Purpose of the document

This document contains minimal bullet-point style documentation to the free open-source Matlab library SMEKlib.

For more theoretical background information, please check out the ICEM'18 paper 'A High-Performance Open-Source Finite Element Analysis Library for Magnetics in MATLAB'.

What is SMEKlib?

SMEKlib, named after the Finnish word for electromechanics *SähköMEKaniikka*, is an open-source Matlab library primarily tailored for analysing and designing rotating electrical machines.

Originally, SMEKlib was developed in the Aalto University Research Group of Electromechanics, between 2013 and 2019. Since March 2019, it is primarily supported by SMEKlab Ltd, a boutique motor design company based in Finland.

Who can use SMEKlib?

SMEKlib is published under a permissive open-source license called MIT license. It is free to use for both academic and industrial work, as long as the license conditions are followed.

Many SMEKlib examples also use the open-source mesh generator *gmsh*. However, *gmsh* is a separate software published under its own license. It is not coupled with SMEKlib or vice versa, and any SMEKlib user can opt to use any other mesh generator of their choice.

Where to get support?

Primarily, please first check the examples under the repository. It is quite likely that whatever you need is already hidden in there. Furthermore, do also check the (somewhat lagging) documentation of the main functions used in the examples.

For small questions, you can reach out to antti@smeklab.com.

For more involved support or custom projects, please see section 'Industrial use'.

Important bullet points

This section describes some important aspects related to the *recommended* way of performing electromagnetic analysis on rotating electrical machines. However, it is not the *only* way – SMEKlib is a library rather than a design software, after all. Furthermore, other types of analysis (like mechanical and thermal) are somewhat less mature and require a more hands-on approach. Finally, rotating machines *can* also be analysed in ways other than described here – it only requires more usage of lower-level matrix assembly functions and similar.

For the most complete, most recent example, see 'Induction Motor (37 kW complete example)' in the Examples folder.

Meshing and pre-processing

Finite element meshes are defined with the following arrays:

- p : 2 x number-of-nodes array, defining the x- and y-coordinates of the mesh nodes. Each column defines one node.
- t : 3 x number-of-elements array, defining the (integer) indices of the three nodes defining each element. Each column defines one element.
    - For second-order elements, the size of the array is 6 x number-of-elements.

Optionally, the following arrays are often computed automatically:

- e : 2 x number-of-edges array, defining all edges in the mesh. Each column defines an edge, by the (integer) indices of the nodes making up each edge.
- t2e : 3 x number-of-elements array, defining which edges border each element.
- e2t: 2 x number-of-edges array, defining which elements border each edge.

Mesh objects are the recommended way of using meshes, although some older functions also support structures with .p and .t fields. The most commonly used mesh objects and their initialization calls are

- msh = MachineMesh(p, t, matel) : mesh object for rotating machines. See below for *matel*. Normally, the airgap is NOT included in p or t. See below.
- msh = SimpleMesh(p, t) : Simple mesh object type for simple problems.

Important things to consider about the p and t arrays:

- If a symmetry sector (and not entire cross-section) is analysed, the clock-wise boundary of both stator and rotor should lie on the x-axis.
- Airgap should not be meshed in advance.

The *MachineMesh* objects then have the following important characteristics, methods, etc:

- e, t2e, and e2t arrays are computed automatically after initialization.
- matel : a 1 x number-of-elements array specifying the index of material making up each elements. See *help get_defaultMaterials*
- symmetrySectors : number of symmetry sectors, only one of which is simulated
- periodicityCoeff : periodicity coefficient between symmetry sectors. See below.
- namedNodes : a container array for important nodes, see below.
- namedElements : a container array for important elements, see below.
- generateMovingBand() : a method for automatically generating a two-layer airgap triangulation for movement modelling with the moving band method:

The important, aforementioned "named nodes" then include the following:

- "Dirichlet" : set with msh.namedNodes.set("Dirichlet", indices_of_nodes) : specifies that the nodes listed in the 1 x N array indices_of_nodes belong to a homogeneous Dirichlet boundary = magnetic insulation = typically outer yoke side of stator.
- "Periodic_master" and "Periodic_slave" : nodes belonging to the periodic boundaries of a symmetry sector. Typically, the clockwise boundary is used as the master side. There must be a 1-on-1 correspondence between the *radial* coordinate of each master-slave node pair. The vector potential solution on the slave-side boundary is then msh.periodicityCoeff times that of the master side.
- "n_ag_s" and "n_ag_r" : nodes belonging to the stator and rotor side of the airgap boundaries.

Important named elements include:

- rotel : rotor elements, used in plotting motion. Plain array.
- statorConductors : cell array, with each cell containing the elements (as indices to the columns of t) belonging to each stator conductor. Normally, when the stator winding is modelled as infinitely stranded,

a "conductor" means either the winding area of one stator slot, or one winding layer of one slot in a two-layer winding.
- rotorConductors : cell array of rotor conductors, if any.
- PMs : 2 x N cell array, where each column defines a permanent magnet. One row lists the elements belonging to each magnet, while the other specifies the remanence flux density vector for each element in the PM.

Generating a mesh with gmsh

SMEKlib contains a limited-functionality wrapper to *gmsh* called *gwrap,* based on file-IO. The following bullet points summarize the core philosophy of it, after which it is best learned from the examples.
- Only surfaces are defined; and not points and lineloops etc.
- Surfaces are best defined piecewise with the addPcws method
    - As in, the curve bounding a surface from the outside is defined in pieces
    - Supported pieces include straight lines and circle arcs less than 180 degrees
    - Pieces are defined with point 2D coordinates (and possible angles if needes)
    - The *tolerance* of each piece defines the maximum edge length on it. Use negative integer to fix (recommended for periodic boundaries).
    - Each piece can be named to extract the edges belonging on it from the resulting mesh.
    - Each surface is named. Holes in a surface must be be other surfaces already defined.
- The mesh is then returned as the p and t arrays, as well as a container object listing named surfaces and edges.

Simulation

Two important classes are related to the actual processing: MachineSimulation and SimulationParameters.

SimulationParameters *pars* is initialized with "key"-value pairs, the most common of which include:
- U : line-to-line supply voltage. Default 400. Can be a function handle for time-stepping analysis.
- f : supply frequency. Default 50.
- slip : slip
- isDC : true/false if the machine is synchronous/asynchronous. Affects harmonic analysis. Default false.
- N_periods : number of supply periods to analyse. Can be fractional.
- N_stepsPerPeriod : number of time-steps per period. Integer.
- maxIter : maximum number of nonlinear iterations. Default 30.
- silent : true/false, print simulation progress to Matlab command prompt. Default true.

Uninitialized parameters are set to their default values. See class implementation for all parameters and defaults.

The class has one important method
- pars.ts() : returns a 1xN array of times used for time-stepping.

A MachineSimulation *simc* is then initialized with the MachineMesh object and a structure *dims* of the most important motor dimensions (see below). The important methods then include
- simc.run_harmonic(pars) : runs harmonic analysis. Works for both async and syn motors. See pars.isDC.
- simc.run_static(pars) : runs static analysis with current supply. Phase currents are specified in pars.U, and rotor angles in pars.rotorAngles.
- simc.run_stepping(pars) : run time-stepping analysis.
- simc.fluxplot(step, pars) : plot flux density and flux lines at *step*. Use -1 as step for harmonic, or
- simc.fluxplot(step, pars, type) : type can be 'harmonic', 'static', or 'stepping'.

- simc.Is : phase currents from time-stepping, for the simulated symmetry sector only (if dims.a > 1, phase currents for the actual motor are dims.a*simc.Is). Note: **not** line currents.
- simc.Ish : phase current phasors from harmonic analysis.

The class also contains some important properties, described next.

simc.matrices: a struct containing e.g.
- Ms, Mr : stator and rotor mass matrices
- Cs, Cr : circuit coupling matrices. See the conference paper linked earlier.
- Ls, Lr : loop matrices for circuits
- Zew_s, Zew_r : end-winding impedance matrices

simc.results : a struct containing
- Xh : results from harmonic analysis
- Xs : same for static analysis
- Xt : time-stepping

msh : MachineMesh object, reference.
nu_struct, nu_fun : for evaluating nonlinear material BH behaviour. Overwrite nu_fun to supply your own.

The dims structure can include e.g. the following fields:
- D_so : outer diameter of stator. Only used in flux plotting.
- p : number of pole-pairs
- N_series : number of turns per each single coil (each consisting of a pair of slots).
- a : number of parallel paths
- c : amount of coil short-pitching, in slot pitches
- N_layers : number of winding layers.

Post-processing

Some important post-processing tools include
- fluxplot, Is, Ish method of MachineSimulation
- ts method of SimulationParameters
- sim_compute_torque(simc, pars, type) : compute torque with the weighted Maxwell stress tensor (eggshell) method, evaluated over the non-distorting airgap layer. *type* as with *fluxplot*.
- sim_IronLosses (simc, pars) : compute iron losses with the Bertotti method, from the last el. period of time-stepping results. Loss coefficients are obtained from *get_defaultMaterials*.
- sim_compute_CageLosses (simc, pars) : compute eddy-current losses in rotor cage. Can easily be modified to compute losses in any solid conductor.
- Plain Matlab scripting : for computing DC copper losses in stranded stator winding or end-winding, power factors, efficiencies, THDs, etc. You are an engineer, after all :)

Industrial use

As mentioned earlier, SMEKlib is completely free for industrial use, provided that the license conditions are followed. That been said, the Github version (that this manual is for) is no longer actively developed. Instead, the following options are recommended for professional users:

SMEKlib Pro

- A professional, actively developed extension to the free SMEKlib.
- More motor types easily.
- More materials, more extensive loss analysis.
- Graphical user-interfaces.
- License includes full editing rights to source code.
- *Pricing: 2500 EUR / 5 users / year OR as negotiated.*

SMEKlib Priority Support

- For custom industrial projects based on the open-source version of SMEKlib.
- Perfect for smaller, flexible projects.
- Includes motor design support.
- *Pricing: 100 EUR / hour.*

SNEKlib (SMEKlib in Python)

- Python version of SMEKlib Pro
- No Matlab licenses required
- Ongoing project - can be tailored *for you*.
- *Pricing: as with SMEKlib Pro.*