NoLOAD, Open Software for Optimal Design and Operation using Automatic Differentiation

Lucas Agobert, Sacha Hodencq, Benoit Delinchant, Laurent Gerbaud, Wurtz Frederic

NoLOAD, Open Software for Optimal Design and Operation using Automatic Differentiation

Lucas AGOBERT, Sacha HODENCQ, Benoît DELINCHANT, Laurent GERBAUD, Frederic WURTZ

Univ. Grenoble Alpes, CNRS, Grenoble INP, G2Elab, 38000 Grenoble, France
E-mail: lucas.agobert@grenoble-inp.fr; sacha.hodencq@grenoble-inp.fr; benoit.delinchant@grenoble-inp.fr

Abstract. Purpose : Solving non-linear optimization problems such as physical component sizing or optimal control of a system is still a challenge. Automatic Differentiation (AD) is a way to provide useful information regarding model local sensitivity and helps algorithm finding minima. AD is not new but has recently found great interest in machine learning community. **Methodology** : In this paper, we are introducing NoLOAD, a Python open source software that helps designers to associate non-linear models to optimization algorithms with AD. Different AD packages are compared (Autograd / Jax) as well as hardware architectures (CPU / GPU). **Findings :** Jax package is more performative than Autograd package for complex models, although it uses more memory to solve optimization problems. **Originality :** NoLOAD is easy to use for designers because there is no need to change the simulation model which is totally independent of the specifications. It is also a lightweight library than can be used for embedded hardware optimization.

Keywords: Optimization, Automatic Differentiation, NoLOAD, Jax, Autograd

INTRODUCTION TO OPTIMIZATION PROBLEMS

Physical systems designs can be improved by inverse problems resolution. Inverse problems consists in computing model inputs while knowing model outputs, also called specifications. Optimization problems are considered as inverse ones. Once a physical system's model is formulated, optimization procedure consists in computing input variables, so that an objective function (that usually represents the economic cost of the system) is minimized as much as it can be. The input variables must be included in a certain gap delimited by bounds, to assure the solution obtained is physically possible. Moreover, equality and inequality constraint functions are often considered, and their conditions must also be respected, to avoid physical sizing issues. The mathematical expression to define an optimization problem is the following:

$$(1)\quad \min_{\mathbf{x}\in\boldsymbol{\Omega}} \mathbf{f}(\mathbf{x}) \text{ with } \mathbf{c}(\mathbf{x}) = \mathbf{0} \text{ and } \mathbf{d}(\mathbf{x}) = \mathbf{0}$$

With f: $\mathbb{R}^N$ -> $\mathbb{R}$ the objective function, c: $\mathbb{R}^N$ -> $\mathbb{R}^M$ the equality constraints, d: $\mathbb{R}^N$ -> $\mathbb{R}$ the inequality constraints, $\boldsymbol{\Omega} \subset \mathbb{R}$ the set of admissible values.

To solve constrained optimization problems, algorithms use Jacobian computations. While it is quite easy to calculate gradients for linear models, it is harder to do it for non-linear models. Two procedures can be applied: the first one is that the designer linearizes its model around a working point then the algorithm determines the linearized model's gradient, and the second one is to develop methods to directly compute the non-linear model's gradient.

Several Python libraries for solving optimization problems were developed in the past: two examples are OpenMDAO [1] and Gekko [2]. They use a similar syntax to define optimization problems, combining model equations and specifications. However, this syntax cannot be accessible to everyone. Therefore, the Grenoble Electrical Engineering Laboratory (G2Elab) developed a new open source software NoLOAD, which is easy to understand for designers. It is intended to solve 2 kinds of optimization problems for non-linear systems:

- System sizing: It consists in computing optimal values of a system's physical components to respect given specifications.
- Optimal control: It consists in computing the optimal control command to apply to a given system to respect wished outputs.

The first part of this article deals with Automatic Differentiation (AD), which is a method to compute gradients, its advantages to solve optimization problems and the Python libraries of AD that will be used in NoLOAD.
The second part is the presentation of the NoLOAD software, its syntax and its benefits compared to existing software.
The third part is an enumeration of benchmarks, based on real electromagnetic systems, that will be used to compare computation performance between AD libraries.
The last part is about the comparison of AD libraries performances to solve optimization problems, on benchmarks presented in the third part.

AUTOMATIC DIFFERENTIATION (AD)

AD is the exact numerical calculation of a function's gradient [3][4]. To compute it, AD uses the Chain rule.

Supposing a y function such as: y = f(u) = f(g(x)) = (f o g)(x), the chain rule states:

$$\frac{dy}{dx} = \frac{dy}{du} * \frac{du}{dx} \quad (2)$$

With this theorem, any function can be seen as a composition of usual mathematical functions (such as exponential, logarithm, trigonometric ones), for which the derivative is easy to compute. This allows the AD technique to consume less memory than formal calculation. Moreover, AD provides more precise computations than other derivation methods such as the finite-difference one.

There are two AD modes that allow to compute the derivative of any function: the forward mode and the reverse mode.

The forward mode is the easiest to understand and to implement. The function is evaluated at the wished point, and each part of the computation is seen as an intermediate variable *vi*, then each intermediate variable is derived in parallel, so that the gradient of the function is computed according to a chosen independent variable. This process must be repeated several times if derivatives need to be computed according to lots of input variables. Therefore, forward mode is more efficient for functions f: $\mathbb{R}^n$ -> $\mathbb{R}^m$ with m>n because fewer iterations will be required to compute the Jacobian matrix.

Besides, the reverse mode consists in computing the derivative backwards after evaluating the function at the wished point. Each intermediate variable is associated with an adjoint $\tilde{v}i$ that represents the outputs sensitivity according to *vi*, and that is described by the following formula:

$$\tilde{\nabla} i = \frac{\partial y}{\partial vi} \quad (4)$$

By starting from the output, the derivative's computation allows to find the wished adjoint. This process must be repeated several times if derivatives for functions with lots of outputs need to be computed. Therefore, reverse mode is more efficient for functions f: $\mathbb{R}^n$ -> $\mathbb{R}^m$ with n>m because fewer iterations will be required to compute the Jacobian matrix.

Several python libraries allow to implement AD: PyADOL-C, PyCppAD and Theano use the operator overloading technique, whereas CasADI uses the source transformation technique [5]. This last technique is more complex than the previous one but leads to faster run-time speeds. AD is a key of recent successes of deep learning [6], mainly based on reverse mode because deep learning problems are composed of multiple inputs and one output (the objective function), while forward and reverse modes can be required in the domain of design engineering [7]. However, two recently developed AD libraries were preferred to be added to NoLOAD: Autograd[1] and Jax[2]. They both have been developed by the Harvard Intelligent Probabilistic System (HIPS) Group. They can both differentiate Python code using NumPy library, and both integrate reverse and forward modes. One major difference is that Jax includes the XLA functionality (Accelerated Linear Algebra) that makes it able to run either on GPU (Graphics Processing Unit) or TPU (Tensor Processing Unit) hardware. That is why Jax should theoretically provide better computation performances than Autograd. However, Jax only runs on Linux operating system and not yet on Windows one, unlike Autograd that can run on both software.

WHAT IS NOLOAD ?

NoLOAD (Non-Linear Optimization using Automatic Differentiation) is a Python free and open source software[3] able to automatically connect the model, to an optimization algorithm. It allows a designer to define constraints on the input and output parameters of the model, and one or more objective functions to minimize. One benefit of NoLOAD is to be accessible without optimization knowledge, by coupling automatically the design specifications to the model and the optimization algorithm using AD. NoLOAD is shared with the licence Apache 2.0, making its models usable both for researchers and designers. The source code is accessible and can be discussed online, in particular for reporting issues and for comparison with existing tools.

Two different versions of NoLOAD were developed: one running with the Autograd library, which will mainly be used for the simplest models and academic usages, whereas the other version running with Jax library will be used by researchers and preferred to solve complicated models. The optimization algorithm is a SciPy SLSQP (Sequential Least Quadratic Squares Programming) based on quasi–Newton method with a BFGS update (Broyden-Fletcher-Goldfarb-Shanno).

The class diagram of NoLOAD software is shown in *Figure 1*:

[1] https://porter.io/github.com/HIPS/autograd
[2] https://github.com/google/jax
[3] https://pypi.org/project/noload/

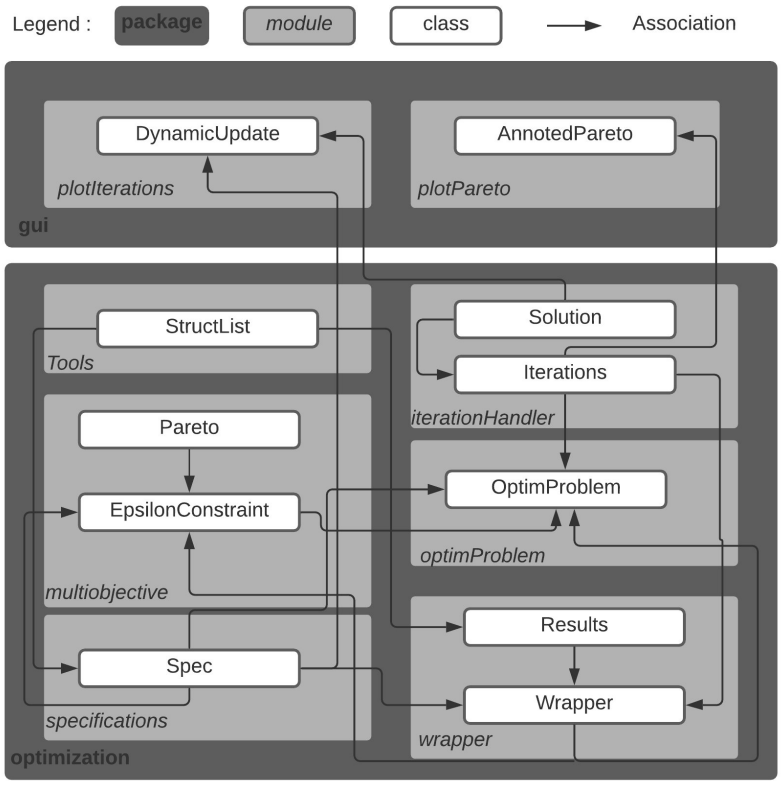

Figure 1: Class Diagram of NoLOAD.

To illustrate this diagram, an example of an optimization problem solved by NoLOAD is explained below :

First, the simulation model is defined as a Python function : in this case, the objective function to minimize is the Ackley one.

```
def ackley(x,y):
      import jax.numpy as np
      import math
      fobj =-20* np.exp(-0.2* np.sqrt(0.5*(np.square(x)+np.square(y))))-
np.exp(0.5*(np.cos(2* math.pi * x)+ np.cos(2* math.pi * y)))+
math.exp(1)+20
      return locals().items()
```

Then, the specifications wanted by the designer are written in with the *Spec* class of the *specifications* module (see *Figure 1*).

```
from noload.optimization.optimProblem import Spec, OptimProblem
spec = Spec(variables={'x':2, 'y':2}, bounds={'x':[-5, 5], 'y':[-5, 5]},
      objectives=['fobj'], eq_cstr={}, ineq_cstr={})
```

In this problem, there are 2 input variables ‘x’ and ‘y’ to optimize. The research domain of each one is limited by the gap [-5;5] and to start the algorithm, both have 2 as initial value. This is an unconstrained optimization problem. If there were constraints, they would have been specified in 2 dictionaries : one for the inequality constraints, and the other for the equality ones.

After that, the problem is defined with the *OptimProblem* class of the *OptimProblem* module (see *Figure 1*).

```
optim = OptimProblem(model=ackley, specifications=spec)
```

It is composed of the model simulation and the specifications given before.

Then, the simulation is run with options such as the algorithm used to solve the problem ('SLSQP' means Sequential Least Squares Quadratic Programming), the precision accorded to the objective function value, and the maximum number of iterations the algorithm can reach.

```
result = optim.run(method='SLSQP',ftol=1e-5, maxiter=500)
result.printResults()
```

The output given by the Python software is the following :

```
Optimization terminated successfully.      (Exit mode 0)
      Current function value: 6.6443758178991l7e-05
      Iterations: 9
      Function evaluations: 20
      Gradient evaluations: 9
{'x':1.5781116638803522e-05, 'y':1.739422385733534e-05}
{'fobj':6.644375817899117e-05}
```

The optimal solution found by the algorithm is [0;0] and the minimum value of the objective function is 0.

As shown in the previous example, NoLOAD provides a different syntax to define optimization problems unlike other Python libraries such as OpenMDAO or Gekko: the model and the specifications are totally independent, so that a designer does not have to adapt his way of writing his model. Moreover, NoLOAD is a lightweight library which can be used for hardware embedded optimization. NoLOAD also has a heuristic characteristic: it computes the number of inputs and outputs of a given model, and uses the right mode between forward and reverse, to improve the computation performances. It also allows problems with vectorial constraints.

BENCHMARKS

To test performances of AD, two optimal sizing benchmarks are proposed (electromagnetic transformer, electromagnetic claw-pole generator), and one optimal control benchmark (homeroom temperature control, including ODE solving) for which the temporal horizon is a parameter to increase complexity.

The simplest benchmark corresponds to a transformer sizing [8] (*Figure 2)*, which has 4 input variables to optimize:

- bt: magnetic induction (T) / dI: current density (A/mm²) / h: height of the windings (m) / N2: number of turns in the secondary.

There also are 2 parameters that will be constant during the optimization:

-mat_bob: discrete parameter of the winding: copper if mat_bob=0 or aluminium if mat_bob=1 / mat_CM: discrete parameter of the magnetic circuit: M6 if mat_CM=0 or M2 if mat_CM=1

For this benchmark, 8 different outputs can be considered:

- Material Price: cost of raw materials (€) / Total Price: total cost of the transformer (€) / Losses Price: cost of total losses (€) / Iron losses: cost of iron losses (€) / Joule losses: cost of joule losses (€) / Efficiency: transformer efficiency / U1CCpu: short circuit voltage (% of nominal voltage) / llt: total length of the transformer (m)

For the optimization problem, Total Price will be chosen as the objective function to minimize, U1CCpu = 6% as equality constraint, and llt < 1,2 m as inequality constraint.

Figure 2: Electromagnetic transformer

The second benchmark corresponds to an optimal control one [9], for which the size (number of inputs/outputs) can vary according to the chosen energy horizon management in days. The goal of this benchmark is to optimize a heating power, so that a building room temperature is kept above a threshold value for each hour of k days, with k the chosen energy management horizon, according to other collected data related to the building (outside temperature, sun power, room occupants power).

That is why, for an energy management horizon of k days, this benchmark is composed of :

- An input vector corresponding to the heating power Pheat of 24*k elements (one per hour).
- An objective function to minimize that represents the energy cost of the system in €.
- An output vector of 24*k-1 elements, named Tint, in which each element corresponds to the room temperature at each hour (except the first one which is already fixed). This vector is considered as an inequality constraint: the temperature must be above 20°C between 6 am and 8 am and between 6 pm and 11 pm, or above 0°C otherwise.

The third benchmark is a claw-pole generator sizing [10] (*Figure 3)*, which is described by :

- 55 inputs to optimize, including loop fluxes : 10 for each of the 3 phases of the claw-pole generator ; and also rotor reluctances, stator and rotor sections, continuous currents on each phase of the stator and the rotor, stator width, claw width.
- An objective function which represents the weighted sum of phases yields.
- 76 outputs, including 36 equality constraints : 12 functions for each phase which must be equal to 0 ; and 40 inequality constraints : 21 on inductions which must be constrained between 0.01 and 2 T, and 19 other ones such as average currents for each phase, and density currents for each phase of the rotor and the stator.

Figure 3: Claw-pole generator [11]

The design specifications of these benchmarks are summarized in *Table 1*, showing the increasing problem complexity.

	Transformer	**Heating control (1 day)**	**Claw-pole generator**
Python model file weight (kilobytes)	9	12	61
Design or control parameters:	4 scalars	a 24 element-wised vector	55 scalars
Total of 1 objective function + constraints on model outputs	3 scalars	24	77 scalars
including equality constraints:	1 scalar	/	36 scalars
including inequality constraints:	1 scalar	a 23 element-wised vector	40 scalars

Table 1: Characteristics of the benchmarks design specification

Other benchmarks developed with NoLOAD are available online : https://gricad-gitlab.univ-grenoble-alpes.fr/design_optimization/noload_benchmarks_open

In the next section, the performances between Autograd and Jax are compared on 3 benchmarks, which come from real electrical systems with different kind of optimal design specifications. Hardware performances are part of the comparison since the Jax library is able to use GPU (Graphics Processing Unit) acceleration.

AUTOMATIC DIFFERENTIATION PERFORMANCES

Calculations were performed on a server whose characteristics are indicated here : Ubuntu 20.04, CPU: 2,4GHz Intel XEON Silver 4210R with 10 cores, RAM: 31 GB, GPU: NVIDIA Quadro RTX 6000. The memory usage, average simulation time, and iterations number for 100 optimizations starting with a random initial guess are given for each use case in *Table 2*. The optimization algorithm uses the SciPy Python package, particularly the SLSQP method based on quasi–Newton method with a BFGS update, using 10^{-5} precision for the objective value in the stopping criterion.

Autograd / Jax (**CPU** or **GPU**)	**Transformer**	**Heating control (1 day)**	**Claw-pole generator**
Iterations number	**12** **12** **13**	**29** **30** **30**	**124** **132** **132**
Simulation time (seconds)	**0,15** **0,85** **1,65**	**13,5** **6,5** **4**	**1650** **54** **30**
Memory used	**23,16 Kb** **1,90 Mb** **1,90 Mb**	**2,38 Mb** **4,92 Mb** **4,94 Mb**	**1,59 Mb** **32,23 Mb** **33,07 Mb**

Table 2: NoLOAD optimization performances comparison for Autograd and Jax on 3 benchmarks

Then, the simulation times for the optimal heating control model according to its size are displayed in *Figure* 4:

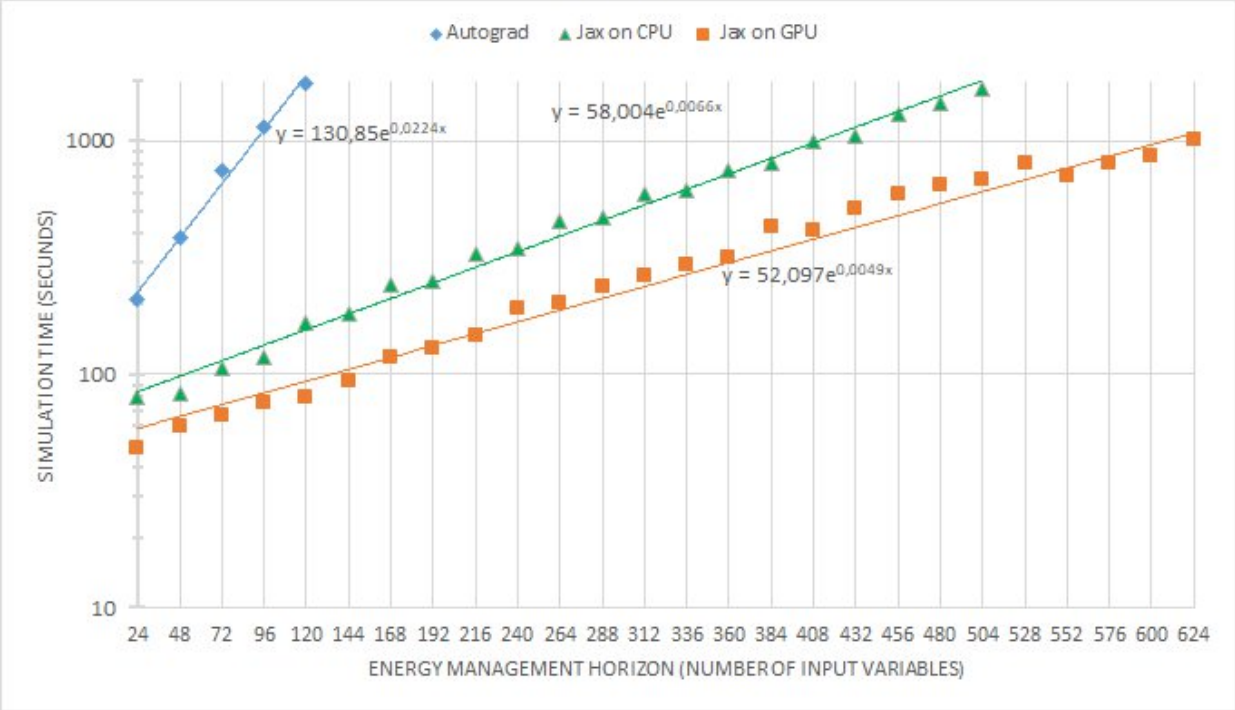

Figure 4: Simulation times of the optimal heating control against the number of days (time step).

By doing linear regression, equations of trend exponential curves are the following:

- for Autograd: $yA = 130{,}85e^{0{,}0224x}$
- for Jax on CPU: $yJCPU = 58{,}004e^{0{,}0066x}$
- for Jax on GPU: $yJGPU = 52{,}097e^{0{,}0049x}$

For an optimal heating control problem of 120 input variables (which belongs to a 5-days energy management horizon), ratios are computed to quantify the difference in speed between Jax and Autograd :

- Ratio Autograd / Jax CPU = $130{,}85e^{0{,}0224*120}/58{,}004e^{0{,}0066*120}$ = ***15,02***
- Ratio Autograd / Jax GPU = $130{,}85e^{0{,}0224*120}/52{,}097e^{0{,}0049*120}$ = ***20,51***

Therefore, for this optimal heating control problem, Jax running on GPU is 15 times faster than Autograd and Jax running on CPU is 20 times faster than Autograd.

PERFORMANCES ANALYSIS AND CONCLUSIONS

Even using a random initial guess, the solutions are the same for both AD libraries, and the iteration number are equivalent allowing concluding accuracies of the Jacobian are similar. However, Autograd uses less memory than Jax, which can be important for some problems. On the other hand, Jax is several times quicker than Autograd for big problems such as claw-pole generator (*Table 1*) or for the heating control (*Figure 4*).

It is known that non-linear constrained optimizations are hard problems to solve. With the increasing complexity of systems to design, which may embed the energy management strategy, the number of constraints and design variable is increasing too. We are still looking for complexity reduction strategies, focusing for instance on important parameters using sensitivity analysis, or trying to reduce the analysis time period with clustering technical for instance. However, optimal implementation of computer programs is also critical and we have shown

in this paper that AD, which is a key ingredient in optimization of electromagnetic devices, can bring significant performance differences.

The results presented in this article were computed in a limited research setting, with the use of AD and the operator overloading implementation in Python. If other computer tools were used, better results could have been found. For instance, by implementing the AD in Julia, the transformer benchmark was optimized in less than 2 milliseconds, which is 100 times quicker than with NoLOAD. However, operator overloading is a less complex way to implement AD.

ACKNOWLEDGEMENTS

The authors would like to thank P.Haessig and E.Antunes, respectively associate professor and master thesis student at the Centrale Supelec Engineering School for the Julia computations mentioned in the conclusion.

REFERENCES

[1] Heath, C. and Gray, J. (2012). OpenMDAO: Framework for Flexible Multidisciplinary Design, Analysis and Optimization Methods. *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference20th AIAA/ASME/AHS Adaptive Structures Conference14th AIAA*.

[2] Beal, L., Hill, D., Martin, R. and Hedengren, J. (2018). GEKKO Optimization Suite. *Processes*, 6(8), p.106.

[3] Griewank, A. and Walther, A. (2009). *Evaluating derivatives : principles and techniques of algorithmic differentiation*. Philadelphia, Pa.: Society For Industrial & Applied Mathematics ; Cambridge.

[4] Enciu, P., Wurtz, F., Gerbaud, L. and Delinchant, B. (2009). Automatic differentiation for electromagnetic models used in optimization. COMPEL - *The international journal for computation and mathematics in electrical and electronic engineering*, 28(5), pp.1313–1326.

[5] Turkin, A. and Thu, A. (2016). Benchmarking Python Tools for Automatic Differentiation. *arXiv:1606.06311* [cs]. [online] Available at: http://arxiv.org/abs/1606.06311 [Accessed 26 Sep. 2021].

[6] Raschka, S., Patterson, J. and Nolet, C. (2020). Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence. *Information*, 11(4), p.193.

[7] Pearlmutter, B. (n.d.). Automatic differentiation in machine learning: a survey. *www.academia.edu*. [online] Available at: https://www.academia.edu/17708071/Automatic_differentiation_in_machine_learning_a_survey [Accessed 26 Sep. 2021].

[8] Dinh, V.-B., Delinchant, B., Wurtz, F. and Dang, H.-A. (2018). Building Modelling Methodology Combined to Robust Identification for the Temperature Prediction of a Thermal Zone in a Multi-zone Building. *Lecture Notes in Computer Science*, pp.226–237.

[9] Delinchant, B., Mandil, G. and Wurtz, F. (2018). Comparing life cycle cost and environmental impact optimizations of a low voltage dry type distribution transformer. COMPEL - T*he international journal for computation and mathematics in electrical and electronic engineering*, 37(2), pp.645–660.

[10] Delale, A., Albert, L., Gerbaud, L. and Wurtz, F. (2004). Automatic Generation of Sizing Models for the Optimization of Electromagnetic Devices Using Reluctance Networks. *IEEE Transactions on Magnetics*, 40(2), pp.830–833.

[11] Albert, L. (2004). *Modélisation et optimisation des alternateurs à griffes. Application au domaine automobile*. [online] Available at: https://tel.archives-ouvertes.fr/tel-00007091/document [Accessed 5 Oct. 2021].