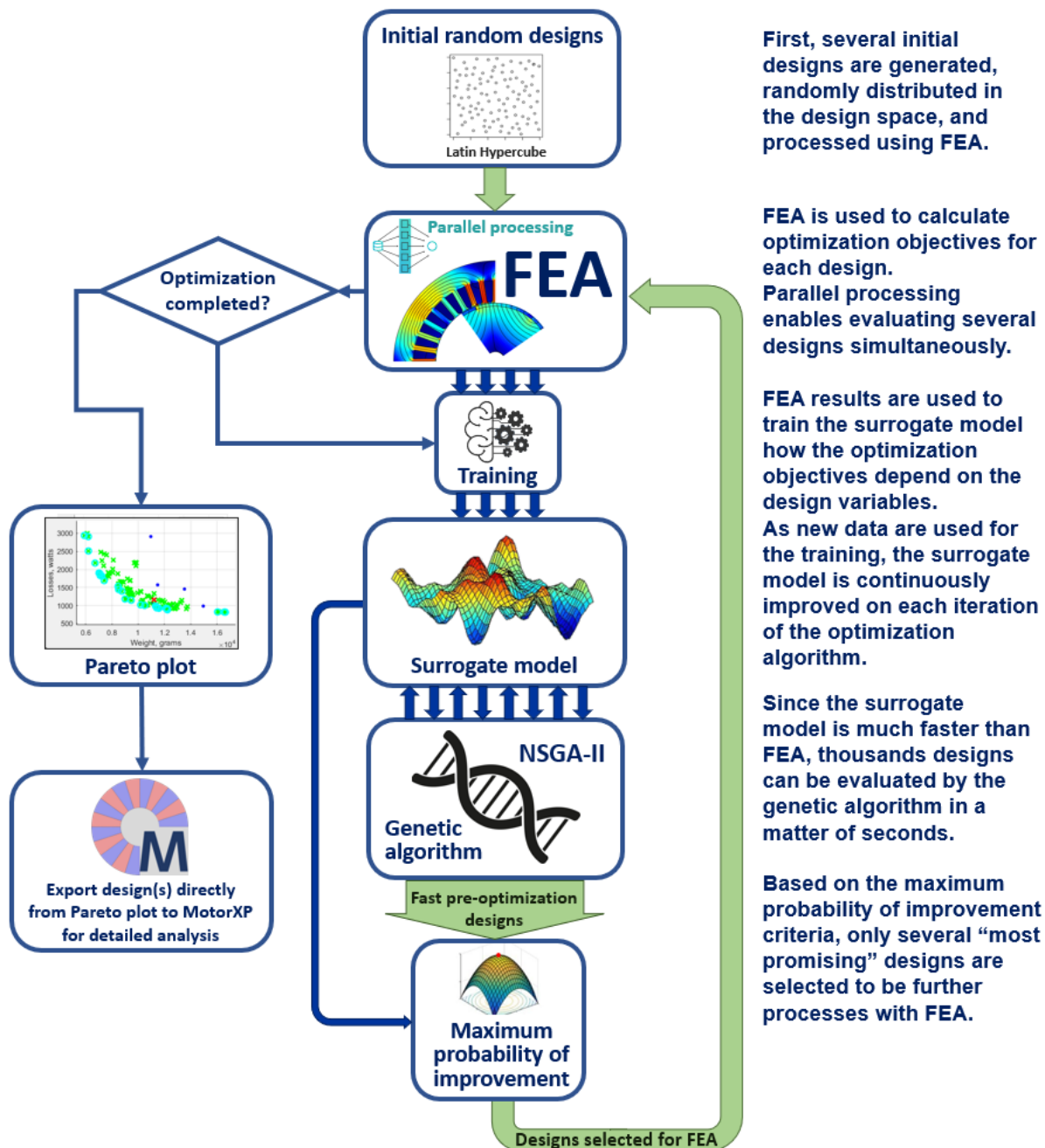


Figure 5. Optimization algorithm flowchart.

6.2.3. Design function (on an example of the *evalDesigns_SMPMSM* function)

Syntax

```
[Objectives, motorProps, Objectives_pareto, Userdata] =  
evalDesigns_SMPMSM(motorProps, settingsMagnetostatic, file,  
maxNumberOfProcesses, designParams, msg0, Userdata)
```

Description

This function is used for evaluating every generated design. The ***designFunction*** is called on each iteration of the optimization algorithm receiving the design variable values and returning the objective values for each design.

The source code of ***evalDesigns_SMPMSM*** can be found in `\CustomScripts\evalDesigns_SMPMSM.m`. The algorithm implemented in the ***evalDesigns_SMPMSM*** function is similar to that illustrated in Figure 4. Initially, all the designs are assembled using the parallel processing functions described in [section 5](#). Then, the target mechanical speed is set up for each design and the stator current of each design is adjusted to produce the target torque. To do that, the functions for running the magnetostatic FE simulations in parallel described in [section 5.2](#) are used. Finally, after the stator currents are determined, the multi-step magnetostatic FE simulations are performed for each design. To speed up the analysis, the number of rotor positions is set to 2 points per cogging torque period and the time is set to one electrical period. It was previously determined for the motor under study that these simulation settings provide reasonable accuracy for the motor parameters used for the analysis such as torque and losses. It is also possible to select either motor or generator operating mode. In case of the motor mode the current will be adjusted to achieve the target torque, for the generator operating mode the output electrical power is used as a target parameter.

Inputs

motorProps – see [section 3.2](#).

settingsMagnetostatic – see [section 3.3](#).

file – initial mxp-file (base design).

maxNumberOfProcesses – maximum number of MotorXP-PM processes that can be run in parallel; the same as described in [section 5.3](#).

designParams – design variable values for each design generated by the optimization algorithm.

msg0 – text message showing the optimization algorithm current iteration number to be displayed in the MATLAB command window.

Userdata – structure that store user-defined variables and settings.

Outputs

Objectives – optimization objective values for each design used by the optimization algorithm.

motorProps – see [section 3.2](#).

Objectives_pareto – optimization objective values for each design displayed on the Pareto plot.

Userdata – structure that store user-defined variables and settings.

6.2.4. plotParetoFromFile functionSyntax

```
plotParetoFromFile(fileOptimizationData,Xlimits,Ylimits,Zlimits)
```

Inputs

fileOptimizationData – mat-file with the optimization algorithm results (file name or full file path).

Xlimits – limits of the x-axis in the follow format [xmin, xmax].

Ylimits – limits of the y-axis in the follow format [ymin, ymax].

Zlimits – (used if there are three optimization objectives) limits of the z-axis in the follow format [zmin, zmax].

Input arguments **Xlimits**, **Ylimits**, **Zlimits** are optional.

Description

This function is used to display the Pareto plot for the optimization algorithm results stored in a mat-file **fileOptimizationData**. Inside the **plotParetoFromFile** the original **plotPareto** function is called. Figure 6 shows an example of the Pareto plot. The green **x** are the designs generated by the optimization algorithm, the blue dots **•** are the initial designs randomly generated using Latin hypercube sampling before the optimization started and the red **x** is the reference design, which can be optionally displayed on the Pareto plot for comparison purposes.

Right-clicking on the Pareto plot design point opens the context menu allowing to save the selected design to an mxp-file for further analysis, as shown in Figure 6. It is possible to show design variable bounds right-clicking inside the box without selecting any design. The full source code how these features are implemented can be found in:

```
\m\scrptapi\plotPareto.m
```

```
\m\scrptapi\ParetoMXPrighclickmenu.m
```

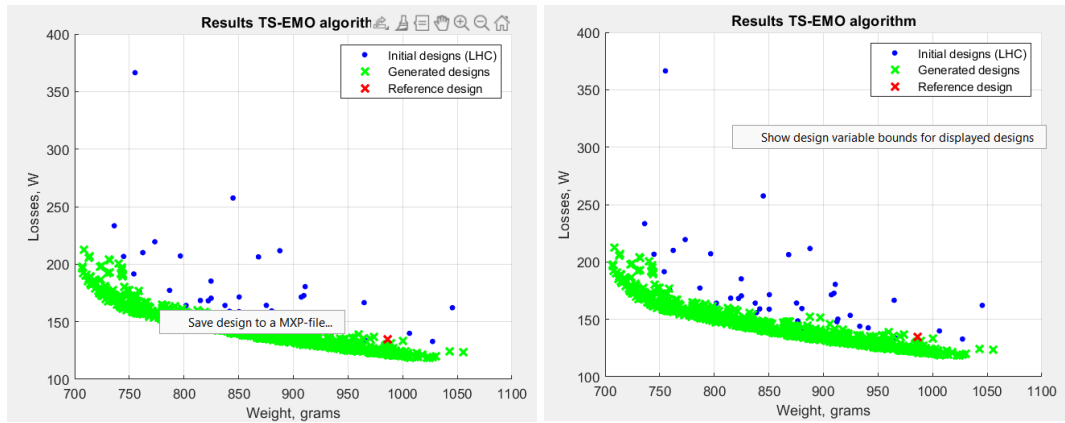


Figure 6. Example of the Pareto plot with the right-click context menu options.

6.3. Example of the automatic optimization workflow implementation

This section outlines the steps required to implement an optimization workflow. Figure 7 provides a schematic overview of the code, and the full source code can be found in:

CustomScripts\runTSEMO_MXPPM_SMPMSM_outrunner_example2.m

CustomScripts\evalDesigns_SMPMSM_outrunner_example2.m

This example implements the automatic optimization of a permanent magnet synchronous motor with outer rotor minimizing two objectives: motor weight and motor losses for the target operating point of 1.8 kW and 3000 RPM by varying several geometry parameters of the motor. The motor used as a reference design for this example can be found in

SimFiles/OptimizationBaseDesign/BaseDesign12s14pOutrunner.mxp. The parallel processing is used to speed up the optimization process allowing running FE simulations for several designs at the same time. The *evalDesigns_SMPMSM_outrunner_example2* function similar to the *evalDesigns_SMPMSM* function, previously described in [section 6.2.3](#), is used by the optimization algorithm for processing generated designs and calculating the objective values.

The lower and upper bounds for the optimization variables are defined by array *bounds*. The optimization variables used in this example can be divided into optional and mandatory variables. The optional optimization variables are

- Outer diameter.
- Air gap length.
- Lamination length.
- Inner diameter.

Each optional optimization variable has corresponding parameters defining its lower and upper bounds. For example, *outerDiameter_min* and *outerDiameter_max* variables define the bounds for the machine's outer diameter. However, the user can specify equal *outerDiameter_min* and *outerDiameter_max*, so the

outer diameter will be the same for all the designs, i.e. the outer diameter will not be treated as an optimization variable. This is the reason why these optimization variables are called “optional”, so the user can control whether to include these variables into the optimization or use fixed values. When the user specifies different values for *outerDiameter_min* and *outerDiameter_max*, these values will be added to the *bounds* array to define the lower and upper bounds for the optimization variable corresponding to the machine’s outer diameter.

The Slot Opening Depth is kept constant (1 mm). The mandatory optimization variables and their default lower and upper bounds are listed in Table 15.

Table 15. Mandatory optimization variables

Optimization variable	Symbol	Lower bound	Upper bound
Tooth width	<i>ratio_toothWidth</i>	4 mm	8 mm
Stator back iron depth coefficient	<i>coef_statorBackIronDepth</i>	0.3	0.8
Rotor back iron depth to stator back iron depth ratio	<i>ratio_rotorBackIronDepth</i>	0.8	1.2
Slot opening width	<i>slotOpeningWidth</i>	3 mm	6 mm
Tooth tip angle	<i>toothTipAngle</i>	15 el.degrees	25 el.degrees
Magnet depth	<i>magnetDepth</i>	1.5 mm	3 mm
Magnet Angle	<i>magnetAngle</i>	110 el.degrees	170 el.degrees

As can be seen from Figure 7, some designs are penalized (the design weight is increased) if the motor losses are too high. The penalty is especially useful if the design space is wide enough, i.e., geometry parameters vary in a wide range. In this case, the optimization algorithm tends to minimize the motor weight rather than losses as the relationship between the motor weight and its geometry is much more straightforward. Purposely increasing the weight as a penalty for too high losses helps guiding the optimization algorithm to minimize losses.

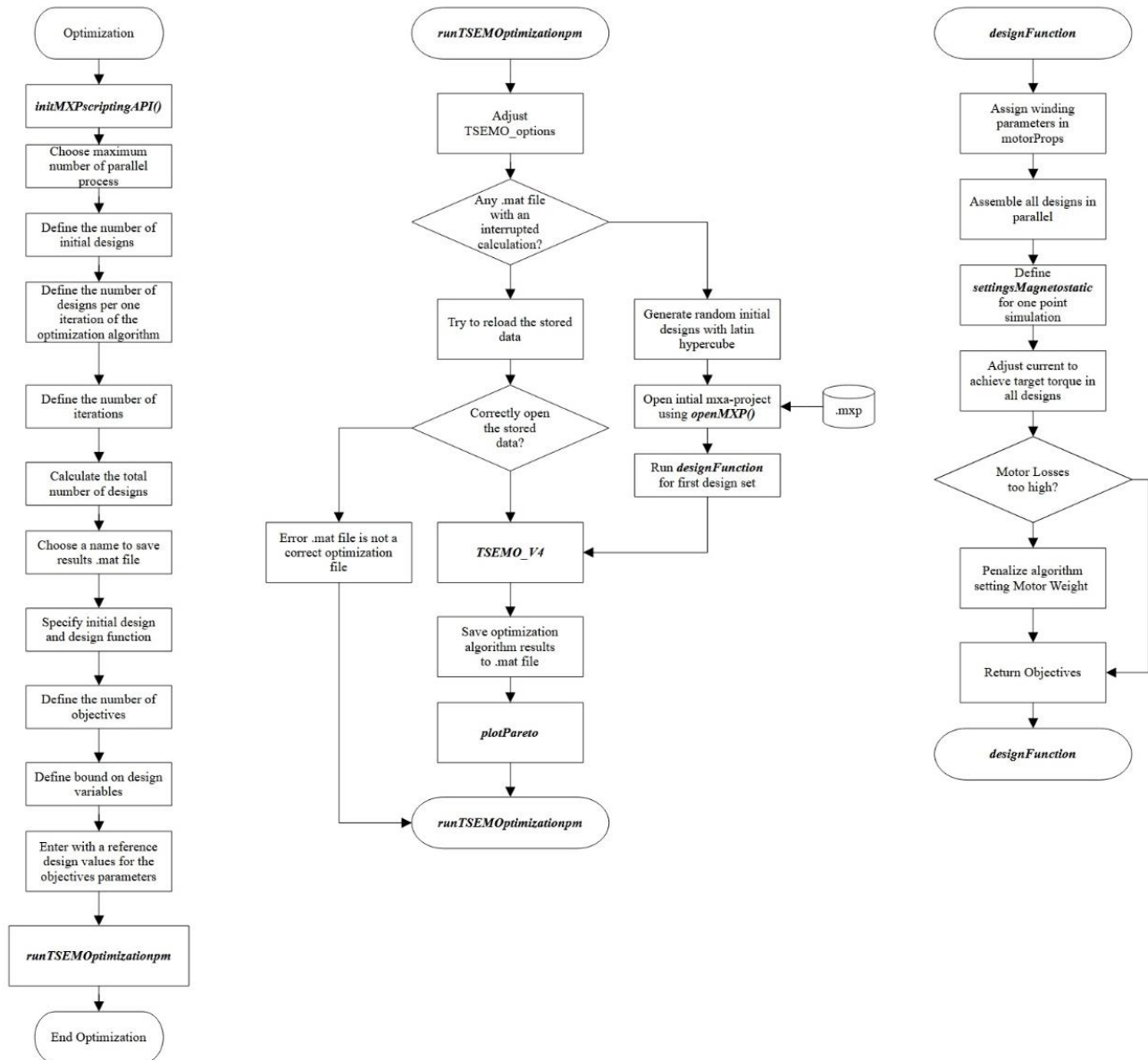


Figure 7. Code diagram of the optimization workflow example.

6.4. Example of the MATLAB code for automation of the motor and generator design workflows

The example described in [section 6.3](#) has already demonstrated the machine optimization workflow automation, but it requires the user to define the exact ranges for each optimization variable which requires some preliminary studies on the machine's geometry to determine sufficient bounds for the optimization variables. Too narrow bounds will increase the risk of finding not optimum solution while too wide bounds will lead to very long searches which may also fail to find the optimum solution because of the tremendous search space. Too wide optimization variable bounds will also lead to a large percentage of designs with inconsistent geometry dimensions. To avoid these problems, the optimization variables not linked to exact geometry dimensions are proposed and some basic machine sizing rules are

utilized to significantly narrow down the search space but without limiting the explored design space i.e. not excluding the eligible design candidates.

This example demonstrates a MATLAB code which will automatically design motors and generators with surface-mounted permanent magnets according to the user's requirements. The code combines the basic machine's sizing rules with an advanced optimization algorithm to fully automate the machine's design process. In this example, the optimization algorithm is configured to optimize the machine to minimize its weight and maximize efficiency (minimize losses) for the specific operating conditions (target speed and target power) defined by the user. The code can be used to automatically design a particular motor or generator, explore the design space or build your own routings for automation of the design workflows. This example is represented by three m-files, which can be found in the *CustomScripts* folder. Two m-file scripts called ***runTSEMO_MXPPM_SMPMSM_inrunner_example.m*** and ***runTSEMO_MXPPM_SMPMSM_outrunner_example.m*** implement the automatic design of the motor with inner and outer rotor respectively. The ***evalDesigns_SMPMSM*** function, the third m-file of this example, is used by the optimization algorithm for processing generated designs and calculating the optimization objective values. There are two optimization objectives in this example: minimization of the machine's weight and minimization of the machine's losses for target speed and target power. The current version of the optimization algorithm supports two or three optimization objectives for the same run. More optimization objectives can be set up by organizing several runs of the optimization algorithm. There are several input parameters required from the user and listed at the beginning of the script (see ***runTSEMO_MXPPM_SMPMSM_inrunner_example.m*** and ***runTSEMO_MXPPM_SMPMSM_outrunner_example.m***)

initdesignfile variable defines the initial or base mxp-file. This file can be created using MotorXP-PM Design Studio with the following steps:

1. Open MotorXP Design Studio and create a new project;
2. Select machine type, either 'Inner rotor' or 'Outer rotor';
3. Specify desired number of slots and number of pole pairs, select either 'Single layer' or 'Double layer' winding;
4. For the **Stator geometry** select 'Parallel tooth', for the **Rotor geometry** select 'Surface mounted radial';
5. Go to the **Winding Editor** tab and specify the desired coil span (for concentrated winding the coil span is 1, for distributed winding the coil span is more than 1);
6. For the 'Double layer' winding, select the layers orientation, either 'Upper / Lower' or 'Left / Right';

7. Go back to the **Geometry Editor** tab and select materials for the stator and rotor iron cores, magnets and stator winding; specify stacking factor for the stator and rotor iron cores and expected temperatures for the stator winding and magnets;
8. Number of magnet segments more than one can be optionally specified to reduce eddy current losses in magnets;
9. Save the resulting mxp-file. The recommended directory for base mxp-files is `\SimFiles\OptimizationBaseDesign`.

save2filename variable defines the mat-file to save the optimization algorithm results. The results are saved on each iteration of the optimization algorithm, so if the process was interrupted, it can be restarted from the point where it was interrupted.

motortype variable defines either the machine is with inner rotor or outer rotor.

coilFillFactor variable defines the fixed coil/slot fill factor for all the designs generated by the optimization algorithm. All the designs are compared with the same slot fill factor, so the winding resistance is determined only by the slot geometry to fulfil fair comparison of the designs.

machineMode variable defines either the machine is operated in the motor or generator mode.

The machine in this example is optimized for the specific operating conditions defined by the *targetSpeed* and *targetPower* variables. To ensure each design is operated under the same output power the current for each design is adjusted in a similar way how it is described in [section 5.3](#).

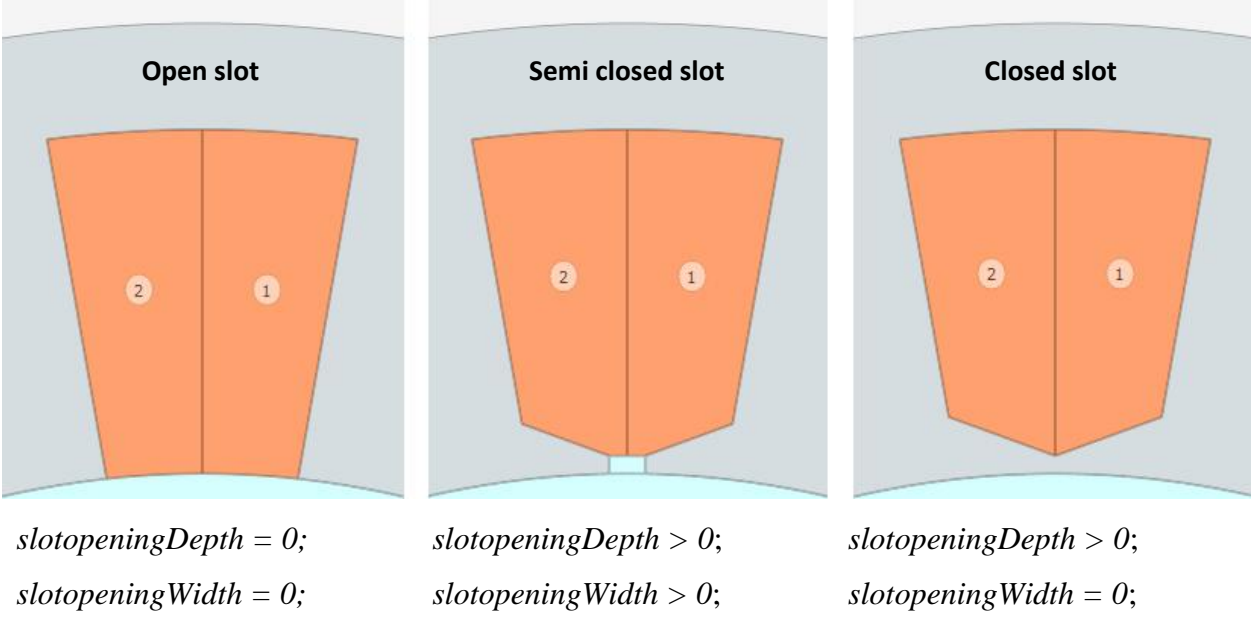
The lower and upper bounds for the optimization variables are defined by array *bounds*. The optimization variables used in this example can be divided into optional and mandatory variables. The optional optimization variables are

- Outer diameter
- Air gap length
- Lamination length
- Inner diameter
- Magnet weight

Each optional optimization variable has corresponding parameters defining its lower and upper bounds. For example, *outerDiameter_min* and *outerDiameter_max* variables define the bounds for the machine's outer diameter. However, the user can specify equal *outerDiameter_min* and *outerDiameter_max*, so the outer diameter will be the same for all the designs, i.e. the outer diameter will not be treated as an optimization variable. This is the reason why these optimization variables are called “optional”, so the user can control whether to include these variables into the optimization or use fixed values. When the user specifies different values for *outerDiameter_min* and *outerDiameter_max*, these values will be added

to the *bounds* array to define the lower and upper bounds for the optimization variable corresponding to the machine's outer diameter.

slotopeningDepth and *slotopeningWidth* variables define the slot opening dimensions. Depending on the *slotopeningDepth* and *slotopeningWidth* values, there are three options for the slot openings: open slot, semi closed slot and closed slot as shown below:



The *magnetWeight* variable can be used to optionally define the magnet material weight as a fixed value for all the designs. If *magnetWeight* does not exist or empty, the magnet weight will be treated as an optimization variable. In this case, the approach described in [1] is used to define the lower and upper bounds for the magnet material weight. First, the initial magnet volume *magnetVolume0* is calculated as:

$$magnetVolume0 = \frac{targetPower}{f_1 B_r H_c}$$

where

targetPower – the output power the machine is optimized for

f_1 – operational machine's frequency

B_r – residual flux density of the magnet

H_c – coercivity of the magnet

Then the magnet volume is calculated as

$$magnetVolume = C_v \cdot magnetVolume0$$

The default bounds for the magnet volume coefficient C_v are defined as 0.2 ... 4. Finally, the magnet weight is calculated by multiplying the magnet volume by the mass density of the magnet.

Reference machine design weight (in grams) and losses (in Watts) can be optionally defined for comparison with the designs generated by the optimization algorithm. The reference design is defined by variables *refWeight* and *refLoss*. The reference design will be displayed as a red **x** on the Pareto plot as shown in Figure 6.

The mandatory optimization variables and their default lower and upper bounds are listed in Table 16 and shown in Figure 8.

Table 16. Mandatory optimization variables

Optimization variable	Symbol	Lower bound	Upper bound
Tooth width angle to slot pitch angle ratio at the air gap	<i>ratio_toothWidth</i>	0.2	0.8
Stator back iron depth coefficient	<i>coef_statorBackIronDepth</i>	0.3	1
Rotor back iron depth to stator back iron depth ratio	<i>ratio_rotorBackIronDepth</i>	0.8	1.2
Magnet Angle	<i>magnetAngle</i>	110 el.degrees	179 el.degrees

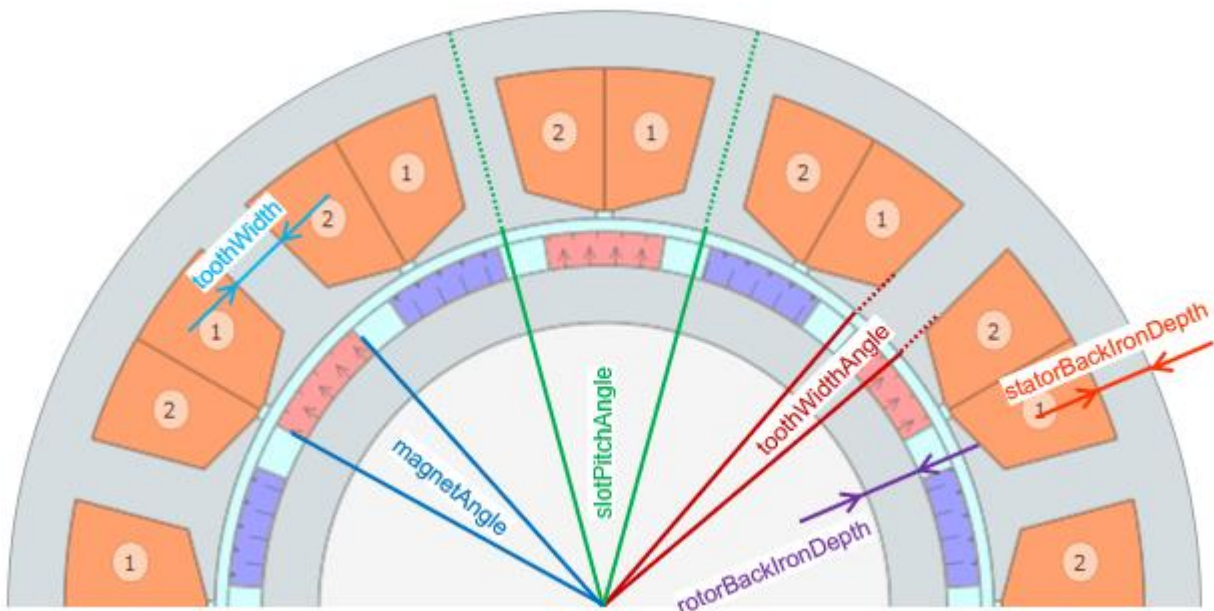


Figure 8. Explanation on the mandatory optimization variables.

The first optimization variable from Table 16 determines the relationship between the areas occupied by stator teeth and slots and is defined as a ratio between the tooth width angle and slot pitch angle (as show in Figure 8):

$$ratio_toothWidth = \frac{toothWidthAngle}{slotPitchAngle}$$

The second variable from Table 16 shows the relationship between the stator back iron depth and tooth width. For the concentrated winding (coil span equals one) this relationship is:

$$statorBackIronDepth = coef_statorBackIronDepth \cdot toothWidth$$

For the distributed winding (coil span is higher than one) $statorBackIronDepth$ also depends on the number of stator slots (N_s) and number of pole pairs:

$$statorBackIronDepth = \frac{coef_statorBackIronDepth \cdot toothWidth \cdot N_s}{4 \cdot nPolePairs}$$

The third variable from Table 16 defines a ratio between the rotor back iron depth and the stator back iron depth:

$$rotorBackIronDepth = ratio_rotorBackIronDepth \cdot statorBackIronDepth$$

Finally, the fourth optimization variable $magnetAngle$ corresponds the angle of the magnet arc in electrical degrees. Note that full pole pitch corresponds to 180 electrical degrees (see Figure 8).

Once all the optimization variables and fixed geometry parameters are defined, the iterative search is organized to find the machine's geometry satisfying all these parameters. In some cases, this search may lead to an erroneous machine's geometry with inconsistent dimensions. To prevent this, some geometry limits are defined, so the search is interrupted once one of these limits is reached. The limits are defined as follows:

maximum magnet depth to the total machine's depth ratio $ratio_magnetDepth_max$:

$$\frac{magnetDepth}{(outerDiameter - innerDiameter)/2} < 0.7$$

minimum stator slot depth to the total stator depth ratio $ratio_slotDepth_min$:

$$\frac{slotDepth}{slotDepth + statorBackIronDepth} > 0.1$$

As already discussed in [section 6.3](#), the penalty is used to guide the optimization algorithm to search for designs with lower losses. If the losses exceed some specified threshold, the design is penalized – the

weight of the design is multiplied by a penalty coefficient. The penalty coefficient is increased proportionally to the degree of violation, i.e. how much the losses exceed the penalty threshold. In the previous example given in [section 6.3](#) the penalty loss threshold was directly defined by the user. In this example the penalty loss threshold is calculated based on the initial designs generated using Latin hypercube sampling. After the initial designs are evaluated, first 1/5 most efficient design are selected, and the penalty loss threshold is calculated as an average loss value for these designs. Then the penalty loss threshold *penaltyLossThreshold* is stored in the *Userdata* structure. As already described in [section 6.2.2](#), the *Userdata* structure is used to store the user-defined variables and settings.

7. MATLAB code debugging tips

There can be some difficulties while debugging the MATLAB code that is using the parallel processing functions of the MotorXP-PM scripting API. In case an error occurs, most of the time the [getAssembleMXP_par](#) and [getMStimestepping_par](#) functions retrieve empty data structures without showing the actual error message. It allows continuing the execution of the code without interruption but creates difficulties understanding the reason for the error. A good programming practice in this case is to check for empty data and repeat the operation with an analog serial processing function. For example, the following code is used in the [evalDesigns SMPMSM](#) function to handle the designs which failed assembling with parallel processing:

```
% check if any design failed assembling and try again
for i = 1:nDesigns
    if isempty(cell_Geometry_ready{i}) || ~isempty(cell_Error_ready{i})
        motorProps = getMotorProps(motorProps, designParams, Userdata, i);
        try
            [Geometry, Windings, Mesh, Materials, SubdomainProperty, Error, motorProps_output, Weight] = assembleMXP(file, motorProps);
        catch ME
            Stop=1;
            rethrow(ME)
        end
        if isempty(Geometry)
            cell_Error_ready{i} = 'Unknown error';
        else
            cell_Geometry_ready{i} = Geometry;
            cell_Windings_ready{i} = Windings;
            cell_Mesh_ready{i} = Mesh;
            cell_Materials_ready{i} = Materials;
            cell_SubdomainProperty_ready{i} = SubdomainProperty;
            cell_Error_ready{i} = Error;
            cell_motorProps_output_ready{i} = motorProps_output;
            cell_Weight_ready{i} = Weight;
        end
    end
end
end
```

The MATLAB *try-catch* construction with a break point put on the *catch* statement can also be used for code debugging and analyzing errors. The following code and break point are used for debugging the possible errors while running magnetostatic FE simulations in the [evalDesigns](#) function:

```
414 % check if any simulation failed and try again
415 for i = 1:nDesigns
416     Error = cell_Error{i};
417     if isempty(Error) && enableRun(i)
418         if isempty(cell_Results_ready{i})
419             Geometry = cell_Geometry{i};
420             Windings = cell_Windings{i};
421             Mesh = cell_Mesh{i};
422             Materials = cell_Materials{i};
423             SubdomainProperty = cell_SubdomainProperty{i};
424             settingsMagnetostatic = setParampm(settingsMagnetostatic, 'RMS supply current', arr_Current{i});
425             try
426                 [Results, Timesteppingdata] = runMStimestepping([], Geometry, Mesh, Windings, settingsMagnetostatic, Materials, SubdomainProperty, 0);
427             catch ME
428                 Stop=1;
429                 rethrow(ME)
430             end
431             cell_Results_ready{i} = Results;
432             cell_Timesteppingdata_ready{i} = Timesteppingdata;
433         end
434     end
435 end
```

It should be kept in mind that not all combinations of design variables result in a valid design and not all invalid designs can be automatically recognized. It is the responsibility of the user of the current API to check for results validity, such empty results, NaN, Inf or zero values.

Finally, there are log files recording all the operations of the program, which can also help identifying possible problems. They can be found in:

`C:\Users\[username]\AppData\Local\VepcoTech\MotorXP-PM Design Studio\logs`

`C:\Users\[username]\AppData\Local\VepcoTech\MotorXP-PM_Stub\logs`