1 Introduction:

1.1 Motivation:

As there is more data being generated than ever before and new experiments, we need a systematic and automatic way to deduce various mathematical patterns and laws in these data. Through the use of symbolic regression we can utilise these data, and in an explainable manner deduce various new physical laws. In this research I have also extended this beyond physics and have applied this to biological data sets which is a novel application of this method. Perhaps extend this beyond or add a sectionsaying this can also be applied to nlp and that it can learn the rules in language and writing etc. Talk a little about the way this is used outside of this niche use case, and in research, so of course I need to look and research into this.

2 Previos Work:

2.1 Literature Review:

2.2 Introduction:

Humanity has spent millennia observing the world, creating concepts that describe the variables, such as mass and force, to derive laws. In physics, like with all human endeavours, new discoveries and ways of thought are based upon previous works, creating a natural bias in the way new problems are approached. All existing theories, are therefore somewhat biased, this combined with our pre-existing bias in our biological brains, can introduce some hurdles to our future progress [**?**] [**?**].

In the 17th Century, Kepler had gotten his hands on the word's most precise data tables on the orbits on planets, using this he spent close to half a decade, and after numerous unsuccessful attempts, he had began a scientific revolution at the time, describing Mar's orbit to be an ellipse [**?**]. In essence, scientists throughout history, much like Kepler, have spent a great deal of time, discovering the right expressions to match the relevant data they have, this at it's core is symbolic regression. Now, a few centuries later, even with exponential increases in orders of magnitude in our capability to perform calculations through computers, the process of discovering natural laws and the way to express them, has to some extent resisted automation.

One of the core challenges of physics and artificial intelligence, is finding analytical relations automatically, discovering a symbolic expression that accurately matches the data from an unknown function. This problem, due to it's nature, is NP-hard [**?**] in principle. The vastness of the space of mathematical constants, adds to the difficulty. This literatire review aims to present the recent advances in discovery of emphirical laws through data powered by artificial intelligence. It focuses on methodoogies that dimish human bias through seeking solutions without assumptions. We will explore various techniques employed to achieve these goals, which includes reducing the search space, and analyse the effectivness of these methods.

Figure 1: This is the orbit of Earth and Mars around the Sun.

2.3 Symbolic Regression:

Symbolic regression, is a technique that analyses and searches over the space of traceable mathematical expressions to find the best fit for a data set. By not requiring prior information about the model, it is unbiased. There are a plethora of various strategies that have been implemented in solving for empirical laws [**?**], we will explore some of them below. It is also worth mentioning, that unlike other well-known techniques for regression, (eg:

neural networks), that are essentially black boxes, symbolic regression, aims to extract white-box models and is easy to analyse.

Brute Force:

Symbolic Regression (SR), is interpretable [**?**], unlike Neural Networks (NN), which are often considered more explainable. The difference is interpretability allows us to comprehend how the model works, like observing how gears move in a glass box, while explainable means you get an overview of why a certain output was achieved, even without knowing the full nuances of it's inner workings.

There however, are some challenges associated with SR, in comparison to function fitting (NN). SR, starts with nothing, a blank slate, and it has to learn the entire expression [**?**], unlike function fitting which just tweaks an already existing function. The exponential search space [**?**] , causes it to be extremely computationally expensive to explore all possibilities. This combined with the face that, most optimisation algorithms expect a smooth search space [**?**], however SR lack's smooth interpolation, small changes in the potential solutions (expression), ie: $x3and x3 + 0.1$ can significantly alter the the output. Finally, if the nature of the problem is badly posed [**?**], there might potentially be multiple solutions to the same data. Imagine trying to find a single polynomial equation with only two points of data, the need to balance finding accurate expressions with finding the most simplistic and generalisable fit, is sometimes troublesome.

The brute force approach of simply trying all possible combinations of symbolic expressions within some defined space. The model will subsequently increase the complexity over time, and will stop when either the fitting errors lowers below some defined limit or exceeds the upper limit of runtime. While in theory can solve all of our problems, in practise takes longer than the age of our universe to finish. In essence it's like searching for a singular drop in the ocean. Thankfully, there are some ways of pruning the search space, and drastically reducing the time taken to solve for the most accurate expression.

Partial Derivatives:

Partial derivatives, of some function f, with multiple variables such as x and y, is it's dervative with respect to one of those two variables, while the other variables in the function are kept constant. Formally, given a function with two or more variables, $f(x_1, x_2, \ldots, x_n)$, the partial derivative of f with respect to x_i, where x_i is some value x in $(x_1, x_2, \ldots, x_i, \ldots, x_n)$, gives the rate of change of f with respect to x_i. It is calculated by taking the ith derivative of f with respect to x_i, whilst holding the other variables fixed. [**?**] [**?**]

The partial derivative of a function $f(x, y)$ with respect to x is denoted $\frac{\partial f}{\partial x}$ [**?**] and is defined:

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \left[\frac{f(x+h,y) - f(x,y)}{h} \right]$$

Once you pass in the experimental data, you can pre-process the data, using calculated partial derivatives, for every pair of existing variables. Many physical laws, involve rates of change, and partial derivatives help us represent them. Furthermore it also guides the search process, as the algorithm can use the derivative to accurately represent the underlying laws involved. Through comparing how well the partial derivatives derived through the experimental data compared to the potential expression, the algorithm can assess the accuracy and feasibility of the expressions involved. This strategy can even be extended to prune the search space further, this could be achieved through incorporating knowledge of physics into the constraints for the partial derivatives. These concepts will be illustrated with an example below.

Consider a iron rod, that has been heated up, such that it is hotter on one side than the other. Now it is intuitive to say that closer to the heat source, the temperature will be higher than further along the rod, where it will be colder. We can illustrate this temperature distribution with a function:

$$T(x, y, z)$$

where T is the temperature at a point in the rod, and (x,y,z) are the coordinates along the axis in 3 dimensions. This leads to these 3 partial derivatives:

$$\frac{\partial T}{\partial x},\ \frac{\partial T}{\partial y},\ \frac{\partial T}{\partial z}$$

These partial derivatives, gives us information about the direction and magnitude of heat flow at various points on the rod. The algorithm then searches for an equation T(x,y,z), that sufficiently predicts the observed temperature distribution and it's partial derivatives, deriving laws such as the heat transfer equations, or elasticity relationships.

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

Through using partial derivatives, we have in essence redefined the search criteria for the algorithm, through it's measure of the accuracy in comparison of potential solutions over the invariants represented in the experimental data [**?**] . This also leads to the pleasant finding, that it can additionally capture relationships that represent other identities of the system, beyond invariants and heat transfer equations.

You can subtly guide the type of laws that such an algorithm finds, by selectively picking the variables to input into the algorithm,. For example providing velocities and force to find laws of motion.

Dimensional Analysis:

Dimensional Analysis is a method of solving problems usually in maths and physics, where we analyse the relationships between different physical quantities, by comparing their "units." It is a powerful method of reducing the complexity of systems, enabling engineers and scientists to analyse problems that we can't even pose, much less solve the equations of [**?**].

Using the fact that numerous questions in science can be simplified by requiring the dimensions/units of the right and left hand side of the expression to be equal, we can transform the question into a smaller number of variables, which all have no dimension [**?**]. It has been automated to find the integer powers of expressions and has proven to be useful especially when the power is an irrational number.

Here is a general strategy that showcases how dimensional analysis can be used:

Let's say we have a variable in an equation that can be broken down into it's fundamental units, such as (second, kilograms, ampere ...) to various powers. We can then take this, and represent each of the units as vectors, such that each of the fundamental units, is assigned a dimension, and it's important to note, this then allows us to represent any physical quantity as a product of these units, so let us construct a vector v, with 3 integers, where each corresponding integer represents the power of each of the fundamental units.

Given that we want to derive an expression, such as $y = f(x_1, \ldots, x_n)$ we can then create some matrix M. Each of the columns of the given matrix, is the unit vector v of the corresponding variable x_i. We then need to define another vector to represent the units of y, which will be called z. If we let the solution be some vector s, solving $Ms = z$, this then lets us raise the powers on both sides, to elevate the independent variables, to make this equation dimensionally consistent.

Taking the null space of the matrix M, where $MV = 0$, allows us a basis to create a dimensionless group, allows for a simplification of the problem.

This is also more intuitive to understand physical phenomena, the nature of physics comprehension, making this vital in further understanding derived laws, making the process easier to explain and understand [**?**]. Therefore, this is a crucial tool, for cultivating a deeper understanding of physics effectively [**?**].

Genetic Programming:

Genetic programming (GP), is a special evolutionary algorithmic technique, where the individuals are seen as programs that evolve, starting for a population, is iteratively "evolved," transforming the populations of individual programs, intro other populations. This new generation of programs are created using some genetic operations or survival criteria, mimicking natural evolutionary condition on earth.

A very basic overview, shows that genetic programming algorithms, consists of initializing the population, then evaluation of the said population through some predefined metrics and functions, followed by selection of the fittest programs based on the score given by the metric, and "genetic operation," such as reproduction, mutation and cross-over. The algorithm then iterates these steps thousands of times, through many generations, and finally terminates once the desired result has been achieved.

We can use genetic programming, and tweak the algorithm, and combine it with symbolic regression, to help us derive laws.

Consider modelling the various potential formulas as a tree, which is composed of various functions in the nodes. These functions can vary from arithmetic operations , mathematical functions, or defined unique operators. Then we can program the fitness function [**?**], and use it to measure how well the given potential expression in the population compares with the given databases, and given the nature of genetic programming, the better performing functions are more likely to be passed down into the next generation. Then after many iterations, we can give the solution with the best performance.

There are various ways to implement the fitness function, and for example we can use a criteria like this, along with mean squared error [**?**]:

$$V = 2X + N \cdot ln(M/N)$$

Here M is the mean squared error, and N is the number of data points, X is the number of parameters used on the genetic programming algorithm. The lower the value of V is, the better the model performs. The performance of this stratergy can then be evaluated with various other metrics, to judge how well the algorithm performs.

3 PySR

This section describes the relevent implementations that are completed as of 10 December 2024.

3.1 Momentum Laws:

To generate the dataset, I chose 100 data points, and created two variables mass (M) and acceleration (a), each represented in two dimensions. Then the data points were generated using $numpy.random.randn$ function. The force (F), was then calculated to be the produce of these two data sets. Mass and acceleration were concatenated along the same axis using $numpy.concatenate$, resulting in a combined dataset. This is partially because the model used here, $PySRRegressor$ expects a single array as input, and this helps highlight the relationship

between these variables to the symbolic regression algorithm.

Then model performed symbolic regression, configured with 40 iterations along with a customer loss function, taken to be the squred diffrence between the prediction and the target variable.

$$\boldsymbol{\mathcal{L}(\hat{x}, x) = (\hat{x} - x)^2}$$

The model was trained on this dataset, upon termination, it produced a list of potential candidate formulae, from which I manually identified the correct expression, $F = m\dot{a}$.

Algorithm 1: Symbolic Regression for $F = M \cdot A$

Result: A symbolic representation approximating $F = M \cdot A$
Initialization:
Generate random data for mass (M) and acceleration (A);
Compute target force values: $F = M \cdot A$;
Combine M and A into input matrix X;
while *Symbolic regression process* **do**
    Train the symbolic regression model with the following settings:;
    Binary operators: Multiplication ($*$);
    Unary operators: None;
    Loss function: Mean squared error between predictions and targets;
    Iterations: 40;
    if *Current symbolic representation improves loss* **then**
        Update the symbolic model;
        Save the current best expression;
    else
        Continue exploration of new symbolic expressions;
    end
end

Similarly, the other laws of momentum, were also dervied using this approach.

$$\mathbf{F}\Delta t = \Delta \mathbf{p} = m(\mathbf{v}_f - \mathbf{v}_i) \tag{1}$$

$$m_1\mathbf{v}_{1,i} + m_2\mathbf{v}_{2,i} = m_1\mathbf{v}_{1,f} + m_2\mathbf{v}_{2,f} \tag{2}$$

3.2 Pendulum Laws:

The data is generated using numpy. The simulation involves, Euler's method to solve the pendulum's equation of motion. Through taking small and discrete steps, the method approximates the solution. The equation for a simple pendulum is given by:

$$\alpha = -\frac{g}{L}\sin(\theta) \tag{3}$$

α angular acceleration (rad/s^2)

g acceleration due to gravity (m/s^2)

L length of the pendulum (m)

θ angular displacement (rad)

The Euler technique approximates the changes in angular velocity and displacement over some small step in time, as follows:

$$\omega_{i+1} = \omega_i + \alpha_i \Delta t \tag{4}$$

$$\theta_{i+1} = \theta_i + \omega_{i+1} \Delta t \tag{5}$$

The function iterates through a few hundred time steps, updating the angular velocity and displacement at each time step. To prevent errors accumulating due to numerical drift, which are small errors that accumulate and become significant due to the inherent nature of approximation methods like Eulers. To keep the values coherent, a wrap around operation is used to ensure the angular displacement is within the range of $[\pi, -\pi] radians$.

3.3 Noise:

This section investigates the model's robustness to noise. To simulate varying levels of interference, artificial noise was systematically introduced during the data generation phase, building upon the previously established model framework. Noise was modelled by generating random numbers within a range of progressively increasing magnitude, utilizing Python's random library. The objective was to observe and quantify the degradation in model accuracy as a function of increasing noise levels, as well as explore ways to mitigate it.

3.3.1 How noise affects the model:

To introduce noise into the generated dataset, I imported Python's random library and used the randint function. To systematically vary the level of noise, I created an additional function that incrementally increased the parameters passed to randint, causing each successive dataset to become progressively noisier.

After generating these noisy datasets, the symbolic regression model was applied to each one, and the resulting equations were analyzed. The relationship between noise level and model performance was then visualized through a graph. Additionally, the time library was used to measure how long each run of the model took.

PLOT HERE:

3.3.2 Denoise function:

Following the initial noise analysis, a subsequent experiment incorporated a denoising method (implemented using a Python library) applied to the data prior to processing by the pysr model. The results, as depicted in the accompanying plot, indicate that this denoising approach improved model performance up to a specific noise threshold. However, beyond this critical level, performance degraded comparably for both the denoised and non-denoised datasets, suggesting the denoise function's effectiveness diminished at higher noise intensities.

PLOT HERE:

4 Symbolic Regression from First Principles:

4.1 A Brute-Force Approach:

The core and essential component of any symbolic regression model lies in its ability to generate and traverse the search space of potential equations and expressions that best fit the given data. To streamline the process and validate the functionality of my expression generation, I began by implementing simple two-variable equations, specifying the operations used within the equation. This approach was initially limited to basic operations, with plans to extend it to accommodate constants and additional complexities.

def generate$_e xpressions(variables, constants, operators, max_d epth)$:
symbols = [sp.Symbol(v) for v in variables] + [sp.Integer(c) for c in constants] expressions = []
for a, b in permutations(symbols, 2): for op in operators: try: if op == '+': expressions.append(a + b) elif op == '-': expressions.append(a - b) elif op == '*': expressions.append(a * b) elif op == '/': expressions.append(a / b) elif op == '**': expressions.append(a ** b) except: continue return expressions
Subsequently, I refined this approach by designing a recursive method to generate expressions, allowing for the creation of more robust and diverse equations from the available variables. This process is dynamic, making it adaptable to a variety of input configurations.

def recursive$_e xpressions(operators, variables, constants, max_d epth = 10) : symbols = [sp.Symbol(v) for v in variables]$
def build$_e xpressions(current_d epth) : if current_d epth == 0 : returnsymbols.copy()$
new$_e xpressions = []prev_e xpressions = build_e xpressions(current_d epth - 1)$
for a in prev$_e xpressions : for b in prev_e xpressions : for op in operators : try : if op == ' + ' : new_e xpressions.append(a + b) elif op == ' - ' : new_e xpressions.append(a - b) elif op == ' * ' : new_e xpressions.append(a * b) elif op == ' / ' : new_e xpressions.append(a/b) elif op == ' ** ' : new_e xpressions.append(a * * b) except : continue$
return new$_e xpressions$
all$_e xprs = build_e xpressions(max_d epth) return all_e xprs$

4.1.1 Exploiting Physical Properties:

The next step involves truncating the generated expressions to prune the search tree as efficiently as possible. One effective method for achieving this is by leveraging the symmetrical properties of physical equations and recognizing their mathematical equivalence. This includes removing redundant or duplicate expressions that do not contribute new information.

This is the approach I used to achieve this:

4.1.2 Dealing with constants:

Another approach I employed to further prune the set of generated expressions was by filtering out any expressions that did not contain all the specified variables. This step helps optimize the process by reducing the number of irrelevant expressions, ultimately saving computation time during the evaluation phase.

4.1.3 Dealing with powers:

To apply powers to expressions, I incorporated the power operation directly into the generated expressions. This not only allows for the creation of more complex models but also facilitates further pruning of the search tree by avoiding the generation of redundant expressions that already incorporate powers.

def apply$_p owers(expressions, powers, max_d epth = 2) : def recursive_p owers(expr, depth) : results = set() if depth == 0 : return expr$

for power in powers: powered = expr ** power results.add(powered)
deeper = recursive$_powers(powered, depth - 1)results.update(deeper)$
return results
all$_results = set()forexprinexpressions : all_results.add(expr)all_results.update(recursive_powers(expr, max_depth))$
return list(all$_results)$
I implemented a filtering mechanism based on whether the expression included a power operation. This further reduces the size of the search tree by eliminating unnecessary branches. While a more robust model would derive this power operation from scratch, I opted for this approach to optimize computation time and maintain flexibility.

def filter$_powers(expressions, target_powers) : filtered = []$
for expr in expressions: found = any(sub.is$_Powandsub.args[1]intarget_powersforsubinsp.preorder_traversal(expr))iffound :$
$filtered.append(expr)$
return filtered

4.1.4 Chaining powers and constants:

The next step was to chain together powers and constants, applying both to the generated expressions. While the existing design already supports chaining, it is essential to properly filter the results to ensure the search tree remains as compact as possible.

Although constants can be filtered using the existing method, the power operations are embedded within the constants, which causes the previous power filter to no longer function as expected. Consequently, I redesigned the filtering process to operate recursively, allowing it to handle both powers and constants effectively.

code:

However, this approach sometimes results in expressions that feature chained constants, such as sin(sin()). To refine the model further, I introduced an additional filter to remove expressions with multiple instances of the same constant chained together.

4.1.5 Loading data:

Next, I needed an efficient way to load the data I had created. At this stage, my primary focus was on rapid testing. To facilitate this, I initially generated some dummy data values. Afterward, I decided to store the data as a NumPy array, as this would offer significant speed advantages over using text files. Several factors contribute to this, such as NumPy arrays being stored in memory, the efficiency of the underlying binary data format, and NumPy's use of C, which allows it to vectorize operations, greatly enhancing performance.

def load$_data(x : np.ndarray, y : np.ndarray, var_names) : assertx.shape[1] == len(var_names)returnx, y, [sp.Symbol(v)fo$
As shown, I perform a check to ensure that the number of variables provided matches the shape of the array X, where X represents the input data, and y represents the target data, i.e., the final result. For example, X contains the mass and acceleration values, while y contains the corresponding values of the force f as calculated by the equation f=maf=ma. This serves as a basic validation to confirm that the number of columns in the input data corresponds correctly to the variables provided.

4.1.6 How to mitigate noise in data:

Ways to mitigate the noise and it's affects on the model were explored. Functions such as "denoise," in the symbolic regression library helped to some extent. However after a certain point, such methods do not seem to

offer much assistance.
I also made my own denoise algorithm. I implemented various different denoise algorithms to see what effects they had. Firstly I implemented a simple moving avergae as a way to mitigate the noise in the dataset. reword this -¿ " Simple and fast, smooths data well by averaging neighbors. However, it blurs sharp changes and is sensitive to extreme outlier values, pulling the average significantly and distorting the signal." These were my results, this is the pseudo code, explain the alogrithm

The second denoise algoirthm I implemented is a median filter, and this is what effects it has, and this is how i implemented it. Insert Pseudo code. reword: "Excellent at removing spikes and preserving edges better than averaging. Less affected by outliers. Can sometimes slightly distort the overall shape of the signal, especially with large window sizes."

Finally this is the third algorithm that I had implemented for denoising. Wavelet Denoising, this is the effects, and this is the pesudo code. Reword this -¿ "Transforms data to isolate noise, preserving both smooth and sharp signal features effectively. More complex to understand and requires careful selection of wavelet type and parameters for optimal results, which can be tricky."

4.1.7 Evaluating expressions:

Next, I evaluate the expressions that have been generated. I assign the input variables to the corresponding columns of the data in increasing order. These values are then substituted into the expressions, and the model runs the calculations, producing an array of outputs for each expression. This process essentially evaluates every pruned expression and returns a NumPy array of results based on the input data.

def evaluate$_e xpression(expression, variables, X)$:
symbols = [sp.Symbol(v) for v in variables] func = sp.lambdify(symbols, expression, modules='numpy') try: inputs = [X[:, i] for i in range(X.shape[1])] return func(*inputs) except Exception as e: return np.full(X.shape[0], np.nan)
Following the approach outlined in the paper [insert citation here], I utilized a medium error description length loss function, implementing it as described. The error is calculated using the squared difference to ensure all errors are positive, and a constant of 1 is added to guarantee that all errors are greater than 1 when taking the logarithm.

def mean$_e rror_d escripton_l ength(result, original) : result = np.array(result).flatten()original = np.array(original).flatt$
total$_l og_e rror = 0.0n = len(result)$
for i in range(n): error = abs(original[i] - result[i]) log$_e rror = np.log2(1 + error * *2)total_l og_e rror+ =$
$log_e rrorprint(f"Indexi : true = original[i], pred = result[i], error = error, log_e rror = log_e rror : .4f")$
return (total$_l og_e rror/n)$

4.2 Polynomial Fit Module:

Now that the core of the algorithm is functional—handling constants, powers, variables, generating expressions, and filtering redundancy using physical principles like symmetry—I aimed to extend the program by implementing a polynomial fitting module. The goal of this technique is to efficiently fit data to a polynomial model, as many functions in physics (or parts of them) are well-approximated by low-order polynomials, and polynomial fitting is a computationally inexpensive method for this specific class of functions. The technique generates all possible polynomial terms up to a specified degree (e.g., degree 4) and creates a linear equation for each data point where the unknowns are the polynomial coefficients. The system of equations is solved using standard methods such as least squares, and the Root Mean Squared Error (RMSE) of the fit is calculated. If the RMSE is below a predefined tolerance (denoted as pol), the polynomial is accepted as a solution. This approach serves as a fast base case in the recursive algorithm, quickly solving problems that are simple polynomials,

and it can also handle sub-problems transformed into polynomial form by other modules, such as dimensional analysis or function inversion.

4.2.1 Data Loading:

To begin, I developed the data loading function. The goal was to accept a NumPy array containing the data, along with a list of variables. The function then compares the shape of the data array with the number of variables provided to ensure that the input is consistent and sufficient for further processing.

def load$_data(data_array, variables)$:
num$_cols = data_array.shape[1] num_vars = len(variables)$
if num$_cols != num_vars$: $raiseValueError(f"Datahasnum_datacolscolumnsbutnum_varsvariableswereprovided.")$
return data$_array$

4.2.2 Generating polynomial expressions:

The next step involves generating polynomial expressions. The function returns a list of polynomial expressions based on the input coefficients, variables, and operators, considering a specified maximum degree for the terms.

The function works by first creating symbolic representations for the variables. It then iterates over all possible combinations of powers for the variables up to the specified degree and combines these terms using the provided operators. Finally, the generated expressions are simplified and returned as a list.

def generate$_expressions(coeffs, variables, operators, max_degree)$:
vars$_syms = [sp.Symbol(v)forvinvariables]$
expressions = []
for powers in itertools.product(range(1, max$_degree + 1), repeat = len(vars_syms)) : terms = [c * (v *$
$*p)forc, v, pinzip(coeffs, vars_syms, powers)]foropsinitertools.product(operators, repeat = len(terms)-$
$1) : expr = terms[0]forop, terminzip(ops, terms[1 :]) : ifop ==' +' : expr = expr + termelifop ==$
$-' : expr = expr-termelifop ==' *' : expr = expr*termelifop ==' /' : expr = expr/termexpressions.append(sp.simpli$
return expressions

4.2.3 Filtering the Polynomial expressions:

The filter$_expressionsfunctionprogrammaticallyfilterssymbolicexpressionsbasedonbothstructuralandsemanticconstrain$
$scalesymbolicfilteringtaskswherestrictmathematicalstructuresmustbeenforced.$

The initial version of this function worked for symbolic constants (e.g., sin, cos, etc.) but failed to handle numbers or integer coefficients. This issue was identified during testing, prompting me to rewrite the function so that it could also handle integer coefficients properly.

def filter$_expressions(expressions, required_vars, required_constants, required_power) : result = []$
required$_vars = sp.Symbol(v)forvinrequired_vars$
for expr in expressions: symbols$_ok = required_vars.issubset(expr.free_symbols)$
if not all(any((isinstance(sub, const) if isinstance(const, type) else sub == const) for sub in sp.preorder$_traversal(expr))forconstin$
$continue$
power$_ok = any(isinstance(sub, sp.Pow)andsub.exp == required_powerforsubinsp.preorder_traversal(expr))$
if symbols$_okandpower_ok : result.append(expr)$
return result

4.2.4 Evaluating expressions:

The next step involves fitting the filtered expressions to the dataset. The model fitting function fits polynomial expressions to the input data by determining the optimal set of coefficients that minimize the error between the predicted and actual output values. It evaluates multiple polynomial degrees, up to a specified maximum, and selects the degree that results in the lowest error, thereby ensuring an optimal balance between accuracy and complexity.

I utilize the Root Mean Squared Error (RMSE) to calculate the loss, and the function returns a list of loss values, one for each expression.

def evaluate$_e xpressions(expressions, variables, data, y_t rue) : results = []forexprinexpressions : try : func = sp.lambdify(variables, expr, modules =' numpy')y_p red = np.array([func(*row)forrowindata])rmse = np.sqrt(np.mean((y_p red-y_t rue)**2))results.append((expr, rmse))exceptExceptionase : print(f"Skippingexprduetoerro e")returnresults$

To begin the fitting process, I take an expression, substitute the input variables with the corresponding values from the dataset, and compute the predicted yy-values based on the equation. I then calculate the difference between the predicted values and the true target values (yy), which are the actual outputs. This difference is used to compute the Root Mean Squared Error (RMSE), which quantifies the prediction error for the expression.

4.2.5 Best Polynomial Fit:

After calculating the RMSE values for each expression, I select the expression with the lowest RMSE as the most accurate polynomial fit for the data. This ensures that the chosen model has the best performance in terms of minimizing prediction error.

def bestFit(results): return min(results, key=lambda x: x[1])

4.3 Dimensional Analysis:

Physical equations must be dimensionally consistent, meaning the units on both sides of the equation must match, which severely limits the possible forms of the unknown function. This dimensional constraint provides a strong simplification of the problem, significantly narrowing the scope of valid equations. AI Feynman addresses this by applying dimensional analysis as the first step, simplifying the problem by identifying which combinations of variables are dimensionally consistent. The units of the variables—such as mass, length, and time—are represented as vectors of integer powers, forming a linear system based on the unit vectors of the input and target variables. Solving this system and finding the null space reveals dimensionless combinations of variables, which transforms the problem into one of finding a function of these new dimensionless variables. This process typically reduces the number of independent variables that the algorithm needs to search over, drastically shrinking the combinatorial search space for subsequent steps, such as polynomial fitting, brute force, and neural network-guided searches. As a result, the reduction in variables leads to a significant boost in efficiency, making these searches faster and more likely to succeed.

4.3.1 Handling Units:

The AI Feynman database was accessed, and the units.csv file was downloaded to better understand the units present in the dataset. Upon reviewing the required units, a unit table was created in the form of an array, where each unit corresponds to a unique power of the fundamental SI units. Additionally, the basic SI units were implemented as an array/list to facilitate this mapping.

UNIT$_T ABLE =$

'mass': [1, 0, 0, 0, 0, 0, 0], 'length': [0, 1, 0, 0, 0, 0, 0], 'time': [0, 0, 1, 0, 0, 0, 0], 'temperature': [0, 0, 0, 1, 0, 0, 0], 'current': [0, 0, 0, 0, 1, 0, 0], 'amount': [0, 0, 0, 0, 0, 1, 0], 'luminous$_intensity' : [0, 0, 0, 0, 0, 0, 1]$,
There were also relevent derived units included.

4.3.2 Construct Matrix and Target Vector:

This function constructs the dimensional matrix MM and the target vector bb, which are essential for performing dimensional analysis. It accepts lists of independent and dependent variable names, along with a dictionary that maps each variable name (as the key) to its corresponding unit vector (as the value). The unit vectors for the independent variables are retrieved through dictionary lookup, using lowercase variable names (i.e., var.lower()) to ensure case-insensitivity. These vectors are then efficiently assembled into the columns of matrix MM using numpy.column$_stack, whiletheunitvectorofthedependentvariableformsthetargetvectorbb.Thisapproachensuresusabilityth$
$insensitivity, leveragestheperformancebenefitsofnumpy.column_stack, andincludesexpliciterrorhandlingtopreventissu$

def get$_matrix_target(independent_vars, dependent_var, units_dict) : try : M = np.column_stack([units_dict[var.lower()]forvar$
$raiseValueError(f"Independentvariable'e.args[0]'notfoundinunitdictionary.")$
try: b = np.array(units$_dict[dependent_var.lower()])exceptKeyError : raiseValueError(f"Dependentvariable'dependent_var$
return M, b

4.3.3 Solving Dimension and Basis Units:

The solveDimension function solves the system of equations Mp=bMp=b for the unknown vector pp, where MM is the dimensional matrix and bb is the target vector. The function begins by converting the input matrices MM and bb into symbolic matrices using SymPy's Matrix class. It then attempts to solve for pp using the LU decomposition method (LUsolve), which is efficient for solving linear systems. If this process fails, an error is raised, indicating the issue encountered during the solution attempt. Additionally, the function computes the null space of matrix MM, representing the set of dimensionless combinations of the variables. The function returns two outputs: the solution vector pp and the null space UU, which provides insight into any dimensionless combinations of the input variables. This method ensures robust error handling and leverages symbolic computation for accuracy.

def solveDimension(M, b): M$_sym = sp.Matrix(M)b_sym = sp.Matrix(b)$
try: p = M$_sym.LUsolve(b_sym)exceptExceptionase : raiseValueError(f"FailedtosolveMp = b : e")$
U = M$_sym.nullspace()$
return p, U

4.3.4 Data Transformation Function:

The generate$_dimensionless_data functionisdesignedtotransformadatasetintodimensionlessformbyapplyingthescalingfa$

If the null space UU is provided, the function proceeds to generate new dimensionless variables by computing the product of the input data dataxdatax raised to the powers specified by each vector in UU. These newly generated dimensionless variables are stacked together to form a transformed input dataset data$_xprime.Ifnonullspaceisprovided, theori$

def generate$_dimensionless_data(data_x, data_y, p, U)$:
if not isinstance(p, np.ndarray): p = np.array(p).astype(np.float64).flatten()
p = p.reshape(-1, 1) scaling$_factor = np.prod(np.float_power(data_x, p), axis = 0)$
data$_yprime = data_y/scaling_factor$
dimensionless$_vars = []$
if U: dimensionless$_vars = []foruinU : new_var = np.prod(np.float_power(data_x, u), axis = 0)dimensionless_vars.append(ne$
data$_xprime = np.vstack(dimensionless_vars)else : data_xprime = data_xNotransformationifnodimensionlessgroups$
return data$_xprime, data_yprime$

4.3.5 Symbolic Transformation Generator:

This function generates the symbolic mathematical expressions corresponding to the dimensional analysis transformation. It accepts the original independent variable names (independent$_v ars$), $the exponent vectors for scaling(p), and the dime$

The function first creates symbolic representations of each independent variable using sp.symbols from the SymPy library. Then, using the scaling exponents pp, it constructs the symbolic expression symbolic$_p$ $representing the unit-fixing scaling factor xpxp through sp.Mul, which allows for the multiplication of terms. This expression effectively represe$

Next, the function iterates through each exponent vector uu in the null space UU, building the corresponding symbolic expressions for the dimensionless combinations. Each new variable is constructed by applying the powers from the exponent vector uu to the original input variables, and the resulting expressions are added to a list. This process ensures that both the scaling factors and the dimensionless combinations are represented as symbolic mathematical expressions, which are essential for understanding the relationship between the variables in the dimensional analysis.

def symbolicTransformation(independent$_v ars, p, U) : symbols = [sp.symbols(var) for var in independent_v ars]$
symbolic$_p = sp.Mul(*[symbols[i] * *p[i] for i in range(len(independent_v ars))])$
symbolic$_U = [] for u in U : expr_u = sp.Mul(*[symbols[i] * *u[i] for i in range(len(independent_v ars))]) symbolic_U.append(expr$
return symbolic$_p, symbolic_U$

4.4 Neural Network Fitting:

The next critical component involves using neural networks to predict and compute gradients from the dimensionless data, providing valuable insights for visualization. While neural networks do not directly solve for symbolic expressions, they serve as powerful tools for approximating complex relationships within the data. By training a neural network on the dimensionless variables, it can predict the output for a given input and calculate gradients, offering a smooth, differentiable function that helps visualize how changes in the input variables influence the model's predictions. This capability allows us to gain a deeper understanding of the underlying behavior of the system. Although neural networks do not offer an explicit symbolic expression, they provide a flexible and efficient way to visualize the functional dependencies, aiding in the interpretation of complex patterns that may be difficult to express symbolically. Ultimately, the network's predictions and gradients can be used to explore and understand the data, even if the true functional form remains implicit.

4.4.1 SymbolicNetwork:

This class defines the neural network architecture used as a universal function approximator within the symbolic regression framework. It inherits from torch.nn.Module, the base class for all neural network modules in PyTorch. The constructor ($_{init}$) $initializes the network structure, accepting the number of input features (n_input) and defaulting to a single output (n_output). The mode$
class SymbolicNetwork(nn.Module): def $_{init}(self, n_input, n_output=1): super(SymbolicNetwork, self).__init__() self.model = nn.Sequential(nn.Linear(n_input,$

def forward(self, x): return self.model(x)

4.4.2 Preparing the data:

This function preprocesses raw input (data$_x$) $and output (data_y) data into a format suitable for PyTorch model training and valida$
$validation split is crucial for monitoring generalization and model performance.$

def prepare$_d ata(data_x, data_y, batch_s ize, train_s plit = 0.8) : x_t ensor = torch.tensor(data_x, dtype =$
$torch.float32) y_t ensor = torch.tensor(data_y, dtype = torch.float32).unsqueeze(1) Ensure y is (N, 1)$
num$_s amples = x_t ensor.size(0) split_i dx = int(num_s amples * train_s plit)$

$x_train, x_val = x_tensor[: split_idx], x_tensor[split_idx :] y_train, y_val = y_tensor[: split_idx], y_tensor[split_idx :$
]
$train_dataset = TensorDataset(x_train, y_train) val_dataset = TensorDataset(x_val, y_val)$
$train_loader = DataLoader(train_dataset, batch_size = batch_size, shuffle = True) val_loader = DataLoader(val_dataset, bat$
$batch_size, shuffle = False)$
return $train_loader, val_loader$

4.4.3 Training the Network:

TThis function orchestrates the supervised training process for the provided PyTorch neural network model, with the primary goal of adjusting the model's parameters (weights and biases) to minimize the difference between its predictions and the true target values using the training data, while also monitoring performance on unseen validation data. The function starts by transferring the model to the specified compute device (either 'cpu' or 'cuda') and initializes the Adam optimizer, a commonly used adaptive learning rate optimization algorithm. The optimizer is linked to the model's parameters, with the learning rate set as a hyperparameter. The Mean Squared Error (MSE) loss function is used, which is well-suited for regression tasks as it minimizes the squared error between predicted and actual values. The training loop iterates through multiple epochs, where the model is trained using batches from the training data, and the optimizer updates the model's parameters based on the gradients computed from the loss function. During the validation phase, the model's performance is evaluated without gradient calculation to save computational resources. This standard PyTorch training loop, which uses DataLoader for efficient batch processing, ensures the model is correctly trained and evaluated on both training and validation datasets, with printed outputs for tracking progress.

def $train_network(model, train_loader, val_loader, epochs, learning_rate, device) : model = model.to(device)$
optimizer = optim.Adam(model.parameters(), lr=$learning_rate) loss_fn = nn.MSELoss()$
for epoch in range(epochs): model.train() $train_loss = 0.0$
for $x_batch, y_batch in train_loader : x_batch, y_batch = x_batch.to(device), y_batch.to(device)$
predictions = model($x_batch) loss = loss_fn(predictions, y_batch)$
optimizer.zero$_grad() loss.backward() optimizer.step()$
$train_loss += loss.item() * x_batch.size(0)$
$train_loss /= len(train_loader.dataset)$
model.eval() $val_loss = 0.0 with torch.no_grad() : for x_val, y_val in val_loader : x_val, y_val = x_val.to(device), y_val.to(device) predi$
$model(x_val) loss = loss_fn(predictions, y_val) val_loss += loss.item() * x_val.size(0)$
$val_loss /= len(val_loader.dataset)$
print(f"Epoch epoch+1/epochs — Train Loss: $train_loss : .6f | Val Loss : val_loss : .6f$")

4.4.4 Predict Function:

This function performs inference using a trained PyTorch model, generating output predictions for a given set of input data. It is designed to take a trained model, input data as a NumPy array ($x_numpy), and the target computation device ('cpu' or' torch.float32 and transferred to the specified device. The core prediction step occurs within a with torch.no_grad() context mana friendly output format that is compatible with typical post − processing and analysis workflows.$

def predict(model, $x_numpy, device) : model.eval() x_tensor = torch.tensor(x_numpy, dtype = torch.float32).to(device)$
with torch.no$_grad() : predictions = model(x_tensor)$
return predictions.cpu().numpy()

4.5 Pareto Frontier Optimisation:

The Pareto frontier provides a valuable method for balancing the trade-off between accuracy and simplicity (complexity) in symbolic regression. Rather than searching for a single "best" formula, the objective is to identify all Pareto-optimal formulas, which are those for which no other formula is both simpler and more

accurate. To implement this, a 2D plot is created where the x-axis represents the formula's complexity and the y-axis represents the mean error description length. As candidate formulas are generated through methods such as brute-force search, transformations, or recursive combinations, each candidate is evaluated and plotted as a point on the frontier. A critical aspect of this approach is Pareto pruning: any candidate formula that is dominated – meaning that another formula on the frontier has both equal or lower complexity and equal or lower loss (with at least one strict inequality) – is discarded. This pruning process reduces the search space and improves computational efficiency by eliminating redundant solutions, especially when combining results from subproblems. Additionally, it enhances robustness against noise by inherently favoring simpler formulas that achieve a given accuracy. The final Pareto frontier provides a spectrum of optimal solutions, offering the user a range of trade-offs between complexity and accuracy.

4.5.1 Points:

The Point class is a fundamental data structure within the Pareto frontier analysis framework, encapsulating the key characteristics of a single candidate formula evaluated during the symbolic regression process. Its constructor initializes each instance with three attributes: complexity, a numerical score representing the formula's complexity; accuracy, a measure of the formula's performance; and the formula itself, which is the associated symbolic expression. This design consolidates the essential metrics (complexity, accuracy) needed for Pareto dominance checks into a single object, simplifying the management and comparison of candidate solutions. Functions operating on the Pareto frontier can efficiently access the complexity and accuracy attributes to determine dominance relationships, facilitating the maintenance of a non-dominated set of optimal formulas.

class Point: def $_{init}{}_{(self,complexity,accuracy,expression):self.complexity=complexityself.accuracy=accuracyself.expression=expression}$
def $_{repr}{}_{(self):returnf"Point(complexity=self.complexity,accuracy=self.accuracy,expression=self.expression)"}$

4.5.2 Pareto Point Set:

The update$_{pareto_points}$$function maintains a Pareto frontier of symbolic expressions, balancing model complexity and accuracy dominated points, where a point A dominates point B if A is both no more complex and no less accurate, with at least one of these of fs between complexity and accuracy remain. In symbolic regression, this is crucial as the goal is to identify expressions that a$

def update$_{pareto_points}(current_points, new_formulas, losses) : new_points = []forexpr, lossinzip(new_formulas, losses) :$
$complexity = calculate_complexity(expr)new_points.append((complexity, loss, expr))$
combined = current$_{points} + new_points$
pareto = [] for pt in combined: if not any(dominates(other, pt) for other in combined if other != pt): pareto.append(pt)
seen = set() unique$_{pareto} = []forptinpareto : key = (pt[0], pt[1], str(pt[2]))ifkeynotinseen : unique_pareto.append(pt)seen.a$
return unique$_{pareto}$

4.6 Plotting:

The need arose to find ways to visualize and better understand the processes occurring behind the scenes in the symbolic regression program. Humans are inherently visual creatures, and visualizations provide a more powerful means of comprehending data compared to simply examining strings of numbers or expressions, which makes it difficult to identify patterns or errors manually. Plotly was chosen over the more commonly used Matplotlib for its superior aesthetic appeal and its enhanced interactive environment, offering a more intuitive and engaging experience for exploring the data and model results.

4.6.1 Plot for Pareto front:

Visualizing the Pareto frontier is crucial for interpreting the results of symbolic regression as it offers an intuitive graphical representation of the trade-off between model complexity and predictive accuracy (or loss). The provided code leverages the Plotly Express library to generate this visualization. A Pandas DataFrame (df) is used to organize the data, containing columns for 'Complexity' (x-axis), 'Loss' (y-axis, representing inaccuracies such as MEDL), and the corresponding symbolic 'Expression'. The px.scatter function generates a scatter plot that maps complexity against loss, with the 'Expression' data being displayed as labels on each point using the text argument. Additional customization via fig.update$_t racesenhancesthevisibilityofthepoints(markersize, color)andoptimizesla$
$dominatedsolutions.Byplottingquantitativemetrics, itrevealswheresignificantaccuracyimprovementsrequiresubstanti$

def plot$_p areto_f rontier_p lotly(pareto_p oints, title = "ModernParetoFrontier") : df = pd.DataFrame('Complexity' : [pt[0],$
fig = px.scatter(df, x='Complexity', y='Loss', text='Expression', title=title) fig.update$_t races(marker = dict(size =$
$10, color =' red'), textposition =' topcenter') fig.update_l ayout(plot_b gcolor =' lightgrey', paper_b gcolor ='$
$white', font = dict(family = "Arial", size = 14), title_x = 0.5)$
return fig

4.7 Main method bringing it together Regressor:

4.7.1 main solver

Part 1: Dimensional Analysis and Symbolic Scaling
The function first applies Dimensional Analysis (DA) to reduce the problem's dimensionality. It computes a transformation matrix and scaling vector by solving the unit balance equation between independent and target variables. If no dimensionless groups are found, the result is a simple scaling law expressed symbolically. Otherwise, it computes and displays the dimensionless groups and the scaling part of the solution. This ensures the model operates on physically meaningful, unit-consistent quantities, simplifying the functional search space and preventing non-physical relationships. If DA completely solves the problem, the function terminates here by outputting the symbolic solution. Part 2: Neural Network Approximation of Dimensionless Relation
When dimensionless groups exist, the function generates dimensionless data and fits a lightweight symbolic neural network to model the relationship between transformed inputs and outputs. A standard training pipeline is executed: data preparation, model instantiation, training with backpropagation, and prediction. The model's mean squared error (MSE) is computed on the dimensionless targets to evaluate fit quality. Neural network outputs (predictions and gradients) serve as a flexible preliminary approximation, potentially capturing nonlinear relations that guide the subsequent, more rigid symbolic regression steps. Part 3: Polynomial Candidate Generation and Filtering
Following the neural fit, the function applies Polynomial Fitting (PF) techniques to generate candidate symbolic expressions. Polynomial terms are systematically composed using provided variables, coefficients, and operators up to a specified degree. Generated expressions are filtered based on structural criteria, such as variable presence and coefficient validity, to ensure physical plausibility. Each polynomial is evaluated against the original data, and a best-fit candidate is selected according to its error score. If a candidate perfectly matches (zero error), the function outputs the discovered symbolic expression and terminates. Part 4: Brute Force Symbolic Search and Evaluation
If no polynomial yields a perfect fit, the function initiates a Brute Force (BF) symbolic search. It exhaustively generates expressions by combining variables, constants, operators, and powers. Multiple filters (symmetry, variable relevance, power range, constant handling) systematically prune the expression set to reduce computational complexity and preserve physically meaningful candidates. The remaining expressions are evaluated against the dataset, seeking the best match based on performance metrics. This final exhaustive step ensures that even highly non-obvious symbolic relations can be discovered if they exist within the defined operator and degree space.

Module:	*Tests:*	*Results:*
Dimensional Analysis	Get, Solve, Generate, Symbolic Transformation	Passed
Neural Network	Load, Network, Train, Predict, Gradient	Passed
Plotting	Pareto, Gradient	Passed
Polynomial Regressor	Load, Generate, Filter, Evaluate, Best fit	Passed
Brute Force	Load, Generate, Recursive, Evaluate, Loss, Variable Filtering, Constants, Powers	Passed
Main Solver	Main Solver, Laws	Passed

4.8 Testing:

Every module, function and file, were thoroughly tested using dedicated tests. The boundary conditions and inserted tests to make sure the function behaved as envisioned. There was also significant robust testing functions written when chaining together various techniques and models, in order to ensure everything was working smoothly.

4.8.1 Unit Testing:

I've written unit tests for every function, every evaluation method and have rigorously tested their functionality, edge cases, normal inputs etc.

Rigorous unit testing was employed to validate the correctness of each module within the symbolic regression framework. Key components including Pareto frontier operations, polynomial generation and evaluation, and dimensional analysis routines were individually tested using structured unit tests. Each test assessed expected outputs under standard and edge-case inputs.
Test results confirmed correct behavior, and outputs aligned with theoretical expectations (e.g., correct Pareto set retrieval, successful polynomial fitting on synthetic data, and dimensionally consistent variable transformations). A summary of testing outcomes is shown below. Furthermore, synthetic datasets based on known biological formulas were used to validate the regressor's ability to recover exact symbolic relationships, demonstrating the system's practical reliability.

4.8.2 Integration Testing:

To validate the interoperability and functional correctness of the developed software system, a comprehensive suite of integration tests was designed and executed. These tests focused on the interactions between the core Python modules, including Dimensional Analysis (dimensionalAnalysis), Polynomial Fitting (polynomialFit), Brute-Force Symbolic Regression (bruteForce), Neural Network (neuralNetwork) components, Pareto front optimization (pareto), and visualization (plots). Utilizing Python's unittest framework, the tests simulated realistic data analysis workflows, verifying the seamless flow of data and control between modules.

Key tested interactions included the preprocessing of data using Dimensional Analysis feeding into various model fitting approaches (PF, BF, NN), the evaluation of candidate expressions generated by BF and PF modules and their subsequent management on a Pareto front, and the symbolic transformation capabilities of the Dimensional Analysis module. Further tests confirmed the internal consistency of complex operations within the Brute-Force module and the functionality of gradient calculations in the Neural Network. Successful execution of these tests demonstrates the robustness of the integrated system and its capability to perform complex, multi-stage symbolic regression tasks as designed, ensuring reliable data handoffs and component communication within the software architecture.

4.8.3 Performance Testing:

To quantitatively evaluate the computational efficiency and scalability of the core symbolic regression components, dedicated performance tests were executed on the bruteForce and polynomialFit modules. Utilizing Python scripts leveraging the time module for execution duration measurement and the psutil library for monitoring Resident Set Size (RSS) memory usage, these tests targeted the primary computational bottlenecks: symbolic expression generation and expression evaluation against numerical data.

The methodology involved systematically varying key parameters influencing complexity, such as expression generation depth (bruteForce), maximum polynomial degree (polynomialFit), the number of input variables, and the size of the input data sample for evaluation. By recording execution time and memory consumption under these varying conditions, the tests aimed to characterize the performance scaling of each module's algorithms. This analysis provides crucial insights into the computational complexity and resource requirements associated with each approach, enabling an understanding of their practical limitations and scalability characteristics. The results quantify the performance trade-offs inherent in different symbolic search strategies and inform the feasibility of applying the developed system to large-scale scientific discovery tasks by establishing empirical benchmarks for time and memory demands.

4.8.4 AI-Feynman Dataset:

5 Applying the model to Biological Data:

However, its application to biological systems remains an underexplored frontier. Biological processes are inherently noisy, high-dimensional, and complex, often lacking explicit governing equations. As a result, there exists a significant research gap in employing symbolic regression techniques for biological data modeling. A limited number of studies attempt this integration, making it a novel and potentially revolutionary field of study.

In this work, we apply symbolic regression to a fundamental biological process: the prediction of nucleic acid melting temperature (Tm) from DNA/RNA base counts. The melting temperature is computed through a well-established empirical formula known as the Wallace rule, defined as:
Tm=2(A+T)+4(G+C)
Tm=2(A+T)+4(G+C)

where A,T,G,CA,T,G,C denote the counts of adenine, thymine, guanine, and cytosine bases, respectively.

To test the approach, synthetic data is generated programmatically. Arrays are constructed where each sample consists of four integer inputs corresponding to base counts. The target yy values are computed exactly according to the Wallace formula. Variables (['A', 'T', 'G', 'C']) and constants ([2, 4]) are explicitly defined to align with the biological context.

The symbolic regression system first performs dimensional analysis, even though biological units are less rigorously defined compared to physics. Then, using a neural-symbolic approach followed by polynomial and brute-force search stages, the system attempts to recover the original expression.

This study highlights how biological systems can be rendered into symbolic, mechanistic expressions, bridging the gap between empirical observation and interpretable models, and opening new avenues for data-driven biological discovery.