

Article

DynPy—Python Library for Mechanical and Electrical Engineering: An Assessment with Coupled Electro-Mechanical Direct Current Motor Model

Damian Sierociński ^{1,*} , Bogumił Chiliński ¹ , Franciszek Gawiński ² , Amadeusz Radomski ² 
and Piotr Przybyłowicz ³ 

¹ Department of Computer Techniques, Institute of Machine Design Fundamentals, Faculty of Automotive and Construction Machinery Engineering, Warsaw University of Technology, 02-524 Warsaw, Poland; bogumil.chilinski@pw.edu.pl

² Faculty of Automotive and Construction Machinery Engineering, Warsaw University of Technology, 02-524 Warsaw, Poland; fgawinski@gmail.com (F.G.); amadeuszradomski@icloud.com (A.R.)

³ Department of Mechanics, Institute of Machine Design Fundamentals, Faculty of Automotive and Construction Machinery Engineering, Warsaw University of Technology, 02-524 Warsaw, Poland; piotr.przybylowicz@pw.edu.pl

* Correspondence: damian.sierocinski@pw.edu.pl

Abstract: *DynPy* is an open-source library implemented in *Python* (version 3.10.12) programming language which aims to provide a versatile set of functionalities for mechanical and electrical engineers. It enables the user to model, solve, simulate, and report analysis of dynamic systems with the use of a single environment. The *DynPy* library comes with a predefined collection of ready-to-use mechanical and electrical systems. A proprietary approach to creating new systems by combining independent elements defined as classes, such as masses, springs, dampers, resistors, capacitors, inductors, and more, allows for the quick creation of new, or the modification of existing systems. In the paper examples for obtaining analytical and numerical solutions of the systems described with ordinary differential equations were presented. The assessment of solver accuracy was conducted utilising a coupled electro-mechanical model of a direct current motor, with *MATLAB/Simulink* (R2022b) used as a reference tool. The model was solved in *DynPy* with the hybrid analytical–numerical method and fully analytically, while in *MATLAB/Simulink* strictly numerical simulations were run. The comparison of the results obtained from both tools not only proved the credibility of the developed library but also showed its superiority in specific conditions.

Keywords: coupled electro-mechanical model; software tools for mechanical engineering; software tools for circuit simulation; solving methods; numerical techniques; analytical solution; Python programming language; direct current motor



Academic Editors: Ioana-Gabriela Sirbu and Lucian Mandache

Received: 15 November 2024

Revised: 4 January 2025

Accepted: 7 January 2025

Published: 13 January 2025

Citation: Sierociński, D.; Chiliński, B.; Gawiński, F.; Radomski, A.; Przybyłowicz, P. *DynPy*—Python Library for Mechanical and Electrical Engineering: An Assessment with Coupled Electro-Mechanical Direct Current Motor Model. *Energies* **2025**, *18*, 332. <https://doi.org/10.3390/en18020332>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Simulations are an essential part of modern science, as they provide a multitude of benefits. Starting with the possibility of analysing very complicated processes, impossible or barely impossible to be conducted in real-life conditions, to ‘what if’ scenarios, hypotheses evaluation, understanding of processes, and optimisations, ending with designing and testing new technologies and solutions. The goal of this article is to present the abilities of the independent, *Python*-programming-language-based *DynPy* (‘DynamicPython’) [1] library and compare the results obtained from simulations of a dynamic system conducted

with well-known, field-tested commercial software. A growing demand for clean energy and transportation introduces new challenges to multiple engineering branches [2–5]. As a result, a continuous seeking for more reliable, durable, safe, and effective hardware design is seen. This usually means a necessity for analysing yet more complicated phenomena [6], for which nontrivial approaches are needed, most likely to be investigated with the use of some software.

DynPy is an open-source project aimed at providing an engineering tool that allows, among other functionalities: performing symbolic computations, conducting numerical simulations, and creating comprehensive reports of conducted calculations. It supports a few useful and suitable typesetting systems for reporting engineering and scientific activities, e.g., *Markdown* and *LaTeX*. Another advantage of the developed library is the significant number of dynamic components it provides. This allows for quickly simulating the behaviour of systems described by mathematical dependencies characterizing their dynamic states. *DynPy* is conceptualised, developed, and maintained by the authors and other developers. Throughout the years it has been utilized by various researchers in the field of mechanical engineering. Due to the constantly growing capabilities of the developed tool, its use in research could be categorized regarding the formerly available set of functionalities. In [7], the proprietary methodology for creating dynamic systems, as well as numerical and analytical solving algorithms, were utilised. Further, in [8,9], chosen elements of a text editing submodule were additionally used for adding formulas and other minor typesetting activities. Finally, the article in [10] was the first one to be created as a whole strictly with *DynPy*'s functionalities—from model creation, through simulations, to manuscript generation. The authors are not aware of any other examples but do not exclude such a possibility, as the library is available in a public *GitHub* (version 2.34.1) repository. Such circumstances are the effect of the current article being the first one explicitly about *DynPy*, as the state of development allows for presentation to the broader public.

In the case of electrical systems, the description of their dynamics most often takes the form of ordinary differential equations (further referred to as ODE). Due to the complexity of the formulas, solutions can be found in analytical forms or with the use of numerical algorithms. The biggest advantage of the analytical approach lies in the provision of exact solutions for the whole domain, which ensures high repeatability and, thus, reliability of achieved results. Even though it is not always possible to directly obtain closed-form formulas, the stated aspects push scientists to find different mathematical tools, which lead to overwork of hitherto numerically analysed phenomena with analytical methods also in the case of electrical circuits [11–14]. Although, in the current state, the *DynPy* library does not provide as specialised functionalities as presented in the cited literature, it is crucial to note its ability in this aspect. Firstly, a broad spectrum of typical electrical elements is predefined with the possibility of further enlargement. Secondly, basic analytical and numerical solvers are implemented. Thanks to the utilisation of an object-oriented paradigm and open-source code, more sophisticated approaches can be introduced on the user's demand.

As noted earlier, the ability to achieve analytical solutions is case-specific and highly depends on the type of equations describing the system. Numerical simulations became such a powerful tool due to the provision of approximated results to problems, which could not be worked out by traditional means [15–17]. Multiple types of numerical solving methods have been introduced throughout the years [18]. In the case of ODE, three main families of algorithms can be recognised: explicit single-step, explicit multistep, and their implicit analogues dedicated to stiff systems of differential equations [19]. Typical single-step methods for ODE may be divided into the Runge–Kutta (RK, fixed step size) and Runge–Kutta–Fehlberg (RKF, variable step size) approaches. Examples of their utilisation in analyses of electrical circuits in both nonstiff and stiff ODEs can be found in

[20–28]. Moving on to multistep algorithms, again two main groups could be highlighted, known as Adams and predictor–corrector methods. The Adams family methods in analyses of electrical circuits were presented in [29–35]. Predictor–corrector (PC) algorithms include both explicit and implicit algorithms at once, which results in obtaining a set of properties connecting their benefits. The sequence of operations is as follows: prediction (explicit, P), evaluation (E), correction (implicit, C), evaluation (E) [36,37]. The described rule is outright applicable to predictor–corrector algorithms based on Adams–Bashforth and Adams–Moulton methods, but constant research on the effects of numerical approximations in the development of different approaches including explicit–explicit or implicit–implicit PC algorithms [38–45]. The *DynPy* environment in its current state provides six types of numerical solving techniques, inherited from the *SciPy* (version 1.15.0) [46] module: *RK45*—explicit Runge–Kutta method of order 5(4); *RK23*—explicit Runge–Kutta method of order 3(2); *DOP853*—explicit Runge–Kutta method of order 8; *Radau*—implicit Runge–Kutta method of the Radau IIA family of order 5; *BDF*—implicit multistep variable-order (1 to 5) method based on a backward differentiation formula for the derivative approximation; and the *LSODA*—Adams/BDF method with automatic stiffness detection and switching. Simultaneously with other development and maintenance works, actions are taken to create *DynPy*-embedded, standalone versions of these and other useful algorithms, so a package can become more versatile and less susceptible to third-party changes in the code, possibly due to the previously stated inheritance.

Solving and simulating electrical circuit behaviour can be performed with the use of multiple software tools available on the market. While some of them strictly require the user to formulate code instructions, others have more advanced *Graphical User Interfaces* which provide the possibility to create a representation of analysed problems by block diagrams or 3D models. Vendors such as *MathWorks* (Natick, USA) or *Wolfram* (Champaign, IL, USA) offer platforms of both types [47–50]. Each of them uses individual language, mutual between different developed environments. The mentioned programs serve a multipurpose function, being used by engineers of imaginably all branches. Throughout the years, many additional packages and add-ins were introduced, so the functionalities of interest for specific sectors of the industry could be grouped and state better fitment to the consumers [51,52]. When such universal tools are not found as a suitable solution for any reason, more task-oriented software is accessible, such as *TINA*, whose scope is restricted to electrical/electronic circuit modelling and simulations [53]. All of the highlighted examples, despite their undeniable advantages, share one undisputable drawback—to obtain a full experience of their abilities, whether in general or in exact context, a license has to be acquired, which often does not include additional toolboxes or add-ins offered independently at different prices. Although it is possible to find free and open-source alternatives, such as *Scilab* [54] or *CircuitLab* [55], they are not flawless—less developed or completely lack predefined electrical elements, one-off types of interfaces, and a smaller group of users effects in worse community support (what becomes a serious problem when diving into analyses of more complex phenomena, which, may even outcome as too complicated for the utilised software), and so on.

2. Materials and Methods

2.1. *DynPy* Library—General Overview

The decision to use *Python* programming language for the implementation of the *DynPy* library was dictated by its numerous advantages. It is used by hundreds of thousands of programmers worldwide in areas such as web scripting, system programming, user interfaces, and product customisation [56,57], and the opinion is becoming widespread that it is an optimal choice for scientists and engineers seeking a high-level language for

writing scientific applications [58]. As a high-level language, its syntax is more comprehensible for users, putting more focus on the task that has to be carried out rather than analysing the aspects of computing areas that naturally are not taken into consideration by nonprogrammers, such as memory management. The general purpose character of the language results in a wide variety of up-to-date libraries addressing multiple fields, such as scientific computing, data analysis, data visualisation, and more. Thanks to that, it was possible to use pre-existing modules as a foundation to implement functionalities desired functionalities, instead of writing the code from scratch. *NumPy* (version 2.2.1), *pandas* (version 2.2.3), *SciPy* (version 1.15.0), and *SymPy* (version 1.13.3) [59] are concurrently used for data processing, computations, simulations, and various mathematical operations. *Pandas*, in collaboration with *Matplotlib* (version 3.10.0), is used for presenting data and results in tabular and graphical form, while *PyLaTeX* (version 1.4.2) provides crucial methods required for text formatting and generating .tex and .pdf files. Therefore, *Jupyter Notebook* (version 7.3) is perfectly suited as a working environment with the *DynPy* library [60]. The ability to execute code in separate cells allows for conducting calculations in a specified order. The visualisation of operations facilitates understanding and control over the performed computations. Additionally, the option of combining code, text, and images makes the developed document readable and more accessible to other users.

In the process of selecting a programming paradigm from multiple which *Python* (version 3.10.12) offers, the decision was made to adopt an object-oriented programming approach, as it enables the creation of structured and modular solutions. Object-oriented code is more transparent and flexible, facilitating project management and development and providing versatility and usefulness. In the context of the project, it enables easier management of multiple modules and provides a structure that facilitates collaboration among multiple software developers.

It is worth mentioning at this point that the work on the *DynPy* library project follows an Agile approach, which is based on flexibility, adaptability, and continuous adjustment to changing needs. In contrast to the classical technique, which assumes planning and implementation in one cycle, the Agile methodology focuses on short-cycle stages. The goal is divided into smaller parts, and work is carried out step by step with the possibility of adapting to new conditions. This not only provides greater flexibility but also enables faster delivery of useful engineering tools [61]. The project as a whole is not presented further in this work. Instead, individual sections are presented and discussed.

The algorithms implemented in the library allow the solving and simulating of linear and nonlinear systems, both numerically and analytically. The numerical techniques inherited from *SciPy* provide a possibility for the analysis of linear and weakly and strongly nonlinear problems. The analytical approaches are defined in `dynpy.solvers.linear` and `dynpy.solvers.nonlinear` submodules. On top of the typical approaches utilised when working on linear ODEs, a few useful methods are defined which enable dealing with nonlinear problems. Firstly, a functionality of linearisation around an introduced point is provided, where calling the `linearized` method on the provided equations of motion returns a new set of formulas based on Taylor polynomial expansion of the nonlinear terms. Secondly, perturbational solvers are implemented, such as the Multiple Scales Method. The effectiveness in dealing with such problems in both numerical and analytical ways was tested in the previously referred literature, namely [7,8]. In detail, the library allows for the application of a few approaches for solving ODEs according to the class of the problem under analysis:

- **Linear:** *DynPy* provides analytical and numerical solvers for linear systems. The scope of the application of analytical solvers is limited by the order of the equation and its parameters. The system of two second-order equations can be solved analytically

in general cases (all parameters given as arbitrary symbols). For selected cases of parameter arrangement, equations of higher order are also possible to solve. All other linear cases can be efficiently solved with numerical solvers as described previously.

- **Nonlinear:** This class of equations can be mostly solved by numerical solvers provided in *DynPy*. It results in the `TimeDataFrame` object (*DynPy* modification of *pandas* `DataFrame` object, providing data structure optimised for handling results in time and frequency domain), a uniform interface for further analysis (e.g., calculations, transformations) and manipulation (embedding the tables and graphs in the report) of results. This approach is consistent with an analytical approach, which the user can handle in a standardised way as presented before. Moreover, the selected cases are possible to solve in an analytical way if the equation is solvable from the theoretical point of view. First-order and Bernoulli equations are implemented currently.
- **Perturbational:** Selected cases can be solved with the usage of the Multiple Scale Method, which is an analytical method for solving systems with weak nonlinearities or parametrical properties. The equation has to be described by the perturbational parameter (from a practical point of view—a small physical part of the system). *DynPy* provides several tools that allow to create a perturbational form of the equation and solve it with the same methodology and in the same environment as previously described cases.

The *DynPy* library is equipped with a series of dynamic models that simulate real systems, the base of which is a `ComposedSystem` class. An electrical circuit can be constructed by leveraging its functionalities and incorporating components provided by the `dynpy.models.electric.elements` module (e.g., `Resistor`) or by utilizing predefined systems, the availability of which continues to expand. An exemplary electrical circuit represents a Thevenin model of the lithium-ion battery, a code snippet of which is provided in Appendix A. Using such a class eliminates the need for independently designing a system while also allowing for easy analysis. Additionally, the models include graphical visualisations, as shown in Figure 1. Expanding the *DynPy* library with various models is an effective way to enhance its functionality. Thanks to a standardised approach to building models, adding more becomes simpler.

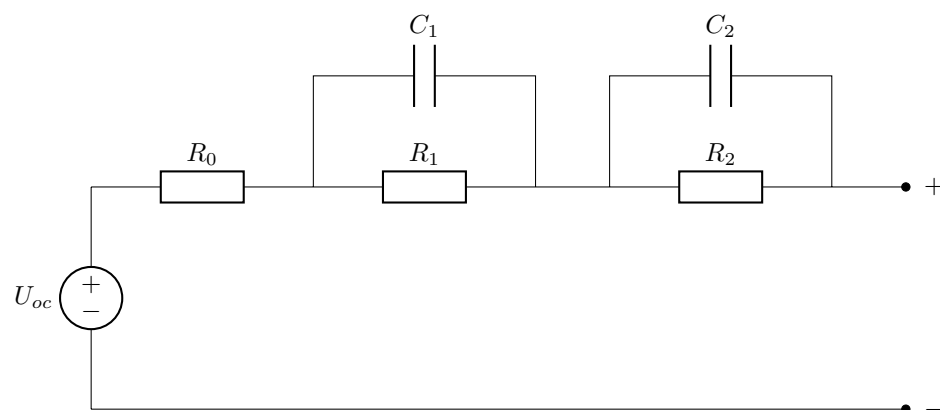


Figure 1. Graphical representation of the BatteryCell dynamic model.

2.2. Solvers

The core of the simulational capabilities of the *DynPy* library lies in its `dynpy.solvers` module. Technically, the most important class of all defined in its submodules is `ODESystem`, which serves as a template for creating objects in the form of differential equations. In the scope of the class body, there is an example demonstrating its use to create a differential equation and subsequently solve it symbolically and numerically using the tools provided

by the library, which can be displayed by calling the `help(ODESystem)` command in the interpreter.

Preparation of the defined model for simulations in the *DynPy* environment starts with the creation of `ODESystem` class instance. To satisfy its necessary arguments, equations of motion of the system have to be introduced, as well as dependent variables and the degree of the differential equation. The ODEs can be introduced in two different ways, either by directly defining them as an input argument or by creating a dynamic system class and using its `_eoms` property defined in the parent class `ComposedSystem`. The latter method is presented, as it shows *DynPy*'s functionalities in a wider aspect. A code snippet of such a constructed `DCMotor` class is presented in Appendix B. Such a class can be defined directly in the interpreter or (preferably) in the proper module. In the case of providing the definition in a module, it is necessary to import it into an interpreter. Next, creating a noncalled instance is suggested, so it gives access to inherited and internally defined objects. A properly constructed class enables the return of equations of motion, which can be directly introduced as an argument to `ODESystem`. With a similar approach, a matrix of dependent variables can be formulated. The last step is the provision of an ODE order. As a result, a complete representation of the `ODESystem` type object is obtained. A graphical representation of the described steps and generated output is shown in Figure 2.

```
In [4]: from dynpy.models.electric.motor import DCMotor

        dcmotor=DCMotor()
        ode = ODESystem(odes=dcmotor._eoms, dvars = Matrix([[dcmotor.i_w], [dcmotor.omega_s]]),
        ode_order=None)
        ode

Out[4]: 
$$\begin{bmatrix} -L_w \frac{d}{dt} i_w(t) - R_w i_w(t) + U_z - k_e \omega_s(t) \\ -B \omega_s(t) - J \frac{d}{dt} \omega_s(t) - M_l + k_m i_w(t) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ for } \begin{bmatrix} i_w(t) \\ \omega_s(t) \end{bmatrix} \text{ with } \{i_w(t) : 0.0, \omega_s(t) : 0.0\}$$

```

Figure 2. Definition and output of initialised `ODESystem` component.

The prepared object allows for solving the system of equations analytically using the solution method (Figure 3). By default, a matrix equation is returned, where the left-hand side represents dependent variables and the right-hand side represents a complete solution. An additional `as_eq_list` method was developed, which converts the original output into a list of equations of motion as *SymPy*'s `Eq` class type, so they can be easily reported in the $\text{\LaTeX}$ document. For clarity of view, a `display` command, being a part of the *IPython* module, was utilised to produce output, including an asterisk to unpack the list elements. The symbolic solution is very extensive, but substituting numerical values provides simplification (Figure 4).

```
In [6]: display(*ode.solution.as_eq_list())

Out[6]: 
$$i_w(t) = \frac{BU_z}{BR_w + k_e k_m} + C_1 e^{\left( \frac{-BL_w - JR_w + \sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w} \right) t} + C_2 e^{\left( \frac{-BL_w - JR_w - \sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w} \right) t} + \frac{M_l k_e}{BR_w + k_e k_m}$$


$$\omega_s(t) = \frac{C_1 (BL_w - JR_w - \sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m})}{2Jk_e} e^{\left( \frac{-BL_w - JR_w + \sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w} \right) t} + \frac{C_2 (BL_w - JR_w + \sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m})}{2Jk_e} e^{\left( \frac{-BL_w - JR_w - \sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w} \right) t} - \frac{M_l R_w}{BR_w + k_e k_m} + \frac{U_z k_m}{BR_w + k_e k_m}$$

```

Figure 3. Preview of analytical solution generation.

To generate a dataset of numerical values for a chosen time span, three additional methods are called on the solution. Firstly, `with_ics`, which takes a list of initial conditions (ICs) as an argument. Based on the provided ICs, integration constants are evaluated and substituted. Secondly, by utilizing the `subs` method, the system's symbolical param-

eters are replaced with numbers provided by a user in the form of a dictionary. Lastly, `compute_solution` is called on the predefined time span. Executed code results in a table of analytical solutions and their derivatives calculated for subsequent time stamps (Figure 5).

```
In [8]: data = {
        dcmotor.U_z: 0,
        dcmotor.R_w: 2,
        dcmotor.L_w: 0.1,
        dcmotor.k_e: 0.1,
        dcmotor.k_m: 0.1,
        dcmotor.J: 0.1,
        dcmotor.B: 0.5,
        dcmotor.M_obc: 0,
        }
        display(*ode.solution.subs(data).as_eq_list())
```

$$\text{Out[8]: } i_w(t) = C_1 e^{-19.9330343736593t} + C_2 e^{-5.06696562634074t}$$

$$\omega_s(t) = -0.0669656263407467 C_1 e^{-19.9330343736593t} - 14.9330343736593 C_2 e^{-5.06696562634074t}$$

Figure 4. Analytical solution with substituted parameter values.

```
In [9]: time_span=np.linspace(0,10,100)
        ode.solution.with_ics([100,3]).subs(data).compute_solution(t_span=time_span)
```

Out[9]:

	$i_w(t)$	$\omega_s(t)$	Derivative($i_w(t), t$)	Derivative($\omega_s(t), t$)
t				
0.00000	1.000000e+02	3.000000e+00	-8.608148e+02	1.918967e+01
0.10101	1.304901e+01	4.938351e+00	-4.872766e+02	1.877927e+00
0.20202	1.560275e+00	3.379379e+00	-6.410174e+01	-1.414086e+01
0.30303	9.916314e-02	2.081611e+00	-7.981756e+00	-1.051465e+01
0.40404	-5.220122e-02	1.255208e+00	-7.195325e-01	-6.574752e+00
...	...	...	...	...

Figure 5. Analytical solution values calculated for subsequent time stamps.

An approach to obtain a strictly numerical solution is presented in Figure 6. The substitution of parameter values is provided on equations of motion derived from the DCMotor class instance. Later, the resulting object is treated with a numerised method, which prepares the model for numerical computations. In the same line, `with_ics` is utilised to input initial conditions, and `compute_solution` with a defined time span solves the system, returning a dataset of results.

The presented methodology is well suited for analysing a system with a single set of chosen parameter values. The more sophisticated way of conducting simulations, which enables the possibility of analysing the changes in the system's behaviour for multiple values of a single parameter is provided by `NumericalAnalysisDataFrame` class. It includes the `from_model` method, which takes as arguments an object of `ODESystem` type, analysed parameter (symbol) and a list of its values, reference parameters values (a dictionary, excluding the parameter under investigation), and a list of dependent variables and an index of `DataFrame` to be generated (where `time_span` is a set of time stamps, and `dcmotor.ivar` is a symbolic definition of time as an independent variable derived from `DCMotor` class, with `ivar` being independent variable). Due to the necessity of providing multiple sets of initial conditions to properly initiate a `DataFrame`, the additional method `insert_ics` was developed so that a simple list of values can be introduced. Such a setup directly outputs a

result of numerical simulations (Figure 7) after executing `perform_simulations` command. To obtain numerical values calculated for the analytical solution, slight modifications have to be performed, as described in Figure 8. For all of the presented approaches, plots are easily generated by adding `.plot()` after `compute_solution` or `perform_simulations`, which can be additionally formatted to show chosen results with *pandas* library options (as a visible `droplevel` command, for example).

```
In [12]: ode = ODESystem(odes=dcmotor._eoms.subs(data), dvars = Matrix([[dcmotor.i_w], [dcmotor.omega_s]]), ode_order=1)
ode.numerized().with_ics([100,3]).compute_solution(t_span=time_span)
```

```
Out[12]: {}
{}
params case
{}
()
```

0.9646971225738525

	i_w(t)	omega_s(t)	Derivative(i_w(t), t)	Derivative(omega_s(t), t)
t				
0.00000	1.000000e+02	3.000000e+00	-860.803214	1.918889e+01
0.10101	1.305018e+01	4.938272e+00	-487.258431	1.876711e+00
0.20202	1.563953e+00	3.379134e+00	-64.103083	-1.414076e+01
0.30303	1.000626e-01	2.081552e+00	-7.999305	-1.051347e+01
0.40404	-5.206788e-02	1.255200e+00	-0.723916	-6.574464e+00
...	...	...	...	...

Figure 6. Numerical solution obtained by ODESystem with numerised method.

```
In [13]: data = {dcmotor.U_z: 0, dcmotor.L_w: 0.1, dcmotor.k_e: 0.1, dcmotor.k_m: 0.1, dcmotor.J: 0.1, dcmotor.B: 0.5, dcmotor.M_abc: 0, dcmotor.R_w: 1}
ode = ODESystem(odes=dcmotor._eoms, dvars = Matrix([[dcmotor.i_w], [dcmotor.omega_s]]), ode_order=1)
R_w_values=[1,2]
Z=[dcmotor.i_w, dcmotor.omega_s]
num_df= NumericalAnalysisDataFrame.from_model(ode,
                                                dcmotor.R_w,
                                                R_w_values,
                                                reference_data = data,
                                                coordinates = Z,
                                                index=pd.Index(time_span, name=dcmotor.ivan)
                                                ).insert_ics([100,3])
num_df.perform_simulations().droplevel([0], axis=1)
```

```
Out[13]: params case
{}
()
```

1.031759262084961

0.9487321376800537

	Eq(R_w, 1)		Eq(R_w, 2)	
	i_w(t)	omega_s(t)	i_w(t)	omega_s(t)
t				
0.00000	1.000000e+02	3.000000e+00	1.000000e+02	3.000000e+00
0.10101	3.605813e+01	6.579413e+00	1.305018e+01	4.938272e+00
0.20202	1.273867e+01	5.675824e+00	1.563953e+00	3.379134e+00
0.30303	4.340684e+00	4.018798e+00	1.000626e-01	2.081552e+00
0.40404	1.380077e+00	2.621939e+00	-5.206788e-02	1.255200e+00
...	...	...	...	...

Figure 7. Numerical solutions for multiple values of analysed parameters obtained by NumericalAnalysisDataFrame.

```

In [14]: data = {dcmotor.U_z: 0, dcmotor.L_w: 0.1, dcmotor.k_e: 0.1, dcmotor.k_m: 0.1, dcmotor.J: 0.1, dcmotor.B: 0.5, dcmotor.M_obc: 0}
ode = ODESystem(odes=dcmotor._eoms, dvars = Matrix{[[dcmotor.i_w], [dcmotor.omega_s]]}, ode_order=1)
R_w_values=[1,2]
Z=[dcmotor.i_w, dcmotor.omega_s]
num_df= NumericalAnalysisDataFrame.from_model(ode.subs(data).solution.with_ics([100,3]),
                                              dcmotor.R_w,
                                              R_w_values,
                                              reference_data = data,
                                              coordinates = Z,
                                              index=pd.Index(time_span, name=dcmotor.i_var)
                                              )

num_df.perform_simulations().droplevel([0],axis=1)

Out[14]: 0.00848841667175293
         0.007567644119262695

```

	Eq(R_w, 1)		Eq(R_w, 2)	
	i_w(t)	omega_s(t)	i_w(t)	omega_s(t)
t				
0.00000	1.000000e+02	3.000000e+00	1.000000e+02	3.000000e+00
0.10101	3.605363e+01	6.580331e+00	1.304901e+01	4.938351e+00
0.20202	1.273593e+01	5.676397e+00	1.560275e+00	3.379379e+00
0.30303	4.338814e+00	4.019198e+00	9.916314e-02	2.081611e+00
0.40404	1.378492e+00	2.622263e+00	-5.220122e-02	1.255208e+00
...	...	...	...	...

Figure 8. Analytical solutions for multiple values of analysed parameters obtained by Numerical-AnalysisDataFrame.

3. Theory and Calculation

3.1. DC Motor—Tested System

The accuracy and effectiveness of the developed *DynPy* library in the aspect of solving systems of ODEs and conducting simulations were investigated with a model of a direct current electric motor described by a system of two differential equations.

Based on the equivalent circuit (Figure 9) and Kirchhoff's second law, the electrical equation of the motor was derived:

$$U_z = E + U_{Lw} + U_{Rw} \quad (1)$$

where:

U_{Lw} —voltage related to the rotor inductance;

E —electro-motive force of induction;

U_z —voltage supplying the rotor;

U_{Rw} —voltage across the rotor winding resistance.

The voltage across the resistance of the rotor windings is proportional to the current flowing through them:

$$U_{Rw} = R_w i_w \quad (2)$$

where the following are used:

i_w —rotor winding current;

R_w —equivalent resistance of the rotor windings.

The voltage across the rotor winding inductance is proportional to the rate of change in the current flowing through it (magnetic circuit losses are neglected):

$$U_{Lw} = L_w \dot{i}_w \quad (3)$$

where the following are used:

L_w —equivalent inductance of the rotor windings;

$\dot{i}_w$ —rotor winding current rate of change.

When the rotor rotates, a counter-electro-motive force (EMF) is induced in its windings, the value of which is proportional to the angular velocity of the rotor:

$$E = k_e \omega_s \quad (4)$$

where the following are used:

ω_s —angular velocity of the rotor;

k_e —back electro-motive force constant.

By substituting each component into Equation (1), we obtain

$$U_z = L_w \dot{i}_w + R_w i_w + k_e \omega_s \quad (5)$$

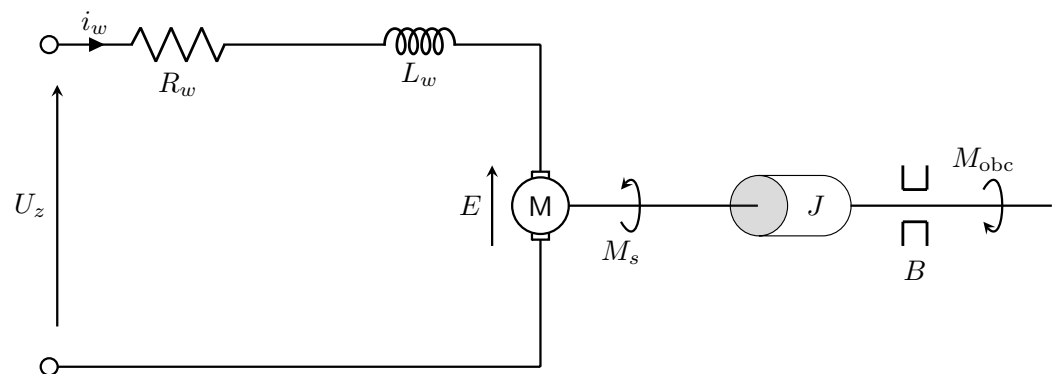


Figure 9. The equivalent circuit diagram of the analysed DC motor system.

The rotor torque has to overcome torques opposing the movement and is proportional to the current flowing through it. Assuming that the magnetic flux of the stator remains constant, two formulas were derived:

$$M_s = M_a + M_l + M_r \quad (6)$$

$$M_s = k_m i_w \quad (7)$$

where the following are used:

M_s —rotor torque;

M_l —engine load torque;

k_m —torque constant;

M_r —rotor motion resistance torque;

M_a —rotor angular acceleration torque.

The torque values associated with the angular acceleration and the rotor's motion can be expressed as

$$M_a = J \dot{\omega}_s \quad (8)$$

$$M_r = B \omega_s \quad (9)$$

where the following are used:

B —coefficient of viscous friction reduced to the rotor shaft;

J —moment of inertia reduced to the rotor shaft;

$\dot{\omega}_s$ —angular acceleration of the rotor.

Substituting the individual components described in Formulas (8) and (9) into Equation (6) we obtain

$$k_m i_w = M_l + B\omega_s + J\dot{\omega}_s \quad (10)$$

As a result of the performed steps, equations of motion of the system were obtained:

$$-U_z + L_w \dot{i}_w + R_w i_w + k_e \omega_s = 0 \quad (11)$$

$$M_l + B\omega_s + J\dot{\omega}_s - k_m i_w = 0 \quad (12)$$

The proposed model was solved with two techniques—with a hybrid approach, which utilises both analytical and numerical solutions at once (HAN), and strictly analytically, with forcing input as the Heaviside function. The HAN solution was completed in two steps. The system was first solved analytically for constant external forcing with the use of *DynPy* library (the solution was not calculated manually). Later, the obtained solution was updated numerically by alternately introducing dictionaries with parameter values filled in such a manner, so a step character of forcing could be imitated. As input functions in the model, the rotor supply voltage (U_z) and the motor load torque (M_l) were considered:

$$U_z(t) = \begin{cases} 10 & \text{for } t = \{(0, 2) \cup (4, 6) \cup (8, 10)\} \\ 0 & \text{for } t = \{(2, 4) \cup (6, 8)\} \end{cases} \quad (13)$$

$$M_l(t) = \begin{cases} 0.2 & \text{for } t = \{(1, 3) \cup (5, 7) \cup (9, 10)\} \\ 0 & \text{for } t = \{(0, 1) \cup (3, 5) \cup (7, 9)\} \end{cases} \quad (14)$$

Both are represented by step functions described in Equations (13) and (14), with graphical interpretation shown in the plots (Figures 10 and 11). A detailed view of selected mathematical dependencies of the analysed system derived from the DCMotor class, and the full form of the analytical solution are defined in Appendix C.

The equations of motion for a system with Heaviside step function forcing have the following form:

$$L_w \dot{i}_w + R_w i_w + k_e \omega_s - U_z \theta \left(\sin \left(\frac{2\pi t}{T} \right) \right) = 0 \quad (15)$$

$$B\omega_s + J\dot{\omega}_s + M_l \theta \left(-\cos \left(\frac{2\pi t}{T} \right) \right) - k_m i_w = 0 \quad (16)$$

where the following are used:

- θ —Heaviside function;
- T —cycle length.

The step function takes a sinewave as an argument, to which additional conditions were provided for proper representation of a demanded character of input voltage and motor load. The analytical solution of Equations (15) and (16) is provided in Appendix C, while the code snippet with the implementation of the HAN method is in Appendix D.

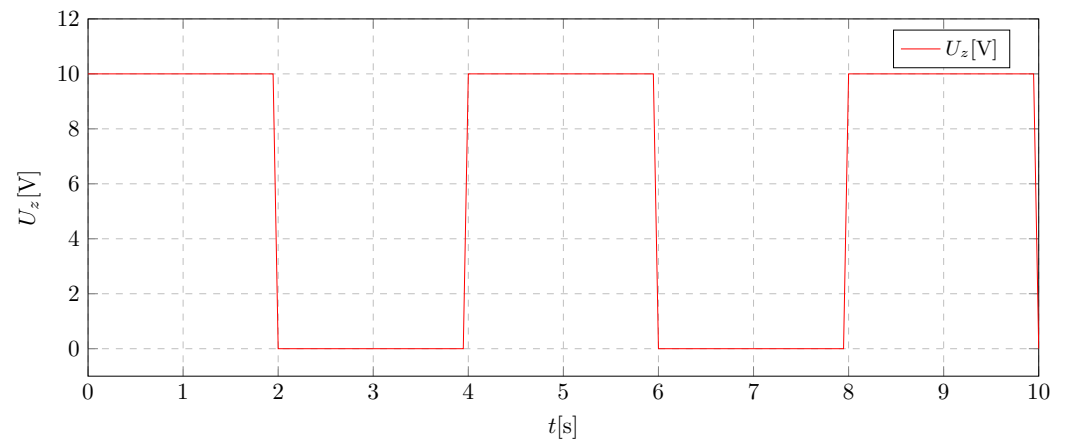


Figure 10. Rotor supply voltage.

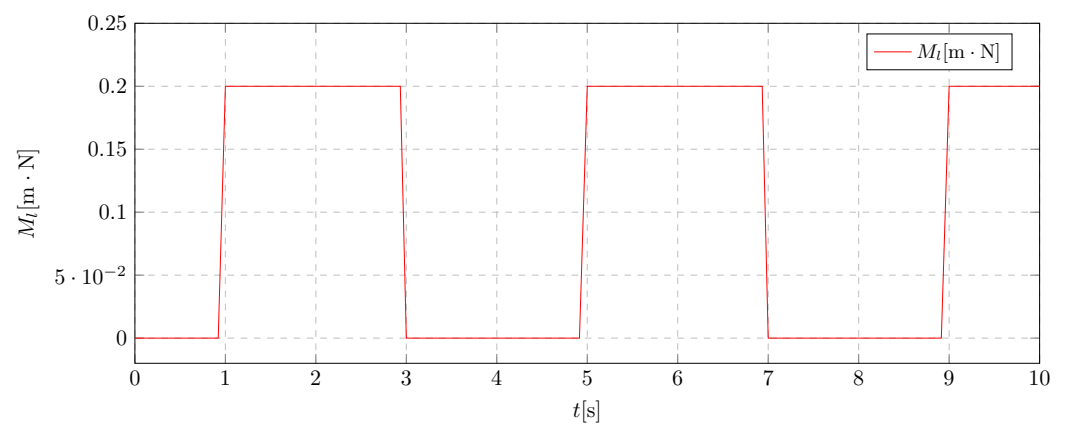


Figure 11. Motor load torque.

3.2. DC Motor—Conducting Simulations Using the DynPy Library

The parameter values of the model were selected and presented in Table 1. Results of simulations for armature current and rotor angular velocity obtained with the HAN method are provided in Figures 12 and 13, while plots presented in Figures 14 and 15 apply to the fully analytical solution.

Table 1. Parameter values adopted for calculations.

Parameter	Unit	Value
R_w	Ω	2.0
L_w	H	0.1
k_e	$\frac{\text{Vs}}{\text{rad}}$	0.1
k_m	$\frac{\text{Nm}}{\text{A}}$	0.1
J	$\frac{\text{kgm}^2}{\text{s}^2}$	0.1
B	$\frac{\text{Nms}}{\text{rad}}$	0.5
T	s	4

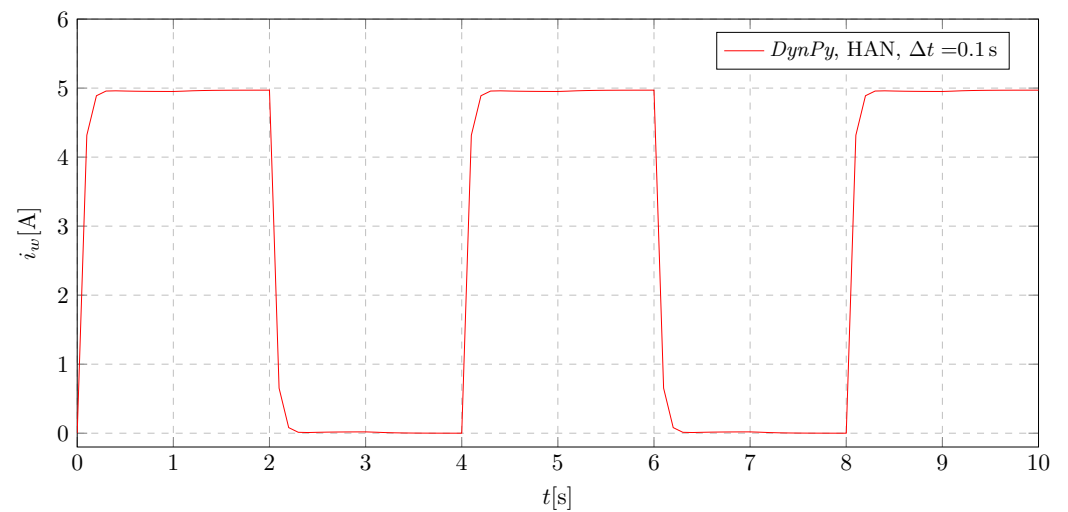


Figure 12. Armature current—simulation results obtained from *DynPy* library (HAN, fixed step size 0.1 s).

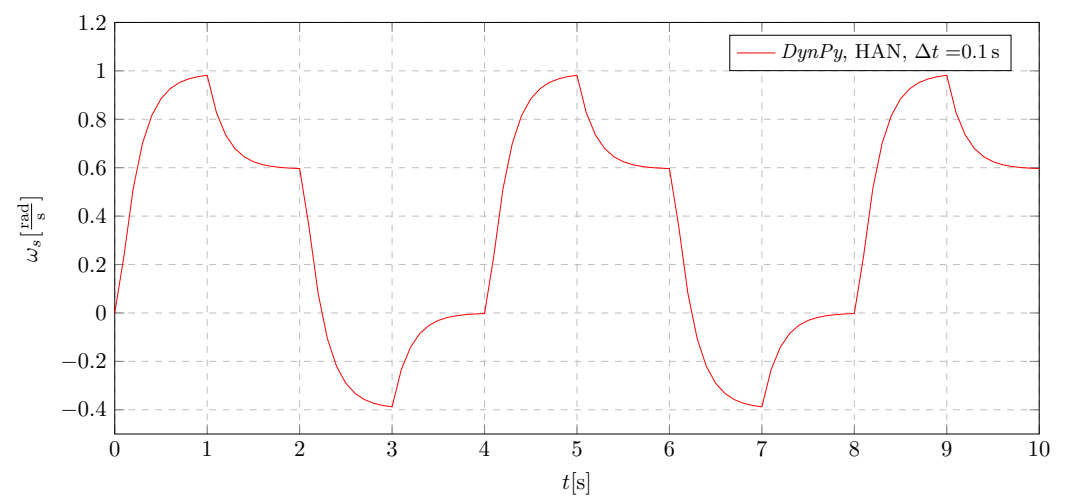


Figure 13. Rotor's angular velocity—simulation results obtained from *DynPy* library (HAN, fixed step size 0.1 s).

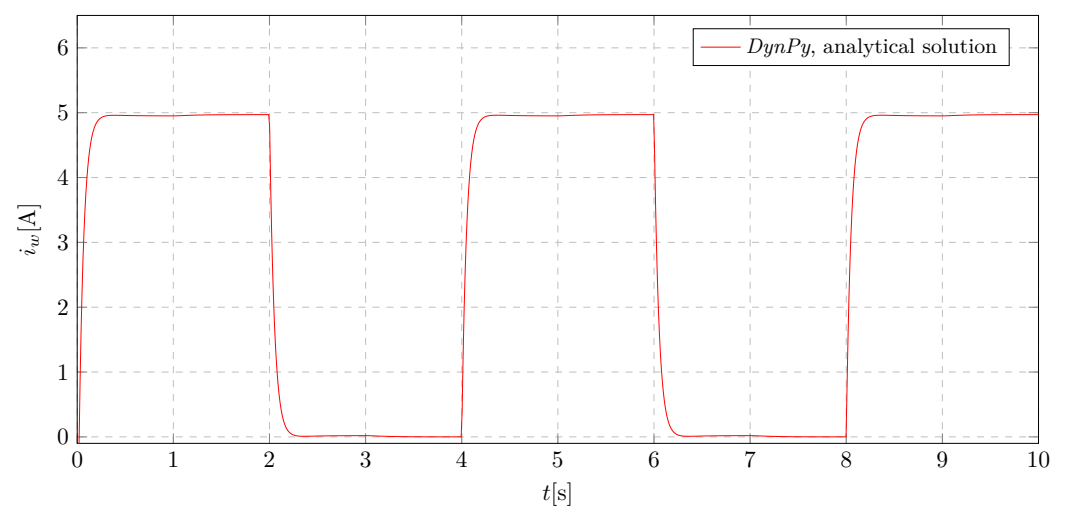


Figure 14. Armature current—simulation results obtained from *DynPy* with fully analytical solution.

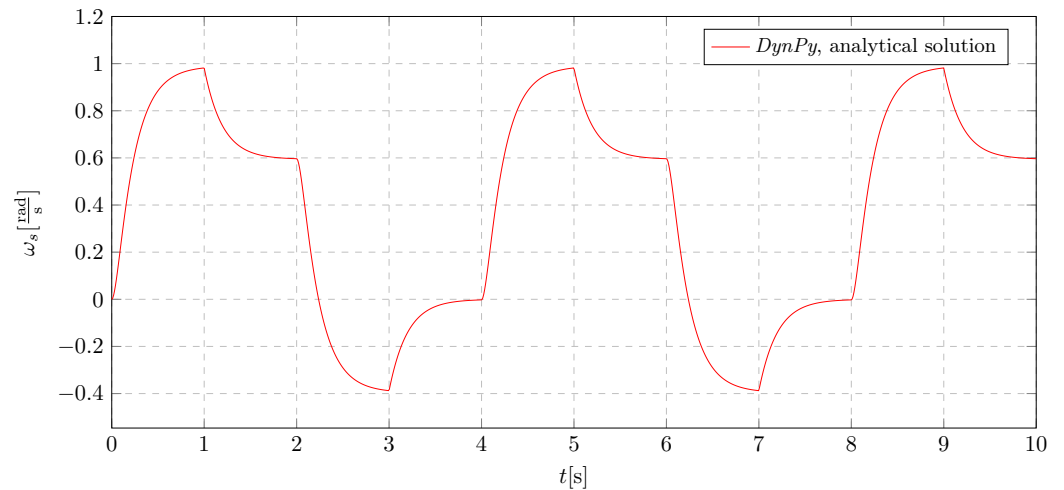


Figure 15. Rotor's angular velocity—simulation results obtained from *DynPy* with fully analytical solution.

In Figures 16 and 17, corresponding plots were juxtaposed. The fully analytical solution is visibly smoother in areas where rapid changes in analysed values occur (selected time stamps of 1 s interval are presented in Table 2). The main reason behind this might be identified as a much bigger number of samples utilised to generate a plot (one thousand time stamps), but the aspect of a fully analytical solution being more accurate than HAN, in general, cannot be omitted and will be analysed in future works. Nonetheless, it is safe to assume that even if the HAN approach with reduced step size would result in results indistinguishable from the analytical solution, it will still require more preparation, resulting in higher overall time cost. Since the HAN solution provides the possibility to compare the behaviour of the numerical solution regarding changes in step size and a comparison with commercial software in this aspect, it was selected for further consideration in the study. The integration step is 0.1 s, and selected time stamps of 1 s interval are presented in Table 3.

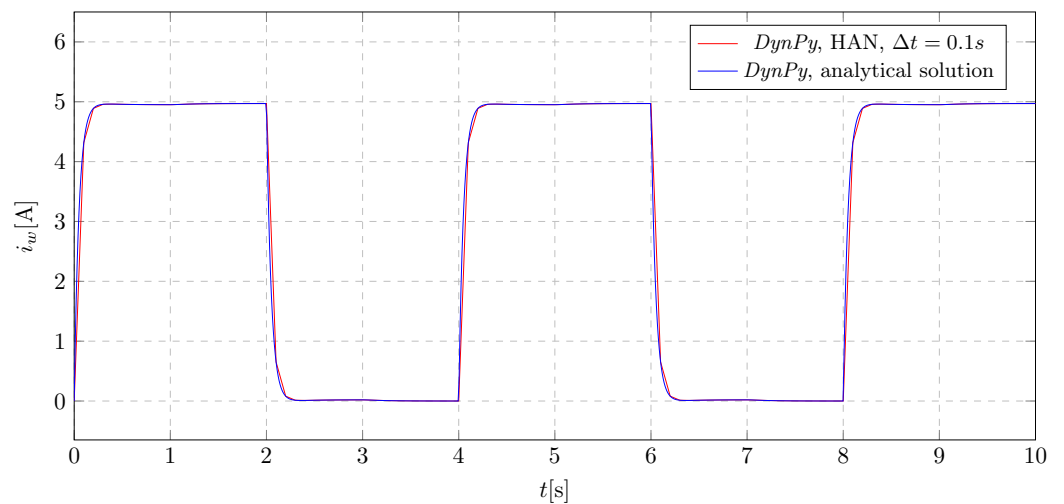


Figure 16. *DynPy*—comparison of results for armature current obtained with hybrid analytical–numerical and fully analytical solution.

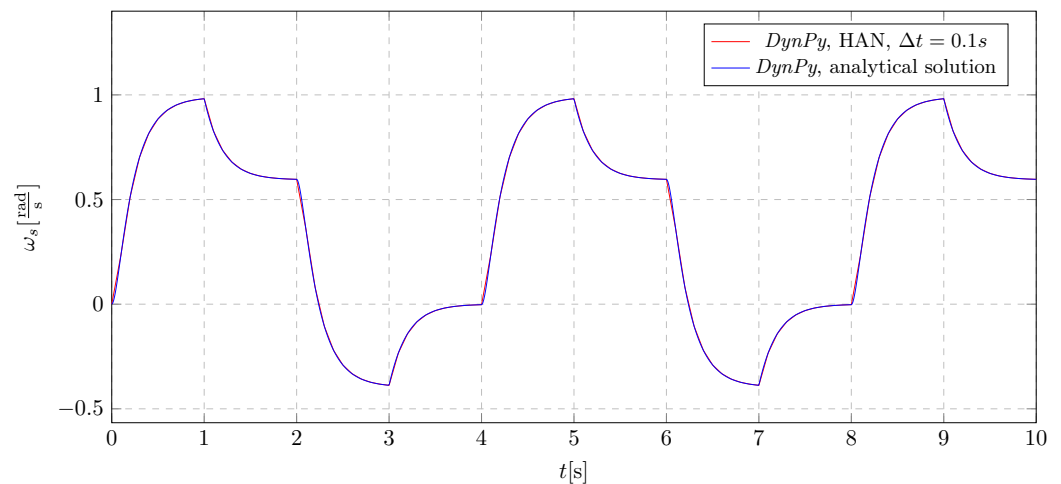


Figure 17. *DynPy*—comparison of results for angular velocity of the rotor obtained with hybrid analytical–numerical and fully analytical solution.

Table 2. Results of the simulation conducted using the *DynPy* library—fully analytical method.

$t[s]$	$i_w[A]$	$\omega_s[\frac{rad}{s}]$
0.000000	0.000000	−0.000000
1.000000	4.951052	0.979779
2.000000	4.754254	0.596283
3.000000	−0.000552	−0.381824
4.000000	0.384783	−0.001621
5.000000	4.951041	0.972042
6.000000	4.384571	0.594716
7.000000	−0.000541	−0.374180
8.000000	0.739952	0.000653
9.000000	4.951030	0.964474
10.000000	4.950499	0.596505

Table 3. Results of the simulation conducted using the *DynPy* library—HAN method.

$t[s]$	$i_w[A]$	$\omega_s[\frac{rad}{s}]$
0.000000	−0.000000	0.000000
1.000000	4.951055	0.981733
2.000000	4.970133	0.596505
3.000000	0.019241	−0.387659
4.000000	0.000164	−0.002446
5.000000	4.951056	0.981718
6.000000	4.970133	0.596505
7.000000	0.019241	−0.387659
8.000000	0.000164	−0.002446
9.000000	4.951056	0.981718
10.000000	4.970133	0.596505

3.3. DC Motor—Conducting Simulations Using MATLAB/Simulink

MATLAB/Simulink, as a powerful tool for the simulation and modelling of dynamic systems, has found wide application in various engineering fields. In the context of this paper, it was used as a reference tool for comparison with the results obtained from the *DynPy* library, since its widespread use in the fields of control systems, drive systems, automation, and many other areas of engineering makes conducted simulations a reliable reference point.

Reference simulations were carried out using both variable and fixed step solvers.

This approach enables a detailed analysis of the impact of different integration strategies on simulation results and allows for a direct comparison with the *Python* implementation, as the *DynPy* library HAN solution utilises a fixed step approach. The designed model of the DC motor used for conducting simulations in the *MATLAB/Simulink* environment is depicted in Figure 18.

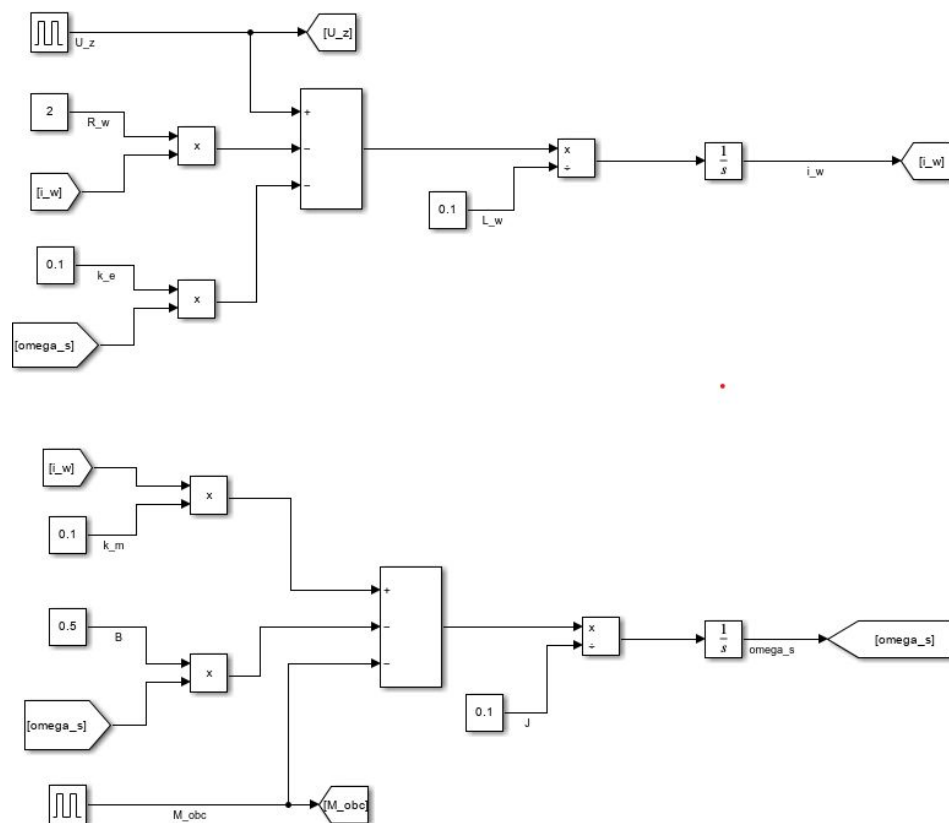


Figure 18. Block diagram representing a DC motor designed in *MATLAB/Simulink*

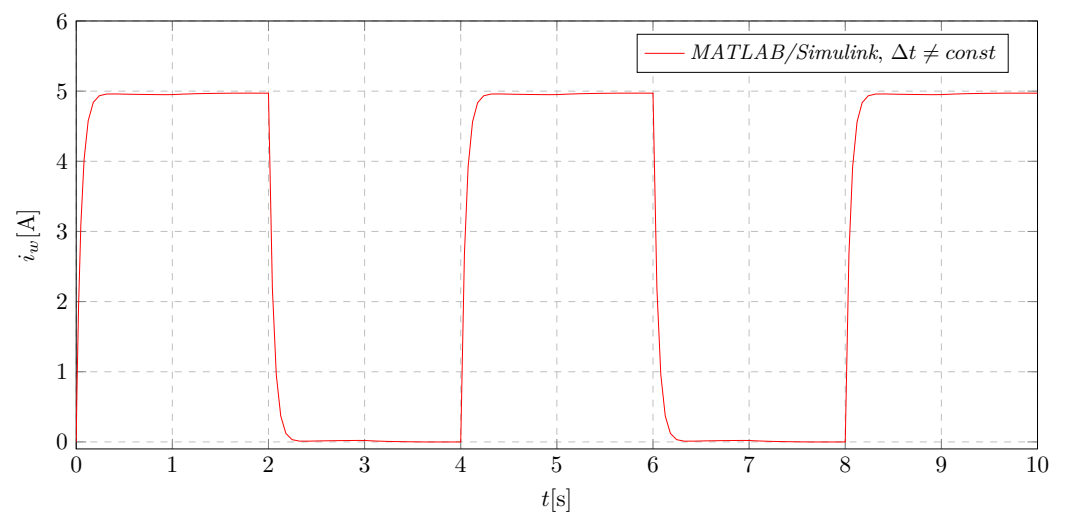
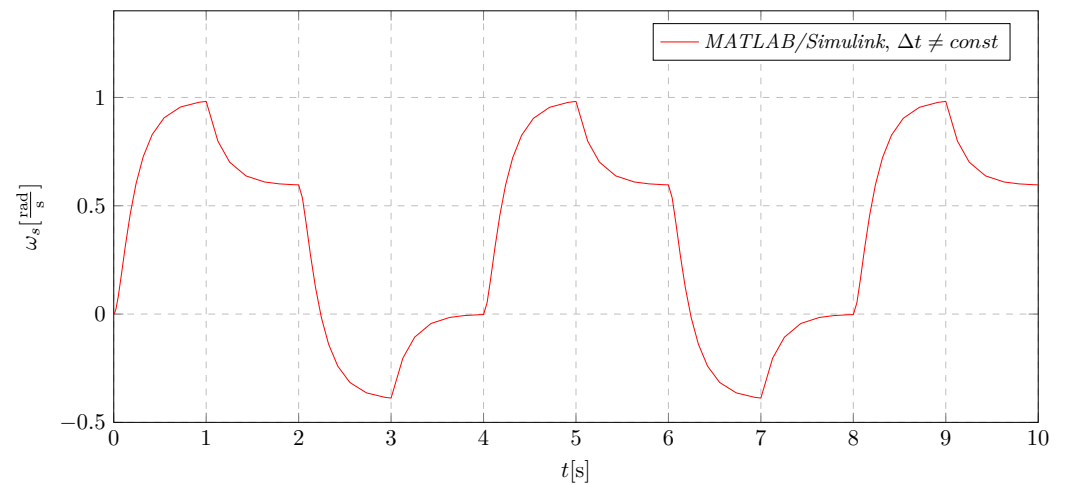
The results obtained using the variable integration step are presented in Table 4 (Figures 19 and 20), while those with a fixed step set to be the same as in the simulations using the *DynPy* library (0.1 s) are in in Table 5 (Figures 21 and 22).

Table 4. Results of the simulation conducted with variable integration step in *MATLAB/Simulink*.

$t[s]$	$i_w[A]$	$\omega_s[\frac{rad}{s}]$
0.000000	0.000000	0.000000
1.000000	4.950632	0.981745
2.000000	4.970501	0.596487
3.000000	0.019819	−0.387680
4.000000	−0.000201	−0.002427
5.000000	4.950692	0.981726
6.000000	4.970500	0.596487
7.000000	0.019819	−0.387680
8.000000	−0.000201	−0.002427
9.000000	4.950692	0.981726
10.000000	4.970500	0.596487

Table 5. Results of the simulation conducted with a constant integration step of 0.1 s in MATLAB/Simulink.

$t[s]$	$i_w[A]$	$\omega_s[\frac{rad}{s}]$
0.000000	0.000000	0.000000
1.000000	4.950963	0.982077
2.000000	4.970140	0.596408
3.000000	0.019333	−0.388004
4.000000	0.000157	−0.002349
5.000000	4.950964	0.982063
6.000000	4.970140	0.596408
7.000000	0.019333	−0.388004
8.000000	0.000157	−0.002349
9.000000	4.950964	0.982063
10.000000	4.970140	0.596408

**Figure 19.** Armature current—simulation results in MATLAB/Simulink with variable integration step.**Figure 20.** Angular velocity of the rotor—simulation results in MATLAB/Simulink with variable integration step.

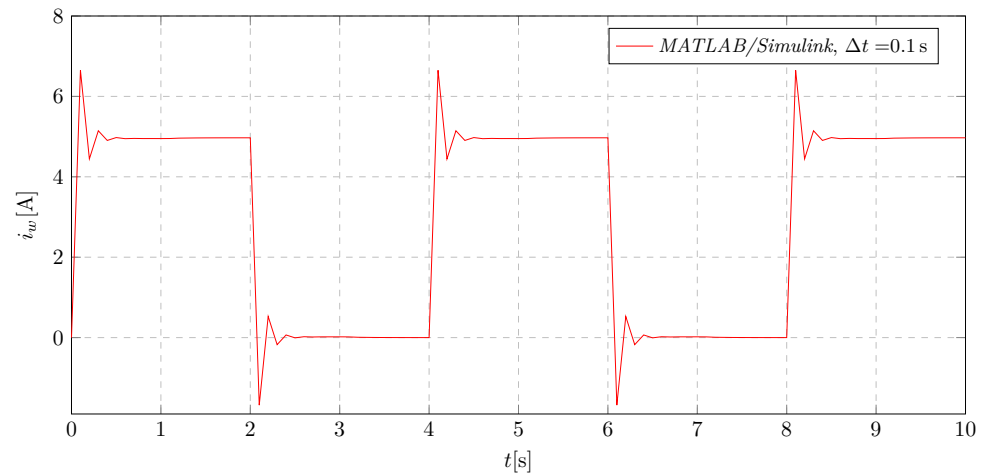


Figure 21. Armature current—simulation results in MATLAB/Simulink with a specified constant integration step of 0.1 s.

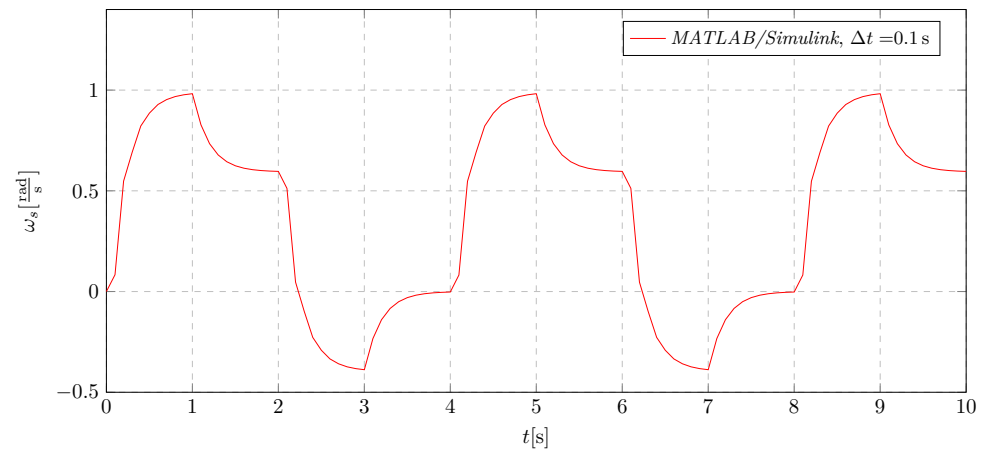


Figure 22. Angular velocity of the rotor—simulation results in MATLAB/Simulink with a specified constant integration step of 0.1 s.

After initial analysis of the simulation results obtained with variable and constant integration steps, significant differences were noticeable both in the achieved values and in the shape of the current intensity in the rotor windings and the rotor angular velocity. Therefore, a decision was made to reduce the constant integration step to a value of 0.01 s and conduct further simulations. In Figures 23 and 24 are the results obtained after implementing this modification. Values of selected time stamps are listed in Table 6.

Table 6. Results of the simulation conducted with a specified constant integration step of 0.01 s in MATLAB/Simulink.

$t[s]$	$i_w[A]$	$\omega_s[\frac{rad}{s}]$
0.000000	0.000000	0.000000
1.000000	4.951055	0.981734
2.000000	4.970133	0.596505
3.000000	0.019241	−0.387659
4.000000	0.000164	−0.002446
5.000000	4.951056	0.981718
6.000000	4.970133	0.596505
7.000000	0.019241	−0.387659
8.000000	0.000164	−0.002446
9.000000	4.951056	0.981718
10.000000	4.970133	0.596505

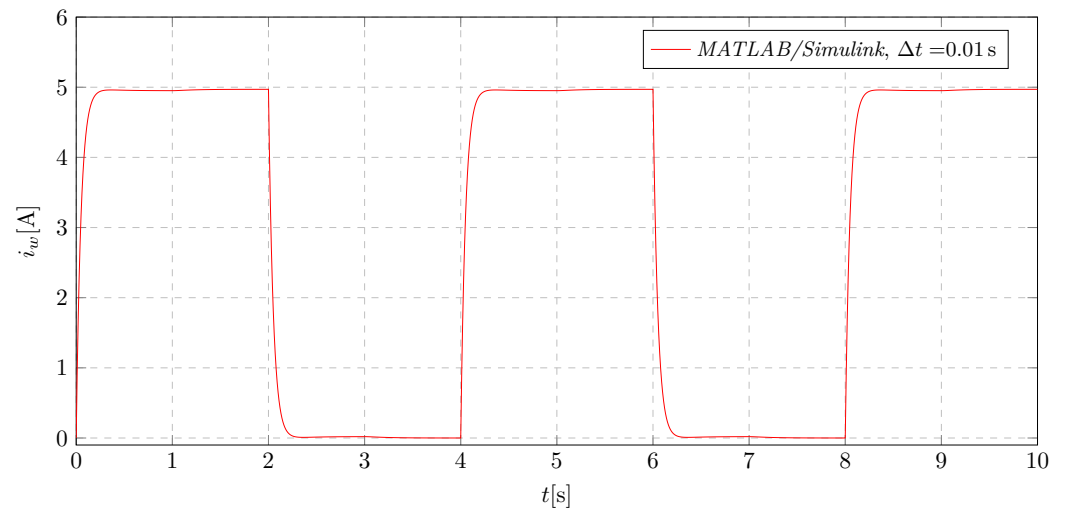


Figure 23. Armature current—simulation results in MATLAB/Simulink with a specified constant integration step of 0.01 s.

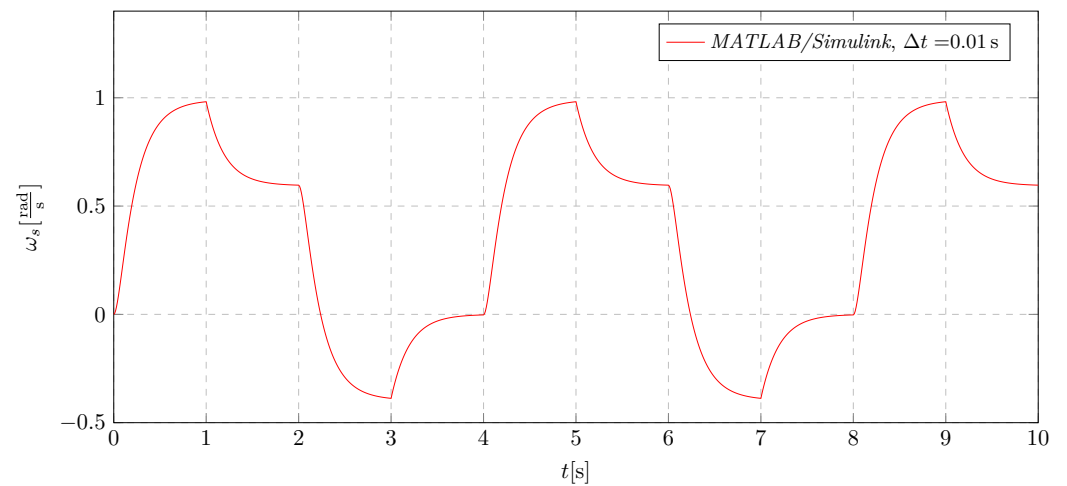


Figure 24. Angular velocity of the rotor—simulation results in MATLAB/Simulink with a specified constant integration step of 0.01 s.

3.4. Application of DynPy for Electrical Circuits

DynPy library enables to modelling of electric circuits in a similar way as it handles electro-mechanical and mechanical systems. Electrical modelling was performed based on a serial DC RLC circuit (Figure 25). A single-degree-of-freedom, second-order system was introduced, in which the Heaviside function provides the variable value of the resistance. Such a definition of equations of motion introduces a more challenging problem, as the nonconstant parameter directly affects the behaviour of the first derivative of the dependent variable, while in the case of the DC motor system, the forcing vector was subjected to periodic changes:

$$-U + L\ddot{q} + R_{min}\dot{q} + \frac{q}{C} + R\theta\left(\sin\left(\frac{2\pi t}{T}\right)\right)\dot{q} = 0 \quad (17)$$

where the following are used:

- R —amplitude of the variable part of the resistance;
- R_{min} —constant value of the resistance;
- L —inductance;
- C —capacitance;
- U —source voltage;

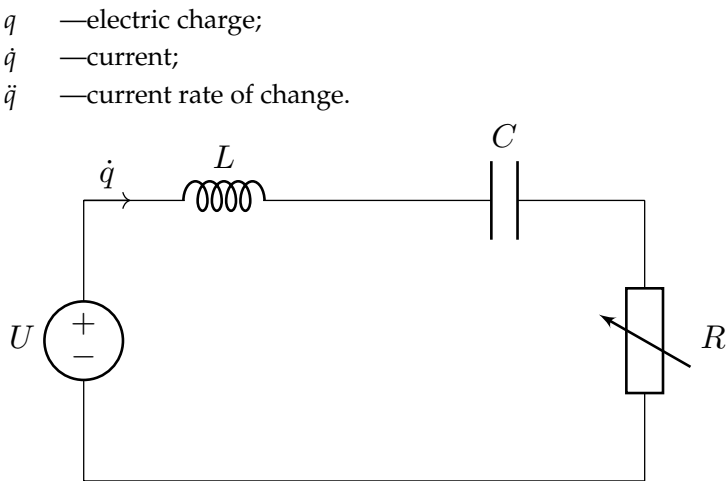


Figure 25. Schematic representation of the analysed RLC circuit.

The simulational model adopts the parameters for modulation control of the current in a power circuit. Values of the system’s parameters are provided in Table 7. Parameters were selected in a way that allows for presenting analysed phenomena in an understandable form. The period was selected arbitrarily, then other parameters were calculated to ensure decay times lower than the period. Under these conditions, the circuit can operate as a current control system that can be utilised to simulate electronic commutation processes or polarisation changes in direct current machines. The resistance signal is shown in Figure 26 and described with the following formula:

$$R(t) = \begin{cases} 0.1 & \text{for } t = (0,2) \cup (4,6) \cup (8,10) \\ 20 & \text{for } t = (2,4) \cup (6,8) \end{cases}$$

(18)

Table 7. Parameter values adopted for simulations of RLC circuit.

Parameter	Unit	Value
U	V	10.0
R	Ω	19.9
R_{min}	Ω	0.1
L	H	1
C	F	100
T	s	4

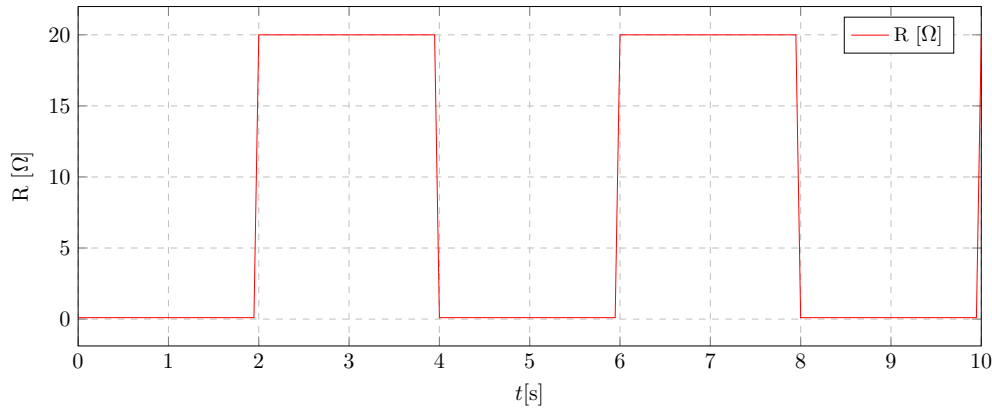


Figure 26. Resistance signal.

To solve the equation, the HAN method was utilised in a similar way as in the case of the DC motor. First, the solution (19) and its first derivative representing current (20) for a system with stationary resistance was returned from an instance of ODEs system class:

$$q = C_1 e^{t\left(-\frac{R}{2L} - \frac{\sqrt{CR^2 - 4L}}{2\sqrt{CL}}\right)} + C_2 e^{t\left(-\frac{R}{2L} + \frac{\sqrt{CR^2 - 4L}}{2\sqrt{CL}}\right)} - \frac{2CLU}{\sqrt{CR}\sqrt{CR^2 - 4L} + CR^2 - 4L} + \frac{2CLU}{\sqrt{CR}\sqrt{CR^2 - 4L} - CR^2 + 4L} \quad (19)$$

$$\begin{aligned} \dot{q} = & C_1 \left(-\frac{R}{2L} - \frac{\sqrt{CR^2 - 4L}}{2\sqrt{CL}} \right) e^{t\left(-\frac{R}{2L} - \frac{\sqrt{CR^2 - 4L}}{2\sqrt{CL}}\right)} \\ & + C_2 \left(-\frac{R}{2L} + \frac{\sqrt{CR^2 - 4L}}{2\sqrt{CL}} \right) e^{t\left(-\frac{R}{2L} + \frac{\sqrt{CR^2 - 4L}}{2\sqrt{CL}}\right)} + \frac{CRU}{\sqrt{CR}\sqrt{CR^2 - 4L} + CR^2 - 4L} \\ & + \frac{\sqrt{CU}\sqrt{CR^2 - 4L}}{\sqrt{CR}\sqrt{CR^2 - 4L} + CR^2 - 4L} + \frac{\sqrt{CU}\sqrt{CR^2 - 4L}}{\sqrt{CR}\sqrt{CR^2 - 4L} - CR^2 + 4L} \\ & - \frac{CRU}{\sqrt{CR}\sqrt{CR^2 - 4L} - CR^2 + 4L} \end{aligned} \quad (20)$$

Formulas (19) and (20) were later introduced into a slightly modified (inputted system, data dictionaries, and number of time stamps) `han_solution_loop` function, presented in Appendix D. The HAN method was used twice to obtain results for 0.1 s and 0.01 s time steps. Simultaneously, the system was created in the *MATLAB/Simulink* environment (Figure 27) and solved there separately with variable, 0.1 s, and 0.01 s step sizes.

The considered case underwent the entire process of electrical circuit modelling and its further analysis. The *MATLAB/Simulink* environment was used again as a reference tool for the assessment of the performance of the demonstrated library. A detailed view of the comparison between RLC circuit results generated with *DynPy* and *MATLAB/Simulink* is provided in Subsection 4.2. The results presented there confirm the outcomes obtained previously with the analysis of the DC motor.

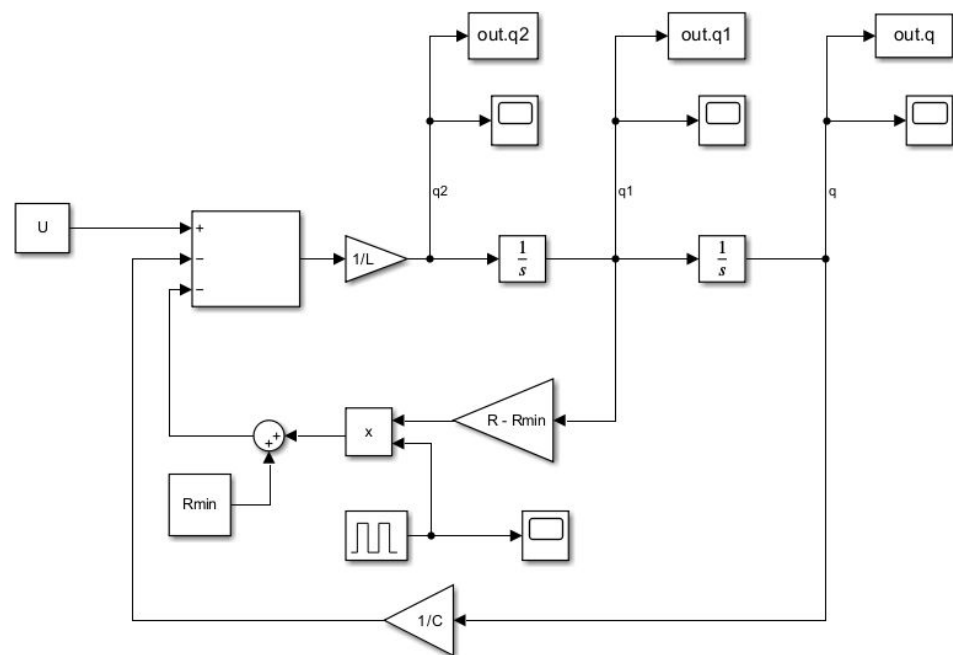


Figure 27. Block diagram representing an RLC circuit designed in extitMATLAB/Simulink.

4. Results and Discussion

4.1. DC Motor

The results of computer simulations were subjected to comparative analysis to assess the effectiveness of the developed library. The listed plots and tables show a comparison of the armature current and rotor angular velocity of the DC motor obtained using *DynPy* (with a constant integration step size of 0.1 s) and the following:

- *MATLAB/Simulink* with a variable integration step size (Figure 28, Table 8);
- *MATLAB/Simulink* with a constant step of 0.1 s (Figure 29, Table 9);
- *MATLAB/Simulink* with a constant step of 0.01 s (Figure 30, Table 10).

The simulation results obtained using the *DynPy* library with a constant integration step size of 0.1 s and in *MATLAB/Simulink* using a variable integration step size confirm the agreement between the results outputted from both tools. To present the compatibility of the results more clearly, the characteristics of both simulations are overlaid in (Figure 28). Only minor differences are noticeable in the rotor current plots in areas of dynamic changes in characteristics. The results of the simulation conducted in *MATLAB/Simulink* are more precise, and the curve representing them is smoother. These differences result from the use of a variable integration step size in the simulation using *MATLAB/Simulink*. This approach allows for adjusting the integration step size depending on the rate of change in the function, resulting in better accuracy of results in areas of dynamic changes. In the case of rotor angular velocity characteristics, there are no significant differences—the plots are almost identical.

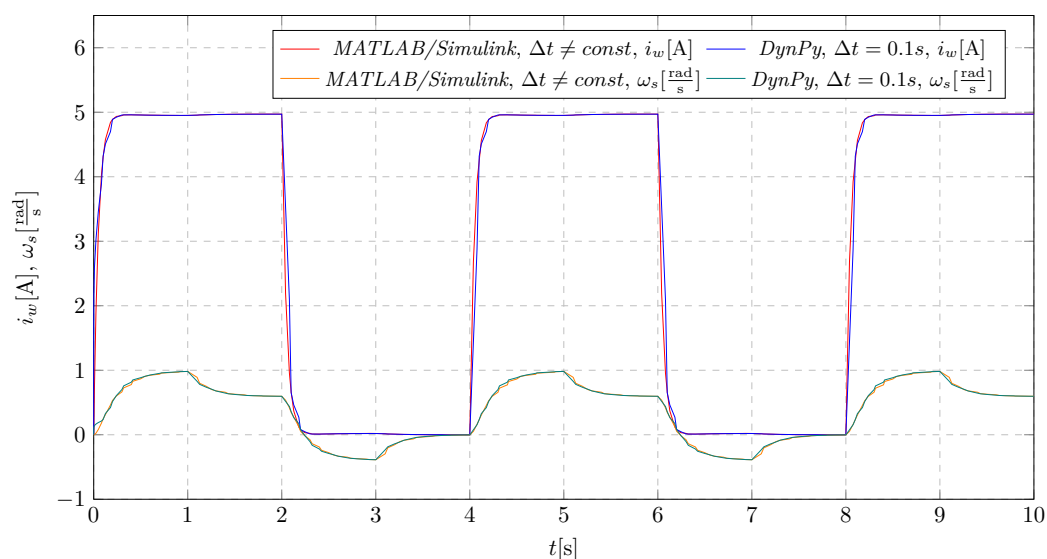
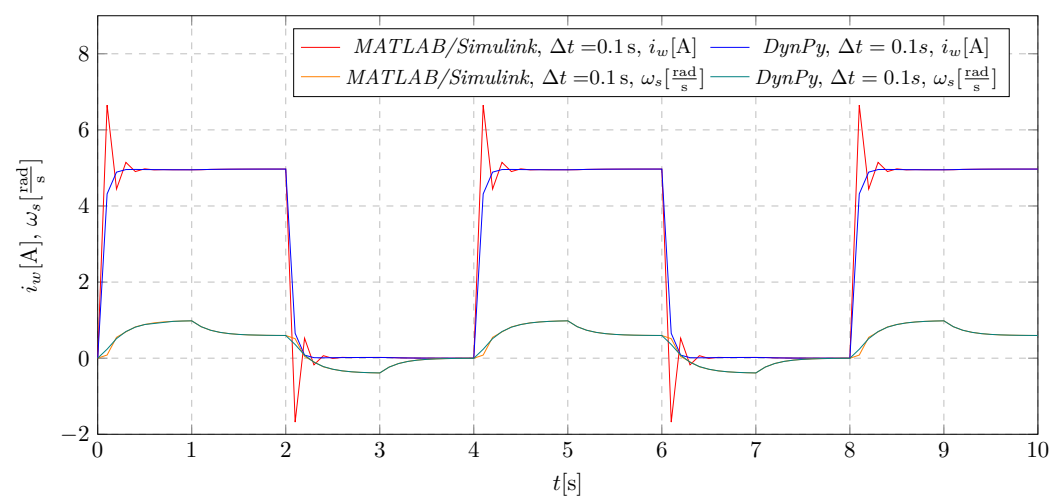


Figure 28. Comparison of simulations conducted using the *DynPy* library with a fixed integration step size of 0.1 s and *MATLAB/Simulink* with a variable integration step size.

Next, the simulation results were compared, where in both software, a constant integration step of 0.1 s was specified (Figure 29). Significant differences in the characteristics of rotor current intensity are visible. In *MATLAB/Simulink*, the waveform of the rotor current intensity undergoes oscillations in regions of rapid changes in its value, while the corresponding plot generated using the *DynPy* library remains stable. In the same intervals as the rotor current intensity, there are discrepancies in the characteristics of angular velocity. However, they are less noticeable, and their differences are acceptable. The disagreement in the results may stem from various factors such as the numerical implementation of the algorithm in both environments or the handling of numerical errors.

Table 8. DC motor: comparison of results for chosen time stamps—*MATLAB/Simulink* with variable step and HAN method with 0.1 s step.

$t[s]$	<i>MATLAB</i> , $\Delta t \neq const$		<i>DynPy</i> HAN, $\Delta t = 0.1\text{ s}$	
	$i_w[A]$	$\omega_s[\frac{rad}{s}]$	$i_w[A]$	$\omega_s[\frac{rad}{s}]$
0.100000	4.303243	0.246351	4.317466	0.236242
1.100000	4.954424	0.889913	4.955557	0.827837
2.100000	0.662744	0.345930	0.652732	0.359291
3.100000	0.015932	−0.295513	0.014739	−0.233768
4.100000	4.242498	0.233902	4.317565	0.234769
5.100000	4.954449	0.890017	4.955558	0.827828
6.100000	0.662744	0.345930	0.652732	0.359291
7.100000	0.015932	−0.295513	0.014739	−0.233768
8.100000	4.242498	0.233902	4.317565	0.234769
9.100000	4.954449	0.890017	4.955558	0.827828

**Figure 29.** DC motor - comparison of simulations conducted using the *DynPy* library with a fixed integration step size of 0.1 s and *MATLAB/Simulink* with a fixed integration step size of 0.1 s.**Table 9.** DC motor: comparison of results for chosen time stamps—*MATLAB/Simulink* with 0.1 s step and HAN method 0.1 s step.

$t[s]$	<i>MATLAB</i> , $\Delta t = 0.1\text{ s}$		<i>DynPy</i> HAN, $\Delta t = 0.1\text{ s}$	
	$i_w[A]$	$\omega_s[\frac{rad}{s}]$	$i_w[A]$	$\omega_s[\frac{rad}{s}]$
0.100000	6.650000	0.083333	4.317466	0.236242
1.100000	4.952507	0.827282	4.955557	0.827837
2.100000	−1.679797	0.512135	0.652732	0.359291
3.100000	0.017789	−0.233214	0.014739	−0.233768
4.100000	6.650094	0.081924	4.317565	0.234769
5.100000	4.952508	0.827273	4.955558	0.827828
6.100000	−1.679797	0.512135	0.652732	0.359291
7.100000	0.017789	−0.233214	0.014739	−0.233768
8.100000	6.650094	0.081924	4.317565	0.234769
9.100000	4.952508	0.827273	4.955558	0.827828

Due to discrepancies in the simulation results with the specified constant integration step of 0.1 s, it was decided to reduce its value to 0.01 s in the case of *MATLAB/Simulink* software and conduct further simulations. The juxtaposed characteristics confirm the consistency of the simulation results, and for better presentation, the characteristics are overlaid (Figure 30). The graph is very similar to the plot in Figure 28, with only minor,

negligible differences in the values of rotor current intensity and rotor angular velocity in the areas of rapid changes.

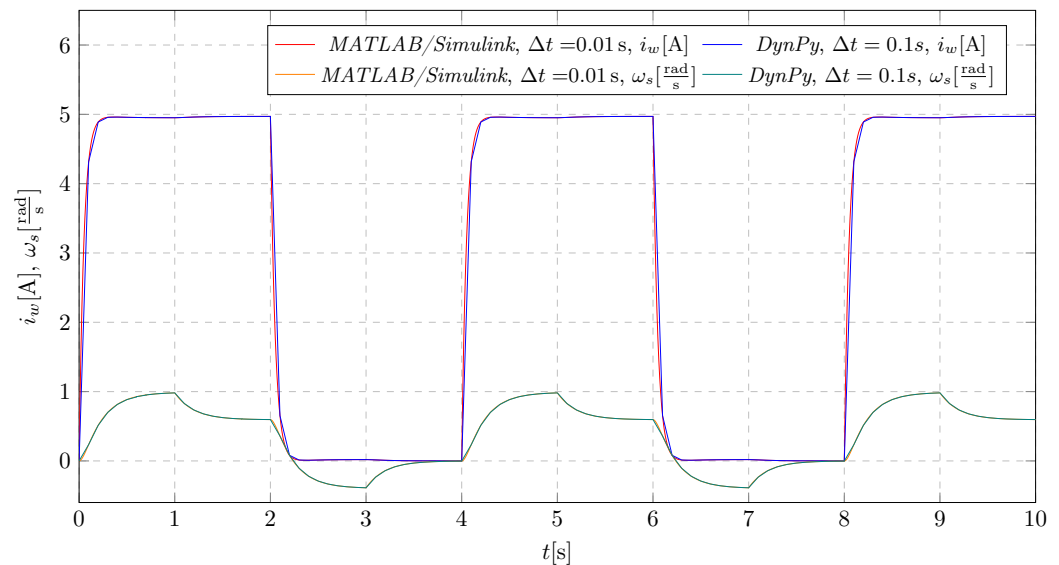


Figure 30. Comparison of simulations conducted using the *DynPy* library with a fixed integration step size of 0.1 s and *MATLAB/Simulink* with a fixed integration step size of 0.01 s.

Table 10. DC motor: comparison of results for chosen time stamps—*MATLAB/Simulink* with 0.01 s step and HAN method with 0.1 s step.

t [s]	<i>MATLAB</i> , $\Delta t = 0.01$ s		<i>DynPy</i> HAN, $\Delta t = 0.1$ s	
	i_w [A]	ω_s [$\frac{\text{rad}}{\text{s}}$]	i_w [A]	ω_s [$\frac{\text{rad}}{\text{s}}$]
0.100000	4.317996	0.236209	4.317466	0.236242
1.100000	4.955557	0.827836	4.955557	0.827837
2.100000	0.652203	0.359324	0.652732	0.359291
3.100000	0.014740	−0.233768	0.014739	−0.233768
4.100000	4.318094	0.234735	4.317565	0.234769
5.100000	4.955557	0.827827	4.955558	0.827828
6.100000	0.652203	0.359324	0.652732	0.359291
7.100000	0.014740	−0.233768	0.014739	−0.233768
8.100000	4.318094	0.234735	4.317565	0.234769
9.100000	4.955557	0.827827	4.955558	0.827828

4.2. RLC Circuit

Similarly, the comparison was prepared for the RLC circuit:

- With both solvers utilizing constant step of 0.1 s (Figure 31, Table 11);
- With both solvers utilizing constant step of 0.01 s (Figure 32, Table 12);
- With a variable integration step size in *MATLAB/Simulink*, 0.1 s and 0.01 s step size in *DynPy* (Figure 33, Table 13).

When considering the comparison of simulations for a 0.1 s step size, both qualitative and quantitative discrepancies are clearly visible (Figure 31). Above all, the curve generated in *MATLAB/Simulink* is smooth, while in the case of HAN, sudden changes in current value appear in the second, third, sixth, and seventh second. These changes have a periodic character and are directly related to the implementation of the method. The switch of datasets is realised every second, so to ensure the continuity of the solution, new initial values are introduced to its consecutive parts. As these initial conditions are obtained directly from the previous part of the solution for the same time stamp at which the next part of the solution starts, vertical shifts appear in the plot. A similar situation is expected

to take place at every dataset change, yet might not be visible due to the different slopes of the plot at other time stamps. On the other hand, the maximum amperage in the fourth and eighth seconds is higher in *MATLAB/Simulink*. The fact that the HAN method predominantly utilises an analytical approach lets us conclude that in this aspect, *DynPy* is more reliable.

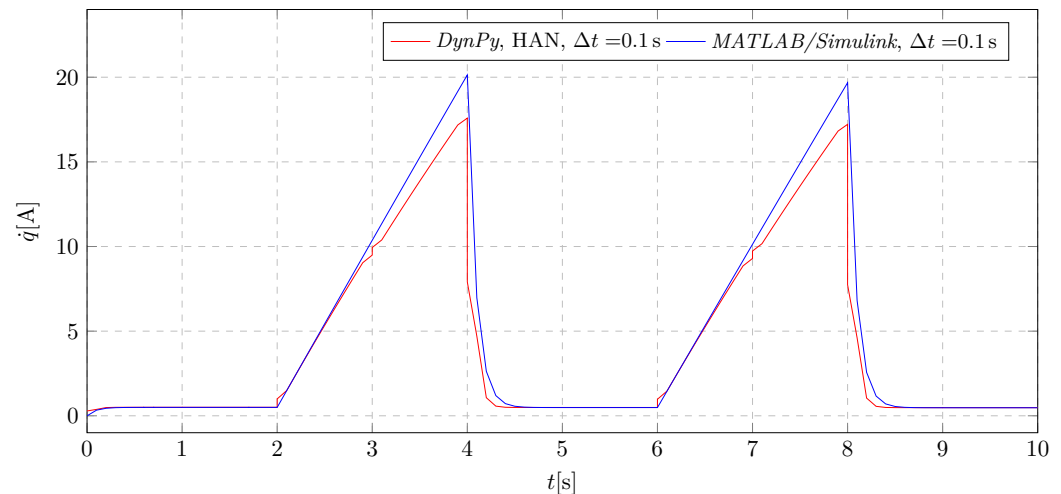


Figure 31. RLC circuit - comparison of simulations conducted using the *DynPy* library with a fixed integration step size of 0.1 s and *MATLAB/Simulink* with a fixed integration step size of 0.1 s.

Table 11. RLC circuit: comparison of results for chosen time stamps—*MATLAB/Simulink* with 0.1 s step and HAN method with 0.1 s step.

$t[s]$	<i>DynPy</i> , HAN, $\Delta t = 0.1$ s	<i>MATLAB/Simulink</i> , $\Delta t = 0.1$ s
	$\dot{q}[A]$	$\dot{q}[A]$
0.000000	0.000000	0.000000
1.000000	0.502298	0.497255
2.000000	0.502046	0.497013
3.000000	9.494228	10.368359
4.000000	17.585402	20.136108
5.000000	0.491754	0.486170
6.000000	0.491507	0.485625
7.000000	9.294929	10.130792
8.000000	17.216256	19.674735
9.000000	0.481431	0.475031
10.000000	0.481189	0.474498

The plot in Figure 32 compares the results from both environments with the step size set to 0.01 s. In this scenario, only qualitative difference is exposed—both graphs are equally smooth and generally take the same shape. As previously, maximum current values at the fourth and eighth second are smaller in comparison with *MATLAB/Simulink*.

Finally, Figure 33 shows the comparison between both constant step size HAN solutions and variable step size output from *MATLAB/Simulink*. The graph obtained from *MATLAB/Simulink* performs marginally better qualitatively than the 0.1 s HAN curve. In both cases, small, general waviness appears, as well as sudden rises in amperage at the third and seventh seconds (which also happen at the second and sixth seconds in the case of the 0.1 s step size HAN solution). The 0.01 s step size HAN method utilised in *DynPy* performs the best from the compared approaches, ensuring the highest smoothness and reliability of results.

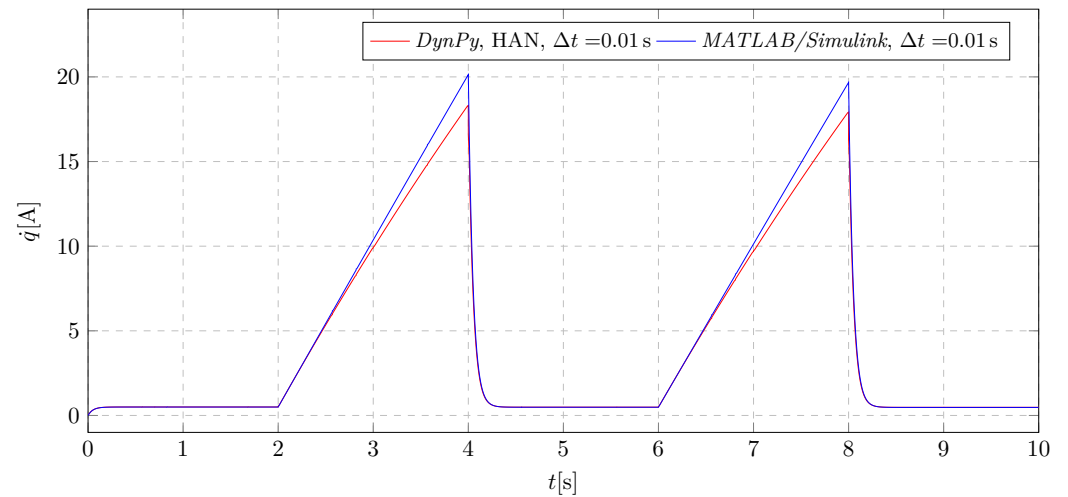


Figure 32. Comparison of simulations conducted using the *DynPy* library with a fixed integration step size of 0.01 s and *MATLAB/Simulink* with a fixed integration step size of 0.01 s.

Table 12. RLC circuit: comparison of results for chosen time stamps—*MATLAB/Simulink* with 0.01 s step and HAN method with 0.01 s step.

$t[s]$	<i>DynPy</i> , HAN, $\Delta t = 0.01$ s $\dot{q}[A]$	<i>MATLAB/Simulink</i> , $\Delta t = 0.1$ s $\dot{q}[A]$
0.000000	0.000000	0.000000
1.000000	0.502287	0.497263
2.000000	0.502034	0.497013
3.000000	9.898286	10.368359
4.000000	18.310948	20.136108
5.000000	0.491532	0.485869
6.000000	0.491285	0.485625
7.000000	9.686345	10.130792
8.000000	17.918876	19.674735
9.000000	0.481007	0.474736
10.000000	0.480765	0.474498

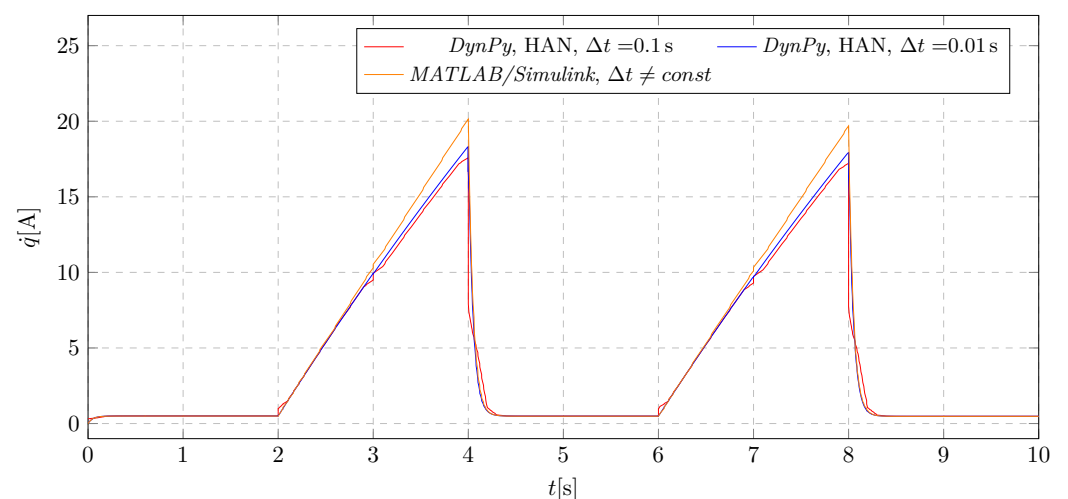


Figure 33. Comparison of simulations conducted using the *DynPy* library with a fixed integration step sizes of 0.01 s and 0.1 s compared with *MATLAB/Simulink* variable integration step size.

Table 13. RLC circuit: comparison of results for chosen time stamps—*MATLAB/Simulink* with variable step and HAN method with 0.1 s and 0.01 s step.

$t[s]$	$DynPy, \Delta t = 0.1 \text{ s}$	$DynPy, \Delta t = 0.01 \text{ s}$	$MATLAB/Simulink,$ $\Delta t \neq const$
	$\dot{q}[A]$	$\dot{q}[A]$	$\dot{q}[A]$
0.000000	0.000000	0.000000	0.000000
1.000000	0.502298	0.502287	0.497267
2.000000	0.502046	0.502034	0.497013
3.000000	9.494228	9.898286	10.295847
4.000000	17.585402	18.310948	20.144906
5.000000	0.491754	0.491532	0.485902
6.000000	0.491507	0.491285	0.485620
7.000000	9.294929	9.686345	10.099166
8.000000	17.216256	17.918876	19.692078
9.000000	0.481431	0.481007	0.474862
10.000000	0.481189	0.480765	0.474468

Regardless of the selected step size, from a quantitative point of view, the hybrid analytical–numerical method utilised with the *DynPy* library showed better performance than the strictly numerical approach. When considering the step size, when selected properly, it also ensures high smoothness, resulting in an overall high quality of the obtained results.

5. Conclusions

This paper presents a general investigation of the *DynPy* library and its application for scientific and engineering activities. An overview of the developed tool was presented, with deeper insight into its modules and functionalities responsible for the provision of analytical and numerical solutions to systems described with linear and nonlinear ordinary differential equations. Nonlinear (more realistic representation of physical systems) and linear (idealised for theoretical or simplified analysis) models can be adopted and solved in an efficient way.

The solution for a model under consideration obtained with the *DynPy* library was presented in hybrid analytical–numerical (HAN) and fully analytical forms, where in the case of the latter, Heaviside forcing inputs were included. The HAN methodology, which enables the possibility of setting the integration step size, was chosen for comparison with *MATLAB/Simulink* results. The results obtained from the HAN method were compared with numerical simulations conducted in *MATLAB/Simulink* environment.

The analysis of results for the DC motor case leads to the conclusion that for cases with a variable integration step and a constant step of 0.01 s in *MATLAB/Simulink*, the obtained results are consistent with the results obtained using the *DynPy* library with the specified constant step of 0.1 s. This confirms the correctness of the implementation in the developed library. However, in the case of a constant step of 0.1 s in *MATLAB/Simulink*, significant quantitative and qualitative discrepancies were observed between the results—both in terms of values and waveform shapes. Deviating values indicate that under specific conditions, the *DynPy* library may respond better to changes in system dynamics than the *MATLAB/Simulink* tool. Specifically, the hybrid analytical–numerical method not only proved to return proper results in opposition to simulation with the same step size in *MATLAB/Simulink* but also presented high accuracy with 0.01 s step simulations in the stated reference environment.

Summarizing the simulations of the RLC circuit, the main conclusion is that regardless of the selected step size, thanks to the solution being based on an analytical approach, the

results are quantitatively better than the strictly numerical ones obtained from each analysed *MATLAB/Simulink* case. Qualitative discrepancies are different between approaches subjected to comparison. It is clearly visible that the *DynPy* HAN solution with a properly set step size provides smoother curves than *MATLAB/Simulink* with variable steps. Hence, the analytical solution updated numerically with appropriate frequency provides the most reliable results.

The comparisons of simulation results indicate that the numerical algorithms in the *DynPy* library have been implemented correctly, and the results obtained are accurate. The paper itself was generated strictly with the use of *DynPy*'s functionalities—all steps from building the model, through finding an analytical solution, conducting the simulations and their visualisation, to generating editable `.tex` files were carried out in the same environment. Thus, not only the accuracy and reliability of the tool were proven but its versatility and usefulness as well. Conclusions drawn from this study will enable further optimisation and development of the library for various engineering applications. Some of the future research directions include the development of a more user-friendly interface for hybrid analytical–numerical methods, the expansion of built-in analytical solvers with more types of step functions, and general refining and maintenance. It is worth mentioning that the library undergoes constant development, and numerous new features and fixes were added during and after the writing of this article.

By no means do the authors of this article imply the overall superiority of the *DynPy* library over well-known and field-tested software, but the main advantages have to be pointed out for comparison. Firstly, it is written in the *Python* programming language. Although it requires the user to obtain at least a basic knowledge of its structure and syntax, it also assures the fact that insight into developed algorithms and their working principles is possible, as well as local editing, in contrast to previously described software. Moreover, a large user community of the language, in general, ensures additional support on top of the one possible to obtain from *DynPy*'s built-in guides and functioning `help` methods. Thanks to the utilisation of the object-oriented paradigm, a unique way to build dynamic systems was implemented. Said paradigm also enables the user to create self-defined basic elements, without the need of downloading additional toolboxes or add-ins. Such an approach results in a low level of initial theoretical, mathematical, and modelling knowledge needed to use the tool. At no point when building the system, manual definition of equations of motion or governing equations is necessary (but still possible if needed), since they are derived directly from the user-provided composition of the model. Of course, verification of the obtained results is necessary, as with any other tool. Another advantage lies in the scope of solving possibilities. The same model can be solved both analytically (if possible from the mathematical point of view) and numerically in the same environment. Lastly, *DynPy* also works as a text editing tool. Thanks to the implementation of self-written and *PyLaTeX* inherited functionalities, fully formatted papers like this one are possible to generate without the need for workspace change, so the model preparation, solution finding, simulations, and reporting are all in one place.

Author Contributions: D.S., data curation, formal analysis, software, supervision, validation, and writing—original draft; B.C., conceptualisation, formal analysis, software, supervision, and writing—review and editing; F.G., data curation, investigation, methodology, software, validation, and writing—original draft; P.P., formal analysis, supervision, and writing—review and editing; A.R., software and writing—original draft. All authors have read and agreed to the published version of the manuscript.

Funding: This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Data Availability Statement: The data presented in this study are openly available in GitHub, at <https://github.com/bogumilchilinski/dynpy> (accessed on 14 November 2024).

Conflicts of Interest: The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. BatteryCell Class Code

```

1      class BatteryCell(ComposedSystem):
2
3          scheme_name = 'thevenincircuit.PNG'
4          real_name = 'liioncell.PNG'
5
6          R_1=Symbol('R_1', positive=True)
7          R_2=Symbol('R_2', positive=True)
8          C=Symbol('C', positive=True)
9          U=Symbol('U', positive=True)
10         q_1=dynamicsymbols('q_1')
11         q_2=dynamicsymbols('q_2')
12         t = Symbol('t')
13         U_li = Function('U_li')(t)
14         U_oc = Symbol('U_oc')
15         R_0 = Symbol('R_0')
16         I_li = Function('I_zad')(t)
17         R_th = Symbol('R_th')
18         C_th = Symbol('C_th')
19         SOC = Function('SOC')(t)
20         SOC_init = Symbol('SOC_init')
21         C_rated = Symbol('C_rated')
22         t_0 = Symbol('t_0')
23         U_th = Function('U_th')(t)
24         funcI= Function('funcI')(t)
25
26     def __init__(self,
27                 R_1=None,
28                 R_2=None,
29                 C=None,
30                 U=None,
31                 q_1=None,
32                 q_2=None,
33                 t=None,
34                 U_th=None,
35                 R_th=None,
36                 C_th=None,
37                 I_li=None,
38                 ivar=Symbol('t'),
39                 **kwargs):
40
41         if t is not None: self.t = t
42         if R_1 is not None: self.R_1 = R_1
43         if R_2 is not None: self.R_2 = R_2
44         if C is not None: self.C= C
45         if U is not None: self.U = U
46         if q_1 is not None: self.q_1 = q_1
47         if q_2 is not None: self.q_2 = q_2
48         if U_th is not None: self.U_th = U_th
49         if R_th is not None: self.R_th = R_th
50         if C_th is not None: self.C_th = C_th
51         if I_li is not None: self.I_li = I_li
52         self.qs = [self.q_1, self.q_2]
53         self._init_from_components(**kwargs)
54
55     @property

```

```

56     def components(self):
57         components = {}
58         self._resistor_1 = Resistor(self.R_1, self.q_1, qs=[self.q_1])
59         self._resistor_2 = Resistor(self.R_2, q0=self.q_2, qs=[self.q_2])
60         self._voltagesource = VoltageSource(self.U, q0 = self.q_1, qs=[self.
q_1])
61         self._capacitor = Capacitor(self.C, q0 = self.q_1-self.q_2, qs=[self.
q_1, self.q_2])
62
63         components['resistor_1'] = self._resistor_1
64         components['resistor_2'] = self._resistor_2
65         components['voltagesource'] = self._voltagesource
66         components['capacitor'] = self._capacitor
67
68         return components
69
70     def voltage_ode(self):
71
72         t = self.ivar
73         U_th = self.U_th
74         R_th = self.R_th
75         C_th = self.C_th
76         I_li = self.I_li
77
78         uth_eq=Eq(diff(U_th,t, evaluate=False),((U_th)/(R_th*C_th))+I_li/C_th
)
79         ode_th = ODESystem(odes=uth_eq.rhs+uth_eq.lhs, dvars=U_th, ode_order
=1)
80
81         return ode_th
82
83     def uth_simulation(self,data_dict1,celldata, step_time,step_current):
84
85         t = self.ivar
86         U_th = self.U_th
87         R_th = self.R_th
88         C_th = self.C_th
89         I_li = self.I_li
90         U_li = self.U_li
91         U_oc = self.U_oc
92         R_0 = self.R_0
93         t_0 = self.t_0
94         SOC = self.SOC
95         C_rated = self.C_rated
96         SOC_init = self.SOC_init
97         ode_th = self.voltage_ode()
98
99         napiecie_eq=Eq(U_li,U_oc-R_0*I_li-U_th)
100
101         calka=integrate(I_li, (t, t_0, t))
102         SOC_eq_bezcalki = Eq(SOC,((1)/(C_rated)))
103         SOC_eq = Eq(SOC,SOC_init+((1)/(C_rated))*calka)
104
105         n_points=step_time[-1]*4
106         t_array=np.linspace(0,step_time[-1]*1,n_points)
107
108         prad_list=[]
109         u0_list=[]
110         u_array=[]
111         soc_array=[]
112
113         exprI=0*Heaviside(t-0)
114         for i in range(len(step_time)):

```

```

115         exprI=exprI + step_current[i]*Heaviside(t-step_time[i])
116     funcI = lambdify(t, exprI)
117
118     tabelka=ode_th.subs(I_li,exprI).subs(data_dict1).numerized(backend='
numpy').compute_solution(t_array,[0])
119
120     U_th_list = list(tabelka.iloc[:,1])
121
122
123
124
125     for i in range(n_points):
126         prad_list.append(funcI(t_array[i]))
127         soc_array.append(float((SOC_eq_bezcalki.rhs.subs(data_dict1)*(-1)
*(np.trapz(prad_list[0:i], x=t_array[0:i]))/3600+SOC_init.subs(data_dict1
))))
128
129         soc_calk = int(soc_array[i]*100)
130         data_dict1[self.U_oc]= celldata.at[soc_calk, 'OCV']
131         data_dict1[self.R_0]=celldata.at[soc_calk, 'R0']
132
133         u0_list.append( prad_list[-1] * data_dict1[self.R_0])
134         u_array.append(data_dict1[self.U_oc]-U_th_list[i]-u0_list[i])
135
136
137     wartosci = [u_array,prad_list,soc_array,t_array]
138
139     return wartosci
140
141
142 def _voltage_response(self,data_dict1,celldata,step_time,step_current):
143
144     t=self.t
145     U_li= self.U_li
146     wyn = self.uth_simulation(data_dict1,celldata,step_time,step_current)
147     t_array = wyn[3]
148     u_array = wyn[0]
149
150     df_u = pd.DataFrame({t : t_array})
151     df_u[U_li] = u_array
152     df_u = df_u.set_index(t)
153
154     return df_u
155
156 def _soc_level_response(self,data_dict1,celldata,step_time,step_current):
157
158     t=self.t
159     SOC= self.SOC
160     wyn = self.uth_simulation(data_dict1,celldata,step_time,step_current)
161     t_array = wyn[3]
162     soc_array = wyn[2]
163
164     df_SOC = pd.DataFrame({t : t_array})
165     df_SOC[SOC] = soc_array
166     df_SOC = df_SOC.set_index(t)
167
168     return df_SOC
169
170
171 def _current_forcing(self,data_dict1,celldata,step_time,step_current):
172
173     t=self.t
174     I_li= self.I_li

```

```

175         wyn = self.uth_simulation(data_dict1, celldata, step_time, step_current)
176         t_array = wyn[3]
177         prad_list = wyn[1]
178
179         df_I = pd.DataFrame({t : t_array})
180         df_I[I_li] = prad_list
181         df_I = df_I.set_index(t)
182
183         return df_I
184
185     %

```

Appendix B. DCMotor Class Code

```

1     class DCMotor(ComposedSystem):
2         scheme_name = 'DC_motor.png'
3
4         U_z = Symbol('U_z', positive=True)
5         R_w = Symbol('R_w', positive=True)
6         L_w = Symbol('L_w', positive=True)
7         E = Symbol('E', positive=True)
8         U_Rw = Symbol('U_Rw', positive=True)
9         U_Lw = Symbol('U_Lw', positive=True)
10        k_e = Symbol('k_e', positive=True)
11        M_s = Symbol('M_s', positive=True)
12        B = Symbol('B', positive=True)
13        J = Symbol('J', positive=True)
14        M_obc = Symbol('M_l', positive=True)
15        k_m = Symbol('k_m', positive=True)
16        M_a = Symbol('M_a', positive=True)
17        M_r = Symbol('M_r', positive=True)
18        i_w=dynamicsymbols('i_w')
19        omega_s=dynamicsymbols('omega_s')
20
21
22
23        def __init__(self,
24                    U_z=None,
25                    R_w=None,
26                    L_w=None,
27                    E=None,
28                    U_Rw=None,
29                    U_Lw=None,
30                    k_e=None,
31                    M_s=None,
32                    B=None,
33                    J=None,
34                    M_obc=None,
35                    k_m=None,
36                    M_a=None,
37                    M_r=None,
38                    ivar=Symbol('t'),
39                    **kwargs):
40
41            if U_z is not None: self.U_z = U_z
42            if R_w is not None: self.R_w = R_w
43            if L_w is not None: self.L_w = L_w
44            if E is not None: self.E = E
45            if U_Rw is not None: self.U_Rw = U_Rw
46            if U_Lw is not None: self.U_Lw = U_Lw
47            if k_e is not None: self.k_e = k_e
48            if M_s is not None: self.M_s = M_s
49            if B is not None: self.B = B

```

```

50         if J is not None: self.J = J
51         if M_obc is not None: self.M_obc = M_obc
52         if k_m is not None: self.k_m = k_m
53         if M_a is not None: self.M_a = M_a
54         if M_r is not None: self.M_r = M_r
55
56         self.ivar = ivar
57         self.qs = [self.i_w, self.omega_s]
58         self._init_from_components(**kwargs)
59
60     @property
61     def components(self):
62         components = {}
63
64         self.resistor = CurrentDependentResistor(-self.R_w, self.i_w, qs=
self.qs, ivar=self.ivar, frame=base_frame)
65         self.inductor = Resistor(-self.L_w, self.i_w, qs=self.qs, ivar=self.
ivar, frame=base_frame)
66 #         self.inductor = Inductor(-self.L_w, self.i_w, qs=self.qs, ivar=self
.ivar, frame=base_frame)
67         self.voltage_source = VoltageSource(-self.U_z, self.i_w, qs=self.qs,
ivar=self.ivar, frame=base_frame)
68         self.electromotive_force = ElectromotiveForce(self.k_e*self.omega_s,
self.i_w, qs=self.qs, ivar=self.ivar, frame=base_frame)
69         self.engine_load_torque = EngineLoadTorque(self.M_obc, self.omega_s,
qs=self.qs, ivar=self.ivar, frame=base_frame)
70         self.rotor_torque = RotorTorque(-self.k_m*self.i_w, self.omega_s, qs=
self.qs, ivar=self.ivar, frame=base_frame)
71         self.rotor_angular_acceleration_torque =
RotorAngularAccelerationTorque(-self.J, self.omega_s, qs=self.qs, ivar=
self.ivar, frame=base_frame)
72         self.rotor_resistance_movement_torque = ResistanceMovementTorque(-
self.B, self.omega_s, qs=self.qs, ivar=self.ivar, frame=base_frame)
73
74         components['resistor'] = self.resistor
75         components['inductor'] = self.inductor
76         components['voltage_source'] = self.voltage_source
77         components['electromotive_force'] = self.electromotive_force
78         components['engine_load_torque'] = self.engine_load_torque
79         components['rotor_torque'] = self.rotor_torque
80         components['rotor_angular_acceleration_torque'] = self.
rotor_angular_acceleration_torque
81         components['rotor_resistance_movement_torque'] = self.
rotor_resistance_movement_torque
82
83         return components
84
85     def symbols_description(self):
86         i_roc=self.i_w.diff(self.ivar)
87         self.sym_desc_dict = {
88             self.U_z: r'voltage supplying the rotor',
89             self.R_w: r'equivalent resistance of the rotor windings',
90             self.L_w: r'equivalent inductance of the rotor windings',
91             self.E: r'electromotive force of induction',
92             self.U_Rw: r'voltage across the rotor winding resistance',
93             self.U_Lw: r'voltage related to the rotor inductance',
94             self.k_e: r'back electromotive force constant',
95             self.M_s: r'rotor torque',
96             self.B: r'coefficient of viscous friction reduced to the rotor
shaft',
97             self.J: r'moment of inertia reduced to the rotor shaft',
98             self.M_obc: r'engine load torque',
99             self.k_m: r'torque constant',

```

```

100         self.i_w: r'rotor winding current',
101         i_roc: r'rotor winding current rate of change',
102         self.omega_s: r'angular velocity of the rotor',
103         self.M_a: r'rotor angular acceleration torque',
104         self.M_r: r'rotor motion resistance torque',
105         self.omega_s.diff(self.ivar): r'angular acceleration of the rotor'
106     },
107     return self.sym_desc_dict
108
109 def get_default_data(self):
110     default_data_dict = {
111         self.U_z: [10],
112         self.R_w: [2],
113         self.L_w: [0.1],
114         self.k_e: [0.1],
115         self.k_m: [0.1],
116         self.J: [0.1],
117         self.B: [0.5],
118         self.M_abc: [0.2]
119     }
120     return default_data_dict
121
122
123 def _dimensionless_ode(self):
124
125     from ...solvers.linear import ODESystem, FirstOrderLinearODESystem
126
127     t = self.ivar
128     tau = Symbol('tau')
129
130     syms_dict = {
131         # dcmotor.U_z: 0,
132         self.R_w: 1,
133         self.L_w: 1,
134         # dcmotor.k_e: Symbol('kappa', positive=True),
135         # dcmotor.k_m: Symbol('kappa', positive=True),
136
137         self.k_e: Symbol('kappa', negative=True),
138         self.k_m: Symbol('kappa', negative=True),
139
140         self.J: 1,
141         self.B: 1,
142         t: tau,
143         # dcmotor.M_abc: 10,
144     }
145     dvars = Matrix([[self.i_w], [self.omega_s]]).subs({t: tau})
146
147
148     ode = ODESystem(odes=(self._eoms.subs(syms_dict))*-1, dvars = dvars,
149                     ode_order=1, ivar=tau)
150
151     #return FirstOrderLinearODESystem.from_ode_system(ode)
152     return ode
153
154 %

```

Appendix C. Detailed View of DCMotor Class Dependencies and Equations of Motion Solutions

The structure of the DCMotor class, which is inherited from the parent class ComposedSystem, enables the user to display crucial mathematical information about the

analysed system. To obtain the equations of motion, Lagrange's equations of a second kind are utilised, which are built based on the type and number of components of which the system is made. As such, the kinetic energy, potential energy, and dissipative potential of the DC motor can be displayed by the following commands:

```
1 dcmotor=DCMotor
2 dcmotor()._kinetic_energy
3 dcmotor()._potential_energy
4 dcmotor()._dissipative_potential
```

The resulting output, provided in (A1)–(A3), requires some clarification. Referenced formulas are created by another functionality of the *DynPy* library intended for generating .tex and .pdf documents and SympyFormula objects by directly introducing properties described previously as arguments:

```
1 SympyFormula(dcmotor()._kinetic_energy)
2 SympyFormula(dcmotor()._potential_energy)
3 SympyFormula(dcmotor()._dissipative_potential)
```

$$\frac{J\omega_s^2}{2} - \frac{L_w i_w^2}{2} \quad (\text{A1})$$

$$(\text{A2})$$

$$\frac{B\omega_s^2}{2} - \frac{R_w i_w^2}{2} \quad (\text{A3})$$

In general, these properties do not return equations but simply symbolic descriptions of the quantities of interest. Since the potential energy of the analysed system is equal to zero, nothing appeared in (A2), and on top of that, (A1) and (A3) lack the equality signs; hence, no actual right- and left-hand sides can be determined. This does not affect the usability of the interpreter and can be easily improved for cases of reporting by introducing additional symbols and Eq class, as in the following example:

```
1 T=Symbol('T', positive=True)
2 V=Symbol('V', positive=True)
3 D=Symbol('D', positive=True)
4 SympyFormula(Eq(T, dcmotor()._kinetic_energy, evaluate=False))
5 SympyFormula(Eq(V, dcmotor()._potential_energy))
6 SympyFormula(Eq(D, dcmotor()._dissipative_potential))
```

This slight modification results in the following formulas:

$$T = -\frac{J\dot{\phi}^2}{2} - \frac{L_w \dot{q}^2}{2} \quad (\text{A4})$$

$$V = 0 \quad (\text{A5})$$

$$D = -\frac{B\dot{\phi}^2}{2} - \frac{R_w \dot{q}^2}{2} \quad (\text{A6})$$

Further in this appendix, to avoid ambiguities, all of the quantities and relations generated from class built-in methods are presented in proper mathematical representation. Another feature of the system that is possible to obtain is its Lagrangian, called by

```
1 L=Symbol('L', positive=True)
2 SympyFormula(Eq(L, dcmotor()._lagrangian))
```

$$L = -\frac{J\dot{\phi}^2}{2} - \frac{L_w \dot{q}^2}{2} \quad (\text{A7})$$

Also, inertia, stiffness, damping, and external forces matrices are available. To show them in the form of equations, MatrixSymbol objects have to be defined, which include the size of the matrices as an argument:

```

1 M=MatrixSymbol('M',2,2)
2 K=MatrixSymbol('K',2,2)
3 C=MatrixSymbol('C',2,2)
4 F=MatrixSymbol('F',2,1)
5 SympyFormula(Eq(M, dcmotor().inertia_matrix()))
6 SympyFormula(Eq(K, dcmotor().stiffness_matrix()))
7 SympyFormula(Eq(C, dcmotor().damping_matrix()))
8 SympyFormula(Eq(F, dcmotor().external_forces()))

```

$$M = \begin{bmatrix} -L_w & 0 \\ 0 & -J \end{bmatrix} \quad (\text{A8})$$

$$K = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad (\text{A9})$$

$$C = \begin{bmatrix} -R_w & -k_e \\ k_m & -B \end{bmatrix} \quad (\text{A10})$$

$$F = \begin{bmatrix} -U_z \\ M_l \end{bmatrix} \quad (\text{A11})$$

The equations of motion are accessible through

```
1 dcmotor()._eoms
```

A single matrix of equations is returned; hence, to obtain separate formulas, they need to be extracted and introduced into Eq class with right-hand sides equal to zero (or with the as_eq_list method, as in the main part of this article):

```

1 SympyFormula(Eq(dcmotor()._eoms[0], 0))
2 SympyFormula(Eq(dcmotor()._eoms[1], 0))

```

$$U_z - L_w \dot{i}_w - R_w i_w - k_e \omega_s = 0 \quad (\text{A12})$$

$$M_l + k_m i_w - B \omega_s - J \dot{\omega}_s = 0 \quad (\text{A13})$$

Several more quantities are obtainable, such as natural frequencies or frequency response functions, which are not further described. A full form of the solution for the ODESystem object of DCMotor is presented in Equations (A14) and (A17), as well as their derivatives (A16) and (A17):

$$\omega_s = \frac{U_z k_m}{B R_w + k_e k_m} - \frac{M_l R_w}{B R_w + k_e k_m} + C_1 e^{-\frac{B t}{2 J}} e^{-\frac{R_w t}{2 L_w}} e^{-\frac{t \sqrt{B^2 L_w^2 - 2 B J L_w R_w + J^2 R_w^2 - 4 J L_w k_e k_m}}{2 J L_w}} + C_2 e^{-\frac{B t}{2 J}} e^{-\frac{R_w t}{2 L_w}} e^{\frac{t \sqrt{B^2 L_w^2 - 2 B J L_w R_w + J^2 R_w^2 - 4 J L_w k_e k_m}}{2 J L_w}} \quad (\text{A14})$$

$$\begin{aligned}
i_w = & \frac{BU_z}{BR_w + k_e k_m} + \frac{M_l k_e}{BR_w + k_e k_m} + \frac{BC_1 e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2k_m} \\
& + \frac{BC_2 e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2k_m} \\
& + \frac{C_2 \sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m} e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2L_w k_m} \\
& - \frac{C_1 \sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m} e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2L_w k_m} \\
& - \frac{C_1 J R_w e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2L_w k_m} \\
& - \frac{C_2 J R_w e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2L_w k_m}
\end{aligned} \tag{A15}$$

$$\begin{aligned}
\dot{\omega}_s = & - \frac{BC_1 e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2J} \\
& - \frac{C_1 R_w e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2L_w} \\
& - \frac{BC_2 e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2J} \\
& - \frac{C_2 R_w e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2L_w} \\
& + \frac{C_2 \sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m} e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2JL_w} \\
& - \frac{C_1 \sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m} e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2JL_w}
\end{aligned} \tag{A16}$$

$$\begin{aligned}
\dot{i}_w = & - \frac{C_1 k_e e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{L_w} - \frac{C_2 k_e e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{L_w} \\
& + \frac{C_1 J R_w^2 e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2L_w^2 k_m} \\
& + \frac{C_1 R_w \sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m} e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2L_w^2 k_m} \\
& + \frac{C_2 J R_w^2 e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2L_w^2 k_m} - \frac{BC_1 R_w e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2L_w k_m} \\
& - \frac{BC_2 R_w e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2L_w k_m} \\
& - \frac{C_2 R_w \sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m} e^{-\frac{Bt}{2J}} e^{-\frac{R_w t}{2L_w}} e^{-\frac{t\sqrt{B^2 L_w^2 - 2BJL_w R_w + J^2 R_w^2 - 4JL_w k_e k_m}}{2JL_w}}}{2L_w^2 k_m}
\end{aligned} \tag{A17}$$

Similarly, a solution of equations of motion including Heaviside forcing based on sinewave and their derivatives are presented subsequently in (A18)–(A19) and (A20)–(A21). For clarity

of view (very long components of the equations disabled proper line breaking), their dimensionless versions are provided, where τ is dimensionless time, i is the imaginary unit, κ is the dimensionless coupling coefficient, μ is the torque reduced to the shaft rotation angle, v is the voltage reduced to armature current, and N is the number of Heaviside function cycles.

$$\begin{aligned} i_w(\tau) = & C_1 e^{\tau(-i\kappa-1)} + C_2 e^{\tau(i\kappa-1)} + \left(-\frac{\kappa\mu}{-\kappa^2-1} - \frac{e\mu e^{\tau(i\kappa-1)}}{2\kappa e^{i\kappa} + 2ie^{i\kappa}} - \frac{e\mu e^{i\kappa} e^{\tau(-i\kappa-1)}}{2\kappa-2i} \right) \theta\left(-\cos\left(\frac{2\pi\tau}{T}\right)\right) \\ & + \sum_{n=-N}^N \left(\frac{ive^{\frac{T}{2}} e^{Tn} e^{\tau(i\kappa-1)}}{2\kappa e^{\frac{iT\kappa}{2}} e^{iT\kappa n} + 2ie^{\frac{iT\kappa}{2}} e^{iT\kappa n}} + \frac{v}{-\kappa^2-1} - \frac{ive^{\frac{T}{2}} e^{Tn} e^{\tau(-i\kappa-1)} e^{\frac{iT\kappa}{2}} e^{iT\kappa n}}{2\kappa-2i} \right) \theta\left(-Tn - \frac{T}{2} + \tau\right) \\ & + \sum_{n=-N}^N \left(-\frac{ive^{Tn} e^{\tau(i\kappa-1)}}{2\kappa e^{iT\kappa n} + 2ie^{iT\kappa n}} - \frac{v}{-\kappa^2-1} + \frac{ive^{Tn} e^{\tau(-i\kappa-1)} e^{iT\kappa n}}{2\kappa-2i} \right) \theta(-Tn + \tau) \end{aligned} \quad (A18)$$

$$\begin{aligned} \omega_s(\tau) = & \left(\frac{e\mu e^{\tau(i\kappa-1)}}{2\kappa e^{i\kappa} + 2ie^{i\kappa}} - \frac{\mu}{\kappa^2+1} - \frac{e\mu e^{i\kappa} e^{\tau(-i\kappa-1)}}{2\kappa-2i} \right) \theta\left(-\cos\left(\frac{2\pi\tau}{T}\right)\right) + iC_1 e^{\tau(-i\kappa-1)} - iC_2 e^{\tau(i\kappa-1)} \\ & + \sum_{n=-N}^N \left(\frac{\kappa v}{\kappa^2+1} - \frac{ve^{Tn} e^{\tau(i\kappa-1)}}{2\kappa e^{iT\kappa n} + 2ie^{iT\kappa n}} - \frac{ve^{Tn} e^{\tau(-i\kappa-1)} e^{iT\kappa n}}{2\kappa-2i} \right) \theta(-Tn + \tau) \\ & + \sum_{n=-N}^N \left(-\frac{\kappa v}{\kappa^2+1} + \frac{ve^{\frac{T}{2}} e^{Tn} e^{\tau(i\kappa-1)}}{2\kappa e^{\frac{iT\kappa}{2}} e^{iT\kappa n} + 2ie^{\frac{iT\kappa}{2}} e^{iT\kappa n}} + \frac{ve^{\frac{T}{2}} e^{Tn} e^{\tau(-i\kappa-1)} e^{\frac{iT\kappa}{2}} e^{iT\kappa n}}{2\kappa-2i} \right) \theta\left(-Tn - \frac{T}{2} + \tau\right) \end{aligned} \quad (A19)$$

$$\begin{aligned} \frac{d}{d\tau} i_w(\tau) = & -C_1 e^{-\tau} e^{-i\kappa\tau} - C_2 e^{-\tau} e^{i\kappa\tau} + \frac{e\mu e^{i\kappa} \theta(-\cos(\frac{2\pi\tau}{T}))}{2\kappa e^{\tau} e^{i\kappa\tau} - 2ie^{\tau} e^{i\kappa\tau}} + \frac{e\mu e^{i\kappa\tau} \theta(-\cos(\frac{2\pi\tau}{T}))}{2\kappa e^{\tau} e^{i\kappa} + 2ie^{\tau} e^{i\kappa}} \\ & - iC_2 \kappa e^{-\tau} e^{i\kappa\tau} - iC_1 \kappa e^{-\tau} e^{-i\kappa\tau} - \frac{2\pi\kappa\mu \sin(\frac{2\pi\tau}{T}) \delta(\cos(\frac{2\pi\tau}{T}))}{-T\kappa^2 - T} \\ & - \frac{e\kappa\mu e^{i\kappa} \theta(-\cos(\frac{2\pi\tau}{T}))}{2\kappa e^{\tau} e^{i\kappa\tau} - 2ie^{\tau} e^{i\kappa\tau}} - \frac{e\kappa\mu e^{i\kappa\tau} \theta(-\cos(\frac{2\pi\tau}{T}))}{2\kappa e^{\tau} e^{i\kappa} + 2ie^{\tau} e^{i\kappa}} \\ & - \frac{2e\pi\mu e^{i\kappa} \sin(\frac{2\pi\tau}{T}) \delta(\cos(\frac{2\pi\tau}{T}))}{2T\kappa e^{\tau} e^{i\kappa\tau} - 2iT e^{\tau} e^{i\kappa\tau}} - \frac{2e\pi\mu e^{i\kappa\tau} \sin(\frac{2\pi\tau}{T}) \delta(\cos(\frac{2\pi\tau}{T}))}{2T\kappa e^{\tau} e^{i\kappa} + 2iT e^{\tau} e^{i\kappa}} \\ & + \sum_{n=-N}^N \frac{v\delta(-Tn - \frac{T}{2} + \tau)}{-\kappa^2-1} + \sum_{n=-N}^N -\frac{v\delta(-Tn + \tau)}{-\kappa^2-1} \\ & + \sum_{n=-N}^N \frac{ive^{Tn} e^{iT\kappa n} \delta(-Tn + \tau)}{2\kappa e^{\tau} e^{i\kappa\tau} - 2ie^{\tau} e^{i\kappa\tau}} + \sum_{n=-N}^N \frac{ive^{Tn} e^{i\kappa\tau} \theta(-Tn + \tau)}{2\kappa e^{\tau} e^{iT\kappa n} + 2ie^{\tau} e^{iT\kappa n}} \\ & + \sum_{n=-N}^N \frac{\kappa ve^{Tn} e^{iT\kappa n} \theta(-Tn + \tau)}{2\kappa e^{\tau} e^{i\kappa\tau} - 2ie^{\tau} e^{i\kappa\tau}} + \sum_{n=-N}^N \frac{\kappa ve^{Tn} e^{i\kappa\tau} \theta(-Tn + \tau)}{2\kappa e^{\tau} e^{iT\kappa n} + 2ie^{\tau} e^{iT\kappa n}} \\ & + \sum_{n=-N}^N -\frac{ive^{Tn} e^{iT\kappa n} \theta(-Tn + \tau)}{2\kappa e^{\tau} e^{i\kappa\tau} - 2ie^{\tau} e^{i\kappa\tau}} + \sum_{n=-N}^N -\frac{ive^{Tn} e^{i\kappa\tau} \delta(-Tn + \tau)}{2\kappa e^{\tau} e^{iT\kappa n} + 2ie^{\tau} e^{iT\kappa n}} \\ & + \sum_{n=-N}^N \frac{ive^{\frac{T}{2}} e^{Tn} e^{i\kappa\tau} \delta(-Tn - \frac{T}{2} + \tau)}{2\kappa e^{\tau} e^{\frac{iT\kappa}{2}} e^{iT\kappa n} + 2ie^{\tau} e^{\frac{iT\kappa}{2}} e^{iT\kappa n}} + \sum_{n=-N}^N -\frac{ive^{\frac{T}{2}} e^{Tn} e^{i\kappa\tau} \theta(-Tn - \frac{T}{2} + \tau)}{2\kappa e^{\tau} e^{\frac{iT\kappa}{2}} e^{iT\kappa n} + 2ie^{\tau} e^{\frac{iT\kappa}{2}} e^{iT\kappa n}} \\ & + \sum_{n=-N}^N -\frac{\kappa ve^{\frac{T}{2}} e^{Tn} e^{i\kappa\tau} \theta(-Tn - \frac{T}{2} + \tau)}{2\kappa e^{\tau} e^{\frac{iT\kappa}{2}} e^{iT\kappa n} + 2ie^{\tau} e^{\frac{iT\kappa}{2}} e^{iT\kappa n}} \\ & + \sum_{n=-N}^N \frac{ive^{\frac{T}{2}} e^{Tn} e^{\frac{iT\kappa}{2}} e^{iT\kappa n} \theta(-Tn - \frac{T}{2} + \tau)}{2\kappa e^{\tau} e^{i\kappa\tau} - 2ie^{\tau} e^{i\kappa\tau}} \\ & + \sum_{n=-N}^N -\frac{ive^{\frac{T}{2}} e^{Tn} e^{\frac{iT\kappa}{2}} e^{iT\kappa n} \delta(-Tn - \frac{T}{2} + \tau)}{2\kappa e^{\tau} e^{i\kappa\tau} - 2ie^{\tau} e^{i\kappa\tau}} \\ & + \sum_{n=-N}^N -\frac{\kappa ve^{\frac{T}{2}} e^{Tn} e^{\frac{iT\kappa}{2}} e^{iT\kappa n} \theta(-Tn - \frac{T}{2} + \tau)}{2\kappa e^{\tau} e^{i\kappa\tau} - 2ie^{\tau} e^{i\kappa\tau}} \end{aligned} \quad (A20)$$

$$\begin{aligned}
\frac{d}{d\tau}\omega_s(\tau) = & iC_2e^{-\tau}e^{i\kappa\tau} - C_1\kappa e^{-\tau}e^{-i\kappa\tau} - C_2\kappa e^{-\tau}e^{i\kappa\tau} - iC_1e^{-\tau}e^{-i\kappa\tau} \\
& - \frac{2\pi\mu \sin\left(\frac{2\pi\tau}{T}\right)\delta\left(\cos\left(\frac{2\pi\tau}{T}\right)\right)}{T\kappa^2 + T} + \frac{e i \mu e^{i\kappa\tau}(-\cos\left(\frac{2\pi\tau}{T}\right))}{2\kappa e^\tau e^{i\kappa\tau} - 2ie^\tau e^{i\kappa\tau}} \\
& - \frac{e i \mu e^{i\kappa\tau}\theta(-\cos\left(\frac{2\pi\tau}{T}\right))}{2\kappa e^\tau e^{i\kappa} + 2ie^\tau e^{i\kappa}} - \frac{e \kappa \mu e^{i\kappa}\theta(-\cos\left(\frac{2\pi\tau}{T}\right))}{2\kappa e^\tau e^{i\kappa\tau} - 2ie^\tau e^{i\kappa\tau}} - \frac{e \kappa \mu e^{i\kappa\tau}\theta(-\cos\left(\frac{2\pi\tau}{T}\right))}{2\kappa e^\tau e^{i\kappa} + 2ie^\tau e^{i\kappa}} \\
& - \frac{2e i \pi \mu e^{i\kappa} \sin\left(\frac{2\pi\tau}{T}\right)\delta\left(\cos\left(\frac{2\pi\tau}{T}\right)\right)}{2T\kappa e^\tau e^{i\kappa\tau} - 2iT e^\tau e^{i\kappa\tau}} + \frac{2e i \pi \mu e^{i\kappa\tau} \sin\left(\frac{2\pi\tau}{T}\right)\delta\left(\cos\left(\frac{2\pi\tau}{T}\right)\right)}{2T\kappa e^\tau e^{i\kappa} + 2iT e^\tau e^{i\kappa}} \\
& + \sum_{n=-N}^N \frac{\kappa v \delta(-Tn + \tau)}{\kappa^2 + 1} + \sum_{n=-N}^N -\frac{\kappa v \delta\left(-Tn - \frac{T}{2} + \tau\right)}{\kappa^2 + 1} \\
& + \sum_{n=-N}^N \frac{v e^{Tn} e^{iT\kappa n} \theta(-Tn + \tau)}{2\kappa e^\tau e^{i\kappa\tau} - 2ie^\tau e^{i\kappa\tau}} + \sum_{n=-N}^N \frac{v e^{Tn} e^{i\kappa\tau} \theta(-Tn + \tau)}{2\kappa e^\tau e^{iT\kappa n} + 2ie^\tau e^{iT\kappa n}} \\
& + \sum_{n=-N}^N -\frac{v e^{Tn} e^{iT\kappa n} \delta(-Tn + \tau)}{2\kappa e^\tau e^{i\kappa\tau} - 2ie^\tau e^{i\kappa\tau}} + \sum_{n=-N}^N -\frac{v e^{Tn} e^{i\kappa\tau} \delta(-Tn + \tau)}{2\kappa e^\tau e^{iT\kappa n} + 2ie^\tau e^{iT\kappa n}} \\
& + \sum_{n=-N}^N \frac{v e^{\frac{T}{2}} e^{Tn} e^{i\kappa\tau} \delta\left(-Tn - \frac{T}{2} + \tau\right)}{2\kappa e^\tau e^{\frac{iT\kappa}{2}} e^{iT\kappa n} + 2ie^\tau e^{\frac{iT\kappa}{2}} e^{iT\kappa n}} + \sum_{n=-N}^N -\frac{v e^{\frac{T}{2}} e^{Tn} e^{i\kappa\tau} \theta\left(-Tn - \frac{T}{2} + \tau\right)}{2\kappa e^\tau e^{\frac{iT\kappa}{2}} e^{iT\kappa n} + 2ie^\tau e^{\frac{iT\kappa}{2}} e^{iT\kappa n}} \\
& + \sum_{n=-N}^N \frac{i\kappa v e^{Tn} e^{iT\kappa n} \theta(-Tn + \tau)}{2\kappa e^\tau e^{i\kappa\tau} - 2ie^\tau e^{i\kappa\tau}} + \sum_{n=-N}^N \frac{v e^{\frac{T}{2}} e^{Tn} e^{\frac{iT\kappa}{2}} e^{iT\kappa n} \delta\left(-Tn - \frac{T}{2} + \tau\right)}{2\kappa e^\tau e^{i\kappa\tau} - 2ie^\tau e^{i\kappa\tau}} \\
& + \sum_{n=-N}^N -\frac{i\kappa v e^{Tn} e^{i\kappa\tau} \theta(-Tn + \tau)}{2\kappa e^\tau e^{iT\kappa n} + 2ie^\tau e^{iT\kappa n}} + \sum_{n=-N}^N -\frac{v e^{\frac{T}{2}} e^{Tn} e^{\frac{iT\kappa}{2}} e^{iT\kappa n} \theta\left(-Tn - \frac{T}{2} + \tau\right)}{2\kappa e^\tau e^{i\kappa\tau} - 2ie^\tau e^{i\kappa\tau}} \\
& + \sum_{n=-N}^N \frac{i\kappa v e^{\frac{T}{2}} e^{Tn} e^{i\kappa\tau} \theta\left(-Tn - \frac{T}{2} + \tau\right)}{2\kappa e^\tau e^{\frac{iT\kappa}{2}} e^{iT\kappa n} + 2ie^\tau e^{\frac{iT\kappa}{2}} e^{iT\kappa n}} \\
& + \sum_{n=-N}^N -\frac{i\kappa v e^{\frac{T}{2}} e^{Tn} e^{\frac{iT\kappa}{2}} e^{iT\kappa n} \theta\left(-Tn - \frac{T}{2} + \tau\right)}{2\kappa e^\tau e^{i\kappa\tau} - 2ie^\tau e^{i\kappa\tau}}
\end{aligned} \tag{A21}$$

Appendix D. Hybrid Analytical–Numerical Solution Code

Code snippet of the hybrid analytical–numerical solution:

```

1  def han_solution():
2
3      dane0 = {U_z: 10, R_w: 2, L_w: 0.1, k_e: 0.1, k_m: 0.1, J: 0.1, B: 0.5,
4               M_1: 0}
5      dane1 = {U_z: 10, R_w: 2, L_w: 0.1, k_e: 0.1, k_m: 0.1, J: 0.1, B: 0.5,
6               M_1: 0.2}
7      dane2 = {U_z: 0, R_w: 2, L_w: 0.1, k_e: 0.1, k_m: 0.1, J: 0.1, B: 0.5,
8               M_1: 0.2}
9      dane3 = {U_z: 0, R_w: 2, L_w: 0.1, k_e: 0.1, k_m: 0.1, J: 0.1, B: 0.5,
10               M_1: 0}
11
12      dcmotor=DCMotor
13      ode = ODESystem(odes=dcmotor()._eoms, dvars=Matrix([dcmotor.i_w, dcmotor.
14                  omega_s]), ode_order=1)
15
16      sym_sol = ode.solution
17
18      num_1 = sym_sol.subs(dane0).with_ics([0, 0]).compute_solution(t_span=np.
19                  linspace(0, 1, 11))[[i_w, omega_s]]

```

```

14     num_2 = sym_sol.subs(dane1).with_ics([num_1.iloc[-1][i_w], num_1.iloc
[-1][omega_s]], ivar0=1).compute_solution(t_span=np.linspace(1, 2, 11))[[
i_w, omega_s]]
15     num_3 = sym_sol.subs(dane2).with_ics([num_2.iloc[-1][i_w], num_2.iloc
[-1][omega_s]], ivar0=2).compute_solution(t_span=np.linspace(2, 3, 11))[[
i_w, omega_s]]
16     num_4 = sym_sol.subs(dane3).with_ics([num_3.iloc[-1][i_w], num_3.iloc
[-1][omega_s]], ivar0=3).compute_solution(t_span=np.linspace(3, 4, 11))[[
i_w, omega_s]]
17     num_5 = sym_sol.subs(dane0).with_ics([num_4.iloc[-1][i_w], num_4.iloc
[-1][omega_s]], ivar0=4).compute_solution(t_span=np.linspace(4, 5, 11))[[
i_w, omega_s]]
18     num_6 = sym_sol.subs(dane1).with_ics([num_5.iloc[-1][i_w], num_5.iloc
[-1][omega_s]], ivar0=5).compute_solution(t_span=np.linspace(5, 6, 11))[[
i_w, omega_s]]
19     num_7 = sym_sol.subs(dane2).with_ics([num_6.iloc[-1][i_w], num_6.iloc
[-1][omega_s]], ivar0=6).compute_solution(t_span=np.linspace(6, 7, 11))[[
i_w, omega_s]]
20     num_8 = sym_sol.subs(dane3).with_ics([num_7.iloc[-1][i_w], num_7.iloc
[-1][omega_s]], ivar0=7).compute_solution(t_span=np.linspace(7, 8, 11))[[
i_w, omega_s]]
21     num_9 = sym_sol.subs(dane0).with_ics([num_8.iloc[-1][i_w], num_8.iloc
[-1][omega_s]], ivar0=8).compute_solution(t_span=np.linspace(8, 9, 11))[[
i_w, omega_s]]
22     num_10 = sym_sol.subs(dane1).with_ics([num_9.iloc[-1][i_w], num_9.iloc
[-1][omega_s]], ivar0=9).compute_solution(t_span=np.linspace(9, 10, 11))[[
i_w, omega_s]]
23
24     num_sol = pd.concat([num_1, num_2.set_index(pd.Series(np.linspace(1,2,11)
)), num_3.set_index(pd.Series(np.linspace(2,3,11))), num_4.set_index(pd.
Series(np.linspace(3,4,11))), num_5.set_index(pd.Series(np.linspace
(4,5,11))), num_6.set_index(pd.Series(np.linspace(5,6,11))), num_7.
set_index(pd.Series(np.linspace(6,7,11))), num_8.set_index(pd.Series(np.
linspace(7,8,11))), num_9.set_index(pd.Series(np.linspace(8,9,11))),
num_10.set_index(pd.Series(np.linspace(9,10,11)))])
25     num_sol=num_sol[-num_sol.duplicated()]
26
27     return num_sol
28
29     %

```

Main procedure implemented in a loop:

```

1     def han_solution_loop():
2
3         dane0 = {U_z: 10, R_w: 2, L_w: 0.1, k_e: 0.1, k_m: 0.1, J: 0.1, B: 0.5,
M_1: 0}
4         dane1 = {U_z: 10, R_w: 2, L_w: 0.1, k_e: 0.1, k_m: 0.1, J: 0.1, B: 0.5,
M_1: 0.2}
5         dane2 = {U_z: 0, R_w: 2, L_w: 0.1, k_e: 0.1, k_m: 0.1, J: 0.1, B: 0.5,
M_1: 0.2}
6         dane3 = {U_z: 0, R_w: 2, L_w: 0.1, k_e: 0.1, k_m: 0.1, J: 0.1, B: 0.5,
M_1: 0}
7
8         dcmotor=DCMotor
9         ode = ODESystem(odes=dcmotor()._eoms, dvars=Matrix([dcmotor.i_w, dcmotor.
omega_s]), ode_order=1)
10
11         sym_sol = ode.solution
12
13         data_list=[dane0,dane1,dane2,dane3]
14         data_list_select=0
15         ics_list=[[0,0]]
16         num_list=[]

```

```

17
18     for sol_iter in range(0,10):
19
20         num=sym_sol.subs(data_list[data_list_select]).with_ics(ics_list[-1],
21         ivar0=sol_iter).compute_solution(t_span=np.linspace(sol_iter,sol_iter
22         +1,11))[[i_w,omega_s]]
23         data_list_select+=1
24
25         if data_list_select ==4:
26             data_list_select=0
27
28         ics_list.append([num.iloc[-1][i_w], num.iloc[-1][omega_s]])
29         num_list.append(num.set_index(pd.Series(np.linspace(sol_iter,sol_iter
30         +1,11))))
31
32     num_sol=pd.concat(num_list)
33     num_sol=num_sol[-num_sol.duplicated()]
34
35     return num_sol
36
37 %

```

References

1. GitHub. Bogumilchilinski/Dynpy. Available online: <https://github.com/bogumilchilinski/dynpy> (accessed on 4 May 2024).
2. Guo, Y.; Liu, L.; Ba, X.; Lu, H.; Lei, G.; Yin, W.; Zhu, J. Designing High-Power-Density Electric Motors for Electric Vehicles with Advanced Magnetic Materials. *World Electr. Veh. J.* **2023**, *14*, 114. [\[CrossRef\]](#)
3. Martinez, S.S.; Samaras, C. Electrification of Transit Buses in the United States Reduces Greenhouse Gas Emissions. *Environ. Sci. Technol.* **2024**. [\[CrossRef\]](#) [\[PubMed\]](#)
4. Siddhartha, P.; Sujeeth, T.; Shiva, B.; Ramprabhakar, J. Integration Of Renewable Energy Sources With Power Management Strategy For Effective Bidirectional Vehicle To Grid Power Transfer. *Procedia Comput. Sci.* **2023**, *218*, 9–23. [\[CrossRef\]](#)
5. Kebede, A.A.; Kalogiannis, T.; Mierlo, J.V.; Berecibar, M. A comprehensive review of stationary energy storage devices for large scale renewable energy sources grid integration. *Renew. Sustain. Energy Rev.* **2022**, *159*, 112213. [\[CrossRef\]](#)
6. Mathew, M.S.; Kolhe, M.L.; Kandukuri, S.T.; Omlin, C.W. Data driven approach for the management of wind and solar energy integrated electrical distribution network with high penetration of electric vehicles. *J. Clean. Prod.* **2023**, *421*, 138467. [\[CrossRef\]](#)
7. Mackojć, A.; Chyliński, B. Preliminary modelling methodology of a coupled payload-vessel system for offshore lifts of light and heavyweight objects. *Bull. Pol. Acad. Sci. Tech. Sci.* **2021**, *70*, 139003. [\[CrossRef\]](#)
8. Chilinski, B.; Mackojc, A.; Mackojc, K. Analytical solution of parametrically induced payload nonlinear pendulation in offshore lifting. *Ocean. Eng.* **2022**, *259*, 111835. [\[CrossRef\]](#)
9. Mackojc, A.; Chilinski, B.; Zalewski, R. Preliminary research of a symmetrical controllable granular damper prototype. *Bull. Pol. Acad. Sci. Tech. Sci.* **2022**, *70*, 141002. [\[CrossRef\]](#)
10. Radomski, A.P.; Sierociński, D.J.; Chyliński, B.D. Proposition of a structural health monitoring model for a concept of an innovative variable mass pendular tuned mass damper. *Diagnostyka* **2024**, *25*, 1–10. [\[CrossRef\]](#)
11. Hassan, M.S.; Safwat, I.M.; Abounemra, A.M.E.; Ahmed, H.N. A generalized closed-form analytic design technique for solid-state Marx generators. *Int. J. Circuit Theory Appl.* **2024**, *52*, 4026–4039. [\[CrossRef\]](#)
12. Wang, B.; Zhang, C.; Xia, Y.; Zhang, F. Analytical solution of serial ferroresonance triggered by circuit breaker operation in solidly grounded power grids. *Int. J. Electr. Power Energy Syst.* **2023**, *154*, 109447. [\[CrossRef\]](#)
13. Fevzi Çakmak, Z.A.; Tür, M.R. Analysis of Open Circuit Voltage MPPT Method with Analytical Analysis with Perturb and Observe (P&O) MPPT Method in PV Systems. *Electr. Power Compon. Syst.* **2023**, *52*, 1528–1542. [\[CrossRef\]](#)
14. Alqhtani, M.; Saad, K.M.; Shah, R.; Hamanah, W.M. Discovering novel soliton solutions for (3+1)-modified fractional Zakharov–Kuznetsov equation in electrical engineering through an analytical approach. *Opt. Quantum Electron.* **2023**, *55*, 1149. [\[CrossRef\]](#)
15. Valbuena Godoy, J.; Negri, S.; Oliva, F.; Antoniazzi, A.; Faranda, R.S. Innovative Fault Current Evaluation Method for Active DC Grids. *Electronics* **2024**, *13*, 847. [\[CrossRef\]](#)
16. Zhou, P.; Ghabeli, A.; Ainslie, M.; Grilli, F. Characterization of flux pump-charging of high-temperature superconducting coils using coupled numerical models. *Supercond. Sci. Technol.* **2023**, *36*, 115002. [\[CrossRef\]](#)
17. Du, Y.; Qiao, J.; Wu, J.; Xu, Y.; Nan, T.; Dong, S.; Hu, Z.; Liu, M. Performance analysis of acoustically actuated magnetoelectric antennas via equivalent circuit method. *J. Appl. Phys.* **2024**, *135*, 114101. [\[CrossRef\]](#)

18. Arnold, M. Numerical methods for simulation in applied dynamics. In *Simulation Techniques for Applied Dynamics*; Arnold, M., Schiehlen, W., Eds.; Springer: Vienna, Austria, 2009; pp. 191–246. [\[CrossRef\]](#)
19. Tatjewski, P. *Numerical Methods*, 1st ed.; Oficyna Wydawnicza Politechniki Warszawskiej: Warsaw, Poland, 2015.
20. Wang, Y.; Zhao, G. A comparative study of fractional-order models for lithium-ion batteries using Runge Kutta optimizer and electrochemical impedance spectroscopy. *Control. Eng. Pract.* **2023**, *133*, 105451. [\[CrossRef\]](#)
21. Baron, B.; Kolanska-Pluska, J.; Łukaniszyn, M.; Spałek, D.; Kraszewski, T. Solution of nonlinear stiff differential equations for a three-phase no-load transformer using a Runge-Kutta implicit method. *Arch. Electr. Eng. (Online)* **2022**, *71*, 1081–1106.
22. Yousri, D.; Mudhsh, M.; Shaker, Y.O.; Abualigah, L.; Tag-Eldin, E.; Abd Elaziz, M.; Allam, D. Modified Interactive Algorithm Based on Runge Kutta Optimizer for Photovoltaic Modeling: Justification Under Partial Shading and Varied Temperature Conditions. *IEEE Access* **2022**, *10*, 20793–20815. [\[CrossRef\]](#)
23. Malarvizhi, M.; Karunanithi, S. Study of electrical circuits using Runge Kutta method of order 4. *J. Comput. Math.* **2021**, *5*, 109–120.
24. Teoh, A.K. Transient Analysis of Electrical Circuits Using Runge-Kutta Method. 2017. Available online: <https://api.semanticscholar.org/CorpusID:86832491> (accessed on 10 October 2024).
25. Maffezzoni, P.; Codecasa, L.; D’Amore, D. Time-Domain Simulation of Nonlinear Circuits Through Implicit Runge-Kutta Methods. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2007**, *54*, 391–400. [\[CrossRef\]](#)
26. Suhag, A. Transient analysis of electrical circuits using Runge-Kutta method and its application. *Int. J. Sci. Res. Publ.* **2013**, *3*, 486–490.
27. Patil, M.B.; Hari, V.V.S.P.K. Circuit simulation using explicit methods. *arXiv* **2023**, arXiv:2301.04595.
28. Fadnavis, S. Round off Error propagation in simulation of RC circuit Subtitle: Simulation of RC circuit. In Proceedings of the International Conference on Advances in Computing and Information Technology–ACIT 2014, Bangkok, Thailand, 4–5 January 2014.
29. Kazmierski, T.J.; Wang, L.; Al-Hashimi, B.M.; Merrett, G.V. An Explicit Linearized State-Space Technique for Accelerated Simulation of Electromagnetic Vibration Energy Harvesters. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2012**, *31*, 522–531. [\[CrossRef\]](#)
30. MOHAMAD ZAKI, F.A.; Isa, S. Numerical Solutions of Applications in Ordinary Differential Equation by Implicit Multistep Method. *Enhanc. Knowl. Sci. Technol.* **2022**, *2*, 500–507.
31. Kuznyetsov, O. Modeling of electromechanical systems based on induction machine using method of average voltage on the integration step length. *Sci. Bull. Ser. C Electr. Eng. Comput. Sci.* **2017**, *79*, 169–180.
32. Adeniran, O. Numerical solution of some selected RLC circuits. *Sospoly J. Sci. Agric.* **2017**, *2*, 1–11.
33. Tural-Polat, S.N.; Dincel, A.T. Wavelet methods for fractional electrical circuit equations. *Phys. Scr.* **2023**, *98*, 115203. [\[CrossRef\]](#)
34. Avci, I. Spectral collocation with generalized Laguerre operational matrix for numerical solutions of fractional electrical circuit models. *Math. Model. Numer. Simul. Appl.* **2024**, *4*, 110–132. [\[CrossRef\]](#)
35. Doménech-Asensi, G.; Kazmierski, T.J. Generation of new power processing structures exploiting genetic programming. In Proceedings of the 2017 IEEE 26th International Symposium on Industrial Electronics (ISIE), Edinburgh, UK, 19–21 June 2017; pp. 729–732. [\[CrossRef\]](#)
36. Sels, T.; Craenenbroeck, T.V.; Hameyer, K.; Mertens, R.; Declercq, J. Comparison of a Combined Time-Harmonic—Transient Finite Element Analysis of Fast Transient Oscillations with Laboratory Measurements. 2000. Available online: <https://api.semanticscholar.org/CorpusID:125250908> (accessed on 7 October 2024).
37. Condon, M.; Dautbegovic, E.; Brazil, T. An Efficient Numerical Algorithm for the Transient Analysis of High-Frequency Non-Linear Circuits. In Proceedings of the 2002 32nd European Microwave Conference, Milan, Italy, 23–26 September 2002. [\[CrossRef\]](#)
38. Cai, Y.; Zhang, J.; Yu, W. A Predictor-Corrector Method for Power System Variable Step Numerical Simulation. *IEEE Trans. Power Syst.* **2019**, *34*, 3283–3285. [\[CrossRef\]](#)
39. Kurobe, H.; Sekine, T.; Asai, H. Predictor-corrector latency insertion method for fast transient analysis of ill-constructed circuits. In Proceedings of the 17th Asia and South Pacific Design Automation Conference, Sydney, Australia, 30 January–2 February 2012; pp. 365–370.
40. Noda, T.; Yamamoto, K.; Nagaoka, N.; Ametani, A. A predictor-corrector scheme for solving a nonlinear circuit. In Proceedings of the IPST 1997, Seattle, WA, USA, 22–26 June 2016; Volume 95, pp. 5–10.
41. Tripodi, E.; Musolino, A.; Rizzo, R.; Raugi, M. A new predictor-corrector approach for the numerical integration of coupled electromechanical equations. *Int. J. Numer. Methods Eng.* **2016**, *105*, 261–285. [\[CrossRef\]](#)
42. Aadithya, K.; Keiter, E.; Mei, T. Predictor/Corrector Newton-Raphson (PCNR): A Simple, Flexible, Scalable, Modular, and Consistent Replacement for Limiting in Circuit Simulation. In *Scientific Computing in Electrical Engineering*; Springer: Cham, Switzerland, 2020; pp. 201–208. [\[CrossRef\]](#)
43. Abdulsalam, S.; Xu, W.; Dinavahi, V. Modelling and simulation of three-phase transformers for inrush current studies. *Gener. Transm. Distrib.* **2005**, *152*, 328–333. [\[CrossRef\]](#)

44. Bayrak, M.A.; Can, E. Numerical Solution of Fuzzy Differential Equations by Milne'S Predictor-Corrector Method. *Math. Sci. Appl. E-Notes* **2015**, *3*, 137–153. [\[CrossRef\]](#)
45. Sowa, M. A subinterval-based method for circuits with fractional order elements. *Bull. Pol. Acad. Sci. Tech. Sci.* **2014**, *62*, 449–454. [\[CrossRef\]](#)
46. Virtanen, P.; Gommers, R.; Oliphant, T.E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nat. Methods* **2020**, *17*, 261–272. [\[CrossRef\]](#)
47. Semenova, M.; Kazazaeva, D.; Sabychikova, A. Simulation and analysis of transition processes in linear electrical circuits using MatLab software package. *E3s Web Conf.* **2023**, *389*, 1045. [\[CrossRef\]](#)
48. Khanahmedova, S. Investigation of Some Electrical Circuits Using Simulation in Matlab/Simulink. Available online: <https://cyberleninka.ru/article/n/investigation-of-some-electrical-circuits-using-simulation-in-matlab-simulink/viewer> (accessed on 8 October 2024).
49. Mladenović, V.M.; Porrat, D.; Lutovac, M.D. Simulation of OFDM transmitters and post processing with SchematicSolver and Mathematica as a computer algebra system. In Proceedings of the Papers 5th European Conference on Circuits and Systems for Communications (ECCSC'10), Belgrade, Serbia, 23–25 November 2010; pp. 278–281.
50. Yadir, S.; Aazou, S.; Maouhoub, N.; Rais, K.; Benhmida, M.; Assaid, E. Illuminated solar cell physical parameters extraction using mathematica. In Proceedings of the 2009 IEEE/ACS International Conference on Computer Systems and Applications, Rabat, Morocco, 10–13 May 2009; pp. 63–64.
51. MATLAB. Specialized Power Systems. Available online: https://www.mathworks.com/help/sps/specialized-power-systems.html?s_tid=CRUX_lftnav (accessed on 3 May 2024).
52. Modeler, W.S. Modelica: Electrical. Available online: <https://reference.wolfram.com/system-modeler/libraries/Modelica/Modelica.Electrical.html> (accessed on 3 May 2024).
53. Asadi, F. Electric and Electronic Circuit Simulation using TINA-TI. In *Electric and Electronic Circuit Simulation Using TINA-TI*; River Publishers: Aalborg, Denmark, 2022.
54. Pérez Martínez, M.; Ramos Guardarrama, J.; Santos Baranda, J.; Silvério Freire, R.C. Use of Scilab software as a didactic tool in electrical circuits laboratory practices. *Ing. Energética* **2022**, *43*, 71–79.
55. CircuitLab. Online Circuit Simulator & Schematic Editor. Available online: <https://www.circuitlab.com/> (accessed on 3 May 2024).
56. Boys, B.; Dodwell, T.; Hobbs, M.; Girolami, M. PeriPy—A high performance OpenCL peridynamics package. *Comput. Methods Appl. Mech. Eng.* **2021**, *386*, 114085. [\[CrossRef\]](#)
57. Lutz, M. *Programming Python*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2001.
58. Oliphant, T. Python for Scientific Computing. *Comput. Sci. Eng.* **2007**, *9*, 10–20. [\[CrossRef\]](#)
59. Meurer, A.; Smith, C.P.; Paprocki, M.; Čertík, O.; Kirpichev, S.B.; Rocklin, M.; Kumar, A.; Ivanov, S.; Moore, J.K.; Singh, S.; et al. SymPy: Symbolic computing in Python. *PeerJ Comput. Sci.* **2017**, *3*, e103. [\[CrossRef\]](#)
60. Kluyver, T.; Ragan-Kelley, B.; Pérez, F.; Granger, B.; Bussonnier, M.; Frederic, J.; Kelley, K.; Hamrick, J.; Grout, J.; Corlay, S.; et al. Jupyter Notebooks—A publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*; Loizides, F., Schmidt, B., Eds.; IOS Press: Amsterdam, The Netherlands, 2016; pp. 87–90.
61. Dyba, T.; Dingsoyr, T. What Do We Know about Agile Software Development? *IEEE Softw.* **2009**, *26*, 6–9. [\[CrossRef\]](#)

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.