Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electric Power Engineering

Jostein Hovde Aarvoll

# The development of calcGenProg and GenProgApp

## Visualization and graphical user interface for design of salient pole generator

Master's thesis in Energy and the Environment
Supervisor: Arne Nysveen
June 2021

NTNU
Norwegian University of
Science and Technology

Jostein Hovde Aarvoll

# The development of calcGenProg and GenProgApp

## Visualization and graphical user interface for design of salient pole generator

Master's thesis in Energy and the Environment
Supervisor: Arne Nysveen
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electric Power Engineering

NTNU
Norwegian University of
Science and Technology

NTNU
Kunnskap for en bedre verden

DEPARTMENT OF ELECTRIC POWER ENGINEERING

VISUALIZATION AND GRAPHICAL USER INTERFACE FOR DESIGN OF SALIENT POLE GENERATOR

---

# The development of *calc*GenProg and GenProg*App*

---

*Author:*
Jostein Hovde Aarvoll

*Supervisor:*
Arne Nysveen

June, 2021

# Abstract

This report covers the working principals, and development of the *improved* version of GenProg first created by Alexander Lundseng and Ivar Vikan in 2010. The work described in this report can be considered a direct sequel to the work performed the previous semester, where the objective was to create a visual representation of the calculated generator, and create a *Graphical User Interface* for GenProg. This initial work uncovered several shortcomings of the underlying function GenProg. The shortcomings are explained and the rework is presented, but only one is presently *implemented.*

A system of input sanitation and control of the calculated parameters was developed with an accompanying error message report for the user.

The cross sectional view and graphical user interface initially developed the previous semester was further improved upon. Such as a complete vectorization of the cross-sectional image, and the option to only render specific segments of the generator.Several *under the hood* improvements to the GUI was also implemented.

The result is presented with a complete design example for a salient pole generator, and an discussion with what was accomplished, and what work lies ahead. Both immediately and subsequently there after. The project did resolve many of the issues related to GenProg, and provides a strong foundation for future development.

# Contents

# List of Figures

# List of Tables

# Acknowledgement

The project would not have been possible without the help of my supervisor, Arne Nysveen, who gave me the freedom to work on, and take the project in a direction where my skills could be best utilized. I must also thank my father, Hans Christian Hovde, a HVAC engineer who helped with some of the theory covered in this report. Finally I would like to thank Pia for giving me the support I needed to continue working despite the challenges I faced.

# 1 Introduction

**Background and motivation** GenProg is a script developed over several years by students doing their master thesis at NTNU. Initially I was tasked with creating a *graphical user interface* and a *visual representation* of the calculated machine. Supervisor desired to implement a FEM analysis, but due to my limited experience with such software the effort shifted towards a *simple* cross-section. As the project progressed it became apparent that a significant effort had to be made towards analyzing and understanding the underlying source code for *GenProg.* Beyond what was provided by accompanying documentation, and comments in the source code.

**Objectives** The goal of the project was to further improve the GenProg application developed in the Autumn of 2020. From the preceding iteration the overall user experience needed to be improved, and the *core* script GenProg needed a major overhaul. More specifically the tasks are as follows:

- Fix bugs and issues related to the *old* GenProg script.
- Streamline and optimize the core *GenProg* function.
- Comprehensive input sanitation.
- Create a robust framework for performing a control of the calculated parameters, and display this to the user.
- Increase the *fidelity* of the cross-sectional view by means of vectorization.
- Add new functionality to the cross-sectional application.
- Develop the app for use as an educational tool, and lay the ground work for future development.

# 2 Nomenclature

- *old*GenProg - The core script GenProg first developed by Lundseng and Vikaan in 2010.
- *calc*GenProg - restructured and improved variant of GenProg developed for this project. Also sometimes refered to simply as *the improved version*
- *Initial* or *previous* GenProg refers to the adapted version of GenProg developed as an interim solution for the previous semesters project. Generally identical to the core script GenProg, but converted to a *function.*
- *Previous semester project* - Semester project by the author of this report. Provides much of the groundwork for this *master project.*
- *GenProgApp* or just *app* refers to the the graphical user elements of GenProg. (Although technically not an *app*, the GUI was created using MATLABs *app designer* environment and the name stuck.)
- *GUI* - Graphical User Interface
- *Vector* - Unless specified, refers to a 1 dimensional matrix array containing numeric or Boolean values. Usually a row-vector, but colomn vectors do occur.
- *Required parameters* - Refers to the 1-dimensional vector containing all the *required parameters* for calcGenProg. Abbreviated simply as *req* or *req_* in the source code and documentation.
- *Optional parameters* - Referes to the 1-dimensional vector containing all the *optional parameters* for calcGenProg. Abbreviated as *opt* or *opt_* in code form.
- *Object* - Refers to a *logical* geometric object represented by an x and y vector of boundary positions for use in the cross-sectional view for the calculated generator. Can be considered a *polygon.*
- *Typecheck* - Refers to both the *input sanitation* and *output validation* of calcGenProg parameters. *Resolution* - In the context of the cross sectional view the term *resolution* refers to the number of positions on a curved object unless otherwise specified.
- *var* - Variable. Usually an intermediate, or calculated parameter.
- *const* - Constant. A variable that does not change.
- *bool* - Boolean data type. Logical true or false.
- *mat* - Matrix. 2-Dimensional matrix array.
- *vec* - Vector. 1-Dimensional matrix array.
- *Source code* - The *actual* MATLAB code.

# 3 Theory

## 3.1 Detailed description GenProg

The core script *GenProg* is based of an e*xample of design procedure* written by prof E. Westgaard dated 1964 [5]. This compendium describes the design of a typical salient pole generator ranging from 10 to 50 MVA. The document gives a detailed walk-through of a typical design procedure based of a few key parameters obtained from the customers needs and returns every parameter used to describe a complete generator. The procedure was then adapted into a set of formulas in their semester project [4] by Ivar Vikan and Alexander Lundseng. This set of formulas was then used for their master thesis the following semester [3]. Since then numerous students have used, and added functionality to the script to suit their needs. However it is assumed the *core* script remains in almost the exact same state as when it was first completed almost 10 years ago. A considerable effort has been made to *understand* the core script in order to adapt it for use in a graphical user interface, and improve its inner workings.

### 3.1.1 GenProg Summary

In this section a brief summary of the GenProg script will be described. The script works of a *Input file* usually labeled *filename_Input.xls.* Inside this Excel worksheet the user can define a set of 24 *required* and up to 42 *optional* parameters. These parameters are then loaded into the MATLAB script and over 2500 lines of code calculates a total of 111 parameters which is then written to a second Excel worksheet titled *Output.xls.* From here the user can read the output data, and use them for whatever purpose they wish. The working principal can be divided into several stages. The actual workings of each *stage* can be quite different depending on which optional parameters are defined by the user.

### 3.1.2 Stage 1 - Stator Calculations

The script starts by working out the basic dimensions of the stator. Depending on what parameters the user has defined, this stage calculates the inner diameter of the stator iron, Number of slots, nominal current and voltage as well as many other parameters related to the external dimensions of the stator. Stage 1 of the script can be considered the most important as most of the other stages, and sub-stages, can be directly traced back to parameters calculated here. The inner diameter is used for determining almost all other geometric dimensions in the generator, and the currents and voltages are used for determining magnetic parameters which again are used for a plethora of other parameters.

**Slot Dimensions and Armature Calculations** After the *external* stator dimensions are determined the script begins calculating the slot dimensions. The script differentiates weather *number of turns per coil* is equal to 1 or not. If TNR is equal to one it is assumed the armature is of *roebel strand winding type* and if not, it is assumed to be of *form winding type.* The user can define all the slot dimensions themselves, but the script can also use the *target current density stator* parameter for determining the appropriate slot dimensions.

**More Armature Calculations** When the slot and armature parameters are determined in the previous sub-stage, the script can calculate the remaining stator dimensions such as outer diameter and *total copper cross section area.* It is at this sub-stage the *air gap flux density* can be calculated which is a key parameter for further calculations. In addition to the magnetic parameter it is now possible to calculate the total resistance for the stator which is used later on to determine losses.

### 3.1.3 Stage 2 - Rotor calculations

After the stator calculations are completed the script begins calculating rotor parameters. The rotor is assumed to be of *round rotor* with a uniform air gap in order to simplify some of the calculations [5]. If the pole dimensions are not set by the user under *optional pole dimensions*, an initial pole is calculated based on functions from the previous stator calculations. After the initial rotor pole parameters are found the script calculates the majority of the magnetic parameters surrounding the entire generator. Particularly the flux density in stator and the (initial) pole core as well as in the rotor and stator yoke. As these are the basis for calculating the field parameters.

**Field winding** After the *Required magnetization* has been calculated the field winding dimensions can be determined. If the dimensions, and number of field winding's has not been defined by the user, the parameters are calculated based of the *number required ampere turns*, and *target field current density.* If the values for flux density in the pole core exceeds allowable limits, the script increases the pole core width and then runs the same calculations for a new set of parameters. This is repeated until a stop requirement is reached. The same method is applied if there is not enough vertical (or horizontal) space available for the field winding. This is the same method described in Westgaards compendium [5].

After the pole dimensions and field winding parameters are finalized, the script has completed all the geometric parameters needed to visualize the generator.

### 3.1.4 Stage 3 - Loss calculations

The next stage calculates all the losses in the machine. From DC losses in stator and rotor, but also AC losses, Iron losses, and magnetization losses.

### 3.1.5 Stage 4 - Thermal calculations

After all the losses has been calculated the thermals for the machine can be calculated. For the sake of this report it is assumed this section works correctly and no further effort has been made trying to understand its principals.

### 3.1.6 Stage 5 - Reactances and Time Constants

The machines equivalent circuit is used to calculate the sub-transient, transient and stationary reactances and accompanying time contsants.

### 3.1.7 Stage 6 - Mechanical calculations

The final stage is a very simple calculation to determine the total weight of the entire generator as well as its rotating inertia.

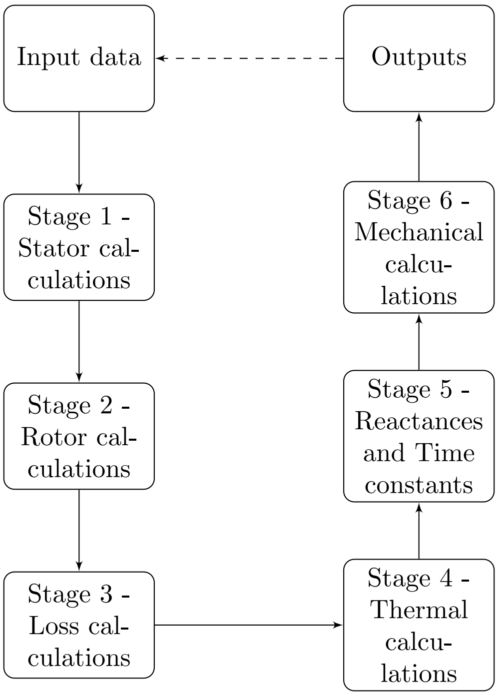


Figure 1: Flowchart for *GenProg*

## 3.2 Tensile strength of steel

The calculation for tensile strength for starts with the *von Mises yeld criterion* which can be expressed mathematically as follows:

$$J_2 = k^2$$

where **k** is the yield stress of the material in pure *shear*. The *magnitude* of the shear yield in stress in pure shear is $\sqrt{3}$ times lower than the tensile stress in simple tension case. This allows for:

$$k = \frac{\sigma_y}{\sqrt{3}}$$

$\sigma_y$ is the tenisel yeild strength of the material. If we set the von Mises stress equal to the yield strength and combine the above equations, the von Mises yesld criterion can be expressed as [7]:

$$\sigma_v = \sigma_y = \sqrt{3J_2}$$

or

$$\sigma_v^2 = 3J_2 = 3k^2$$

Substituting $J_2$ with the *Cauchy stress tensor* components gives:

$$\sigma_v^2 = \frac{1}{2}[(\sigma_{xx} - \sigma_{yy})^2 + (\sigma_{yy} - \sigma_{zz})^2 + (\sigma_{zz} - \sigma_{xx})^2 + 6(\sigma_{yz}^2 + \sigma_{zx}^2 + \sigma_{xy}^2)] \tag{1}$$

Further proof is beyond the scope of this report (and beyond what can be expected by a electrical engineer). For the moment it can best be described as the yield strength of *ductile materials*, such as steel, when its *second invariant of deviatronic stress* reaches a critical value. I.E when $\sigma_v = \sigma_y$ [7].

For the purpose of this project only the 2-dimensional components can be considered I.e $\sigma_{zz} = 0$, $\sigma_{yz} = \sigma_{xz} = 0$. Solving equation 1 for $\sigma_v$ gives equation 2:

$$\sigma_v = \sqrt{\sigma_x^2 + \sigma_y^2 - \sigma_x \sigma_y + 3\sigma_{xy}^2} \tag{2}$$

Equation 2 gives an expression for the tensile stress required to permanently deform the ductile material.

## 3.3 Dynamic pressure

The dynamic pressure is the *kinetic energy* of a flowing fluid, liquid or gas [6] and can be described by equation 3.

$$p = \frac{1}{2}\rho v^2 \ [Pa] \tag{3}$$

and the *hydraulic power* is defined by pressure [Pa] times flow $[m^3/s]$ :

$$P_h = p\ q\ [W] \tag{4}$$

# 4 Preceding work

## 4.1 Introduction

The preliminary work for this master project was completed over a 6 week period the preceding semester (Autumn 2020). During this time the initial *App* was created, and system of creating a cross sectional view was developed. Due to the nature of the previous semesters work, and its immediate continuation for this semesters master project, a condensed version of the semester project *report* is presented in section 4.

## 4.2 Cross-sectional view

The script created a matrix array for the object, or set of objects, and displayed them for the user. All calculations was performed *symbolically*, meaning all the geometric entities are calculated from the result of GenProg, and not *hard-coded.* The cross-section script generates a cross-sectional image for *any* arbitrary GenProg calculated machine. The Script can be divided into 7 distinct categories:

- Stator section
  - Stator *ring*, defined by the inner and outer diameter
  - Stator slots, wedges, and separators. Consisting mostly of simple rectangles.
  - Armature winding strands
- Rotor section
  - Pole core
  - Pole shoe
  - Field winding
  - Damper bar

### 4.2.1 Stator ring

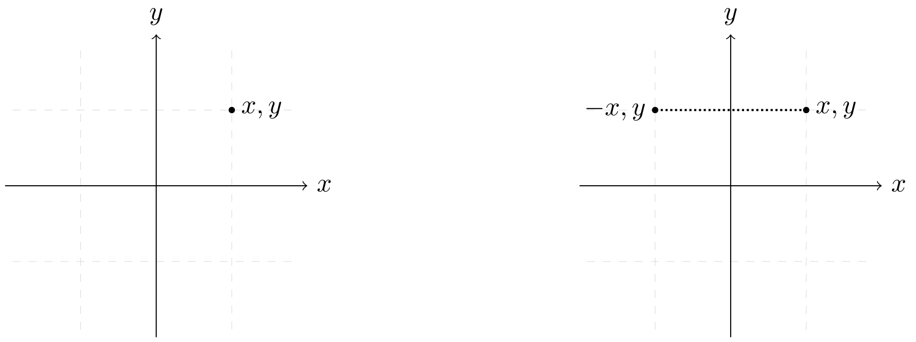
Initially the stator was created by creating a linear x vector ranging from 0 to *Rsi*, and a second x-vector ranging from 0 to *Rsy.* Inner and outer radius for the stator iron respectively. From this x-value vector the corresponding y-value vector could be easily calculated by using trigonometrical formulae. see figure 3 and4.

### 4.2.2 Stator slot

The advantage of the *old* system of *drawing on a canvas* approach made it possible to simply *paint over* the undesired elements. Meaning the slots could simply be painted over the stator ring with a simple square extending a little bit beyond the inner edge, and colored the same as the background (black). As with the other miscellaneous slot elements like the slot wedges, separators, and insulation material, an initial object was created, and rotated around the origin of the generator.

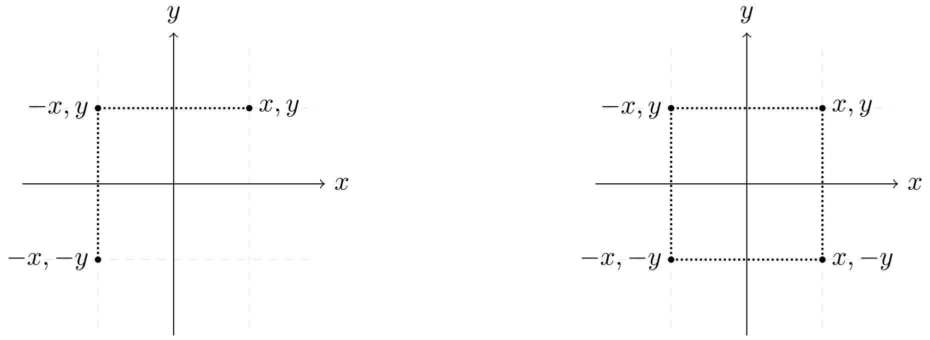
### 4.2.3 Armature winding

The armature winding matrix was created by starting with a *zero* winding. Experiments where conducted in order to create a *rounding* in the corners of each individual winding. GenProg assumes



(a) First corner

(b) Second corner



(c) Third corner

(d) Fourth corner

Figure 2: Example of how a simple rectangle can be created for use in a cross sectional view. *from preceding project report*

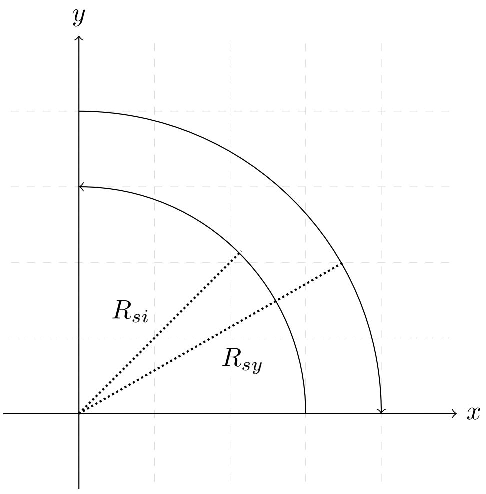


Figure 3: Visual representation of how the stator ring is created *from preceding project report*

a 2% rounding factor. Meaning 2% of the cross-sectional area is *lost* to rounding of the corners. The same method used to create the stator ring, was used to create the rounded corners. See figure 4. The *zero winding* could then be duplicated for each strand per bar, until the *initial* slot was filled. Then each strand was duplicated and rotated into each of the remaining slots. Only the Roebbel strand, and a incorrectly labeled *single colomn strand*-type was developed. Both of these armature winding layouts are pretty much deprecated and is not present in the current iteration of the cross sectional function. The initial cross-sectional script system did not work for f*orm-winding* or $TNR \neq 1$.

### 4.2.4 Pole Core and Pole shoe

The pole *core* was the created using 8 (then reduced to 6 with the inclusion of the pole shoe) positions. See figure 5 for visual reference. A recurring *anchor-point* being the inner radius of the stator, and the minimum air gap. All dimensions are calculated relative to this position. Two different pole shoe types was included:

- NEBB type
- ASEA type

**NEBB** *Single radius.* The Pole shoe shape consists of a arc with a single radius from each end of the pole core. The arc radius has its origin off-centre from the rest of the generator. See figure 6b for exaggerated geometry.

**ASEA** The ASEA type pole shoe differs from NEBB in that it utelizes the same radius for the arc as for the stator iron, but truncates before the edge of the pole shoe is reached. See figure 6a for visual representation of the geometry with important ratios included.

### 4.2.5 Damper bars

The damper bars, and the *damper slots* was placed along the pole shoe edge with a edgedistance equal to 3mm. The slot pitch $\tau_s$ was *translated* to the pole shoe surface, and the script tried to place the damper bars equal to 0.8, or 1.2 times the *translated* slot pitch. If the number of damper bars exceeded the available space defined by 0.8 times $\tau_s$, they where evenly crammed in there without regards to the slot pitch.

The damper bar *slots* are placed perpendicular to the tangent of the pole shoe edge, however for the case of ASEA, the *edge* is assumed to extend all the way to the pole shoe edge. The consequence being that the damper bars in the extreme position aren't placed exactly 3mm from the pole shoe edge. In some cases the damper bars can protrude beyond the edge of the pole shoe.

### 4.2.6 Rendering

After the object matrices was generated, they could be fed to a *insert shape* function together with a color vector which in turn *painted* the objects within a $n \times n \times 3$ RGB matrix. It should be noted that this process was multiple order of magnitude slower than the *actual* object creation itself.

### 4.2.7 Result

The result of the cross-sectional *endeavour* can appear crude and slow. However the *underlying* matrices proved a robust and precise method of representing geometric objects. Only minor changes was necessary to use the object matrices in the current *vectorized cross-sectional view variant.* Most of the adaptations taking place *outside* the matrix creation. An example can be viewed in figure 7

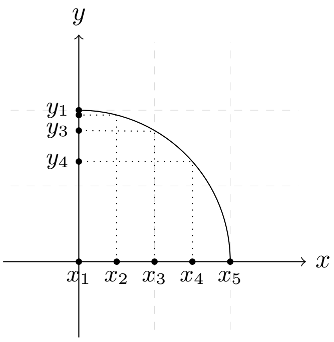


Figure 4: One quarter of a circle polygon with evenly spaced x-positions, and their corresponding y-positions. *from preceding project report*

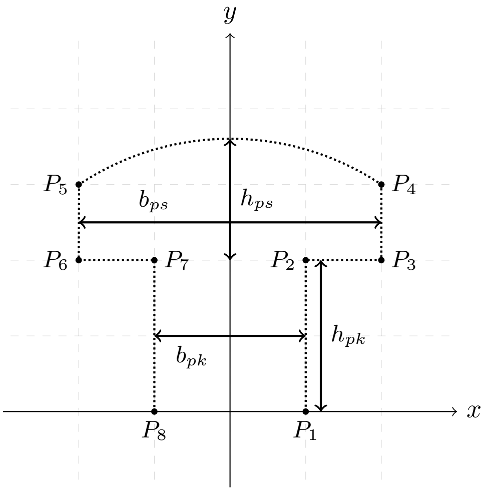
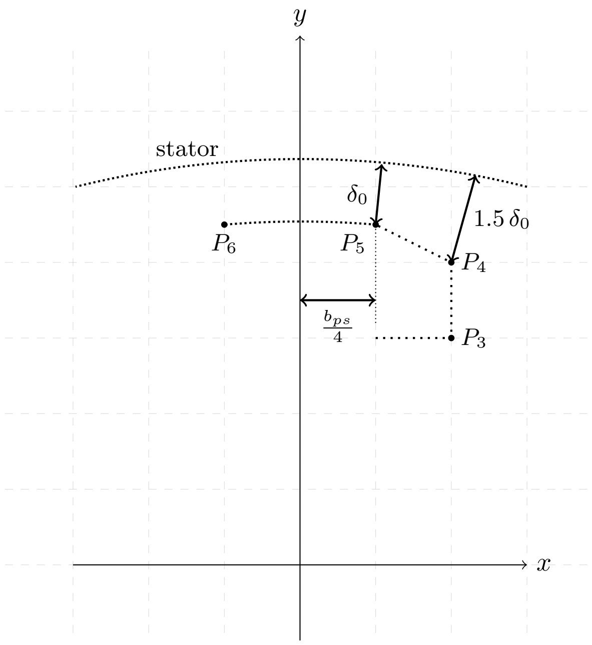
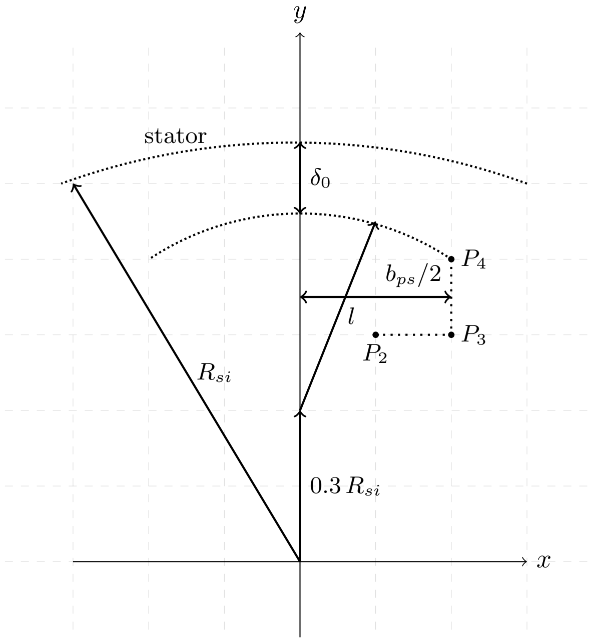


Figure 5: Visual representation of a basic pole core *from preceding project report*



(a) Exaggerated visual representation of the ASEA pole shoe shape. *from preceding project report*



(b) Exaggerated visual representation of the NEBB single radius pole shoe shape. *from preceding project report*

Figure 6: Exagerated visualization of the two different pole shoe shapes initially implemented

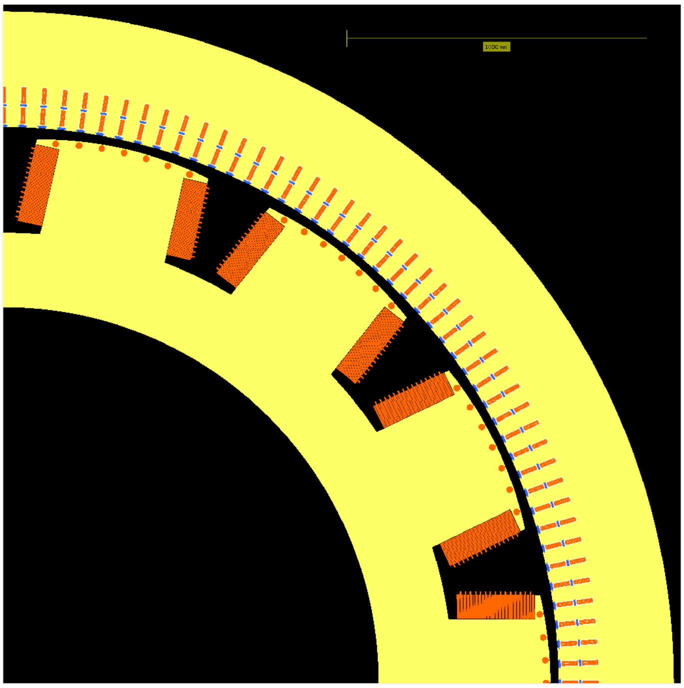


Figure 7: Cross-sectional view created during preceding project. *from preceding project report*

## 4.3 GUI

The initial *draft* for a *Graphical user interface* was created during the aforementioned 6 week period. The layout was a straight forward interpretation of the Excel worksheet utilized by Lundseng and Vikan [3]. The GUI elements was created using MATLAB's *app designer* environment.

## 4.4 GenProg

Particular emphasis was placed on *not* altering GenProgs functionality during the preceding semester project, in order to not *break* the workings of the script. If a bug was introduced at this stage it was deemed not likely to be rectified in a timely manner. The script was simply altered with a function handle that could be called upon, rather then read from the Excel worksheet. The GenProg script was assumed to work perfectly at this stage.

# 5 Improved GenProg

## 5.1 GenProg restructure

### 5.1.1 Background

The main issue with GenProg in its previous iteration was the sheer size of the script. It consisted of approximately 2500 lines of code contained in a single file. This in itself is not a problem, but when a user tries to run a calculation with a set of input parameters, and the calculation returns completely unreasonable results (or don't even run at all), the task of figuring out exactly where the culprit is located is extremely tedious. Added to this was the fact that several *segments* of code was seemingly copy-pasted to more than one location which made reading the code confusing as the lines repeated itself. It should be noted that the original source code has excellent variable names, so when comments are not present, it is still possible to deduce what the variable is supposed to be. However the script needed a major restructuring in order to facilitate future work, and make the script more *readable*.

### 5.1.2 Method

The entire GenProg script was thoroughly read from start to finish and compared to the original design procedure presented by Westgaard [5]. The script was then divided into several smaller independent functions. This served three purposes:

1. Give the author of this report a better understanding of how GenProg is *supposed* to work.
2. Make the GenProg code more comprehensible for later users.
3. The independent functions would facilitate implementation of more advanced features.

In addition the nomenclature was changed where applicable for the *input parameters*. Previously the *input parameters* where initially read from an Excel worksheet, and was then henceforth called

```
inputvarxls
```

Since the *improved version* does not use Excel, new nomenclature indicates a *input parameter* as:

```
inputvar_
```

Please note that the use of an underscore can be used on a per-function basis in addition to the *input parameters* passed to the parent function.

### 5.1.3 Improvements and corrected deficiencies

In addition to the above mentioned procedure several syntax errors was corrected as they where discovered. Ranging from mundane to potentially critical.

Some honorable mentions include:

- Inocorrect syntax for if-statements. Particularly for if-statements where a variable is checked whether it is within a specific range. The correct syntax is:

  ```
  if var > lower limit && var < upper limit
  ```

  rather than:

```
lower limit < var < upper limit
```

which was the case for several sections.

- While-loops was also used erroneously. Although not frequently used they never had a way to exit the while-loop, and had the potential to never enter them in the first place. The correct discipline is to also include a *iteration limit*, and set the condition for the while-loop as a *constant* rather than a function *outside* the while loop. Se example below.

  ```
  var = 1;
  it = 0;
  while var > 0.1 && it < 100
      %code here...
      it = it + 1;
  end
  ```

  This ensures that the while-loop is initiated, and it can be terminated after a set number of iterations. (in this case 100)

- Removed unnecessary *disp* commands to avoid needlessly cluttering the command window.

### 5.1.4 Result of restructure, *calc*GenProg

The the core script *GenProg* was divided into 23 different functions, and child functions. Each functions is entirely self-contained, and is only dependent on the arguments passed to it from a script or parent function. See figure 8, 9, and 10 for complete working principle of the core script *calcGenProg*. In addition to the aforementioned figures, table 1 provides a list of all 23 functions and a short description

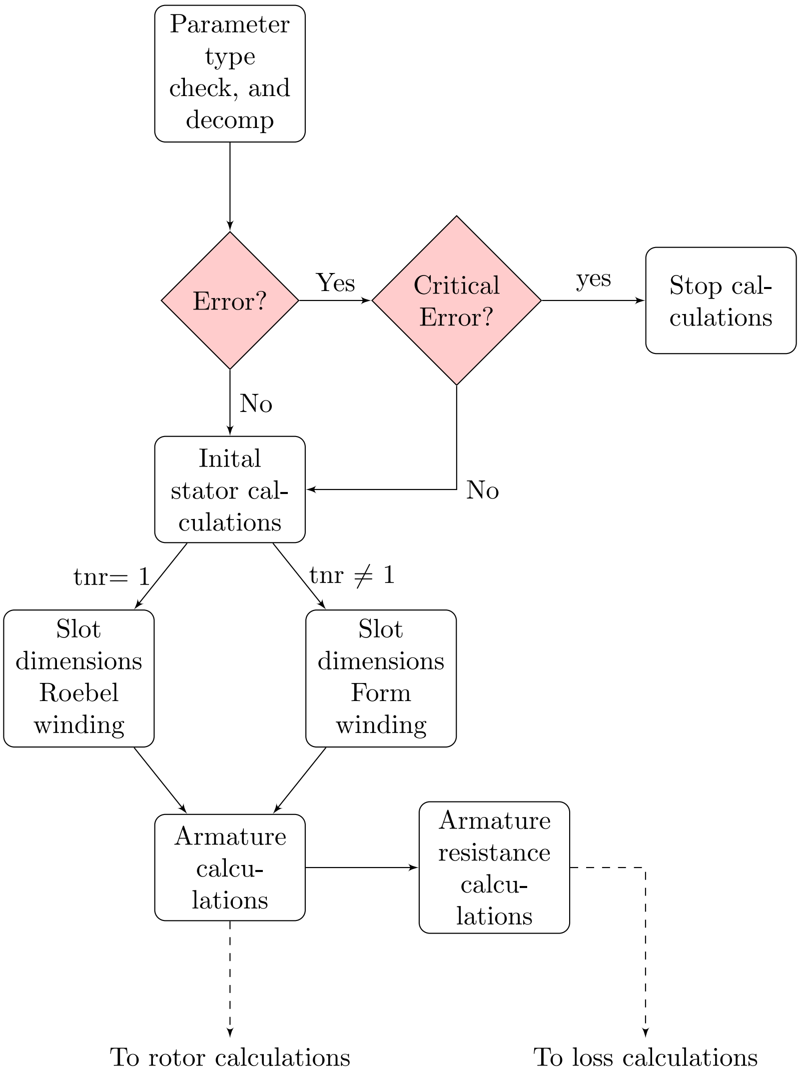


Figure 8: GenProg stator calculations and initial input sanitation

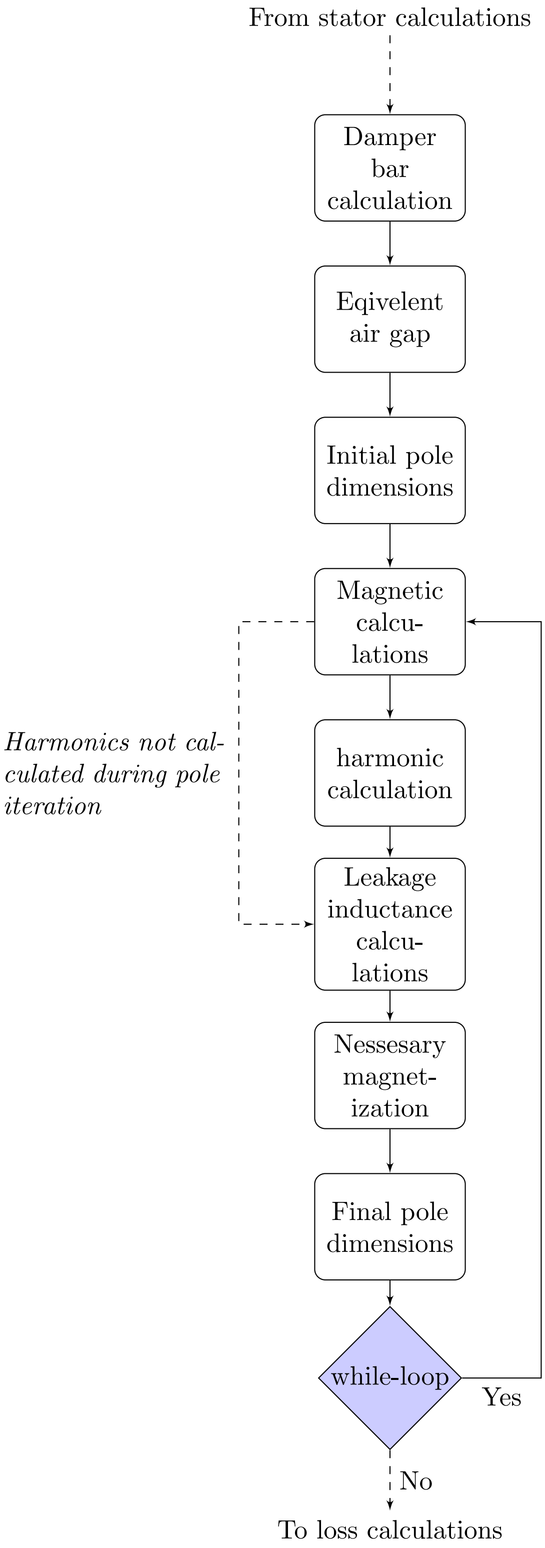


Figure 9: GenProg rotor calculations

Table 1: complete list of the core calcGenProg functions and child-functions.

| Function file name | Description | Parent function: |
|---|---|---|
| *calcGenprog.m* | Parent function containing all the other main function | *N/A* |
| *calcStator.m* | Function for returning important stator parameters such as inner diameter and armature currentsand number of slots. | *calcGenProg.m* |
| *calcC.m* | Function for calculating stator parameters if generator voltage is not set by user. | *calcStator.m* |
| *clacSlots.m* | Function for generating list of possible slot combinations if not user defined | *calcStator.m and calcC.m* |
| *diagSlot.m* | Function for displaying list of possible slot combinations, and prompting user to choose one | *calcStator.m and calcC.m* |
| *calcSlotDimTNReq1.m* | Function for returning slot dimensions for roebbel coil armature | *calcGenProg.m* |
| *calcSlotDimTNRnoteq1.m* | Function for returning slot dimensions for form coil armature | *calcGenProg.m* |
| *calcArm.m* | Function for returning remaining stator calculations | *calcGenProg.m* |
| *calcAirgap.m* | Function for returning minimum air gap | *calcArm.m* |
| *calcRac.m* | Stator resistance calculation | *calcGenProg.m* |
| *calcDamp.m* | Function for calculating damper bar parameters | *calcGenProg.m* |
| *calcCarter.m* | Function for calculating Carters coefficient | *calcGenProg.m* |
| *calcPoleDim.m* | Function for calculating initial pole dimensions | *calcGenProg.m* |
| *calcMagnetic.m* | Function for calculating magnetic parameters | *calcGenProg.m and calcFieldDim.m* |
| *calcMagneticUdrop.m* | Function for calculating magnetic voltage drop | *calcMagnetic.m* |
| *calcHarm.m* | Function for calculating harmonics in the machine | *calcGenProg.m* |
| *calcInd.m* | Function for calculating inductance parameters | *calcGenProg.m and calcFieldDim.m* |
| *calcMagNes.m* | Function for calculating nessesary magnetization | *calcGenProg.m and calcFieldDim.m* |
| *calcFieldDim.m* | Function for calcaulating field winding dimensions, and final pole dimensions | *calcGenProg.m* |
| *calcLoss.m* | Function for calculating losses in the machine | *calcGenProg.m* |
| *calcThermal.m* | Function for calculating thermal parameters for the machine | *calcGenProg.m* |
| *calcReacTime.m* | Function for calculating the machines reactances and time constants | *calcGenProg.m* |
| *calcMech.m* | Function for calculating mechanical properties for the machine | *calcGenProg.m* |

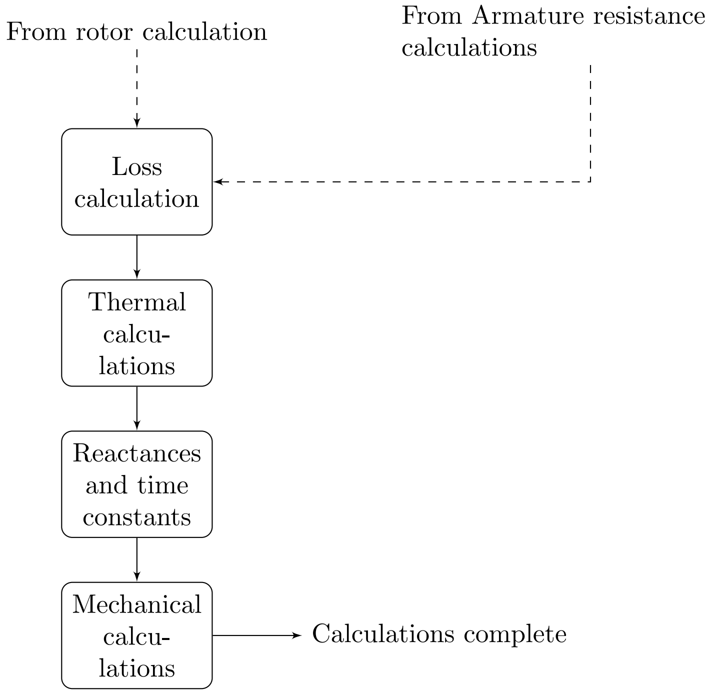


Figure 10: GenProg auxiliary calculations

## 5.2 Additional *GenProg* functionality

### 5.2.1 Default values

In addition to the restructure some functions where developed to add to the *GenProg* core functionality. The first being the *defaultVal.m* function which serves to gather the *default value* of parameters. It was discovered that some key *required parameters* was only labeled as such because the script required a non-zero value for them. Previously this was handled by simply asking the user to type the value in to the command window, but this would obviously be cumbersome in context with the GenProg *App*, and GUI. By introducing a set of default values for parameters the list of required parameters was reduced from 24 down to 12. This change is only visible for the *app* section of this project as any interaction with the Excel worksheet still uses the old combination of 24 required parameters and 42 optional. See table 2 for list of the current default parameter and their value.

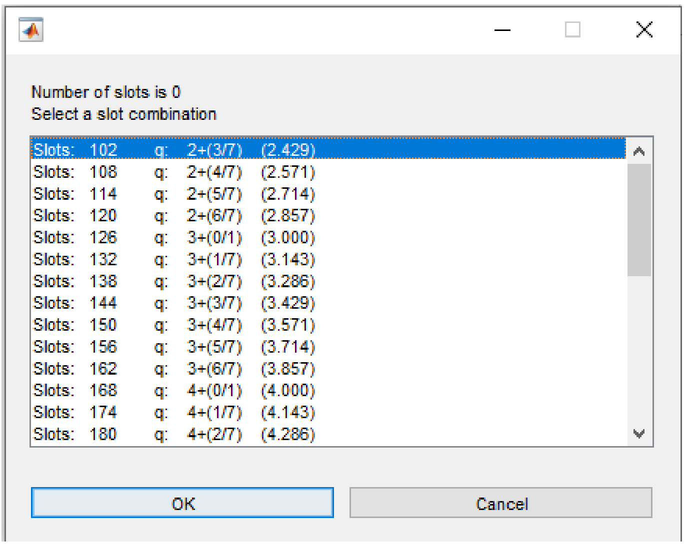
### 5.2.2 Choosing the slot number

Previously *number of slots in stator* was an optional parameter where if set to 0 (not defined by user) the script would generate a table of possible combinations, and write this to a separate Excel worksheet. The script then waited for the user to check this table, and type the desired number of slots into the command window. This was obviously not practical with regards to the *app* functionality of GenProg. The source code for generating the possible slot alternatives was kept mostly intact, and only formatted to remove duplicates, and arrange them ascending. The function *calcSlots.m* returns a *slot table* with $n$ rows containing the number of slots, and $q$ as a fraction and $q$ as a decimal number.

A second function is invoked to take the resultant *slot table* and displays a *dialog box* for the user to select the desired number of slot from a list. See figure 11 for an arbitrary example where the user has not defined the number of slots in the machine.

Table 2: List of parameter and their corresponding default value

| Parameter | variable name | Default value |
|---|---|---|
| turns per coil | tnr | 1 |
| parallel circuits | pnr | 1 |
| height of a single armature strand | hcus | 0.0018 [m] |
| height of slot wedge | hspk | 0.006 [m] |
| height middle strip divider | hm | 0.007 [m] |
| heigth glide strip and spring | hgls | 0.002 [m] |
| roebbel sepperator | drs | 0.0005 [m] |
| strand insulation | dicu | 0.0001 [m] |
| height between slot wedge and air gap | hds | 0.001 [m] |
| number of parallel strands in armature | ndlp | 2 |
| current density in stator | Ss | 3 [A/mm] |
| current density rotor | Sfi | 3 [A/mm] |
| maximum flux density stator tooth | Btmx | 1.7 [T] |
| maximum flux density pole core | Bpmx | 1.6 [T] |
| maximum flux density yoke | Bymx | 1.3 [T] |
| stationary reactance | xd | 1.2 [pu] |
| transient reactance | xd' | 0.4 [pu] |
| sub-transient reactance | xd" | 0.15 [pu] |
| core section length | bcs | 0.04 [m] |
| cooling duct length | bv | 0.008 [m] |
| field voltage | Vf | 200 [V] |


Number of slots is 0
Select a slot combination

```
Slots:  102   q:  2+(3/7)  (2.429)
Slots:  108   q:  2+(4/7)  (2.571)
Slots:  114   q:  2+(5/7)  (2.714)
Slots:  120   q:  2+(6/7)  (2.857)
Slots:  126   q:  3+(0/1)  (3.000)
Slots:  132   q:  3+(1/7)  (3.143)
Slots:  138   q:  3+(2/7)  (3.286)
Slots:  144   q:  3+(3/7)  (3.429)
Slots:  150   q:  3+(4/7)  (3.571)
Slots:  156   q:  3+(5/7)  (3.714)
Slots:  162   q:  3+(6/7)  (3.857)
Slots:  168   q:  4+(0/1)  (4.000)
Slots:  174   q:  4+(1/7)  (4.143)
Slots:  180   q:  4+(2/7)  (4.286)
```

Figure 11: Example of the slot dialog box

## 5.3 Slot Calculation Rework

### 5.3.1 Background and problem

As the project moved on to improve the user experience it was realized from the generated cross sectional view that the GenProg script calculated slightly wrong slot dimensions with regards to the amount of copper, and available space for said copper when the winding is of roebel type. This was a know issue from the previous semester. Another glaring weakness is how the script handles input data in this section: If the user has defined the total slot height, it assumed the rest of the slot dimensions was also set by the user. Meaning: The user had to decide whether to define *all* the slot parameters, or *none of them.* If the user enters the slot dimensions there is no guarantee the calculated parameters are *correct* in that there is enough space for the desired amount of copper or number of armature strand and so on. In other words you run the risk of creating a *nonphysical* machine.

### 5.3.2 Solution requirements

Based on the aforementioned deficiencies it was decided to rework how GenProg calculates the slot dimensions. The requirements was as follows:

- Modify the functions for calculating the slot dimensions with a clear hierarchy of which parameter override which.
- Always return *valid* values for the slot.
- Inform the user if the function encounters a discrepancies with the input parameters and the calcualted parameters.

This was done for both the roebel-strand, and form coil winding types. $tnr = 1$ and $tnr \neq 1$ respectively. Although there are some slight differences between the two cases, the basic working principle can be seen in figure 12. The detailed description described in section 5.3.3 still apply to both cases.

### 5.3.3 Detailed description of the slot rework

The slot dimension calculations can be divided into two *branches.* The first being weather the user has defined the total slot height, and the second not defined the total slot height. See figure 12 for complete flow diagram.

**Total slot height defined by user** Before the function for slot dimensions is called, the script already has the total slot width from the initial slot calculations performed by *calcStator.m* function. Meaning if the user has also defined the total slot height the area available for the copper is *finite.* First step is calculating the available width for the copper strands. If the *number of parallel stands* combined with the *width of one strand* (both parameters are user definable) exceeds the available width, the *number of parallel strands* is discarded, and the *width of one strand* gets precedence.

The *height of one strand* is considered a *constant*, meaning it either has a default value, or user defined value. This simplifies the process of finding *number of vertical strands*, as it is finite as well. If the user defined number vertical of strands exceed the *maximum* number of vertical strands available in the slot layer, the user defined value is discarded in favour of the maximum calculated value.

**Total slot height NOT defined by user** The *number of parallel strands* and *width of one strand* procedure is identical to the other branch, but the *required copper area* is calculated beforehand as a function of the current in one winding, and the *target current density* parameter. Since the slot width already has been defined, the only variable is vertical height, and *number of vertical strands* is found by dividing *required copper area* by *number of parallel strands* and *width of one strand.*

When the number of parallel and vertical strands are calculated, the remaining miscellaneous calculations and parameters can be calculated without difficulty. The main difference between the function for $TNR = 1$ and $TNR \neq 1$ is that for the case for $TNR \neq 1$ the calculations is firstly done per turn, and then added up to make up the entire coil. This case also omits the top and bottom most strand which is a necessity for $TNR = 1$.

It should be noted that if the user has not defined either *TNR*, *number of vertical strands* and *number of parallel strands* (ndlp), the script will automatically choose $ndlp = 2$ and $TNR = 1$ as shown in table 2. In other words the parent function for GenProg will always choose the *roebel winding* for the armature without user intervention. It possible to call the script for $TNR \neq 1$ with *number of turns per coil* equal to 1. All the user has to do is define *number of parallel strands* $\neq 2$. There is no option to manually choose winding layout. This was done deliberately in order to ensure the script always returned valid parameters, and remove any possibility for the user to choose *incorrectly.*

## 5.4 Rotor calculation rework

### 5.4.1 The problem

The problem with the rotor calculations only appear when the user has NOT defined both the *pole core height* and *total field winding height.* This initiates a segment of the code that *iterates* for the final pole core and field winding dimensions. This can be visualized in figure 9 by the while-loop decision. In reality there are two nested while-loops, one for the pole core height, and one for target current density. The problem is the stop-criterion for both loops is the *delta-value* from the previous, and current iteration given as a ratio. When this ratio approaches zero the calculations are considered complete. (both loops stop when the ratio is below 5%). On paper this sounds reasonable, but the problem arises when either the initial guess of the pole core (which is performed in *Pole dimension* in figure 9) is too far off from the *correct* value, or when any of the intermittent calculations, i.e *magnetic calculations*, *Leakage inductance calculations*, or *necessary magnetization* (in figure 9) returns an invalid parameter. Either while loop can rapidly converge to a negative value, or infinity. Both of which will exit the while-loop under the presumption that the values are satisfactory. This bug was recognized early in the projects cycle, but was mistakenly assumed to be a bug arisen from the *restructure* of GenProg. And not with GenProg itself as was the case. This segment of the source code also had a problem with the parameters not converging at all, and seemed to oscillate. This was at the time solved by implementing a *iteration limit* described in section **??** to stop the while loop when the iterations ran amok.

Much effort was put into trying to resolve the issue, but a combination of the limited time frame, and the seemingly random behaviour made it clear that it would be easier, and faster, to create a *fundamentally new* solution. Rather than fixing the bug in the old source code. It also became apparent that there was not enough time for either of the alternatives, so the proposed solution is only *described* here, and only serves as a suggestion.

### 5.4.2 Proposed solution

The biggest problem with the current rotor calculation is the potential for *invalid* parameters. In order to ensure a valid set of rotor calculations, the problem should be approached with a finite lower limit, and gradually increased until the parameters reach an appropriate stop criteria, or the *upper limit* is reached. This approach stands in contrast with the current system of an initial

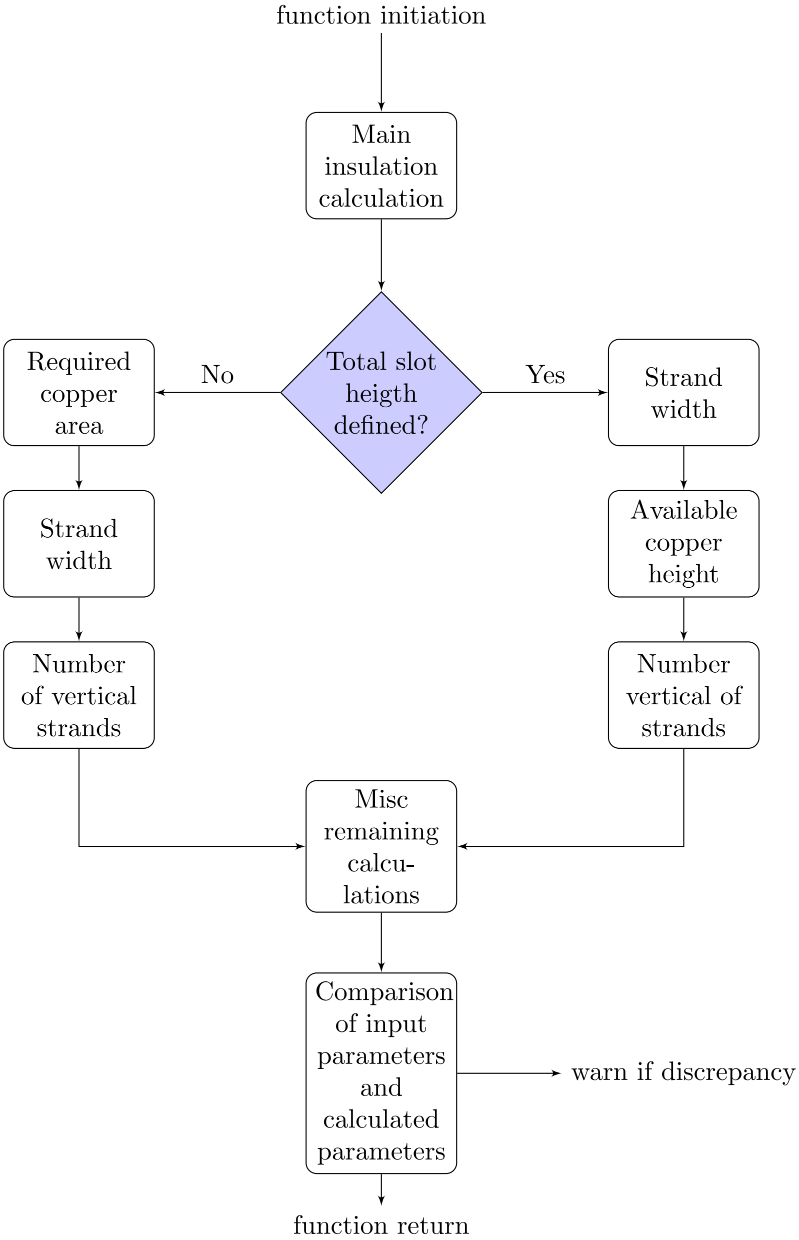


Figure 12: Flowchart for the calculating slot dimensions. Improved variant

*guess*, and then hope the calculations converge on correct values.

Since the stator is already calculated at this stage of GenProg, and clearly defined, the upper and lower limit can be found geometrically. For example: ”how big can each individual pole be before there are no more space?” or: ”how many field winding can you stack on top of each other before they collide with the ones from the next pole over” and so on. It should be noted that in the old (and new) version of GenProg the parameter *target pole flux density*, *Bpmx* is unused throughout the source code.

After the upper and lower limit is found, these can be used directly, or the function can iterate to find a more optimal solution based on one or multiple stop criteria. During the iteration process the parameter(s) that is to be changed, should do so *linearly.* I.e with a fixed step-length. This would help remedy some of the more nonlinear characteristics of the magnetic calculations that can cause the calculations to *oscillate.* GenProg consists of mostly *simple* calculations that require an almost negligible amount of computing power and is well suited for *cyclic iteration.* A proprietary benchmark showed that the entire GenProg script could run 100 000 times in 8 seconds on an average desktop computer. The performance gained by using a more sophisticated method does not warrant the potential for erroneous result.

The solution can be taken a step further by incorporating the stator calculations as well. If the *stop criteria* is reached *before* the upper limit is reached, the stator should be made with a smaller radius, or shorter height. Correspondingly if the *upper limit* is reached before the stop criteria, the stator should be made bigger. Either radially, or vertically. The stop criteria is yet to be determined, but some obvious suggestions include:

1. Current density field winding
2. Flux density in the pole core

The entire stator and rotor calculations can be described by the flow diagram in figure 13. For a reasonably optimal solution, the stop criteria, and upper limit are *equal.*

**Workaround** As an interim solution the total field winding height, and pole core height are *required* by the user, and the visualization of the generator must be used to alter the design.

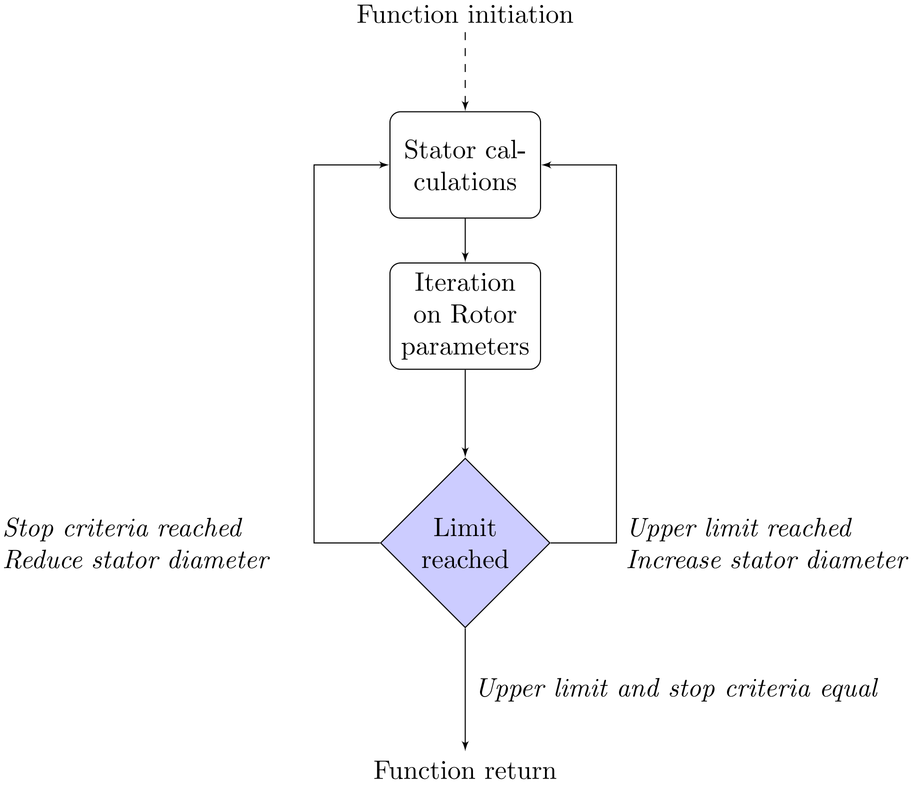


Figure 13: Simplified flow diagram for the suggested solution to the unreliable rotor calculation

# 6 Improved visualization of the calculated generator

## 6.1 Vectorizing the Cross section

### 6.1.1 Background

From the preceding semester project a *point matrix* image of the calculated generator was displayed for the user. However this *image* was more of an interim solution before a properly vectorized cross section image could be developed. The old system of a *point matrix* was easy to create, and gave a reasonable visualization of the calculated machine, but it lacked *fidelity* and *flexibility.* The image could only be square, meaning only 90 mechanical degrees could be rendered. This *coarse* diagram then needed a ridiculous resolution in order to be useful for fine adjustment of the design. This again required an unreasonable amount of RAM just to *store* the matrix. See section 4 for a condensed version of the preliminary work detailing the *old* cross-sectional system.

The *old* system utilized a set of of *objects* represented by boundary positions. Each object has at least 3 boundary positions, where each boundary position was represented by an x-position and a y-position. Each object could then be represented by a *x-vector* and a corresponding *y-vector.* Objects with the same dimensions (meaning same number of boundary conditions), could then be stacked on top of each other, where each row is a new *object.* See figure 14 for an arbitrary set of objects. This feature was created for storing for example all the armature strand objects, as they easily reach 10s of thousands for a complete machine, and pass them all to the plotter function as a single matrix.

Initially these x- and y-matrices was then fed to a *plotter* which *translated* the object to the correct position within the *frame.* It should be noted that the *point matrix* consisted of coordinates in *millimeters*, and the plotter translated these millimeter positions into the resolution of the frame in such a way that the *outer radius* of the stator reached 90% of the frame edge. See figure 7 for reference. This process was slow, and a cross-sectional image with a resolution of 10 000 by 10 000

$$
\begin{aligned}
object_1 &= [x_{11}, x_{12}, \cdots, x_{1n}] && (row\ 1)\\
&\vdots\\
object_m &= [x_{m1}, x_{m2}, \cdots, x_{mn}] && (row\ m)
\end{aligned}
$$

Figure 14: x-matrix for an arbitrary number $m$ object with $n$ boundary positions

could take up to 90 seconds to render, in addition to using large amounts of RAM. The lack of speed could be attributed to the *plotter* having to physically *fill in* the space the object occupied with a color vector for every pixel, and store the resultant matrix array. This method was from very early on understood to only be a temporary solution before a robust method of *vectorizing* the objects could be developed. The *objects* themselves being of sufficient fidelity with their Carteisian coordinates in millimeters and double float-point accuracy required no further alterations. See section 4 for a more in depth description of *how* the point matrices are generated.

Please note that the *old* plotter utilized the *inserShape* command function from the toolbox *Computer Vision Toolbox*, and this function required the *object* to be represented by one vector containing both the X and Y boundary coordinates. However the X and Y coordinates where individually generated and later combined to facilitate the use of the aforementioned command.

### 6.1.2 Requirements for the new plotter

The requirement for the new *plotter* function was as follows:

- Use the preceding method of x and y vectors for representing the complete object in order to keep the amount of new code to a minimum.
- Increase the *performance* meaning the completed image should be quick and responsive for the user to pan and zoom.
- Vastly increased fidelity over the old system.

In MATLAB this functionality already exists in the shape of the *plot* command. So this was used as a basis for the new plotter. Several *iterations* of the plotter was continuously developed as new functionality was needed, but the *final* iteration can be seen in figure 3 in appendix A.

### 6.1.3 Final version of plotter

The final iteration of the plotter revolves around the use of *polyshape* command. Polyshape creates a logical polygon object from the aforementioned x and y matrices. This *object class* is natively supported by the plot command.

The function iterates through each row in the matrices. The plotter assumes the x and y matrices are of equal size. Each row is handled separately and plotted row-by-row. An interesting feature was discovered when some elements in a position vector was a NaN, or *Not a Number.* Technically numeric, but neither real or complex. When the polyshape function encounters such a numeric value it indicates a new polygon. If all the *remaining* positions are *Not a Number* no new polygon is ever initiated. This feature was exploited for added functionality described in chapter 6.3.2.

When the polyshape function encounters two individual positions that are equal the function returns a warning indicating that there are duplicate positions and the polygon has been altered. This is not of critical importance, but if the cross-sectional view source code returns objects with a lot of NaN's the command window can be cluttered with warnings. A simple system of detecting, and deleting the NaN's before the actual polyshape is created and plotted is present. Only NaN's are deleted as this is the only *duplicate* positions that should normally occur.

### 6.1.4 Added features, cross section

Previously the cross sectional view could be considered a canvas where an object or a segment of an object could be *painted over* as a way of removing them. For the vectorized code this is no longer possible, and therefore a robust method of deleting, subtracting, and merge the object became necessary. for this purpose three important functions was developed:

- *unionfnc.m* Function for merging objects. Takes one *original* object, and one (or more) object(s) to be merged to the original. Both inputs and outputs are in the familiar x-vector, y-vector format. Please note the initial name for this function was *rotorfnc.m* as it was first used to merge the rotor ring object with the pole core objects. It was later renamed as it could be used for other parts of the script, but any reference to *rotorfnc.m* in the source code refers to *unionfnc.m.*
- *subtractfnc.m* Function for subtracting multiple objects from one object. As *unionfnc.m* this function also uses the established x-vector, y-vector format. It should be noted that this function had a peculiar *boundary case* where somehow the subtract function returned a *invalid* polygon. Why this occured was never quantified, so a work-around solution was created. A simple if statement checks whether the returned polygon is valid (contains at least three *points*). If not returns three NaN's as positions. In essence a *null* polygon that will be ignored by the plotter.
- *deltefnc.m* The final *important function* was the *deletefnc.m* function. As the name implies, the function was developed together with the *blank matrix* shown in the flow diagram in figure 17. See chapter 6.3.2 for detailed description of the entire *blank system.* The function serves as a method of only rendering specific parts of the calculated machine.

## 6.2 Indexing the three different phases

### 6.2.1 Background and motivation

One of the projects future goals is to at some point be able to export the calculated generator to COMSOL in order to run a FEM analysis on the machine. Although this was not possible in the given time-frame, the work described in this report has been done to facilitate FEM implementation in the future. As a part of this on-going effort a robust system of laying out the armature winding was created. For the moment it is only utilized in the cross sectional view of the generator.

**The problem** The challenge with indexing the armature winding according to their phase stems from the fact that the *number of slots per phase per pole* usually does NOT equal an integer. For *fractional slot winding* machines this number is always an integer + a fraction. For the rest of this section this is number is referred to as $q$, and can be easily calculated by equation 5:

$$q = \frac{Q_s}{m\, N_p} \tag{5}$$

where $Q_s$ is total number of slots, $m$ is number of phases and $N_p$ is number of poles in the machine.

Obviously one slot layer can only contain the winding of one phase, so in order for the fraction to be complete the winding must be distributed unevenly. The *problem* can be summarized by saying there is a discrepancy between the mechanical degrees, and electrical degrees between the the rotor poles and stator slots respectively. A considerable amount of time was spent on trying to quantify the problem and find a solution. Numerous alternatives was tested with basis in the *fraction* part of q, as this was seen as the most *elegant* alternative, but none yielded sufficient result.

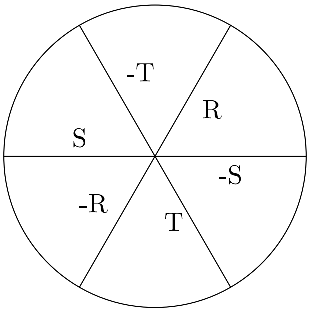


Figure 15: Visual representation of *revolver.m*

**The solution** The solution ended up being more of a brute force approach. First the function determines the total number of sectors in the machine, meaning how many times the *pattern* repeats itself for the winding layout. This is equal to the *greatest common divisor* for number of slots, and number of pole *pairs.* One sector rotation is then assumed to be 360 *mechanical* degrees. The corresponding *electrical* degree can then be written out as a vector containing all the degrees in one sector. This electrical degree can then be fed into a second function named *revolver.m* which takes an electrical degree as an argument, and returns a column vector where each row represent each phase R, S, and T. The function can be explained by a simple analogy. Consider for a moment the cylinder of a revolver with 6 shots. Each shot represents either the positive or negative R, S, and T phase. Only one cartridge can line up with the firing mechanism at a time, meaning it does not matter how many times you spin the cylinder. You will always line up with one of the six shots contained within the cylinders 360 degrees.

**Example** The best way to illustrate this is with an example. Take an arbitrary machine with $q = 2 + \frac{1}{2} = 2.5$. If this machine is three phase, and has 20 poles it would give a total of 150 slots. The *greatest common denominator* is 10, so the machine has 10 sector with each sector having 15 slots, and 2 poles. This gives the following electrical degrees for each slot in one sector:

$$eldeg = [0\ 24\ 48\ 72\ 96\ 120\ 144\ 168\ 192\ 216\ 240\ 264\ 288\ 312\ 336]$$

This column vector is fed element by element through the revolver function, and the following matrix array is generated:

$$\begin{array}{r} Phase\ R = \\ Phase\ S = \\ Phase\ T = \end{array} \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

For the second layer in the slot, the winding layout is multiplied by -1 and circularly shifted by the coil pitch. In this example the second layer is shifted to the right by 5 indents. This then boils down to a 6 × $n$ matrix (where $n$ is $Q_s/sectors$) and each row represent either phase R, S, and T in two layers of the stator slots.

$$\begin{array}{r} Phase\ R = \\ Phase\ S = \\ Phase\ T = \\ Phase\ R = \\ Phase\ S = \\ Phase\ T = \end{array} \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 \\ -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{array}{l} \left.\begin{array}{l} \\ \\ \\ \end{array}\right\} First\ Layer \\ \left.\begin{array}{l} \\ \\ \\ \end{array}\right\} Second\ Layer \end{array}$$

**Implementation** At the moment this *winding layout matrix* is only used for the visual representation of the calculated generator, but it is hoped that simple the *1, -1, and 0* and each column representing one slot in the sector can be utilized when the script is expanded with FEM analysis in COMSOL. For the complete code in its entirety see function 1 and 2 in appendix B.

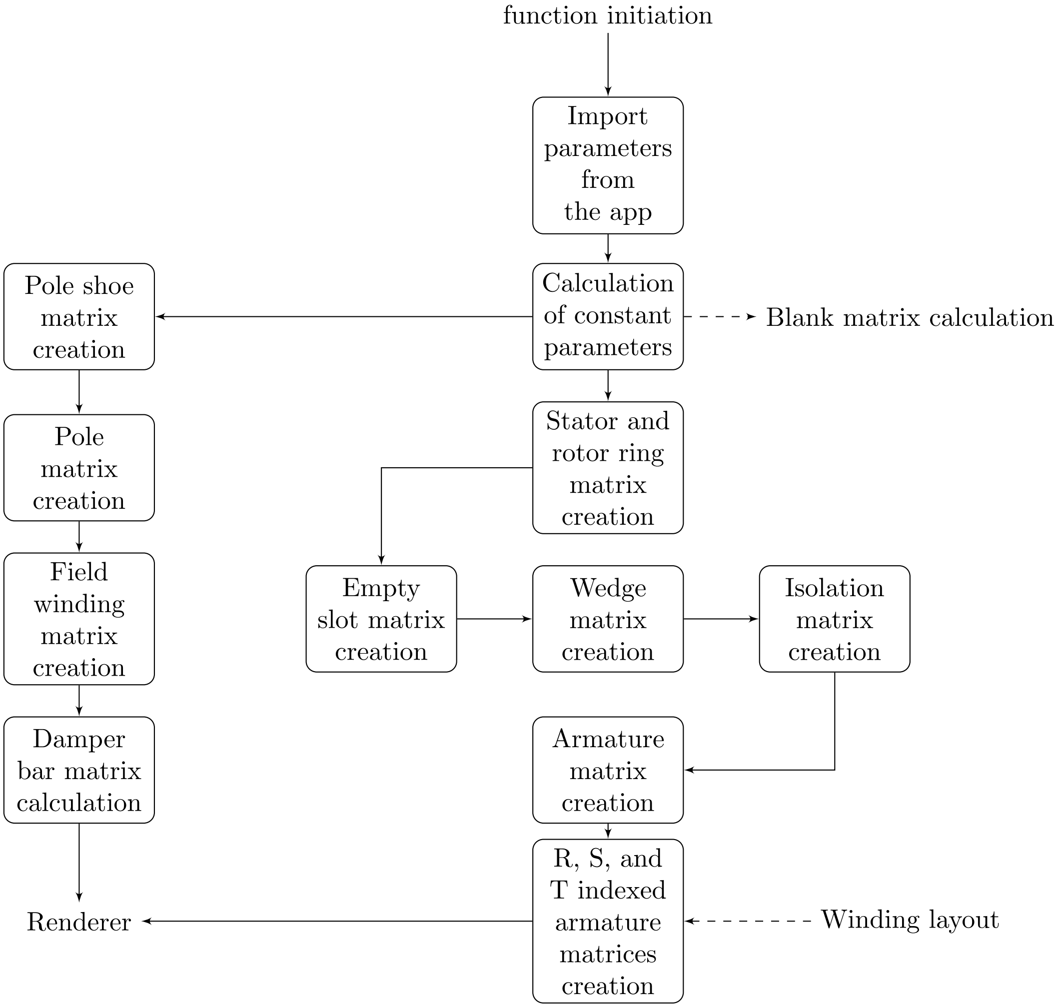


Figure 16: Flow diagram for the cross-sectional view matrix creation

## 6.3 General improvements to the cross section

### 6.3.1 Cross sectional script restructure

In much the same manner as described in chapter 5.1 the source code function for generating the cross-sectional view was restructured with legibility in mind. Chronologically this was done *before* the restructure of *GenProg*, but the process itself was much simpler. In part because the script was infinitely more familiar, and lacked GenProgs *interdependence*. The restructured working principle can be viewed in figure 16 and 17.

### 6.3.2 *deletefnc.m* and the blank object

From the very beginning it was the desire of the supervisor to be able to only display a particular segment of the generator. Previously the point matrix method did not provide such functionality as described in section 6.1.1. The image was *locked* to 90 degrees because it was not possible to *remove* excess machine objects, in addition to other limitations. Objects *outside* the frame could be ignored since they where not visible.

The method used was the *deletefnc.m* function combined with a *blank object*. In essence the function compares a object, and the *blank object*. Based on this comparison the function determines weather the input-object is inside, outside, or on the border of the blank object. Combined with a *render array* which determines which parts of the generator to be rendered.

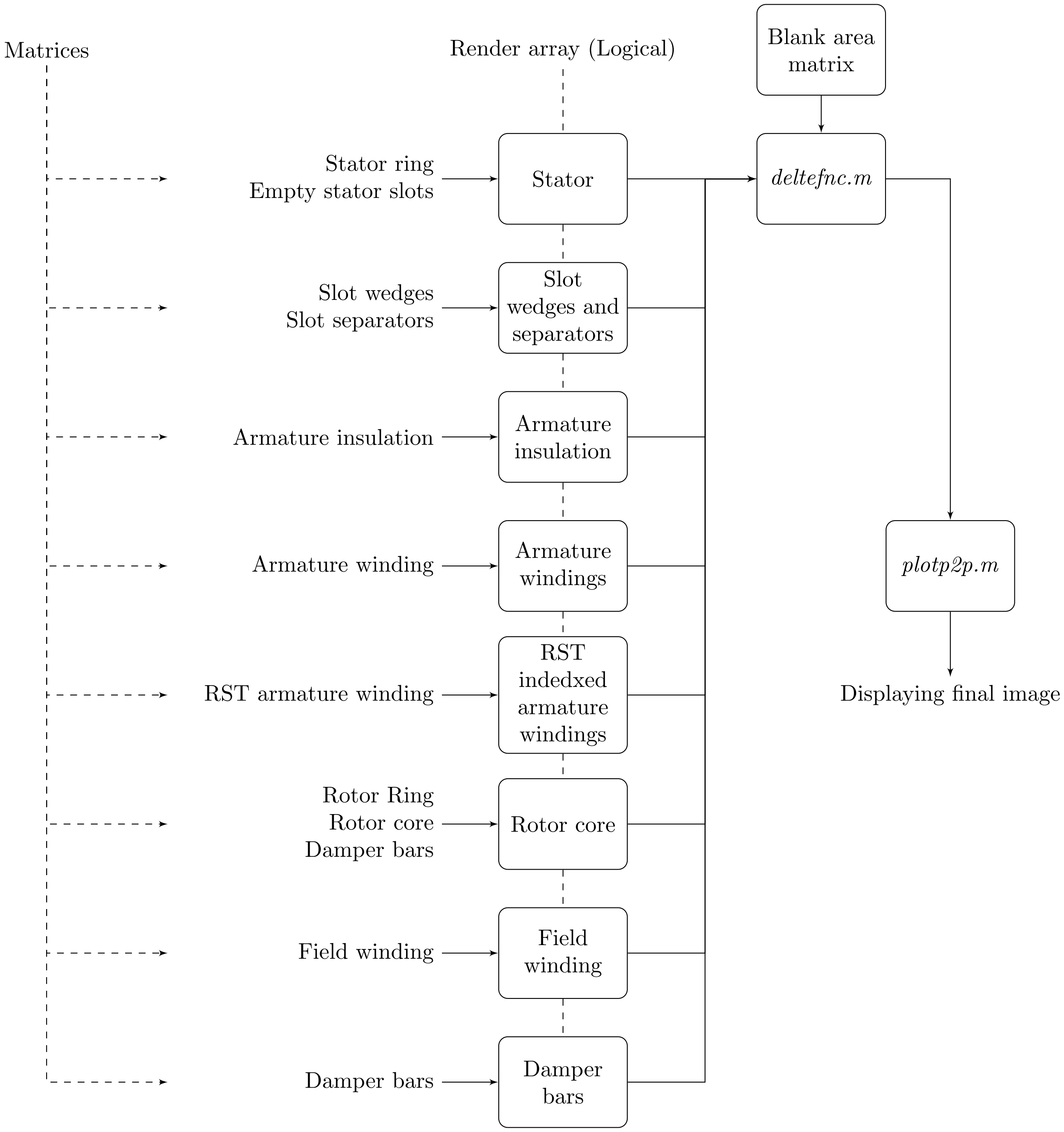
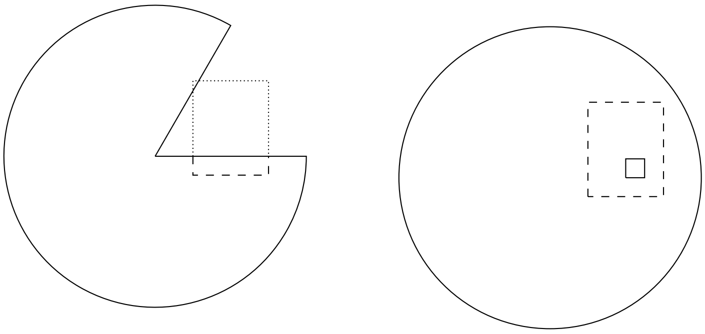


Figure 17: Flow diagram for *renderer* part of *CSGenVec360fnc.m*



(a) Inverse Pizza slice (b) Key-hole variant

Figure 18: The two different *blank objects* present in the SC function.

**Blank object** Before *deletefnc.m* is described in detail it is useful to understand the blank object first. At the moment only the *inverse pizza slice* and *keyhole* is present, but the possibilities are limitless. See figure 18a 18b for visual representation. The *inverse Pizza slice* variant has an outer radius slightly larger than the outer radius of the calculated machine. The empty slice is equal to the *degrees-to-render* parameter from the app. If this is set to 0 (default) the angle to render is equal to the angle between two poles.

The Key-hole variant is a complete circle with the same radius as the semi-circle found in the *inverse Pizza slice*, and a *Key-hole* that encompass one slot, and the immediate surrounding stator iron.

**deletefnc.m** The function compares the blank object with one or multiple sets of objects. Based on this comparison one of three things can occur:

1. The object is completely *outside* the blank object.
2. The object is completely *inside* the blank object.
3. The object is on the *border* of the blank object.

If the object that is to be compared consists of several rows (i.e more than one object) each row is compared separately. The function utilizes the *inpolygon* command, witch takes the x and y vectors for both the blank object and the object to witch it is to be compared, and returns a *logical array* with the same dimensions as the compared object. If none of elements are *true*, the object is completely outside the blank area, and the object can be retained as-is. If all the elements are *true* the object is completely inside the blank area, and can be omitted entirely. If however, *some* of the elements are true, but not *all* of them, the object is in a *border condition*, where some of it is inside and the rest is outside.

This poses a problem. The object can simply be subtracted using the *subtractfnc.m* function previously explained in section 6.1.4, but if the resultant object has different *dimensions* (meaning different number of positions) it cannot be put back into the same matrix array as the rest of the objects that perhaps are not in a border condition. This is best explained using an example.

Take the damper bars for an arbitrary machine. The damper bars consist of a circle along the leading edge of the pole shoe. Each circle consists of 24 positions. A machine has 7 damper bars per pole, and since the user has not defined the number of degrees to render, it will use the default value. From this calculation all the damper bars for the two poles in the image will be generated. For this example this will result in a 14 by 24 x and y-matrix. This poses a problem because two of these damper bars will be in a *border condition*, and after being subtracted with the blank object they do not share the same dimensions as with the rest of the damper bars.

This was solved by exploiting a feature that was discovered when developing the *plotp2p.m* function. The plotter interpreted a *NaN* as a divider between two objects, but if multiple where chained together no new object is ever initiated. Therefore the *deltefnc.m* function simply compares the the input *vector* with the original, and if not equal simply fills in the remaining positions with NaN's.

This proved to be simple and robust method of removing excess machine objects. without altering the the previously created object matrices.

### 6.3.3 Fixes

Although the actual amount of *minor fixes* was not documented, and usually not severe enough to require a considerable amount of time, a few are worth mentioning here.

**Linspace** Previously all the vector arrays was created with the following syntax:

```
vector = start : steplength : stop
```

For vectors with simple start, stop, and increments where all variables are neatly partiable with each other, the start *and* stop number is guaranteed to be included. This is not always the case, and a problem could be encountered where the stop variable was not included as the last element in the vector. Although rare, it was first encountered for a special machine when playing around with the *number of damper bars parameter*. From the previous semester report the *degree per damper bar* vector was generated with the following code:

```
radBarvec = NDs/2* degPerBar - degPerBar/2 : -degPerBar :...
-NDs/2 * degPerBar + degPerBar/2;
```

But for some combinations of *number of damperbars*, NDs, the last element was not included in the vector. This caused the function to crash because of a discrepancy between *number of damperbars* and number of elements in *radBarVec*. The solution was the *linspace* command which ensures the start and stop value is included in the vector. The syntax for the command is:

```
vector = linspace(start, stop, steps);
```

And for the damperbar scenario:

```
radBarvec = linspace(NDs/2 * degPerBar - degPerBar/2 ,...
-NDs/2 * degPerBar + degPerBar/2 , NDs)
```

The *linspace* command mostly replaced the column operator in the improved cross sectional view source code, as it was a much more robust way of creating vectors. It should be noted that the cross sectional view script frequently utilizes for-loops where the *number of something* is used as a parameters rather than the length of the vector used within it. This programming discipline makes the loops easier to read, but the matrices actually manipulated within need to be of correct size.

The exact cause of the phenomenon is not known, but is likely due to float point inaccuracies.

**Rounding** From the previous semester the rounding of certain elements was done by means of a linear x-vector, and the corresponding y-vector was calculated by means of trigonometry. From figure 4 we can see that a large amount x-positions was required to get the desired fidelity when the angle approach zero. With the old *point matrix* image, this was not recognized at the time, as the images fidelity was not good enough to observe the issue . In addition the *default value* for the *rounding resolution* was 100. With the new vectorized cross sectional view it was discovered that the time spent *rendering* the frame was roughly proportional to the *number of positions* within the frame rather than the *size* of the frame as was the case previously. As part of optimizing the new vectorized image, the resolution for all the objects with rounded corners was reduced. At this stage it became obvious that the method should be altered to use a *linearly spaced angle vector* and then calculate the x- and y-positions element by element. This gave a rounded object with a completely homogeneous appearance. In contrast to what i described in section 4.

**Default values** As mentioned in section 7.1 and 6.3.3 the default values for *ring resolution*, *shoe resolution*, and *winding resolution* was removed as an optional parameter from within the app. By means of trial and error a *resolution*, here meaning the number of positions on the rounded object, of 100 gave adequate visual fidelity for the stator ring, rotor ring, and the pole shoe. A resolution of more than a 100 yielded little in regards to visual fidelity, at an increasing cost of render-time. For the damper bars a resolution of only 24 gave *good enough* fidelity as the objects where usually *small* compared to other objects in the frame. The armature and field windings initially had a resolution of 10 per corner, but this could be reduced to as low as 2 or even 1 without much lost visual fidelity. The armature windings have by far the greatest impact on performance because of the sheer amount of armature strands in a frame, rather than the resolution of the rounding itself. A variant where only a simple solid rectangle could be rendered as a simplified imitation of a geometrically advanced armature bar. However this was only theorized and not implemented due to result achieved from just experimenting with the rounding resolution.

# 7 App improvements

## 7.1 Front-end improvements

The basic Layout of the app is mostly unchanged from the previous semester project. The most obvious change is the removal of the Cross sectional view from the centre of the app window. This was done as a result of the cross-sectional view improvements. The responsiveness of the cross-section windows was very poor in the app windows. Almost to the point of being unusable, and it was decided to move the cross-sectional view to a separate window. This freed up valuable screen real estate, and allowed the user to resize the cross sectional view windows separately. Several of the edit-fields was renamed, and / or moved to a more appropriate tab. Some honorable mentions include:

- *Specific ratio* was changed to: *Slot / tooth ratio*
- *Current density* input for rotor and stator was changed to *Target current density*
- *Number of strands in a bar*, *Number of strands per turn*, and *number of strands on top of eachother per turn* was changed to *Number of strands per bar*, *Number of horizontal strands per turn*, and *Number of vertical strands per turn* respectively. This was done to reflect the slot dimensions rework described in section 5.3.
- Cross section properties tab was refined by removing options for the rounding resolutions. This was done to reflect changes done within the cross sectional part of the script.
- The option to choose the winding layout was removed as this property is determined symbolically based on the calculated parameters.
- Removed several depricated edit-fields from the cross sectional view properties tab.

## 7.2 Back-end improvements

### 7.2.1 Parameter handling

Most of the resources allocated on improving the GenProg *App* was sunk into improving the back-end part of the app. One of the main issues with the GenProg app is its amount of *parameters.* Initially the parameters was put into numeric arrays, and then used as arguments for the different functions. This provided a simple and non-intrusive method of controlling the parameters passed on to the GenProg parent function, and the cross-section function, without significantly altering the source code. This was particularly important for *old* GenProg as it was suffering from a severe case of *Please don't touch it. No one knows how it works.* This boiled down to 2 *input vectors* and 10 output vectors. All of which needed to be read and written for several different app functionalities. Each element in each vector corresponds to a specific *edit field* in the app window.

A function was created for the following list of operations.

- Reading input parameters from an Excel worksheet and writing to the corresponding edit field in the app window
- Reading input parameters from the app window for use in in GenProg
- Save input parameters from the app to an existing, or new Excel worksheet
- Writing calculated parameters from GenProg to the app window edit fields
- Reading calculated parameters from the app window for use in the cross-sectional view function

This way of relating to the parameters poses a couple of problems. Very early on these functions was moved outside the *app designer* environment because it improved the readability of the *code view* for the GenProg app, but this meant that if the developer wanted to rename, or move parameters, the change had to be done in all 5 functions individually, in addition to where ever they where used. Obviously this was not good enough, and a better solution had to be developed.

**Solution** Before a more elegant solution can be devised an interim solution is created. App designer has functionality for a system of *global variable* that can be declared for use *inside* the app (private) or *globally* (public). This means that the 2 + 10 arrays that previously had to be passed back and forth between functions, could be stored only one place, and accessed without the need to pass it around as was the case previously. This made it vastly more practical to add or move parameters, and since this was handled within the app editor, renaming one edit field would also change the correct variable everywhere. In its current iteration its as simple as just renaming the edit field in the app designer, without the need to manually edit the edit-field variable in the underlying script.

### 7.2.2 Save functionality

A save functionality is of course a necessity with this type of software. Although a *save button* was present from last semester, it was a *dummy* component as the project ran out time before it could be properly implemented.

In order to save some time and complexity only the input parameters are saved. Either to a brand new Excel Worksheet, or overwrite an existing one (Manual backup of worksheets before overwriting is encouraged). The save function was made in such a way to copy the old format of 24 + 42 *required* and *optional* parameters respectively. Although this numbering is not technically the case anymore, this ensures backwards compatibility with existing documents.

In the future a better solution for storing and reading parameters should be developed. Excel worked well enough previously, but the platform should be phased out because problems when interacting with the Excel worksheet *and* Matlab simultaneously. Newer versions of Matlab even warns the user to not use *xlsread* as its *not recomended* starting with version Matlab R2019a due to performance and compatibility. Problems where encountered when using the the app on Macintosh machines, but could be worked around by converting from *.xls* format to the newer *.xlsx* format.

# 8 Type Check

## 8.1 System of detecting and displaying errors

### 8.1.1 Error code

Two different systems was developed and tried before a final version was decided upon. Initially the solution was envisioned with a kind of *rating system* where a warning or error was given a relative *score* indicating the severity or lack thereof. This system worked well, but it was deemed impossible to cover *all* the errors that *could* occur. Therefore efforts shifted towards making a robust and simple system for *adding* errors down the line instead. The problem with this initial system was that each error needed both an index location and a value, meaning the *error string* had to be a set length, in addition to having a value. This again made the addition of new errors tedious as it required changes in multiple locations in order to work, and more importantly if done incorrectly did not work at all.

The second iteration omitted the *index / value* system entirely in favor of a more *traditional* system of a simple *error code*. An error vector is initiated and when an error or warning is detected, its error code is simply appended. At the end of the GenProg function the error vector is used to generate a simple dialog box containing all the error and warning messages for the user. This greatly simplified the process of adding new errors as the developer only needs to add an error code, error or warning condition, and a corresponding error message. See section 8.4 for complete procedure for adding new errors.

The error vector can be retained for the entire GenProg function (or any other function), but in its current iteration only the input parameters and the complete calculated parameters are controlled. See table 4, 5 for complete list of implemented errors and warning.

The *type of error* can be categorized into three distinct categories.

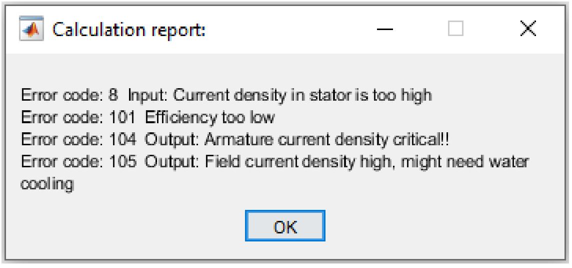
- Critical error - Only occurs during the initial stage and is intended to stop GenProg before any calculation is performed as it will cause a crash.
- Error - An error that is severe, but will not cause GenProg to crash, but cause it to calculate *invalid* parameters.
- Warning - Intended to give the user a warning or point attention to a particular parameter.

### 8.1.2 Error Message, and Calculation Report

One unique error code corresponds to a unique error message. The error codes are divided into brackets of 100 where each centuriate indicates the *type of error.* At the moment only three brackets exists with exception for one special case, the pole shoe height.

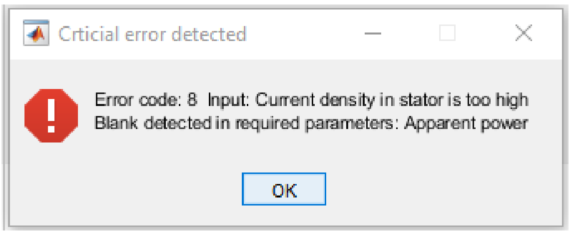
- 1-99 Input error or warning
- 100-199 Output error or warning
- 200-299 Output array validation

When the *error code vector* reaches the end, all the errors can be fed element by element to a *error message array* that compiles the complete error message array, and display it as a dialog box for the user. See figure 19 for an example of a calculation report.



Error code: 8 Input: Current density in stator is too high
Error code: 101 Efficiency too low
Error code: 104 Output: Armature current density critical!!
Error code: 105 Output: Field current density high, might need water cooling

Figure 19: Example of a typical calculation report



Error code: 8 Input: Current density in stator is too high
Blank detected in required parameters: Apparent power

Figure 20: Example of a critical error, and failed blank-check

## 8.2 Input Sanitation

### 8.2.1 Background

The original GenProg script lacked ANY input sanitation. As stated in Lundseng and Vikans master report from 2010 it is the users responsibility to ensure the input-parameters are correct [3]. It was believed that a comprehensive input-sanitation of the required and optional parameters would ensure a more stable GenProg, and help the user understand why the calculation returned what it did. The input sanitation is incorporated into the new *calcGenProg.m* parent function and is performed before any calculation is attempted. The logical function can be viewed at the top in figure 8. If any *Critical* is detected, the script stops, and returns empty *output vectors*.

**Blank check** As mentioned in section 5.1 the required parameters (reduced from 24 down to 12 parameters) is *required* to have a value. Besides the regular type check of the input parameters, the function *typeCheckBlank.m* performs a blank check of the *required parameters vector*, and returns a logical array with a *true* if the script detected one or more zero-element in *required parameters*. The first element denotes whether a blank has been detected or not, and is considered a *critical error*.

Please note that both *moment of inertia* and *skewing* can be zero, and the blank check ignores these parameters if a blank is detected.

## 8.3 Out Control

### 8.3.1 Basic parameter check

currently only a few parameters are checked directly after GenProg is completed. These are: *Efficiency* and *current density* in both the stator and rotor. See table 5 for complete list.

### 8.3.2 Ventilation

GenProg calculates a number of thermal parameters including *Cooling air flow* and *Maximum air speed*. This can be used to calculate the dynamic pressure using equation 3 and the hydraulic power using equation 4. A typical fan is considered to have an efficiency of 50% (including safety margins) so a rough estimate for the required cooling fan *power* can be expressed. The function *outVentCheck.m* compares this rough estimate of fan power with the generators rated power. A warning and error is generated at 1% and 20% respectively after discussions with supervisor and an HVAC engineer.

The motivation for this out control was because of the way GenProg calculates the thermal parameters. Unless specified by the user, the thermal calculations is performed with gradually increasing airflow until the thermals where within specifications. This meant that GenProg could calculate incredibly unrealistic cooling parameters, and a simple condition like mentioned above could give the user some indication to check their input parameters for excessive loading conditions.

*outVentCheck* also trigger an error when the Maximum cooling air speed exceeds 20 m/s. Excessive air speeds should be avoided as the associated pressure drop in cooling ducts requires building specifications that are outside specified standards. As a reference ISO 15138 which covers *Heating, ventilation and Air-conditioning in offshore and petroleum installations* recommends 10 m/s and a maximum of 15 m/s as anything over this *requires considerations for extra noise insulation and high losses* [2].

### 8.3.3 Pole shoe height

After discussions with supervisor regarding the cross sectional view, and after the restructure described in section 5.1, it became apparent that a more sophisticated method of calculating the pole shoe height should be implemented. Unless specified by the user, its default value is 50mm. This poses a problem as the pole shoe must be strong enough to contain the copper field winding during a run away situation in order to prevent a catastrophic failure.

Initially a calculation of the pole shoe height was supposed to be implemented as a part of *calcGenProg*, but this was not possible without a major rework of the entire rotor calculation. The pole shoe height is required *before* the field dimensions are calculated. As an interim solution the pole shoe height calculation is implemented as an output control *after* calcGenProg is completed.

**Implementation** In order to simplify the calculation, a *sheer break* is assumed to not be a factor. Meaning the shoe is assumed to break due to *tensile stress*, and not *sheer stress.* This simplifies the calculation because it allows us to set $\sigma_y = 0$ in equation 2, and only consider the horizontal stress ($\sigma_x$) and *biaxial* stress $\sigma_{xy}$. From figure 21 we can see the forces manifest themselves in the corner between the *pole shoe* and *pole core.* This is where a failure will first occur. The general plane stress must not exceed the *yield strength* of the material in order to avoid permanent deformation. In addition it is assumed the force $F$ is applied directly along the y-axis.

From equation 2 the expression for $\sigma_x$ and $\sigma_{xy}$ needs to be calculated.

The formula for $\sigma_x$ is as follows:

$$\sigma_x = \frac{M}{I} = \frac{\frac{1}{2}lF}{\frac{h_{ps}t^3}{12}} = \frac{6Fl}{h_{ps}t^3} \tag{6}$$

Where $t$ is the length of the *machine* (z-axis), M is *resulting torque at inner corner*, I is *Second moment of area.* The Force $F$ can be calculated by the centrifugal force of the field winding on the pole shoe:

$$F = m_{cuf}\ \omega_r^2 r$$

Please note that the *effective* pole shoe height is used. By subtracting the damper bar area and slot from $h_{ps}$. This is not strictly correct, but for the purpose of this project it is deemed *close enough*, and serves as a *worst case scenario* where a damper bar is located dirrectly underneath the joint between the pole core and pole shoe. Finding the actual *effective* pole shoe height proved too difficult.

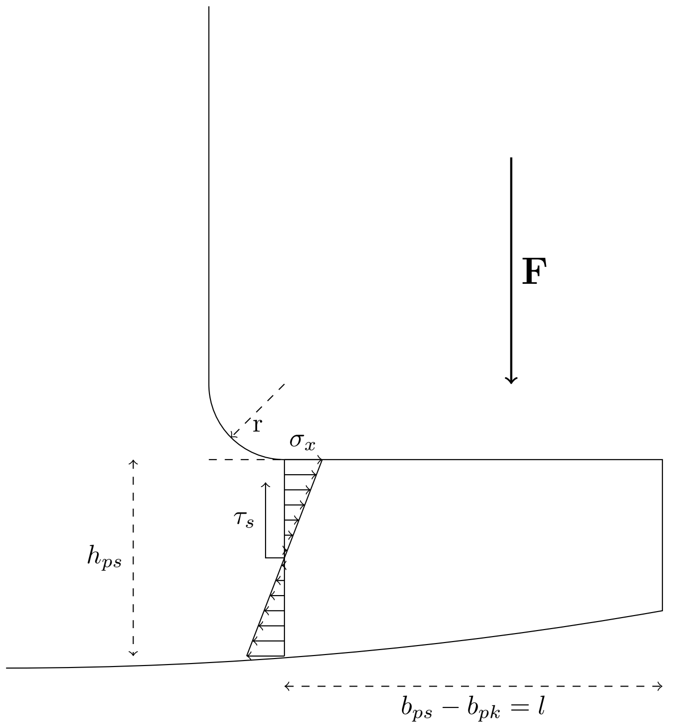


Figure 21: Diagram of the forces applied to the pole shoe of a salient pole generator

The formula for $\sigma_{xy}$ is:

$$\sigma_{xy} = \frac{F}{h_{ps} t} = \tau_s \tag{7}$$

Which is the *torque* exerted on the pole shoe.

The expression for $\sigma_x$ and $\sigma_{xy}$ can be put into equation 2 and gives equation 8 which can be calculated numerically:

$$\sigma_v = \sqrt{\left(\frac{6Fl}{h_{ps} t^2}\right)^2 + \left(\frac{F}{h_{ps} t}\right)^2} \tag{8}$$

**Steel** After discussion with supervisor it was decided the calculated *general plane stress*, $\sigma_v$, must not exceed 2/3 of the tensile yield strength of the material. The steel chosen was a type intended for use in automotive industry with a yield strength of around 590 $MPa$, or $N/mm^2$. It should be noted that this can easily be changed for another value at a later time, and purely serve as a *proof of concept*. In the future there should be an option for trying different types of material, and maybe even *observe* how this change impacts the design.

**Form Factor** The yield strength of the pole shoe is dependent on *form factor* $\alpha$. The form factor can be found in a number of ways, but a simple *stepped* approach was chosen based on figure 22. The form factor is chosen from the ratio of the radius $r$ and the effective pole shoe height. See table 3.

The *stress concentration factor* $\beta$ is mentioned in literature [1], but is not implemented in the calculations.

Then we can establish the final expression:

$$\sigma_{nom} = \alpha \ \sigma_v \tag{9}$$

Table 3: Function table for form factor $\alpha$

| $ratio = r/h_{eff}$ | $\alpha$ |
|---|---|
| $ratio > 0.1$ | 1.9 |
| $0.1 \geq ratio > 0.05$ | 2.2 |
| $ratio \geq 0.05$ | 3 |

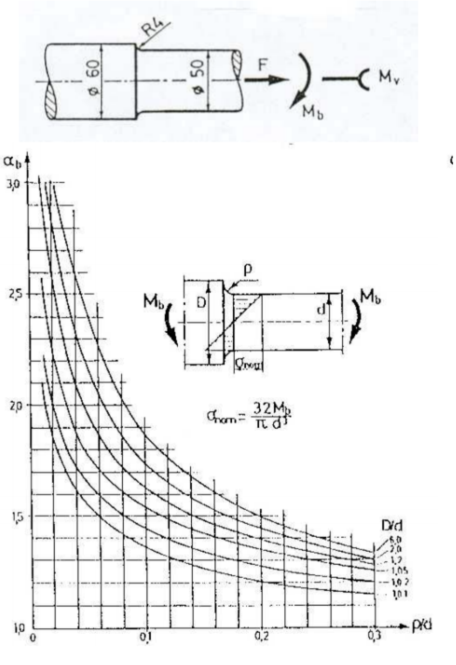


Figure 22: Form factor $\alpha$ for axle with rounded stepping exposed to bending stresses [1]

*OutPoleShoeCheck.m* only generates an error if the nominal stress $\sigma_{nom}$ exceeds the allowable stress for the material. I.e:

$$\sigma_{nom} > \frac{2}{3}590 \quad [MPa]$$

If the function detects an error it will *recursively* call itself with gradually increasing pole shoe height (1mm increments) until no error is generated. The *new* Pole shoe height is then displayed to the user as a part of the calculation report together with the necessary increase in height.

Note: This error is the only error where the user is given feedback on a *recommended* new value. This was done as a interim solution until it can be implemented as an integral part of the rotor calculations.

### 8.3.4 Array validation

Two functions was developed for checking weather the calculated parameters are valid. Each of the 10 output *vectors* are checked for non numeric and negative values. If any of the calculated parameters are not numeric or positive, it indicates that something has gone wrong in that specific section of calcGenProg, and should be investigated further by the user.

## 8.4 New error procedure

The procedure for adding new errors would be as follows:

1. Create an *error condition* where appropriate. This can be wherever the user whishes, but it is good practice to combine the error conditions in a tidy and orderly manner so it can easily be found.
2. *Append* the error vector either directly or in the parent function by using the following command:

   ```
   err(end+1) = errorcode
   ```

   Make sure the error code is unique.
3. Add an error message to *errList.m* at the correct indent and bracket. Example: error code = 225, the error message should be placed as the 25th element in the 201 - 300 bracket.

## 8.5 Result of the type checking

In total the entire *typecheck* function tree consists of 13 functions and child-functions and a total of 39 unique errors and warnings. A list of errors and warnings can be viewed in tables 4,5 and 6. A list of developed functions can be viewed in table 7.

Table 4: List of errors for input parameters.

| Code | Type | Comment |
|---|---|---|
| 1 | Critical Error | Length of *required parameters* is not equal to 12 |
| 2 | Critical Error | Any element of *required parameters* are not numeric |
| 3 | Error | Any element of *required parameters* are not positive |
| 4 | Criticial Error | Length of *optional parameters* is not equal to 54 |
| 5 | Critical Error | Any element of *optional parameters* is not numeric |
| 6 | Error | Any element of *optional parameters* is not positive |
| 7 | Critical Error | *Number of poles* is not an even number |
| 8 | Error | Target current density stator exceeds $3.5A/mm^2$ |
| 9 | Error | Target current density rotor exceeds $3.5A/mm^2$ |
| 10 | Critical Error | Nominal voltage and utilization factor is mutually exclusive |
| 11 | Critical Error | Gross iron length and Air gap flux density is mutally exclusive |

Table 5: List of output errors

| Code | Type | Comment |
|---|---|---|
| 101 | Warning | Efficiency lower than 95% |
| 102 | Error | Efficiency lower than 85% |
| 103 | Warning | Current density stator warning $3A/mm^2$ |
| 104 | Error | Current density stator Error $6A/mm^2$ |
| 105 | Warning | Current density rotor warning $3A/mm^2$ |
| 106 | Error | Current density rotor warning $6A/mm^2$ |
| 107 | Warning | Substantial hydraulic power required for cooling |
| 108 | Error | Required hydraulic power higher than 20% of generator rated power |
| 109 | Error | Cooling air speed too high |
| 110 | Error | Pole shoe height critical. Risk of failure during maximum rotational speed. |

Table 6: Output array validation

| Code | Type | Comment |
|---|---|---|
| 201 | Error | Stator parameters not positive |
| 202 | Error | Stator parameters not numeric |
| 203 | Error | Rotor parameters not positive |
| 204 | Error | Rotor parameters not numeric |
| 205 | Error | Magnetic parameters not positive |
| 206 | Error | Magnetic parameters not numeric |
| 207 | Error | Losses not positive |
| 208 | Error | Losses not numeric |
| 209 | Error | Reactances and time constants not positive |
| 210 | Error | Reactances and time constants not numeric |
| 211 | Error | Thermal parameters not positive |
| 212 | Error | Thermal parameters not numeric |
| 213 | Error | Mechanical parameters not positive |
| 214 | Error | Mechanical parameters not numeric |
| 215 | Error | Critical data not positive |
| 216 | Error | Critical data not numeric |
| 217 | Error | Extra parameters not positive |
| 218 | Error | Extra parameters not numeric |

Table 7: List of function relating to the type check and output controll for GenProg

| Function file name | Description | Parent function |
|---|---|---|
| *typeCheck.m* | Initial function for checking input parameters | *calcGenProg.m* |
| *parDecomp.m* | Function for unpacking the two input vectors, required and optional parameters | *typeCheck.m* |
| *typeCheckBlank.m* | Function for checking for missing required parameters | *typeCheck.m* |
| *returnErr.m* | Function for returning empty output matrices if a critical error is detected | *calcGenProg.m* |
| *parComp.m* | Function for compressing the calculated parameters into the 10 output matrices | *calcGenProg.m* |
| *outControll.m* | Initial function for output controll | *calcGenProg.m* |
| *outValid.m* | Function for validating the output matrices | *outControll.m* |
| *outVecValid.m* | Function for checking if any element in a matrix is either negative, or not numeric | *outValid.m* |
| *outVentCheck.m* | Function for generating errors related to ventilation | *outControll.m* |
| *outPoleShoeCheck.m* | Function for generating errors related to the pole shoe | *outControll.m* |
| *sprintErr.m* | Function for assembling and displaying errors and warning messages | *calcGenProg.m* |
| *errList.m* | Function for returning a single error string from a single error code | *sprintErr.m* |
| *parList.m* | Function for returning a string contiaing the name of a required parameter. Used in conjunction with the result of the blank chek. | *sprintErr.m* |

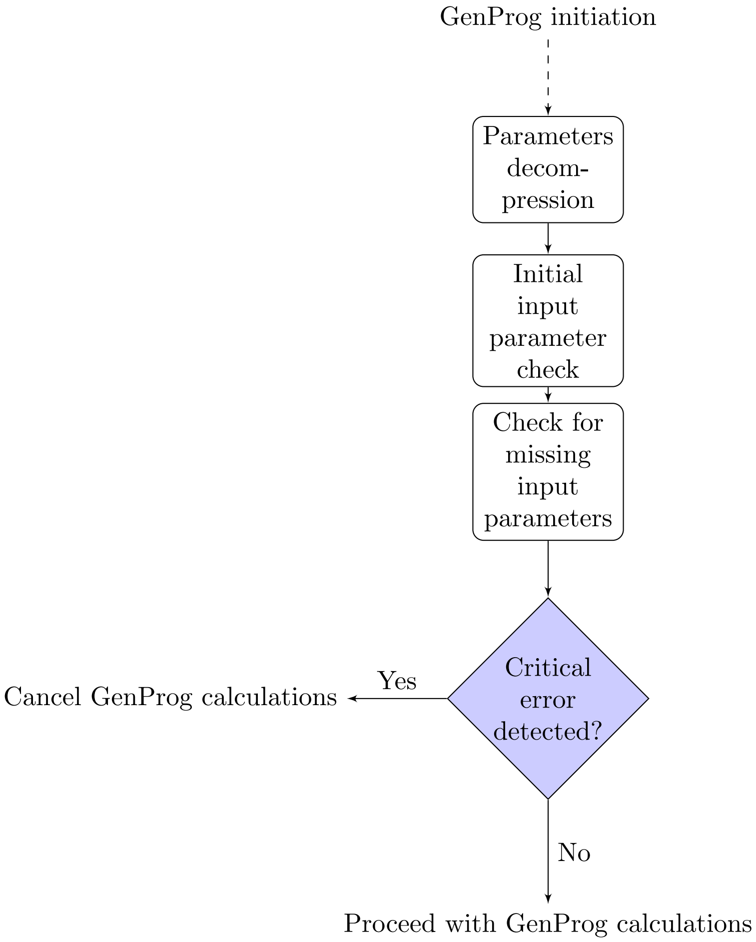


Figure 23: Flow diagram of *input sanitation* for calcGenProg

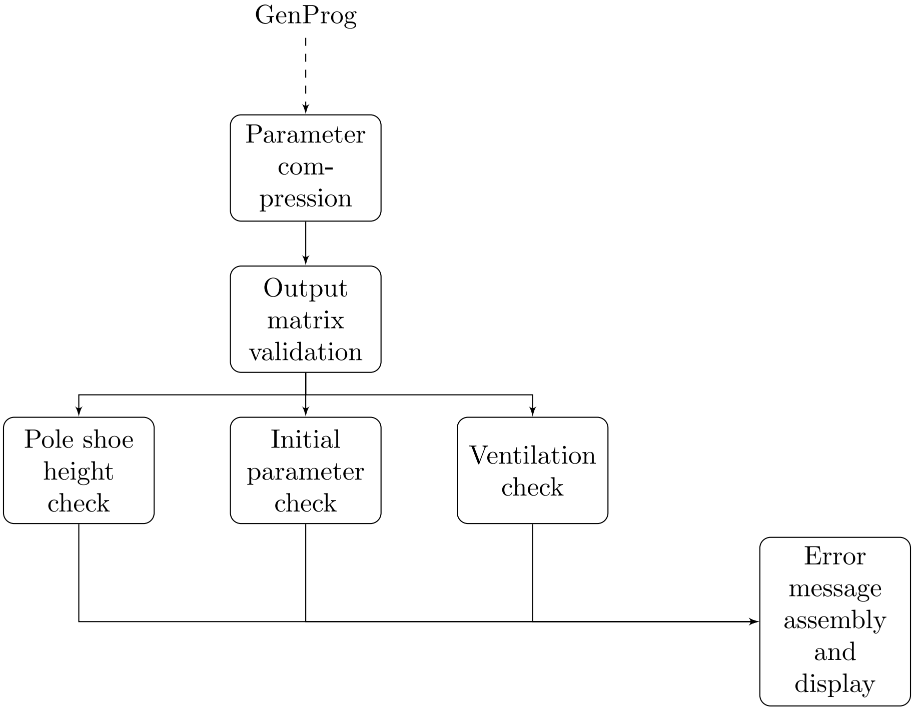


Figure 24: Flow diagram for output control, and final error message assembly and display

# 9 Result and discussion

## 9.1 Final result

### 9.1.1 Graphical user interface

This section includes result and discussion for both the app improvements and cross sectional view.

**App** The app was further developed from the previous semesters project with the following features and improvements:

- Reduced number of *required* parameters from 24 to 12.
- Corrected and resolved several linguistics and grammatical errors in *edit fields*.
- Restructured several parameter sentences to improve legibility.
- Added save functionality.
- Improvements to how the app reads and writes to the different *edit fields*.
- Reduce the amount of work needed to make changes to the graphical user interface.
- Moved the cross-sectional view *outside* the main app window.
- Developed a system of displaying a *calculation report* for the user.

**Cross sectional view** The following features and improvements was added to the cross-sectional view:

- Completely vectorized the cross-sectional view.

- Developed functionality to only render specific parts of the calculated generator, or omit them entirely.
- Method for indexing the three phases in the armature winding.
- Restructured the matrix creation into tidy and coherent individual functions and child functions.
- Separated the armature winding matrix creation into two different types: Form winding, and Roebel winding.
- Changed the method for determining the armature winding layout from a strictly user defined option to a symbolical variant.

### 9.1.2 calcGenProg

The following features and improvements was made to *GenProg*:

- Adapted the core *GenProg* script into separate unique functions and child-functions.
- Resolved several syntax flaws in if-statements.
- Improved how while-loops where used with guaranteed initiation, and return strategies.
- Completely rework the slot and armature winding calculations.
- Changed variable nomenclature.
- Adapted the existing function for choosing *number of stator slots* for use in a GUI environment.
- Added a system of assigning default values to parameters.
- Completely phased out Excel in *calcGenProg*.

### 9.1.3 Input sanitation and output control

The input sanitation and output control was created as a integral part of calcGenProg. It includes the following features:

- Developed a system for sanitizing the input parameters.
- Detect, and stop the calculations if a *critical error* where to occur.
- General validation for the 10 output vectors
- Miscellaneous parameter control for a limited set of specific calculated parameters.
- Estimating the structural integrity of the pole core using *von Mises yield* criterion.
- Validating the validity of ventilation parameters.

## 9.2 Design of an example generator

To illustrate the result a complete design example is presented in section 9.2. From installing the app, to viewing the result of the calculations.

### 9.2.1 Installation and startup

**Step 1** - Locate the installation file *GPapp.mlappinstall* in the source-code, and follow install instructions. This will add the *GPApp* to MATLABs *app tab.*

**Step 2** - To start the app simply launch it from the *app tab* within MATLAB. Make sure the *current folder* selected contains all the Excel files the user wishes to read from, and save to.

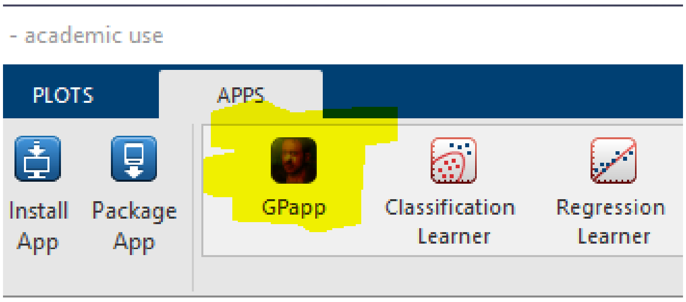
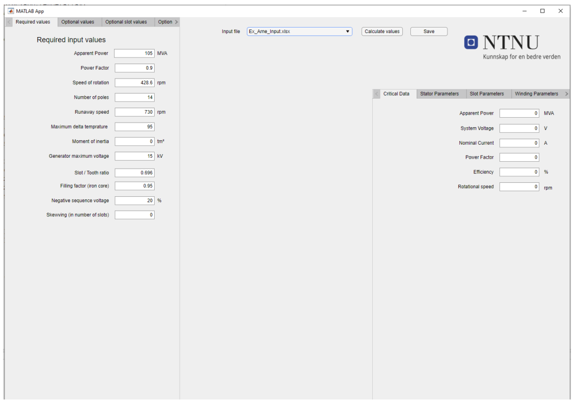


Figure 25: GPApp in MATLAB

**Step 3** - The App windows is presented to the user. From here the design process can begin.



Input file: Ex_Arne_Input.xlsx

Required input values

Apparent Power 105 MVA
Power Factor 0.9
Speed of rotation 428.6 rpm
Number of poles 14
Runaway speed 730 rpm
Maximum delta temprature 95
Moment of inertia 0 tm$^2$
Generator maximum voltage 15 kV
Slot / Tooth ratio 0.696
Filling factor (iron core) 0.95
Negative sequence voltage 20 %
Skewving (in number of slots) 0

Apparent Power 0 MVA
System Voltage 0 V
Nominal Current 0 A
Power Factor 0
Efficiency 0 %
Rotational speed 0 rpm

Figure 26: GPApp window

### 9.2.2 Design process

The user can either choose one of the available machine designs from the drop down menu, or design it completely from scratch. For the most *basic* machine only the *required parameters* tab needs be considered.

**Apparent power** An even value of 100MVA is chosen. It should be noted that the core functionality of *calc*GenProg is based on a compendium written by Westgaard [5], for machines ranging from 10-50MV.

**Power factor** NVE recommends a power factor of 0.86, and a minimum of 0.9. For our fictional generator a power factor of 0.80 is chosen.

**Speed of rotation and Number of Poles** Must be chosen together in order to achieve 50hz frequency. An arbitrary number of poles equal to 14 is chosen, which gives a speed of rotation equal to 428.6 rpm.

**Runaway speed** Not all that critical and twice the speed of rotation is a realistic baseline.

**Maximum delta temperature** A maximum of 95 degrees is industry standard, and corresponds to *insulation class B*.

**Moment of inertia** Can be omitted entirely, and set to 0 as the generator does not require *additional* rotating mass.

**Generator maximum voltage** Minimum of $\sqrt{2}$ higher than the desired nominal voltage, but a safety margin is recommended for adverse running conditions.

**Slot / tooth ratio** Can be played around with, but a good starting point is around 0.7.

**Filling factor** A filling factor of 0.95, or 95 % is *realistic* for the stator iron.

**Negative sequence voltage** A negative sequence voltage of 20% is required by NVE.

**Skewing** We are intending to create a *fractional slot machine* and therefore it is not necessary to include skewing. If $q$ is equal to an integer, skewing is recommended to reduce harmonic vibrations.

### 9.2.3 Running the calculations

When the required parameters have been filled in, the user is ready to run the calculations. Start by pressing *Calculate values*.

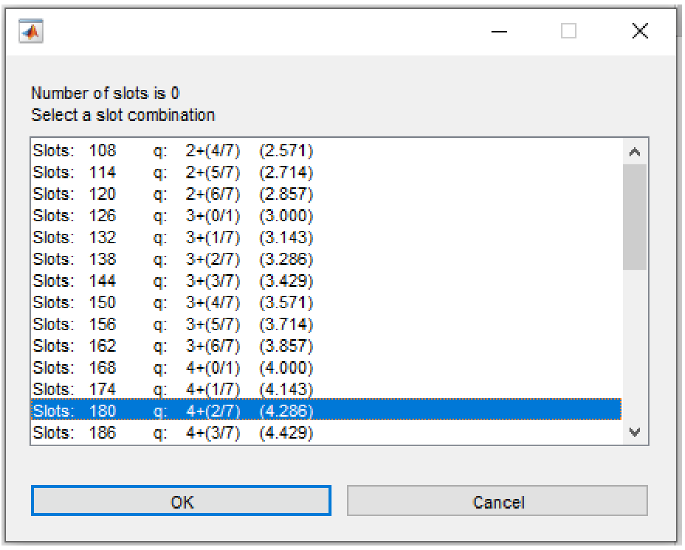

Input file Example_Generator_Input.xlsx

Figure 27: Calculate values button in GPApp

**Number of slots** During the calculation process the app will promt the user for *number of slots*. The user can choose from a list of appropriate values, and we choose 180 slots for our example machine. The user should make note of the number chosen, and fill it in under *optional parameters* when the calculations are completed. This is to prevent the app from prompting the user next time time the calculations are performed.

### 9.2.4 Interpret the result

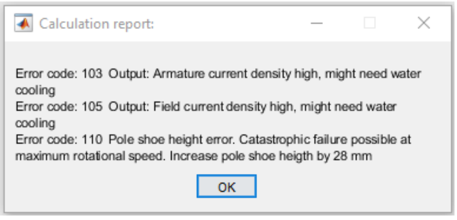
After *calc*GenProg is completed, all the 112 calculated parameters are displayed in their respective edit fields on the right hand side of the app window. From here the user can see all the describing parameters for the calculated machine. To help the user interpret the calculated parameters a *calculation report* is generated with a list of *error codes* and the accompanying error, or warning message. From the example we see that the following errors and warnings should be addressed:



```
Number of slots is 0
Select a slot combination
Slots:  108     q:  2+(4/7)   (2.571)
Slots:  114     q:  2+(5/7)   (2.714)
Slots:  120     q:  2+(6/7)   (2.857)
Slots:  126     q:  3+(0/1)   (3.000)
Slots:  132     q:  3+(1/7)   (3.143)
Slots:  138     q:  3+(2/7)   (3.286)
Slots:  144     q:  3+(3/7)   (3.429)
Slots:  150     q:  3+(4/7)   (3.571)
Slots:  156     q:  3+(5/7)   (3.714)
Slots:  162     q:  3+(6/7)   (3.857)
Slots:  168     q:  4+(0/1)   (4.000)
Slots:  174     q:  4+(1/7)   (4.143)
Slots:  180     q:  4+(2/7)   (4.286)
Slots:  186     q:  4+(3/7)   (4.429)
```

Figure 28: Slot number dialog box for example machine

- Pole shoe height should be increased by an additional 28 mm.
- Armature current density too high.
- Field current density too high.



```
Error code: 103  Output: Armature current density high, might need water
cooling
Error code: 105  Output: Field current density high, might need water
cooling
Error code: 110  Pole shoe height error. Catastrophic failure possible at
maximum rotational speed. Increase pole shoe heigth by 28 mm
```

Figure 29: Calculation report for example machine

Looking at the generated cross sectional visualization in figure 30 the some observations can be made:

- Rotor yoke collides with field winding.
- Visual inspection of the pole shoe height reveals the inadequate thickness to withstand the stresses exerted during a runaway situation.

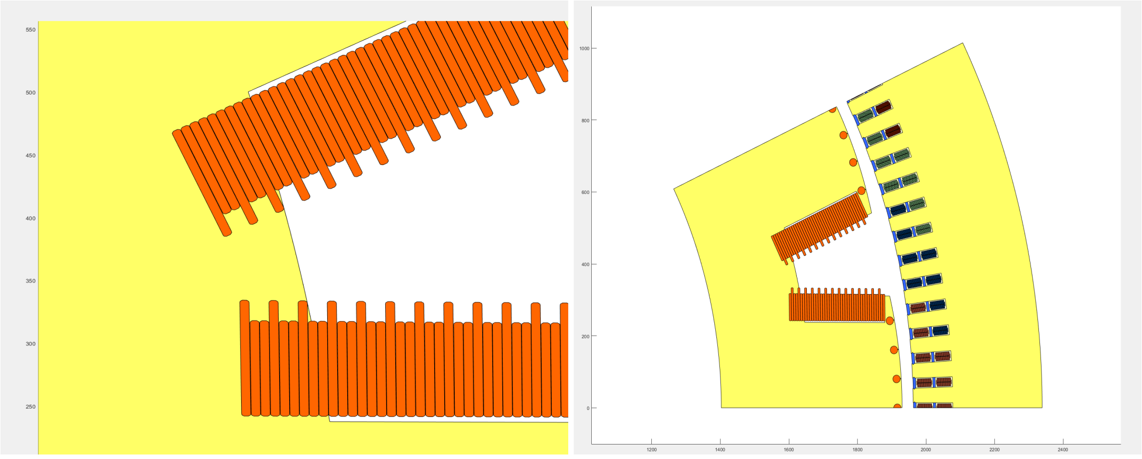


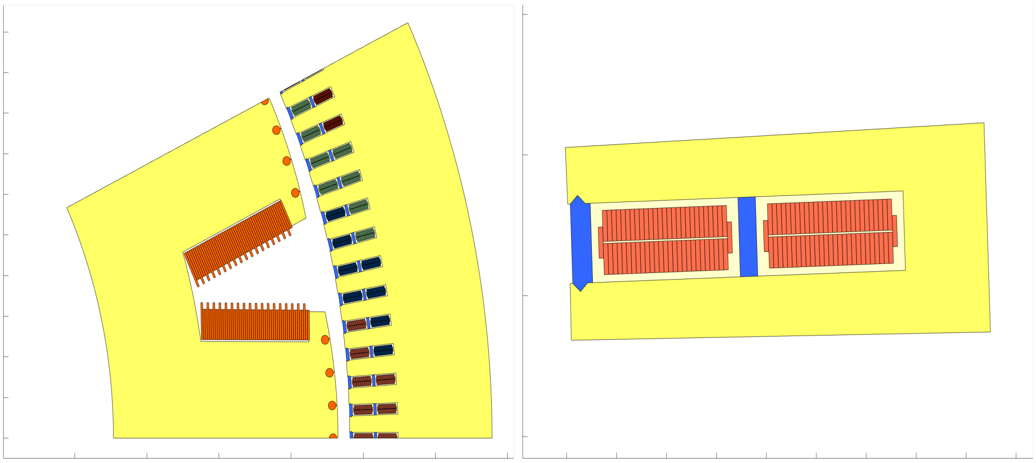
Figure 30: Initial cross-sectional view of the calculated generator

### 9.2.5 Finalizing the design

From the aforementioned lists the following adjustments are made:

- Reduce target current density of the armature winding, and field winding to 2.5 A/mm
- Manually set pole core height and total field winding height
- Increase pole shoe height from 50mm to 75mm.

**Finalizing the design** The parameters should be adjusted and calculations repeated until the app no longer generates any error codes, or the machine satisfies the users requirement.



(a) Final cross section

(b) Final cross section slot only

Figure 31: Final cross-sectional view of the calculated generator

Table 8 through 20 contains all the final input parameters and calculated parameters for the final machine. These parameters gives an accurate description of a salient pole generator which cross section can be viewed in figure 31. Note that the *Pole shoe height* parameter had to be further increased to 80 mm to accommodate the increased weight of the field winding, and that the ventilation airspeed exceeds the recommended maximum of 15 m/s. To further alter the design the possibility of reducing the surface loading should be considered in order to reduce the required cooling. Either by increasing the length, or radius of the machine.

Table 8: Example generator final set of required input parameters

| Parameter | Value | Variable name |
|---|---|---|
| Apparent power | 100 $MVA$ | *Sn* |
| Power factor | 0.8 | *Cosphi* |
| Speed of rotation | 428.6 $rpm$ | *ns* |
| Number of poles | 14 | *Np* |
| Runaway speed | 730 $rpm$ | *nr* |
| Maximum delta temprature | 95 | *dTmx* |
| Moment of inertia | 0 | *M* |
| Generator maximum voltage | 15 $kV$ | *Vmx* |
| Slot / tooth ratio | 0.7 | *budbd* |
| Iron filling factor | 0.95 | *kFe* |
| Negative sequence voltage | 20% | *Vnmx* |

Table 9: Example generator final set of optional parameters

| Parameter | Value | Variable name |
|---|---|---|
| Target Nominal Voltage | 10500 $V$ | *Un* |
| Number of slots | 180 | *Qs* |
| Target current density Stator | 2.5 $A/mm^2$ | *Ss* |
| Target current density Rotor | 2.5 $A/mm^2$ | *Sfi* |
| Pole core height | 300 $mm$ | *hpk* |
| Total field winding height | 300 $mm$ | *hf* |
| Pole shoe height | 80 $mm$ | *hps* |

Table 10: Example generator *Critical Data*

| Parameter | Value | Variable name |
|---|---|---|
| Apparent power | 100 $MVA$ | *Sn* |
| System voltage | 10500 $V$ | *Un* |
| Nominal current | 5499 $A$ | *In* |
| Power factor | 0.8 | *Cosphi* |
| Efficiency | 98.36 % | *Eff* |
| Rotational speed | 428.6 $rpm$ | *ns* |

Table 11: Example generator *Stator Parameters*

| Parameter | Value | Variable name |
|---|---|---|
| Utilization factor | 7.428 | *C* |
| Inner diameter | 3.924 $m$ | *Di* |
| Outer diameter | 4.714 $m$ | *Dy* |
| Gross iron length | 1.74 $m$ | *lb* |
| Net iron length | 1.52 $m$ | *ln* |
| Number of cooling ducts | 37 | *nv* |
| Number of turns per phase | 30 | *Ns* |
| Relative polepitch | 0.8556 | *y* |
| Coil span | 11 | *Ww* |
| Skewing | 0 | *s* |

Table 12: Example generator *Slot Parameters*

| Parameter | Value | Variable name |
|---|---|---|
| Number of slots | 180 | *Qs* |
| Slots per pole and phase | 4.286 | *q* |
| Winding factor | 0.931 | *kw* |
| Slot height | 134.4 $mm$ | *hs* |
| Slot width | 28.2 $mm$ | *bu* |
| Tooth width | 40.29 $mm$ | *bd* |
| Slot pitch | 68.49 $mm$ | *tauu* |
| Wedge height | 6 $mm$ | *hspk* |
| Bar separator height | 7 $mm$ | *hm* |
| Slot wedge spacer height | 2 $mm$ | *hgls* |
| Distance wedge and air gap | 1 $mm$ | *hds* |
| Roebel separator | 0.5 $mm$ | *drs* |

Table 13: Example generator *Winding Parameters*

| Parameter | Value | Variable name |
|---|---|---|
| Number of turns per coil | 1 | *tnr* |
| Number of parallel circuits | 2 | *pnr* |
| Armature loading | 802.8 $A/cm$ | $As$ |
| Stator current density | 2.5 $A/mm^2$ | *Ss* |
| Number of strands per bar | 56 | *ndl* |
| Main insulation | 2.768 $mm$ | *dij* |
| Width of a strand | 11.08 $mm$ | *bcus* |
| Height of a strand | 1.8 $mm$ | *hcus* |
| Strand insulation | 0.1 $mm$ | *dicu* |
| Winding length | 6.074 $m$ | *lav* |
| Cross section of stator bar | 1095 $mm^2$ | *Acus* |
| Stator winding resistance 20° | 0.00157 Ω | *Rdc20* |
| Stator winding resistance 75° | 0.00191 Ω | *Rdc75* |
| Stator winding resistance factor | 1.112 | *Kra* |
| Slot resistance factor | 1.188 | *Krad* |
| Maximum resistance factor | 1.642 | *Kmax* |

Table 14: Example generator *Rotor Parameters*

| Parameter | Value | Variable name |
|---|---|---|
| Minimum air gap | 32.17 $mm$ | *delta0* |
| Equivalent air gap | 40.02 $mm$ | *deltame* |
| Pole shoe width | 611.3 $mm$ | *bps* |
| Pole shoe height | 80 $mm$ | *hps* |
| Pole core width | 469 $mm$ | *bpk* |
| Pole core height | 300 $mm$ | *hpk* |
| Number of turns per pole | 54 | *nf* |
| Field current | 1177 $A$ | *If* |
| Field winding width | 69.66 $mm$ | *bcuf* |
| Field winding heigth | 5.26 $mm$ | *hcuf* |
| Cross section field winding | 360.5 $mm^2$ | *Af* |
| Current density field winding | 3.12 $A/mm^2$ | *Sf* |
| Rotor winding resistance 20° | 0.1719 Ω | *Rf20* |
| Rotor winding resistance 75° | 0.2091 Ω | *Rf* |
| Relative pole width | 0.7 | *alfar* |
| Number of damper bars | 7 | *NDs* |
| Cross section of damper bars | 366.6 $mm^2$ | *AcuD* |

Table 15: Example generator *Magnetic Parameters*

| Parameter | Value | Variable name |
|---|---|---|
| Air gap | 0.966 T | *Bdelta* |
| Stator core | 1.3 *T* | *Bys* |
| Stator tooth | 1.961 *T* | *Bdmax* |
| Pole core | 1.86 *T* | *Bdrmx* |
| Rotor ring | 1.525 *T* | *Byr* |
| | | |
| Air gap | 30 760 *At* | *Umdelta* |
| Stator core | 127 *At* | *Umys* |
| Stator tooth | 2098 *At* | *Umd* |
| Pole core | 1925 *At* | *Umdr* |
| Rotor ring | 1019 *At* | *Umyr* |
| | | |
| Relative magnetization | 1.797 *pu* | *Efpu* |
| Relative induced voltage | 1.055 *pu* | *Eipu* |
| Total required magnetization | 63 550 *At* | *Tetamn* |

Table 16: Example generator *Loss Calculations*

| Parameter | Value | Variable name |
|---|---|---|
| Iron loss Stator core | 209.1 *kW* | *PFe* |
| Windage and bearing losses | 353.8 *kW* | *Pfw* |
| Copper loss rotor | 158.9 *kW* | *Prnl* |
| | | |
| DC stator loss | 173.2 *kW* | *Pcusdc* |
| AC stator loss | 19.32 *kW* | *Pcusac* |
| Additional copper loss rotor | 126.7 *kW* | *Prfl* |
| Additional losses | 290.3 *kW* | *Padd* |
| Magnetization losses | 19.99 *kW* | *Pmagn* |
| | | |
| Total losses | 1332 *kW* | *Ptot* |

Table 17: Example generator *Reactances and Time Constants*

| Parameter | Value | Variable name |
|---|---|---|
| Armature reaction reactance (d-axis) | 0.9384 *pu* | *Xmd* |
| Armature reaction reactance (q-axis) | 0.5282 *pu* | *Xmq* |
| Leakage reactance | 0.0860 *pu* | *Xsigma* |
| Synchronous reactance (d-axis) | 1.024 *pu* | *Xdpu* |
| Synchronous reactance (q-axis) | 0.614 *pu* | *Xqpu* |
| Transient reactance (d-axis) | 0.2436 *pu* | *Xdt* |
| Sub-transient reactance (d-axis) | 0.1806 *pu* | *Xdtt* |
| Sub-transient reactance (q-axis) | 0.2077 *pu* | *Xdtt* |
| Transient time constant (d-axis) | 1.949 *s* | *Tdt* |
| Sub-transient time constant (d-axis) | 0.0630 *s* | *Tdtt* |
| Sub-transient time constant (q-axis) | 0.0465 *s* | *Tqtt* |

Table 18: Example generator *Thermal Calculations*

| Parameter | Value | Variable name |
|---|---|---|
| Cooling air flow | 30.85 $m^3/s$ | *qth* |
| Maximum air speed | 19.16 $m/s$ | *vim* |
| Maximum temperature rise in: | | |
| Stator winding | 72 $^\circ K$ | *Temp2(6)* |
| Stator tooth | 59 $^\circ K$ | *Temp2(4)* |
| Stator core | 52 $^\circ K$ | *Temp2(2)* |
| Stator end winding | 55 $^\circ K$ | *Temp2(7)* |
| Field winding | 50 $^\circ K$ | *Temp2(13)* |
| Rotor end winding | 39 $^\circ K$ | *Temp2(10)* |
| Pole core | 14 $^\circ K$ | *Temp2(14)* |
| Air temperature rise in: | | |
| End winding area | 2 $^\circ K$ | *Temp2(21)* |
| Air gap | 9 $^\circ K$ | *Temp2(20)* |
| Stator winding surrounding area | 21 $^\circ K$ | *Temp2(19)* |
| Middle of cooling duct | 23 $^\circ K$ | *Temp2(18)* |
| End of cooling duct | 23 $^\circ K$ | *Temp2(17)* |
| Outlet | 25 $^\circ K$ | *Temp2(16)* |

Table 19: Example generator *Mechanical Calculations*

| Parameter | Value | Variable name |
|---|---|---|
| Calculated moment of inertia | 233.5 $tm^2$ | *M2* |
| Weight of machine | 307.7 *tonne* | *Gtot* |

Table 20: Example generator *Harmonics*

| Parameter | Value | Variable name |
|---|---|---|
| Telephone Harmonic Factor | 0.0043 % | *THF* |

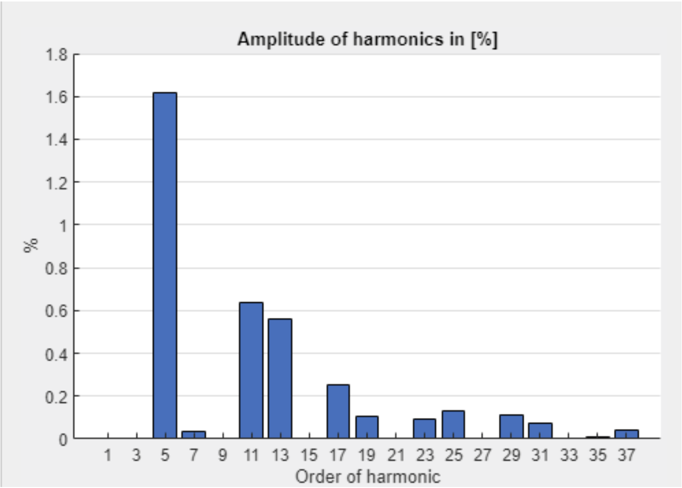


Figure 32: Example generator harmonics analysis

## 9.3 Discussion

One of the main goals was to develop the entire GenProg system into an *educational tool* and many of the decisions taken during the course of the project was taken to reflect this goal. In terms of achieving this goal the project was only partially successful. As discussed in section 9.3.1 and 9.3.2, some unmistakable shortcomings were discovered and not rectified in the given time frame. It was hoped that the input sanitation would guarantee the validity of the calculations, but this turned out not to be the case and a major rework was necessary, but not completed in full.

However the calculation report given to the user, and the cross-sectional view gives the user an *intuitive* way of interpreting the result from the calculations, that was not present in the *old* version of GenProg. And the introduction of a *default* value section in the source code helped reduce the number of *required* parameters which could confuse an inexperienced user. It should be noted that the current set of *default* parameters and their value should be expanded upon in the future to further reduce the amount of required parameters.

### 9.3.1 GenProg discussion

Initially it was believed a thorough input sanitation was enough to guarantee a valid result from GenProg, with only minor alterations the *core* source code. However, early in the projects time-cycle it was realized this was a naive assumption and a *major* restructure had to take place at some point or in order to rectify short comings of GenProg, and facilitate more advanced functionality in the future. The fact that that it gave the author of this report an in-depth understanding of the working principals of how a Generator is *actually* put together, and how GenProg worked, was more a of a useful by-product rather than the primary objective in itself.

The restructure revealed three shortcomings of GenProg. The only one that was adequately rectified and implemented, was the slot dimensions, and armature winding calculation *rework* described in section 5.3. The second shortcoming was the calculation of the rotor dimensions if the user has not defined some key pole dimensions. A proposed solution was described, but regrettably not implemented due to time constraints. As a result the proposed solution is pure speculation, and should be treated as such for the future work. The third shortcoming discovered is the magnetic calculation. Particularly the *magnetic voltage drop* calculation for the steel need a second look. It is believed a lot of the problems encountered with GenProg can be attributed this section of GenProg, and its highly nonlinear characteristic.

### 9.3.2 GenProg*App* discussion

Most of the work associated with the *App* Part of GenProg was *under-the-hood*, and not immediately obvious for the user if familiar with the initial version developed in the preceding semester. Effort was made to simplify future changes to the already established *elements* as these changes frequently occurred during discussion with supervisor. These *changes* (meaning moving of app elements to different panels, and renaming labels to clarify the language etc) initially required considerable work due to how the information was handled both inside and outside the *app environment*.

For future development the department expressed a desire to one day move the entire GenProg system *into the cloud* and make it *open source*. This would entail that all the app elements of GenProg has to be re created on an appropriate platform anyway. Developing the app in its current state has not been a priority for the author, as its benefit for future work is deemed *limited*. In fact for a properly *open-source* system the entire calcGenProg calculations should be rewritten in a different language. For example *Python*, as MATLAB code requires a software license, and is not really suited for web, or open-source applications.

It should also be noted that the *app designer* environment is somewhat limited in terms of advanced graphical user input functionality. Supervisor expressed a desire to implement advanced functionality like being able to click on specific objects in the cross sectional view, and then opening a dialog box with all the parameters for that component. For example clicking on one of the slot areas would open a dialog box with all the relevant parameters. This solution would be very intuitive, and makes a lot of sense from an educational point of view. However *app designer* does not include tools to easily create such functionality, so it would require the development of an entirely new system. This again probably is possible, but would require a phenomenal effort for it to work as intended. Of which there is no guarantee. In fact it would probably be more efficient to create a new GUI on a different platform entirely where such functionality is easier to implement. It is reasonable to assume that MATLAB, and the *app designer* platform has reached a ceiling of what is currently possible. It should be noted that currently the *app* source code is close to 4000 lines of code, and would not benefit from becoming any larger than it already is.

### 9.3.3 Cross-sectional view discussion

The work on the cross sectional view seamlessly continued from the previous semester, and was adapted to a fully vectorized variant. The old *point matrix* image from the previous semester project is completely deprecated. However it should be noted that a a *tiny* variant (1000 x 1000) could be used to *immediately* get a thumbnail-like representation of the calculated generator. The vectorized image still takes a couple of seconds to render (on a moderately powerful desktop PC), but has infinitely more fidelity compared to the old variant. Both the point matrix and fully vectorized variants use almost the same set set of matrices to generate the visual representation of the calculated generator. This proves that the time spent on developing said matrices was not wasted in the preceding semester. Even though it came at the expense of other intermediate objectives.

The addition of the indexing of the three different phases in the armature winding is only used in the cross sectional window, but as explained in section 6.2, it is hoped the system can be utilized in a future FEM analysis implementation.

### 9.3.4 Coding discipline

As was the case with the preceding project, the comments in the source code developed for this project is thorough and in-depth. The developer is painfully aware of what it is like to continue working on a project where a considerable amount of source code is written by someone else. For the readers convenience a complete list of variables is provided in appendix B for *core* calcGenProg source code. For the source code explicitly created for this project the comments accompanying an arbitrary parameters should be adequate. See appendix A for a few examples. For the complete source code please refer to attached folder.

### 9.3.5 Timesheet

Note that the resource allocation described in table 21 is an approximation, and not definitive as it was created after the fact, and not measured over the course of project. It is assumed a total of 750 hours is 100%.

Table 21: Resource allocation

| Intermediate objective | Percent | Hours |
|---|---|---|
| Vectorizing cross-sectional view | 15% | 110 |
| Cross-section restructure | 5 % | 37.5 |
| clacGenProg restructure | 40 % | 300 |
| GenProgApp | 10 % | 75 |
| Writing the report | 30 % | 225 |

## 9.4 Conclusion and future work

Unfortunately effort still remains for *GenProg* to fulfill its potential as a design software for salient pole generators. However immediately succeeding work should focus on the following tasks as the foundation is more or less completed.

- The rotor calculations should be reworked as described in section 5.4.
- The magnetic calculations needs a second revision.
- Implementation of FEM analysis of the calculated machine, and later as a an integral part of the calculations themselves.

Long term goals was expressed by the department for a web-based solution. Due to the nature of such a solution, it is the authors conviction that it is mutually exclusive with the implementation of FEM analysis software. The first reason being a eventual web application should move away from MATLAB, and use another language, which makes COMSOL interaction more complicated. The second reason is speed. The current version of calcGenProg is extremely fast, and requires relatively little computing power to run. From past experience a FEM analysis does use a lot of computing power, and as such is not well suited for the linear iteration process described in section 5.4 of this report. A more sophisticated method of iterating might be required in order to reduce the *amount* of FEM simulations required for a single calcGenProg calculation, but even this has limited potential, as a fully implemented FEM analysis will likely require more computing power to perform in a timely manner. This will stand in stark contrast to the current version of calcGenProg that require very little in this regard.

# Bibliography

[1] Heninng Johansen. *Beregning av spenninger generelt.* 2017.

[2] *Petroleum and natural gas industries — Offshore production installations — Heating, ventilation and air-conditioning.* Standard. International Organization for Standardization, Mar. 2018.

[3] Ivar Vikan and Alexander Lundseng. *Beregning av generatorer ved modernisering av kraftverk (NTNU).* 2010.

[4] Ivar Vikan and Alexander Lundseng. *Beregning av Vannkraftgeneratorer (NTNU).* 2009.

[5] Proff E. Westgaard and O.W. Andersen. *Dimensjoneringsesempel for synkronmaskin.* 1965.

[6] Wikipedia. *Dynamic pressure — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Dynamic%20pressure&oldid=1015639325. [Online; accessed 06-May-2021]. 2021.

[7] Wikipedia. *Von Mises yield criterion — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Von%20Mises%20yield%20criterion&oldid=1006628220. [Online; accessed 17-April-2021]. 2021.

# Appendix

## A Matlab source Code

### A calcGenProg

```
function [windLay,sec] = calcWindLayout(Qs,Np_,Ww)
%function for calculating winding layout...


%quick input sanetization...

pp = Np_/2;                                 %pole pairs
sec = gcd(Qs,pp);                           %number of sectors...
nrPpPS = pp/sec;                            %number of Pole-pairs per sector...
drad = (sec/Qs)*360*nrPpPS;                 %degree step length

eldeg = 0:drad: nrPpPS * 360 - drad;        %degree vector with all slot in a sector and their cor

windLay = zeros(3 , Qs/sec);                              %empty vector for RST phase...
for j = 1:length(eldeg)
    windLay(:,j) = revolver(eldeg(j));                  %uses "revolver" to determine correct phas
end

windLay = [windLay; circshift(windLay,Ww,2) * -1];      %phase shift for second layer...
end
```

Listing 1: Function for generating winding layout

```
function [out] = revolver(deg)
%function for returning R S T phase of the winding layout for a given electrical angle...

deg = rem(deg,360);         %reducing angle

if deg >= 0 && deg < 60
    out = [1;0;0];      %R positive
elseif deg >= 60 && deg < 120
    out = [0;0;-1];     %T negative
elseif deg >= 120 && deg < 180
    out = [0;1;0];      %S positive
elseif deg >= 180 && deg < 240
    out = [-1;0;0];     %R negative
elseif deg >= 240 && deg < 300
    out = [0;0;1];      %T positive
else
    out = [0;-1;0];     %S negative
end
end
```

Listing 2: Revolver function

## B Cross-sectional view

```
function plotp2p(app,x,y,color)
%function for plotting. Creates a "polygon" from boundary coordinates.

[rows,colm] = size(x);
    for i = 1:rows
        %remove NaN elements
        delete = false(1,colm);
        if any(isnan(x(i,:)))
            tempx = x(i,:);
            tempy = y(i,:);
            for j = 1:length(tempx)
                if isnan(tempx(j))
                    delete(j) = true;
                end
            end
            tempx(delete) = [];
            tempy(delete) = [];
        else
            tempx = x(i,:);
            tempy = y(i,:);
        end
        pol = polyshape(tempx,tempy);
%         pol = plot(app.UIAxesCS , pol);
%         pol.FaceColor = color;
%         pol.FaceAlpha = 1;
%         hold(app.UIAxesCS, 'on')
        pol = plot(pol);
        pol.FaceColor = color;
        pol.FaceAlpha = 1;
        hold on

    end
end
```

Listing 3: Function *plotp2p.m*

# B Variable name and description

## A calcGenProg

Alphabetical sorting of *most* of the parameters utilized by calcGenProg. Please note in order to *extract* the alphabetized list the *old* GenProg had to be used. Some parameters *may* be deprecated.

Table 22: calcGenProg variables A

| Variable | Full name | Type | Comment |
|---|---|---|---|
| ADs | Permeability factor for a damperbar | *var* | |
| Acus | Area single armature bar | *var* | |
| Afadd | Extra area for cooling fin rotor winding | *var* | |
| Amin | Unknown | *var* | Related to cooling |
| Ams | Estimated armature loading | *var* | Based on *utilization factor* |
| Apk | Permeability pole core - pole core | *var* | |
| As | Surface armature loading | *var* | |
| Asigma | Permeability pole - pole | *var* | |
| aSf | Delta current density field winding | *var* | Pole iteration |
| af | Ratio between middle and max field winding length | *const* | |
| alfar | Relative pole arc | *const* | |
| aqth1 | Unkonwn | *var* | Initiation variable |
| as | Distance extreme damper bars | *var* | |
| ath11 | Thermal coefficient armature winding - stator iron | *const* | |
| ath16 | Unknown thermal coefficient | *var* | |
| ath27 | Unknown thermal coefficient | *var* | |
| ath3 | Unknown thermal coefficient | *var* | |
| ath8 | Unknown thermal coefficient | *var* | |
| AcuD | Area single damper bar | *var* | |
| Af | Area single field winding | *var* | |
| Aftot | Total field winding CS area | *var* | |
| Ampk | Leakage coefficient | *var* | |
| Amsdelta | Estimated armature loading | *var* | Based on *utilization factor* |
| Aps | Permeability pole shoe - pole shoe | *var* | |
| Ascm | Surface armature loading | *var* | in centimetre |
| a | Coefficient for calculating pole leakege flux | *var* | |
| aaSf | Unknown initiation variable | *var* | pole iteration |
| aksq | Decleration variable for skewing | *var* | |
| alphaKra | Coefficient for determining AC resistance | *var* | |
| aqth2 | Unknown | *var* | Initiation variable |
| ath1 | Thermal coefficient<br>stator frame - stator iron | *const* | |
| ath13 | Unknown thermal coefficient | *var* | |
| ath26 | Unknown thermal coefficient | *var* | |
| ath28 | Unknown thermal coefficient | *var* | |
| ath43 | Unknown thermal coefficient | *var* | |

Table 23: calcGenProg variables B

| Variable | Description | Type | Comment |
|---|---|---|---|
| Bd | Maximum tooth flux density | *var* | |
| Bdmax | Maximum tooth flux density | *var* | As a function of the slot height |
| Bdrmx | Flux density rotor yoke | *var* | |
| Bxx | Unknown variable | *var* | Substitutes Bdrmx |
| Bys | Flux density stator yoke | *var* | |
| bcs | Core section length | *var* | |
| bcuf3 | Temp field winding width | *var* | Pole iteration |
| bcus_ | Width armature strand | *var* | Input |
| bdmin | Minimum stator tooth width | *var* | |
| bf | Inner width field winding | *var* | |
| bif_ | Insulation field winding | *var* | Input |
| bpk_ | Pole core width | *var* | Input |
| bu | Slot width | *var* | |
| bv | Cooling duct length | *var* | |
| Bdelta | Flux density air gap | *var* | |
| Bdmin | Minimum tooth flux density | *var* | As a function of the slot height |
| Bpmx | Maximum flux density pole core | *var* | Input, Not utilized |
| Bymx | Maximum flux density yoke | *var* | Input |
| bD | Maximum flux density stator tooth | *var* | |
| bcu | Total armature copper width | *var* | |
| bcuf_ | Field winding width | *var* | Input |
| bd | Stator tooth width | *var* | |
| beta | Empirical coefficient for calculating stator dimensions | *const* | |
| bi | Field - core insulation | *var* | |
| bi_ | Field - core insulation | *var* | Input |
| bps | Pole shoe width | *var* | |
| budbd | Slot tooth ratio | *var* | |
| bve | Equivelant air gap width | *var* | |
| Bdelta_ | Flux density air gap | *var* | Input |
| Bdrmn | Flux density top of pole core | *var* | |
| Btmx | Maximum tooth flux density | *var* | |
| Byr | Flux density rotor yoke | *var* | |
| bcr | Field winding cooling fin length | *const* | |
| bcuf | Field winding width | *var* | |
| bcus | Width armature strand | *var* | |
| bdmax | Maximum stator tooth width | *var* | |
| beta2 | Unknown variable | *var* | |
| bif | Insulation field winding | *var* | |
| bpk | Pole core width | *var* | |
| bps_ | Pole shoe width | *var* | Input |
| bu_ | Slot width | *var* | Input |

Table 24: calcGenProg variables C

| Variable | Description | Type | Comment |
|---|---|---|---|
| C | Utilization factor | *var* | |
| C2 | Temp utilization factor | *var* | Pole iteration |
| C2delta | Temp utilization factor variable | *var* | Pole iteration |
| Cend | Coefficient for permeability for end area | *const* | |
| Ckonst | Temp utilization factor variable | *var* | Pole iteration |
| Cm | Unknown variable | *var* | Related to calculating drag losses |
| Cm2 | Unknown variable | *var* | Related to calculating drag losses |
| Cosphi | Powerfactor | *var* | |
| C_ | Utilization factor | *var* | Input |
| cp | Thermal conductivity insulation | *const* | |

Table 25: calcGenProg variables D (D2l - d9)

| Variable | Description | Type | Comment |
|---|---|---|---|
| D2l | Empirical coefficient for calculating stator dimensions | *var* | |
| Df2 | Diameter through centre field winding | *var* | |
| Dps | Diameter through centre pole shoe | *var* | |
| Dry | Rotor yoke outer diameter | *var* | |
| d12 | Unknown thermal coefficient | *var* | |
| d2 | Unknown thermal coefficient | *var* | |
| d38 | Unknown thermal coefficient | *var* | |
| d42 | Unknown thermal coefficient | *var* | |
| dTmx | Maximum delta temprature | *var* | |
| delta0_ | Air gap length | *var* | Input |
| deltame | Equivalent air gap | *var* | |
| deltath16 | Unknown thermal coefficient | *var* | |
| dij | Main insulation | *var* | |
| dl12 | Unknown thermal coefficient | *var* | |
| dl42 | Unknown thermal coefficient | *var* | |
| D4l | Empirical coefficient for calculating stator dimensions | *var* | |
| Di | Stator inner diameter | *var* | |
| Dr | Rotor yoke middle diameter | *var* | |
| Dw | Unknown variable | *var* | |
| d15 | Unknown thermal coefficient | *var* | |
| d29 | Unknown thermal coefficient | *var* | |
| d39 | Unknown thermal coefficient | *var* | |
| d7 | Unknown thermal coefficient | *var* | |
| dTmxt | Unknown variable | *var* | Related to thermal calculations |
| delta2 | Load angle | *var* | |
| deltamx | Maximum actual air gap | *var* | |
| deltath28 | Unknown thermal coefficient | *var* | |
| dij_ | Main insulation | *var* | Input |
| dl17 | Unknown thermal coefficient | *var* | |
| drs | Roebel sepparator width | *var* | |
| Delta_ps | Unknown variable | *var* | |
| DiMax | Maximum stator inner diameter | *var* | |
| Dr2 | Unknown variable | *var* | |
| Dy | Stator outer diameter | *var* | |
| d17 | Unknown thermal coefficient | *var* | |
| d3 | Unknown thermal coefficient | *var* | |
| d4 | Unknown thermal coefficient | *var* | |
| d9 | Unknown thermal coefficient | *var* | |

Table 26: calcGenProg variables D (delta0 - dl41)

| Variable | Description | Type | Comment |
|---|---|---|---|
| delta0 | Air gap length | *var* | |
| deltade | Equivalent air gap d-axis | *var* | |
| deltaqe | Equivalent air gap q-axis | *var* | |
| dicu | Armature strand insulation | *var* | |
| diw | Turn insulation | *var* | |
| dl29 | Unknown thermal coefficient | *var* | |
| drs_ | Roebel sepparator width | *var* | Input |
| Df | Diameter through centre of field winding | *var* | |
| Di_ | Stator inner diameter | *var* | Input |
| Dri | Outer diameter rotor axle | *const* | |
| d1 | Unknown thermal coefficient | *var* | |
| d18 | Unknown thermal coefficient | *var* | |
| d30 | Unknown thermal coefficient | *var* | |
| d41 | Unknown thermal coefficient | *var* | |
| dTa | Allowed temperature rise in cooling air | *const* | |
| delta0e | Equivalent air gap | *var* | |
| deltadef | Intermediate variable | *var* | Related to magnetic voltage drop |
| deltaqef | Intermediate variable | *var* | Related to magnetic voltage drop |
| dicu_ | Armature strand insulation | *var* | Input |
| diw_ | Turn insulation | *var* | Input |
| dl41 | Unknown thermal coefficient | *var* | |

Table 27: calcGenProg variables E

| Variable | Description | Type | Comment |
|---|---|---|---|
| EP | Extra parameters | *vector* | |
| EQpuCx | Unknown intermediate variable | *var* | Per unit |
| Eb | Intermediate relative magnetization | *var* | |
| Eff | Efficiency | *var* | |
| Efpu | Relative magnetization | *var* | Absolute value |
| EfpuCx | Relative magnetization | *var* | |
| Ei | Intermediate relative induced voltage | *var* | |
| Eipu | Relative induced voltage | *var* | Absolute value |
| EipuCx | Relative induced voltage | *var* | |
| epsilon | Slot reduction | *var* | |

Table 28: calcGenProg variables F

| Variable | Description | Type | Comment |
|---|---|---|---|
| FId | Maximal flux density in single stator tooth | *var* | |
| FIm | Maximal flux density through armature winding | *var* | |
| FIsigma | Pole leakage flux | *var* | |
| FIsigmaps | Pole shoe leakage flux | *var* | |
| Fa | Intermediate armature reaction | *var* | |
| Fdelta | Intermediate magnetic voltage drop for air gap | *var* | |
| FeOld_ | *Old* Iron sheets | *bool* | Indicates use of old iron sheets |
| Fg | Unknown thermal coefficient | *var* | |
| Fw | Unknown magnetic variable | *var* | |
| f | frequency | *var* | |
| fi2 | Phase shift during nominal load | *var* | |
| ficu | Resistance coefficient | *var* | |
| fimksi | Unknown intermediate variable | *var* | |
| fj | Coefficient net vs gross iron length | *const* | |
| fsp | Unknown thermal coefficient | *const* | |
| fw | Average winding factor | *const* | |

Table 29: calcGenProg variables G

| Variable | Description | Type | Comment |
|---|---|---|---|
| G | Thermal conductivity matrix | *mat* | |
| G13 | Unknown thermal coefficient | *var* | |
| G18 | Unknown thermal coefficient | *var* | |
| G27 | Unknown thermal coefficient | *var* | |
| G3 | Unknown thermal coefficient | *var* | |
| G43 | Unknown thermal coefficient | *var* | |
| GD2 | Total moment of inertia | *var* | |
| GD2flywheel | Moment of inertia for flywheel | *var* | |
| Gadd | Additional mass from misc components | *var* | |
| Gpk | Total pole core mass | *var* | |
| Gpspm | Pole shoe mass per meter length | *var* | |
| Gtot | Total machine mass | *var* | |
| gamma2 | Mechanical degree between two adjacent poles | *var* | |
| G1 | Unknown thermal coefficient | *var* | |
| G15 | Unknown thermal coefficient | *var* | |
| G2 | Unknown thermal coefficient | *var* | |
| G28 | Unknown thermal coefficient | *var* | |
| G38 | Unknown thermal coefficient | *var* | |
| G7 | Unknown thermal coefficient | *var* | |
| GD2add | Moment of inerta for misc components | *var* | |
| GDp2 | Total moment of inertia for poles and field windings | *var* | |
| Gf | Total field winding mass | *var* | |
| Gpkpm | Pole core mass per meter length | *var* | |
| Gr | Rotor yoke mass | *var* | |
| g | Average phase shift between two rods in the same slot | *var* | |
| gammaFe | Density iron | *const* | |
| G11 | Unknown thermal coefficient | *var* | |
| G16 | Unknown thermal coefficient | *var* | |
| G26 | Unknown thermal coefficient | *var* | |
| G29 | Unknown thermal coefficient | *var* | |
| G39 | Unknown thermal coefficient | *var* | |
| G8 | Unknown thermal coefficient | *var* | |
| GD2add_ | Moment of inerta for misc components | *var* | Input |
| GDr2 | Moment of inertia rotor yoke | *var* | |
| Gflywheel | Flywheel mass | *var* | |
| Gps | Total pole shoe mass | *var* | |
| Gsw | Total armature winding mass | *var* | |
| gamma | Coefficient related to minimum air gap | *const* | |
| gammacu | Density copper | *const* | |

Table 30: calcGenProg variables H

| Variable | Description | Type | Comment |
|---|---|---|---|
| Hdrmn | Field strength top of core | *var* | |
| h2 | Unknown thermal coefficient | *var* | |
| harm | Harmonic data matrix | *mat* | |
| harma | Intermediate harmonic data matrix | *mat* | |
| hcuf | Single field winding height | *var* | |
| hcus_ | Armature strand thickness | *var* | Input |
| hf | Total field winding height | *var* | |
| hgls_ | Slot wedge spacer + spring | *var* | Input |
| hk4 | Unknown harmonic variable | *var* | |
| hm | Bar separator | *var* | |
| hpk_ | Pole core height | *var* | Input |
| hpt | Pole tooth height | *var* | |
| hspk_ | Slot wedge height | *var* | Input |
| hyr | Rotor yoke height | *var* | |
| Hdrmx | Field strength bottom of pole core | *var* | |
| h3 | Unknown variable | *var* | |
| harm2 | Harmonic component matrix | *mat* | |
| harmb | Intermediate harmonic data matrix | *mat* | |
| hcuf_ | Single field winding height | *var* | Input |
| hds | Distance slot wedge and air gap | *var* | |
| hf_ | Total field winding height | *var* | Input |
| hk | Harmonic indexing variable | *var* | |
| hk44 | Harmonic helper variable | *var* | |
| hm_ | Bar separator | *var* | Input |
| hps | Height pole shoe | *var* | |
| hs | Total slot height | *var* | |
| hstav | Armature bar heigth | *var* | |
| hyr_ | Rotor yoke height | *var* | Input |
| h1 | Unknown thermal variable | *var* | |
| h7 | Unknown thermal variable | *var* | |
| harm3 | Intermediate harmonic data matrix | *mat* | |
| hcu | Total copper heigth in one slot | *var* | |
| hcus | Armature strand thickness | *var* | |
| hds_ | Distance slot wedge and air gap | *var* | Input |
| hgls | Slot wedge spacer + spring | *var* | |
| hk3 | Harmonic indexing variable | *var* | |
| hkr | Field collar height | *var* | |
| hpk | Pole core height | *var* | |
| hps_ | Pole shoe height *var* | *Input* | |
| hspk | Slot wedge height | *var* | |
| hs_ | Total slot height | *var* | Input |
| hys | Stator yoke height | *var* | |

Table 31: calcGenProg variables I

| Variable | Description | Type | Comment |
|---|---|---|---|
| IDtot | Maximum damper bar current | *var* | |
| INT | Index variable for magnetic voltage drop | *var* | |
| Ic | Armature winding current | *var* | |
| Idpu | Current d-axis | *var* | Per unit |
| IdpuCx | Complex current d-axis | *var* | Per unit |
| If | Field current | *var* | |
| In | Nominal current | *var* | |
| Inm | Armature bar current | *var* | |
| Inpu | Nominal current | *var* | |
| InpuCx | Complex nominal current | *var* | |
| InputFile | Excel worksheet file name | *str* | Deprecated |

Table 32: calcGenProg variables J

| Variable | Description | Type | Comment |
|---|---|---|---|
| None | | | |

Table 33: calcGenProg variables K

| Variable | Description | Type | Comment |
|---|---|---|---|
| Kch | Unknown constant | *const* | |
| Kfl | Coefficient for air gap shape | *const* | |
| Kr_aver | Unknown variable | *var* | |
| Krad | Resistance coefficient for armature winding | *var* | |
| Krau | Resistance coefficient for top layer | *var* | |
| k1 | Intermediate variable for calculating permeability factor | *var* | |
| kC | Carters coefficient | *var* | |
| kCs | Carters coefficient for armature slot | *var* | |
| kFed | Tooth saturation coefficient | *var* | |
| kL | Skin effect coefficient | *var* | |
| kckj | Carters coefficient for cooling duct | *var* | |
| kdx | d-axis coefficient | *var* | |
| kfi | Field winding leakage reactance coefficient | *var* | |
| km | Ratio air gap induction idle vs middle | *var* | |
| kmf | Pole flux leakage coefficient | *var* | |
| kpw | Unknown cooling coefficient | *var* | |
| ksi | Reduced conductor height | *var* | |
| kvw | Bearing and fan loss coefficient | *var* | |
| kwsv | nth harmonic winding coefficient | *var* | |
| Kf | Frequency correction factor | *const* | |
| Kmx | Resistance coefficient for top strand | *var* | |
| Kra | Resistance factor for armature winding | *var* | |
| Krao | Resistance coefficient for bottom layer | *var* | |
| k | Damper bar resistance coefficient referred to stator | *var* | |
| k2 | Intermediate variable for *number of slots* determination | *var* | |
| kCr | Carters coefficient for damper bars | *var* | |
| kFe | Iron filling factor | *var* | |
| kFey | Yoke saturation coefficient | *var* | |
| k_Af | Cooling fin cross section coefficient | *var* | |
| kd | Unknown dividing factor | *var* | |
| kf | Unknown conversion factor | *const* | |
| kl | Factor for linear rise in driving field winding | *var* | |
| kmek | Surface roughness coefficient | *const* | |
| kp | Unknown variable | *var* | |
| kqx | q-axis coefficient | *var* | |
| ksq | Skewing factor | *var* | |
| kw | Winding factor | *var* | |

Table 34: calcGenProg variables L (l1 - lsq)

| Variable | Description | Type | Comment |
|---|---|---|---|
| l1 | Unknown thermal parameter | *var* | |
| l2 | Unknown thermal parameter | *var* | |
| l7 | Unknown thermal parameter | *var* | |
| lambdaair | Thermal conductivity cooling air | *var* | |
| lambdad | Permeability factor for single tooth | *var* | |
| lambdalew | Permeability factor for winding head(?) | *const* | |
| lambdau | Permeability factor for slot with two winding layers | *var* | |
| lambdaw | Permeability factor for winding head(?) | *const* | |
| lav | Total armature winding length | v*var* | |
| lav_ | Total armature winding length | v*var* | Input |
| lb | Gross iron length | *var* | |
| lb_ | Gross iron length | *var* | Input |
| LC | Loss calculations | *vec* | Output |
| Ld | Tooth leakage inductance | *var* | |
| Ldelta | Air gap leakage inductance | *var* | |
| Ldeltav | Intermediate leakage inductance variable | *var* | |
| Ldt | Transient d-axis inductance | *var* | |
| Ldtt | Sub-transient d-axis inductance | *var* | |
| Le | End plate thickness | *const* | |
| lew | Unknown intermediate variable | *var* | |
| Lfmd | Mean field winding length | *var* | |
| Lfmx | Maximum field winding length | *var* | |
| Lfp | Inner field winding length | *var* | |
| lfs | Copper length in single phase winding | *var* | |
| Lfsigma | Field winding leakage inductance | *var* | |
| lm | Equivalent iron length | *var* | |
| Lma | Armature reaction inductance | *var* | |
| Lmd | Armature reaction inductance d-axis | *var* | |
| LmD | Damper bar leakage inductance | *var* | Referred to stator |
| LmDd | Damper bar leakage d-axis inductance | *var* | Referred to stator |
| LmDq | Damper bar leakage d-axis inductance | *var* | Referred to stator |
| Lmq | Armature reaction inductance q-axis | *var* | |
| ln | Net iron length | *var* | |
| Lqtt | Sub-transient q-axis inductance | *var* | |
| lrac | Intermediate Resistance calculation variable | *var* | |
| lrr | Rotor ring length | *var* | |
| Lsigma | Total leakage inductance | *var* | |
| lspolh | Intermediate thermal calculation variable | *var* | |
| Lsq | Twist leakage inductance | *var* | |

Table 35: calcGenProg variables L (lth1 - Lw)

| Variable | Description | Type | Comment |
|---|---|---|---|
| lth1 | Thermal conductivity structural steel | *const* | |
| lth12 | Thermal conductivity winding insulation | *var* | |
| lth15 | Thermal conductivity copper | *const* | |
| lth17 | Thermal conductivity winding insulation | *var* | |
| lth18 | Thermal conductivity copper | *const* | |
| lth2 | Thermal conductivity steel | *const* | |
| lth29 | Thermal conductivity winding insulation | *var* | |
| lth3 | Unnknown thermal constant | *const* | |
| lth30 | Thermal conductivity copper | *const* | |
| lth38 | Thermal conductivity steel | *const* | |
| lth39 | Thermal conductivity copper | *const* | |
| lth4 | Thermal conductivity steel | *const* | |
| lth41 | Thermal conductivity steel | *const* | |
| lth42 | Thermal conductivity winding insulation | *var* | |
| lth7 | Thermal conductivity steel | *const* | |
| lth9 | Thermal conductivity steel | *const* | |
| lthl41 | Unknown thermal constant | *const* | |
| Lu | Slot leakage inductance | *var* | |
| Lw | winding head leakage inductance | *var* | |

Table 36: calcGenProg variables M

| Variable | Description | Type | Comment |
|---|---|---|---|
| m | Number of phases | *const* | |
| M | Moment of inertia | *var* | |
| M2 | Moment of inertia for entire machine | *var* | Deprecated |
| MC | Mechanical calculations | *vec* | Output |
| mcu | Unknown intermediate variable | *var* | |
| MD | Main data, or Nameplate data | *vec* | Output |
| mds | Total tooth mass | *var* | |
| mFe | Total staor mass | *var* | Without teeth |
| MP | Magnetic parameters | *vec* | Outupt |
| my0 | Permeability air | *const* | |
| myair | Viscosity air | *const* | |
| myrpk | Assumed permeability pole core | *const* | |
| myryr | Assumed permeability rotor yoke | *const* | |

Table 37: calcGenProg variables N

| Variable | Description | Type | Comment |
|---|---|---|---|
| N | Unknown constant | *const* | |
| Ncr | Number of cooling field windings | *var* | |
| ndl | Number of strands per armature turn | *var* | |
| ndlh | Number of vertical armature strands | *var* | |
| ndlh_ | Number of vertical armature strands | *var* | Input |
| ndlp | Number of horizontal armature strands | *var* | |
| ndlp_ | Number of horizontal armature strands | *var* | Input |
| ndl_ | Number of strands per armature turn | *var* | Input |
| NDs | Number of damper bars per pole | *var* | |
| NDs_ | Number of damper bars per pole | *var* | |
| nf | Number of field winding turns | *var* | |
| nf_ | Number of field winding turns | *var* | Input |
| Np | Number of poles | *var* | |
| nr | Runaway speed | *var* | |
| ns | Synchronous rotational speed | *var* | |
| Ns | Number of armature windings in series per turn | *var* | |
| Nsp | Number of stator slots above one pole | *var* | |
| Nu | Unknown intermediate variable | *var* | |
| nv | Number of stator cooling ducts | *var* | |
| Nw | Unknown variable | *var* | |

Table 38: calcGenProg variables O

| Variable | Description | Type | Comment |
|---|---|---|---|
| OBra | *Actual* surface loading | *var* | Rotor |
| OBrt | *Allowable* surface loading | *var* | Rotor |
| OBsa | *Actual* surface loading | *var* | Stator |
| OBst | *Allowable* surface loading | *var* | Stator |

Table 39: calcGenProg variables P

| Variable | Description | Type | Comment |
|---|---|---|---|
| p | Pole pairs | *var* | |
| P | Rated power | *var* | |
| P10 | Iron losses for 1kg of steel | *var* | |
| P10_ | Iron losses for 1kg of steel | *var* | Input |
| P2 | Unknown thermal coefficient | *var* | |
| Padd | Auxillary losses | *var* | |
| Pcusac | AC armature losses | *var* | |
| Pcusdc | DC armature losses | *var* | |
| Pend | Unknown intermediate loss variable | *var* | |
| Pend0 | Iron losses at end of core, idle | *var* | |
| PendL | Iron losses at end-plate, nominal | *var* | |
| PendP | Iron losses at end-plate, idle | *var* | |
| PFe | Total iron losses in stator yoke | *var* | |
| PFed | Iron losses stator teeth | *var* | |
| PFey | Iron losses in stator yoke | *var* | |
| Pfw | Bearing and fan losses | *var* | |
| Pmagn | Excitation losses | *var* | |
| Pmagn_ | Excitation losses | *var* | Input |
| pnr | Number of parallel circuits | *var* | |
| pnr_ | Number of parallel circuits | *var* | Input |
| polklaring | Pole tolerance | *var* | |
| Pps | Unknown intermediate loss variable | *var* | |
| Ppsfl | Unknown intermediate loss variable | *var* | |
| Ppsfl_a | Unknown intermediate loss variable | *var* | |
| Ppsnl | Pole shoe losses | *var* | |
| Pqth | Losses that must be transported out of the machine | *var* | |
| Pr | Rotor losses | *var* | |
| Pre | Rotor losses at end of pole | *var* | |
| Prfl | Additional rotor losses due to full load condition | *var* | |
| Prhow | Total surface drag losses | *var* | |
| Prhow1 | Intermediate loss variable | *var* | |
| Prhow2 | Intermediate loss variable | *var* | |
| Prl | Rotor losses *along* pole core | *var* | |
| Prnl | Rotor idle losses | *var* | |
| psicu | Intermediate resistance variable | *var* | |
| psimksi | Intermediate harmonic variable | *var* | |
| Ptot | Total losses | *var* | |
| Pwarming | Losses that cause heating of ventilation air | *var* | |

Table 40: calcGenProg variables Q

| Variable | Description | Type | Comment |
|---|---|---|---|
| q | Number of slots per phase and pole | *var* | |
| qm | Number of voltage phase vectors | *var* | |
| Qs | Number of slots | *var* | |
| Q_ | Number of slots | *var* | Input |
| qth | Cooling air-flow | *var* | |
| qth_ | Cooling air-flow | *var* | Input |

Table 41: calcGenProg variables R (R15 - Rth15)

| Variable | Description | Type | Comment |
|---|---|---|---|
| R15 | Unknown thermal parameter | *var* | |
| R18 | Unknown thermal parameter | *var* | |
| R2 | Unknown thermal parameter | *var* | |
| R30 | Unknown thermal parameter | *var* | |
| R38 | Unknown thermal parameter | *var* | |
| R39 | Unknown thermal parameter | *var* | |
| R7 | Unknown thermal parameter | *var* | |
| Rac | AC armature resistance | *var* | |
| Raco | Resistance top armature bar | *var* | |
| Racpu | AC armature resistance | *var* | Per-unit |
| Racu | Reistance bottom armature bar | *var* | |
| Rdc | DC armature resistance | *var* | |
| Rdc20 | DC armature resistance at 20 $^{\circ}C$ | *var* | |
| Rdcprm | DC armature resistance per metre | *var* | |
| Re | Thermal conductivity matrix | *mat* | |
| Re2 | Inverse thermal conductivity matrix | *mat* | |
| Re3 | Intermediate matrix for use in thermal calculation | *mat* | |
| Re3x | Intermediate thermal index variable | *var* | |
| Re3y | Intermediate thermal index variable | *var* | |
| Rf | Field winding resistance at 75 $^{\circ}C$ | *var* | |
| Rf20 | Field winding resistance at 20 $^{\circ}C$ | *var* | |
| rho20 | Resistivity coefficient at 20 $^{\circ}C$ | *const* | |
| rho75 | Resistivity coefficient at 75 $^{\circ}C$ | *const* | |
| rhoth | Density of air at 40 $^{\circ}C$ | *const* | |
| Rlrdelta | Reynolds number for end of rotor | *var* | |
| Rlsdelta | Reynolds number for rotor surface | *var* | |
| RmD | Damper bar resistnace | *var* | Referred to stator |
| Rmf | Field winding resistance | *var* | Referred to stator |
| RP | Rotor parameters | *vec* | Output |
| Rqth | Unknown thermal parameter | *var* | |
| rr | Pole shoe radius | *var* | |
| Rrdc | Rotor resistance | *var* | |
| Rref | Resistance refferance value | *var* | |
| RT | Reactances and time constant vector | *vec* | Output |
| Rth1 | Thermal resistance stator iron - stator frame | *var* | |
| Rth10 | Thermal resistance heat generated in stator iron | *var* | |
| Rth11 | Thermal resistance armature winding - stator teeth | *var* | |
| Rth12 | Thermal resistance armature rod centre - outer edge insulation | *var* | |
| Rth13 | Thermal resistance armature rod - cooling air | *var* | |
| Rth14 | Unknown thermal resistance | *var* | |
| Rth15 | Thermal resistance armature rod centre - armature rod end | *var* | |

Table 42: calcGenProg variables R (Rth16 - Rth9)

| Variable | Description | Type | Comment |
|---|---|---|---|
| Rth16 | Unknown thermal resistance | *var* | |
| Rth17 | Unknown thermal resistance | *var* | |
| Rth18 | Unknown thermal resistance | *var* | |
| Rth19 | Unknown thermal resistance | *var* | |
| Rth2 | Unknown thermal resistance | *var* | |
| Rth20 | Unknown thermal resistance | *var* | |
| Rth21 | Unknown thermal resistance | *var* | |
| Rth22 | Unknown thermal resistance | *var* | |
| Rth23 | Unknown thermal resistance | *var* | |
| Rth24 | Unknown thermal resistance | *var* | |
| Rth25 | Unknown thermal resistance | *var* | |
| Rth26 | Unknown thermal resistance | *var* | |
| Rth27 | Unknown thermal resistance | *var* | |
| Rth28 | Unknown thermal resistance | *var* | |
| Rth29 | Unknown thermal resistance | *var* | |
| Rth3 | Unknown thermal resistance | *var* | |
| Rth30 | Unknown thermal resistance | *var* | |
| Rth31 | Unknown thermal resistance | *var* | |
| Rth32 | Unknown thermal resistance | *var* | |
| Rth33 | Unknown thermal resistance | *var* | |
| Rth34 | Unknown thermal resistance | *var* | |
| Rth35 | Unknown thermal resistance | *var* | |
| Rth36 | Unknown thermal resistance | *var* | |
| Rth37 | Unknown thermal resistance | *var* | |
| Rth38 | Unknown thermal resistance | *var* | |
| Rth39 | Unknown thermal resistance | *var* | |
| Rth4 | Unknown thermal resistance | *var* | |
| Rth40 | Unknown thermal resistance | *var* | |
| Rth41 | Unknown thermal resistance | *var* | |
| Rth42 | Unknown thermal resistance | *var* | |
| Rth43 | Unknown thermal resistance | *var* | |
| Rth5 | Unknown thermal resistance | *var* | |
| Rth6 | Unknown thermal resistance | *var* | |
| Rth7 | Unknown thermal resistance | *var* | |
| Rth8 | Unknown thermal resistance | *var* | |
| Rth9 | Unknown thermal resistance | *var* | |

Table 43: calcGenProg variables S

| Variable | Description | Type | Comment |
|---|---|---|---|
| s | Skewing | *var* | |
| S11 | Unknown thermal parameter | *var* | |
| S12 | Unknown thermal parameter | *var* | |
| S13 | Unknown thermal parameter | *var* | |
| S15 | Unknown thermal parameter | *var* | |
| S16 | Unknown thermal parameter | *var* | |
| S17 | Unknown thermal parameter | *var* | |
| S18 | Unknown thermal parameter | *var* | |
| S26 | Unknown thermal parameter | *var* | |
| S27 | Unknown thermal parameter | *var* | |
| S28 | Unknown thermal parameter | *var* | |
| S29 | Unknown thermal parameter | *var* | |
| S3 | Unknown thermal parameter | *var* | |
| S30 | Unknown thermal parameter | *var* | |
| S38 | Unknown thermal parameter | *var* | |
| S39 | Unknown thermal parameter | *var* | |
| S4 | Unknown thermal parameter | *var* | |
| S41 | Unknown thermal parameter | *var* | |
| S42 | Unknown thermal parameter | *var* | |
| S43 | Unknown thermal parameter | *var* | |
| S8 | Unknown thermal parameter | *var* | |
| S9 | Unknown thermal parameter | *var* | |
| Sd | Stator tooth smallest area | *var* | |
| SD | Allowed damper bar current density | *const* | |
| Sf | Current density field winding | *var* | |
| Sfi | Initial current density field winding | *var* | |
| Sfm | Initial current density field winding | *var* | 1e6 |
| Sfmm | Current density field winding | *var* | 1e-6 |
| sigf | Intermediate reactance variable | *var* | |
| sigmacu75 | Conductance copper at 70 $^\circ C$ | *var* | |
| Sl12 | Unknown thermal parameter | *var* | |
| Sl17 | Unknown thermal parameter | *var* | |
| Sl29 | Unknown thermal parameter | *var* | |
| Sl41 | Unknown thermal parameter | *var* | |
| Sl42 | Unknown thermal parameter | *var* | |
| Sn | Rated apparent power | *var* | |
| SP | Stator parameters | *vec* | Output |
| Ss | Current density stator | *var* | |
| Ssmm | Current density stator | *var* | 1e-6 |
| Ss_ | Current density stator | *var* | Input |
| Su | Total armature slot area | *var* | |

Table 44: calcGenProg variables T

| Variable | Description | Type | Comment |
|---|---|---|---|
| Ta | Unknown thermal parameter | *var* | |
| Tam | Unknown thermal parameter | *var* | |
| taumd | Middle pole core pitch | *var* | Metre |
| taumn | Lower pole core pitch | *var* | Metre |
| taumx | Upper pole core pitch | *var* | Metre |
| taup | Pole pitch | *var* | Metre |
| taupr | Pole shoe arc length | *var* | Metre |
| taups | Pole pitch | *var* | Number of slots |
| taupt | Pole shoe pitch | *var* | Metre |
| taur | Damper bar pitch | *var* | Relative |
| tauu | Slot pitch | *var* | Metre |
| tauukj | Cooling duct width | *var* | |
| tauyr | Mean arc length per pole rotor yoke | *var* | |
| tauys | Mean stator yoke flux travel length | *var* | |
| TC | Thermal calculations | *var* | Output |
| Tdt | Transient time-constant | *var* | |
| Tdt0 | Transient time-constant | *var* | Per-unit |
| Tdtt | Sub-transient d-axis time-constant | *var* | |
| Tdtt0 | Sub-transient d-axis time-constant | *var* | Per-unit |
| Temp | Intermediate temperature matrix | *mat* | |
| Temp2 | Temperature matrix | *mat* | |
| Tetamn | Total required magnetization *var* | | |
| Tetasigma | Intermediate leakage magnetization | *var* | Ampere-turn |
| thc | Thermal conductivity insulation | *const* | |
| THF | Armature winding<br>Telephonic Harmonic Factor | *var* | |
| tnr | Number of turns per coil | *var* | |
| tnr_ | Number of turns per coil | *var* | Input |
| Tp | Intermediate thermal vector | *vec* | |
| Tqtt | Sub-transient q-axis time-constant | *var* | |
| Tqtt0 | Sub-transient q-axis time-constant | *var* | Per-unit |

Table 45: calcGenProg variables U

| Variable | Description | Type | Comment |
|---|---|---|---|
| Umd | Slot magnetic voltage drop | *var* | |
| Umdelta | Air-gap magnetic voltage drop | *var* | |
| Umdr | Pole core magnetic voltage drop | *var* | |
| Umtot | Total magnetic voltage drop | *var* | |
| Umyr | Rotor yoke magnetic voltage drop | *var* | |
| Umys | Statro yoke magnetic voltage drop | *var* | |
| Un | Nominal terminal voltage *var* | | |
| Unpu | Nomnial terminal voltage | *var* | Per-unit |
| Un_ | Nominal terminal voltage | *var* | Input |

Table 47: calcGenProg variables W

| Variable | Description | Type | Comment |
|---|---|---|---|
| w | Angular velocity $\omega$ | *var* | |
| Wew | Mean pole pitch | *var* | |
| Wewm | Pole pitch | *var* | Metre |
| wm | Mechanical angular velocity | var | |
| Ww | Coil span | *var* | |

Table 46: calcGenProg variables V

| Variable | Description | Type | Comment |
|---|---|---|---|
| v | Winding factor index variable | *var* | |
| V | Rotor peripheral speed | *var* | kilo-feet / min |
| Vf | Excitation circuit voltage | *var* | |
| vi | Cooling air-speed | *var* | |
| VId | Angle d-axis current | *var* | |
| vim | Cooling air-speed | *var* | |
| vmid | Mean cooling duct air-speed | *var* | |
| Vmx | Maximum generator voltage | *var* | Unused |
| Vnmx | Negative sequence voltage | *var* | Percent |
| vpl | Intermediate thermal parameter | *var* | |
| Vr | Maximum peripheral velocity | *var* | |
| vy | Cooling air-speed | *var* | |

Table 48: calcGenProg variables X

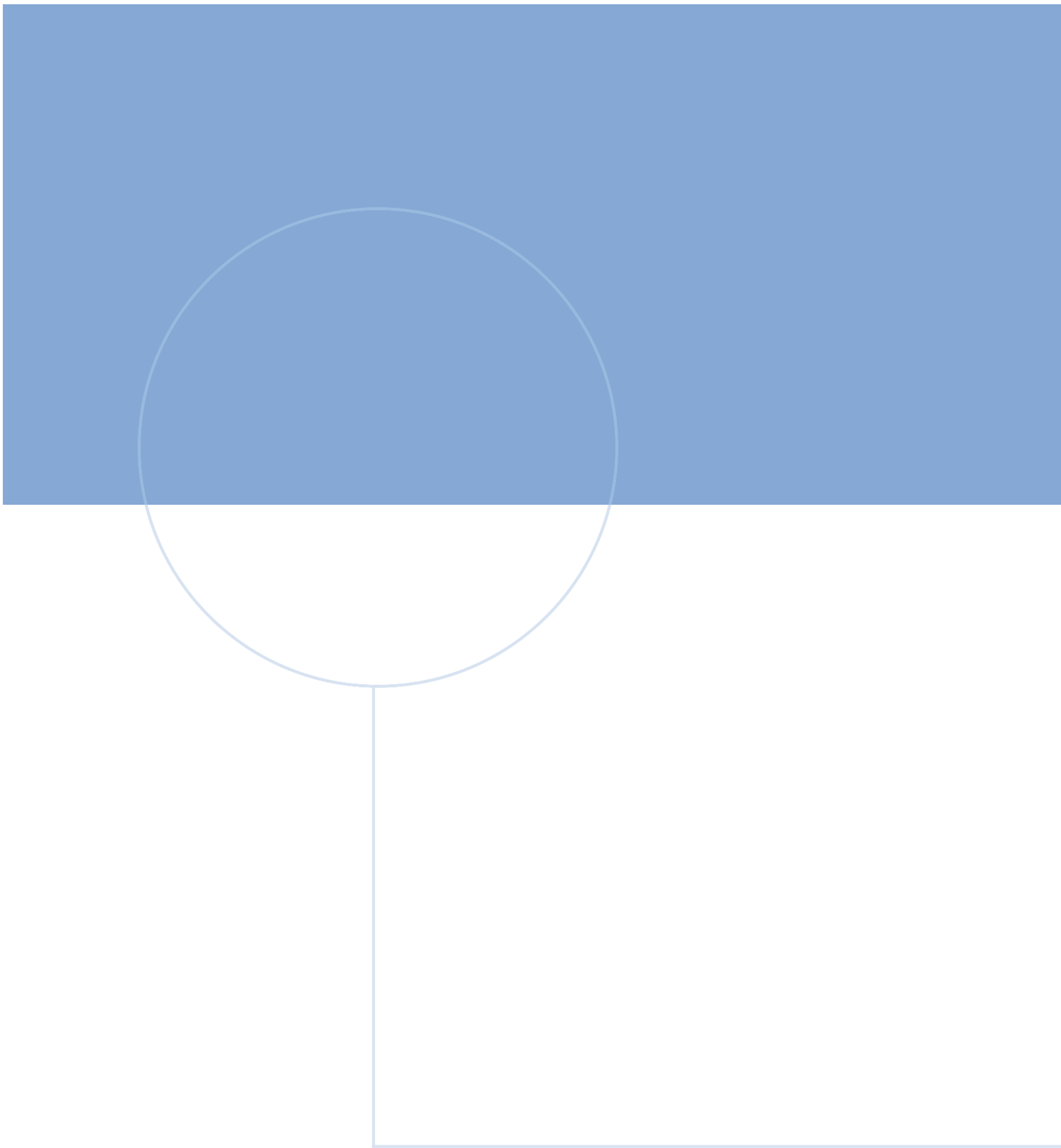
| Variable | Description | Type | Comment |
|---|---|---|---|
| xadw | Intermediate relative armature reaction reactance | *var* | |
| xd | Maximum synchronous reactance | *var* | Input |
| xd1 | Maximum transient reactance | *var* | Input |
| xd2 | Maximum sub-transient reactance | *var* | Input |
| Xdpu | Synchronous reactance d-axis | *var* | |
| Xdt | Transient reactance d-axis | *var* | |
| Xdtt | Sub-transient reactance d-axis | *var* | |
| xdw | Initial relative synchronous reactance | *var* | |
| Xf | Field winding relative leakage reactance | *var* | |
| xlw | Initial relative leakage reactance | *var* | |
| Xma | Armature reaction reactance | *var* | |
| Xmd | Armature reaction reactance d-axis | *var* | |
| Xmdpu | Armature reaction reactance d-axis | *var* | Per-unit |
| Xmq | Armature reaction reactance q-axis | *var* | |
| Xmqpu | Armature reaction reactance d-axis | *var* | Per-unit |
| Xqpu | Synchronous reactance q-axis | *var* | Per-unit |
| Xqtt | Sub-transient reactance q-axis | *var* | |
| Xsigma | Leakage reactance | *var* | Per-unit |

Table 49: calcGenProg variables Y

| Variable | Description | Type | Comment |
|---|---|---|---|
| y | Coil span | *var* | |
| Y | Unknown constant | *const* | |
| yQ | Relative pole pitch | *var* | |
| y_ | Coil span | *var* | Input |

Table 50: calcGenProg variables Z

| Variable | Description | Type | Comment |
|---|---|---|---|
| zeta | Intermediate resistance variable | *var* | |
| zetad | Intermediate d-axis time-constant factor | *var* | |
| zetaq | Intermediate q-axis time-constant factor | *var* | |
| zt | Number of vertical strands per slot | *var*z | |



NTNU
Norwegian University of
Science and Technology