PolySolve

A graph-based equation solver combining deterministic computation with machine-learning models for accurate step-by-step solutions.

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Computer Science,
Mathematics

By:

Conner Olsen
Alexander Gu
with input from Alexander Siracusa

Project Advisor:

Randy C. Paffenroth

Date: May 2025

Abstract

PolySolve is a graph-based calculator that solves multi-step, domain-specific problems by uniting a deterministic computational core with three narrowly-scoped helpers, one Support Vector Machine and two Large Language Models (LLMs). Existing computational tools, including advanced calculators and Large Language Models (LLMs), face limitations when solving complex, multi-step problems requiring domain-specific reasoning. Traditional calculators are fast and reliable but inflexible, as they require precise input, lack the capability to automatically select and apply equations, and cannot generate contextual natural language explanations. LLMs, on the other hand, are very flexible but can suffer from factual inaccuracies, lack the reliability needed for precise computation, and are computationally expensive for high-parameter models. PolySolve introduces a hybrid methodology that provides the benefits of both approaches. A deterministic core maps equations and variables to a bipartite graph and applies a dependency-guided search algorithm to automatically select relevant equations and derive accurate solution steps. Three small machine learning models are integrated for strictly non-computational tasks: A fine-tuned LLM extracts information from natural language input, such as word problems, into a predefined format, which is then passed to our deterministic core for a solution. An SVM-based variable matcher handles input variations such as typos, abbreviations, or synonyms of known variables, improving robustness at input time. Finally, the output LLM takes the algebraic steps from the deterministic core, and converts them to a natural language explanation with added context. Crucially, none of these models are involved in computing the answer, and only serve to enhance the usability of the calculator. Together, these components deliver accurate, verifiable solutions that end-to-end LLMs cannot guarantee, with the added flexibility and natural language output that traditional calculators lack, closing the gap between computational rigor and user-friendly LLMs.

Acknowledgments

The completion of this project and the development of PolySolve would not have been possible without the support, guidance, and contributions of several individuals and organizations.

First and foremost, we extend our deepest gratitude to our project advisor, Professor Randy Paffenroth. His sagely guidance and consistent encouragement were greatly appreciated throughout the entire project lifecycle, from conceptualization to implementation and reporting. We have grown as researchers and developers under his guidance.

Special thanks are also due to our third unofficial member, Alexander Siracusa. Although unable to formally be part of the MQP team, driven purely by his intellectual curiosity and interest, he joined the PolySolve team through an Independent Study Project (ISP). He shaped the project's development through his technical contributions, frontend expertise, and humor. E.

Furthermore, our gratitude extends to Professor Snehalata Kadam and Professor Izabella Stroe. They sponsored a crucial precursor ISP in 2023, which laid the foundations for this MQP. They provided critical insights by acting as domain experts and potential end-users, offering feedback, shaping project requirements, and allowing the use of their course materials for our dataset. We also thank our fellow classmate Howard Chao, who participated in the initial ISP and contributed to the development of the first prototype. We likewise appreciate the willingness of Professor Thomas Noviello to provide the use of his course material for our datasets.

Finally, we acknowledge Worcester Polytechnic Institute (WPI) for providing the essential framework, resources, and opportunity to undertake this Major Qualifying Project.

Contents

List of Tables

List of Figures

1 Executive Summary

We present PolySolve, a novel computational tool designed to address limitations in existing calculators (e.g., WolframAlpha [25], Mathway [26]) and Large Language Models (LLM) [13] when solving complex, multi-step, domain-specific problems, particularly in areas like introductory physics. Traditional calculators lack inherent knowledge of domain-specific formulas and automatic equation selection, multi-step reasoning, and providing contextual explanations. While LLMs offer natural language capabilities and multi-step reasoning, they suffer from factual inaccuracies (hallucinations), high computational cost, and lack of reliability for precise calculations.

PolySolve bridges this gap by integrating a deterministic computational engine with optional, limited use of small LLMs. The core of PolySolve represents systems of equations as a bipartite graph where nodes represent equations and variables. A computational engine utilizes a dependency-guided variation of Uniform Cost Search (UCS) [34] on this graph to automatically identify the relevant equations and determine the optimal sequence of steps required to solve for a desired variable, given a set of known variables. Python's SymPy library [29] is employed within this engine for parsing, simplifying, and solving individual algebraic equations or small systems identified during graph traversal.

To enhance usability, PolySolve incorporates two optional LLM-driven components using fine-tuned Llama 3.2 1B and 3B models trained with QLoRA. The first is a Word Problem Parser that extracts "given" and "wanted" variables from natural language problem descriptions into a structured format. The second is an explanation generator that translates the engine's procedurally generated step-by-step solution path into natural language, contextualized for the problem domain. Crucially, LLMs are not involved in the core mathematical computation, removing the risk of calculation errors due to hallucination.

Furthermore, PolySolve includes a Variable Matching System to handle variations in user input for variable names (e.g., typos, abbreviations). This system uses a Support Vector Machine (SVM) classifier, trained using a One-versus-One [37] strategy on a dataset augmented with generated misspellings and abbreviations. Input terms are converted to vector representations using CountVectorizer (character subsequences up to length 3) followed by dimension reduction via Truncated SVD [22], allowing the SVM to map potentially noisy inputs to canonical variable names stored in a domain-specific database or classify them as "No Guess" if confidence is low.

PolySolve demonstrates a hybrid approach that leverages the accuracy of algorithmic computation for core solving tasks while using LLMs selectively for user-interface enhancements, such as natural language

input parsing and output explanation. This results in a tool that can automatically handle multi-step problems within a defined domain, select necessary equations, provide accurate results, and offer understandable, step-by-step explanations. Figure 1 provides an overview of the full pipeline of PolySolve.

Identified limitations include a reliance on pre-defined domain-specific equation databases, the inability to dynamically model complex physical interactions (such as multi-force collisions), and difficulties handling variables with multiple context-dependent equations. Future work could explore a controlled expansion of LLM roles for equation database management or context-aware equation selection, alongside developing capabilities for dynamic interaction modeling, while carefully managing the trade-off between flexibility and computational accuracy. PolySolve serves as a proof-of-concept for building more reliable and user-friendly computational tools by strategically combining deterministic algorithms with targeted AI capabilities.

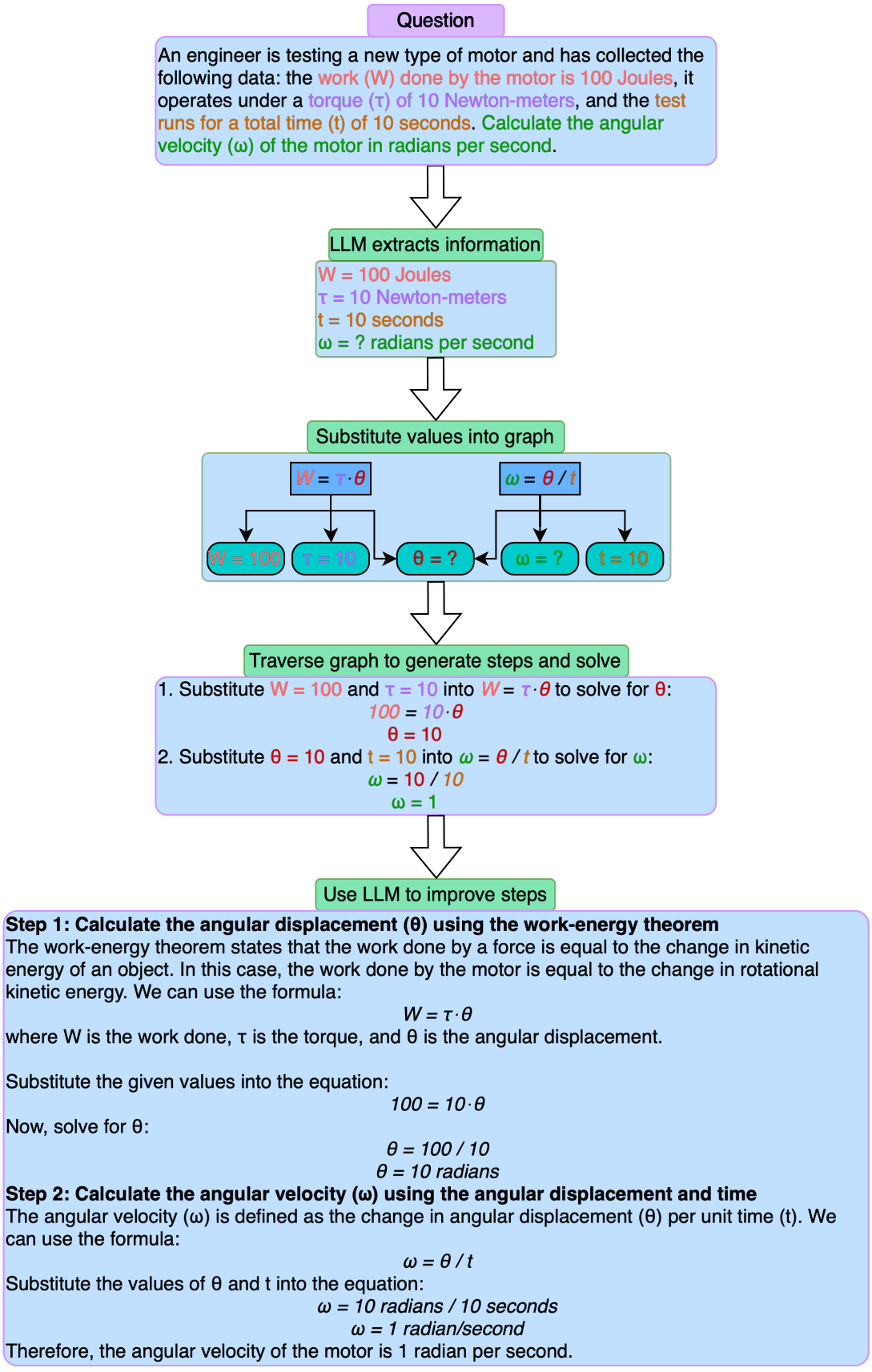

Figure 1: As is detailed in Section 4.1, the above figure showcases the full pipeline of the calculator.

2 Introduction

Solving real-world problems often demands more than plugging numbers into a single formula; it requires determining the necessary equations to use (e.g., related physics formulas) and then applying them in order so that each intermediate result supplies the variables needed by the next. Conventional calculators execute individual expressions with speed and precision, yet place the entire burden of equation selection, sequencing, and intermediate bookkeeping on the user. When many interdependent variables are involved that burden grows quickly, turning the act of using the calculator into a slow error-prone workflow.

Existing calculator platforms such as WolframAlpha [25], Mathway [26], and Symbolab [39] provide powerful computational engines. However, they fall short when dealing with multi-step problems requiring domain-specific reasoning. As illustrated by examples in Appendix A, these tools enforce rigid input formats—such as only accepting single-letter variables and providing a single line for input; are unable automatically select and apply relevant equations; and lack context persistence between sequential calculations, forcing users to manually track intermediate results in multi-step scenarios. Thus, for multi-step problems the user must identify every relevant equation, supply those equations in a single correctly ordered request, and manually carry intermediate results forward if the solution spans multiple inputs. Furthermore, the explanations provided on these platforms do not provide natural-language explanations that connect algebraic manipulations to the underlying concepts.

General purpose numerical environments such as MATLAB [27] remove the single-line input and single-letter variable restrictions and maintain session state, yet they still require users to script every algebraic step manually; like the web calculators, they do not offer automatic equation selection, do not accept natural language input, and do not provide step-by-step natural language explanations.

Large-language models (LLMs) such as ChatGPT appear to offer an all-in-one remedy: they can read a free-form word problem, infer the governing equations, produce an answer, and narrate the reasoning process [32]. In practice, however, purely generative pipelines suffer from well-documented issues. Hallucinations jeopardise numerical fidelity [13]; inference is costly and slow, especially when only a few constants change between iterations [8]; and the entire chain of reasoning must be regenerated after even minor edits, making interactive exploration cumbersome.

To illustrate these limitations, consider a basic mechanics exercise: determine the final angular velocity ω of a rotating object given its linear acceleration $a = 4\,\mathrm{m/s^2}$, elapsed time $t = 10\,\mathrm{s}$, radius $r = 2\,\mathrm{m}$, and an initial angular velocity of $\omega_0 = 0$. From this problem it is trivial to identify the provided information

and what is required:

- Given: $a = 4\,\mathrm{m/s^2}$, $t = 10\,\mathrm{s}$, $r = 2\,\mathrm{m}$, $\omega_0 = 0$
- Wanted: ω

However, none of the available tools can solve for ω using just this information. They cannot automatically determine which equations are needed or the correct order in which to apply them based only on the known and desired variables. Rather, the user must be responsible for identifying and applying the intermediate steps correctly:

1. Compute angular acceleration α using $\alpha = a/r$, resulting in $\alpha = 2$.
2. Use an equation such as $\omega = \omega_0 + \alpha t$ to solve for ω.
3. Given an initial condition $\omega_0 = 0$, derive $\omega = 20$ as the final result.

Although one may manually solve the problem and can use tools to handle portions of the process, it is time-consuming and invites transcription and sequencing errors, even for experts, especially as the complexity of the problem grows.

These observations point to a clear gap: we lack a system that marries the rigour and reproducibility of deterministic computation with the convenience of conversational, step-by-step assistance for multi-equation problems. To address that gap PolySolve, a graph-based calculator that combines a deterministic computational core with three narrowly scoped machine-learning models. The deterministic core maps equations and variables to a bipartite graph and employs a dependency-guided search algorithm to automatically select the minimal set of equations that link the given data to the target unknown, then generating a step by step solution. For added input flexibility and natural language explanation, we incorporates three small Machine Learning models in limited roles: a input LLM parses natural language problems into structured input, an SVM-based matcher resolves typos and synonyms in variable names, and a output LLM converts the algorithmically derived solution steps to a natural language explanation with added context.

By restricting the models to these non-computational tasks, the risk and impact of hallucinations are significantly mitigated, as the core mathematical accuracy relies solely on deterministic computational. This approach of combining the flexibility of Machine Learning models with the accuracy of deterministic computation allows PolySolve to offer automatic equation selection, real-time updates as givens change, and clear, domain-specific explanations—without the cost and reliability issues of end-to-end generative approaches.

3 Background

3.1 Notation

Notation	Description
$\mathbf{a}$	Vector (column vector by default).
θ	Vector*.
A	Matrix.
$\begin{bmatrix}\mathbf{a}_1 & \dots & \mathbf{a}_n\end{bmatrix}$	An $m \times n$ matrix, for $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n \in \mathbb{R}^m$.
$A^\intercal$	Transpose of matrix A.
AB	Matrix product of matrices A and B.
A_{ij}	Entry of matrix A on row i and column j.
A_{i*}	Row i of matrix A.
A_{*j}	Column j of matrix A.
$\mathcal{A}^c$	The complement of set $\mathcal{A}$.
$\lvert\mathcal{A}\rvert$	The length of set $\mathcal{A}$.

Table 1: This table a standard set of notation that will be used throughout this paper.
*Vector containing the parameters of a Machine-Learning model (see Section 3.4).

3.2 Graph-based System of Equations Representation

The calculator represents a system of equations as a bipartite graph—a graph whose vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other—where equations occupy one set and variables the other. A connection between an equation node and a variable node means that the variable is present in the equation. For example, the node for "x + y = z" would be connected to the nodes for "x", "y", and "z". Through this representation, a path existing between two variable nodes means that they are related and that one can be solved in terms of the other by applying the equations in the path. More broadly, a tree with several variable nodes as leaves ("leaf variables") means that any one leaf variable can be solved in terms of all the other leaf variables. On the other hand, if a path or tree cannot exist between some variable nodes, then it cannot be written in terms of the others.

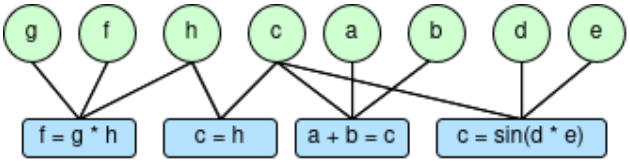

Figure 2: Example bipartite graph of equations (in blue) and variables (in green). If there exists a path between two given variables, then one can be solved for in terms of the other. More generally, if there exists a tree between a set of variables, then one can be written in terms of all the other variables. In other words, the process of building such a tree is equivalent to solving for a variable in terms of a predefined set of other variables.

3.3 Algebraic Word Problem

An algebraic word problem provides 'given variables' (variables with known numerical values) and asks the reader to solve for the values of 'wanted variables' (variables with no known numerical or symbolic value). For example, see the following.

> "A simple pendulum has a length L of 0.75 meters. The pendulum mass is displaced horizontally from its equilibrium position by a distance x of 5.0 mm and then released. Calculate the maximum speed of the mass, denoted v_max. Take g = 9.81 m/s/s."

Such a problem involves given variables—L = 0.75 [m], x = 5.0 [mm], and g = 9.81 [m/s/s]—and wanted variables—v_max and time.

Solving an algebraic word problem involves extracting the given and wanted variables and then applying algebra to the set of equations that relate them to determine the values of the wanted variables. Continuing the above example, one possible approach is to apply the below equations.

$$U_g = K$$
$$K = (1/2) * m * v^2$$
$$U_g = m * g * L * (1 - \cos(\theta))$$
$$\theta = \arcsin(x/L)$$

The problem solver needs to identify the relevant equations and then chain them together in sequence, as illustrated above, then fill in the provided information to arrive at the correct values.

3.4 Machine Learning

Much of the functionality involved in interacting with the calculator is implemented with machine learning models. For example, the calculator can optionally parse word problems to extract key information (see Section 5.3). Writing code from scratch to algorithmically extract the given and wanted information from text can be arduous due to the complex nature of natural language. A machine learning model, on the other hand, can learn to recognize patterns in text to extract relevant information.

A machine learning model, in essence, makes predictions on a set of data and can improve its predictions by implicitly learning patterns from instances of said data. A parametric machine learning model f can be represented as the following [16, p. 17]:

$$\hat{\mathbf{y}} = f(\mathbf{x}, \theta) \tag{1}$$

where

- $\mathbf{x} \in \mathbb{R}^{d_{\text{input}}}$ denotes input data to the model.
- $\theta \in \mathbb{R}^{d_{\text{parameters}}}$ denotes the model's parameters, which are adjusted as the model trains for more accurate predictions.
- $\hat{\mathbf{y}} \in \mathbb{R}^{d_{\text{output}}}$ denotes the output (i.e., prediction) of the model.

During supervised training, the aim is to improve the accuracy of the predictions of f by lowering the loss function $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$, where $\mathbf{y} \in d_{\text{output}}$ is the expected output of the model f.

3.5 Neural Networks

Neural networks, a type of parametric machine learning model, computes $\hat{\mathbf{y}}$ by passing $\mathbf{x}$ through a sequence of layers, each with its own set of parameters. Though equation 1 applies to neural networks, the following is a more concrete representation:

$$\hat{\mathbf{y}} = f_n(f_{n-1}(...(f_2(f_1(\mathbf{x}, \theta_1), \theta_2)\dots), \theta_{n-1}), \theta_n) \tag{2}$$

where

- $\mathbf{x} \in \mathbb{R}^{d_{\text{input}}}$ is the data that is fed into the network.

- $\hat{\mathbf{y}} \in \mathbb{R}^{d_{\text{output}}}$ is the prediction that the network returns.
- $n \in \mathbb{N}$ is the total number of layers in the neural network.
- f_i (for $i \in \{1, 2, \ldots, n\}$) denotes the function associated with layer i.
- $\theta_i \in \mathbb{R}^{d_{\text{parameters}_i} \times d_{\text{parameters}_{i+1}}}$ (for $i \in \{1, 2, \ldots, n\}$) denotes the parameters associated with layer i.

Qualitatively, $\mathbf{x}$ is fed into the first layer with its computations influenced by the parameters θ_1, then the layer outputs $f_1(\mathbf{x}, \theta_1)$, which is then fed into the second layer with its own set of parameters θ_2, and so on through all n layers.

For Multi-Layered Perceptrons, a specific kind of neural network, each "fully connected" layer f_i processes incoming data $\tilde{\mathbf{x}}$ with an affine transformation $W_i\tilde{\mathbf{x}}_i + \mathbf{b}_i$, followed by some nonlinear function σ (e.g., ReLU, Sigmoid, Tanh)[4], where:

- $i \in \{1, \ldots, n\}$ indexes the n layers in the network.
- $d_{\text{in}\,i}$ is the number of input features for layer i.
- $d_{\text{out}\,i}$ is the number of output features for layer i.
- $\tilde{\mathbf{x}}_i \in \mathbb{R}^{d_{\text{in}\,i}}$ denotes the layer's input data.
- $W_i \in \mathbb{R}^{d_{\text{out}\,i} \times d_{\text{in}\,i}}$ contains the layer's "weights."
- $\mathbf{b}_i \in \mathbb{R}^{d_{\text{out}\,i}}$ contains the layer's "bias."
- weights W_i and bias $\mathbf{b}_i$ are parameters of the layer.

The affine operations enables efficient matrix computations, while the nonlinear operations allow neural networks to learn complex relationships [23].

3.5.1 Representation Learning

For high performance in applied settings, classic machine learning models, such as linear regression, decision trees, and support vector machines, often rely heavily on manual, problem-specific feature engineering. Although they are effective for structured problems (e.g., predicting housing prices from other metrics), these models on their own struggle with unstructured data (e.g., images, text, audio), where meaningful features are often subtle, hierarchical, or context-dependent. Crafting such features requires domain expertise, trial and error, and computational resources, which limit scalability and practical applicability.

Neural networks sidestep this bottleneck by learning feature representations $f_i(\tilde{\mathbf{x}}_i, \theta_i)$ (for $i \in \{1, 2, \ldots, n\}$) directly from raw data. Unlike other machine learning models, neural networks process minimally preprocessed inputs (e.g., pixel values, word segments, audio clips) and iteratively refine features through multiple layers of computation. By learning its own feature representations, neural networks require less feature engineering to perform well, in exchange for requiring larger datasets and more computation to train [23].

Said trade-off can be adjusted based on the model's complexity, which increases for larger number of layers n and more parameters in the neural network. Theoretically, a sufficiently complex neural network can model any arbitrary relationship in data due to the universal approximation theorem [1, p. 4], which enables neural networks to be widely applicable.

3.5.2 Backpropagation

To improve the accuracy of a neural network, its parameters $\theta_1, \theta_2, \ldots, \theta_n$ are adjusted in a process known as backpropagation, which computes the gradient of the network's loss function $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$ with respect to each parameter $(\theta_i)_j \in \mathbb{R}$ in the network. These gradients are then used to update the parameters via an optimization algorithm such as gradient descent.

The backpropagation process can be broken down into two main phases. Firstly, a forward pass is made, whereby the network outputs a prediction $\hat{\mathbf{y}}$ with which its corresponding loss $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$ is computed based on the expected output $\mathbf{y}$; for instance, the network could have an MSE loss function $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2}(\hat{\mathbf{y}} - \mathbf{y})^2$. Secondly, a backward pass is done, whereby the gradients of the loss with respect to each parameter are computed using the chain rule of calculus. The chain rule allows us to decompose the gradient of the loss $\mathcal{L}$ with respect to a parameter $(\theta_i)_j \in \mathbb{R}$ into the product of gradients across the layers. Concretely, for a given layer i, the gradient of the loss $\mathcal{L}$ with respect to its parameters θ_i can be expressed as below.

$$\frac{\partial \mathcal{L}}{\partial \theta_i} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \theta_i} \tag{3}$$

where $\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}}$ is the gradient of the loss with respect to the network's output, and $\frac{\partial \hat{\mathbf{y}}}{\partial \theta_i}$ is the gradient of the network's output with respect to the parameters θ_i.

Since the output of one layer serves as the input to the next, the gradients can be computed recursively. Starting from the last layer, the gradient of the loss with respect to the output of layer i is given by the following:

$$\frac{\partial \mathcal{L}}{\partial f_i} = \frac{\partial \mathcal{L}}{\partial f_{i+1}} \frac{\partial f_{i+1}}{\partial f_i} \tag{4}$$

where $\frac{\partial \mathcal{L}}{\partial f_{i+1}}$ is the gradient of the loss with respect to the output of layer $i+1$, and $\frac{\partial f_{i+1}}{\partial f_i}$ is the gradient of the output of layer $i+1$ with respect to the output of layer i. By recursively applying the chain rule, the gradients for each layer are computed, starting from the last layer and moving backward through the network.

Once the gradients $\frac{\partial \mathcal{L}}{\partial \theta_i}$ are computed, the parameters θ_i are updated using an optimization algorithm such as gradient descent. The update rule for each parameter θ_i is given by the following:

$$\theta_{i+1} \leftarrow \theta_i - \alpha \frac{\partial \mathcal{L}}{\partial \theta_i} \tag{5}$$

where $\alpha \in \mathbb{R}$ is the learning rate, a hyperparameter that controls how much θ_i is updated.

One iteration of forward pass, backward pass, and parameter update counts as one epoch. This process continues for multiple epochs, until some end condition is met (e.g., a set number of epochs, $\frac{\partial \mathcal{L}}{\partial \theta_i}$ under a set threshold). Backpropagation allows for the efficient training of neural networks with many parameters.

3.6 Transformers

Transformers [41] have emerged as a prevalent application of neural networks in recent years, seeing wide use in natural language processing, computer vision, and audio processing tasks [24]. As many large language models (LLMs) are essentially Transformers applied to natural language problems, an overview over the architecture of Transformers is essential in understanding how the LLMs integrated with our calculator output the desired text. As will be shown in Section 3.6.3, a key strength of transformers compared to alternative architectures is their ability to isolate salient details from large amounts of information [15].

3.6.1 Data Representation - Tokenization and Embeddings

A transformer model processes a sequence of data. However, many types of data are not explicitly sequences. Tokenization is a process that splits a piece of data into a sequence of smaller chunks, called tokens. For instance, the text "the quick brown fox jumps over the lazy dog" can be split into the sequence of tokens $\tau =$ ("the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"). Furthermore, the data that transformers take in, process, and output are commonly represented as vectors ("embeddings") in

the same vector space ("embedding space"), in which each dimension of these vectors captures a semantic meaning [30]. Concretely, each token $t \in \tau$ is mapped to a one-hot vector $v \in \mathbb{R}^{d_{\text{embedding}}}$, where $d_{\text{embedding}}$ can be, for instance, the number of distinct tokens in the sequence. Continuing the previous example, the sequence of tokens τ (of length 9) can be mapped to a sequence of embeddings $\xi = (\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_4, \mathbf{e}_5, \mathbf{e}_6, \mathbf{e}_1, \mathbf{e}_7, \mathbf{e}_8)$, where $\mathbf{e}_k \in \mathbb{R}^{d_{\text{embedding}}}$ denotes a vector with 1 in the kth entry and 0's elsewhere. Equivalently, τ can be mapped to $E = \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_8 \end{bmatrix}^\intercal \in \mathbb{R}^{9 \times d_{\text{embedding}}}$. Transformers iteratively refine the initial embeddings through each of its layers[7]. By having the initial, intermediate, and final embeddings all members of the same vector space, their dimensions encode the same semantic information.

3.6.2 Positional Encoding

A key advantage of transformers lies in their capability to accelerate the processing of vast amounts of data by handling numerous embeddings in parallel [36]. However, this approach requires the transformer to know the position of an embedding within a sequence without reference to that sequence. For this, the entries of each embedding are adjusted to bake-in the embedding's positional information. The below example aims to illustrate the need for this step.

3.6.2.1 Example

Let $\tau_1 =$ ("dog", "bites", "man") and $\tau_2 =$ ("man", "bites", "dog") be input sequences, with corresponding embeddings $E_1 := \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \end{bmatrix}^\intercal, E_2 := \begin{bmatrix} \mathbf{e}_3 & \mathbf{e}_2 & \mathbf{e}_1 \end{bmatrix}^\intercal \in \mathbb{R}^{3 \times d_{\text{embedding}}=4}$. Though it is clear that the different positioning of "dog" and "man" is essential to the meaning of the sentence (and thus the transformer should treat E_1 and E_2 differently), a key downstream operation is invariant to the order of these tokens[5], and thus will treat E_1 and E_2 as the same (see Section 3.6.3). One way of amending this is to augment E_1 and E_2 to yield new embeddings such that the position of a token is known by looking at just the token, without the need to look at the entire sequence.

$$E_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \text{ and } E_2 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Each row represents a token, and the exact ordering of the rows is ignored by transformers. To make the transformers sensitive to the exact ordering of the rows, one possible approach is to add 0.11 to all entries in the first row, 0.22 to all entries in the second row, and 0.33 to all entries in the third row. Below are the

modified matrices, denoted $p(E_1)$ and $p(E_2)$.

$$p(E_1) = \begin{bmatrix} 1.11 & 0.11 & 0.11 & 0.11 \\ 0.22 & 1.22 & 0.22 & 0.22 \\ 0.33 & 0.33 & 1.33 & 0.33 \end{bmatrix} \text{ and } p(E_2) = \begin{bmatrix} 0.11 & 0.11 & 1.11 & 0.11 \\ 0.22 & 1.22 & 0.22 & 0.22 \\ 1.33 & 0.33 & 0.33 & 0.33 \end{bmatrix}$$

Now, if a random row is selected, say...

$$\begin{bmatrix} 1.33 & 0.33 & 0.33 & 0.33 \end{bmatrix}$$

It is clear that this is $\mathbf{e}_1 + .33$; i.e., the embedding corresponding to "dog" positioned as the third word of the sequence. With this, the Transformer has information on the positioning of each token, without reference to the entire embedding matrix. Note that in practice, more sophisticated approaches are employed (e.g., sinusoidal, rotary, FLOATER)[5].

3.6.3 Attention Mechanism

Processing sequential data, such as text relevant to Large Language Models (LLMs) (see Section 3.6), presents a unique challenge: understanding how different parts of the sequence relate to each other [15]. For instance, in the sentence "The dog chased the ball until it was tired," understanding what "it" refers to requires linking it back to "The dog". Simple sequential processing might struggle with such long-range dependencies or identifying the most relevant contextual words for understanding a specific word [24]. The attention mechanism, a core component of the Transformer architecture used in many LLMs [41], provides a mathematical framework to address this. It allows the model to dynamically assess and quantify the relevance between all pairs of elements (tokens) in an input sequence when generating an updated representation for each element. This ability is rooted in calculating pairwise relevance scores derived from dot products, transforming these scores into probability distributions, and using these probabilities to weigh the contribution of each token's information. Furthermore, its design is computationally advantageous, enabling significant parallel processing, which is crucial for training models on the vast datasets required for LLMs.

The attention mechanism starts with an input sequence represented as a matrix of token embeddings, $E \in \mathbb{R}^{n \times d_{\text{embedding}}}$, where n is the sequence length, and $d_{\text{embedding}}$ is the dimension of each token's vector representation (as described in Section 3.6.1).

Instead of using the raw embeddings directly for all calculations, the attention mechanism first transforms them using linear projections. Three distinct weight matrices—$W_Q \in \mathbb{R}^{d_{\text{embedding}} \times d_k}$, $W_K \in \mathbb{R}^{d_{\text{embedding}} \times d_k}$, and $W_V \in \mathbb{R}^{d_{\text{embedding}} \times d_v}$—are introduced. These are parameter matrices, the entries of which are learned from MLPs (see Section 3.5). The input embedding matrix E is multiplied by each of these weight matrices to create three specialized representations:

- $Q = EW_Q \in \mathbb{R}^{n \times d_k}$
- $K = EW_K \in \mathbb{R}^{n \times d_k}$
- $V = EW_V \in \mathbb{R}^{n \times d_v}$

Concretely, this projects the original $n \times d_{\text{embedding}}$ embedding matrix into three new matrices (Q, K, V), potentially with different dimensions (d_k and d_v, often $d_k = d_v \leq d_{\text{embedding}}$). By learning different weight matrices (W_Q, W_K, W_V), the model can learn to use different aspects of the original token embeddings for different roles in the attention calculation. This separation enables the model to learn complex interaction patterns within the sequence.

To quantify the relationship between every pair of tokens, the mechanism relies fundamentally on the dot products between the Q and K matrices. This calculation, performed efficiently for all pairs via matrix multiplication, yields a matrix of raw scores:

$$S = QK^\intercal \tag{6}$$

Here, each entry S_{ij} in the resulting score matrix $S \in \mathbb{R}^{n \times n}$ depends on the dot product between the vectors Q_{i*} and ${K_{j*}}^\intercal$. This dot product serves as an unnormalized score indicating the initial assessment of relevance or compatibility between each possible pair of tokens i, j. Before converting these scores into probabilities, they are typically scaled for numerical stability during training:

$$\tilde{S} = \frac{QK^\intercal}{\sqrt{d_k}} \tag{7}$$

The division by $\sqrt{d_k}$ helps prevent the entries S_{ij} helps prevent an issue during backpropagation (see Section 3.5), known as vanishing gradients [41], where the values update in very small increments, which slows the training process.

Afterwards, the dot-product interactions between Q and K can be represented as probabilities by applying a so-called softmax function along each row $\tilde{S}_{i*}$, as shown below:

$$A = \text{softmax}\left(\tilde{S}\right) \tag{8}$$

The softmax function transforms each vector $\tilde{S}_{i*}$ into a probability distribution A_{i*}, where its entries are nonzero and add to 1. Crucially, each entry A_{ij} can be interpreted as relative weights, where larger entries correspond to greater relevance between tokens i and j. A higher score on the dot product matrix S_{ij} corresponds to a higher probability A_{ij} after the row-wise softmax normalization.

Finally, these row-wise probability distributions A_{i*} are used to compute the updated representation for each token. This is achieved by calculating a weighted sum of entries of V, where the weights for updating the i-th token are given by the i-th row of A:

$$H = AV = \text{softmax}\left(\frac{QK^\intercal}{\sqrt{d_k}}\right)V \tag{9}$$

The output matrix $H \in \mathbb{R}^{n \times d_v}$ contains the new, context-aware representations. Concretely, each row H_{i*} is computed as $H_{i*} = \sum_{j=1}^{n} A_{ij} V_{j*}$. This means the updated representation for token i—H_{i*}—is formed by aggregating the vectors V_{j*} of all tokens in the sequence. Each V_{j*} contributes to this sum in proportion to A_{ij}, which represents the relevance of token j to token i, as determined by the initial dot product between Q and K. This allows the model to selectively synthesize information from across the entire sequence based on these learned, probabilistic relevance weights. These representations H are then passed on to further layers within the Transformer model.

3.6.4 Multi-Head Attention

The attention mechanism described in Section 3.6.3 allows the model to weigh the relevance of different tokens when creating updated representations. However, it does so using a single set of learned projection matrices (W_Q, W_K, W_V). This means the model learns one specific way to project the input embeddings and calculate relevance scores. Yet, relationships between tokens in a sequence can be multifaceted. For example, one token might be relevant to another due to syntactic structure, while also being relevant due to semantic similarity. A single attention calculation might struggle to capture these diverse types of relationships simultaneously, potentially averaging them out.

Multi-Head Attention (MHA) addresses this limitation by performing multiple attention calculations in parallel, each focusing on potentially different aspects of the token relationships. Instead of just one set of projection matrices, MHA utilizes h independent sets, where h is the number of "attention heads".

The core idea is to allow the model to jointly attend to information from several different representations. With multiple heads, the model can learn different types of relevance patterns concurrently. Each head can specialize in capturing a particular kind of interaction.

Formally, MHA starts with the same input embedding matrix $E \in \mathbb{R}^{n \times d_{\text{embedding}}}$. For each head $i \in \{1, 2, \ldots, h\}$, a distinct set of learned projection matrices is used:

- $W_{Q,i} \in \mathbb{R}^{d_{\text{embedding}} \times d_k^*}$: Query projection matrix for head i.
- $W_{K,i} \in \mathbb{R}^{d_{\text{embedding}} \times d_k^*}$: Key projection matrix for head i.
- $W_{V,i} \in \mathbb{R}^{d_{\text{embedding}} \times d_v^*}$: Value projection matrix for head i.

These parameter matrices are learned during training via backpropagation, just like the single set Q, K, V using only one attention mechanism (as shown in Section 3.6.3).

A common practice, primarily for computational efficiency, is to set the output dimensions for each head such that the total computational cost is similar to that of a single attention mechanism with the full dimensions (d_k, d_v from Section 3.6.3). Specifically, the dimensions are often chosen as $d_k^* = d_k/h$ and $d_v^* = d_v/h$. This way, each head works with smaller dimensional projections, but the combined capacity across all heads is maintained. For each head i, the input embeddings E are projected using its specific matrices:

- $Q_i = EW_{Q,i} \in \mathbb{R}^{n \times d_k^*}$
- $K_i = EW_{K,i} \in \mathbb{R}^{n \times d_k^*}$
- $V_i = EW_{V,i} \in \mathbb{R}^{n \times d_v^*}$

Then, the attention mechanism (Equation 9) is applied independently for each head using its respective Q_i, K_i, V_i. Note that the scaling factor now uses the head-specific dimension d_k^*:

$$H_i = \text{softmax}\left(\frac{Q_i {K_i}^\intercal}{\sqrt{d_k^*}}\right) V_i \tag{10}$$

Each head i produces an output matrix $H_i \in \mathbb{R}^{n \times d_v^*}$. Since these calculations are independent for each head, they can be performed in parallel, which is computationally advantageous.

After computing the outputs for all h heads, their results $(H_1, H_2, \ldots, H_h)$ need to be combined to produce a single output tensor that captures the information learned across all heads. This is achieved by

concatenating the head outputs along the columns:

$$C(H_1, H_2, \ldots, H_h) = \begin{bmatrix} H_1 & H_2 & \ldots & H_h \end{bmatrix} \in \mathbb{R}^{n \times d_v} \tag{11}$$

Finally, the concatenated output is passed through one more linear projection, governed by another learned weight matrix $W_O \in \mathbb{R}^{h \cdot d_v^* \times d_{\text{embedding}}}$. Often, the final output dimension $d_{\text{embedding}}$ is chosen to be the original embedding dimension $d_{\text{embedding}}$, allowing the output of the MHA layer to have the same shape as its input, facilitating stacking multiple layers in the Transformer architecture.

$$M(E) = C(H_1, H_2, \ldots, H_h)W_O \tag{12}$$

This final projection allows the model to learn how to best combine and mix the information gathered from the different attention heads. By enabling the model to learn multiple ways of assessing token relevance in parallel and combining these perspectives, Multi-Head Attention allows Transformers to build richer, more nuanced representations of sequential data, capturing complex dependencies essential for understanding language.

3.6.5 Overall Architecture

Having detailed the core components such as tokenization, positional encoding, and Multi-Head Attention (MHA), we now outline how these pieces fit together within a standard Transformer layer and how the model generates output sequentially. Much like how neural networks stack layers (Section 3.5), Transformer models typically stack multiple identical "Transformer blocks" [41].

Each Transformer block processes an input sequence of embeddings $E_{in} \in \mathbb{R}^{n \times d_{\text{embedding}}}$ and outputs a sequence of refined embeddings $E_{out} \in \mathbb{R}^{n \times d_{\text{embedding}}}$ of the same dimension. A common structure for such a block involves two main sub-layers: a Multi-Head Attention (MHA) sub-layer, then a Feed-Forward Network (FFN) sub-layer.

First, the input embeddings E_{in} are processed by the MHA mechanism (Section 3.6.4) to produce context-aware representations $M(E_{in})$. This output is then combined with the original input E_{in} using a residual connection [11] and passed through Layer Normalization [41] . This operation can be expressed as the following:

$$E_{intermediate} = \text{LayerNorm}(E_{in} + M(E_{in})) \tag{13}$$

The residual connection ($E_{in} + M(E_{in})$) helps gradients flow during backpropagation (Section 3.5) in deep networks, making training easier. Concretely, the residual connection enables $M(E_{in})$ to be treated as a transformation that refines the input. If no changes made to the input results in optimal performance, Transformers can simply learn to set $M(E_{in}) = 0$ through backpropagation. After the residual connection, Layer Normalization normalizes entries of the resulting matrix to have a mean of 0 and a standard deviation of 1, enabling a faster and more reliable training process.

The intermediate embeddings $E_{intermediate}$ are then processed by a position-wise Feed-Forward Network (FFN). This FFN typically consists of two linear transformations with a non-linear activation function (like ReLU) in between, applied independently to each position (embedding) in the sequence. Similar to the MHA sub-layer, a residual connection and Layer Normalization are applied:

$$E_{out} = \text{LayerNorm}(E_{intermediate} + \text{FFN}(E_{intermediate})) \tag{14}$$

where the FFN sub-layer further processes the representations generated by the attention mechanism. The output E_{out} of one Transformer block serves as the input E_{in} for the next block. This stacking allows the model to build increasingly complex and abstract representations of the input sequence.

Crucially, for tasks such as text generation or sequence prediction, which are relevant to Large Language Models (Section 3.6), the Transformer operates in an autoregressive manner. This means it generates the output sequence one token at a time, using the previously generated tokens as input for predicting the next token. Let the input sequence of tokens at timestep t be $\tau_t = (t_1, t_2, \ldots, t_t)$, represented by their embeddings $E_t \in \mathbb{R}^{t \times d_{\text{embedding}}}$ (after including positional encoding). This sequence is passed through the stack of Transformer blocks.

To predict the next token t_{t+1}, the model typically uses the output embedding $\mathbf{h}_t \in \mathbb{R}^{d_{\text{embedding}}}$ corresponding to the last input token t_t from the final Transformer block. This vector $\mathbf{h}_t$ is then passed through a final linear layer (parameterized by a weight matrix $W_{\text{vocab}} \in \mathbb{R}^{\mathcal{V} \times d_{\text{embedding}}}$ and bias vector $\mathbf{b}_{\text{vocab}} \in \mathbb{R}^{\mathcal{V}}$, where $\mathcal{V}$ is the size of the model's vocabulary, i.e., the total number of possible unique tokens it can predict). This produces a vector of raw scores, called logits, $\mathbf{l}_t \in \mathbb{R}^{\mathcal{V}}$:

$$\mathbf{l}_t = W_{\text{vocab}}\mathbf{h}_t + \mathbf{b}_{\text{vocab}} \tag{15}$$

Each element $(\mathbf{l}_t)_j$ in the logits vector corresponds to a potential next token j in the vocabulary. These logits represent unnormalized prediction scores. To convert these scores into a probability distribution over

the entire vocabulary, the softmax function is applied:

$$\mathbf{p}_t = \text{softmax}(\mathbf{l}_t) \tag{16}$$

The resulting vector $\mathbf{p}_t \in \mathbb{R}^{\mathcal{V}}$ contains probabilities, where each element $(\mathbf{p}_t)_j \geq 0$ and $\sum_{j=1}^{\mathcal{V}} (\mathbf{p}_t)_j = 1$. The value $(\mathbf{p}_t)_j$ represents the model's predicted probability that the token j is the correct next token in the sequence.

Finally, a specific token t_{t+1} is chosen based on this probability distribution $\mathbf{p}_t$. The most straightforward approach is sampling the token with the highest probability, but more sophisticated methods can be employed [14]. Once t_{t+1} is selected, it is appended to the input sequence ($\tau_{t+1} = (t_1, \ldots, t_t, t_{t+1})$), and the entire process repeats to predict t_{t+2}, continuing until a special end-of-sequence token is generated or a maximum length is reached. This step-by-step generation process, underpinned by the calculation of token probabilities at each step, is fundamental to how Transformer-based LLMs produce coherent and contextually relevant sequences of text. The parameters within the MHA layers, FFNs, and the final linear layer (W_Q, W_K, W_V, W_O, FFN weights, $W_{\text{vocab}}, \mathbf{b}_{\text{vocab}}$, etc.) are all learned during the model training and finetuning via backpropagation.

4 Methods

4.1 Architecture

The core architecture of our system integrates LLMs (see Section 3.6) with a specialized computational engine to solve word problems featuring multiple equations with unknown variables. We employ a multistage pipeline with four primary components to process and solve mathematical problems:

1. Word Problem Parser
2. Variable Matching System
3. Equation Formation Module
4. Computational Engine

4.2 Word Problem Parser

The Word Problem Parser is responsible for extracting key information from an algebraic word problem (see Section 3.3), outputting the given variables and the wanted variables in a structured format. The parser operates by taking in a user-provided word problem (see Section 3.3) and passing it through a fine-tuned Large Language Model (LLM). The model is finetuned to output structured information when provided a word prompt (see Section 4.2.2 for more details).

By automating this initial parsing stage, we reduce user effort and improve accuracy in problem solving. This ensures that the system correctly interprets a wide variety of physics and engineering word problems, making the computational workflow more efficient and user-friendly.

4.2.1 Processing Workflow

When a user inputs a word problem, the system follows a structured process to extract meaningful information. First, the input is formatted to ensure consistency before being processed by the LLM. Next, the LLM identifies key variables, distinguishing between given quantities (including their numerical values and units) and the desired unknowns. Finally, the extracted data is formatted to be used by the Variable Matching System (see Section 4.3). For illustration, consider the following example.

Example Word Problem

An engineer is testing a motor. The work done by the motor is **100 Joules**, the torque is **10 Newton-meters**, and the test runs for **10 seconds**. Calculate the **angular velocity (ω)** in radians per second.

Figure 3: Above is an example word problem, where the given and wanted variables are explicitly written (the bold formatting is for presentation purposes). As specified in Section 3.3, the kinds of word problems the LLM is expected to handle are those which explicitly state all given variables—variables with associated numerical values—and state a wanted variable—a variable to solve for in terms of the given variables.

Example Output

- **Givens:** $W = 100$ Joules, $\tau = 10$ Newton-meters, $t = 10$ seconds
- **Wanted:** ω (angular velocity)

Figure 4: Above is an example response from the finetuned LLM to the word problem in Figure 3. This structured output would then be passed to the next stage of the calculator (see Section 4.4). During fine-tuning (see Section 4.2.2), several pairs of example word problems and example outputs are provided to serve as training data.

4.2.2 LLM-Based Extraction and Fine-Tuning

To enhance the accuracy of variable extraction from physics word problems, we fine-tuned the pretrained models Llama 3.2 1B [28] and Llama 3.2 3B models using a dataset of 110 problems. Each problem was paired with a formatted output containing the extracted givens and unknowns (see Figure 5). The fine-tuning process followed Supervised Fine-Tuning (SFT) [2], ensuring that the model learned structured mappings from the problems to the extracted values.

Example Input Output Pair

Input:	**Output:**
An engineer is testing a new type of motor and has collected the following data: the work done by the motor is 100 Joules, it operates under a torque of 10 Newton-meters, and the test runs for a total of 10 seconds. Calculate the angular velocity (ω) of the motor in radians per second.	Givens: work = 100 [Joules] ; torque = 10 [Newton-meters] ; time = 10 [seconds], Wanted: ω.

Figure 5: An example word problem with its expected output ("input-output pair"). All example outputs conform to the same structured format, enabling text-based pattern matching to parse the information from the output of the LLM.

The finetuned Llama models are autoregressive, meaning that they compute a prediction $\hat{\mathbf{y}} \in \mathbb{R}^{d_{\text{output}}}$ given a sequence of inputs $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) \in \mathbb{R}^{d_{\text{input}}}$. To accommodate for this, each input-output example is reformatted as a single body of text, which corresponds to a single sequence of tokens (see 3.6.1). The training utilized a chat-based template where a system prompt defines extraction guidelines, a user prompt contained the problem statement, and the assistant response provided the extracted values. An example is shown in Figure 6.

Training Sample

System: You are a helpful chatbot that follows directions. For a given word problem, provide the knowns, unknowns, and wanted. Do not attempt to compute the answer.

User: An engineer is testing a new type of motor and has collected the following data: the work done by the motor is 100 Joules, it operates under a torque of 10 Newton-meters, and the test runs for a total of 10 seconds. Calculate the angular velocity (ω) of the motor in radians per second.

Assistant: Givens: work = 100 [Joules] ; torque = 10 [Newton-meters] ; time = 10 [seconds]. Wanted: ω.

Figure 6: The above is a formatted version of the labeled data shown in Figure 5). This single training example is explicitly put in terms of a single sequence of characters to align with the auto-regressive nature of the particular LLMs employed (Llama 3.2 1B and 3B models), where at each step the LLM predicts the next character given all previous characters of the string it has seen so far. Here, the Large Language Model would be given the beginning part of the string ending in "Assistant:", from which the model is expected to generate the sequence of characters that follow.

By structuring each example and its corresponding label as a formatted single body of text, the model can be trained to autoregressively output the text for the "assistant" role, given all of the other previous text. Finally, the dialogue-based template ensures consistency for more efficient model training.

To efficiently fine-tune the model, we utilize Quantized Low-Rank Adaptation (QLoRA) [12], which allows training with significantly reduced memory requirements while maintaining performance. QLoRA involves freezing the pretrained model weights and injecting trainable low-rank adapter layers, which are optimized during fine-tuning. Additionally, 4-bit quantization is employed to further reduce memory consumption by lowering the precision of model parameters, enabling effective training on consumer-grade GPUs.

By structuring each example and its corresponding label as a formatted single body of text, the model can be trained to autoregressively output the text for the "assistant" role, given all of the other previous text. Finally, the dialogue-based template ensures consistency for more efficient model training.

4.3 Variable Matching System

Our solver contains a domain-specific database of mathematical symbols and equations. These equations and variables are loaded into a solving graph with each variable and equation being represented by a node with edges pointing from equations to each variable within and vice versa. After extracting the list of variables from the LLM, our system matches each variable to the known variable in our database. This includes multiple nomenclature variations for each variable. In order to achieve this, we maintain a database of synonyms and abbreviations for each term. We then generated a dataset of further possible abbreviations and trained a Machine Learning-based word auto-completer (see Section 5.4) to use for matching different possible synonyms and abbreviations to each respective term. Then the variables in the solving graph are updated with the values extracted from the text. Afterwards, the unknown variables are fed into the computational engine in order to solve.

4.3.1 Machine Learning Implementation

The Variable Matching System described above relies on accurately mapping variable representations extracted from user input—which might contain typos, abbreviations, or other variations—to the canonical, standardized terms stored in our domain-specific database (the "vocabulary bank"). To accomplish this robustly, we employ a machine learning model specifically trained for this matching task.

The model's core function is to take an input variant and determine which, if any, of the known vocabulary terms it most likely represents. This is effectively a spellchecking and abbreviation-expansion task tailored to the domain of physics variables. We frame this formally as a multi-class classification problem, where each standardized term in our vocabulary bank constitutes a unique class.

A critical requirement is handling variants that do not correspond to any term in our known vocabulary. Instead of forcing an incorrect match, the model must be able to indicate uncertainty or non-correspondence. Therefore, we include an additional "No Guess" category as a possible output. This allows the model to abstain from making a prediction when the input variant is too ambiguous or doesn't sufficiently resemble any known term.

Concretely, let $\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$ be the set of standardized vocabulary terms. Given an input variant u which may or may not correspond to a term in $\mathcal{T}$, we seek a model function $f(T, u)$ that outputs the corresponding term $t \in \mathcal{T}$ if u is identified as a variant of t, or "No Guess" otherwise. For this model $f(\mathcal{T}, u)$ to be effective, it must exhibit two key behaviors. Firstly, it should correctly classify previously

unseen novel variants (i.e. misspellings, abbreviations) of known terms $t \in \mathcal{T}$. Secondly, it should reliably output "No Guess" for variants u that do not correspond to any term in the vocabulary bank $\mathcal{T}$, either when u represents some term $\Xi \notin \mathcal{T}$ or when u is non-sensical. To enable the "No Guess" capability and handle ambiguity, the model's prediction mechanism incorporates confidence scores. For each potential class k (corresponding to term t_k), the model calculates a confidence value $\mathfrak{c}_k \in \mathbb{R}[0, 1]$. If the highest confidence score among all classes k does not meet a specified threshold $\mathfrak{t} \in \mathbb{R}[0, 1]$, the model outputs "No Guess". This threshold acts as a safeguard against low-certainty classifications, directly addressing the second desired behavior.

4.3.2 Data Preparation

To prepare the vocabulary data for the machine learning model, we perform several preprocessing and augmentation steps. The process begins with a curated list of standard physics terms (e.g., "Spring-Constant", "PositionX_f") sourced from introductory physics courses. These terms are then standardized to ensure consistency and to eliminate superficial variations. To standardize a term, any spaces and special characters are removed, then the remaining characters are converted to lowercase. The first two columns of Table 2 illustrate this transformation.

Original Terms	Standardized Terms	Example Variants
SpringConstant	springconstant	scpringconsltdant, spgconstant, xsprizgcontant, s
k_{spring}	kspring	ksprinin, kspri, kspr, kmspritg, k
PositionX_f	positionxf	positnxf, phositionoxf, dtposiptionxf
vi	vi	vih, ni, li, jvi, vi

Table 2: The above depicts the process of generating dataset of misspellings and abbreviations for the machine learning model employed in the Variable Matching System (see Section 4.3). First, the terms that in the original dataset are standardized, where all characters are lowercase and special characters are removed. Then, from these standardized terms, misspellings and abbreviations (collectively referred to as "variants") are generated.

To make the model robust to common user input variations, such as typos and abbreviations, the standardized dataset is augmented. Both abbreviations and misspellings are generated, as detailed below. To generate an abbreviation, a term is shortened to using only the first n characters, for arbitrary n. For example, abbreviation(velocity)=vel ($n = 3$). For misspellings, we generate variations of the following types:

- Substitution: we randomly choose and replace character(s) in our word.

- Deletion: we randomly choose character(s) in our word to delete.
- Insertion: we randomly choose an index position and a character to insert there.
- Mixing: the word is subject to a combination of substitution, deletion, and insertion.

The frequency and type of these errors are controlled by predefined percentages relative to the word length. Examples of these generated "variants" are shown in the third column of Table 2.

After the dataset is augmented with variant terms, each term (original or variant) ς of length $m_\varsigma \in \mathbb{R}$ is converted into a corresponding vector representation ("embedding") $\mathbf{z}_\varsigma \in \mathbb{R}^{m_\varsigma}$. We use the CountVectorizer [21] technique, focusing on character subsequences (similar to character n-grams [18]). Specifically, we count the occurrences of all possible character sub-sequences of lengths 1,2, and 3 within each word ς. For example, the term "acceleration" would have its corresponding embedding $\mathbf{z}_\varsigma$ contain counts for "a" (2), "c" (2), "e" (2), ..., "ac" (1), "cc" (1), ..., "acc" (1), "cce" (1), etc. for all sub-sequences of length up to 3. Sub-sequences longer than 3 (e.g., "acce") are excluded. This limit (length of 3) is chosen as a balance: it captures significant structural information while preventing the feature space from becoming excessively sparse and large. For a standard English alphabet of $\mathfrak{p} = 26$ characters, the maximum number of possible sub-sequences up to length 3 is $\varpi = 26 + 26^2 + 26^3 = 18278$. Since the CountVectorizer approach of generating embeddings maps each distinct character sub-sequence in the dataset to a separate dimension, the embedding space can have as many as $\varpi = 18278$ possible dimensions. While this high dimensionality can help separate different words effectively, it can also lead to computational inefficiency and potential overfitting issues associated with the Curse of Dimensionality [3], particularly for distance-based models (see Section 4.3.3). To mitigate this, we apply Truncated Singular Value Decomposition (SVD) [22] to the embeddings. Truncated SVD projects the high-dimensional sparse vectors onto a lower-dimensional space by discarding the dimensions corresponding to the smallest singular values, thus creating denser representations while retaining most of the important variance in the data.

The outcome of this data preparation pipeline is a dataset where every original term and generated variant is represented as a dense numerical vector in a lower-dimensional space. This vector representation captures orthographic similarities, enabling the subsequent classification model to measure the "closeness" between an input variant and the known vocabulary terms. For example, the vector for "velocity" will be numerically closer to that of its misspelling "vellocity" compared to that of "acceleration".

4.3.3 Support Vector Machines-based Classification

To classify input variants (represented by the dense vectors generated during data preparation) into one of the known vocabulary terms or a "No Guess" category, we employ a classification model based on Support Vector Machines (SVM). We utilize the `sklearn.svm.SVC` implementation configured with a linear kernel. At its core, SVM is a binary classification algorithm. Given data points belonging to two classes in the embedding space, SVM seeks to find an optimal hyperplane decision boundary that best separates the two classes. This optimality is achieved by maximizing the margin—the distance between the decision boundary and the nearest data points from either class.

SVM naturally provides a measure of confidence for its predictions. A data point's distance to the decision boundary serves as an indicator of classification certainty. Points lying far from the boundary are classified with high confidence, whereas points close to the boundary are considered more ambiguous and receive lower confidence scores.

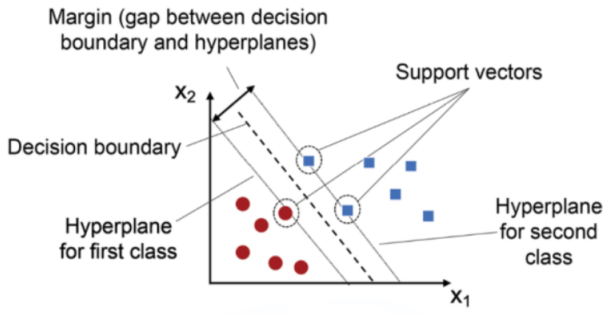

Figure 7: Taken from [20]. SVM is a model that assigns a binary prediction and a confidence score to each data point. Data points closer to the decision boundary are assigned lower confidence scores. During training, SVM with a linear kernel seeks to find the optimal hyperplane decision boundary that best separates the two classes.

Since our problem involves classifying variants into multiple possible vocabulary terms, the inherently binary SVM needs to be adapted. The `SVC` implementation handles this using a One-versus-One strategy: for a problem with $\mathfrak{K}$ distinct classes, One-versus-One involves training $\frac{\mathfrak{K}(\mathfrak{K}-1)}{2}$ separate binary SVM classifiers. Each classifier is trained on data from only two specific classes, learning to distinguish between that pair. During prediction for a new data point, all of these pairwise classifiers are applied. Each classifier "votes" for one of the two classes it was trained on. The final predicted class for the input variant is the one that receives the highest number of votes across all pairwise classifiers.

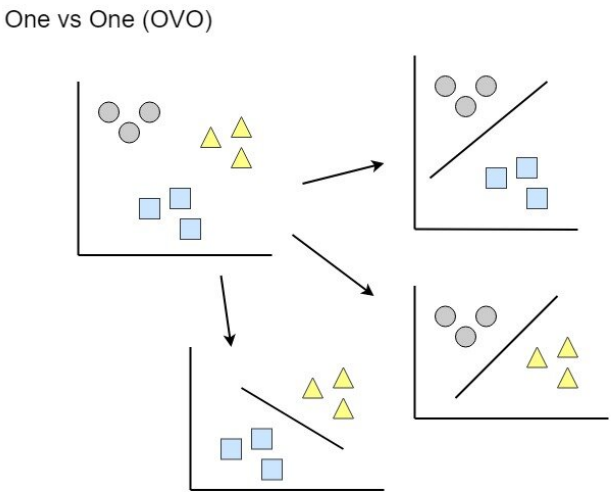

Figure 8: Taken from [37]. The above depicts SVM employed in a One-versus-One approach, which enables SVM, which inherently provides a binary prediction, to be employed in a problem with more than two classes. The approach involves training a separate classifier for each possible pair of classes in the dataset. Then, when predicting the class of a data point during inference, each classifier casts a "vote" based on which of its two classes it predicts, then the data point is assigned to the class that receives the most votes.

By using the One-versus-One approach on the prepared vector representations, the system maps potentially noisy input variant terms to the most likely term in the predefined physics vocabulary, leveraging the margin-maximization properties of SVM for robust classification and its distance-based confidence scores to handle ambiguous cases or inputs not matching any unknown term.

4.4 Computational Engine

Our computational engine utilizes the solving graph containing domain-specific equations and variables and the Python library SymPy [29] (see Section 4.6) to solve for the unknown variables with steps. While SymPy can solve a moderate-size system of equations, it cannot provide step-by-step solutions for doing so [29]. Additionally, to mitigate hallucinations, we wanted to limit the use of LLMs to extracting variables from word problems (see Section 4.2) and to convert the calculator's output to a natural language explanation, and instead delegate tasks important to the calculator's accuracy, such as identifying relevant equations, to a deterministic graph traversal algorithm. Delegating the process of extracting variables from word problems to LLMs is possible for domains such as physics and general mechanics, where variables such as angular velocity have a set of standard equations that can be used to solve them.

Based on a data set of predefined equation-strings (e.g., "x + y = z"), a bipartite graph of equations and variable nodes is constructed as specified in Section 3.2. The following provides technical details on the construction of the graph. For each equation-string, we create one equation node and, for each variable in

the equation, a variable node. For example, "x + y = z" would create a corresponding equation node as well as variable nodes "x", "y", and "z". Each equation node consists of the equation, domain, and a list of edges, each pointing to a corresponding variable node. Each variable node consists of the variable's name, its value, and a list of edges, with each edge connecting to a equation node that contains the variable.

When solving for values of variables in a given system of equations, the calculator first substitutes all known information into the graph (see Section 4.2). Then, to solve for a specific variable, we first construct its dependency graph—a minimal-cost subgraph that includes all equations and variables required to compute the target variable from known values. To generate the dependency graph, we utilize a dependency-guided variant of Uniform Cost Search (UCS) [34]. The cost is the number of unresolved variables in an equation, and we track cumulative cost as the total number of remaining unknowns. The edges of equations to variables are given a weight of one, and the edges from variables to equations are given a weight equal to the number of unknown variables in the target equation. Unlike traditional UCS, which operates on a static graph with fixed edge weights and a single goal node, our approach adapts to a dynamic computation frontier that changes as variables become resolved (unknowns become knowns). The goal is not to reach a particular node, but to fully resolve all dependencies needed to compute the target variable. During traversal, we are tracking cumulative costs as we go (the total number of remaining unknowns). For each graph layer, the nodes that correspond to the least number of cumulative unknown variables are explored first, prioritizing nodes with the fewest unknowns (lowest cost) at each decision point. While traversing we backtrack when a path becomes more costly than alternatives. For any equations with exactly 1 unknown variable, text replacement is used to insert the values of known variables into the equation, then the equation is sent to SymPy to solve for the unknown, after which the value of the previously unknown variable is updated in the graph. This process is continued until there is enough information to find the numerical value of the wanted variable or until the graph is exhausted, in either case we produce a minimum cost dependency graph for the target variable.

If the graph is exhausted, the calculator returns a list of possible equations that could be used to find the value and what information is missing (the unknown variables in said equations). If instead a numerical value for the wanted variable is found, we generate a step-by-step explanation for deriving the answer from the given information. As the resultant dependency graph is a directed acyclic graph, we can generate a topological ordering. We use Kahn's algorithm [17] to generate this ordering, resulting in a list of steps that solve for each dependency in order, to find the target variable.

This set of steps can be rather rudimentary, so we provide the ability to provide additional information by feeding the original problem and our set of steps into an LLM with instructions to expand and

improve upon the steps generated with additional contextual information, such as where the equations used came from, but importantly instruct to directly modify the equations, values, or solution that our solving process generated to limit the change of incorrect information being used.

4.4.1 Examples

Figure 9 provides an example of solving for a variable when it is present in an equation with only one unknown. Here, there are related equations $y + a = 5, b + y = z + a, x + a + b = 10$ with given variables $x = 1, y = 2$ and wanted variable a. Our algorithm starts at node a, then sorts the list of equations in ascending order by the number of unknowns. Here, the equations would be sorted as $y + a = 5$ (only a is unknown), $x + a + b = 10$ (a, b unknown), $b + y = z + a$ (a, b, z unknown). Since $y + a = 5$ is the first equation in the list, the calculator first explores the node corresponding to $y + a = 5$. Since $y + a = 5$ also has exactly one unknown variable, a, the calculator solves for the unknown variable in two steps. First, the calculator modifies the equation node's equation-string "y+a=5" by using text replacement to substitute values of known variables; here, "y" is replaced with "2", yielding "2+a=5". Second, "2+a=5" is passed to SymPy (see Section 4.6) to yield $a = 3$. Since the calculator found a numerical value for the wanted variable a, the graph traversal is finished.

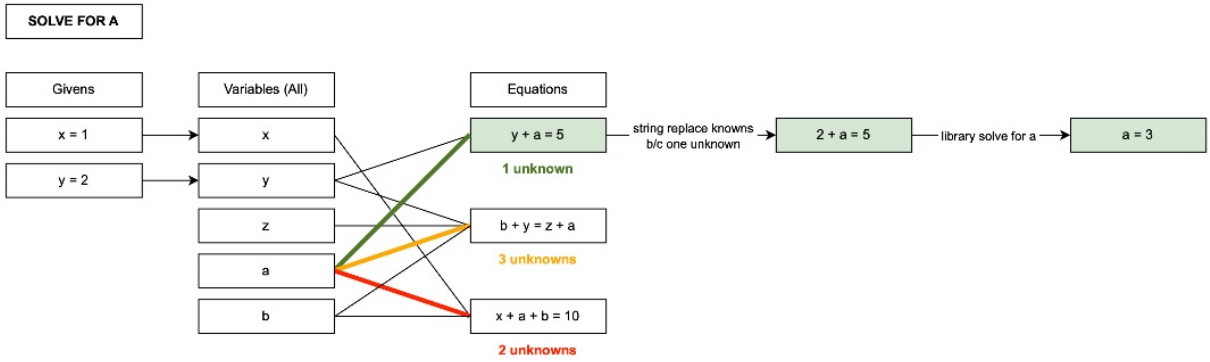

Figure 9: This figure depicts the calculator solving for wanted variable a, when provided equations $y + a = 5, b + y = z + a, x + a + b = 10$ and given variables $x = 1, y = 2$. The calculator prioritizes exploring equation nodes corresponding to equations with the least number of unknowns, so it first explores the node of $y + a = 5$. Since this also happens to have exactly one unknown variable, the calculator solves for the variable's value with text-replacement and SymPy to attain $a = 3$. Since a numerical value for the wanted variable a is found, the calculator stops search through the graph.

For another example, Figure 10 depicts solving for a variable when it is present only in an equation with multiple unknowns. This involves exactly the same equations and given variables as the system depicted in Figure 9, but this time the wanted variable is z. Once again, it starts at the node of the wanted variable (z) and sorts the list of related equations by number of unknowns. Here, the calculator opts for the only

adjacent equation $b+y=z+a$, which contains unknowns a, b, z. In this case, the calculator needs to solve for the other unknowns in the equation, a and b, before attempting to directly solve for z. The list of unknown variables is sorted alphabetically, so the calculator attempts to solve for a first, which involves the steps shown in Figure 9, yielding $a = 3$. Then the calculator attempts to solve for b, which is present in equations $x + a + b = 10$ and $b + y = z + a$. The calculator opts for $x + a + b = 10$, which has b as its only unknown variable. So "x+a+b=10" is changed with text replacement to "1+3+b=10", which is fed into SymPy to attain $b = 6$ (which, again, is the same process as depicted in Figure 9). Note that if the calculator opted to solve for b instead of a first, it would have encountered the equation $x + a + b = 10$ with unknowns a, b, from which the calculator decides to try solving for a before attempting to directly solve for b, leading to the steps depicted in Figure 9. Regardless, the calculator now has enough information ($a = 3$ and $b = 6$) to solve directly for z, since $b + y = z + a$ now has exactly one unknown. The calculator solves for z, then returns $z = 5$. Since the calculator found a numerical value for the wanted variable z, the graph traversal is finished.

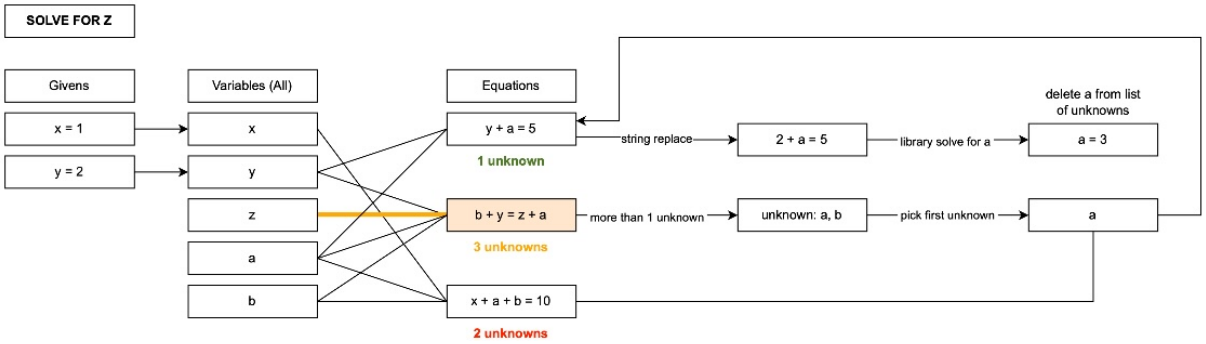

Figure 10: This figure depicts the calculator solving for the wanted variable z. Its only neighboring equation $b+y=z+a$ has unknowns a, b, z. Before attempting to directly solve for z, the calculator attempts to solve for the numerical values of a and b first. The calculator first solves for a (as shown in Figure 9), yielding $a = 3$. The calculator then solves for b, yielding $b = 7$. Then the calculator solves for z, yielding $z = 5$.

While Figure 9 and Figure 10 depict cases where the equations can be solved in sequence, a different method is employed for when multiple equations need to be solved in parallel. For instance, see Figure 11.

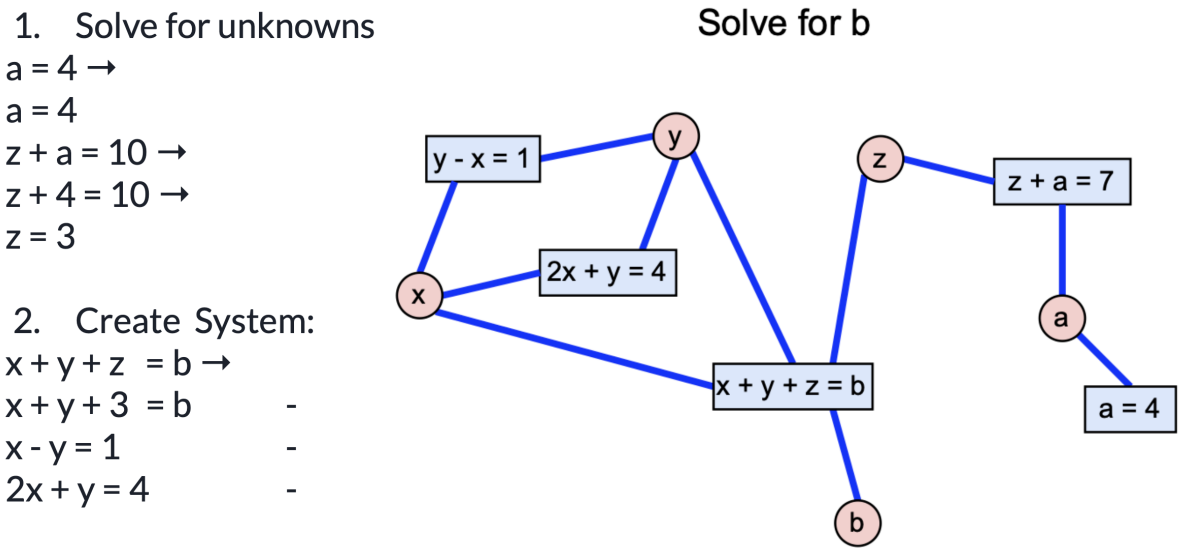

Figure 11: This figure depicts the calculator solving for the wanted variable b. To find b, our algorithm first seeks to find numerical values for the unknown variables x, y, z. While z can be solved by processing equations in sequence (similar to the examples depicted in Figures 9 and 10), solving for the unknowns x, y requires processing multiple equations in parallel—namely, $y - x = 1$ and $2x + y = 4$. The below text describes the process employed for this situation.

Here, the wanted variable is b, and the calculator needs to solve for unknown variables b, x, y. Variable z can be solved as shown in the previous examples, but for variables x, y, both equations $2x+y = 4$ and $y - x = 1$ are needed because x and y are coupled. During graph traversal, all visited equations are stored in a list. Each time an equation is added to the list, the calculator checks whether a solvable system of equations can be built. To do this, it first searches for any subset of equations with n equations and at most n unknowns. If a solvable set of equations is identified—in this case, the set $\{x+y+3 = b, x-y = 1, 2x+y = 4\}$, the following process is used to solve the identified system of equations.

For any equation $f\{x_1, x_2, \ldots, x_n\} = C$, we can isolate any x_k for $k \in \{1, \ldots, n\}$; i.e, $f(\{x_1, x_2, \ldots, x_n\} - \{x_k\}) = x_k$. Then, given any two equations:

$$\text{EQ1: } f\{V\} = C \tag{17}$$

$$\text{EQ2: } f\{U\} = K \tag{18}$$

If $U \subseteq V$ then we can use $f\{U\}$ to solve x_n, then eliminate x_n from $f\{V\}$ by substituting it in terms of all the other variables already included in it. Applying this to the example:

$$\text{EQ1: } x - y = 1 \quad \longrightarrow f(x, y) = A$$

$$\text{EQ2: } 2x + y = 4 \quad \longrightarrow f(x, y) = B$$

1. Isolate x from EQ1

$$x = 1 + y$$

2. Substitute into EQ2

$$2(1 + y) + y = 4 \implies 2 + 3y = 4 \implies 3y = 2 \implies y = \frac{2}{3}$$

3. Back-solve for x

$$x = 1 + \frac{2}{3} = \frac{5}{3}$$

Now using the previous method of direct substitution, we can solve for b

$$x + y + 3 = b \quad \implies \quad \frac{5}{3} + \frac{2}{3} + 3 = b \quad \implies \quad \frac{7}{3} + 3 = b \quad \implies \quad b = \frac{16}{3}$$

$$\boxed{b = \frac{16}{3}}$$

For any given variable present in the set of equations solved with the above algorithm, either its numeric value was found or it wasn't. In the former case, the computed numeric value is stored in the corresponding variable node, and the calculator keeps track of the substitution steps for the calculator's outputted step-by-step explanation. In the latter case, the variables necessary to solve the system are marked (e.g., "pi is needed to solve for radius") and if later in the graph traversal any of the marked variables are updated, the calculator will continue solving for the still-unknown variable.

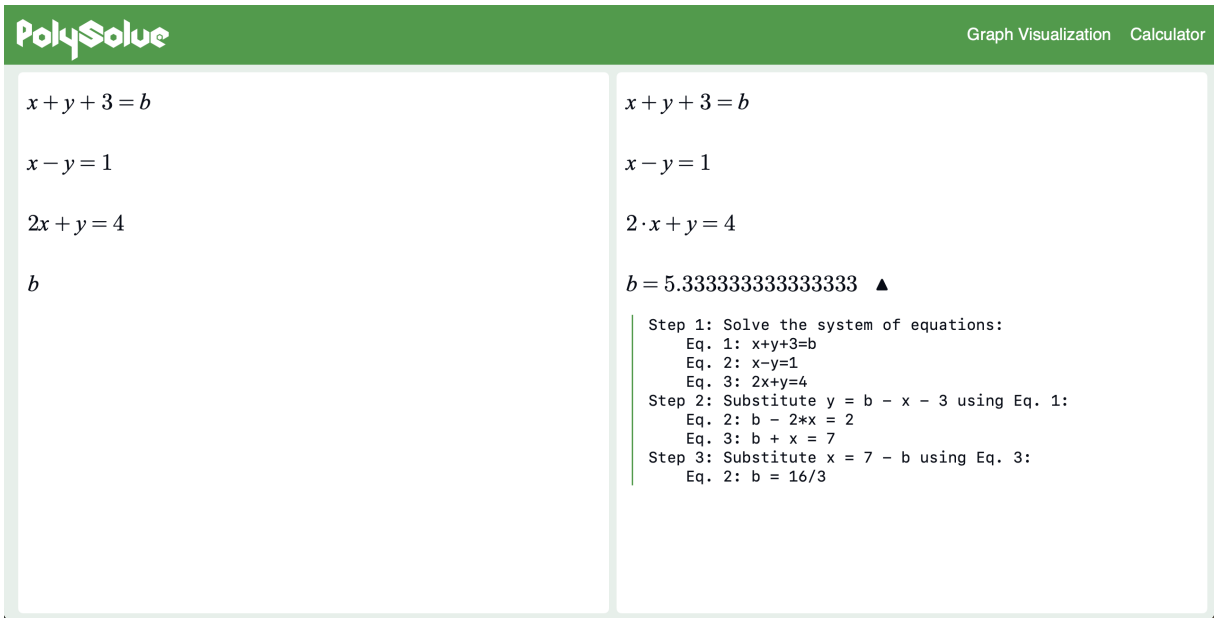

Figure 12: This screenshot showcases an example of the user providing a defined system of equations (left), then receiving an output of the wanted variable(s) along with a set of generated derivation steps (right). The steps can guide the user the process from working with the given variable and equation information to arriving at the values of the wanted variables through an overview of the order in which the equations should be applied.

4.5 Software Development

The calculator website is a web application with three main components: backend, frontend, and database. The backend component handles the computational logic directly involved with the calculator, including the graph traversal approach (see Section 4.4) and any implemented Machine Learning models (see Sections 4.2 and 4.3). The strong and statically-typed programming language Rust [35] was used to ensure a stable development process, along with the Actix Web software package [42] to manage the communication between the backend and the frontend and database. Python [33] was also utilized to implement the Machine Learning models, in conjunction with the symbolic math package SymPy [29] for equation parsing (see Section 4.4). The backend also interfaces with a Huggingface [9] inference endpoint, which hosts and runs our finetuned Llama model (see Section 4.2.2). The frontend component is responsible for the user-facing presentation and interaction of the web application, allowing users to use the calculator, graph, and Machine Learning models. It was implemented using the Svelte [38] framework and Typescript language [40].

Lastly, the database component, implemented with PostgreSQL [10], facilitates the storage and retrieval of data required by the application.

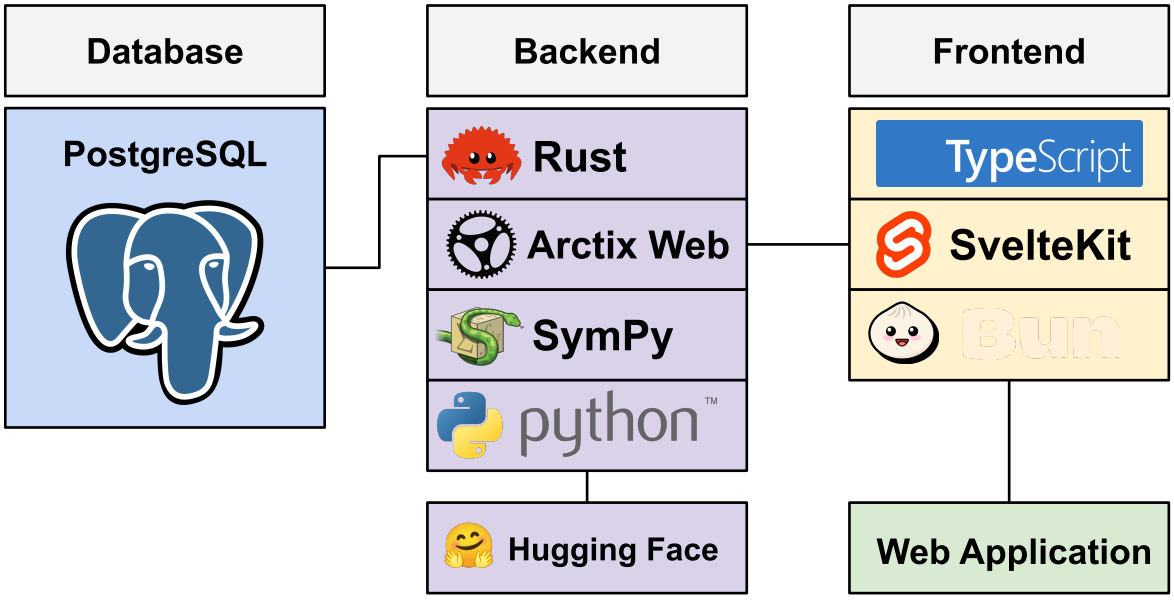

Figure 13: A visual overview of the software tools used. The user interacts with the calculator website (frontend), which relays the provided variable and equation information to the backend, which handles the graph-based computation. Finally, the backend utilizes the database to store graph-based information for continued use.

To maintain the pace and monitor the direction of the development of our project, we adopted an

Agile [6] approach to project management. In a nutshell, Agile involves many short-term planning sessions throughout the project rather than planning everything before the project begins. This allows developers to adapt to any unforeseen developments, which can range anywhere from technical snags to shifts in project priorities.

4.6 SymPy

SymPy [29] was used to handle the parsing of equations and solving of the resulting system of equations. This allows the calculator to return symbolic expressions, which is especially relevant when there is not enough information to find the numerical (scalar) value of a variable, and thus when the variable can only be expressed as a non-scalar symbolic expression. Specifically, the graph-based calculator relies on SymPy's `solve` function to isolate a given variable; for example, it handles the logic for solving y in $x + \sin(y) = z$ and returns $y = \arcsin(z - x)$. As for why the solving approach relies on this function, see Section 4.4.

Another function of SymPy, `sympy.simplify`(equation) [31], is implemented to simplify equations as a preprocessing step to prevent unnecessary updates to the underlying graph representation of the system of equations. Mechanistically, each line is processed, whereby equation and variable nodes are generated if they don't already exist, or have their values updated otherwise. The mentioned edge case can occur, for instance, when equations "x+y-z=-z" and "x+y=0" are entered into the calculator. A naive implementation may treat these equations as separate, and create two equation nodes and three variable nodes, including a node for "z" which connects to the node of "x+y-z=-z". This is in fact not necessary, as (a) the two equations are the same and (b) "x+y-z=-z" provides no information about the value of variable "z". Because of this, the step to simplify the expressions for preprocessing the calculator inputs prevents these kinds of updates that at best provides no additional benefit and at worst complicates the solving procedure. Similarly, additional user inputs that correspond to equations that already exist in the graph will not affect the graph; this ensures that each distinct equation appears in at most one node.

5 Results: Application Walkthrough and Functionality

The culmination of this project is PolySolve, a web-based calculator application designed to streamline the process of solving complex problems involving systems of equations. This section serves as a walkthrough of the application's primary functionalities, demonstrating how each feature contributes to an enhanced user experience, particularly for multi-step, domain-specific calculations.

5.1 Dynamic Equation Input and Real-Time System Updates

The core interaction with PolySolve begins in the main input interface. Users define their problem space by entering equations line by line. For instance, a user might input:

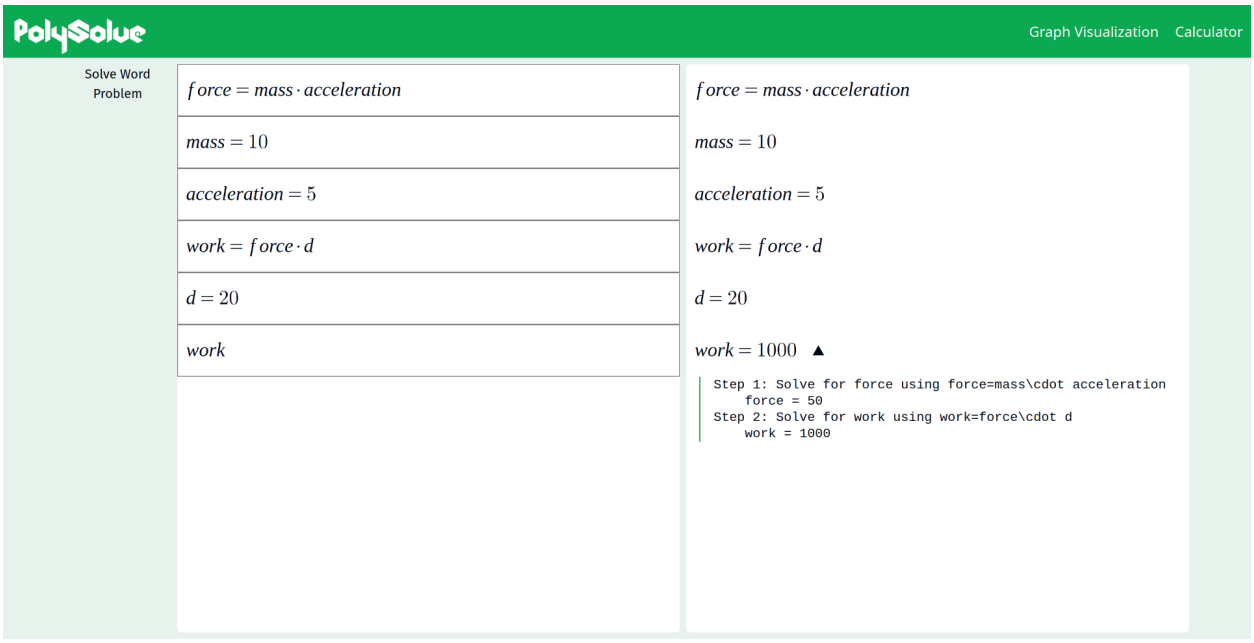

Figure 14: The above screenshot showcases the interface with which the user interacts with the calculator. In each line, the user can assign a value to a variable, relate variables in equations, and request the value of one or more variables. For each line, the calculator outputs a corresponding confirmation or response.

Crucially, as each line is entered, the application's internal representation of the system of equations —modeled as a bipartite graph (detailed in Section 3.2)—updates in real-time. This immediate feedback allows users to see the structure of their problem evolve as they define it. The calculator simultaneously attempts to solve for any variables that become computable based on the current state of the graph, displaying results dynamically. This interactive approach contrasts sharply with traditional calculators that require the full problem context upfront or manual sequencing of calculations. The benefit is a more intuitive process,

where the relationships between variables and equations are built incrementally and transparently.

5.2 Graph Visualization Interface

To further enhance transparency and user understanding, PolySolve provides a dedicated graph visualization interface. This view renders the underlying bipartite graph structure of the computational engine (Section 4.4). Users can visually inspect the connections between equation nodes and variable nodes, mirroring the conceptual representation shown in Figure 2.

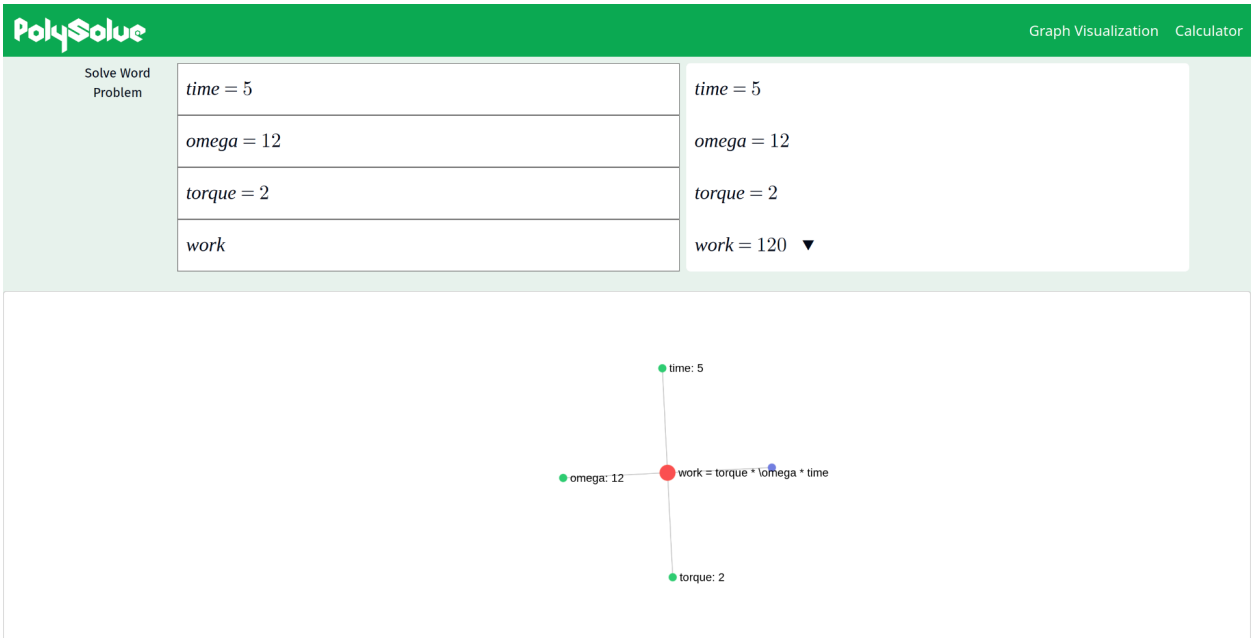

Figure 15: The above figure depicts the calculator displaying a graph visualization for the defined system of equations. This visual insight into the calculator's internal state can be particularly valuable for debugging complex problems and for educational purposes, helping users grasp the structure of equation systems.

5.3 Word Problem Parsing

Recognizing that many problems—especially in domains like physics—are initially expressed in natural language, PolySolve incorporates a word problem parsing feature. Instead of manually translating a problem description into individual equations, users can input the entire word problem directly into a designated area, as shown in Figure 16. For example, entering the problem from Figure 3:

> An engineer is testing a motor. The work done by the motor is **100 Joules**, the torque is **10 Newton-meters**, and the test runs for **10 seconds**. Calculate the **angular velocity** (ω) in radians per second.

The application utilizes the fine-tuned Large Language Model (described in Section 4.2) to parse this text. The LLM extracts the given variables with their values and units (e.g., "work = 100 [Joules]", "torque = 10 [Newton-meters]", "time = 10 [seconds]") and identifies the wanted variable ("omega"). As illustrated in Figure 17, these extracted pieces of information are then automatically formatted and inserted as individual lines into the main equation input area. This significantly streamlines the setup process for algebraic word problems (Section 3.3), reducing manual effort and potential transcription errors, making the tool much more accessible for users who prefer stating problems naturally.

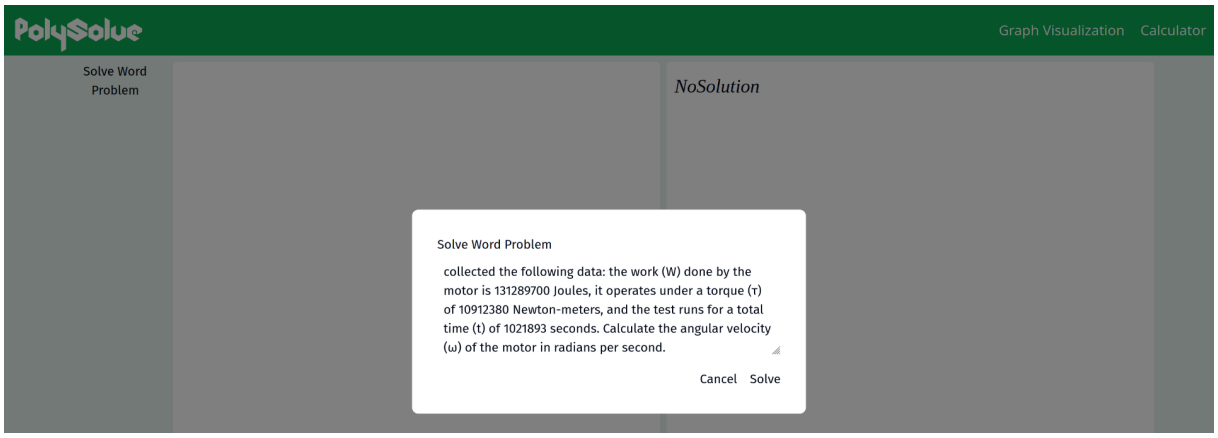

Figure 16: Manual inputs, the calculator also accepts physics word problems (as specified in Section 4.2). A finetuned Large Language Model (LLM) is employed to extract relevant variable information and directly insert them into the calculator. The above is a screenshot displaying the interface in which the user can enter a word problem (for technical details, see Section 4.2). See Figure 17 for the result.

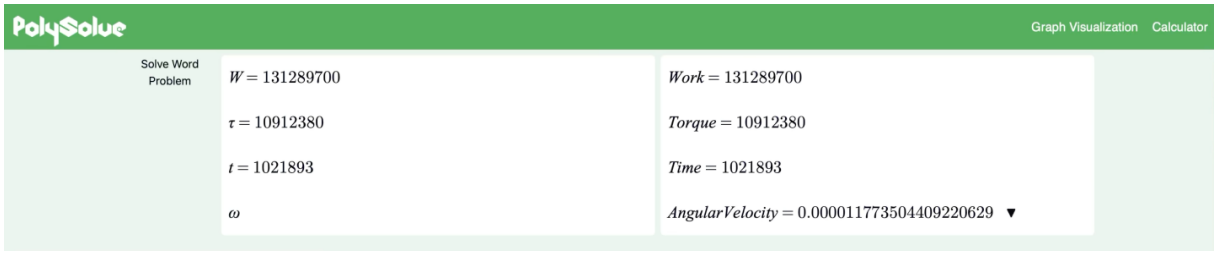

Figure 17: For the word problem entered as shown in Figure 16, the LLM-powered word problem parser extracts relevant variable information, specifically pertaining to given and wanted variables, then enters the extracted information into the calculator.

5.4 Auto-completion and Spell Correction

PolySolve enhances equation input accuracy and efficiency via an integrated auto-completion and spell-correction system (see Section 4.3). When a user enters text, the application presents a dropdown list of suggestions sourced from its domain-specific vocabulary (such as physics terms). Generation of these

suggestions relies on a machine learning model (Section 4.3.3) trained to handle variations such as word completions ("velo" suggesting "velocity," "velocity_initial"), spell corrections ("accelration" suggesting "acceleration"), and abbreviation expansions ("acc" suggesting "acceleration"). The primary benefits of this feature include expedited input, reduced typo frequency, and the promotion of terminological consistency.

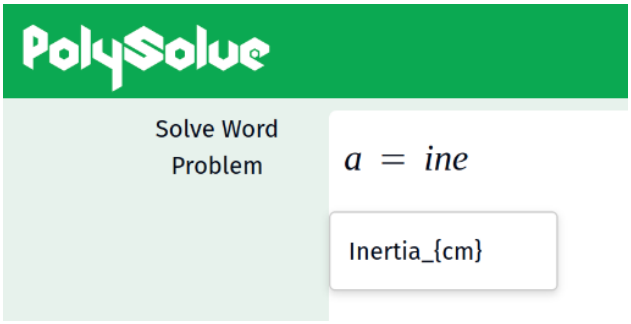

Figure 18: The above figure depicts a dropdown menu presenting one or more options to replace the word currently typed by the user. Each of these options could be a spell correct, completion, or a combination of the two.

5.5 Natural Language Explanation

While the computational engine guarantees accuracy by generating a deterministic, step-by-step solution path (Section 4.4), purely algorithmic outputs can sometimes lack the contextual richness that aids user comprehension. To bridge this gap and enhance usability, PolySolve offers an optional Natural Language Explanation feature.

As illustrated in Figure 19, users can choose to convert the engine's procedurally generated solution steps to a more descriptive explanation. This is achieved by feeding the verified, raw step-by-step output from the computational engine into a fine-tuned LLM (see Section 3.6). Crucially, the LLM's role is strictly limited to translation and contextualization; it does not perform any calculations or alter the logical sequence determined by the engine, thereby preserving the computational integrity of the solution.

The LLM enriches the explanation by incorporating domain-specific context, such as clarifying the physical meaning of variables involved (e.g., identifying 'ω' as angular velocity) and referencing the names or origins of the equations selected by the engine during the solving process. The resulting natural language explanation, presented directly to the user, aims to make the derivation clearer and more intuitive than the raw steps alone, aligning with PolySolve's goal of creating a more user-friendly and understandable computational tool.

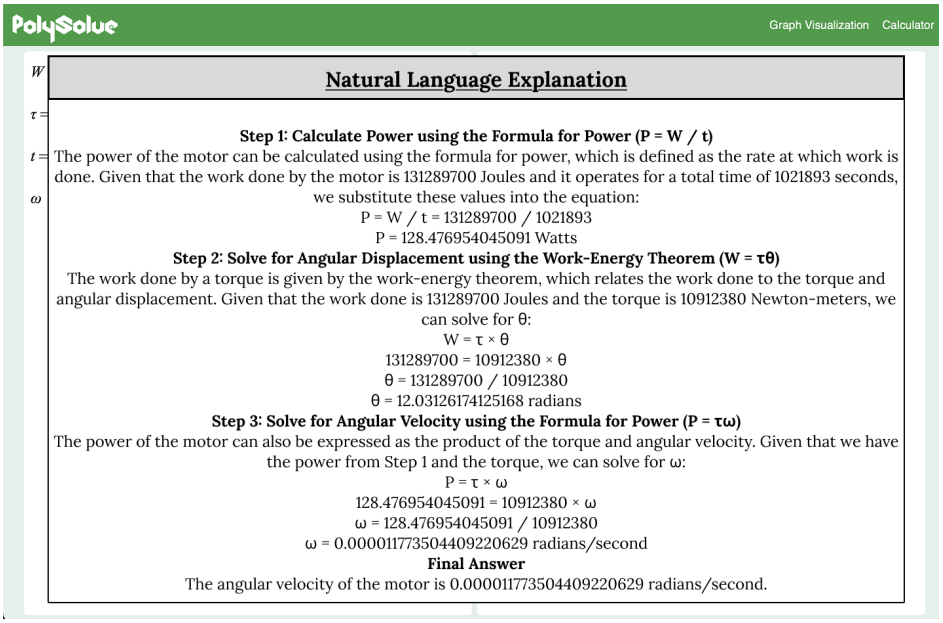

Figure 19: Users can optionally choose to have the calculator output a natural language explanation, which is generated by feeding the raw step-by-step output (see Section 4.4) into an LLM. This provides additional context that the raw output lacks, including the meaning of each variable and the names of the equations selected.

5.6 Integrated Problem-Solving Workflow

In practice, these functionalities combine to create a cohesive workflow. A user might start by pasting a word problem into the parser (Functionality 3). The system extracts givens and wanted variables, populating the input lines. The user can then inspect the automatically generated graph (Functionality 2) to understand the initial state. As they add more equations or modify existing ones (Functionality 1), potentially using auto-complete to ensure accuracy (Functionality 4), the graph and calculated results update instantly. The computational engine continuously works in the background to solve for variables as soon as enough information is available. This integrated approach allows users to fluidly move between defining the problem, observing its structure, and obtaining results, making PolySolve a powerful and user-friendly tool for tackling complex, equation-based problems.

6 Discussion

6.1 Limitations and Future Work

PolySolve requires a predefined database of equations for a specific domain (e.g., introductory physics or engineering). There may also be an instance where a user wants to work across multiple domains at the same time, which can be infeasible due to potential name-space collisions where terminology or symbols used for one domain may have a different meaning for another. However, users can always manually input the necessary equations, providing the flexibility to deal with this on a case-by-case basis. A large language model could potentially automate the expansion of domain-specific equation databases by dynamically identifying and adding relevant equations based on user queries or predefined domain texts. However, this introduces potential LLM-induced inaccuracies and would therefore require additional validation.

The computational engine lacks the ability to model object interactions, such as collisions involving multiple forces. Although it can handle pre-defined collision equations, it does not support the automatic application or combination of these equations when multiple objects or forces interact simultaneously. For example, if three forces act on a body during a collision, the current system cannot dynamically apply the collision equations to calculate the resultant force without explicit manual guidance. Because a dynamic system could manage multiple instances of equations, it would allow complex interactions to be handled. For example, if multiple forces ($Force_1$, $Force_2$, etc.) act simultaneously, the system could duplicate the relevant equations and add them to the graph before calculating the resultant forces. Such dynamic object handling would work for scenarios with multiple interactions, but would require additional work.

Finally, certain variables within the same domain may have multiple valid but context-dependent equations, making it difficult to include all possible equations in a single system. For example, in calculating the components of the gravitational force in an inclined plane starting from the general gravitational force:

$$F_g = mg$$

Depending on the direction of interest, the component of gravitational force acting along the incline is given by Equation (19), and the component acting perpendicular to the incline is given by Equation (20).

$$F_{\text{along}} = mg \sin\theta \tag{19}$$

$$F_{\perp} = mg\cos\theta \tag{20}$$

Given the same set of variables (mass m, gravitational acceleration g, and incline angle θ), the solution varies based on the context. Without explicitly distinguishing these scenarios, assigning both equations to a generic variable (e.g., "Force") would result in ambiguity that could lead to incorrect or contradicting results. Addressing this issue by adding unique variables for each scenario (e.g. F_{along}, $F_{\perp}$) would require extensive manual linking of each new variable to all relevant equations (e.g., $F = ma$), which adds significant effort. An LLM could be utilized to contextually select the correct equation from a predefined list to resolve ambiguities between multiple valid equations. For example, analyzing the context of inputs in natural language, the system could determine when to apply equations such as $F_{\text{along}} = mg\sin\theta$ versus $F_{\perp} = mg\cos\theta$. Although integrating LLM-driven context recognition presents a further risk of LLM errors, constraining the LLM to validated equation lists could limit this risk, but there will always be a potential for error in any system heavily reliant on an LLM.

PolySolve focused on ensuring accuracy over the potential of greater usability with further LLM integration; however, further developments could focus on balancing flexibility provided by further LLM integration with algorithmic validation for greater usability while keeping high accuracy.

6.2 Alternative Methods

During the development of PolySolve, a concurrent work emerged that employs a related methodology: the BeyondX paper[19] introduced the Formulate-and-Solve strategy, which uses an LLM to extract equations from a word problem and SymPy to solve them. While PolySolve's design predates the BeyondX publication and was developed independently, the approaches share a core idea of combining LLM-based information extraction with symbolic solving via SymPy.

However, the methods diverge in their roles for the LLM and the equation-solving process. BeyondX utilizes the LLM to directly generate the system of equations from the word problem. In contrast, PolySolve deliberately constrains the LLM's role to extracting only the numerical values of known variables and identifying the target unknown variable. PolySolve's computational engine handles the key action of selecting and sequencing the equations needed for the solution by traversing a predefined, domain-specific graph of equations.

We speculate that PolySolve's approach—minimizing the LLM's involvement in the core compu-

tational logic—may yield higher accuracy for problems within its supported domains, as it relies less on the LLM's potentially fallible reasoning for equation formulation. Conversely, the BeyondX method offers greater potential flexibility; by relying on the LLM to generate equations, it could theoretically attempt to solve a wider range of arbitrary systems not confined to a predefined equation bank. PolySolve's requirement for an existing equation bank limits its flexibility by necessitating manual creation for each domain of interest.

Beyond the use of LLM extraction and SymPy solving, PolySolve incorporates several distinct contributions. A specifically fine-tuned, small LLM parser dedicated solely to variable extraction from word problems. An SVM-based variable matching system to handle input variations such as typos and abbreviations. A custom graph-based computational engine that algorithmically determines the necessary equations and solution steps—a capability that SymPy alone does not provide. And an LLM to improve the generated steps with domain-specific context without altering the core solution. Additionally, PolySolve presents the complete system as an integrated, interactive, notebook-style calculator on the PolySolve website. Importantly, PolySolve's core computational engine is entirely deterministic and independent of these optional machine learning components; users prioritizing guaranteed accuracy can bypass all ML features, manually input variables, and view the algorithmically generated steps directly. To our knowledge, this specific combination of a deterministic graph-based solver, limited-scope fine-tuned LLMs for parsing and explanation, and the integrated web application represents a novel contribution not fully mirrored in existing work like BeyondX.

7 Conclusion

PolySolve introduces a computational tool designed to bridge the gap between traditional calculators and Large Language Models. By integrating deterministic computational methods with the limited, targeted use of smaller-scale LLMs, PolySolve addresses many limitations inherent in existing solutions. It performs well in domain-specific, multistep problems, automatically selecting relevant equations to solve and providing step-by-step solutions.

The deterministic approach using a computational engine utilizing a bipartite graph to dynamically navigate and resolve dependencies among equations and variables ensures accuracy, greatly lowering error potential coupled to purely LLM-driven systems. Selective integration of LLM for parsing and explanatory tasks provides flexibility, while the limited application decreases the risk of LLM-based errors compromising computational accuracy.

Limitations, such as the requirement of predefined domain-specific databases, an inability to model complex interactions, and difficulties in managing context-dependent equations, provide opportunities for future improvements. Using LLMs for managing equations could provide greater flexibility, but brings an increased risk of LLM-based errors, while a system for dynamically working with complex interactions such as multiforce collisions would expand the potential use cases.

PolySolve demonstrates the capability in balancing limited use of LLMs with a computational engine to allow the flexibility provided by an LLM while maintaining accuracy. Its development serves as a proof-of-concept for more flexible, user-centric computational tools.

References

[1] Midhun T. Augustine. A Survey on Universal Approximation Theorems, July 2024. arXiv:2407.12895 [cs]. URL: http://arxiv.org/abs/2407.12895, doi:10.48550/arXiv.2407.12895.

[2] Dave Bergmann. What is instruction tuning? URL: https://www.ibm.com/think/topics/instruction-tuning.

[3] Adolfo Crespo Márquez. The Curse of Dimensionality. In *Digital Maintenance Management: Guiding Digital Transformation in Maintenance*, pages 67–86. Springer International Publishing, Cham, 2022. doi:10.1007/978-3-030-97660-6_7.

[4] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing*, 503:92–108, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0925231222008426, doi:10.1016/j.neucom.2022.06.111.

[5] Philipp Dufter, Martin Schmitt, and Hinrich Schütze. Position information in transformers: An overview. *Computational Linguistics*, 48(3):733–763, 09 2022. URL: https://doi.org/10.1162/coli_a_00445, arXiv:https://direct.mit.edu/coli/article/48/3/733/111478/Position-Information-in-Transformers-An-Overview, doi:10.1162/coli_a_00445.

[6] Tore Dybå, Torgeir Dingsøyr, and Nils Brede Moe. Agile project management. In Günther Ruhe and Claes Wohlin, editors, *Software Project Management in a Changing World*, pages 277–300. Springer, Berlin, Heidelberg, 2014. doi:10.1007/978-3-642-55035-5_11.

[7] Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. https://transformer-circuits.pub/2021/framework/index.html.

[8] Simeon Emanuilov. Vram for llm inference: Gpu memory requirements for serving large language models. *UnfoldAI*, January 2024. Accessed: 2025-03-25. URL: https://unfoldai.xyz.

[9] Hugging Face. Hugging Face – The AI community building the future., April 2025. URL: https://huggingface.co/.

[10] PostgreSQL Global Development Group. Postgresql, March 2025. URL: https://www.postgresql.org/.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Deep Residual Learning for Image Recognition*, pages 770–778, 2016. URL: https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html.

[12] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models, October 2021. arXiv:2106.09685 [cs]. URL: http://arxiv.org/abs/2106.09685, doi:10.48550/arXiv.2106.09685.

[13] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Trans. Inf. Syst.*, 43(2), January 2025. doi:10.1145/3703155.

[14] HuggingFace. Generation strategies. URL: https://huggingface.co/docs/transformers/en/generation_strategies.

[15] IBM. What is a transformer model?, oct 2023. URL: https://www.ibm.com/think/topics/transformer-model.

[16] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, and Jonathan E. Taylor. *An introduction to statistical learning: with applications in Python*. Springer texts in statistics. Springer, Cham, 2023.

[17] Arthur B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5:558–562, 1962. URL: https://api.semanticscholar.org/CorpusID:16728233.

[18] IOANNIS KANARIS, KONSTANTINOS KANARIS, IOANNIS HOUVARDAS, and EFSTATHIOS STAMATATOS. Words versus character n-grams for anti-spam filtering. *International Journal on Artificial Intelligence Tools*, 16(06):1047–1067, 2007. arXiv:https://doi.org/10.1142/S0218213007003692, doi:10.1142/S0218213007003692.

[19] Kuei-Chun Kao, Ruochen Wang, and Cho-Jui Hsieh. Solving for X and Beyond: Can Large Language Models Solve Complex Math Problems with More-Than-Two Unknowns?, July 2024. arXiv:2407.05134 [cs]. URL: http://arxiv.org/abs/2407.05134, doi:10.48550/arXiv.2407.05134.

[20] Ajitesh Kumar. Support vector machine (svm) python example, March 2023. URL: https://vitalflux.com/classification-model-svm-classifier-python-example/.

[21] Scikit Learn. Countvectorizer. URL: https://scikit-learn/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html.

[22] Scikit Learn. Truncatedsvd. URL: https://scikit-learn/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html.

[23] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015. doi:10.1038/nature14539.

[24] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *AI Open*, 3:111–132, 2022. URL: https://www.sciencedirect.com/science/article/pii/S2666651022000146, doi:10.1016/j.aiopen.2022.10.001.

[25] Wolfram Alpha LLC. Wolfram—alpha, 2025. URL: http://www.wolframalpha.com.

[26] Mathway. Mathway algebra, 2025. URL: https://www.mathway.com/Algebra.

[27] Mathworks. Matlab, 2025. URL: https://www.mathworks.com/products/matlab.html.

[28] Meta. Llama 3.2-1b, 2024. Model released by Meta on September 25, 2024. Available on Hugging Face. URL: https://huggingface.co/meta-llama/Llama-3.2-1B.

[29] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: Symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017. doi:10.7717/peerj-cs.103.

[30] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. URL: https://arxiv.org/abs/1301.3781, arXiv:1301.3781.

[31] Michael Monagan and Roman Pearce. Rational simplification modulo a polynomial ideal. In *Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, pages 239–245, Genoa Italy, July 2006. ACM. URL: https://dl.acm.org/doi/10.1145/1145768.1145809, doi:10.1145/1145768.1145809.

[32] OpenAI. Chatgpt, 2025. URL: https://chatgpt.com.

[33] Python. Python, March 2025. URL: https://www.python.org/.

[34] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[35] Rust. Rust programming language. URL: https://www.rust-lang.org/.

[36] Clayton Sanford, Daniel Hsu, and Matus Telgarsky. Transformers, parallel computation, and logarithmic depth, 2024. URL: https://arxiv.org/abs/2402.09268, arXiv:2402.09268.

[37] Mahmoud Sayed, Asaad Ahmed, Khaled Fathy, and K. Raslan. A deep learning content-based image retrieval approach using cloud computing. *Indonesian Journal of Electrical Engineering and Computer Science*, 29:1577–1589, 03 2023. doi:10.11591/ijeecs.v29.i3.pp1577-1589.

[38] Svelte. Svelte - web development for the rest of us. URL: https://svelte.dev/.

[39] Symbolab. Symbolab, 2025. URL: https://www.symbolab.com.

[40] Typescript. Javascript with syntax for types. URL: https://www.typescriptlang.org/.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL: https://arxiv.org/abs/1706.03762, arXiv:1706.03762.

[42] Actix Web. Actix web. URL: https://actix.rs/.

Appendices

A Existing Options

The provided screenshots below demonstrate the limitations of currently available on-the-shelf calculator options. Generally, existing options fall under one of two pitfalls: (1) their corresponding features explicitly marketed to handle natural language queries can be hard to work with, as they frequently misunderstand queries; (2) even when they are able to solve the provided system of equations and variables, they do not provide a step-by-step explanation about how the answers were derived, which hinders the ability for users to interpret and verify the provided solutions.

Figure 20: As shown above, WolframAlpha's natural language calculator can be hard to work with, for the purposes of solving equations step-by-step. As shown in figure 21, the query has to be significantly reduced before the Wolfram can provide meaningful responses.

Figure 21: When WolframAlpha is able to solve a system of equations and is able to provide the step-by-step algebraic manipulation to derive the answer, it does not provide domain-specific explanations in natural language. This leaves out crucial context, leaving more work for the user to interpret and verify the provided results.

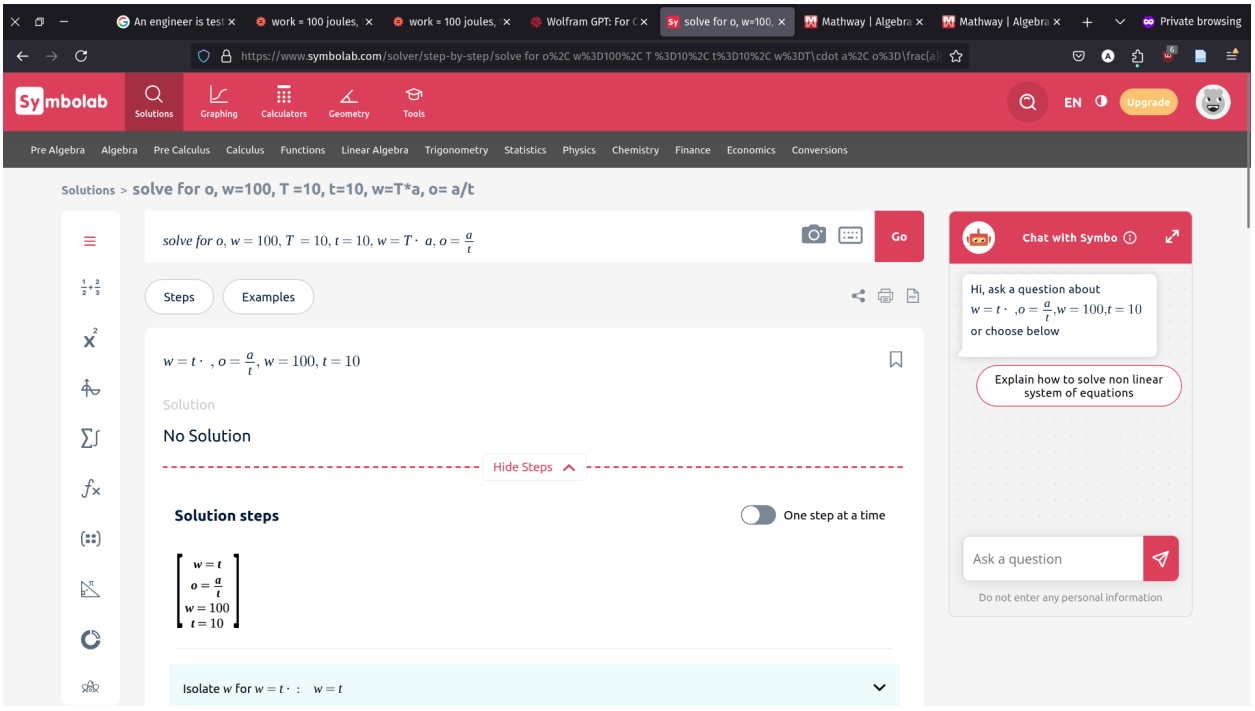

Figure 22: When the Symbolab calculator can solve the system of equations, Symbolab does not provide the step-by-step derivation process taken to arrive at the solution, which hinders interpretability. Again, more burden is placed on the user to handle the interpretation of the problem, which can easily grow infeasible as the scale of the system increases.

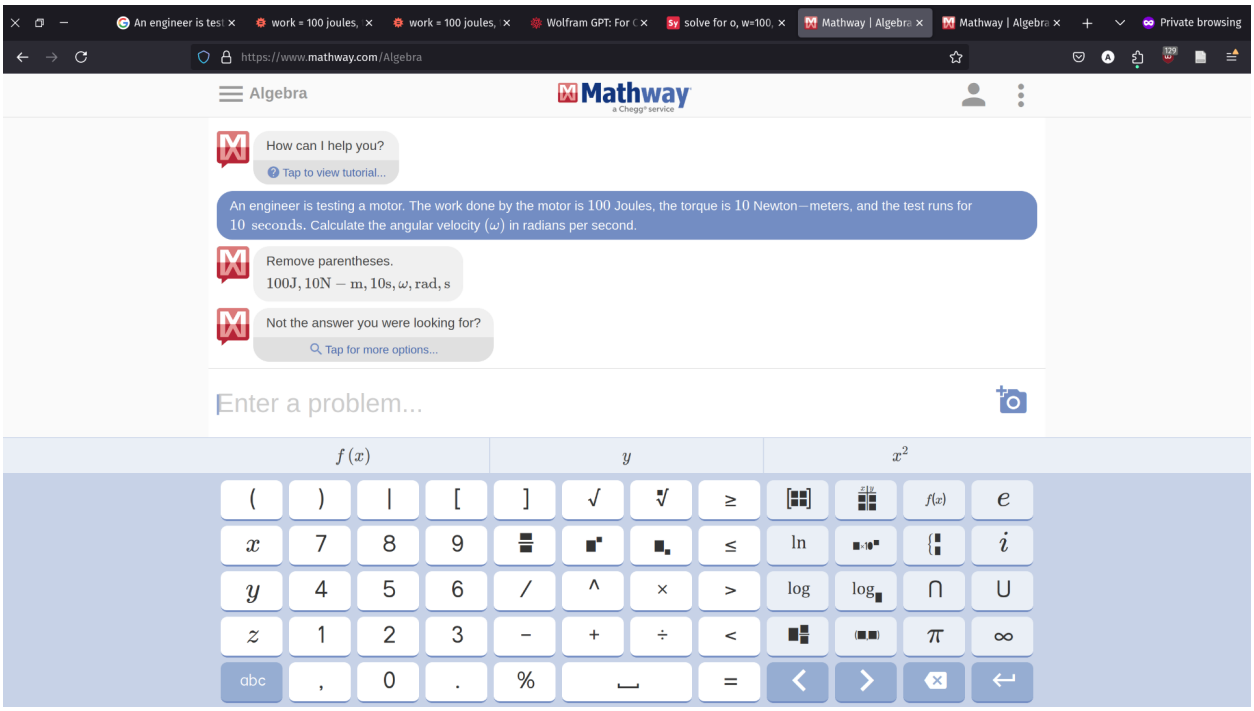

Figure 23: As shown above, WolframAlpha's natural language calculator can be hard to work with, as it easily misinterprets the query. As shown in figure 24, the user is required to significantly reduce the input before the calculator can provide any meaningful responses.

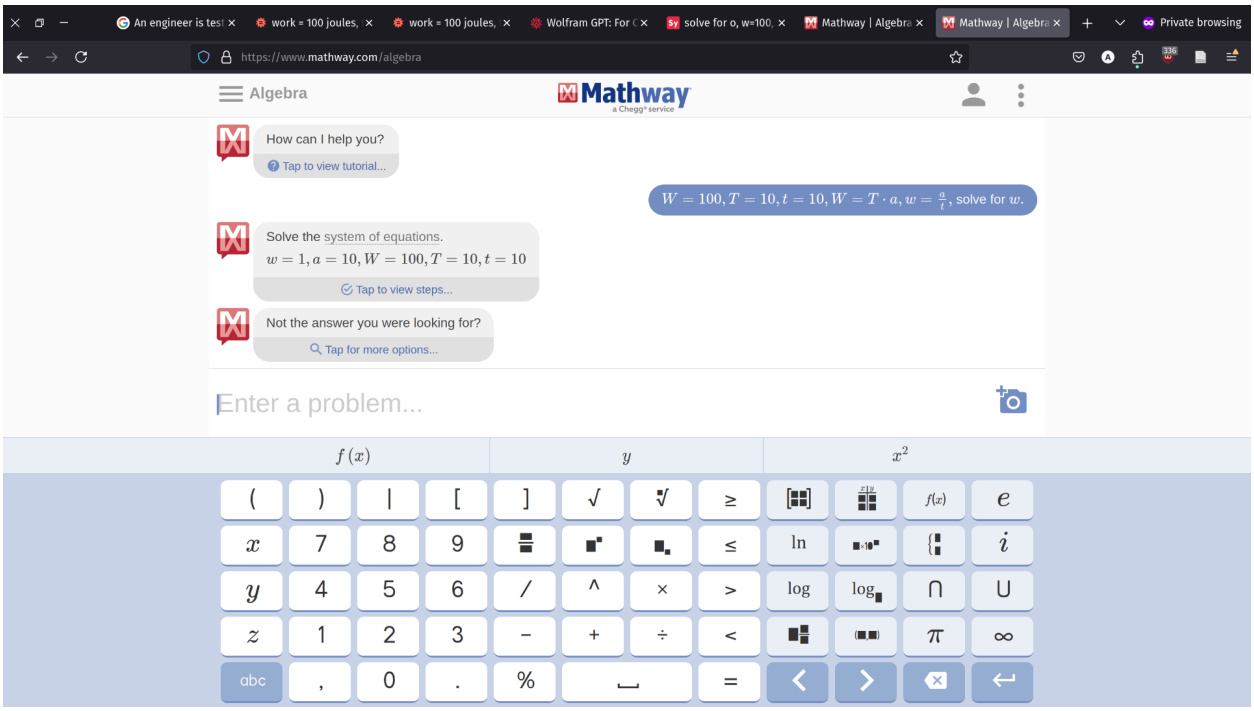

Figure 24: Following Mathway's strict input format—single-letter variables names with one-line input—it can solve the system of equations. However, Mathway has no understanding of the underlying topics involved, only recognizing the symbols as arbitrary variables, thus can only provide the basic algebraic steps without any contextual explanation.