SWAT-EM API Documentation

Release 0.4.0

Martin Baun

Jan 11, 2020

CONTENTS

This documentation is for scripting "SWAT-EM". SWAT-EM is a software for designing and analyzing winding systems for electrical machines. Currently supported are rotating field windings (permanent-magnet motors, induction motors, synchronout reluctance motors) with any number of phases. This can be distributed full pitch or fractional slot winding or tooth-coil winding. The design can be done by

- Generating with manual allocation of the coil sides to stator slots
- Defining individual number of turns for each coil
- Automatic winding generators
- Tables of possible winding systems for slot/pole combinations

Analyzing features

- Calculation of the winding factor based on the voltage star of slots
- Plot of the winding layout
- Plot of stator ampere-conductor distribution and the magnetomotive force (MMF)
- Plot of the slot voltage phasors
- Plot of the winding factor
- Max. possible number of parallel circuit connection of coils

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Installation

To install, you can do:

```
$ pip install swat-em
```

or download it from https://gitlab.com/martinbaun/swat-em/ and install it from the project root directory:

```
$ python setup.py install
```

If you running windows and don't want to deal with a python interpreter you can download an *.exe installer or a portable *.zip version which includes all python dependencies from https://sourceforge.net/projects/swat-em/

1.1.1 Run the programm

Run the programm by starting it from the command line:

```
swat-em
```

or with:

```
swat-em.exe
```

under windows. If you have installed swat-em with the windows installer you can use the desktop icon to start.

1.2 Basic usage

1.2.1 Simple overlapping winding

For the beginning let's have a look how we can collect magnetic flux generated by a permanent-magnet rotor through a coil. The highest flux we get, if we define the coil width W equal to the pole pitch τ_p. However this in practise this often isn't the best choise because of the high harmonic content. Most windings have $W < \tau_p$.

For a symmetric three-phase winding we have to add two more coil which are shifted by 120°. For two poles this is one of the most simplest winding.

Let's have a look how we can model this simple winding with swat-em. First of all we need to import swat-em. The relevant part is the datamodel() object. It includes all data and methods for the winding:

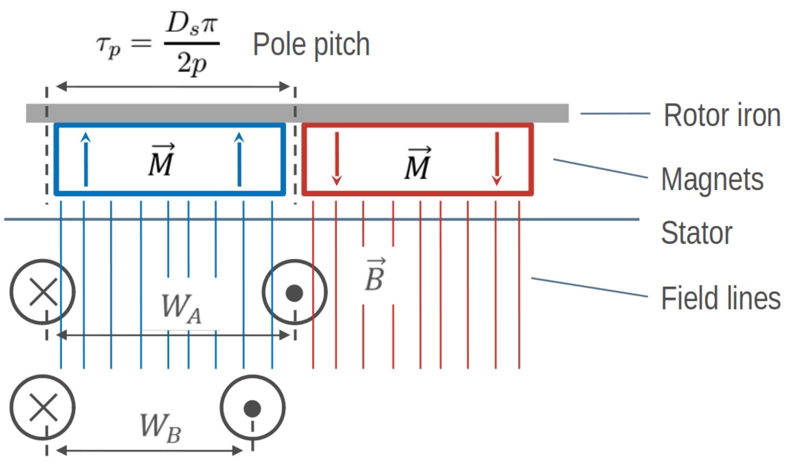

Fig. 1: How to get flux, based on the rotor, through a coil

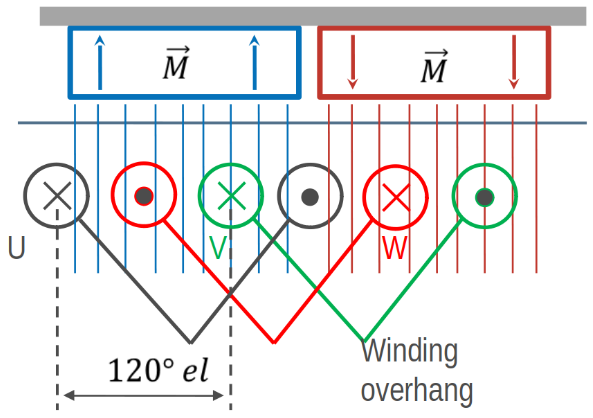

Fig. 2: Overlapping winding with 6 slots, 2 poles and 3 phases

```
from swat_em import datamodel
```

The model has an built-in winding generator for almost every winding for rotating field motors such as permanent-magnet, synchronous or induction machines:

```
>>> wdg = datamodel()       # generate a datamodel for the winding
>>> Q = 6                    # number of slots
>>> P = 2                    # number of pole pairs
>>> wdg.genwdg(Q = Q, P = P, m = 3, layers = 1)
>>> print(wdg)              # print infos for the winding
WINDING DATAMODEL
=================

Title: Untitled
Number of slots:  6
Number of poles:  2
Number of phases: 3
Number of layers: 1
Winding step    : 3
Number of slots per pole per phase: 1
Fundamental winding factor: 1.0, 1.0, 1.0
```

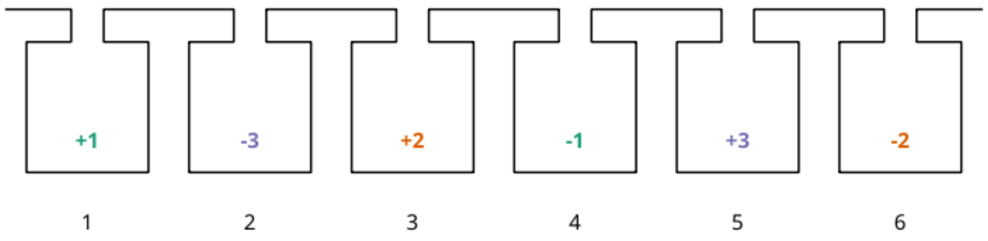

Fig. 3: Generated overlapping winding

1.2.2 Simple tooth-coil winding

Besides of the overlapping winding there is another winding winding systems - tooth coils. To get such a winding the winding step must be exactely $W = 1$. This means, that the distance between a wire and its reverse wire is one slot.

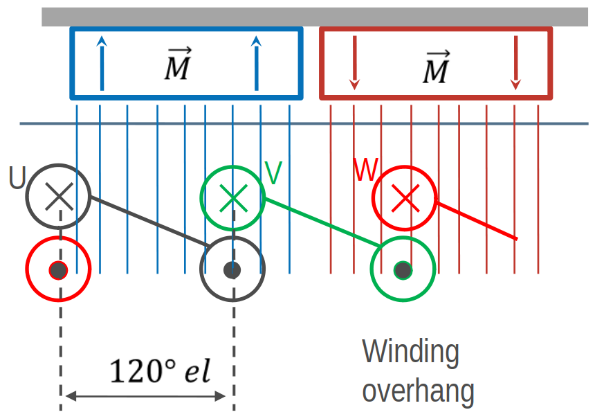

Fig. 4: Tooth-coil winding with 3 slots, 2 poles and 3 phases

We can set the winding step explicite with the keyword 'stepwidth'. Compared to the overlapping winding we need only 3 slots for the two poles. To get a coil around every tooth, we need two winding layers:

```
>>> wdg = datamodel()       # generate a datamodel for the winding
>>> Q = 3                    # number of slots
>>> P = 2                    # number of pole pairs
>>> w = 1                    # step width for the coil in slots

>>> # generate winding automatically
>>> wdg.genwdg(Q = Q, P = P, m = 3, layers = 2, w = w)
>>> print(wdg)              # print infos for the winding
WINDING DATAMODEL
=================

Title: Untitled
Number of slots:  3
Number of poles:  2
Number of phases: 3
Number of layers: 1
Winding step    : 1
Number of slots per pole per phase: 1/2
Fundamental winding factor: 0.866, 0.866, 0.866
```

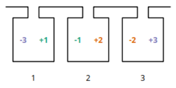

Fig. 5: Winding layout with 3 slots, 2 poles and 3 phases

1.2.3 A more complex winding

A more complex winding (overlapping full pitch winding with coil shortening)

```
>>> wdg = datamodel()
>>> Q = 12
>>> P = 2
>>> w = 5      # without shortening w would be 6 for this winding
>>> wdg.genwdg(Q = Q, P = P, m = 3, layers = 2, w = w)
>>> print(wdg)
WINDING DATAMODEL
=================

Title: Untitled
Number of slots:  12
Number of poles:  2
Number of phases: 3
Number of layers: 2
Winding step    : 5
Number of slots per pole per phase: 2
Fundamental winding factor: 0.933, 0.933, 0.933
```

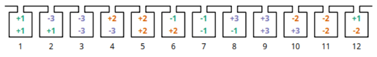

Fig. 6: Winding layout with 12 slots, 2 poles and 3 phases

1.3 Getting Results

After generating a winding, swat-em analyze it and provides the results:

```
>>> wdg = datamodel()
>>> wdg.genwdg(Q = 12, P = 2, m = 3, layers = 1)
>>> print('fundamental winding factor: ', wdg.get_fundamental_windingfactor())
fundamental winding factor:  [0.9659258262890683, 0.9659258262890683, 0.
↪9659258262890684]
>>> print('winding step: ', wdg.get_windingstep())
winding step:  6
```

Get the generated winding layout: For each phase there is a list of the 1st and the 2nd layer. In this example there is only 1 layer, so the second list is empty. An entry of the lists define the slot number in which is a coil-side of the phase is located. A negative number means, that the winding direction is reversed in the slot.

```
>>> print('winding layout:', wdg.get_phases())
winding layout: [[[1, 2, -7, -8], []], [[5, 6, -11, -12], []], [[-3, -4, 9, 10], []]]
```

The winding factor depends on the harmonic number. There are two possible interpretations for the harmonic number: The 'electrical' harmonic numbers the 'mechanical' ordinal numbers multiplyed with number of pole pairs 'p'. Use the 'mechanical' winding factor if you want du determine the possible number of poles your winding can drive and use the electrical winding factor if you know your number of pole pairs and if you want to analyze the harmonic content of the winding for example. Attention: The winding factor is calculated for each phase seperately.

```
>>> nu, kw = wdg.get_windingfactor_el()
>>> for k in range(len(nu)):
>>>     print(nu[k], kw[k])
1 [0.96592583 0.96592583 0.96592583]
3 [-0.70710678 -0.70710678 -0.70710678]
5 [-0.25881905 -0.25881905 -0.25881905]
7 [0.25881905 0.25881905 0.25881905]
9 [-0.70710678 -0.70710678 -0.70710678]
...
```

The datamodel() object stores the data in dictionaries. The user have direct access:

```
>>> print('Data for the machine: ', wdg.machinedata.keys())
Data for the machine:  dict_keys(['Q', 'p', 'm', 'phases', 'wstep', 'turns',
↪'phasenames'])
>>> # ... and all results:
>>> print('Data for the machine: ', wdg.results.keys())
Data for the machine:  dict_keys(['q', 'nu_el', 'Ei_el', 'kw_el', 'phaseangle_el',
↪'nu_mech', 'Ei_mech', 'kw_mech', 'phaseangle_mech', 'valid', 'error', 't', 'wdg_is_
↪symmetric', 'wdg_periodic', 'MMK', 'basic_char'])
```

1.4 Plots

1.4.1 Winding layout

SWAT-EM provides some possibilities for graphical representations. After creating a winding one would like to have a look on the layout, for example. This plot includes all coil sides of all phases in the slots:

```
>>> wdg = datamodel()
>>> wdg.genwdg(Q = 12, P = 2, m = 3, layers = 1)
>>> wdg.plot_layout('plot_layout.png')
```

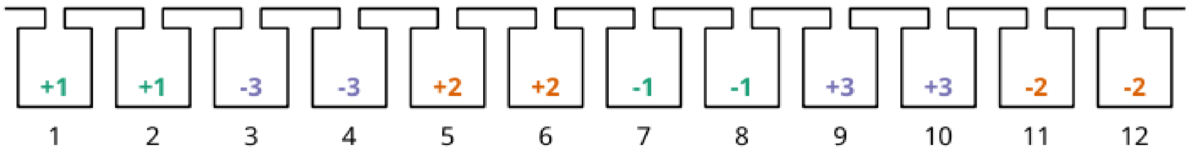

Fig. 7: Plot of the winding layout

1.4.2 Voltage phasors of the star of slot

SWAT-EM calculates the winding factor by the slot voltage phasors. The following is the corresponding visualization.

```
>>> wdg.plot_star('plot_star.png')
```

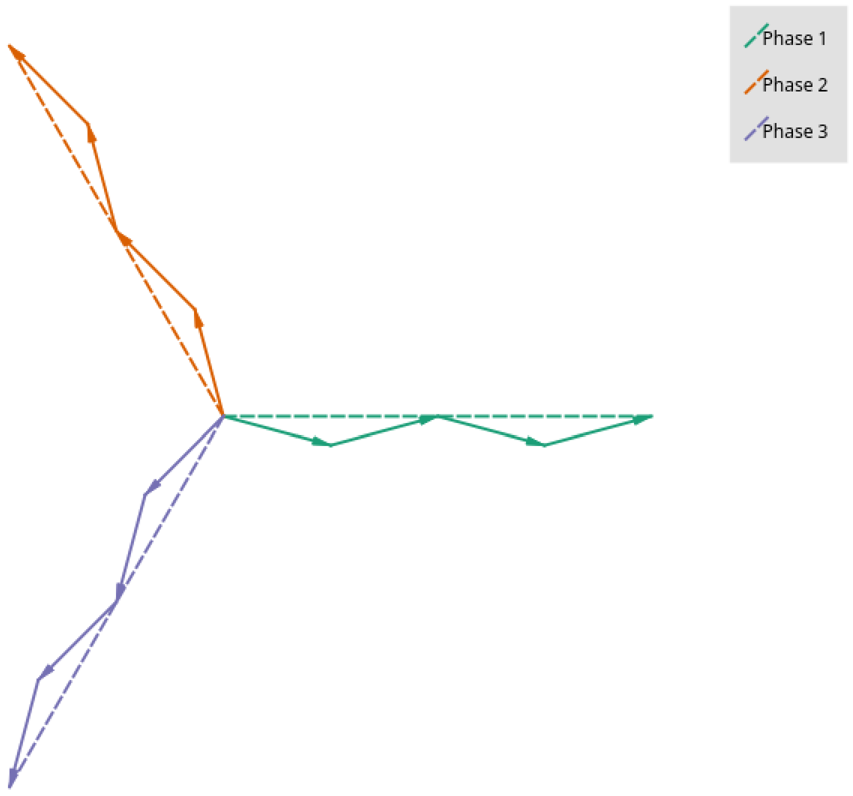

Fig. 8: Plot of the voltage phasors

1.4.3 Winding factor

For the winding factor one have to decide between the mechanical or the electrical winding factor. Attention: For a 2-pole machine the electrical and mechanical winding factor is equal.

```
>>> wdg.plot_windingfactor('plot_wf.png', mechanical = False)
```

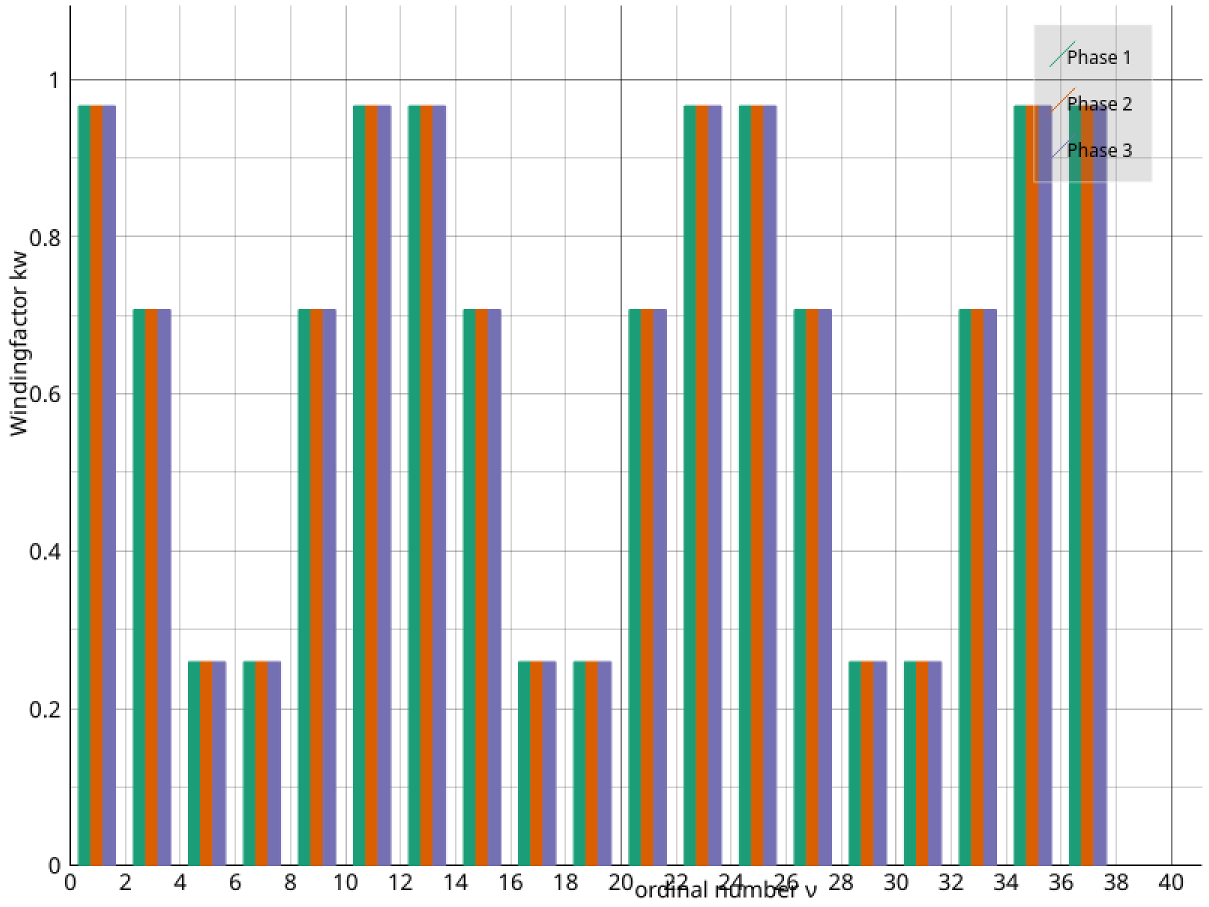

Fig. 9: Plot of the electrical winding factor

1.4.4 Magnetomotive force

The winding generates a current linkage in the slots. The integral of it leads to a magnetic field in the airgap, which is called the ‘Magnetomotive force (MMF)’. It’s a good indicator for the harmonic content of the winding. Also the resultion of the image can be definded:

```
>>> wdg.plot_MMK('plot_MMK.png', res = [800, 600], phase = 0)
```

It also could be usefull to plot at different phase angles

```
>>> wdg.plot_MMK('plot_MMK_20deg.png', res = [800, 600], phase = 20)
```

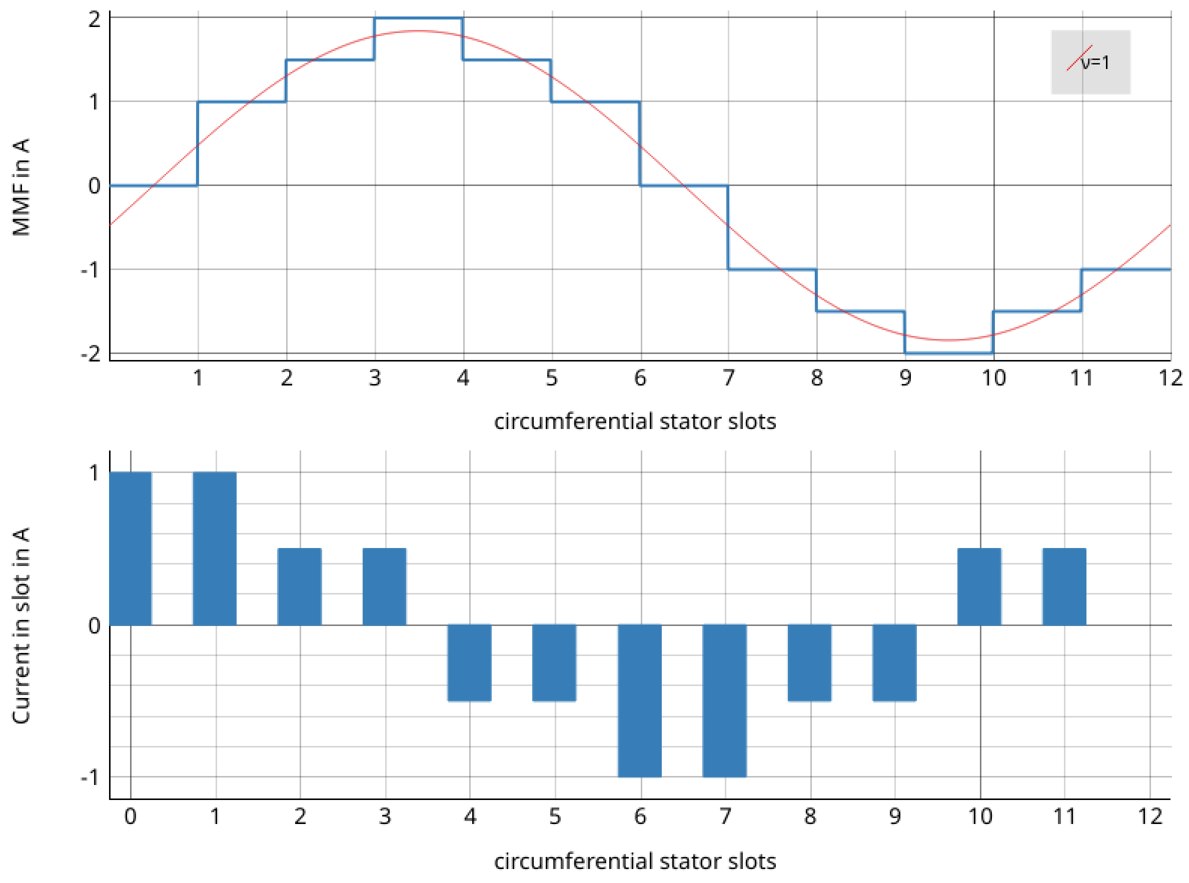

Fig. 10: Plot of the current linkage in the slots and the resulting Magnetomotove force

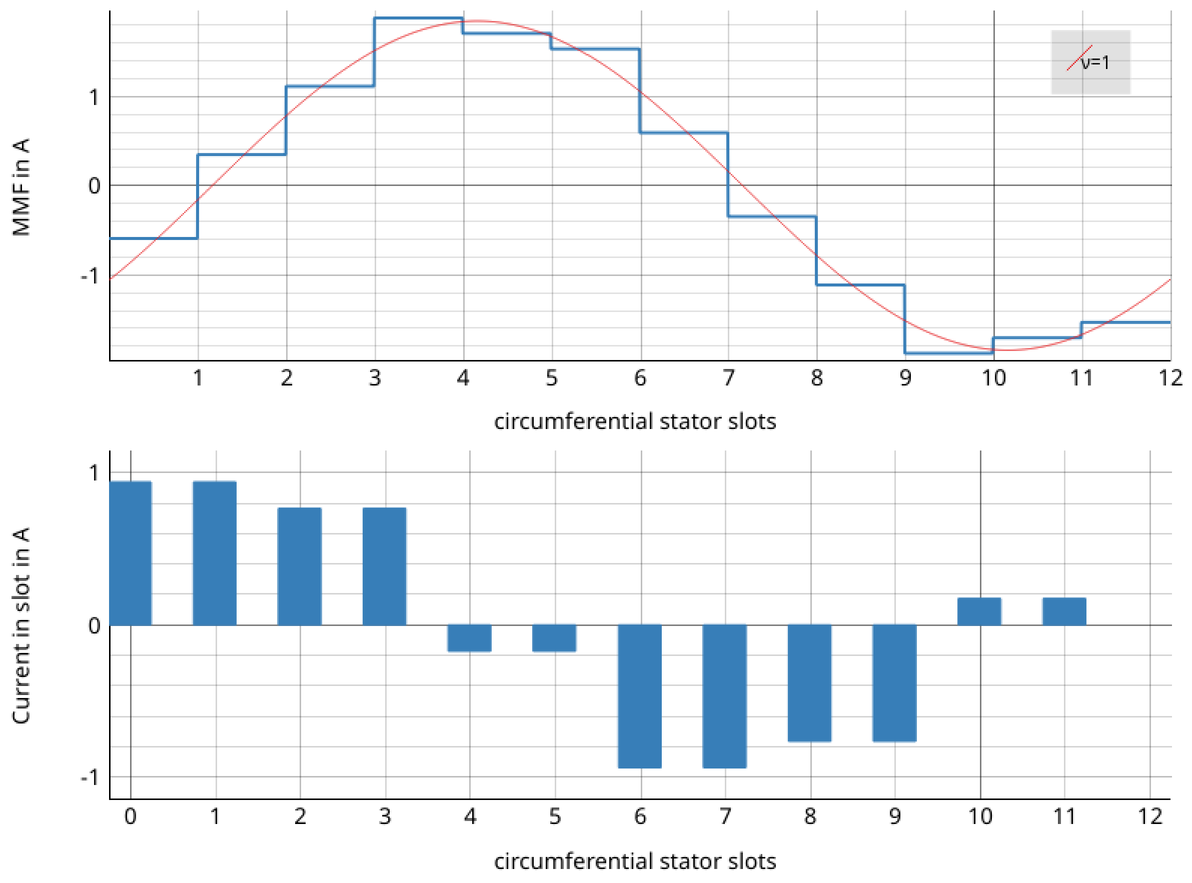

Fig. 11: Plot of the current linkage in the slots and the resulting Magnetomotove force with phaseangle = 20°.

1.5 File IO

1.5.1 Save/load a winding

After creating a winding we can save it as a *.wdg file This file can be used with the GUI for example. swat-em uses the "json" format for the *.wdg files.

```
>>> wdg = datamodel()
>>> wdg.genwdg(Q = 12, P = 2, m = 3, layers = 1)
>>> wdg.save_to_file('myfile.wdg')
```

We can also load an existing winding from file:

```
>>> wdg2 = datamodel()
```

Proof, that the data of the two objects is equal:

```
>>> print('same data?:', wdg.machinedata == wdg2.machinedata)
same data?: True
>>> print('same results?:', wdg.results == wdg2.results)
same results?: True
```

1.5.2 Export to Excel file

The data of an existing winding can exported to an Excel file (*.xlsx). Attention: The old *.xls format is not supported!

```
>>> wdg.export_xlsx('export.xlsx')
```

1.5.3 Text report

A summary of the winding can be exported as a text report:

```
>>> wdg.export_text_report('report.txt')
```

1.5.4 HTML report

Similar to the text report we can create a html report. This also includes the graphics.

```
>>> wdg.export_html_report('report.html')
```

1.6 Reference

class swat_em.datamodel.**datamodel**

Provides a central place for all data. All analysis functions are connect with this class.

analyse_wdg()

analyses the winding (winding factor etc.)

calc_num_basic_windings_t()

Calculates and returns the number of basic windings 't' for the actual winding layout

Returns **t** – Periodicity for the winding layout

Return type integer

`export_html_report` (*fname=None*)
Returns a winding report.

Parameters **`fname`** (*`string`*) – file name for html file. If not given a file is created in the temp dir of the file system (the file name ist returned by this function)

Returns **filnename** – The file name of the html-file which is stored in tmp directory

Return type string

`export_text_report` (*fname*)
Export winding report as a text file.

Parameters **`fname`** (*`string`*) – file name

`export_xlsx` (*fname*)
Export the results to Excel xlsx file.

Parameters **`fname`** (*`string`*) – file name

`genwdg` (*Q*, *P*, *m*, *layers*, *w=-1*, *turns=1*)
Generates a winding layout and stores it in the datamodel

Parameters

- **`Q`** (*`integer`*) – number of slots
- **`P`** (*`integer`*) – number of poles
- **`m`** (*`integer`*) – number of phases
- **`w`** (*`integer`*) – winding step (1 for tooth coils)
- **`layers`** (*`integer`*) – number of coil sides per slot
- **`turns`** (*`integer`*) – number of turns per coil

`get_basic_characteristics` ()
Returns the basic charactericits of the winding as dictionary and a html string

`get_double_linked_leakage` ()
Returns the coefficient of the double linkead leakage flux. This number is a measure of the harmonic content of the MMF in the airgap caused by the winding. As higher the number as higher the harmonics.

Returns **sigma_d** – coefficient of the double linkead leakage flux

Return type float

`get_fundamental_windingfactor` ()
Returns the fundamental winding factors for each phase

Returns **kw** – windings factors, (one entry for each phase)

Return type list

`get_layers` ()
Returns the definition of the winding layout alternative to the 'get_phases' function. For every layer (with the length of the number of slots) there is a sublist which contains the phase-number.

layers[0][0] contains the phase number for first layer and first slot layers[0][1] contains the phase number for first layer and second slot layers[0][Q-1] contains the phase number for first layer and last slot layers[1][0] contains the phase number for second layer and first slot

Returns

- **layers** (*numpy array*) – winding layout
- **slayers** (*numpy array*) – same as 'layers' but as string
- **layers_col** (*numpy array*) – phase colors

`get_notes``()`
Get notes for the winding

Returns **notes** – Some notes

Return type string

`get_num_layers``()`
Returns the number of layers of the actual winding layout

`get_num_phases``()`
Returns the number of phases m

Returns **m** – number of phases

Return type integer

`get_num_polepairs``()`
Returns the number of pole-pairs p

Returns **p** – number of pole-pairs

Return type integer

`get_num_series_turns``()`
Returns the number of turns in series per phase. If the number of coil sides per phase or number of turns per phase is not identically than a mean value of turns of all phases is returned.

Returns **w** – number of turns in series per phase

Return type number

`get_num_slots``()`
Returns the number of slots Q

Returns **Q** – number of slots

Return type integer

`get_phasenames``()`
Returns the names of the phases as a series of characters 'A', 'B', 'C', ... with length of the number of phases

Returns **phasenames** – names of the phases

Return type list

Examples

if there are m = 3 phases: >>> data.get_phasenames() ['A', 'B', 'C']

`get_phases``()`
Returns the definition of the winding layout. For every phase there is a sublist which contains the slot number which are allocated to the phase. phases

phases[0] contains the slot numbers for the first phase phases[1] contains the slot numbers for the second phase phases[m-1] contains the slot numbers for the last phase

Returns **phases** – winding layout

Return type list of lists

get_q()
Returns the number of slots per pole per phase.

Returns **layers** – number of slots per pole per phase

Return type Fraction

get_radial_force_modes(*num_modes=None*)
Returns the radial force modes caused by the winding. The results includes also the modes with a multiple of the phase-number (which aren't there if the machine is star-connected).

Parameters **num_modes** (*integer*) – Max. number of modes. If not given the default value from the config file is used

Returns **MMK** – radial force modes

Return type list

get_text_report()
Returns a winding report.

Returns **report** – Report

Return type string

get_title()
Get the title of the winding

Returns **title** – title

Return type string

get_turns()
Returns the number of turns. If all coil sides has the same number of turns, the return value is a scalar. If every coil side has an individual number of turns, the return value consists of lists with the same shape as the winding layout (phases)

Returns **turns** – number of turns

Return type integer, float or list of lists

get_windingfactor_el()
Returns the windings factors with respect to the electrical ordinal numbers

Returns

- **nu** (*numpy array*) – ordinal numbers
- **kw** (*2D numpy array*) – windings factors, (one column for each phase)

get_windingfactor_mech()
Returns the windings factors with respect to the electrical ordinal numbers

Returns

- **nu** (*numpy array*) – ordinal numbers
- **kw** (*2D numpy array*) – windings factors, (one column for each phase)

get_windingstep()
Returns the winding step

Returns **w** – winding step

Return type integer

`load_from_file` (*fname*, *idx_in_file=0*)

Load data from file.

Parameters **`fname`** (*`string`*) – file name

`plot_MMK` (*filename*, *res=None*, *phase=0*, *show=False*)

Generates a figure of the winding layout

Parameters

- **`filename`** (*`string`*) – file-name with extension to save the figure
- **`res`** (*`list`*) – Resolution for the figure in pixes for x and y direction example: res = [800, 600]
- **`phase`** (*`float`*) – phase angle for the current system in electical degree in the range 0..360°
- **`show`** (*`Bool`*) – If true the window pops up for interactive usage

`plot_layout` (*filename*, *res=None*, *show=False*)

Generates a figure of the winding layout

Parameters

- **`filename`** (*`string`*) – file-name with extension to save the figure
- **`res`** (*`list`*) – Resolution for the figure in pixes for x and y direction example: res = [800, 600]
- **`show`** (*`Bool`*) – If true the window pops up for interactive usage

`plot_star` (*filename*, *res=None*, *ForceX=True*, *show=False*)

Generates a figure of the star voltage phasors

Parameters

- **`filename`** (*`string`*) – file-name with extension to save the figure
- **`res`** (*`list`*) – Resolution for the figure in pixes for x and y direction example: res = [800, 600]
- **`ForceX`** (*`Bool`*) – If true the voltage phasors are rotated in such way, that the resulting phasor of the first phase matches the x-axis
- **`show`** (*`Bool`*) – If true the window pops up for interactive usage

`plot_windingfactor` (*filename*, *res=None*, *mechanical=True*, *show=False*)

Generates a figure of the winding layout

Parameters

- **`filename`** (*`string`*) – file-name with extension to save the figure
- **`m`** (*`list`*) – Resolution for the figure in pixes for x and y direction example: res = [800, 600]
- **`mechanical`** (*`Bool`*) – If true the winding factor is plotted with respect to the mechanical ordinal numbers. If false the electrical ordinal numbers are used
- **`show`** (*`Bool`*) – If true the window pops up for interactive usage

`reset_data` ()

resets all data of the datamodel

reset_results()
Remove all existing results

save_to_file(*fname*)
Saves the data to file.

Parameters **fname** (`string`) – file name

set_machinedata(*Q=None*, *p=None*, *m=None*)
setting the machine data

Parameters

- **Q** (`integer`) – number of slots
- **p** (`integer`) – number of pole pairs
- **m** (`integer`) – number of phases

set_notes(*notes*)
Set additional notes for the winding

Parameters **notes** (`string`) – Some notes

set_num_phases(*m*)
Sets the number of phases m

Parameters **m** (`integer`) – number of phases

set_num_polepairs(*p*)
Sets the number of pole pairs p

Parameters **p** (`integer`) – number of pole pairs

set_num_slots(*Q*)
Sets the number of slots Q

Parameters **Q** (`integer`) – number of slots

set_phases(*S*, *turns=1*, *wstep=None*)
setting the winding layout

Parameters

- **S** (`list of lists`) – winding layout for every phase, for example: S = [[1,-2], [3,-4], [5,-6]]. This means there are 3 phases with phase 1 in in slot 1 and in slot 2 with negativ winding direction. For double layer windings there must be additional lists: S = [[[1, -4], [-3, 6]], [[3, -6], [-5, 2]], [[-2, 5], [4, -1]]] Hint: [[[first layer], [second layer]], ...]
- **wstep** (`integer`) – winding step (slots as unit)

set_title(*title*)
Set the title of the winding

Parameters **title** (`string`) – title

set_turns(*turns*)
Sets the number of turns. If all coil sides has the same number of turns, the parameter should be an scalar. If every coil side has an individual number of turns, the parameter value have to consist of lists with the same shape as the winding layout (phases)

Parameters **turns** (`integer, float or list of lists`) – number of turns

set_windingstep(*w*)
Sets the winding step w

Parameters **w** (*integer*) – winding step

CHAPTER TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX