Multibody System Dynamics
https://doi.org/10.1007/s11044-023-09962-0

RESEARCH

Multibody Models Generated from Natural Language

Johannes Gerstmayr[1] · **Peter Manzl**[1] · **Michael Pieber**[1]

Received: 3 November 2023 / Accepted: 19 December 2023
© The Author(s) 2024

Abstract
Computational models are conventionally created with input data, script files, programming interfaces, or graphical user interfaces. This paper explores the potential of expanding model generation, with a focus on multibody system dynamics. In particular, we investigate the ability of Large Language Model (LLM), to generate models from natural language. Our experimental findings indicate that LLM, some of them having been trained on our multibody code Exudyn, surpass the mere replication of existing code examples. The results demonstrate that LLM have a basic understanding of kinematics and dynamics, and that they can transfer this knowledge into a programming interface. Although our tests reveal that complex cases regularly result in programming or modeling errors, we found that LLM can successfully generate correct multibody simulation models from natural-language descriptions for simpler cases, often on the first attempt (zero-shot).

After a basic introduction into the functionality of LLM, our Python code, and the test setups, we provide a summarized evaluation for a series of examples with increasing complexity. We start with a single mass oscillator, both in SciPy as well as in Exudyn, and include varied inputs and statistical analysis to highlight the robustness of our approach. Thereafter, systems with mass points, constraints, and rigid bodies are evaluated. In particular, we show that in-context learning can levitate basic knowledge of a multibody code into a zero-shot correct output.

Keywords Large language models · Text-based simulation · Modeling · Natural language

1 Introduction

The current state-of-the-art for creation of multibody dynamics simulation models can be classified into graphics user interface (GUI)-based methods, such as with commercial codes Adams (Hexagon), or RecurDyn (FunctionBay), specialized input files or script languages [9, 17] or using the underlying programming language of the simulation code [10, 18].

Natural Language Processing (NLP), an integral part of artificial intelligence, empowers computers to comprehend, interpret, and generate human language [15]. It explores diverse aspects of language, such as syntax – the structure of word arrangement in sentences – and semantics, which focuses on the meaning of words in context. NLP has been impacted by

---

✉ J. Gerstmayr

[1] Department of Mechatronics, Universität Innsbruck, Technikerstrasse 13, Innsbruck, 6020, Tyrol, Austria

Published online: 17 January 2024

the introduction of LLM, particularly the transformer architecture [25]. These LLMs, like Generative Pre-trained Transformer (GPT), have a massive number of parameters and have shown capabilities beyond NLP, such as code completion and generation [1]. Furthermore, they have potential in transforming natural-language descriptions into programmed simulation models.

Recent developments show an upward trend in the size of LLM, see also Fig. 1. Kaplan et al. show in their study on scaling laws that the crossentropy loss correlates with dataset size, computational power, and the number of parameters [16]. The research team of Google Deepmind found that training data should be scaled together with the model size [12]. Notably, GPT-3 showcased the improved few-shot performance of LLM [3], and public attention peaked with the introduction of ChatGPT in 2022. A competing chat system based on LLM is Google Bard, which initially used Language Model for Dialogue Applications (LaMDA) [24], a 137-billion parameter model. As of May 2023, Bard has transitioned to PaLM-2, whose dimensions remain undisclosed but are smaller than its predecessor PaLM, which had 540 billion parameters, due to compute-optimal scaling of parameter size. Chowdhery et al. [5] noted that in the development of their 540-billion parameter Pathways Language Model (PaLM) the effect of the model scale does not seem to be saturated yet. According to a leak the GPT-4 model has 1760 billion parameters, but it is not officially disclosed.

A key challenge for LLM development is the curation and preparation of datasets. For instance, GPT-3 utilized a diverse mix of sources, including WebText2 and Wikipedia, totaling $499 \cdot 10^9$ tokens [3]. Similarly, PaLM was trained on $780 \cdot 10^9$ tokens from various sources, such as Github and social-media conversations [5]. Ensuring test-tasks are not part of the training dataset is crucial to prevent dataset contamination.

Due to the large amount of data required for training of LLM, HuggingFace has emerged as a platform in the world of Large Language Models (LLM), offering a rich repository of training datasets essential for LLM development [28]. The more than 69k datasets, ranging from tiny to huge ones, include comprehensive sets such as Wikipedia as well as GitHub.

1.1 LLM in science and engineering

Specialized LLM have been created, e.g., for science, see Galactica [23], for software engineering [13], or a method focusing on micromechanics, see Ref. [4]. SciBERT [2], based on a pretrained BERT (Bidirectional Encoder Representations from Transformers) [6] base model, has been specifically trained on scientific texts. SciBERT has been trained on a broad corpus of scientific publications and therefore outperforms BERT and surpasses previous models in several categories. Notably, models like BERT and SciBERT have a relatively compact size, allowing for their training without the need for computers with petaFLOPS range performance. Furthermore, the BERT model has been expanded into MatSciBERT [11], a Language Learning Model (LLM) specifically focused on materials science.

1.2 Aims and methods

This paper introduces an approach to generate (multibody) dynamics simulation models using natural language. We briefly mention GPT and their current status, relying to open-source models and closed models as far as data is accessible. Our exploration seeks to understand the capabilities of LLM that have been trained on codes, in particular simulation codes to showcase the current capabilities of existing LLMs in reflecting future advancements. We provide a brief overview of the Exudyn Python interface for setting up simulation models

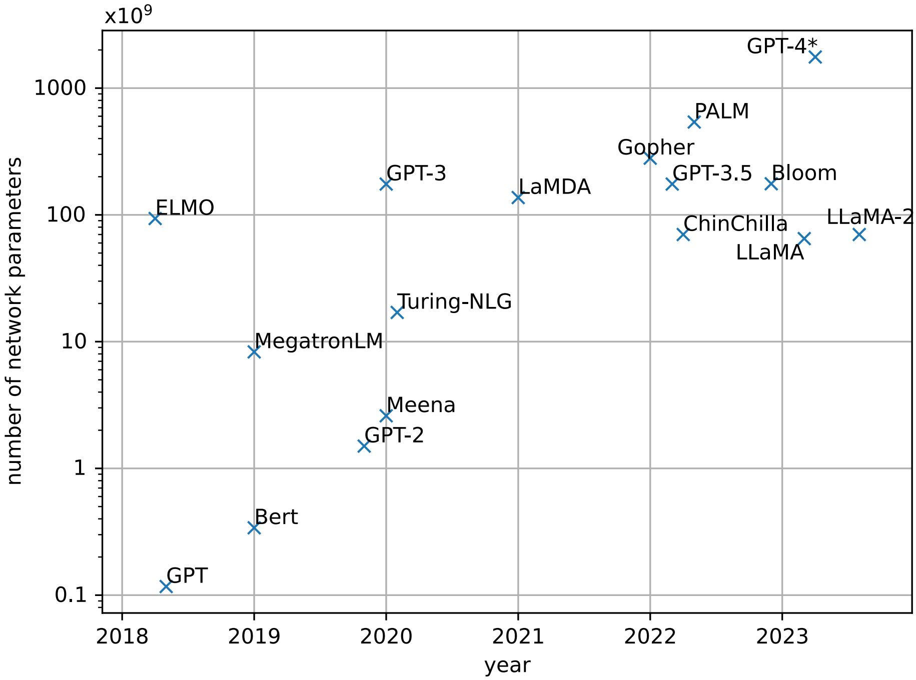

Fig. 1 Number of parameters of LLM over the past five years. Significant advances were made by Megatron in 2019 and GPT-3 in 2020. *The number of parameters of GPT-4 is not officially disclosed but was presumably leaked

and its status at the time that training data had been created, which is September 2021. To validate our approach, we present a dedicated test set and evaluate its performance across various LLM, with full-text responses available in the supplementary material. Using LLM to create Python code for geometrical modeling has been applied, for example, with Blender [7]. In the present paper, we only focus on evaluation of existing LLM. We apply in-context learning [22], which is widely accepted as a means to improve the performance of LLM, but do not attempt full-scale training or fine tuning of open-source LLM.

1.3 Mechanical and simulation models

It should be mentioned that the creation of simulation models also means that a mechanical model is underlying the simulation model, which is why the term (mechanical) model is used in the following as a synonym for both kinds of simulation and mechanical models. To produce a (mechanical) model from natural language, an LLM needs more than just general NLP capabilities. It also requires foundational knowledge in mechanical engineering, including geometry, kinematics, statics, and dynamics. Training an LLM solely on the documentation of a simulation code would not be sufficient.

2 Brief notes on LLM and transformers

In this research we look into different LLM, being GPT-3.5 and GPT-4 [19], which are proprietary models developed by OpenAI, well known by the ChatGPT application.[1] Furthermore, Google's Bard, which currently uses PaLM-2,[2] as well as LLaMA-2[3] from Meta AI are considered in the present paper.

There are only technical reports on GPT-3.5, GPT-4, and PaLM-2 with little information on the underlying model. Many more details are available for LLaMA, which is why we focus here on more detailed descriptions of LLaMA and LLaMA-2. However, the basic transformer structure is the same for all models used here, as far as technical reports disclose.

2.1 Transformers and large language models

The transformer architecture is a neural-network design introduced in 2017 [25], primarily for sequence-to-sequence tasks in natural-language processing. The evolution of transformers departs from recurrent neural networks and long short-term memory (LSTM)-based models by relying entirely on attention mechanisms to draw global dependencies between input and output. The architecture consists of an encoder and/or a decoder, each comprising multiple layers of self-attention and feedforward neural networks. The key innovation, attention [25], allows the model to weigh the significance of different words in a sequence, enabling it to capture long-range dependencies and context effectively. This design has become foundational in the subsequent development of large-scale language models and is shown in the originally proposed architecture in Fig. 2.

Transformers have then been further developed, such as the early model BERT [6, 21] and its subsequent iterations. Generative Pre-trained Transformers (GPT) are a subset of models specifically designed for language tasks. Their core structure is the transformer, which uses a decoder-only variant. Characterized by their massive number of parameters, usually between a billion and a trillion, these models excel in generating coherent, contextually relevant text over lengthy passages. Their foundational neural networks, which are key elements in the various attention heads in transformers, encompass tens to hundreds of millions of parameters, necessitating extensive training data.

The central feature of the transformer is the self-attention mechanism, which enables a word to compute its context by checking the relevance of all other words in the sentence, see Fig. 2. This is done using Query (Q), Key (K), and Value (V) vectors. Herein, the query represents the current word or token one attention mechanism is focusing on and determines how much attention should be paid to other tokens. The key represents all tokens in the input and scores each token's relevance to the current focus. The value provides the content from the input tokens, which, when weighted by the attention scores, gives the output for the current token. In the multi-head attention mechanism of the transformer, multiple attention heads work in parallel, each producing its own output. These outputs are then concatenated and linearly transformed to produce the final output. The model also includes positional encodings to understand the sequence of words, as transformers do not inherently recognize order. The "pretrained" aspect means that GPT models are first trained on vast amounts of text to predict the next word in a sequence. This foundational training equips them with knowledge about grammar, context, and general information. They can be further fine-tuned

---

[1] https://help.openai.com/en/articles/6825453-chatgpt-release-notes.

[2] https://bard.google.com/updates.

[3] https://huggingface.co/spaces/ysharma/Explore_llamav2_with_TGI.

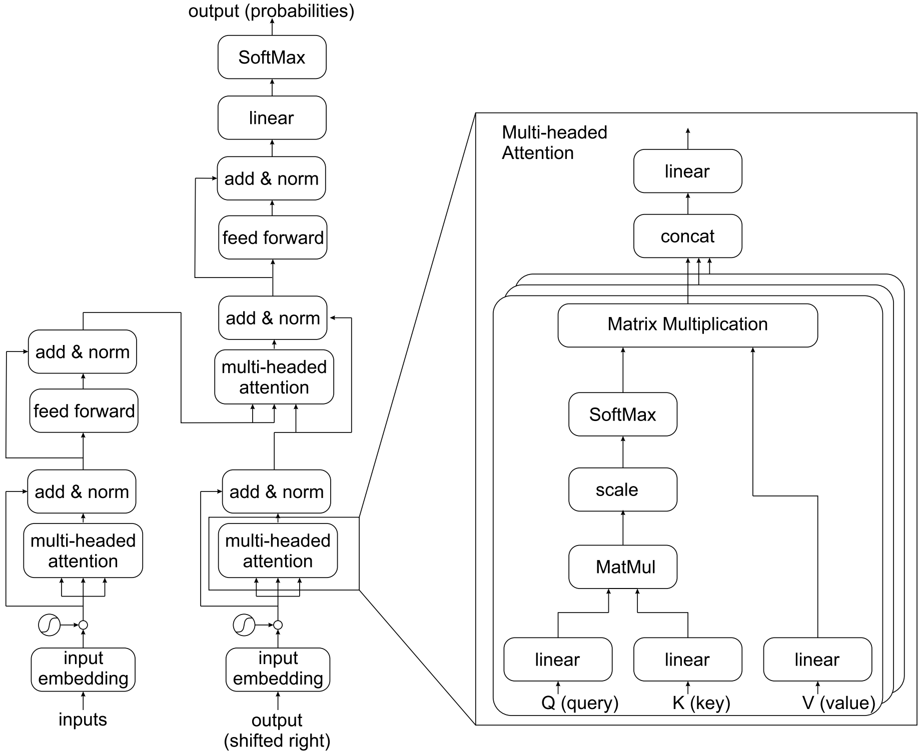

Fig. 2 The original transformer architecture and the operations performed in the multi-headed attention block [25]

for specific tasks, such as chat or code completion. To generate text, a GPT inputs a user-defined text (prompt) and predicts subsequent words in sequence.

In the context of programming, LLM can assist developers by predicting the next lines of code, suggesting code optimizations, or even generating entire code snippets based on a given prompt. LLM showed enormous capabilities in code completion and code generation rather early [1]. By training on vast repositories of code, including code examples, documentation, issue trackers, bug reports, discussion forums, and Q&A pages, these models have gained an understanding of various programming languages and their idiomatic patterns. This not only accelerates the coding process [20] but also aids in reducing bugs and enhancing code quality. If all code resources are available on open-source platforms, such as GitHub, larger LLM may have already performed training on that code.

In addition to direct code completion, LLM have demonstrated the ability to translate natural-language descriptions into functional code. This means that a developer can provide a plain English request, such as “create a function that calculates all prime numbers smaller than 20 in C++,” and the model can generate the corresponding code. This capability bridges the gap between domain experts without coding expertise and software development, enabling more intuitive and collaborative software-design processes.

In particular, LLM may also be trained with the repository of a simulation code, including documentation, the code itself, as well as code examples. However, in order to generate a simulation model from natural language, besides general NLP capabilities, the LLM also requires some mechanical-engineering knowledge such as geometry, kinematics, statics or dynamics. Therefore, particular training of LLM solely with the documentation of a simulation code could be insufficient.

2.2 The LLaMA model

LLaMA-2, introduced in July 2023 as a collaboration between Meta AI and Microsoft, represents a further refinement of LLaMA-1, often denoted as LLaMA.[4] It was launched in three model sizes: 7B, 13B, and 70B parameters. While retaining the architecture of LLaMA-1, LLaMA-2 boasted a 40% increase in training data, with its foundational models being trained on a 2-trillion token dataset. This dataset was curated to omit sites that might disclose personal information and emphasized the inclusion of trustworthy sources and allows a maximum context length of 4096 tokens. While LLaMA-2 is freely available for many commercial applications, debates persist regarding its status as open source. LLaMA is based on the transformer architecture, which has been a cornerstone for language models since 2018. Notably, in contrast to GPT-3, LLaMA uses the SwiGLU activation function instead of ReLU, adopts rotary positional embeddings over absolute positional embedding, and applies root-mean-squared layer normalization.

To operate LLaMA-2, sufficient memory, either through RAM or VRAM on a GPU, is essential for model inference (obtain the output from a trained model). The 7-billion parameter model demands 14 GB since the parameters are in half-precision floating-point format (16 bit). The larger models, with 13 billion and 70 billion parameters, require 26 GB and 138 GB, respectively. It is worth noting that these memory requirements are for inference; training demands more memory to store the gradients.

In addition to the vast memory requirements, as previously mentioned, the computational resources for training or fine tuning are also greater than most researchers have available. For LLM, the number of operations needed for training is also called compute or training compute and is often given in floating-point operations (FLOPS), GPU-hours, or petaFLOPS days as the number of operations are becoming very high. The state-of-the-art model LLaMA-2 70B, which is still too small to solve the problems mentioned in the present paper, required 1.7 million GPU hours of training and very large data sources, requiring at least a cluster with 1000 GPUs and appropriate data-transfer rates in order to perform similar training tasks. For GPT-3 approximately $3.14 \cdot 10^{23}$ FLOPS or 3634 petaFLOPS – days of compute were used. This leads to the conclusion that training an existing model is fully out of sight for the present research.

An alternative approach is Low-Rank Adaption (LoRA) [14] of LLM. With LoRA the weights of the pretrained model are frozen and additional trainable rank decomposition matrices are injected into each layer of the transformer architecture. The authors claimed that this approach reduced the trainable parameters by 10 000 times and the GPU memory requirement by a factor of 3 for the 175-billion version of the GPT-3 model with similar performance than full fine tuning.

3 Multibody simulation models built from Python code

Since 2019, the Python library Exudyn [10] has been developed for creation and simulation of multibody system models. Several popular LLM have been trained occasionally on this

---

[4] https://github.com/facebookresearch/llama.

code and have some basic capabilities to create models from natural language. In particular, the training date of our evaluated versions of GPT-3.5 and GPT-4 ended in September 2021, which is of particular interest for subsequent investigations.

The Exudyn repository has been uploaded on GitHub and made publicly available on January 9, 2020. On September 14, 2021 (Exudyn version 1.1.0), the library had only 4 forks and 15 stars, the latter being a common criterion to select repositories from GitHub for training. OpenAI (GPT-3.5 and GPT-4) does not provide information about which GitHub repositories have been used for training. In the literature on different LLM focusing on code, rather large thresholds for number of GitHub stars are mentioned, being at least 50 stars for the dataset used by PolyCoder [29]. An explanation for several LLM, as shown later, to chose Exudyn for training may be the completely open and clearly stated BSD-3 license, a larger set of files as compared to many higher-ranked repositories and a large number of annotated examples. In particular, a significant amount of documentation, throughout code commenting and file headers may have further influenced the decision to train on this code, as this information is essential in the context of code completion. The highly structured data and widely documented code in the repository included 88 examples and 56 test models in version 1.1.0 may have been a further reason to be chosen for training by several LLM.[5]

3.1 Setup of models in Exudyn

Exudyn models are created solely using the Python language. Python is also available in other multibody codes, such as ProjectChrono [18] or PyDy [8]. As a difference from the latter codes, Exudyn includes a large set of annotated examples. Furthermore, the setup of a rigid-body model, studied within the present paper, follows a simple and systematic approach based on redundant coordinates and constraints.

After import of basic Python libraries, a simulation model is setup by creating a new system, usually denoted as `mbs`. Hereafter, different items are added, such as nodes, objects, markers, loads, and sensors. Nodes are added for definition of (unknown) kinematic quantities. Computational objects are then added to represent bodies, connectors or joint constraints. The relation between nodes and bodies is rather simple, such as mass points requiring point nodes or rigid bodies requiring rigid-body nodes, e.g., based on Euler parameters. Connectors, such as spring-dampers as well as constraints are attached to markers, which need to be attached to bodies or nodes. Hereafter, a spring-damper is created by providing two markers, a spring constant, a damping constant, and a reference length.

All of the items can be added to the `mbs` by using simple commands such as `n=mbs.AddNode(...)` or `o=mbs.AddObject(...)`, in which `n` is the returned node index and `o` is the according object index. Finally, the structure for definition of a node or object is embedded into a class structure, such that a mass point with 5 kg is attached to node `n` by writing:

```
o=mbs.AddObject(MassPoint(nodeNumber = n, physicsMass=5))
```

Finally, the according definition of a multibody system highly depends on the geometry, joint constraints, inertia, and mechanical parameters. In order to create the Python code, the underlying mechanical model needs to be known by the LLM.

The subsequent functions for performing a transient simulation are straightforward and are well reproduced by state-of-the-art LLM. The basic steps for starting the simulation are:

---

[5] In view of further development and training, it should be noted that there are currently 152 examples and 93 test models in Exudyn version 1.7.0. With 114 stars on GitHub, a future training could lead to significantly improved results.

1) finalizing the multibody system: `mbs.Assemble();`
2) setting up simulation settings: `sims = exu.SimulationSettings();`
3) adjustment of the simulation settings parameters;
4) calling the solver: `mbs.SolveDynamic(sims).`

There are some variants such as static solvers as well as special explicit and implicit solver types, not mentioned here. Most of the examples also include commands to add visualization and to start the 3D visualization during or after simulation, which is therefore also reproduced by LLM.

3.2 In-context learning

Even though the approach in Exudyn is highly systematic and versatile, it seems to be more complicated than in other multibody simulation software. One reason for the higher complexity is the availability of flexible bodies, such as beams or modally reduced bodies, which require different approaches, e.g., to represent finite-element nodes. As a main difference from other Python modeling codes that most LLM have been trained on, we mention that rigid bodies require underlying nodes, and cannot be created by one single function. Furthermore, joints are attached to markers, but cannot be directly attached to bodies, similar to loads. This systematic, but more complex approach often leads to wrong assumptions by LLM (and also confuses human users). As a solution, simplified functions have been added to Exudyn since May 2023, all of them available in version 1.7.0. Simplified functions obtain a prefix `Create` and can be directly called for the multibody system, such as `mbs.CreateRigidBody(...)`, which adds a rigid body to the system with simple arguments and also allows to add gravity without defining loads or markers.

In order to provide the information on the simplified commands, in-context learning is used in many of our tests. The following listing shows the first 22 lines of the information, later denoted as context information for mass points, which are pasted into the chat prompt at the beginning of each session:

```
# Use the following information as an update of the internal data of the Python multibody code Exudyn.
# See the following examples to create multibody systems in Exudyn.
# NOTE: mbs.Create...(...) calls several functions in the background to create nodes, objects, markers and loads in Exudyn.
# all quantities are giving as 3D lists [x,y,z] for positions, velocities, ....
# rotations are usually given as rotation matrix (numpy array);
# RotationVector2RotationMatrix([rotX, rotY, rotZ]) computes a rotation around the global x,y,z rotation axis

# create rigid bodies and mass points with distance constraint and joints
import exudyn as exu
from exudyn.utilities import * #includes itemInterface, graphicsDataUtilities and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem() #create a MainSystem ’mbs’ to work with

# create a simple mass point with initial velocity
m1 = mbs.CreateMassPoint(referencePosition=[1,-1,0], initialVelocity = [2,5,0], physicsMass=1, drawSize = 0.2)

# add a ground object:
```

Table 1 Overview of the used LLM. References are given to release notes and homepages and an asterisk* marks undisclosed values. The number of parameters are only officially known for GPT-3 as well as LLaMA-2. The parameters for GPT-4 are from discussion forums and unofficial leaks. The exact number of parameters and max. tokens for Bard are according to PaLM-2, as it is said that Bard is powered by PaLM-2

Model	Max. tokens	Parameters
GPT-3.5	2048	175B
GPT-4	4096	1760B*
Bard	8192*	540B*
LLaMA-2	4096	70B

```
oGround = mbs.AddObject(ObjectGround())

# create spring damper between bodies (using local position) or between nodes, spring-damper may not have size 0; spring reference length is computed from reference configuration
oSD = mbs.CreateSpringDamper(bodyOrNodeList=[oGround, m1], localPosition0=[6,0,0], localPosition1=[0,0,0], stiffness=1e3, damping=1e1, drawSize=0.2)
...
```

This context contains only information on systems with mass points, distance constraints, and spring-dampers and is represented by 680 tokens in GPT-3. For rigid-body systems, a more comprehensive file is used, see the supplementary material, which is represented by 2842 tokens and therefore requires a minimum context length of 4096 tokens to read the input and additionally generate some useful output. The specific comments at the beginning of the context information were added because initial tests resulted in regular syntax errors, such as using 2D vectors instead of 3D.

The objective of the remainder of the paper is to evaluate the capabilities of different GPT regarding accurate creation of multibody dynamics models. In particular, the number of errors in Python models are used to evaluate and compare different approaches. In order to improve the performance, some recent simplifications in Exudyn's Python interface have been added and are made available to the LLM in the local context. As we will show, the simplified modeling, as well as the additional context, boosts the performance in particular for rigid-body systems.

4 Examples and tests

In the present research, we presents six categories of examples for the evaluation of the performance of different LLM. The list of examples is summarized in Table 2, related to knowledge of Exudyn and the creation of basic dynamic and multibody systems. All examples are tested and evaluated with several LLM, using GPT-3.5, GPT-4, Bard (PaLM-2), and LLaMA-2,[6] see Table 1 for details. GPT-3.5 and GPT-4 are accessed using ChatGPT.

Note that the maximum number of tokens are crucial for our tests, as they limit the input as well as the output text. In order to be processed more efficiently, LLM convert text into tokens, often with approximately 32 000 different tokens. Many common English words are represented by only one token. The number of tokens provided in Table 2 are obtained with

---

[6] For LLaMA-2 we use standard settings temperature: 0.9, Top-p (nucleus sampling): 0.6, Repetition penalty: 1.2.

Table 2 Examples and context information for the creation of basic dynamic and multibody dynamics systems; number of tokens counted with Tokenizer. For Examples 4–6.1 the additional context information is not included in the tokens count

Example	Description	Tokens (input)
Example 1	mass–spring-damper SciPy	72
Example 1.1	mass–spring-damper SciPy (rephrased question)	66
Example 2	keyword	7
Example 3	mass–spring-damper Exudyn (without context)	101
Example 4	mass–spring-damper Exudyn	155
Example 4.1	5-mass–spring-damper Exudyn	170
Example 5	double pendulum	118
Example 5.1	10-mass pendulum	11
Example 6	pendulum with rigid bodies	90
Example 6.1	slider-crank mechanism	30
context information	description	tokens
context mass points	mass points, spring-dampers, and constraints	680
context rigid bodies	mass points, rigid bodies, and joints	2842

Tokenizer.[7] We assume that all LLM in this paper represent text by a similar number of tokens.

All examples are given in text form only, using natural language with no or only few technical instructions, e.g., there is no hint on which solver or which method to be used. As all LLM generate nondeterministic results, tests are usually repeated with three trials. The expectation of the tests is to obtain a Python code from the LLM that can be directly processed in Python using the Exudyn package.[8] Tests are evaluated based on correct code syntax and on correct modeling of the dynamic system. In order to clearly distinguish between the two error types, syntax errors (counted by e_{syn}) are all errors that raise a Python error when the code is executed, except for Exudyn's solver failures due to modeling errors. All remaining errors are model errors (counted by e_{mod}), which are more severe because the user has to detect such errors. Note that syntax errors could be even less severe, because they could be resolved by feeding the error back to the LLM – as we will show in some of the examples.

4.1 Example 1: Create a mass–spring-damper in Python/SciPy

Example 1 is aiming to create the mathematical model and compute the solution of a linear mass–spring-damper undergoing a constant force. The example is used to compare the different LLM as all of them are able to generate Python code and to use SciPy [26]. It is not fully known to what extent the considered LLM have been trained on SciPy. However, looking at larger datasets available on HuggingFace [28], we find typical datasets[9] that are

---

[7] https://platform.openai.com/tokenizer.

[8] Due to the widely backwards compatibility of Exudyn with the version from September 2021, all tests are evaluated with Exudyn version 1.7.0. We did not observe syntax errors in the LLM outputs related to a change of version.

[9] HuggingFace's dataset: `python_code_instructions_18k_alpaca` at https://huggingface.co/datasets/iamtarun/python_code_instructions_18k_alpaca.

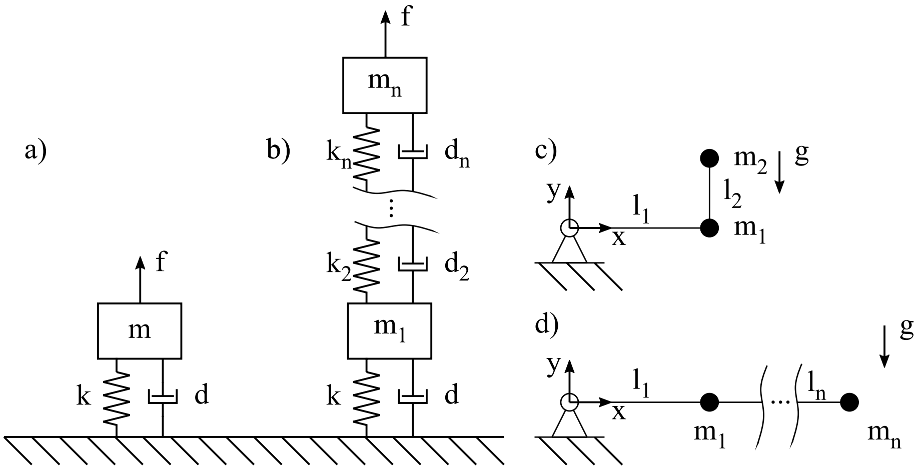

Fig. 3 Test examples based on mass points; a) is used for Example 1, Example 1.1, Example 3, and Example 4; b) for Example 4.1, c) for Example 5 and d) for Example 5.1

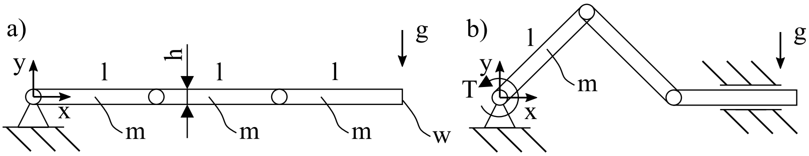

Fig. 4 Test examples based on rigid bodies; a) is used for Example 6 and b) for Example 6.1

Table 3 Prompts for mass–spring-damper system, see Fig. 3a) using SciPy. a) is used for Example 1; b) for Example 1.1

	Prompt
a)	Consider a mass–spring-damper system with the following properties: mass m = 8 kg, stiffness k = 5000 N/m, and damping d = 50 Ns/m. The force applied to the mass is f = 100 N. Create a simulation model using SciPy to simulate the dynamics of the mass–spring-damper system.
b)	Create a simulation model in Python using SciPy to simulate the dynamics of a mass–spring-damper system with the following properties: mass m = 4 kg, stiffness k = 2500 N/m, and damping d = 25 Ns/m. The force applied to the mass is f = 50 N.

solely related to Python, comprising of 18k instructions, of which 67 instructions are directly related to SciPy.

The definition of the model in text form is given in Table 3a). For clarity, the expected model is shown in Fig. 3a), which is not available to the LLM. Example 1.1 uses a slight variation of the input, see Table 3b), in order to evaluate the sensitivity on the specific input text. The results of these tests are summarized in Table 4 and some of the responses are given in Appendix A.1. It is clearly shown that all LLM have been trained for SciPy, however, LLaMA-2 generated many errors and even wrong Python syntax.

Table 4 Number of modeling errors (e_{mod}) and syntax errors (e_{syn}) of Example 1 and Example 1.1 with SciPy

	GPT-3.5		GPT-4		Bard		LLaMA-2	
	e_{mod}	e_{syn}	e_{mod}	e_{syn}	e_{mod}	e_{syn}	e_{mod}	e_{syn}
Example 1								
trial 1	0	0	0	0	1	2	1	2
trial 2	0	0	0	0	1	1	2	1
trial 3	0	0	0	0	0	2	2	1
Example 1.1								
trial 1	0	0	0	0	0	0	2	1
trial 2	0	0	0	0	0	1	2	1
trial 3	0	0	0	0	0	0	2	1

Table 5 Results of Example 2 for the keyword question "Do you know Exudyn". The LLM was trained (yes) or not trained (no) with sources and documentation of Exudyn

	GPT-3.5	GPT-4	Bard	LLaMA-2
interpretation of result	yes	yes	yes	no

Table 6 Prompt for mass–spring-damper in Exudyn, see Fig. 3

	Prompt
Example 3	Consider a mass–spring-damper system with the following properties: mass m = 8 kg, stiffness k = 5000 N/m, and damping d = 50 Ns/m. The force applied to the mass is f = 100 N. Create a simulation model using Exudyn to simulate the dynamics of the mass–spring-damper system. The spring has a length of 5 cm and is relaxed in the initial position. Please write the code with no comments and in one block.

4.2 Example 2: Do you know Exudyn?

Before continuing with further Exudyn examples, we assess whether the LLM is trained on Exudyn. While a definitive answer is elusive, we can infer from responses to specific questions. Representatively, the question "Do you know Exudyn?" is asked in a new prompt without any context. While all models say yes, we evaluate for the following keywords that are specific for Exudyn, namely multibody, simulation, rigid and flexible bodies, connectors, Python and C++. If most of these keywords are present in the answer, we consider the LLM to know the library, meaning that the according GitHub files have been used for training.

The results of these tests are summarized in Table 5 and the responses are given in Appendix A.2. It is clearly shown that only GPT-3.5, GPT-4, and Bard have been trained with sources and documentation of Exudyn, while LLaMA-2 cannot generate information clearly related to Exudyn, in particular containing a wrong focus on powder dynamics.

4.3 Example 3: Create a mass–spring-damper in Exudyn

This example is aiming to create the simulation model of a linear mass–spring-damper undergoing a constant force with Exudyn. The example is used to compare the different LLM, which have been trained on Exudyn. The definition of the model in text form is given in

Table 7 Number of modeling errors (e_{mod}) and syntax errors (e_{syn}) of Example 3 with Exudyn. If the code is riddled with syntax or modeling errors the number of modeling or syntax errors is not available (n.a)

	GPT-3.5		GPT-4		Bard		LLaMA-2	
	e_{mod}	e_{syn}	e_{mod}	e_{syn}	e_{mod}	e_{syn}	e_{mod}	e_{syn}
Example 3								
trial 1	1	4	0	4	n.a.	n.a.	n.a.	n.a.

Table 6. For clarity, the model is given in Fig. 3a), which is not available to the LLM. The results of these tests are summarized in Table 7. It is clearly shown in Table 7 that only GPT-3.5 and GPT-4 have been trained with sufficient codes and documentation of Exudyn. Only one trial has been evaluated in detail, but we did not achieve any fully correct output within several trials.

4.4 Examples 4, 4.1, 5, and 5.1 based on mass points in Exudyn with context

The examples of this section summarize results of LLM regarding mass points with spring-dampers and distance constraints modeled in Exudyn. Since none of the LLM could correctly solve the task posed in Sect. 4.3, in-context learning is applied from now on by prompting appropriate text, see Sect. 3.2, prior to the queries given in Appendix A.3. For clarity, the expected models are shown in Fig. 3, which are not available to the LLM. The results of these tests are summarized in Table 8 and Table 9. We observe the excellent performance of GPT-4, as it only produced one model error. The performance of GPT-3.5 and Bard is worse, while for GPT-3.5 in Example 4, it would be possible to feed back syntax errors within a second iteration. Remarkably, while we observed that Bard did not know Exudyn's way of modeling, it could easily learn from the context and produced only a few errors as compared to Example 3. LLaMA-2 could not sufficiently learn from the provided context and even produced highly erroneous Python syntax.

Clearly, the solution to Example 4 is almost included in the context information. The LLM only has to select the right commands and adjust input parameters, which is already too difficult for LLaMA-2. Example 4.1 goes further beyond the context information, as, for example, spring-dampers are added between mass points, which is not described in the context information.

We also observe variations of GPT-4, using a gravity constant of 10 in Example 5 (and 5.1), trial 1, versus 9.81 in the other trials of the same example. As the specification in Example 5.1 was intentionally a little unclear, the solutions differ considerably. For example, the distances of the 10 mass points are different in the trials, but we marked them as correct. Furthermore, the ways to create bodies and joints vary notably in Example 5.1, showing the abilities of GPT-4 to work with the learned context.

4.5 Examples 6 and 6.1 based on rigid bodies in Exudyn with context

Similar to the examples with mass points, in-context learning based on information for the creation of rigid bodies, joints, spring-dampers, and mass points is used also for Examples 6 and 6.1. For details of the context information for rigid bodies, see the supplementary material, but we note that it does not contain particular information to create chain-like or slider-crank mechanisms. As shown in Table 2, the number of tokens is more than 2048, which would theoretically only work with GPT-4, Bard and LLaMA-2. However, evaluation shows that GPT-3.5 is also able to generate some correct output.

Table 8 Number of modeling errors (e_{mod}) and syntax errors (e_{syn}) of Example 4 and Example 4.1 with Exudyn with context. If the simple task of Example 4 was erroneous, we skipped (skp) Example 4.1. If the code is riddled with syntax or modeling errors the number of modeling or syntax errors is not available (n.a)

	GPT-3.5		GPT-4		Bard		LLaMA-2	
	e_{mod}	e_{syn}	e_{mod}	e_{syn}	e_{mod}	e_{syn}	e_{mod}	e_{syn}
Example 4								
trial 1	0	1	0	0	1	0	n.a.	n.a.
trial 2	0	1	0	0	1	0	n.a.	n.a.
trial 3	0	1	0	0	n.a.	n.a.	n.a.	n.a.
Example 4.1								
trial 1	4	0	0	0	2	1	skp	skp
trial 2	4	0	1	0	1	1	skp	skp
trial 3	4	1	0	0	n.a.	n.a.	skp	skp

Table 9 Number of modeling errors (e_{mod}) and syntax errors (e_{syn}) of Example 5 and Example 5.1 with Exudyn with context. If the simple task of Example 5 was erroneous, we skipped (skp) Example 5.1

	GPT-3.5		GPT-4		Bard	
	e_{mod}	e_{syn}	e_{mod}	e_{syn}	e_{mod}	e_{syn}
Example 5						
trial 1	2	3	0	0	2	1
trial 2	2	6	0	0	4	2
trial 3	0	2	0	0	2	3
Example 5.1						
trial 1	skp	skp	0	0	skp	skp
trial 2	skp	skp	0	0	skp	skp
trial 3	skp	skp	0	0	skp	skp

The definitions of the models in text form are given in Appendix A.4. For clarity, the models are given in Fig. 4, but are not available to the LLM. The results of these tests are summarized in Table 10 and some exemplary outputs are given in Appendix A.4. We observe that GPT-4 produced two wrong lines of code, which could be fixed by putting the error message back as a prompt. Still, GPT-3.5 and GPT-4 performed comparatively well, mostly using wrong position vectors for joints relative to the bodies' reference positions.

In some cases, feeding error back in a second iteration, appropriate training, improved fine tuning or vision-based inputs (with multimodal LLM) could resolve such problems in the future.

Figure 5 shows a screenshot of the triple-pendulum (Example 6) and of the slider-crank mechanism (Example 6.1) created from the output of GPT-4. Note that the geometry of the slider-crank mechanism would not work for full revolutions of the crank, because the crank and conrod have the same lengths. Nevertheless, the direct extension from the triple-pendulum to a slider-crank mechanism has been performed correctly.

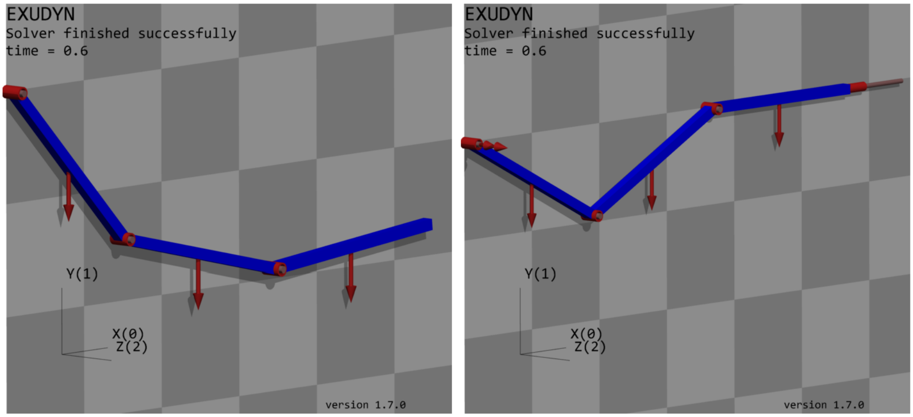

Fig. 5 Visualization in Exudyn using the Python scripts created by GPT-4: triple pendulum with rigid bodies (left, Example 6) and slider-crank mechanism (right, Example 6.1); the visualization parameters (shadow, colors, loads, font size) and the drawing size of joints have been slightly adapted in order to improve visibility (Color figure online)

Table 10 Number of modeling errors (e_{mod}) and syntax errors (e_{syn}) of Example 6 and Example 6.1 in Exudyn with context. If the simple task of Example 6 was erroneous, we skipped (skp) Example 6.1. If the code is riddled with syntax or modeling errors the number of modeling or syntax errors is not available (n.a)

	GPT-3.5		GPT-4		Bard	
	e_{mod}	e_{syn}	e_{mod}	e_{syn}	e_{mod}	e_{syn}
Example 6						
trial 1	4	0	0	0	0°	0
trial 2	0°	1	0	2^{*}	4	0
trial 3	3	0	0	0	3	0
Example 6.1						
trial 1	skp	skp	0	0	0	1
trial 2	skp	skp	0	0	skp	skp
trial 3	skp	skp	0	0	skp	skp

*corrected by putting the error message back as a prompt

$^{\circ}$visualization is erroneous

5 Conclusions and outlook

The performed experiments can be summarized as follows. Even the smallest tested LLM are able to sketch basic simulation codes for dynamic systems, for example in Python/SciPy, nevertheless producing many syntax and some model errors. Advanced LLM, such as GPT-4 that is currently leading in many LLM benchmarks, are highly reliable in writing basic simulation codes for dynamic systems. Furthermore, as shown in [3], simplified modeling language and appropriate information provided in the context greatly improves the reliability of LLM in producing correct simulation code. Regarding multibody system dynamics, advanced LLM show potentialities in creating even advanced rigid-body multibody systems with joints. In general, we observe an unsurpassed speed, e.g., a chain-like multibody system can be created by the LLM in less than a minute. Within software designed for structured data representation in tabular formats, the integration of AI-driven functionalities becomes evident, offering automated recommendations for data visualization strategies, and intelligent formula suggestions derived from the inherent dataset characteristics. Consequently, multibody models generated from natural language could be positioned as a supplementary tool augmenting GUI modeling, rather than a complete replacement.

While the simulation models generated by LLM are not always error free, they can provide significant relief in practice by providing a quick initial approach to implementation so that users do not need to master the complex description language. While not shown here, also parts of models could be created by the LLM.

Due to the way Exudyn data is available, many LLM could learn solely from code examples. This means it could learn blindly, not having any description of kinematics by images. This may, in particular, explain the limitations of geometric and kinematic understanding that became evident during the tests performed. The latter limitation is about to be remedied, as ChatGPT-4V (vision) became available in September 2023, which also supports graphical input. This could enable us to create multibody models from (hand) sketches.

As a main result, we could demonstrate the large differences in quality of the investigated LLM, showing a strong relation of quality and size of the LLM. This allows us to conclude that future LLM, not only increasing in size, but also with longer training time, advanced model structures such as chain-of-thoughts [27], improved hyperparameters, larger training datasets, and detailed training on Exudyn or similar libraries, could perform much better than available models today. Therefore, we conclude that future LLM could create highly complex rigid or flexible multibody systems from natural language.

Appendix A

The Appendix includes exemplary outputs for the LLM, while the full outputs are provided in the supplementary material. In the case of code output, the complete output of the LLM is contained in the ".py" files, where the textual descriptions before or after the code are added as comments and clearly marked as such. Outputs are usually shortened, clearly visible by the line numbers as well as "..." used to indicate previous or subsequent code. Empty lines have been removed for brevity of code snippets.

A.1 Example 1: Create a mass–spring-damper in Python/SciPy

A.1.1 Example 1: GPT-4 trial 1

The following code snippet shows the Example 1 output generated by GPT-4, which is fully correct:

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
# Given system parameters
m = 8.0      # kg
d = 50.0     # Ns/m
k = 5000.0   # N/m
f = 100.0    # N
# Differential equations
def system_dynamics(t, y):
    x, v = y
    dxdt = v
    dvdt = (f -  d  * v -  k  * x) / m
    return [dxdt, dvdt]
# Time span for the simulation
t_span = [0, 5]  # 5 seconds, for instance
initial_conditions = [0, 0]  # Starting from rest at the equilibrium position
# Solve the differential equations
solution = solve_ivp(system_dynamics, t_span, initial_conditions, t_eval=np.linspace(*t_span, 1000))
...
```

The correct plotting with MatplotLib has been omitted from the output.

A.1.2 Example 1: GPT-4 trial 2

The following code snippet shows the Example 1 output generated by GPT-4 in the second trial, also fully correct:

```
...
# Define system parameters
m = 8.0  # kg
k = 5000.0  # N/m
d = 50.0  # Ns/m
f = 100.0  # N
# Define the system ODE
def system_ode(y, t):
    # Unpack the current state
    x, x_dot = y
    # Compute acceleration
    x_double_dot = (f - d*x_dot - k*x) / m
    return [x_dot, x_double_dot]
# Initial conditions: [x(0), x_dot(0)]
y0 = [0.0, 0.0]
# Time array
t = np.linspace(0, 10, 1000)
# Solve the system ODE
sol = odeint(system_ode, y0, t)
...
```

Note that there are slight variations, even of comments, but also conceptual differences in using `odeint` instead of `solve_ivp` in trial 1.

A.1.3 Example 1: Bard trial 1

The following code snippet shows a modeling error in line 15, where `f` is not divided by mass `m`, and syntax errors in lines 14 and 15, which are not compatible with odeint (`x, v` need to be represented by a list and the return value of function must be a list):

```
1   import numpy as np
2   from scipy.integrate import odeint
3   # Define the system parameters
4   m = 8  # Mass (kg)
5   k = 5000  # Spring stiffness (N/m)
6   d = 50  # Damping coefficient (Ns/m)
7   f = 100  # External force (N)
8   # Initialize the state variables
9   x = 0  # Displacement (m)
10  v = 0  # Velocity (m/s)
11  # Set the time step
12  dt = 0.01  # seconds
13  # Create the differential equations
14  def derivs(x, v, t):
15      return v, -(k / m) * x - (d / m) * v + f
```

```
# Simulate the system
t = np.arange(0, 10, dt)
x_sim, v_sim = odeint(derivs, [x, v], t)
...
```

A.1.4 Example 1: LLaMA-2 trial 1

The following code snippet shows a modeling error in line 19, where `d * v` should have a negative sign; furthermore units in lines 15 and 16 must have a preceding comment and the syntax in line 19 is wrong as it misses a line break between `v` and `v_dot`:

```
13  ...
14  # Define the initial conditions
15  x0 = 0 m
16  v0 = 0 m/s
17  # Define the ODE function
18  def ode_func(t, x, v, m, k, d, f):
19      x_dot = v v_dot = (-k * x +  d  * v + f) / m
20      return [x_dot, v_dot]
21  # Integrate the ODE using RK4
22  t = np.arange(0, t_sim, dt)
23  [x, v] = odeint(ode_func, [x0, v0], t, args=(m, k, d, f))
24  ...
```

A.2 Example 2: Do you know Exudyn?

The following table summarizes results for the keyword question "Do you know Exudyn?". The table contains only the first (and main) paragraph of the output, while the complete output is in the supplementary material.

LLM	output (shortened)
GPT-3.5	Yes, I'm familiar with Exudyn. Exudyn is a software tool designed for multibody dynamics simulation and analysis. It's commonly used for simulating and analyzing mechanical systems, including rigid and flexible bodies, joints, constraints, and forces. Exudyn is particularly useful for studying the dynamics of complex mechanical systems such as machinery, vehicles, robots, and other mechanical structures.
GPT-4	Yes, as of my last update in September 2021, Exudyn is a software package primarily focused on multibody system dynamics. It is designed for the simulation of rigid and flexible bodies, connectors, loads, and other elements in a multibody environment.
Bard	Yes, I am familiar with Exudyn. It is an open-source, C++-based Python library for the efficient simulation of flexible multibody dynamics systems. It is designed to easily set up complex multibody models, consisting of rigid and flexible bodies, and to simulate their dynamic behavior. Exudyn supports a variety of numerical methods for solving the equations of motion, including explicit and implicit integration schemes, and modal analysis. It also includes a number of features for post-processing simulation results, such as visualization and data analysis.
LLaMA-2	Yes, I'm familiar with Exudyn. It's a software tool used for simulation and analysis of powder dynamics, particle systems, and granular materials. Exudyn is designed to simulate the behavior of granular materials, such as powders, sands, and gravels, in various industrial and scientific applications.

A.3 Examples 4, 4.1, 5, and 5.1: Mass-point examples in Exudyn with context

The following table lists prompts for examples with mass points, spring-dampers, and distance constraints, see Fig. 3:

	Prompt
Example 4	Consider a mass–spring-damper system with the following properties: mass m = 8 kg, stiffness k = 5000 N/m, and damping d = 50 Ns/m. The force acting on the mass is f = 100 N and applied to the mass in the direction in which the spring expands at the beginning of the simulation. The spring has a length of 5 cm and is relaxed in the initial position. Gravity is neglected and the mass should be placed in the x-direction at the relaxed position of the spring relative to the ground. Create a simulation model using Exudyn to simulate the dynamics of the mass–spring-damper system. Use the previously defined mbs.Create functions. Please write the code with no comments and in one block.
Example 4.1	Consider a 5-mass–spring-damper system with the following properties: mass m = 8 kg, stiffness k = 2000 N/m, and damping d = 25 Ns/m. The force acting on the last mass is f = 100 N and applied to the mass in the direction in which the spring expands at the beginning of the simulation. The springs have a length of 5 cm and are relaxed in the initial position. Gravity is neglected and the first mass should be placed in the x-direction at the relaxed position of the spring relative to the ground and the first spring is connected to the ground. Create a simulation model using Exudyn to simulate the dynamics of the 5-mass–spring-damper system. Use the previously defined mbs.Create functions. Please write the code with no comments and in one block.
Example 5	Consider a double pendulum using two mass points and distance constraints in Exudyn, using the previous update info with mbs.Create functions. The mass points shall have 1 kg and the length of the first link is 2 m and the second link 1 m. The initial configuration is an L-shape, where the first link points along the x-axis and the second link points up. Gravity acts in a negative y-direction. Draw the nodes not as points, mass points have a size of 0.1 m. Please write the code with no comments and in one block.
Example 5.1	Can you add now 10 mass points in a row?

A.4 Examples 6 and 6.1: Prompts for rigid bodies with joints

The following table shows the prompts for examples with rigid bodies and joints:

	Prompt
Example 6	Using the previous information and the already existing information on Python code Exudyn, create a 3-link system of rigid bodies initially aligned along the X-axis with mass 10 kg, length 2 m and W = H = 0.1 m. The rigid bodies are exposed to gravity which acts in Y-direction and the first link is attached to ground with a revolute joint at the left end. Put all output into a single Python script.
Example 6.1	Can you add additionally at the last body a prismatic joint to ground and at the left side of the first body a torque of 100 Nm?

A.4.1 Example 6: Bard trial 2

While Bard's trial 1 for Example 6 is working fully correct (but graphics is missing), trial 2 contains two modeling errors (missing ground joint and wrong position for revolute joint), shown in the following code snippet:

```
...
# Define a graphics object for the rigid bodies
graphicsCube = GraphicsDataOrthoCubePoint(centerPoint = [0,0,0], size=[L,H,W], color=color4orange)
# Define the inertia of the rigid bodies
inertiaCube = InertiaCuboid(density=10, sideLengths=[L,H,W])
# Create the first rigid body
b0 = mbs.CreateRigidBody(inertia = inertiaCube,
                         referencePosition = [0,0,0], #reference position x/y/z of COM
                         referenceRotationMatrix=RotationVector2RotationMatrix([0,0,0]),
                         initialAngularVelocity=[0,0,0],
                         initialVelocity=[0,0,0],
                         gravity = [0,-9.81,0],
                         graphicsDataList = [graphicsCube])
# Create the second rigid body
b1 = mbs.CreateRigidBody(inertia = inertiaCube,
                          referencePosition = [L,0,0], #reference position of body [X,Y,Z]
                          gravity = [0,-9.81,0],
                          graphicsDataList = [graphicsCube])
# Create the third rigid body
b2 = mbs.CreateRigidBody(inertia = inertiaCube,
                          referencePosition = [2*L,0,0], #reference position of body [X,Y,Z]
                          gravity = [0,-9.81,0],
                          graphicsDataList = [graphicsCube])
# Create a revolute joint between the ground and the first rigid body
mbs.CreateRevoluteJoint(bodyNumbers=[b0, b1], position=[0,0,0], axis=[0,0,1], #rotation along global z-axis
                        useGlobalFrame=True, axisRadius=0.02, axisLength=0.14)
# Assemble the MainSystem
mbs.Assemble()
...
```

Note that also the comment for the revolute joint is misleading (no ground).

A.4.2 Example 6.1: GPT-4 trial 1

The following code shows the full output for Example 6.1 trial 1 generated by GPT-4, which is fully correct:

```
import exudyn as exu
from exudyn.utilities import *
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
#graphics data for checkerboard background:
gGround0 = GraphicsDataCheckerBoard(point=[5,0,-0.1], normal=[0,0,1], size=10)
oGround = mbs.AddObject(ObjectGround(visualization=VObjectGround(graphicsData=[gGround0])))
# Link properties
L = 2       # length
W = 0.1     # width
H = 0.1     # height
```

```
m = 10     # mass
# Graphics and inertia for the links
graphicsLink = GraphicsDataOrthoCubePoint(centerPoint=[0,0,0], size=[L,H,W], color=color4blue)
inertiaLink = InertiaCuboid(density=m/(L*H*W), sideLengths=[L,H,W])
# Create 3 links
bodies = []
for i in range(3):
    bi = mbs.CreateRigidBody(inertia=inertiaLink,
                             referencePosition=[(0.5+i)*L, 0, 0],
                             gravity=[0, -9.81, 0],
                             graphicsDataList=[graphicsLink])
    bodies.append(bi)
# Attach the first link to the ground with a revolute joint
mbs.CreateRevoluteJoint(bodyNumbers=[oGround, bodies[0]], position=[0, 0, 0], axis=[0, 0, 1],
                        useGlobalFrame=True, axisRadius=0.02, axisLength=0.14)
# Attach consecutive links with revolute joints
for i in range(1, 3):
    mbs.CreateRevoluteJoint(bodyNumbers=[bodies[i-1], bodies[i]], position=[i*L, 0, 0], axis=[0, 0, 1],
                            useGlobalFrame=True, axisRadius=0.02, axisLength=0.14)
# Add prismatic joint to the last body
mbs.CreatePrismaticJoint(bodyNumbers=[oGround, bodies[-1]], position=[3*L, 0, 0], axis=[1,0,0],
                         useGlobalFrame=True, axisRadius=0.02, axisLength=0.14)
# Add torque of 100 Nm to the left side of the first body
markerTorque = mbs.AddMarker(MarkerBodyRigid(bodyNumber=bodies[0], localPosition=[-0.5*L, 0, 0]))
mbs.AddLoad(Torque(markerNumber=markerTorque, loadVector=[0, 0, 100]))  # 100 Nm torque in z-direction
# Simulation settings and execution
mbs.Assemble()
simulationSettings = exu.SimulationSettings()
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 5
simulationSettings.linearSolverSettings.ignoreSingularJacobian = True
simulationSettings.linearSolverType = exu.LinearSolverType.EigenDense
mbs.SolveDynamic(simulationSettings=simulationSettings,
                 solverType=exu.DynamicSolverType.GeneralizedAlpha)
mbs.SolutionViewer()
```

This final example also shows the elegant solution to store body numbers and use them in loops for creating bodies as well as for creating joints.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s11044-023-09962-0.

Author contributions J.G. provided the main conceptual ideas and structure of the paper, wrote main parts of Sections 1, 3 and 5, performed early experiments and developed the in-context approach. P.M. developed Section 2, contributed to the experiments, and contributed significantly to Sections 1 and 3. M.P. conceptualized and performed the main experiments, evaluated results and wrote Section 4 as well as the appendix. All authors contributed to revision and proof-reading.

Funding Open access funding provided by University of Innsbruck and Medical University of Innsbruck.

Declarations

Usage of LLM Apart from the experiments performed with LLM, as clearly stated in the paper, we used LLM for spell checking, text improvement and translation of text snippets from our mother tongue to English.

Competing interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: Proceedings of the 6th International Conference on Learning Representations (ICLR) (2018)
2. Beltagy, I., Lo, K., Cohan, A.: SciBERT: a pretrained language model for scientific text. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pp. 3615–3620. Association for Computational Linguistics (2019)
3. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. Adv. Neural Inf. Process. Syst. **33**, 1877–1901 (2020)
4. Buehler, M.J.: Melm, a generative pretrained language modeling framework that solves forward and inverse mechanics problems (2023). ArXiv preprint. arXiv:2306.17525
5. Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H.W., Sutton, C., Gehrmann, S., et al.: Palm: Scaling language modeling with pathways (2022). ArXiv preprint. arXiv:2204.02311
6. Devlin, J., Chang, M.-W., Lee, K., Bert, K.T.: Pre-training of deep bidirectional transformers for language understanding (2018). ArXiv preprint. arXiv:1810.04805
7. Fill, H.-G., Muff, F.: Visualization in the era of artificial intelligence: Experiments for creating structural visualizations by prompting large language models (2023). ArXiv preprint. arXiv:2305.03380
8. Gede, G., Peterson, D.L., Nanjangud, A.S., Moore, J.K., Hubbard, M.: Constrained multibody dynamics with python: from symbolic equation generation to publication. In: Proceedings of IDETC/MSNDC 2013, ASME 2013 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (2013)
9. Gerstmayr, J.: HOTINT – a C++ environment for the simulation of multibody dynamics systems and finite elements. In: Arczewski, K., Fraczek, J., Wojtyra, M. (eds.) Proceedings of the Multibody Dynamics 2009 Eccomas Thematic Conference (2009)
10. Gerstmayr, J.: Exudyn – a C++-based Python package for flexible multibody systems. Multibody Syst. Dyn. (2023)
11. Gupta, T., Zaki, M., Krishnan, N.A., MatSciBERT, M.: A materials domain language model for text mining and information extraction. Comput. Mater. **8**(1), 102 (2022)
12. Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L.A., Welbl, J., Clark, A., Hennigan, T., Noland, E., Millican, K., van den Driessche, G., Damoc, B., Guy, A., Osindero, S., Simonyan, K., Elsen, E., Rae, J.W., Vinyals, O., Sifre, L.: Training compute-optimal large language models (2022). ArXiv preprint. arXiv:2203.15556
13. Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., Wang, H.: Large language models for software engineering: a systematic literature review (2023). ArXiv preprint. arXiv:2308.10620
14. Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Lora, W.C.: Low-rank adaptation of large language models (2021). ArXiv preprint. arXiv:2106.09685

15. Jurafsky, D., Martin, J.: Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition (2008)
16. Kaplan, J., McCandlish, S., Henighan, T., Brown, T.B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., Amodei, D.: Scaling laws for neural language models (2020). ArXiv preprint. arXiv:2001.08361
17. Masarati, P., Morandini, M., Mantegazza, P.: An efficient formulation for general-purpose multibody/-multiphysics analysis. J. Comput. Nonlinear Dyn. **9**(4) (2014)
18. Mazhar, H., Heyn, T., Pazouki, A., Melanz, D., Seidl, A., Bartholomew, A., Tasora, A., Negrut, D.: CHRONO: a parallel multi-physics library for rigid-body, flexible-body, and fluid dynamics. Mech. Sci. **4**(49–64), 02 (2013)
19. OpenAI: (2023). GPT-4 Technical Report. ArXiv preprint. arXiv:2303.08774
20. Peng, S., Kalliamvakou, E., Cihon, P., Demirer, M.: the impact of AI on developer productivity: Evidence from GitHub Copilot (2023). ArXiv preprint. arXiv:2302.06590
21. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al.: Language models are unsupervised multitask learners. OpenAI blog **1**(8), 9 (2019)
22. Sun, S., Liu, Y., Iter, D., Zhu, C., Iyyer, M.: How does in-context learning help prompt tuning? (2023). ArXiv preprint. arXiv:2302.11521
23. Taylor, R., Kardas, M., Cucurull, G., Scialom, T., Hartshorn, A., Saravia, E., Poulton, A., Kerkez, V., Galactica, R.S.: A large language model for science (2022). ArXiv preprint. arXiv:2211.09085
24. Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., et al.: Lamda: Language models for dialog applications (2022). ArXiv preprint. arXiv:2201.08239
25. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Adv. Neural Inf. Process. Syst. **30** (2017)
26. Virtanen, P., et al.: SciPy 1.0: fundamental algorithms for scientific computing in Python. Nat. Methods **17**, 261–272 (2020)
27. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models (2023). ArXiv preprint. arXiv:2201.11903
28. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al.: Huggingface's Transformers: State-of-the-Art Natural Language Processing (2019). arXiv preprint. arXiv:1910.03771
29. Xu, F.F., Alon, U., Neubig, G., Hellendoorn, V.J.: A systematic evaluation of large language models of code (2022). ArXiv preprint. arXiv:2202.13169

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.