Institute of Electrical Engineering and Information Technology
Paderborn University
Department of Power Electronics and Electrical Drives
Prof. Dr.-Ing. Joachim Böcker

LEA Project Group

Implementing time domain solver and parallel computing to the FEM Magnetics Toolbox

by

Jonas Hölscher, Othman Abujazar

Supervisor: Nikolas Förster, Till Piepenbrock

Filing Date: November 5, 2023

Abstract

In this project the Finite Element Method Magnetics Toolbox (FEMMT) is enhanced with multiple new features. First, a high performance computing (hpc) function is implemented to enable parallel simulations for FEMMT. This allows for a great increase of performance when using FEMMT for optimization purposes. Next, the simulation time of a single simulation is decreased by implementing multiple techniques to coarsen the mesh. Finally, a time-domain simulation of electromagnetic components, including inductors and transformers, is implemented. This significant advancement allows for a more comprehensive analysis of transient phenomena and dynamic responses. All those features have a great impact on the usability and capability of the Finite Element Method Magnetics Toolbox. Now, not only single and multiple simulations are much faster, the time-domain simulation offers the potential for a deeper understanding of the component's performance under realistic operating conditions, thus facilitating improved design and enhanced predictive accuracy in electromagnetic applications.

Contents

1 Introduction **1**

2 Parallel simulations **3**
2.1 Theoretical background 3
2.1.1 Computer architecture 3
2.1.2 Measuring software speed up 6
2.1.3 Amdahl's law 6
2.2 Implementation in FEM Magnetics Toolbox 7
2.2.1 Python multiprocessing library 7
2.2.2 Implementing high performance computing (HPC) function 8
2.3 Comparison of parallel and serial simulations 9
2.3.1 Measuring single simulation 9
2.3.2 Results of parallel simulation 10
2.3.3 Custom functions for parallel simulations 12

3 Speed up of a single simulation **13**
3.1 Analysis of the runtime of different FEMMT components 13
3.2 Analyzing mesh triangle aspect ratio 14
3.2.1 Implementing the aspect ratio 16
3.3 Analysis of the mesh of rectangular conductors 20
3.3.1 Analysis of simulation precision and performance 20
3.4 Analyzing mesh in the winding window 22
3.4.1 Implementation in FEMMT 22
3.5 Comparison of different meshing strategies 26
3.6 Results of speeding up single simulations 27

4 Frequency and Time Domain Simulation **28**
4.1 Principles of Frequency-Domain and Time-Domain 28
4.2 Advancing to Time-Domain Simulation 29
4.2.1 A Strategic Exploration in FEMMT 29
4.2.2 A Comparative Study using Time-Domain and Frequency-Domain Simulations in Onelab 31

5 Advancements in FEMMT: Incorporating Time-Domain Simulations 34
5.1 A Time-Domain Structure in FEMMT 34
5.2 Constructing Objects for Time-Domain Simulation Analysis 35
5.2.1 Excitation 35
5.2.2 Resolution 36
5.2.3 Parameter Representation in Post Processing 37
5.2.4 Formulation 38
5.2.5 Post Operation 39

6 Time-Domain Simulation Results 40
6.1 Running Simulations with"geo.time_domain_simulation" 40
6.1.1 Parameter Breakdown 40
6.1.2 Using the Function 41
6.2 Visualization of Time-Domain Simulation Results 42
6.3 Validation of Time-Domain Simulation Results 44
6.4 Post Processing in Python 46

7 Conclusion 47
7.1 Future Work 47

References 49

Appendix 50
A.1 Custom hpc for parallel FEMMT execution 50
A.2 Flame graph of basic inductor 50
A.3 Rolling Average Execution in FEMMT 51
A.4 Functions needed for Calculating the average and RMS Values in FEMMT 51

Listings

2.1 run_hpc function signature 9

5.1 Excitation in frequency-domain 35
5.2 Excitation in time-domain 36
5.3 Resolution in time-domain 37
5.4 Magnetic field in frequency-domain 38
5.5 Magnetic field in time-domain 38
5.6 Galerkin equation in frequency-domain 38
5.7 Galerkin equation in time-domain 39
5.8 structure of a file in frequency-domain 39
5.9 structure of a file in time-domain 39

6.1 Time-domain simulation's function 40
6.2 Post Processing in Python 46

A.1 run_hpc function example 50
A.2 Rolling average function 51
A.3 Integral function 51
A.4 Integral function for squared data values 52
A.5 Average function 52
A.6 RMS function 52

List of Figures

2.1 Threads and processes [2] 5
2.2 Parallel simulation study on multiple computers with different process counts 12

3.1 FEMMT model of inductor with and without insulation 15
3.2 Triangle mesh between core and insulation 16
3.3 Triangle schematic between core and insulation 17
3.4 Different aspect ratios 18
3.5 Mesh of model with and without insulation 19
3.6 Mesh of multiple conductor types 21
3.7 Mesh of rectangular conductor with coarse center 21
3.8 Study over multiple center sizes with different mesh sizes 22
3.9 Mesh of basic inductor 23
3.10 Winding window mesh study for different numbers of conductors 24
3.11 Mesh of basic inductor 25
3.12 Study over multiple meshing techniques and different mesh sizes 27

4.1 Time domain vs. frequency domain 29
4.2 Losses in an Inductor Over a Single Cycle 32
4.3 Losses in an Inductor Over a multiple Cycle with 100 time steps 32
4.4 Losses in an Inductor Over a multiple Cycle with 1000 time steps 33

6.1 System response at step 1. 42
6.2 System response at step 25. 43
6.3 System response at step 50. 43
6.4 Inductor winding losses 44
6.5 Inductor 1st turn losses 45
6.6 Inductor 2nd turn losses 45

A.1 Parallel simulation study on multiple computers with different process counts 51

List of Tables

2.1 Simulation times from 30 single core simulations 10

3.1 Results from the flame graph 14

4.1 Comparison of average losses results with different parameters 33
4.2 Comparison of average losses results with different parameters 33

1 Introduction

The quest for optimizing electromagnetic components for various applications in power electronics, automotive industry, and consumer electronics necessitates sophisticated simulation tools that can predict the performance of such components under real-world operating conditions.

The Finite Element Method Magnetics Toolbox (FEMMT) ([1]), developed by the Department of Power Electronics and Electrical Drives (LEA) at Paderborn University, is a versatile tool that has traditionally excelled in frequency-domain simulations of electromagnetic devices. Recognizing the growing demand for time-resolved analyses, the LEA has embarked on a project to incorporate time-domain simulation capabilities into FEMMT, thereby significantly expanding its application scope. In this project, the FEM Magnetics Toolbox is expanded with multiple features:

Since one of the key features of FEMMT is that the models of the inductors and transformers are generated automatically by using a list of parameters. One of the key applications of FEMMT is running optimization problems to find magnetic components which are optimized for specific purposes and values. Therefore, inductors and transformers can be found which are optimal in weight and cost and still comply with all other necessary conditions. While trying to optimize specific magnetic components lots of simulations have to be done. One part of this project is to speed up the simulation time of single and multiple simulations. This is done first by implementing a high performance function in FEMMT which enables to run multiple simulations in parallel. And second by implemented mesh coarsing algorithms to reduce the runtime for single simulations as well.

The other part of the project is about implementing a time-domain simulation. Time-domain simulation has become increasingly important in the design and analysis of these components due to its ability to capture transient behaviors and dynamic responses that are often missed by frequency-domain analyses. It offers a detailed view of how inductors, transformers, and other electromagnetic components react over time to changes in current, voltage, and other stimuli. This is critical for understanding non-linear behavior, resonances, transient effects, and the impact of control strategies in power electronics.

To fulfill the project's objective, an initial task involved conducting a comparative analysis between frequency-domain results and averaged time-domain outcomes for sinusoidal inputs using Onelab examples. This step was crucial for validating the accuracy and reliability of the time-domain simulation within FEMMT. Through this comparative study, we aimed to ensure that the time-domain implementation produced results that were consistent with established frequency-domain simulations, thereby confirming the validity of the new feature.

Overall, the extension of FEMMT with the capability of time-domain simulation significantly broadens the scope of its applicability, allowing researchers and engineers to model and predict the behavior of these devices with greater accuracy when exposed to non-steady state operating conditions. Additionaly the simulations can run much faster by executing multiple simulations at once.

In the first two chapters the implementation of the high performance computing function and the speedup of the single simulation by using different meshing techniques is explained. In the last three chapters the time-domain simulation is discussed, and the implementation is explained.

2 Parallel simulations

Right now FEMMT simulations are having execution times from about 4-10 seconds depending on the model type and complexity. If only a single simulation is done this execution time is sufficient. But for optimization problems, where hundreds or thousands of models are simulated, this can scale up to very long execution times very fast.

The task of simulating a model is very straightforward: At first the model is set up in FEMMT and files which are necessary for the solver are created. Then the mesh is created, and finally the simulation is done. This set of tasks has to be executed in this exact order one after another for every model, and it is obvious that simulating multiple models is independent of each other. Because of this it is possible to run the simulations in parallel on multiple cores.

Modern servers and computer clusters have hundreds of cores which can execute tasks in parallel. This can be used to speed up an optimization problem in FEMMT by running simulations in parallel. Even personal computers nowadays have 4, 8 or 16 cores which makes a parallel simulation useful for everyone.

In this part of the project group the FEMMT software will be expanded by a parallel simulation which shall work on every computer with multiple cores and be easy acessible for every user of FEMMT.

2.1 Theoretical background

In order to understand how parallel execution is done, how it is implemented, what terms are used and how a parallel speed up can be described theoretically, some background information is given here.

2.1.1 Computer architecture

When working with parallel execution in computers some important terms have to be explained properly. The described concepts are simplified in order to understand what is going on in this project, for detailed information please have a look at [2].

2.1.1.1 Process

A process can be thought of an instance of a program running on a computer. In general, it always contains two essential elements: **program code** and a **set of data**. The program code can be stored somewhere in the computers memory and can also be shared by multiple processes [2]. The data contains more information about the process itself, e.g.:

- Process identifier
- Priority of the process
- Pointer to the code in memory
- State of the current process. In a typical Five-State-Model those states could be: New, Ready, Running, Blocked, Exit

This data is usually stored in a so-called **process control block** somewhere in memory.

In order to start a process it has to be forked from an already existing process. When forking, the operating system (OS) copies the process control block of the already running process and changes the identifier. The new process is now a child to the already existing parent process. In a next step it is possible to change the pointer to the execution code.

When a process is created it is usually in the 'New' state. When it can be executed (all necessary requirements are met) it will be set to the 'Ready' state and now waits for the operating system to be assigned to a core to be executed. It changes its state to 'Running' when a core is assigned. A process is 'Blocked' when it was previously running, but the scheduler of the operating system decided that right now a different process shall be executed. When a process finished its task it sets to 'Exit' state.

2.1.1.2 Threads

Next to the already explained processes in most modern operating systems there are constructs known as threads. Threads resemble the executing part of a process. In a multithreading operating system a single process can have multiple threads and therefore multiple executions. It is important to know that every thread in a process works with the same data, and they all share the same resources. This is shown in Fig 2.1. If one thread changes a value in data every other thread can see the changes when they want to access the data.

The big advantage of threads is that they take far less time to create in an existing process instead of creating new processes. It is also faster to switch between threads of the same process instead of switching between different processes.

It is also possible for a process with multiple threads to run each thread on a different core.

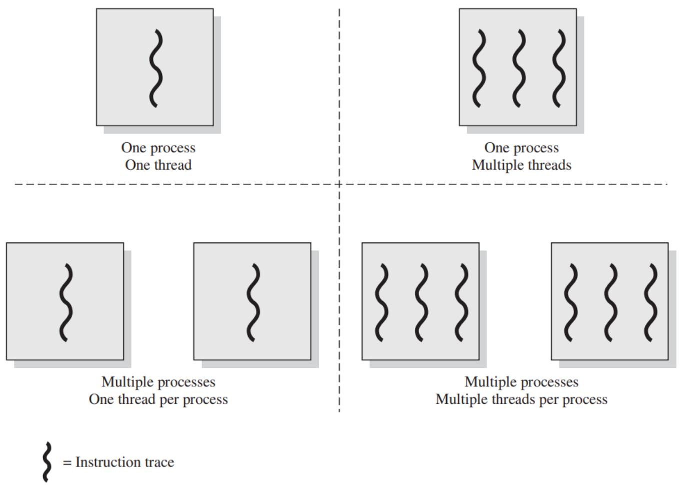


Fig. 2.1: Threads and processes [2]

2.1.1.3 Concurrency and parallelism

Concurrency means that multiple processes can be run independent of each other on one or more cores. For example when there are two processes which are independent of each other and the computer only has a single core it is possible that the processor executes process P1, and after some time P1 will be interrupted and process P2 is executed and so on. After some time both processes are done. In this case they ran concurrently but not in parallel. They can only run in parallel when there are at least two cores, and they are both executed on a different core.

If in a modern operating system tasks are run in parallel or concurrent is not easy to tell. Some processes can be created which are independent of each other and therefore could be run in parallel. Others are dependent on specific data or inputs to the system. But it is completely up to the operating system in which order and on which core a task is executed.

However, when the system user hinself creates processes, usually they are assigned a higher priority than other processes and executed rather quickly. But the user cannot be completely sure if this is the case.

In order for a problem to be parallized efficiently it is necessary that the different subtasks in which the problem is divided are as far as possible independent of each other. Processes are independent when they don't need to access the same data in memory and are not using the same computer ressources (e.g. I/O systems). In the case of this project the different FEMMT Simulations need to be parallelized. This can be done very efficiently because each simulation task needs different data (different model and different input parameters) and the simulation can be executed on each core seperately.

2.1.2 Measuring software speed up

In order to measure the speedup of a program it needs to be compared with the execution time of a reference program. For parallel executions the reference program typically is a program with the same results but in serial execution. So for example in the case of FEMMT a parallel simulation with 10 models on 5 cores would be compared against a serial simulation with the same 10 models on 1 core.

The speedup S_p can be calculated the following way ([3]):

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \tag{2.1}$$

Where T^* is the execution time of the serial program and T_p is the execution time of a parallel execution with p cores. The variable n corresponds to the size of the problem domain. The exact meaning of n differs depending on the problem. In the case of sorting algorithms this is proportional to the amount of items in the list, that will be sorted. In the case of FEMMT this is proportional to the number of simulations.

The ideal speedup is $S_p(n) = p$. According to [3] the speedup could possibly be higher than p but in most cases the real speedup is lower than p. This is due to the fact that some amount of computing power will be used to organize the parallel execution (e.g. setting up in memory), setting up multiple processes on multiple cores, synchonization of different cores, interchanging information between the cores and more.

2.1.3 Amdahl's law

Because most parallel executions don't reach the speedup of p, amdahl's law tries to take the amount of the execution which cannot be parallelized into account ([3]): In a parallel implementation a fraction f of the implementation will be executed in serial and therefore the upper bound for the speedup can be calculated by

$$S_p(n) = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f} \tag{2.2}$$

This takes into account that the part which is executed parallel does execute ideally. It gets clear that the precision of the calculated speedup depends on f and how well it is estimated. This is completely up to the problem and even depends on the implementation. Wether Amdahl's law is suitable for the parallel simulations which are implemented in this chapter will be examined later. But it is clear to see that because the different simulations in FEMMT all work independent of each other and don't need any communication among them the possible speedup can be very high.

2.2 Implementation in FEM Magnetics Toolbox

In this section the implementation details of the parallel simulations in FEMMT are presented. The goal was to implement an HPC (high performance computing) function which is easily accessible for the user, works for every model possible and also provides enough flexibility to run different simulation types. Since the Python multiprocessing library is used for the implementation it will be explained first and also why this library was chosen.

2.2.1 Python multiprocessing library

In general there are a variety of ways on how to implement parallel execution. Those methods work on different software abstraction layers. For a short overview some examples will be mentioned, more information can be found in [3].

SIMD (single-instruction-multiple-data) describes a set of instruction set commands which are directly executed on the processor, this resembles the lowest level of software. As the name suggest with SIMD the processor can calculate the result of a set of data simultaneously instead of calculating the result for one data point.

OpenMP is a standardized programming interface which is, for example, implemented in C/C++ and implements the execution of loops and functions on different threads.

There are other programming interfaces which work on different processes (e.g. MPI).

All of those mentioned ways of developing parallel programs are too low level for the purpose of parallel FEMMT simulations. Those parallelization techniques must be applied in the mesher and solver directly. Since in FEMMT we are using gmsh and GetDP which are developed externally we have no direct access to their code. It also would go far beyond of what can be accomplished in one project group.

Therefore, a higher level Python library is needed. It has to take functions which call gmsh and GetDP as a whole and distribute it among multiple cores. Python provides two packages which could accomplish this:

- threading
- multiprocessing

With the Python **threading** library it is possible to execute code on different threads. But using this one tasks can only be executed concurrently but not in parallel. This is due to the so-called Python Global Interpreter Lock (GIL). The GIL is a mechanism which prevents that multiple threads can be in the control of the Python interpreter at the same time. This is a design-choice made by the developers of Python in order to prevent memory problems which can occur when working careless with parallel execution.

The Python **multiprocessing** library on the other hand creates multiple processes in which every process has its own Python interpreter, so the GIL is not blocking the execution. The

Python documentation refers to subprocesses which are created using the multiprocessing library, but those are just "normal" processes which have the origin Python process as a parent. Therefore, if the parent process is terminated all the child processes are terminated too.

2.2.1.1 Structure of FEMMT

In general the FEM Magnetics Toolbox is split in 3 parts: First part is the python code, completely written here at LEA, which creates the models based on the given parameters and creates the necessary files for the simulation. The second part is gmsh, used to create a mesh of the model and the thrid part is GetDP, which is used to calculate the electromagnetic field using the mesh and the parameters from both previous parts. Both gmsh and GetDP are third-party software components which are implemented in C++. Therefore, they are not affected by the Python GIL and can theoretically be executed in parallel using threads.

So based on the threading and multiprocessing library there are two possible ways to implement a parallel execution of FEMMT:

1. Split between Python and gmsh/GetDP: Execute Python in serial and prepare the necessary files. Then gmsh and GetDP are run parallel using threads.
2. Use FEMMT as a whole unit: For every simulation use a single process and execute every step in the same process.

Altough the first approach does use threads, which are way faster in creation and inter-process communication, the whole Python part of the code is not parallelized and the effort to organize the serial execution of Python and the parallel execution of gmsh and GetDP is much greater than in the second approach. Also, because the simulations can run independently of each other, there is no need for interprocess communication anyway. Therefore, we decided to implement the second approach in FEMMT.

The goal was to implement parallel simulations in FEMMT in an easy and accessible way. A so called hpc (high performance computing) function, which can be called from the users script to make the simulation run in parallel, shall be created. It should have as few parameters as necessary.

2.2.2 Implementing high performance computing (HPC) function

As discussed above the hpc function was implemented using the Python multiprocessing library, and it was parallelized using FEMMT as a whole. In order to run such a parallel simulation it was necessary to get a list of all models which shall be simulated. This has to be done by the user in advance and is not implemented in the hpc function because the models are created by the user anyway and can be completely arbitray.

FEMMT does already allow a unique working directory for every model and every file that is needed for a simulation and also the results of the simulations are then stored in the custom working directory. Therefore, the operating system can access different files in different folders which can be done fully in parallel. But in order for the GetDP solver to work it needs the solver configuration files (*.pro) which implement the desired simulation type. Therefore, the FEMMT solver files: **fields.pro**, **ind_axi_python_controlled.pro**, **solver.pro** and **values.pro** need to be copied to each and every working directory, so they are all independent of each other and can be accessed in parallel.

After the models and directories are prepared the **Pool** function from the multiprocessing library is used to implement the parallel simulation. This function takes a pointer to the function (now called parallel function), which shall be executed in parallel as well as a list of parameters, which are given to the parallel function.

In the parallel function the FEMMT calls to start the simulation are made. If the user wants to implement special behavior, he can also set a custom parallel function (more in section 2.3.3), but for normal cases a default function is implemented.

In this default parallel function the necessary function parameters are the simulation frequency and the simulation current. Using both parameters the FEMMT model will be created, and the simulation is started.

The function signature of the created run_hpc function is given in 2.1.

```python
def run_hpc(n_processes: int, models: List[MagneticComponent],
    simulation_parameters: List[Dict], working_directory: str,
    custom_hpc: Callable = None)
```

Listing 2.1: run_hpc function signature

2.3 Comparison of parallel and serial simulations

In order to quantify the speedup that is done by using the hpc function, multiple simulation studies shall be made. At first a normal serial single simulation is made to be able to compare it to the parallel results. After this the parallel simulations on multiple computers are done and the simulation results are compared.

2.3.1 Measuring single simulation

At first the inductor from the FEMMT basic_example.py is simulated using the default serial simulation. For a better understanding the measurement time is split into 7 independent parts:

1. **Setup time**: Time FEMMT takes to create the necessary objects and prepare the simulation.
2. **High level geo gen time**: Time FEMMT takes to create the model in gmsh.

Tab. 2.1: Simulation times from 30 single core simulations

Description	**Detail**	**Time (mean) / s**	**Variance / s**
Setup time		4.77e-3	2.962e-5
Create model time		1.962e-2	6.005e-7
	High level geo gen time	9.985e-5	2.891e-7
	Generate hybrid mesh time	1.952e-2	2.323
Simulation time		3.958	4.798e-4
	Generate e_m mesh time	1.0741	5.431e-5
	Prepare simulation time	7.788e-3	1.0083e-6
	Real simulation time	2.874	3.648e-4
	Logging time	2.346e-3	2.0494e-7
Execution time		3.983	4.929e-4

3. **Generate hybrid mesh time**: Time FEMMT (and gmsh) takes to build the model in gmsh and execute mesh optimizing routines.
4. **Generate eelectro magnetic mesh time**: Time gmsh takes to generate the electro magnetic specific mesh.
5. **Prepare simulation time**: Time FEMMT takes to prepare the necessary GetDP files.
6. **Real simulation time**: Time GetDP takes to solve the problem.
7. **Logging time**: Time FEMMT takes to collect the simulation results and store them in log files.

The simulation was done on an Intel i7-10700KF with 3.8 GHz and 16 GB RAM. The simulation was repeated 30 times and the averages were calculated. The results can be seen in table 2.1.

It is obvious that the solving in GetDP as well as the mesh generation take up most of the execution time. The code executed in FEMMT takes only a very small amount of time. Additionaly the variance of the exeuction time is with 0.4929ms very low and makes up only 0.01% of the whole execution time. Therefore, it can be assumed that the sum of n consecutive simulations is approximately equal to

$$T_n \approx n \cdot T_{single} = n \cdot 3.983s \tag{2.3}$$

2.3.2 Results of parallel simulation

Now the simulation will be done in parallel. A study is made by using different process counts and then the results are compared to the serial simulation and the speedup is calculated.

2.3.2.1 Simulation of basic example inductor

In order to compare the parallel simulation times with the serial simulation, at first the used FEMMT model will be the same as in section 2.3.1, which is the inductor from the basic example.

In the following, 30 simulations of the same model will be made. Using eq. (2.3) the whole execution time of 30 serial simulations can be approximated:

$$T_n \approx 30 \cdot 3.983s \approx 119.49s \tag{2.4}$$

The parallel simulation was done using 5 cores and the execution time was $28.801s$. This results to a speedup of

$$S_{5,real}(30) \approx 4,149 \tag{2.5}$$

This is close to the possible speedup predicted by amdahls law: Since the setup time for the parallel simulation was about 0.1197s (this includes setting up the necessary files and models) the resulting fraction of the serial part to the parallelized part is $f = \frac{0.1197}{28.801} = 0.00416$, this can be plugged in eq. (2.2) to get a theoretical speedup of:

$$S_{5,theoretical} = \frac{1}{f + \frac{1-f}{p}} \approx 4.918 \tag{2.6}$$

As expected the real speedup is lower than the theoretical speedup $S_{5,real} < S_{5,theoretical}$. Nevertheless, the real speedup is very good and close to the theoretical one.

Therefore, it can be stated that the goal of implementing a high performance computing function to make parallel simulations was successful for the basic inductor and the speedup is somewhat near an expected value.

2.3.2.2 Study for variation in process count

Since the previous study only used a fixed number of cores and the same model, now a broader study is done. The number of cores are varied as well as the model parameters. Since the execution time per core is averaged over the resulting execution time of models with different parameters a more realistic value is returned. The model parameters frequency and air gap position as well as air gap height are changed. Additionaly the simulations are done multiple times and the results are averaged too. The simulations results can be seen in fig. 2.2.

In this case a total of 3 iterations and 36 different models per iteration were simulated. The runtime was averaged over the different iterations. The plot shows an expected result in which the runtime is anti-proportional to the number of processes: $T_{execution}(p) \sim \frac{1}{p}$. For example at the computer with 8 cores, the runtime with 1 core is approximately

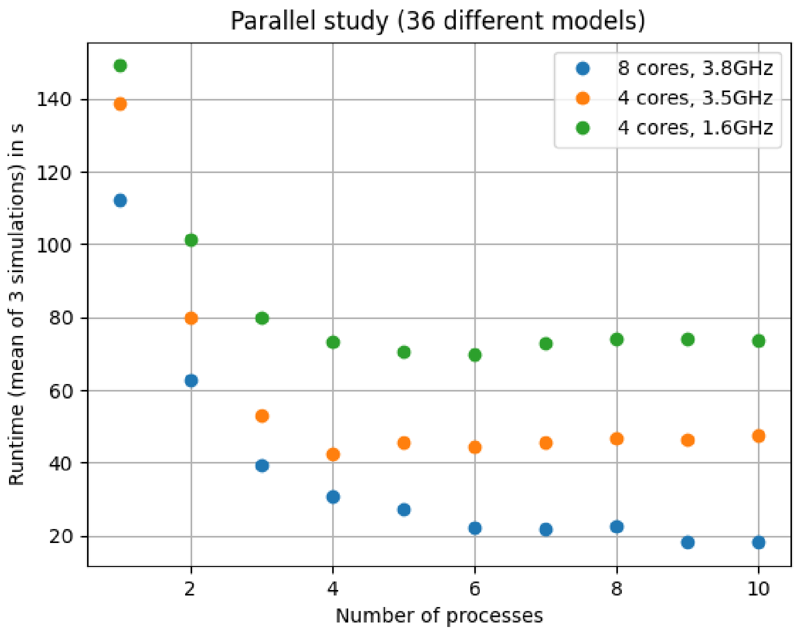


Fig. 2.2: Parallel simulation study on multiple computers with different process counts

$T_{execution}(1) \approx 112s$ and with 2 cores it is approximately $T_{execution}(2) \approx 62.6s$ which is close to half of the exeuction time with 1 core. This can also be observed when looking at 2 and 4 cores or 4 and 8 cores.

Additionaly it can be seen that the runtime does not decrease significantly with a process count higher than 6, although one computer has more than 6 cores. This can have multiple reasons which are most likely due to the operating system. It could be that it does not give the python environment more than 6 cores or the other cores are already occupied with background programs or programs that are critical for the system performance.

2.3.3 Custom functions for parallel simulations

Next to the already implemented default function, which executes the FEMMT functions create_model and single_simulation, a custom function can also be given. This can be used to implement more complex parallel computing tasks. This custom function only recieves one parameter which is a dictionary containing all the values necessary for the simulation. The function itself contains calls to the FEMMT functions which shall be called in parallel. The default function only allows for different frequencies and currents for each model, but a custom hpc function could also vary different parameters, an example is given in the appendix: A.1.

3 Speed up of a single simulation

After creating a hpc function which can run FEMMT simulations in parallel, now the single simulation itself shall be fastened. At first the current state of FEMMT will be analyzed by measuring the runtime of different FEMMT components. Using this analysis the components with high runtime and potential for improvement are pointed out. After that multiple strategies for runtime improvement are presented and implemented.

3.1 Analysis of the runtime of different FEMMT components

In order to analyze the current state of the FEMMT package, a runtime analysis is done. This measures the runtime of every part in the FEMMT package during the simulation of a model and presents the results in a so-called flame graph. This analysis is done using a profiler, in this case the austin profiler was chosen ([4]). Since the flame graph does contain more information than needed, the most important information can be seen in table 3.1. For completion, an image of the flame graph can be seen in the appendix in fig. A.1.

In the table 3.1 only the functions with the highest runtime are listed, and it is also noted whether they are run in FEMMT, gmsh or GetDP.

It gets clear that the FEMMT part has very little impact on the execution time of the wohle FEMMT package with around 0.59%. Therefore, it makes most sense to focus on optimizing the mesh generation and the simulation.

In the next sections multiple approaches for optimizing the mesh in FEMMT are examined.

Tab. 3.1: Results from the flame graph

Layer 1	**Layer 2**	**Layer 3**	**Runtime in μs**
simulate inductor			4627405 (100%)
	create model		24043 (0.52%)
		init gmsh (gmsh)	8263 (0.18%)
		generate hybrid mesh (FEMMT)	15780 (0.34%)
	simulation		4603362 (99.5%)
		generate e_m mesh (gmsh)	1280019 (27.66%)
		update core materials (FEMMT)	11692 (0.25%)
		simulate (GetDP)	3295129 (71.2%)

3.2 Analyzing mesh triangle aspect ratio

For the thermal simulation insulations were introduced to the model. Those insulations should, in theory, not impact the electro magnetic simulation but are important in order to determine the temperatures accordingly. Since a hybrid mesh is used the insulations are also present for the electro magnetic simulation. An image of the model with and without insulation can be seen in fig. 3.1a and fig. 3.1b

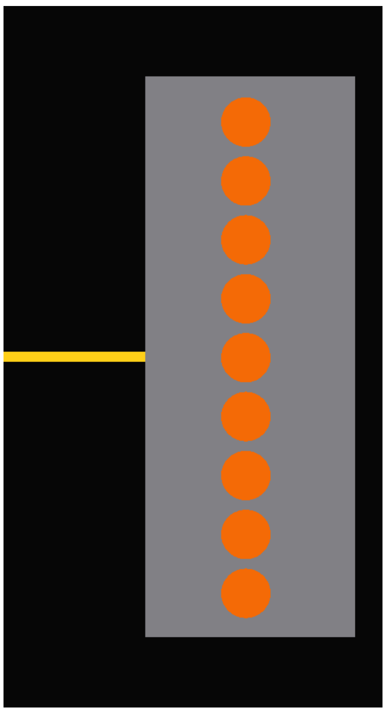
In order for a simpler implementation, the insulations were just added inside the winding window with a short distance between the core called an insulation delta. This insulation distance is set to a very small value, so it is as close as possible to the core, this can be seen in fig. 3.1c, which is a zoomed in image of the red border in fig. 3.1b.

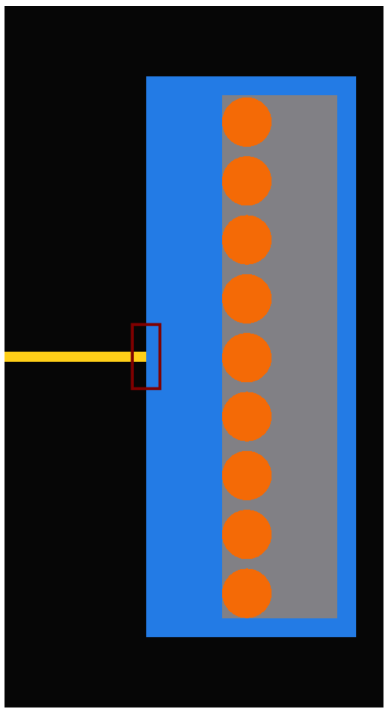
The problem is when the insulation delta is very low, the distance between the insulation and the core is very small and when the mesh tries to create triangles in this area those triangles are very long and stretched out. This is shown in fig. 3.2. Since the width of the triangle is longer than the height, the triangle has a large aspect ratio:

$$aspect\_ratio = \frac{width}{height} \tag{3.1}$$

According to [5] triangles with poor (e.g. high) aspect ratio yield a more poorly conditioned system matrix and therefore the simulation takes longer. As a rule of thumb the aspect ratio shouldn't be higher than 5 ([6]). Currently, the insulation delta is set to 0.00001 and the aspect ratio is approximately 15.1.

The goal is to implement a variable aspect ratio which can be changed by the user. The function has to make sure that the real aspect ratio (the aspect ratio of each triangle in the model) is not bigger than the given aspect ratio.


(a) Model without insulation


(b) Model with insulation

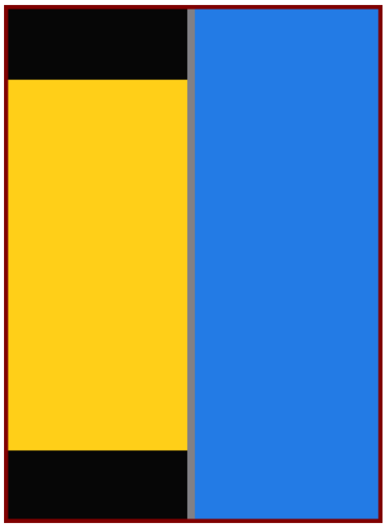

(c) Insulation distance visualized closely

Fig. 3.1: FEMMT model of inductor with and without insulation

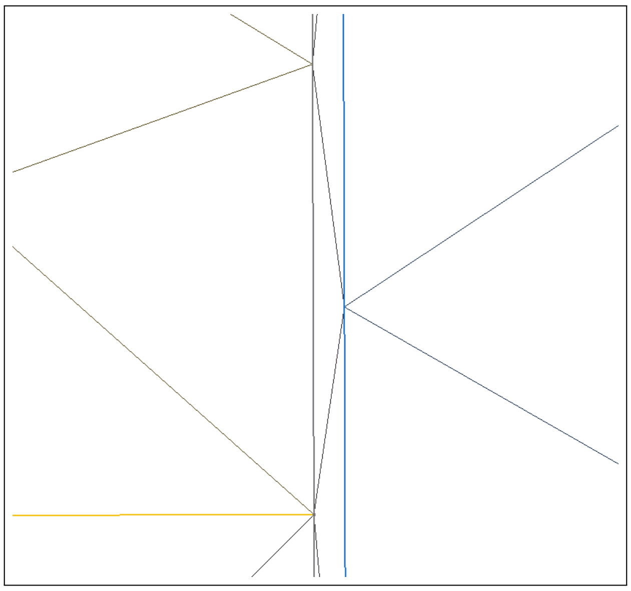


Fig. 3.2: Triangle mesh between core and insulation

3.2.1 Implementing the aspect ratio

The gmsh meshing algorithm works as follows: At first the lines between the points are discretized by a defined element length, this means that along the line between points A and B the algorithms tries to add more points in such way that the distance between them is roughly the given element length. After the lines are discretized, the created points are then used to discretize the surfaces. In the gmsh documentation ([7]) it is stated that the real distance between two points on a line is always smaller or equal the given element length.

In this case the element is the core and the element length is specified by FEMMT, and it is called c_window. The width and height of the triangle in the mesh then is determined by the c_window and the insulation_delta as shown in fig. 3.3.

Since the real height h of the generated triangle is always smaller than the given element length $c\_window \geq h$ the aspect ratio of the real triangle can be calculated as follows:

$$given\_aspect\_ratio = \frac{c\_window}{insulation\_delta} \geq \frac{h}{insulation\_delta} = real\_aspect\_ratio \tag{3.2}$$

Therefore, the real aspect ratio of the generated triangle is always smaller than the aspect ratio from the set parameters, which is exactly what is needed, because a smaller aspect ratio is always better.

If the user now wants to enter an aspect ratio, which shall not be exceeded, the corresponding insulation delta can be calculated like this:

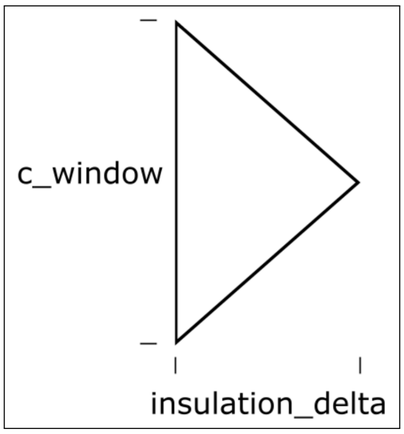


Fig. 3.3: Triangle schematic between core and insulation

$$insulation\_delta = \frac{c\_window}{given\_aspect\_ratio} \tag{3.3}$$

This was now tested in a study where the precision and performance was logged. The study can be seen in fig. 3.4. An aspect ratio of 0 means the simulation was done without insulations.

The plot shows that the simulation without insulations is the fastest, but the precision does vary a lot between models without insulations and models with insulations and different aspect ratios. This difference comes most likely from the mesh in a model with and without insulations. An example can be seen in fig. 3.5. The difference in the mesh comes from the way the gmsh meshing works and how the insulations are embedded in the model. In a simulation without insulations the corner points of the winding window set the mesh size, and then they are scaled down until the points from the round conductor. In a simulation with insulations howewer, the insulations make up the outer parts of the winding window and since they are a rectangle with the same mesh size on each point, the mesh inside the insulation is constant and not a gradient.

Overall it gets clear that lowering the aspect ratio does not have the desired effect: Although the simulation time is much lower the precision of the simulation according to the self inductance and the total losses is not so good. From the problem definition the solution without insulations should be the correct solution. This is because losses in the insulations, like dielectric losses, are not calculated anyway. Therefore, the difference in total losses and self inductance has to come from the mesh.

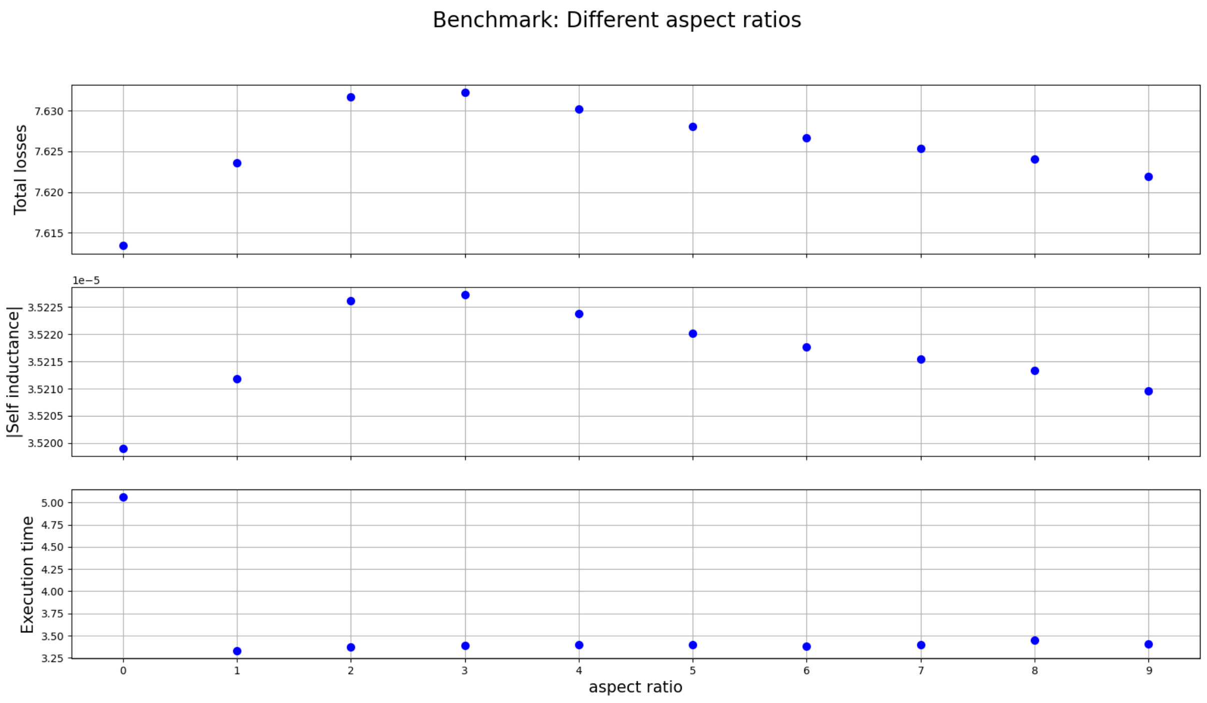
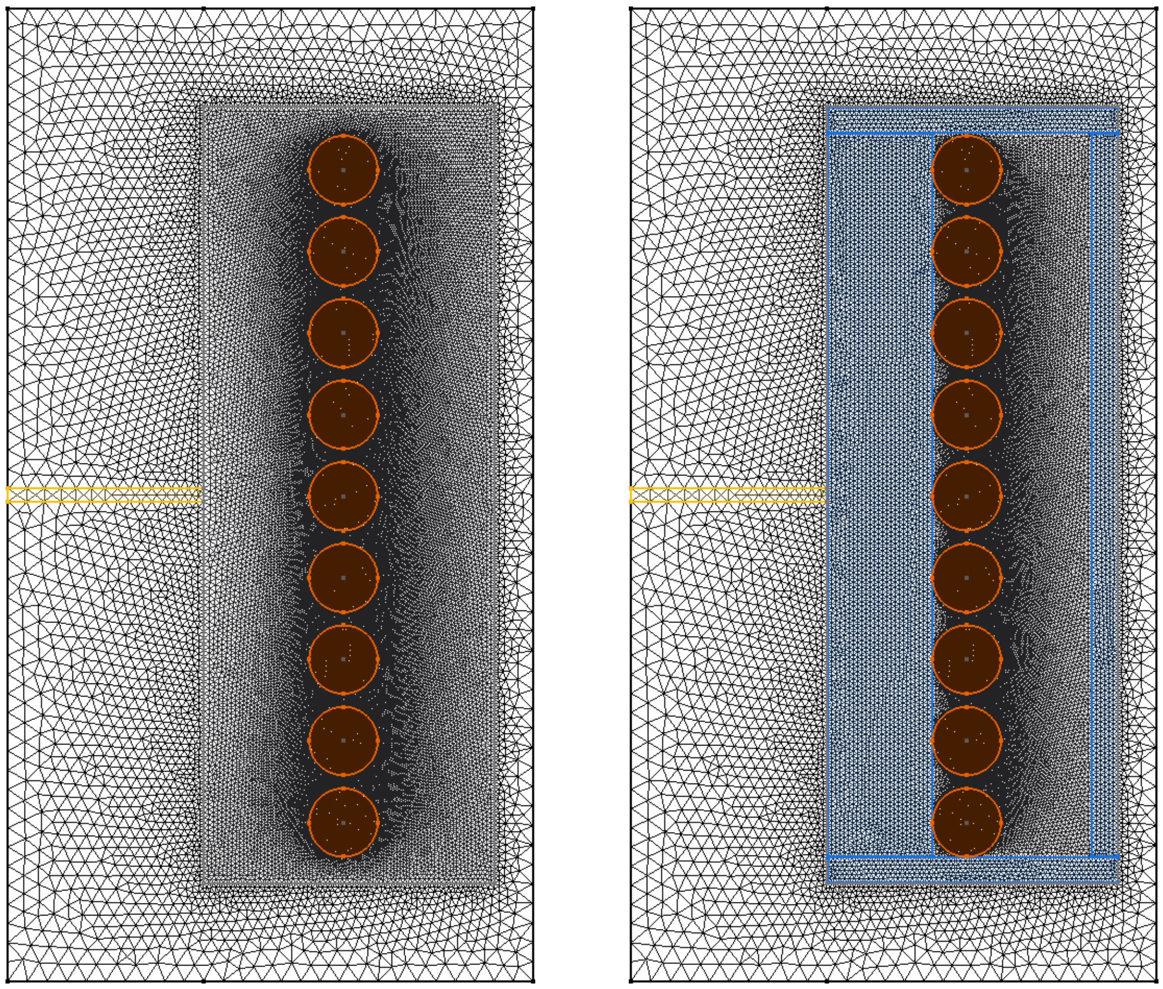


Fig. 3.4: Different aspect ratios



(a) Model without insulation **(b)** Model with insulation

Fig. 3.5: Mesh of model with and without insulation

3.3 Analysis of the mesh of rectangular conductors

For round conductors skin based meshing is already implemented [8]. This means that the mesh inside the round conductor can be fine or coarse depending on the frequency. Therefore, the mesh size of the conductor is lowered when the frequency is higher, because of the smaller skin depth.

A similar consideration can be made for rectangular conductors. Here the idea is that for common frequencies for inductors of this size the skin depth is already very small, therefore the current does only flow in the outer parts of the rectangular conductor. Consequently, the mesh could be set more coarse in the center of the conductor without losing too much simulation accuracy.

In fig. 3.6 a comparison between the current frequency dependent mesh of a round conductor and the mesh of a rectangular conductor can be seen.

Since in gmsh the points determine the mesh size, the idea is to place one or more points in the middle of the rectangular conductor and give them a higher mesh size. The implementation for a single point with a 4 times higher mesh size can be seen in fig. 3.7a.

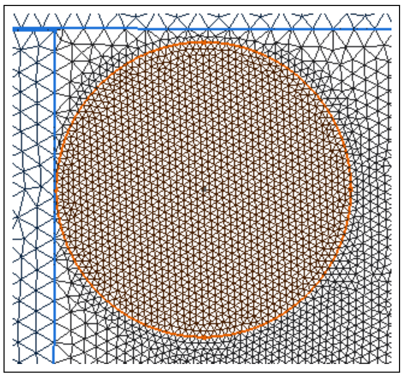
It can be seen that one point is not enough for the rectangular conductor, since it can be varying in width and height. Therefore, multiple points are added for the width dimension and height dimension. An example for multiple points un the width direction can be seen in fig. 3.7b.

3.3.1 Analysis of simulation precision and performance

Since now the mesh in the rectangular conductor is coarser, it is necessary to check if the simulation does return the same results and if the execution time has been lowered.

In fig. 3.8 a study is made with multiple center sizes and different mesh sizes. The plot contains the execution times to measure performance and the total winding losses over the skin depth of the whole inductor to check the precision. The values for center factor = 1 are the same as without. The black dottet horizontal line is the comparison value for a similar FEMM (an open-source tool for 2d magnetic simulation) simulation. When the mesh size/skin depth is lower than 1 that means that there are more than 1 discretization points for the length of the skin depth.

The plot shows that especially for low mesh sizes the execution time with higher center factors is much lower and the total winding loss is the same. Only for lower mesh sizes there are differences between the winding losses for various center factors, but the differences are still quite low. For very coarse meshes the effect of higher center factors on performance is no longer significant because the simulation is very fast anyway. Therefore, the center factor has a good impact on performance when run with a low mesh size and the simulation results are still pretty accurate.


(a) Mesh of round conductor

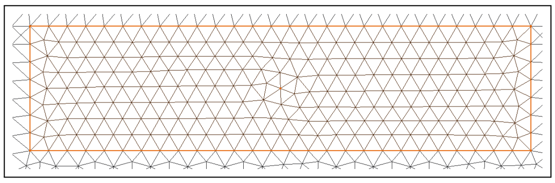
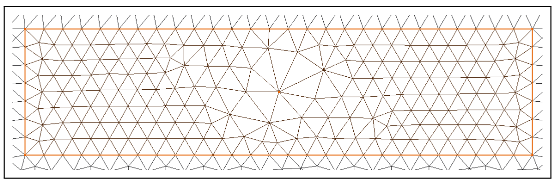

(b) Mesh of rectangular conductor

Fig. 3.6: Mesh of multiple conductor types


(a) Mesh with one center point

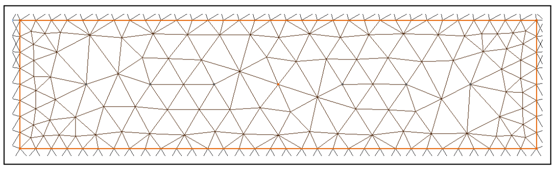

(b) Mesh with multiple center points

Fig. 3.7: Mesh of rectangular conductor with coarse center

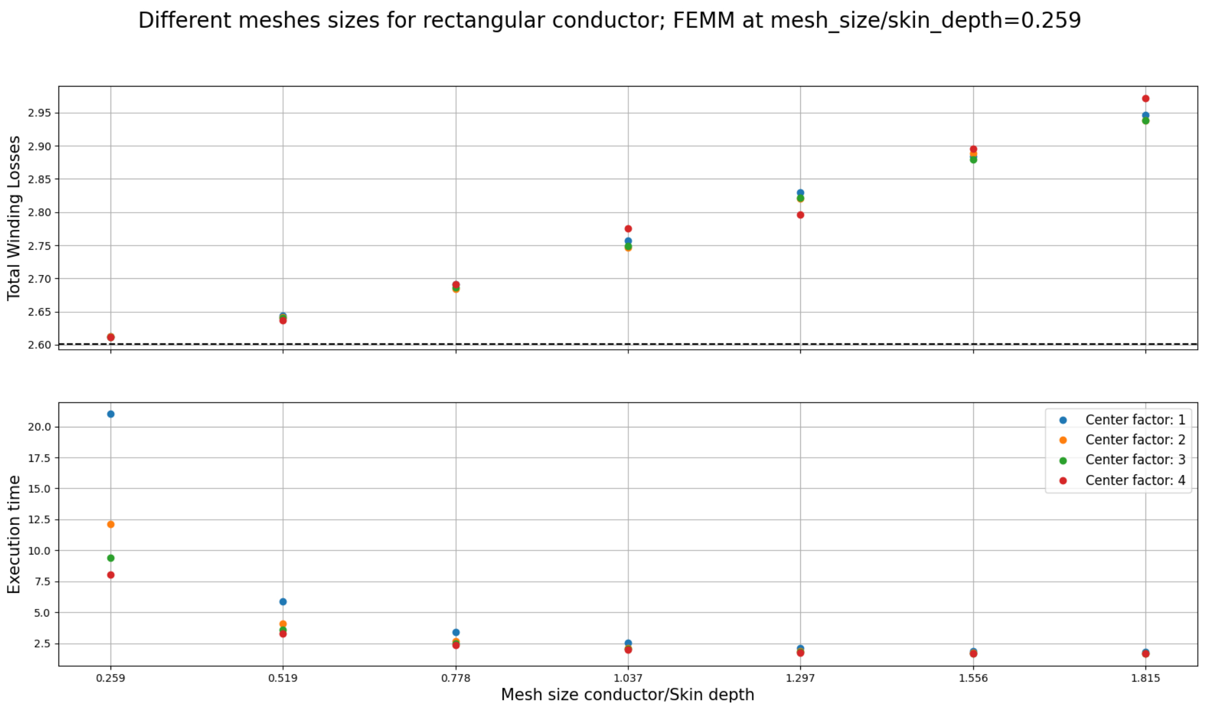


Fig. 3.8: Study over multiple center sizes with different mesh sizes

3.4 Analyzing mesh in the winding window

When looking at the mesh of a model, displayed in fig. 3.9a, the mesh is quite fine inside the winding window. This is because the mesh size is set by points and the corner points of the winding window are set to a low value. But since in this case the conductors are only filling a small portion of the whole winding window, most of it is empty and very finely discretized although, the expected change of the magnetic field is not so big. Therefore, the idea is to add points inside the winding window, which have a higher mesh size similar to section 3.3 with rectangular conductors.

3.4.1 Implementation in FEMMT

An algorithm is needed to place points accordingly inside the winding window. But it is necessary that the points are sufficiently far away from the conductors because next to them the expected field change is much bigger than far away.

The algorithm works as follows: At first a rectangular grid of points with equal spacing is placed over the winding window. Then for each point it will be checked if it is very close to or even inside a round or rectangular conductor. If that is the case the point will be removed from the grid. It also needs to be checked if the point is on a stray path, if an integrated transformer is implemented, in this case it also needs to be removed from the grid. In the end all remaining points are embedded into the mesh and given a specific mesh size, which is higher than the mesh size from the winding window itself.

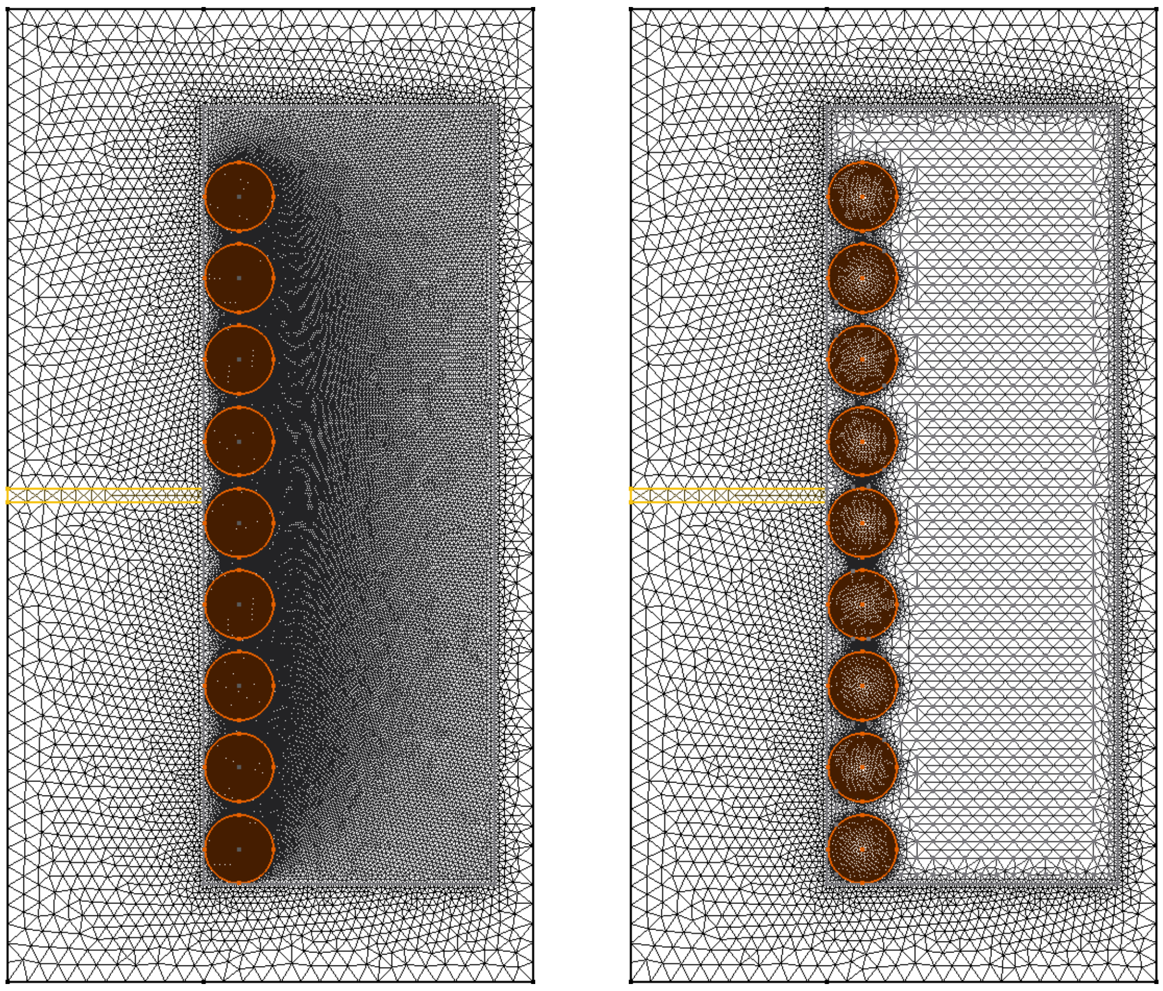

(a) Mesh with normal winding window **(b)** Mesh with coarser winding window

Fig. 3.9: Mesh of basic inductor

If the core has two or more winding windows, this algorithm can be applied on each of them. In fig. 3.9b the resulting mesh is shown. It can be seen that the points do raise the mesh size of the winding window significantly.

In fig. 3.10 a study is shown in which the execution time, total losses and self inductance of simulations with finer mesh in the winding window (namely the winding window rasterization or wwr) is compared with the results of the original model. This is plotted over different numbers of conductors.

The plot shows that even with a higher number of conductors, where the winding window is filled more, the speedup stays almost the same. This is because the grid of points of the algorithm for the winding window rasterization is so small, it still can raise the mesh in areas between the conductors. This can be observed in fig. 3.11b. Also, the precision of the simulation with a coarser winding window is pretty good, hence the differences to the comparison simulation are small.

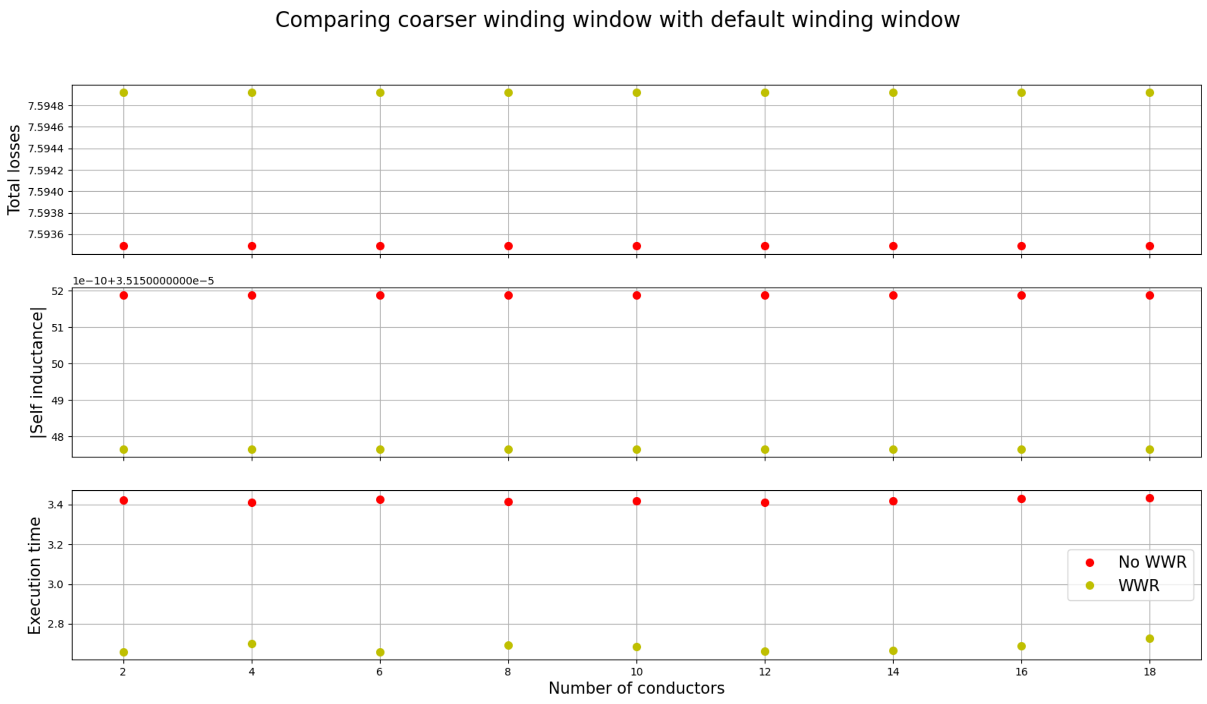
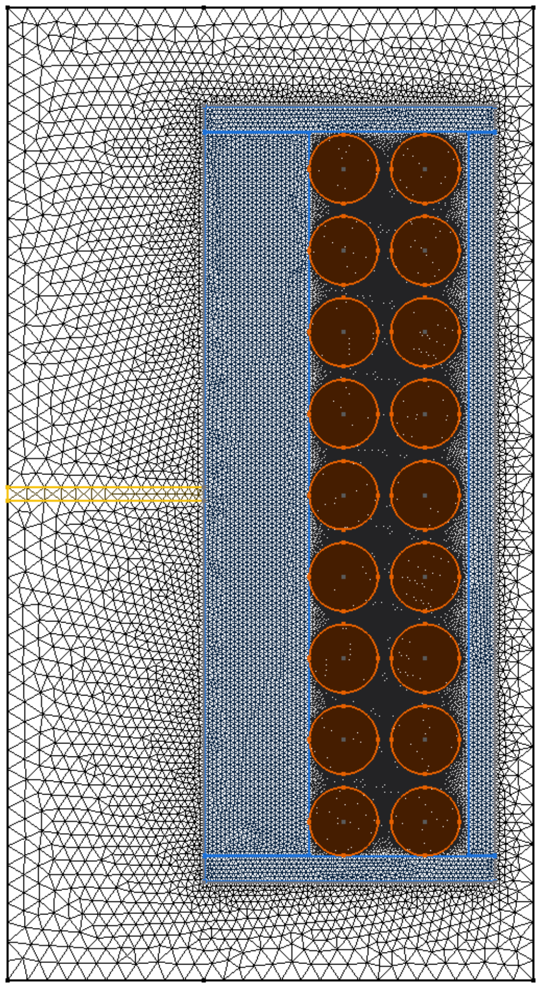


Fig. 3.10: Winding window mesh study for different numbers of conductors

Overall the winding window rasterization is a good way to reduce the runtime of the FEMMT single simulation and still keep a good accuracy.


(a) Mesh with normal winding window

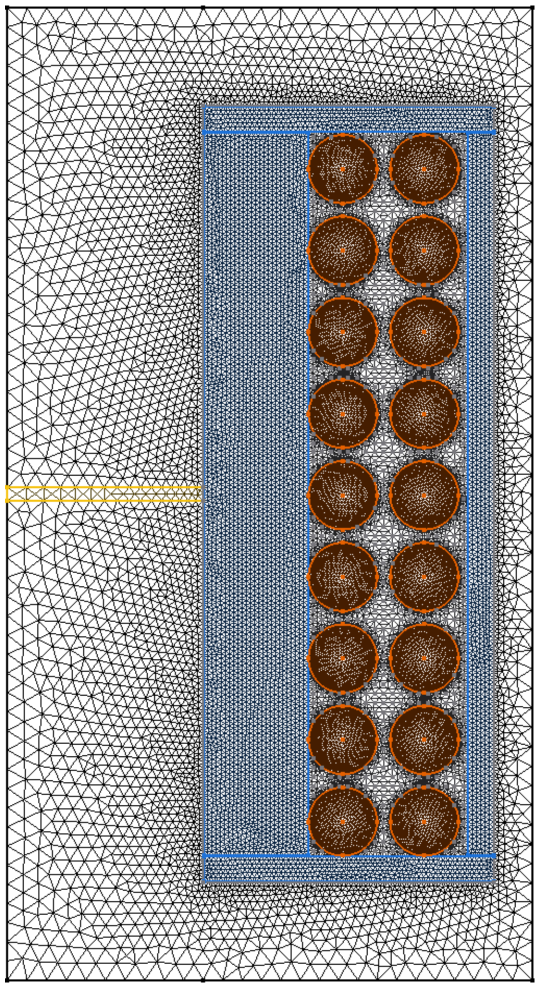

(b) Mesh with coarser winding window

Fig. 3.11: Mesh of basic inductor

3.5 Comparison of different meshing strategies

Now multiple techniques to lower the runtime of the FEMMT simulation were presented. In this section they shall be compared. From the previous section it got clear that changing the mesh in the winding window (section 3.4) has a big impact on the performance as well as the usage of insulations (section 3.2). Therefore, 4 possible configurations of those techniques are analyzed in this chapter. The total losses and the self inductance were logged to check the simulation precision and the execution time for the simulation performance. The possible configurations are:

1. **Insulation+WWR**: Insulations and winding window rasterization are set
2. **Insulation**: Only insulations ares set
3. **WWR**: Only winding window rasterization is set (insulations turned off)
4. **No Insulation, No WWR**: No winding window rasterization and no insulations

In fig. 3.12 the plot can be seen. For low mesh sizes, the results in total losses and self inductance are pretty much the same, but performance wise there are big differences. Additionaly it gets clear that the winding window rasterization has a very big impact on the simulation performance. On the other side the use of insulations does not have such a big impact and only makes the simulation a bit faster.

Overall the simulation results of all those 4 comparisons are very similar even for lower mesh accuracies.

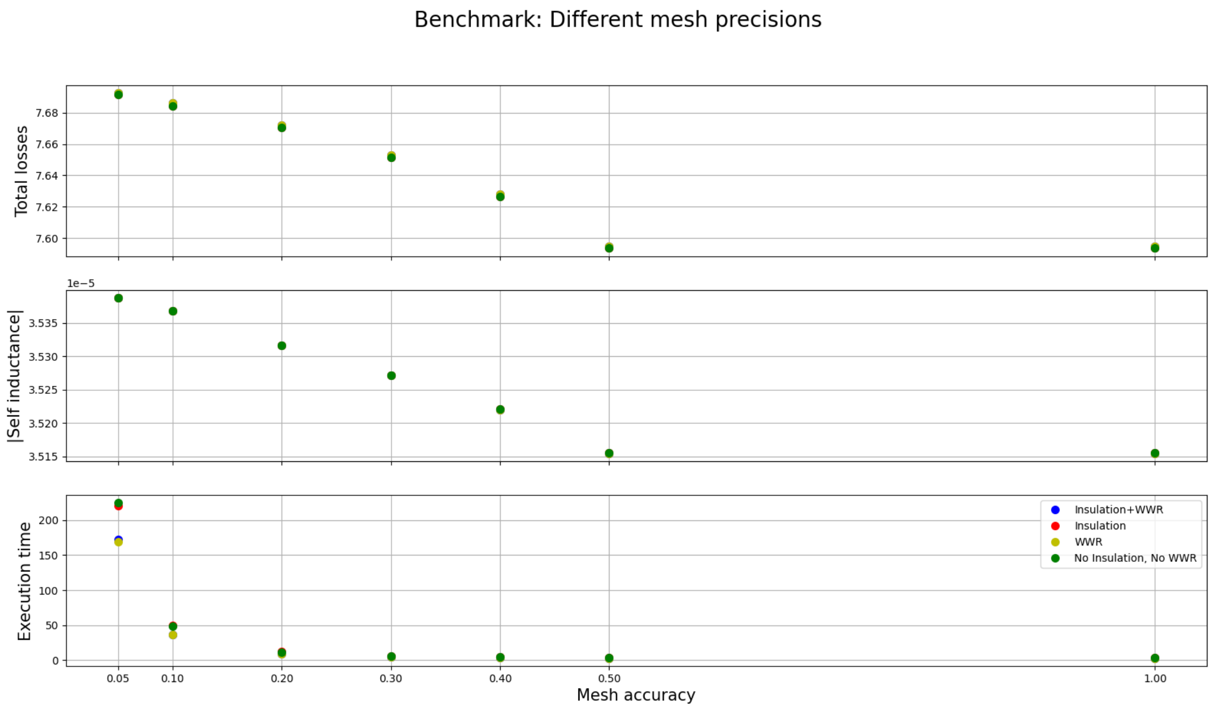


Fig. 3.12: Study over multiple meshing techniques and different mesh sizes

3.6 Results of speeding up single simulations

In this chapter multiple techniques for speeding up the single simulation were presented and implemented. Those techniques were:

- Adjusting insulations by adding aspect ratios
- Adding points in rectangular conductors to make the mesh inside coarser
- Adding points in the winding window to make the mesh coarser

Although the analysis on the aspect ratios showed that thee difference in precision and performance is not so big it also showed that it can make if difference in performance and simulation accuarcy if there are inuslations present or not.

Adding points in the rectangular conductor worked pretty well. The execution time of the simulation went down while the simulation results were still accurate.

Adding points in the winding window was also successful. Because the winding window has such a big area, also when there are many conductors, the coarser mesh makes a huge difference. The performance was increased greatly, and the precision was also fine.

Overall the task of speeding up the single simulation was sucesffully done and the increased performance will have a good impact on future projects in FEMMT.

4 Frequency and Time Domain Simulation

4.1 Principles of Frequency-Domain and Time-Domain

Time−domain and frequency−domain simulations provide two different perspectives for representing and analyzing signals or systems, particularly within fields such as electrical engineering, control systems, and signal processing. A time-domain simulation examines how a system or signal evolves over time. The output of this simulation is typically the system's time response to various inputs, which might be impulses, steps, or random inputs. Conversely, frequency-domain simulations focus on the system or signal's response to different frequencies. Hence, the choice between time−domain and frequency−domain simulations often depends on the specific application and analysis objectives. Each approach carries its own benefits: while time−domain simulation can provide a more intuitive understanding of system evolution, frequency−domain simulation can offer a clearer view of system behavior across various frequencies. The meaning of frequency and time domains can be shown in Fig. 4.1.

However, frequency−domain is not without its drawbacks. Frequency domain analysis often relies on the assumption that the system is linear and time-invariant, which may not always be the case, especially in scenarios where non-sinusoidal conditions are present. Additionally, this type of analysis is not well-suited for examining transient phenomena or non-periodic signals. Core losses, including hysteresis and eddy current losses, are inherently nonlinear, and accurately capturing these phenomena in the frequency domain can pose significant challenges and complexities.

Time−domain analysis excels in capturing the transient behaviors and nonlinearities in the system, providing a more comprehensive and accurate depiction of the core losses, crucial for optimizing the efficiency and performance of electrical machines and transformers. However, it is also important to note that time-domain simulations are computationally intensive and demand a detailed model of the system, which could be a drawback in terms of resource utilization and the time required for simulations.

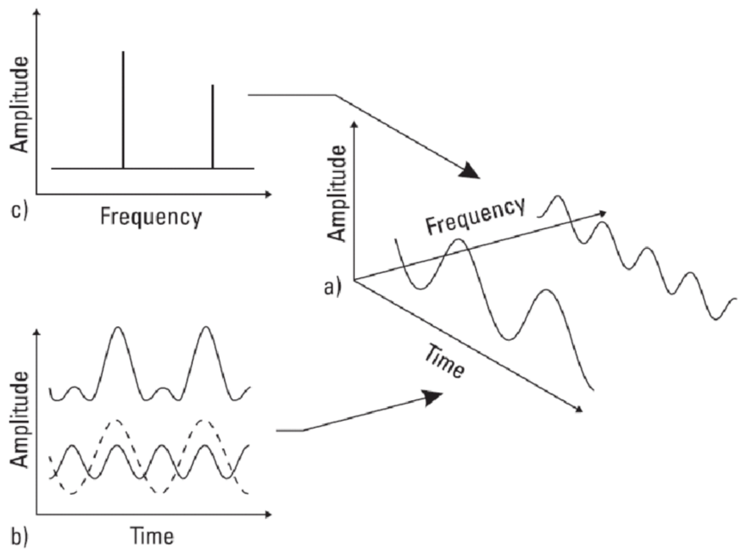


Fig. 4.1: Time domain vs. frequency domain

Despite these advantages, time domain simulations are not without their challenges. They can be computationally intensive and time-consuming, especially for complex systems. Furthermore, they require detailed information about the system and its components to produce accurate results. Interpreting the results from time domain simulations can also be less intuitive than those obtained from frequency domain analysis, particularly when it comes to understanding the impact of various frequency components.

Engineers are increasingly interested in time domain simulations due to their ability to provide accurate representations of nonlinear systems and to capture transient phenomena. This is crucial for understanding system reliability and performance, especially in response to faults, switching events, and other sudden changes. Time domain simulations also play a vital role in the validation of control strategies, ensuring that they perform as expected under various operating conditions ([9]).

In conclusion, while frequency domain analysis offers a simplified and intuitive way to analyze linear, time-invariant systems, time domain analysis stands out for its ability to accurately model and analyze nonlinear systems and capture transient phenomena. This makes it an indispensable tool for engineers, especially when it comes to minimizing core losses in electrical machines and transformers.

4.2 Advancing to Time-Domain Simulation

4.2.1 A Strategic Exploration in FEMMT

In the scope of FEMMT, the simulation was using frequency-domain simulations for different types of inductors and transformers. These simulations have provided valuable insights into these components' behavior and responses to various frequencies, revealing characteristics such as frequency-dependent losses. This approach, however, primarily captures the behavior of these components over a frequency value but lacks detailed information about how they react to changes over time. To address this gap and to gain a more comprehensive understanding of our system's behavior, the planning is to implement time-domain simulations. This approach will allow the user to observe the system's response to different inputs over time, enabling the user to analyze the system's transient behavior and to study different types of waves in greater details. Consequently, this blend of frequency-domain and time-domain simulations will provide a more holistic view of the system, improving the ability to optimize and control the system's performance.

Implementing time-domain simulations allows observing the system's response to various inputs over time, enabling the analysis of the system's transient behavior and providing a deeper insight into the impacts of different types of waves on the system. Such comprehensive analysis enhances the understanding of core losses, aiding in the development of strategies to mitigate these losses and optimize the system's performance.

In time-domain simulations, the initial time (t0), maximum time (t-max), step time (Δt), number of steps (NbSteps) constitute the temporal framework of the simulation process. t0 denotes the initial time at which the simulation is initiated, serving as a reference point where the initial conditions are defined or assumed. Conversely, t-max signifies the maximum time limit of the simulation, marking the point until which the system's behavior is analyzed. Between these two temporal markers, the simulation unfolds in increments defined by step time (Δt), giving the length of the interval for each step in the time-domain simulation. The step time establishes the temporal resolution of the simulation. Smaller step time (Δt) can lead to higher resolution and potentially more accurate results, albeit at the expense of increased computational effort. Larger step time (Δt), while computationally economical, may overlook swift changes in the system's behavior. The careful selection of these parameters, guided by the system's dynamics and the objectives of the simulation, forms a critical component of accurate and efficient time-domain simulation.

The journey into time-domain simulations commences with an in-depth exploration and trial of a time-domain example using the OneLab software suite. This step serves as a practical initiation into the functionality and application of time-domain simulations. Following this, the exploration deepens into the theoretical foundations of these simulations. The focus here is understanding the time-evolution of systems and signals, and how transient and steady-state behaviours emerge in this domain.

A pivotal component of the exploration will be deciphering the role of time derivatives in these simulations. Comprehending the concept of the rate of change of a quantity with respect to time (d ... / dt) will become instrumental in interpreting the results of the time−domain simulations. Complementing this understanding, the concept of temporal resolution or sampling the frequency at which data points are captured in the simulation will be investigated.

4.2.2 A Comparative Study using Time-Domain and Frequency-Domain Simulations in Onelab

Validating the results of time-domain simulations against frequency-domain ones is a key aspect of the simulation process. In the context of components like inductors and transformers, a crucial parameter to compare at the beginning is the losses. The time-domain analysis offers a different perspective. Here, the instantaneous power losses are integrated over a single period and divided by the length of that period to calculate the average loss.

Power calculations are consistent across both domains due to Parseval's theorem 4.1 , which states that the total power of a signal in the time domain is equal to the total power in the frequency domain([10]).

$$\int_{-\infty}^{\infty} |f(t)|^2 \, dt = \frac{1}{2\pi} \int_{-\infty}^{\infty} |F(\omega)|^2 \, d\omega \tag{4.1}$$

The correspondence between the frequency-domain losses and the average losses over a period in the time-domain forms a crucial point of validation. If both simulations are modeling the system accurately, these two quantities should align approximately. Any significant discrepancy could indicate an issue with one of the models or their implementation.

The simulation was executed for an inductor to attain the steady-state interval, ensuring reliable results. This process encapsulates both the transient stage and the eventual steady state, offering a comprehensive view of the inductor's behavior over time. By judiciously varying parameters such as the time step, the maximum time, and the Number of steps, the precision of the simulation was notably enhanced. The time step adjustment influences the granularity of the data, while manipulating the maximum time ensures complete cycles of the inductor's response are captured. This combination of parameter tuning methods fortifies the accuracy of the simulation, thereby yielding trustworthy and insightful results about the system's dynamics.

To authenticate the results of the simulation, a setup configured for a frequency of 50 Hz was employed, with maximum time (tmax) set to one period (T), the number of steps (NbSteps) set to 100, and the step time (Δt) is T/NbSteps. Consequently, the time step was determined as T/NbSteps. However, this configuration didn't enable the system to achieve a steady state within the given timeframe as shown in Fig. 4.2. This incomplete transition to the steady state, in turn, complicated the calculation of average losses over the period, as steady state conditions are typically requisite for such assessments.

Increasing the maximum time and reducing the step time facilitates a comprehensive depiction of the power loss signal over time. By doing that, the complete cycle of power losses can be captured effectively as shown in Fig. 4.3. Additionally, the use of a rolling average technique aligns with the power loss calculations in the frequency domain as seen

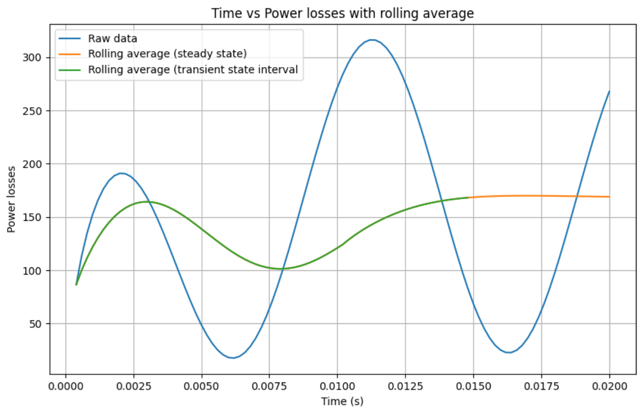


Fig. 4.2: Losses in an Inductor Over a Single Cycle

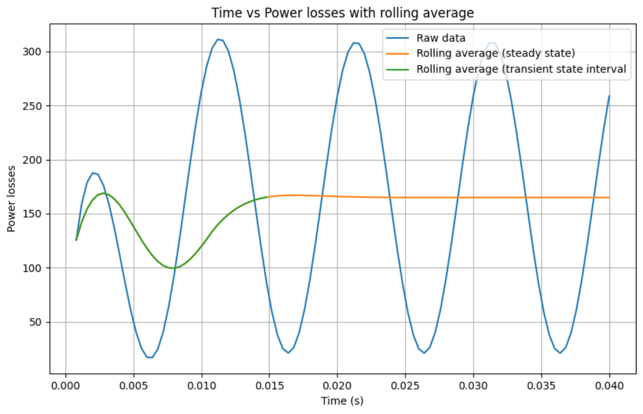


Fig. 4.3: Losses in an Inductor Over a multiple Cycle with 100 time steps

in table 4.1. This comparative method offers a means to validate and cross-verify the simulation results, strengthening the overall reliability of the study.

Increasing the number of steps in the time-domain simulation can augment the alignment between results from frequency and time domains, thereby bolstering the accuracy of the simulation. A finer granularity, achieved by increasing the number of steps, provides a more detailed representation of the system's behavior, resulting in a more precise comparison with the frequency-domain outcomes. This improved correlation between the two domains elevates the overall efficiency and reliability of the simulation process, providing more confidence in the simulation outcomes. The accuracy of results can be seen in Fig. 4.4 and table 4.2.

Tab. 4.1: Comparison of average losses results with different parameters

Characteristic of the inductor	
No. of turns	288
current (RMS)	10 Amp
frequency	50 Hz
NbSteps	100
T	0.02 s
Maximum time(tmax)	2T
Step time(Δt)	2T/NbSteps
Power losses in Frequency domain	171.29 W
Rolling Average (Final Value)	168.09 W

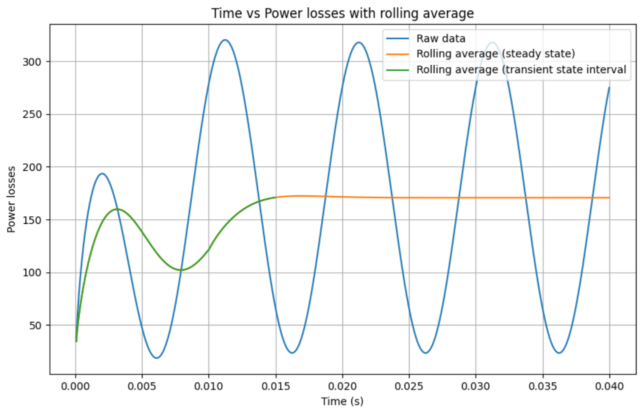


Fig. 4.4: Losses in an Inductor Over a multiple Cycle with 1000 time steps

Tab. 4.2: Comparison of average losses results with different parameters

Characteristic of the inductor	
No. of turns	288
current (RMS)	10 Amp
frequency	50 Hz
NbSteps	1000
T	0.02 s
Maximum time(tmax)	2T
Step time(Δt)	2T/NbSteps
Power losses in Frequency domain	171.29 W
Rolling Average (Final Value)	170.65 W

5 Advancements in FEMMT: Incorporating Time-Domain Simulations

5.1 A Time-Domain Structure in FEMMT

The foundational structure of FEMMT is constructed using a combination of Python files and non-Python text files. The latter category includes simulation scripts for solvers, as well as material details, all of which are crucial for finite element analysis. These files are primarily written in the GetDP language, a powerful environment for addressing problems across various dimensions and temporal states.

FEMMT facilitates finite element method (FEM) simulations through the ONELAB interface, interacting with the mesh generator GMSH and the finite element solver GetDP via an Application Programming Interface (API). The ONELAB (Python) module plays a crucial role in this process, creating a virtual connection between GMSH and GetDP, and handling the transfer of mesh and solver files to a sub-client that runs the simulations

In terms of file types and their roles within FEMMT, several are worth mentioning. Solver files carry the `".Pro"` extension, mesh files are labeled with `".msh"`, field results (such as the distribution of magnetic flux density and loss density) are stored in files ending with `".pos"`, and integrated field results (e.g., flux linkage, loss values, inductance) are saved in `".dat"` files.

A particular file, `"Parameter.Pro"`, acts as a communication hub between the solver scripts and Python, displaying variable parameter values for each simulation run. This includes details such as flag types, frequencies, resistances, phases, number of conductors, winding voltages, and permeability values.

The `"ind_axi_python_controlled_time.pro"` file encompasses a major portion of the GetDP script, describing the problem at hand, defining geometry, physical numbers, boundary conditions, resolution methods, equations, and related objects. Meanwhile, "solver.pro" takes on the task of mathematically solving the equations defined in the script.

As FEMMT evolved, additional files specifically tailored for time-domain simulations were incorporated, including `"ind_axi_python_controlled_time.pro"`, `"solver_time.pro"`, and `"fields_time.pro"`. These new files brought with them adaptations and enhancements to the original equation sets, now allowing for analyses that vary with respect to time. This broadens the scope of FEMMT, empowering it to capture transient behaviors and provide a more comprehensive understanding of electromagnetic components over time.

In essence, the inclusion of these time-domain specific files has enriched FEMMT's capabilities, transforming it from a tool proficient in frequency-domain analyses to a versatile platform adept in both simulations. This marks a significant stride in FEMMT's ongoing development, ensuring its relevance and effectiveness in a wide array of simulation scenarios.

5.2 Constructing Objects for Time-Domain Simulation Analysis

5.2.1 Excitation

In frequency-domain simulations, excitation is typically represented as sinusoidal functions or phasors. The magnitude and phase of the sinusoidal excitation are defined in relation to the operating frequency. This approach simplifies the solution of the electromagnetic field equations by transforming them into a steady-state problem, where the fields are assumed to vary sinusoidally with time. Listing 5.1 shows the applied excitation in frequency-domain. Here, FSinusoidal~{n}[] represents the sinusoidal excitation for each winding, defined by the frequency Freq, and phase shift Phase n. The sign of the excitation is determined based on the phase, introducing a negative sign if the phase is equal to π.

```
For n In {1:n_windings}
FSinusoidal~{n}[] = F_Cos_wt_p[]{2*Pi*Freq, Phase~{n}};
Fct_Src~{n}[] = FSinusoidal~{n}[];
Signn~{n} = (Phase~{n}==Pi) ? -1 : 1;
EndFor
```

Listing 5.1: Excitation in frequency-domain

In contrast, in time-domain simulations, the excitation is defined directly as a function of time. The sinusoidal excitation is replaced by an interpolation of values from a predefined list, which can represent arbitrary waveforms, not just sinusoidal signals. Listing 5.2 shows applied excitation in time-domain. Here, FSinusoidal~{n}[] is determined by linearly interpolating between the time points defined in TimeList, with corresponding current values from CurrentList. This method affords a greater versatility in depicting the excitation, permitting the inclusion of arbitrary waveforms and accommodating for transient events.

```
For n In {1:n\_windings}
FSinusoidal~{n}[] = InterpolationLinear[Time]{ListAlt[TimeList,
 CurrentList~{n}]};
Fct\_Src~{n}[] = FSinusoidal~{n}[];
EndFor
```

Listing 5.2: Excitation in time-domain

5.2.2 Resolution

The term "Resolution" in this context refers to the specific computational procedure or set of instructions defined to solve a physical problem or simulation. In numerical simulations, especially in fields like computational electromagnetics or finite element analysis, a resolution encompasses all the steps necessary to advance the simulation from its initial state to its final state, while capturing all the required data and performing necessary computations along the way. Listing 5.3 shows the builded resolution for time-domain simulation.

- **Analysis Definition**
  - The resolution is named `Analysis`.
  - A system named `A` is defined, associated with the formulation `MagDyn_a`.
- **Operation**
  - **Directory Creation:**
    - Directories for storing results of each winding are created.
    - A loop iterates through each winding, creating a result directory `DirResValsWinding~{n}` for the `n`-th winding.
  - **Solution Initialization:**
    - The solution for system `A` is initialized using `InitSolution[A]`.
  - **Time Loop:**
    - The simulation evolves from `time0` to `timemax` with a time step of `delta_time`.
    - The implicit Euler method is used for time integration, as indicated by the theta parameter being 1. The parameter "1" suggests the use of the implicit Euler method for time integration.
  - **Linearity Case:**
    - For linear simulations: The system generates and solves the equations with `Generate[A]` and `Solve[A]`.

  * For non-linear simulations: An iterative loop is initiated to ensure convergence, involving generating the Jacobian matrix and solving the system equations.

- **Solution Saving:**
  * The solution at the current time step is saved.
- **Post-Processing:**
  * Local mapping or processing is performed.
  * If the current time step is greater than 1, global post-processing is performed.

```
    Resolution {

      { Name Analysis ;
        System {
          { Name A ; NameOfFormulation MagDyn\_a ; }}
        Operation {
          For n In {1:n\_windings}
          CreateDir[DirResValsWinding~{n}];
          EndFor
          InitSolution[A] ;
          TimeLoopTheta[time0, timemax, delta\_time, 1.]{
            If(!Flag\_NL)
            Generate[A] ; Solve[A] ;
            Else
            IterativeLoop[Nb\_max\_iter, stop\_criterion,
            relaxation\_factor]{
              GenerateJac[A] ; SolveJac[A] ;
            }
            EndIf
            SaveSolution[A] ;

            PostOperation[Map\_local] ;
            Test[ \$TimeStep > 1 ]{
              PostOperation[Get\_global];

              //PostOperation[T\_resampled];}}
        }// Operation
      }
    }// Resolution

```

Listing 5.3: Resolution in time-domain

5.2.3 Parameter Representation in Post Processing

In the frequency-domain, parameters and variables are often represented as complex values to capture both the magnitude and phase information of oscillating phenomena. An example of this can be seen in Listing 5.4, where the reluctivity, a material property, is a function of the small change in the magnetic vector potential and the frequency.

```
    # Define magnetic field in frequency domain
    Name h;
    Value {Term {[ nu[{d a}, Freq]*{d a} ]; In Domain; Jacobian Vol
      }
    }

```

Listing 5.4: Magnetic field in frequency-domain

In time-domain simulations, the focus shifts from oscillating phenomena in the frequency space to transient phenomena in the time space. This necessitates a more straightforward representation of parameters and variables, primarily using real values. An example of this can be seen in Listing 5.5, where the reluctivity is now solely a function of the small change in the magnetic vector potential, with the frequency component no longer explicitly present. This change reflects the shift in focus from capturing the steady-state response in the frequency domain to capturing the instantaneous response in the time domain.

```
    # Define magnetic field in frequency domain
    Name h;
    Value {Term {[ nu[{d a}]*{d a} ]; In Domain; Jacobian Vol
      }
    }

```

Listing 5.5: Magnetic field in time-domain

5.2.4 Formulation

The same behavior has been meticulously incorporated into the formulation equations for time-domain, ensuring that the simulations are finely tuned to their respective contexts. In the context of frequency-domain simulations, the equations often involve complex numbers to accommodate the oscillatory nature of the phenomena being studied. In the context of time-domain simulations, the focus shifts from the steady-state oscillations typical of frequency-domain analyses to capturing transient and dynamic behaviors of systems as they evolve over time. Listing 5.6 and 5.7 show an example of a numerical method used to approximate solutions to differential equations, particularly in finite element analysis for frequency and time domain receptively.

```
    Galerkin { [ nu[Norm[{d a}], Freq] * Dof{d a} , {d a} ]   ;
      In Domain\_Lin ; Jacobian Vol ; Integration II ; }

```

Listing 5.6: Galerkin equation in frequency-domain

```
    Galerkin { [ nu[Norm[{d a}], Freq] * Dof{d a} , {d a} ]  ;
      In Domain\_Lin ; Jacobian Vol ; Integration II ; }

```

Listing 5.7: Galerkin equation in time-domain

5.2.5 Post Operation

In transitioning from frequency to time domain simulations, significant modifications have been made to the code behaviors, particularly in terms of how results are stored and structured in output files. These changes are essential to align with the unique characteristics and requirements of each domain, ensuring accurate and efficient data handling. Listing 5.8 and 5.9 show an example in frequency and time domain respectively.

In frequency domain simulations, the output is typically complex, capturing both magnitude and phase information. Consequently, the results are often stored in three distinct lines within the output files, frequency value, real value, and imaginary value.

In contrast, time domain simulations focus on transient behaviors, tracking the system's response over consecutive time steps. As such, the file construction is adapted to two lines, one is for time step and the other for the result value. The structure, with its alternating lines of time steps and result values, stands as a testament to the precision and attention to detail that time domain simulations demand. It ensures that the data is a coherent and intuitive record of the system's journey through the transient landscape of time.

```
      Print[ j2F[ Core ], OnGlobal, Format TimeTable,
      File > StrCat[DirResVals,"CoreEddyCurrentLosses.dat"]];

```

Listing 5.8: structure of a file in frequency-domain

```
    Print[ j2F[ Core ], OnGlobal, Format TimeTable,
    File > StrCat[DirResVals,"CoreEddyCurrentLosses.dat"],
    LastTimeStepOnly, StoreInVariable $j2F];

```

Listing 5.9: structure of a file in time-domain

6 Time-Domain Simulation Results

6.1 Running Simulations with"geo.time_domain_simulation"

The geo.time_domain_simulation function is a versatile tool designed for time-domain simulation in FEMMT. It takes several arguments that allow users to customize the simulation according to their specific requirements as shown in listing 6.1. Here's a breakdown of how to use the function:

```
geo.time_domain_simulation(freq=inductor_frequency,
current=[current_list_winding1],[current_list_winding2], .. ],
time=[time_list],
time_period= 1 / inductor_frequency,
initial_time= 0,
timemax= 2 / inductor_frequency,
NbSteps=50,
delta_time= (2 / inductor_frequency) / 50,
plot_interpolation=False,
show_fem_simulation_results=True,
show_rolling_average=False,
rolling_avg_window_size=50)
```

Listing 6.1: Time-domain simulation's function

6.1.1 Parameter Breakdown

freq: This parameter represents the frequency of the component type that is being simulated.

current: A nested list, where each inner list contains current values to be used in the simulation for a specific winding. The outer list encompasses all windings, with each inner list corresponding to one winding. For a simulation with multiple windings, each winding's current profile is provided as a separate list, allowing the simulation to apply different current values to each winding over time.

time: A list of time points that should correspond one-to-one with the current values in the 'current' list. This list is used to interpolate the simulation data at each specified time step. The length of the 'time' list must be equal to the length of each inner list within the 'current' list to ensure that each current value is associated with a specific time point in the simulation.

time_period: The total period of one cycle of the simulation, usually the inverse of the `freq` parameter.

initial_time: The starting time of the simulation.

timemax: The maximum time limit for the simulation to run. This is where the simulation will stop computing the results.

NbSteps: Number of steps or intervals the simulation will divide the time domain into. This dictates the resolution of the simulation.

delta_time: The time increment between each step of the simulation. It is calculated by dividing the total time by the number of steps.

show_rolling_average: This parameter, typically a boolean, decides whether a rolling average is applied to the results, which can smooth out fluctuations for better analysis.

rolling_avg_window_size: When `show_rolling_average` is True, this parameter defines the size of the window over which the rolling average is computed.

6.1.2 Using the Function

1. **Setup the Simulation Parameters:** Define the frequency, current profile, and time parameters based on the physical scenario you want to simulate.
2. **Initialize the Simulation:** Invoke the `geo.time_domain_simulation` with the defined parameters. Make sure `initial_time` is set correctly to avoid non-physical transient artifacts if necessary.
3. **Run the Simulation:** The function will use the `NbSteps` and `delta_time` to iterate through the simulation, computing the state at each time step.
4. **Post-Processing:** Use the `show_rolling_average` with an appropriate `rolling_avg_window_size` to smooth the data.
5. **Interpret the Results:** After the simulation runs, the user can embark on a comprehensive analysis of the time-domain results to elucidate the dynamics of the system being simulated. The simulation's output is typically stored in designated directories as data files. These files can be accessed and opened. For a detailed inspection of the results at each time step, GMSH's native `.pos` files are particularly useful. They can be directly opened in GMSH, which provides an interactive environment to visualize field distributions, current densities, and other simulation results. By navigating through different time steps within GMSH, the user is able to track the evolution of the system's response over time, which is critical for validating the simulation model and interpreting its physical implications.

6.2 Visualization of Time-Domain Simulation Results

Fig. 6.1, Fig. 6.2, and Fig. 6.3 present snapshots from steps 1, 25, and 50, respectively, of a 50-step simulation process for an inductor. The figures represent the progression of the system's behavior as captured by the time-domain simulation at selected intervals. These snapshots are instrumental in visualizing the transient dynamics that occur over the course of the simulation period.

Each figure captures a unique stage in the simulation, reflecting the response of the system to the applied conditions at that specific instance in time. The visuals serve to provide an intuitive grasp of the temporal evolution without delving into the underlying complexities of the simulation.

As we move through the time steps, the figures collectively tell the story of the system's transient behavior, from the initial conditions to the final state at the conclusion of the simulation.

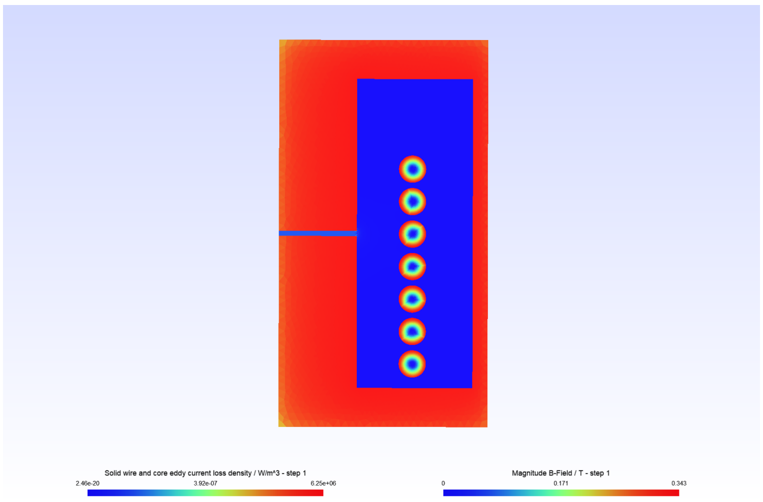


Fig. 6.1: System response at step 1.

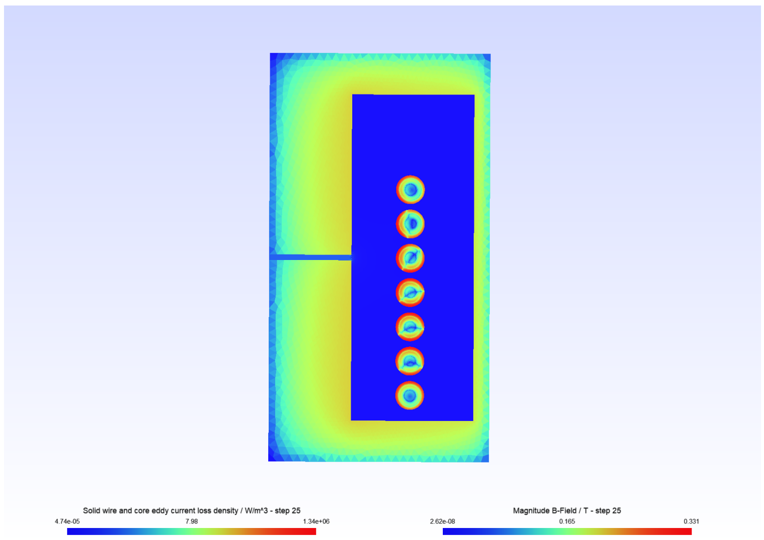


Fig. 6.2: System response at step 25.

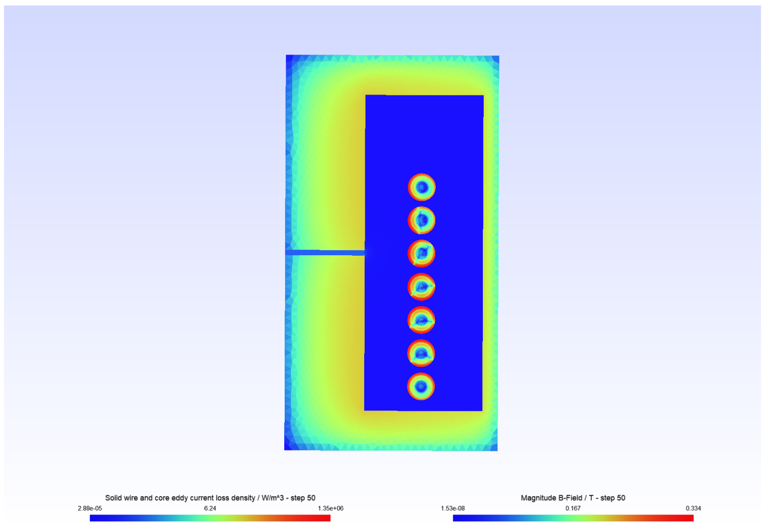


Fig. 6.3: System response at step 50.

6.3 Validation of Time-Domain Simulation Results

To ensure the accuracy of the time-domain simulations conducted in FEMMT, the results were validated through comparison with frequency-domain analysis outcomes. Given that the frequency-domain results in FEMMT have been previously validated against established FEMM and ANSYS simulations, this comparison serves as a robust validation method. By comparing the time-averaged results from the time-domain simulations with the frequency-domain data, the reliability of time-domain simulation processes within FEMMT are affirmed.

The consistency in excitation methods across simulations is noteworthy: both the frequency-domain and time-domain analyses utilized a sinusoidal current excitation. To simulate this in the time domain, a sinusoidal current was synthesized from a list of values, ensuring the input mirrored the periodic nature required for an accurate comparison.

A critical aspect of the validation process involved the examination of average power losses. As illustrated in Fig. 6.4, the average winding losses derived from the time-domain simulations were juxtaposed with those from the frequency-domain analyses within FEMMT.

The validation of losses for individual turns was meticulously conducted. Figures 6.5 and Fig. 6.6 illustrate that the average losses computed from the time-domain simulations correspond closely with those derived from frequency-domain analysis.

The consistency observed between the two domains reinforces the validity of time-domain simulation outcomes, ensuring that the models can reliably replicate the complex dynamics of electromagnetic systems over time.

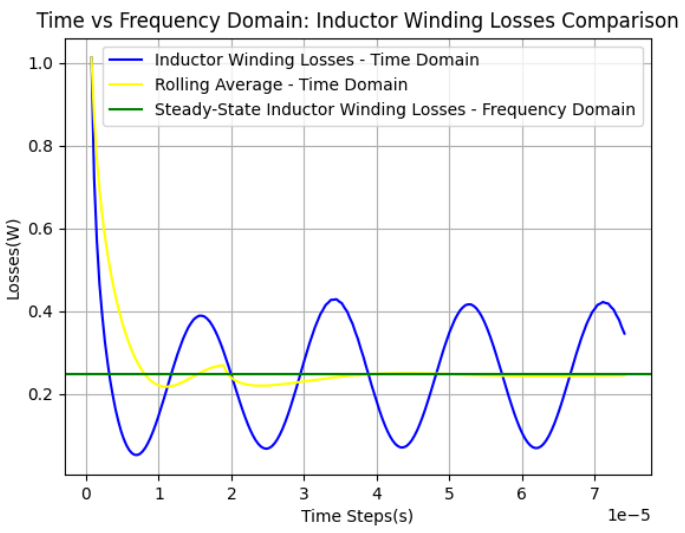


Fig. 6.4: Inductor winding losses

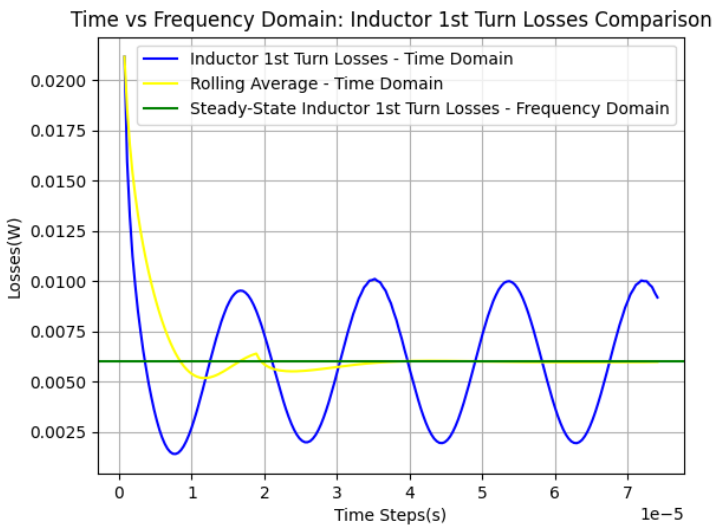


Fig. 6.5: Inductor 1st turn losses

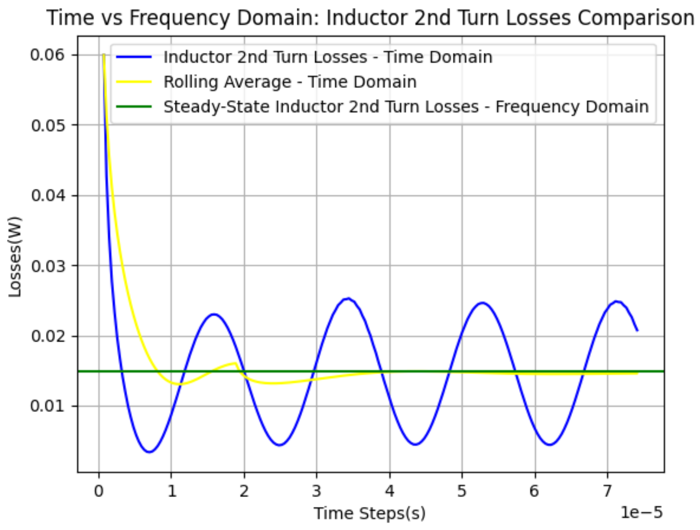


Fig. 6.6: Inductor 2nd turn losses

6.4 Post Processing in Python

One of the central aim is to facilitate a smooth translation of time-domain analysis results into an easily interpretable and well-structured document. The written "calculate_and_write_log" function in Python is a comprehensive post-processing utility crafted to assimilate and interpret the results of simulations for magnetic components such as inductors and transformers as shown in listing 6.2. It incorporates three nested sub-functions that each contribute to the post-processing procedure in a unique way. The nasted function "freq_domain_log" handles the frequency-domain aspects of the simulations, typically associated with steady-state analysis. The nested function "time_domain_log" is specifically tailored for time-domain analysis, which is key for capturing transient responses and the intricate time-varying interactions within the magnetic components. It deals with extracting and logging transient phenomena, which are critical for understanding the performance and reliability of these components under real-world electrical conditions. The nasted fuction "common_log" aggregates common data and calculations relevant to both time-domain and frequency-domain analyses, ensuring that the fundamental parameters are consistently logged.

However, The "time_domain_log" function, in this context, would be responsible for collecting and recording all relevant data during a time-domain simulation, providing a comprehensive understanding of the component's dynamic behavior. This could include instantaneous voltage, current, and flux linkage at each time step of the simulation, providing insight into how the component performs over time. It computes various average losses, such as core and winding losses, which occur during the operation of the magnetic component in the time domain. This is essential for understanding and improving the efficiency of the component. It aggregates and summarizes performance metrics like RMS values, total power, and total losses.

```python
def calculate_and_write_log(self):

def freq_domain_log(sweep_number: int = 1, currents: List = None,
frequencies: List = None, core_hyst_losses: float = None):

def time_domain_log():

def common_log(inductance_dict: dict = None):
```

Listing 6.2: Post Processing in Python

7 Conclusion

The project successfully expanded the functionality of the Finite Element Method Magnetics Toolbox (FEMMT) by implementing a time-domain simulation feature for electromagnetic components and offering parallel simulations by implementing an hpc function. Those features make the FEMMT more versatile and also delivers much higher performance.

The implementation of the hpc function was done successfully by making sure that the different parts of the simulations can run in parallel and the resulting speedup was in an expected range. Speeding up the single simulation also resulted in a lower execution time while the precision of the simulation was not affected too much. This makes FEMMT a more user-friendly and versatile tool since it does lower the execution time for optimization routines which significantly shortens the developement and prototyping time for inductors and transformers.

The integration of the time-domain simulation within FEMMT is a significant advancement, offering a more dynamic andversatile tool for the analysis and design of electromagnetic devices. The frequency-domain results from FEMMT have already been benchmarked and confirmed for their accuracy by comparing them with results from established software such as FEMM and ANSYS. To further ensure the reliability of FEMMT, the time-domain simulation results have been averaged and successfully compared with these frequency-domain results, reinforcing the validity of FEMMT's simulations.

Additionaly the time-domain solver does dicectly benefit from the simulation speedup since the time-domain simulation needs to run a whole simulation per time step. This is possible since the mesh of the frequency domain is the same as of the time-domain. Also, it is possible to run multiple time-domain simulations in parallel.

7.1 Future Work

In the pursuit of enhancing the Finite Element Method Magnetics Toolbox (FEMMT), future work could address the complex challenge of accurately modeling core losses, particularly hysteresis losses, in time-domain simulations. Core losses, consisting of hysteresis, eddy current, and excess losses, are significant in electromagnetic components, affecting performance and efficiency.

Not only the time-domain simulation can be advanced even further, there is still room for improvement in the execution time. The cmesh of the core is only one aspect which could be adressed in the future. It could be analyzed if coarsening the mesh in the core has an impact on the execution time and on the precision of the simulation.

References

[1] Fachgebiet Leistungselektronik und elektrische Antriebstechnik, *Fem magnetics toolbox (femmt)*. [Online]. Available: https://github.com/upb-lea/FEM_Magnetics_Toolbox.

[2] W. Stallings, *Operating systems: Internals and design principles*, 5. ed., internat. ed. Upper Saddle River, NJ: Pearson Education International, 2005.

[3] T. Rauber and G. Rünger, *Parallele und verteilte Programmierung*, ser. Springer-Lehrbuch. Berlin, Heidelberg and s.l.: Springer Berlin Heidelberg, 2000.

[4] G. N. Tornetta, *Austin: A frame stack sampler for cpython*. [Online]. Available: https://github.com/P403n1x87/austin.

[5] W. C. Gibson, *The method of moments in electromagnetics*, Third edition, ser. A Chapman & Hall book. Boca Raton, London, and New York: CRC Press Taylor & Francis Group, 2022. [Online]. Available: https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=6707600.

[6] Solidworks, Ed., *Mesh quality checks*. [Online]. Available: https://help.solidworks.com/2021/english/SolidWorks/cworks/c_mesh_quality_checks.htm.

[7] C. Geuzaine and Remacle Jean-Francois, *Gmsh reference manual*. [Online]. Available: https://gmsh.info/dev/doc/texinfo/gmsh.pdf.

[8] T. Piepenbrock, "Automated fem transformer design for a dual active bridge," Masterarbeit, Universität Paderborn, Paderborn, 2021.

[9] B. Boashash, *Time-frequency signal analysis and processing: a comprehensive reference*. Academic press, 2015.

[10] A. V. Oppenheim, A. S. Willsky, S. H. Nawab, and J.-J. Ding, *Signals and systems*. Prentice hall Upper Saddle River, NJ, 1997, vol. 2.

Appendix

A.1 Custom hpc for parallel FEMMT execution

In A.1 an example for a custom hpc function which varies current and phi_deg instead can be seen.

```python
def custom_hpc(parameters: Dict):
    """Very simple example for a custom hpc_function which can be given
    to the hpc.run() function.

    :param parameters: Dictionary containing the model and the given
    simulation_parameters.
    :type parameters: Dict
    """
    model = parameters["model"]
    simulation_parameters = parameters["simulation_parameters"]

    if "current" not in simulation_parameters:
        print("'current' argument is missing. Simulation will be skipped
    .")
        return
    if "phi_deg" not in simulation_parameters:
        print("'phi_deg' argument is missing. Simulation will be skipped
    .")
        return

    current = simulation_parameters["current"]
    phi_deg = simulation_parameters["phi_deg"]

    model.create_model(freq=250000, pre_visualize_geometry=False)
    model.single_simulation(freq=250000, current=current,
        phi_deg=phi_deg, show_fem_simulation_results=False)
```

Listing A.1: run_hpc function example

A.2 Flame graph of basic inductor

In Figure A.1 the whole flame graph of the basic inductor can be seen. It gets clear that the mesh generation in gmsh and the simulation in GetDP take the most time, the code which is executed in FEMMt to create the necessary configuration files and prepare the models is barely seen.

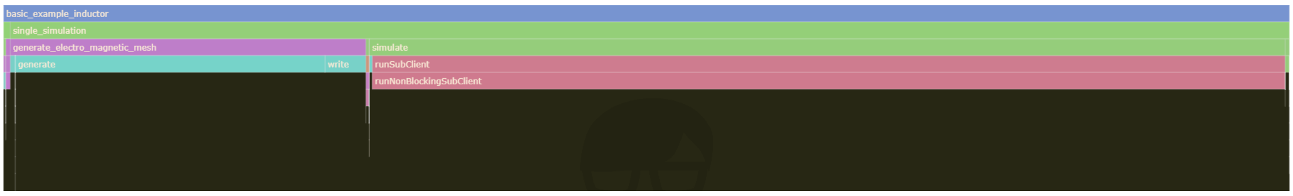


Fig. A.1: Parallel simulation study on multiple computers with different process counts

A.3 Rolling Average Execution in FEMMT

Listing A.2 shows the rolling average implementation in Python.

```python
def rolling_calculation(res_name: str, res_path: str):
  timesteps = []
  data = []
  # Reading data from the file
  with open(os.path.join(res_path, f"{res_name}.dat")) as fd:
    lines = fd.readlines()
  # Extracting timesteps and data values from the file
  for line in lines:
    line_values = line.split()
    if len(line_values) == 2:
      timesteps.append(float(line_values[0]))
      data.append(float(line_values[1]))
  # Error handling: Raising error if window size is greater than
  # the number of data points
  if window_size > len(data):
    raise ValueError("The window size should not be greater than
    the total number of data points.")
  # Creating a pandas DataFrame and computing the rolling average
  df = pd.DataFrame({'data': data})
  df['rolling_avg'] = df['data'].rolling(window_size, min_periods=1).
mean()
  rolling_averages = df['rolling_avg'].tolist()
```

Listing A.2: Rolling average function

A.4 Functions needed for Calculating the average and RMS Values in FEMMT

Listing A.3, A.4, A.5, and A.6 detail the essential functions utilized for integrating time-domain data. These functions are critical for determining average values through direct integration, as well as for computing the root mean square (RMS) values by integrating the square of the data over time

```python
def calculate_quadrature_integral(timesteps, data):

  func = lambda x: np.interp(x, timesteps, data)
```

```
    return quadrature(func, timesteps[0], timesteps[-1])[0]s=1).mean()

```

Listing A.3: Integral function

```
  def calculate_squared_quadrature_integral(timesteps, data):

    func = lambda x: np.interp(x, timesteps, data)**2
    return quadrature(func, timesteps[0], timesteps[-1])[0]

```

Listing A.4: Integral function for squared data values

```
  """
  This function calculates the average in general
  """
   def calculate_average(integral, timesteps):

    total_time = timesteps[-1] - timesteps[0]
    if total_time == 0:
    raise ValueError("Total time cannot be zero.")
    return integral / total_time

```

Listing A.5: Average function

```
  def calculate_rms(squared_integral, timesteps):
  """
  This function calculates the RMS value in general
  """
    total_time = timesteps[-1] - timesteps[0]
    if total_time == 0:
    raise ValueError("Total time cannot be zero.")

    mean_square = squared_integral / total_time
    return np.sqrt(mean_square)

```

Listing A.6: RMS function