AI-ASSISTED DISCOVERY OF PHYSICS LAWS

Harinarayan Asoori Sriram, Ishani Bakshi, Katherine Estevanell, Henry Gaus, Oliver Kahng, Aarav Khatri, Chloe Krawczak, Logan Miller, Joshua Moore, Prasham Shah, Kaitlin Zhang

Advisor: Dr. Minjoon Kouh
Assistant: David Hoyt

ABSTRACT

The discovery of fundamental physics laws has traditionally required extensive human analysis of experimental data. However, as the scope of the field grows, so does the complexity and volume of the data, and in turn, the difficulty of finding underlying physics laws and their mathematical formulations. In this study, we took advantage of advances in artificial intelligence, namely core components of the "AI Feynman" algorithm (Udrescu and Tegmark, 2020), to explore how algorithms inspired by physics properties can discover underlying equations from data. Our approach combines linear and polynomial regression with symmetry detection, powered by artificial neural networks, to sequentially simplify and fit data. We tested our system on synthetic datasets representing various physics equations, evaluating its ability to identify translational, sum, and scale symmetries, as well as its performance under noise. Our results show that the algorithm performs well on low-noise datasets and that the range of data should be considered for symmetry detection. However, our system is built to discover a limited class of algebraic equations. Future work may include an expansion of this algorithm to deal with differential equations or higher noise levels. This work confirms the physics-based approach of the AI Feynman in equation discovery, highlights its current limitations, and suggests potential avenues for future improvement of such methods.

INTRODUCTION

Since the 17th century, physics has been advancing through the rigorous and ongoing activities of taking careful observations about the physical world and capturing them concisely as mathematical equations. For instance, from the study of kinematics, one-dimensional motion with constant acceleration can be described by the equation

$$x_f = x_0 + v_0 t + \frac{1}{2}at^2 \qquad \text{(Equation 1)}$$

where the predicted position of x_f is determined by the initial position of x_0, the initial velocity of v_0, the time t, and a, the acceleration. This equation involves several constants, coefficients, variables, and variables raised to different powers. Similarly, consider Newton's Law of Universal Gravitational attraction, expressed as

$$F_g = \frac{Gm_0 m_1}{(x_0 - x_1)^2} \qquad \text{(Equation 2)}$$

where the force of gravity is F_g, the gravitational constant is G, m_0 and m_1 are the masses of two separate objects, x_0 is the position of the first object, and x_1 is the position of the second object. This equation, too, involves multiple constants, variables, and their algebraic combinations. These equations were discovered through rigorous analysis of data and deep insights by physicists. Not to mention that there also had to be a careful consideration of the effects of noise—random fluctuations in the measurement or recording processes that displace data points from where they theoretically should be—when deriving these equations. As such, we present the principal question of our project as follows: Is it possible to automate the discovery of these equations using an algorithm?

Motivation

Our approach is heavily inspired by—and, in some ways, a continuation of—Udrescu and Tegmark's "AI Feynman" [1]. "AI Feynman" is an algorithm that thinks *simple* and goes back to the roots of computational equation fitting, with one key difference. While "AI Feynman" ultimately just fits lines—the simplest data modeling challenge—it uses data table transformations and AI-powered symmetry detection to fit near any physics function, including all 100 functions in the Feynman dataset (a very diverse set of physics equations).

Inspired by their work and eager to develop some aspects of "AI Feynman" further, we embarked on a journey to replicate and experiment with the core elements of the algorithm. Why? We wanted to see how far their technique can *practically* assist in rediscovering the laws of nature and uncovering new ones.

Symmetries in Physics

Many physical laws exhibit symmetries [2]. Broadly, a symmetry is a transformation that leaves a system unchanged. For instance, a square exhibits 90-degree rotation symmetry, since it remains the same after being rotated 90 degrees about its center. It also exhibits mirror symmetry about certain lines (e.g., a line through the middle of the square) since it remains the same after being reflected about those lines. However, it does not exhibit rotational symmetry for all angles; for example, a square does not look the same after being rotated through 45 degrees.

A common example in physics is translational symmetry, in which physical laws are upheld even after a system's position has changed. A scientist who experiments in one location in the universe would expect the same results were it performed in another location, as demonstrated in equation 2. Take Newton's Law of Gravitation, which states that the gravitational force between two objects is proportional to their masses and inversely proportional to the square of the distance between them (Equation 2).

The actual coordinates of each object individually are not relevant to F_g; rather, it only depends on the difference between their positions—the distance between them. Thus, if one were to transport the Earth and Sun to a different galaxy while maintaining the distance between them, the gravitational force would remain unchanged even though the individual coordinates have changed. We can exploit this property to reduce the number of independent variables in the data. As an example, consider if the individual positions of the Earth and Sun are recorded as x_0 and

x_1, respectively, as shown in Equation 2. As was previously established, the individual values of x_0 and x_1 are not relevant to the output, but their difference is. This can be demonstrated mathematically by adding some constant, α, to both x_0 and x_1. This is analogous to moving the Earth and the Sun to a different galaxy while keeping the distance between them the same.

$$\frac{Gm_0m_1}{((x_0+\alpha)-(x_1+\alpha))^2} = \frac{Gm_0m_1}{(x_0-x_1)^2} = F_g \qquad \text{(Equation 3)}$$

Through this simple test, we show that Newton's Gravitational Law displays translational symmetry. More importantly, we determined that the individual values of x_0 and x_1 are not relevant to the output; rather, the relationship between the two variables—in this case, their difference—is what is important. Thus, in trying to derive a mathematical expression for F_g, it is not useful to keep track of both variables independently in the dataset. Instead, we can condense x_0 and x_1 into a single variable that represents their difference, Δx, encapsulating all the information relevant to the output. In that way, we reduce the number of independent variables from four to three, decreasing the amount of computational effort required to solve the problem.

Although symmetries refer to a broader category of actions or transformations that leave a given system unchanged, for our purposes, we consider a symmetry to exist between two variables when we can apply some paired transformation to two independent variables without changing the function's output. A paired transformation could be adding α to both variables, or adding α to one variable while subtracting α from the other, or multiplying a variable by α while dividing the other by α. This is essentially a mildly generalized version of the "α test" shown above. If we find that a symmetry exists, we know that the output is solely dependent on the two variables' relationship rather than their individual values. Such variables can be condensed into a single variable that encapsulates that relationship. In this paper, our approach builds upon the fundamental symmetries behind many physics laws. We use artificial neural networks to test and identify these symmetries.

METHODS

Our system reimplements a few key portions of the AI Feynman algorithm by Udrescu and Tegmark [1], with the main goal of finding the relation between independent variables and a singular dependent variable. The overall pipeline first performs linear regression, and then, sequentially modifies our existing independent variables by transforming them to higher degrees and also reducing variables to their relationships with each other (Figure 1). At each step, we check the goodness-of-fit, or how well the current version of the discovered equation can describe the data, stopping execution once an accurate-enough equation is reached. The major steps of the pipeline are as described in Figure 1.

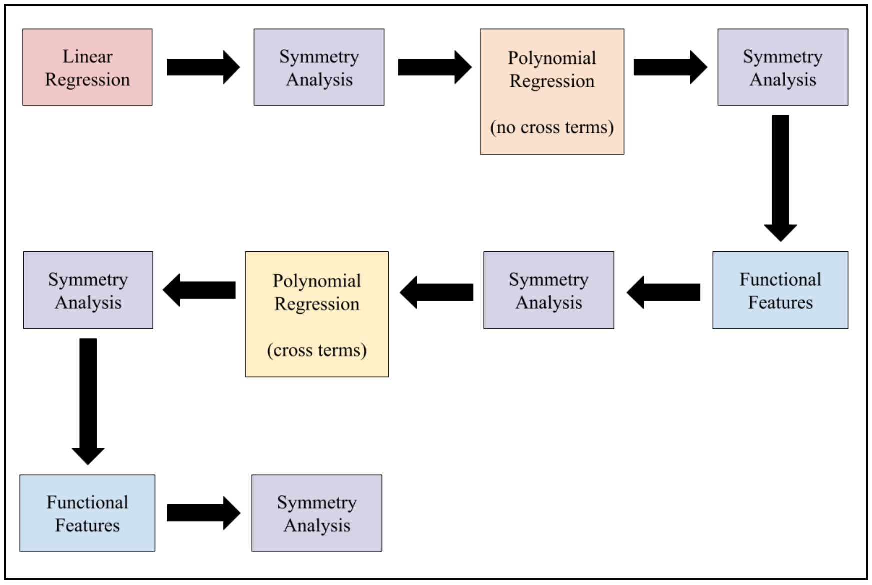


Figure 1 - The flow chart illustrates how our system discovers the underlying equation of the data. Each box represents a distinct step that the algorithm completes. We transition from basic linear regression to polynomial regression without cross terms to functional features to polynomial regression with cross terms to functional features once again. We interweave symmetry checks between each of those steps.

Software Tools and Platform

When designing this system, we used Python for its prevalence in machine learning applications, using Python's Numpy for data manipulation and scikit-learn for linear and multilayer perceptron (MLP) regression, methods for approximating our functions. Libraries like scikit-learn [2] and NumPy [3] provide mathematical functions needed to create linear regression, a statistical method that fits the relationship between independent variables and a dependent variable to a linear function.

Linear Regression

Typically, linear regression is viewed through the lens of single variable linear regression, similar to slope-intercept form (e.g., $y = mx + b$). However, in the application of physics, many equations are multivariate (e.g., $x_f = x_i + vt$), meaning we are trying to find the relationship between multiple independent variables (x_i, v, t) and the dependent variable. The formula for multivariate regression can be described as [5]:

$$y = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n \qquad \text{(Equation 4)}$$

Specifically, β_0 is the y-intercept of our equation, or in other words, the value of y when all the independent variables are equal to 0. Continuing, β_n represents the coefficient of each of the respective independent variables x_n, which means that increasing x_n by 1 would increase the y-value by β_n. The goal of regression is to find the best weights β by minimizing the loss function, which in this case is called the residual sum of squares, also known as RSS:

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \qquad \text{(Equation 5)}$$

In this equation, *n* represents the number of data points, y_i represents the actual y-value, and $\hat{y}$ represents the predicted value. In simpler terms, RSS represents the sum of the squared difference between the actual and predicted y-value. Overall, by squaring the errors, RSS provides a robust method to avoid negative and positive errors canceling out and amplifying larger mistakes. To find the best-fit line, the algorithm identifies the equation that would have the smallest RSS. By minimizing RSS, the algorithm ensures that the selected equation is the closest to the actual observed data points. After establishing our linear model, we evaluate its performance using the R^2 (R-squared) statistic. This metric tells us what proportion of the variability in our target variable can be explained by our input variables. R^2 values range from 0 to 1, where 0 indicates that our model explains none of the variance (essentially no better than guessing the average value every time), and 1 indicates perfect prediction, where our model captures all the underlying patterns in our data. Our system sets a target R^2 value of 0.99, which is an extremely tight fit, as for the most part, we are working with zero-noise, synthetic data. If it exceeds the target value, the system returns the solved linear solution.

Polynomial Regression

Yet, in many cases, even multivariate linear regression cannot explain physics equations. For instance, just using multivariate linear regression, we would not be able to discover the following equation:

$$x_f = x_0 + v_0 t + 0.5at^2 \qquad \text{(Equation 6)}$$

Multivariate linear regression is not able to linearly relate t^2 to x_f, so linear regression alone cannot discover this equation from a data table. However, such polynomial equations are abundant in physics, as evidenced by the above equation. Therefore, if a simple linear model does not fit the data well, the algorithm next checks to see if the target can be modeled as the sum of polynomials of each input variable (Figure 2). To do this, we expand the feature set by raising each input variable to increasing powers (e.g., 2 through d , where $d = 6$ in our implementation). This expands the design matrix with new columns for x^2, x^3,…, x^d for each original variable, still without introducing interaction (cross) terms between variables. In effect, this fits a model where the target is expressed as a sum of low-degree univariate polynomials in each variable. If the best-fit polynomial of degree $\leq d$ achieves a sufficiently low error, this process will find the underlying polynomial expression. If not, we later proceed to a more expressive model: one that can figure out a polynomial that includes cross terms, where x_1 could be multiplied by x_2 instead of by itself, as later described in *Polynomial Cross Terms Generation*.

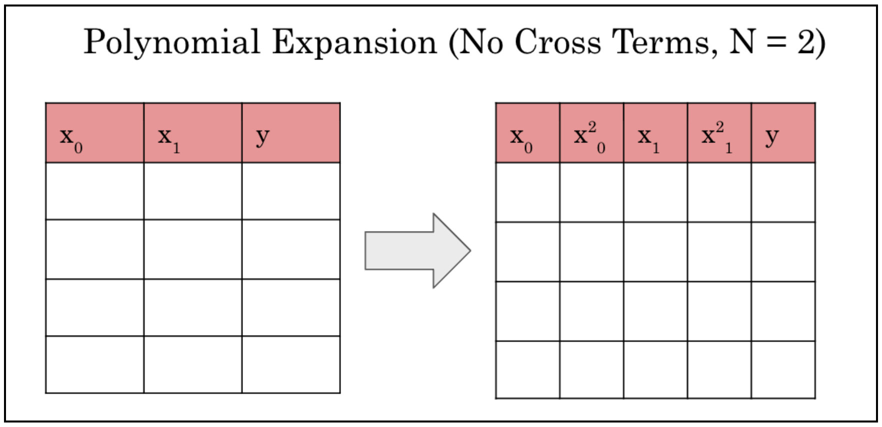


Figure 2 - This representation of polynomial expansion shows columns being added to the data table, which contain the original independent variables raised to higher powers.

Taking the example of:

$$x_f = x_0 + 0.5gt^2 \qquad \text{(Equation 7)}$$

Considering no initial velocity and the constant acceleration of gravity, if our maximum degree of polynomial was set to 2, our algorithm would generate the following set of variables:

$$x_0,\ x_0^2,\ t,\ t^2 \qquad \text{(Equation 8)}$$

Each of these variables would be assigned a weight β, as described in *Linear Regression*, and the algorithm would continue to tune these weights around until the least possible RSS would be achieved. The coefficients would be approximately 0 for x_0^2 and t, 1 for x_0 and $0.5g$ for t^2. As the equation would perfectly fit any data described by $x_f = x_0 + 0.5gt^2$, our target R^2 value would be surpassed, and the model would return the solution.

Combinatorics Justification for Reducing Variables

From a combinatorial standpoint, we consider the following: the restricted case of fitting polynomials of degree at most d in p variables. The number of terms of total degree $\leq d$ in p variables is given by the formula:

$$N_{poly} = \binom{p+d}{d} = \frac{(p+d)!}{p!d!} \qquad \text{(Equation 9)}$$

Even this restricted function class grows combinatorially. For instance, with $p = 3$ variables and $d = 4$,

$$N = \binom{7}{4} = 35 \qquad \text{(Equation 10)}$$

For example, different terms with variables x_0, x_1, x_2, and degree 4 might include x_0^4, $x_0^3x_1$, $x_0x_1x_2^2$, and so on.

With p = 5, d = 6:

$$N = \binom{11}{6} = 462 \qquad \text{(Equation 11)}$$

If each term is allowed an independent coefficient, the parameter space already becomes high-dimensional, complicating any regression or brute-force evaluation scheme.

It is also important to note that polynomial and linear regression together will not work on all physics equations, particularly if we do not explicitly consider cross terms, or terms that contain products of different variables. For example, consider the physics equation

$$E = \frac{1}{2}k[(x_{eq} - x)^2 + (y_{eq} - y)^2 + (z_{eq} - z)^2] \qquad \text{(Equation 12)}$$
$$= \frac{1}{2}k[x^2 - 2xx_{eq} + x_{eq}^2 + y^2 - 2yy_{eq} + y_{eq}^2 + y^2 - 2zz_{eq} + z_{eq}^2]$$

This equation contains several cross terms, such as -$2xx_{eq}$ and -$2yy_{eq}$, all of which would be unable to be deciphered simply through linear or polynomial regression without cross terms.

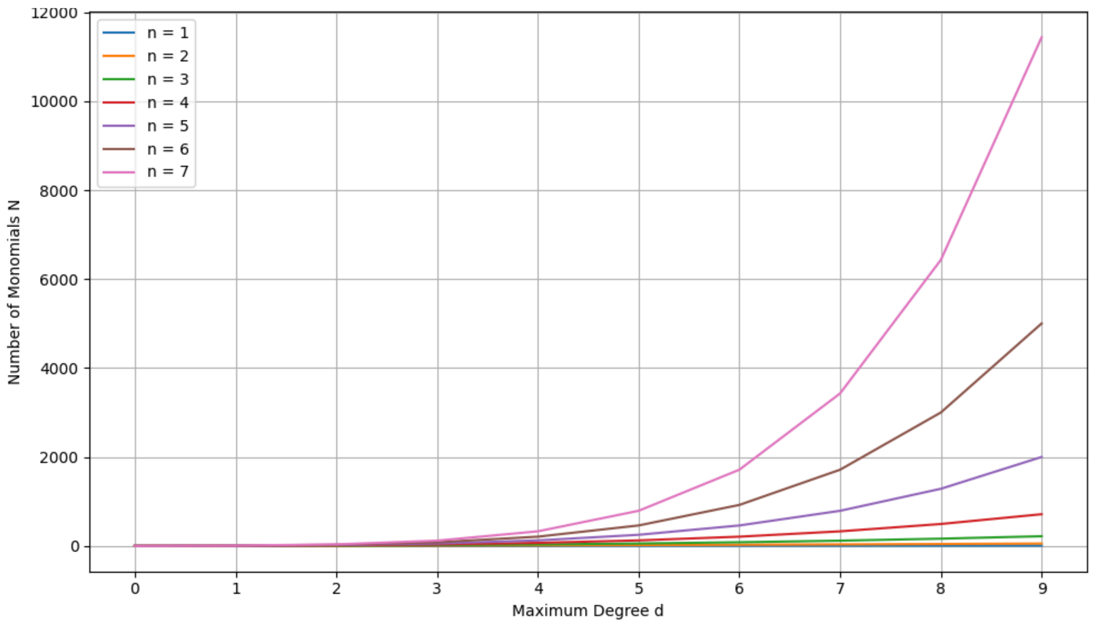


Figure 3 - The maximum degree *d* and the number of variables *n* affect the number of possible monomials *N*. *N* increases rapidly with both *n* and *d*, highlighting the combinatorial infeasibility of checking every possible monomial term when trying to fit more complex data generated from more complex equations.

The number of monomials of total degree $\leq d$ in *p* variables grows at a rapidly increasing rate. However, as the number of variables is reduced, the number of terms is reduced considerably.

Similarly, we notice that just as when the degree and number of variables are increased, the number of classes of polynomials increases dramatically, it follows that when the degree and number of variables are decreased, the number of classes of polynomials likewise decreases. It is from this concept that the idea of reducing variables through symmetry emerges. If we can find a way to reduce the number of variables in a given dataset, then it becomes exponentially easier to solve for the equation at hand.

Reducing the Number of Variables

As established earlier, we can reduce the number of independent variables in the data by exploiting symmetries commonly exhibited by physical laws. Specifically, we search for symmetries in which the output is not dependent on the individual values of two variables, but instead on the relationship between them. In other words, only the relationship between those two variables contributes to the output. This was explored earlier in *Symmetries* through the specific example of translational symmetry in Newton's Law of Gravitation, but translational symmetries also apply to many other physical laws. To further demonstrate how our algorithm identifies these symmetries, consider the equation for elastic potential energy.

$$U_e = \frac{1}{2}k(x_{eq} - x)^2 \qquad \text{(Equation 13)}$$

Here, x_{eq} is the position of the spring at equilibrium, x is the position of the displaced spring, and k is a constant specific to each spring. Figure 4 demonstrates that U_e is only dependent on the difference between x and x_{eq}, not their individual values. For instance, if the entire system is shifted right by some constant, α, the rest position and displaced position of the spring change. However, since they change by the same amount, the distance between them is conserved, yielding the same U_e.

This is similar to the idea explored in *Symmetries*, where translating the x-coordinates of two objects by α does not change the gravitational force between them because the distance between the objects is conserved. Like with the Law of Gravitation, this demonstrates that elastic potential energy displays translational symmetry. Moreover, we find that the individual values of x and x_{eq} do not contribute to the output (since changing them yields the same result). Therefore, in deriving a mathematical expression for U_e from data, we can simplify the problem by combining x and x_{eq} into a single data column called "(x_{eq} - x)." We can generalize this to say that for any mystery function f, if $f(x_0, x_1,...) = f(x_0 + \alpha, x_1 + \alpha, \ldots)$, then there exists a translational symmetry between the variables x_0 and x_1. In such a case, it is not useful to individually track the values of x_0 and x_1, so we condense them into a single variable that encapsulates their difference because that is all that is relevant to the output. In doing so, we reduce the number of variables, simplifying the problem.

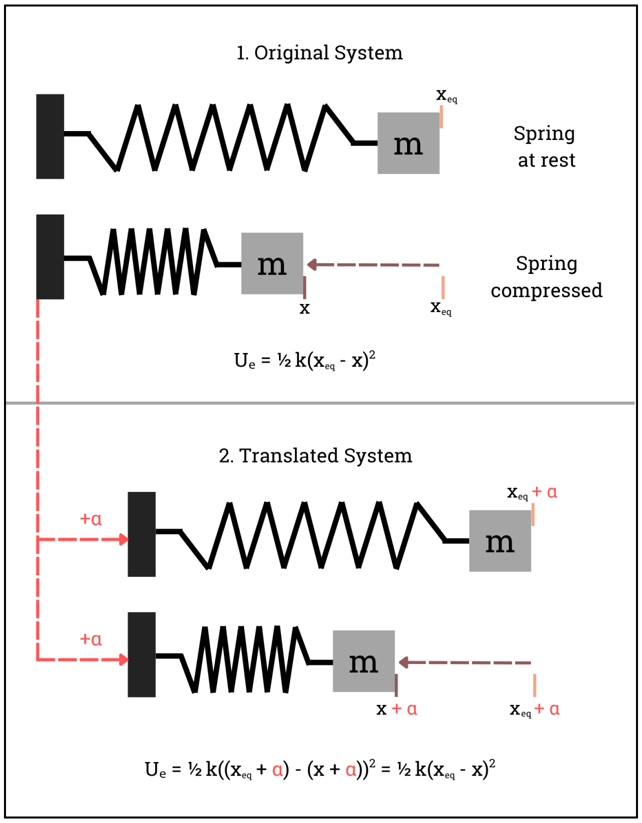
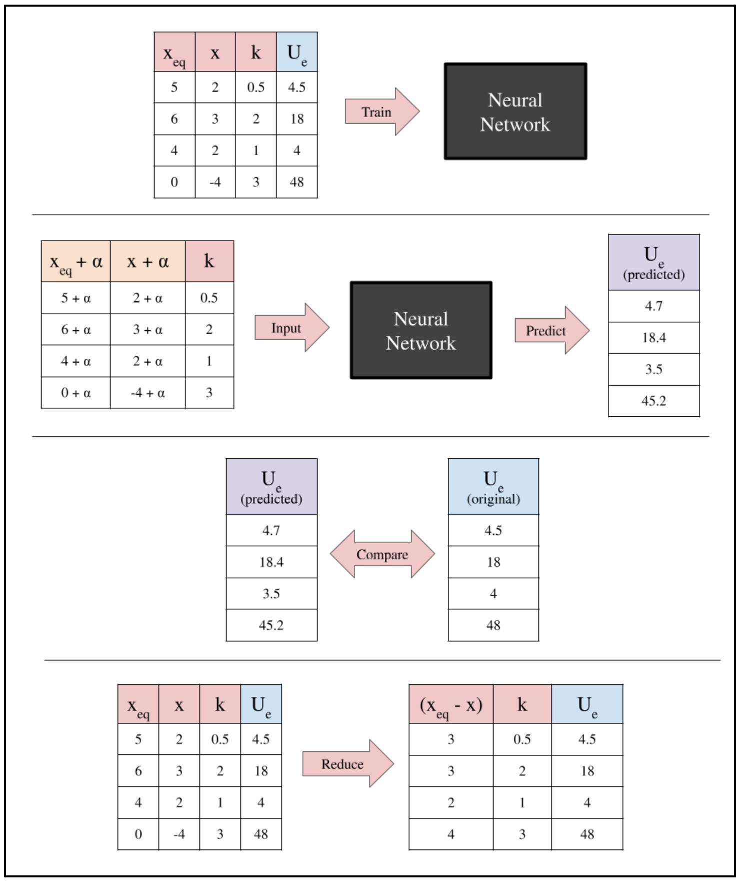


Figure 4 - This diagram shows a spring-mass system before and after a horizontal shift by α. Note that while the positions of the mass and spring change, the compression does not, highlighting the translational symmetry between x and x_{eq}.

This simple test for translational symmetry, adding a constant to two variables, is extremely useful in reducing the complexity of the data. However, as we do not know the mystery function *f*, we cannot simply plug in the transformed values. This is where our neural network comes into play. To illustrate this, we again examine the equation for elastic potential energy, specifically looking at how we would identify the translational symmetry between x and x_{eq}. As shown in Figure 5, we train a neural network to model the original data. Then, we transform the columns of interest, in this case x and x_{eq}, by adding α to each. Now, we can use the neural network to predict new outputs given the transformed $(x + α)$ and $(x_{eq} + α)$ columns. In this way, we use the neural network to predict what $f(x + α, x_{eq} + α, …)$ should output, even if we do not know what *f* is. If the neural network's predictions based on the translated data are close enough to the original data, we conclude that there exists a translational symmetry between x_{eq} and x. We then remove the x_{eq} and x columns, replacing them with a new one, $(x_{eq} - x)$. In searching for symmetries, we perform this test on every combination of two columns.

The importance of this approach becomes more apparent when considering more complex equations. Take the elastic potential energy equation expanded into three dimensions (Equation 12). This equation exhibits three translational symmetries: $(x_{eq} - x)$, $(y_{eq} - y)$, and $(z_{eq} - z)$. Thus, using a neural network to identify these symmetries allows us to decrease the number of variables by three (one for each symmetry), making the task much more computationally manageable. Furthermore, this approach is extensible beyond translational symmetry. For instance, if two variables, x_0 and x_1, are being multiplied, and only their product is relevant to the output, then we can condense them into a single variable, x_0x_1.To test for this, we use a neural network to check if $f(x_0, x_1, \ldots) = f(x_0 \cdot \alpha, x \cdot 1/\alpha, \ldots)$, which is similar to testing for translational symmetry. This would be extremely useful in attempting to derive an equation for the Law of Gravitation (Equation 2). Although F_g is dependent on the product of m_0 and m_1, the individual values of the two variables are not relevant. This provides an opportunity to reduce the number of independent variables by condensing the m_0 and m_1 data columns into a single $(m_0 \cdot m_1)$ column.

The significance of this method is that it does not approach the task of deriving equations from a purely mathematical standpoint. Instead, it takes advantage of our knowledge of physical laws, specifically exploiting symmetries to reduce the number of independent variables. In this way, we can take complex mathematical patterns and break them down into simpler relationships, making the task of deriving an equation from data feasible despite the large combinatorial space of potential expressions.



x_{eq}	x	k	U_e
5	2	0.5	4.5
6	3	2	18
4	2	1	4
0	-4	3	48

$x_{eq} + \alpha$	$x + \alpha$	k
$5 + \alpha$	$2 + \alpha$	0.5
$6 + \alpha$	$3 + \alpha$	2
$4 + \alpha$	$2 + \alpha$	1
$0 + \alpha$	$-4 + \alpha$	3

U_e (predicted)
4.7
18.4
3.5
45.2

U_e (predicted)
4.7
18.4
3.5
45.2

U_e (original)
4.5
18
4
48

x_{eq}	x	k	U_e
5	2	0.5	4.5
6	3	2	18
4	2	1	4
0	-4	3	48

$(x_{eq} - x)$	k	U_e
3	0.5	4.5
3	2	18
2	1	4
4	3	48

Figure 5 - The process of identifying symmetries begins with training the neural network on the original data. Then, the columns of interest are transformed. The neural networks predict the outputs of these transformed columns, and these predictions are compared to the original outputs. If the error of the model is low enough, we conclude that a symmetry exists, and we condense the columns of interest into one that encapsulates their relationship.

Using Neural Networks to Fit Functions

The usage of symmetries to reduce the number of columns intuitively only seems possible with prior knowledge of the actual function or direct testing in the lab. However, symmetries can still be effectively identified if the function can be well approximated from the data. A very powerful way to accomplish this is through the use of machine learning models, and, in our particular case, a type of artificial neural network model called a multi-layer perceptron, or MLP.

Neural networks consist of several layers of "neurons," with adjustable connections between neurons in adjacent layers. A certain non-linear function called an activation function is also applied in between these layers to help the models fit more complex data. Neural networks can be fit or "trained" in a way similar to linear regression, where parameters are tuned in such a way that the error of the model is minimized. In our case, this error is equal to the residual sum of squares divided by the number of data points.

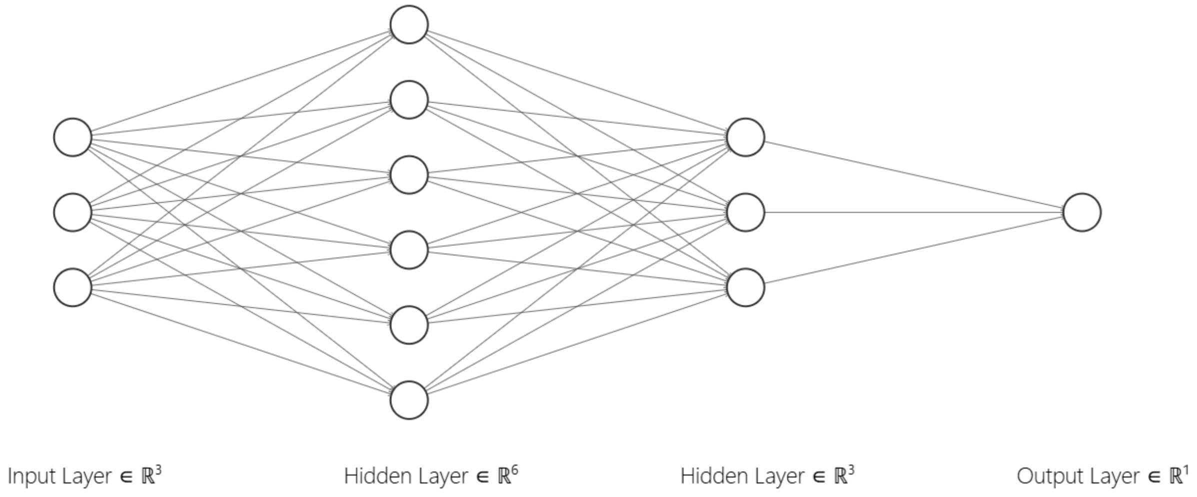

Input Layer $\in \mathbb{R}^3$
Hidden Layer $\in \mathbb{R}^6$
Hidden Layer $\in \mathbb{R}^3$
Output Layer $\in \mathbb{R}^1$

Figure 6 - This is a sample multi-layer perceptron, showcasing the core architecture behind our MLP model. There are several interconnected hidden layers, and each node is connected to the entire layer before it and after it. However, we use a more complex MLP with hidden layer sizes of 500, 200, 100, 50, 20, and 10 to capture the nuances of the function we are trying to find better. [6]

By training our neural network, we can approximate the function very well (for x_i values close to the range present in the dataset), but we do not directly gain strong insight into the nature of the function. In this sense, a neural network functions like a "black box," where we can take in inputs and get outputs, but do not understand the relationship between the variables. However, this gives us a path to effectively test for symmetries by considering the underlying function $f(x_0, x_1, ...)$ and the approximation we have from the neural network $\hat{f}(x_0, x_1, \ldots)$ [7].

As discussed previously, if the variables x_0 and x_1 have translational or subtractive symmetry, then $f(x_0, x_1, \ldots) = f(x_0 + \alpha, x_1 + \alpha, ...)$ for some number α. In other words, adding α to both variables should leave the result unchanged. However, we cannot directly compute this translated term since we do not know the actual function, but given that the function $\hat{f}$ is a good approximation of the function f, we can say that if such a symmetry exists, $f(x_0, x_1, \dots) \approx \hat{f}(x_0 + \alpha, x_1 + \alpha, ...)$. Equivalently, if x_0 and x_1 have a subtractive symmetry, adding α to both variables and approximating the function value with a neural network should give an output that is similar to the original data. By checking whether this is the case, we can look for symmetries between variables and simplify accordingly.

This argument generalizes to other mathematical relationships, such as addition and multiplication. After we find a symmetry and perform the corresponding simplification, we can simply repeat the process with the reduced data, iteratively refining the equation until our model no longer finds any new symmetries.

Extension: Functional Features

Not all functions in physics consist of low-order polynomials. Many of them utilize transcendental functions, such as trigonometric, exponential, or logarithmic functions. While they can be represented in Taylor series, it requires a large number of terms to achieve reasonable accuracy, and they are difficult to interpret directly.

Consequently, we have extended our approach to include new data columns whose values are determined by pre-specified functions of the original variables. For example, if we have columns x_1, x_2, and x_3, we can add columns $sin(x_1)$, $sin(x_2)$, and $sin(x_3)$, after which we perform linear regression on the extended data table. Therefore, we can not only fit polynomials but also arbitrary functions as well, so long as we can identify the elementary functions they may be composed of.

Extension: Polynomial Cross Terms Generation

If none of the above strategies work, as a last resort, our algorithm will expand the data table with cross terms—terms containing products of different independent variables. For instance, when considering the full $x_f = x_0 + v_0t + ½at^2$ equation, if a multiplicative symmetry is not found due to some of the terms' complexity or our model's deficiencies, attempting cross terms would, albeit more slowly, still get us the correct answer.

Specifically, we generate features for all combinations of variables up to a given total degree (Figure 7). For example, with two variables x_0 and x_1 and max degree 6, this expansion yields terms like $x_0^2 x_1^4$ in addition to the simpler powers from the earlier no-cross expansion. In other words, we create a thorough polynomial feature matrix covering any monomials of total degree $\leq d$. Thus, a linear regression on this expanded basis will fit a general cross-term-including polynomial of all the variables up to maximum degree d. This step can capture interactions between variables that the previous no-cross model may not have.

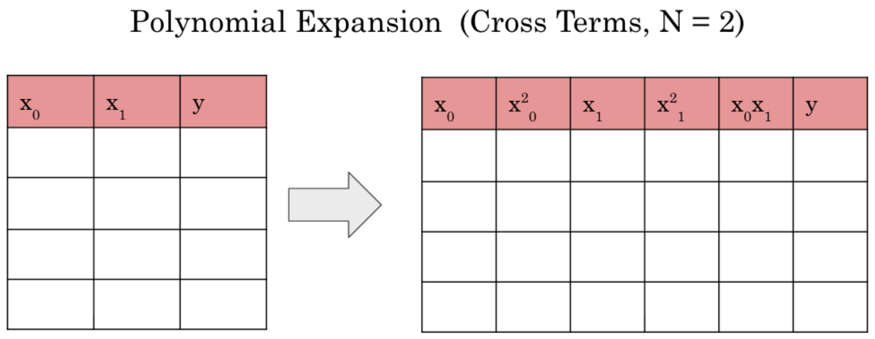


Figure 7 - This is a representation of polynomial expansion with cross terms. The particular cross term shown here is x_0x_1; the other terms would have been generated even in a no-cross-term polynomial expansion.

Considering the example of $x_f = x_0 + v_0t + ½at^2$, with a max degree of 3, the algorithm would generate every combination of terms up to a degree of 3, examples including:

$$x_0,\ x_0^2, x_0^3, \cdots\ v_0t, v_0^2t, \cdots\ at^2 \qquad \text{(Equation 14)}$$

Applying linear regression on these new terms, the algorithm would find the hyperplane with the smallest RSS as being $x_0 + v_0t + ½at^2$, yielding the final result.

RESULTS

Main Result

Our model had a baseline success rate of about 50% on some select equations, which cover all levels of difficulty, though they lean more toward the complex side (Appendix A). However, that percentage is not particularly important, as it depends on the complexity of an arbitrary dataset. What may be more interesting to explore, however, is the kinds of equations that our algorithm struggled with. For equations like the formula for average power and centripetal acceleration, the presence of a non-constant denominator proved troublesome for our algorithm. Although we have a "functional feature" to represent the *1/x* or inverse function, it is applied to the data table very late in the process, so it makes sense that equations with variable denominators were relatively difficult to discover. Many other failures can also be attributed to the sometimes poor timing of functional feature application.

However, one function that the algorithm could not have derived, even if functional features had been integrated perfectly, was the stress-strain relationship, which involved a variable exponent (rather than just a variable base, as in a polynomial). Although there exist techniques for finding exponential functions using logarithmic transformations, no version of our core algorithm would have been able to find the stress-strain relationship. Even equipped with logarithmic transformations, our algorithm still may have failed, as both the base and the exponent are variable in the stress-strain relationship, which significantly complicates the situation. Still, we are proud to say that our algorithm successfully tackled many different classes of physics problems. Moreover, as the discussion here illustrates, our system can easily be extended to do more in the future.

Additional Experiment 1: Effect of Noise

We generated data matrices of 10,000 simulated observations, where the independent-variable columns were randomly sampled from a uniform distribution ranging from -10 to 10, and the final dependent-variable column was calculated with a specified function of these independent variables. Every element of the matrix was then modified with additive noise, sampled from a normal distribution with varying standard deviations, which served as different noise levels. Specifically, the standard deviations or noise levels we tested were 0.0, 0.2, 0.4, and 0.6. We tested a variety of physics equations, primarily from classical mechanics. We ran our algorithm to completion, recording the last equation it found and the associated R^2.

Our equation-discovery algorithm performed worse at higher noise levels, but the effect was even more pronounced for more complicated equations. For instance, for the relatively simple kinematics equation $v_f = v_i + at$, the model still found an approximate solution of the correct form, $v_f = 0.9951v_i + 1.0209at + 0.0897$ with $R^2=0.99$ at a noise level of 0.4 (Appendix B). However, for the oscillation-period equation from classical mechanics, $T = 2\pi/\omega$, which contains complex inverse functions, our equation-finding system collapsed at a noise level of just 0.2 ($R^2 = 0.003$). The equation-finding algorithm similarly struggles with equations with several multiplied terms of higher powers, such as the equation for kinetic energy $K = ½mv^2$, where the fit drops from $R^2=0.996$ at a noise level of 0.2 to $R^2 = 0.045$ at a noise level of 0.4. Thus, informally, our system's performance on "complex" functions appears to degrade much more rapidly than for more "simple" functions at higher noise levels, although that delineation is somewhat arbitrary. See Appendix B for more examples.

Our system occasionally suffers from "underfitting" and "overfitting." For a case like the mechanical-energy equation $E = U_g + U_s + K$, at high enough noise levels, our algorithm underfitted and just outputted a constant: $E = 0.6082$ at a noise level of 0.6 (Appendix B). Conversely, in trying to fit laws like the gravitational-potential-energy formula $U_g = mgh$ at high noise levels, our algorithm produced nonsensical overfit equations like the following:

$$U_g = 1.4682m^2 + 1.4682m - 1.4682g - 1.4682h + 4.1892sin((-m^3 + h^2)(mh^2 - g)) - 1.1642sin(mh^3 - g^4) + 3.6704sin(m^2g^2 - m^2gh) + 0.8092sin(m^2h^2 - mg^3) + 3.0341sin(gh^3 - h^4) + 2.2555sin(g^3h - g^2h^2) - 2.3759sin(mg^2h + mgh^2) - 2.1653sin(m^2 + m - g - h - 6.6109sin(m^4 + g^2h - gh^2 - h^3) + 1.9655sin(mg - mh - g^2 + gh) - 5.1442sin(m^2g - m^2h + mg^2 - mgh) - 15.77$$
(Equation 15)

Future studies may involve tuning the threshold of success (i.e., the cut-off score after which the system stops trying more complex fits) based on the suspected noise level. Such an early-stopping strategy may partially combat the effects of noise by accepting simpler functions that technically have a "worse fit," with the insight that simple equations are often correct in physics.

Additional Experiment 2: Effect of Data Range and Adaptive α

We explored how to determine the transformation parameters for symmetry detection (which are denoted as the α values) by experimenting with different data ranges. Recall that symmetries are discovered by performing a paired operation on two variables (e.g., adding 5 to both variables, or adding 5 to one variable and subtracting 5 from the other) and checking for any significant changes in the predicted outputs. A problem arises, however, when considering the magnitude of the α values. The choice of "5" earlier in this paragraph was completely arbitrary, and it happens to work well on numerical data ranging from about -10 to 10 in each column. However, on smaller-ranged datasets, such as ones where independent-variable values typically range from -1 to 1, a transformation delta of 5 may cause the MLP regressor to reject true symmetries simply because it could not extrapolate so far beyond its training data. On the other hand, for larger-ranged datasets, such as ones where values typically range from -100 to 100, a transformation of just 5 units may not be very significant.

We thus scale the transformation parameter in proportion to the standard deviation of the columns being tested for additive symmetries, and in proportion to the standard deviation of the order of magnitude of the columns being tested for multiplicative symmetries. We tested various equations containing both true symmetries and fake symmetries, using both the fixed-constant approach and adaptive α approach on uniformly distributed data with ranges from -0.5 to 0.5, -1 to 1, -10 to 10, -50 to 50, and -100 to 100.

Empirically, the adaptive approach to setting the α value was more successful than the fixed-constant approach in automatically dealing with large-ranging (e.g., -100 to 100) and small-ranging (e.g., -0.5 to 0.5) datasets (Appendix C). When we set our fixed additive constant to 5, we found that the symmetry algorithm was successful most often on uniformly distributed data ranging from -10 to 10, for which our adaptive-alpha formula predicts a very similar ideal transformation delta of about 5.8, further supporting our new alpha-finding approach.

DISCUSSION

This work is a replication of a few key components of the "AI Feynman" algorithm [1], using a "build-from-scratch" approach. We implemented linear regression, polynomial transformations, and symmetry analysis. Moreover, we explicitly described working with cross terms and incorporating non-polynomial functions as simply extensions of the core functionality of our system. Finally, we expanded on the work of "AI Feynman" by investigating and classifying noise more thoroughly and by finding a formula to adapt symmetry transformations to various data scales.

Our equation-finding approach has limitations. First, the symmetry detection may find false positives or non-existent symmetries due to a too "easy" (too small α) symmetry test. Considering the range of data through adaptive α adjustments is a partial workaround, but it's not perfect. Future research could focus on finding ways to deal with data range more efficiently and adaptively. Second, our current system struggles with functions containing non-integer powers (e.g., square roots) or nested compositions (e.g., *sin(exp((1/x))*) simply because we do not check for them in our extended data tables. Our "functional features" technique offers a partial solution, enabling us to consider arbitrary combinations of functions, including fractional exponents. However, to use functional features, we have to explicitly define what functions the algorithm should consider out of the infinitely large space of possible mathematical functions. A relatively straightforward extension of our work would be to incorporate *log* analysis (taking the *log* of the variables before running the main algorithm). That should enable the discovery of arbitrary exponents, which would be the coefficients of the *log*-modified terms. Still, even *log* analysis cannot handle functions with variable bases and exponents simultaneously, illustrating that there will always be a further frontier. At that point, the choice of how far to extend the core equation-finding system comes down to practicality: we know physics equations tend to be simple, so, in that spirit, we should avoid excessive complexity as much as possible. Third, our system struggles at high noise levels, especially with more complicated functions. Mathematical functions with many multiplied variables, variables in the denominator, variables with higher powers, and other complicated characteristics may lead to greater sensitivity to noise, as even slight noise can significantly obscure such a complicated relationship. Exploring more advanced pre-regression transformations, replacing our MLP with other AI systems, or tuning our MLP's parameters further could all be potential directions for future exploration.

Despite our limitations, this study highlights that symmetries in physics are fundamental and can be exploited to prevent a combinatorial explosion of possibilities in the process of automated equation discovery. Furthermore, symmetry detection can provide deeper insight into the structure of an equation and, ultimately, our physical reality itself.

REFERENCES

[1] S.-M. Udrescu and M. Tegmark, “AI Feynman: A physics-inspired method for symbolic regression”, Science Advances, vol. 6, no. 16, p. eaay2631, Apr. 2020, doi: 10.1126/sciadv.aay2631.

[2] R. P. Feynman, R. B. Leighton, and M. Sands, “Symmetry in physical laws,” in *The Feynman Lectures on Physics*, vol. 1, Reading, MA, USA: Addison‑Wesley, 1964, ch. 52, pp. 52‑1–52‑14.

[3] Pedregosa, F., et al. "Scikit-learn: Machine Learning in Python," in Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[4] C. R. Harris et al., "Array programming with NumPy," Nature, vol. 585, no. 7825, pp. 357–362, Sep. 2020, doi:10.1038/s41586-020-2649-2.

[5] S. Weisberg, *Applied Linear Regression*, 4th ed. Hoboken, NJ, USA: John Wiley & Sons, 2014.

[6] A. LeNail. "NN-SVG: Publication-Ready Neural Network Architecture Schematics," in *Journal of Open Source Software*, vol. 4, no. 33, pp. 747, 2019.

[7] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” Mathematics of Control, Signals, and Systems, vol. 2, no. 4, pp. 303–314, Dec. 1989, doi: https://doi.org/10.1007/bf02551274.

APPENDIX A

Equation	**Able to find a roughly correct relation?**	**R^2**
Total Mechanical Energy (TME) = $U_g + U_s + K$	Yes	1.000
Kinetic Energy (KE) = $\frac{1}{2}mv^2$	Yes	1.000
Gravitational Potential Energy (U_g) = mgh	Yes	1.000
$v_f = v_i + at$	Yes	1.000
Power (P) = $\frac{Fd}{t}$	No	0.995
$x_f = x_i + v_i t + \frac{1}{2}at^2$	No	0.936
Elastic Potential Energy $(U_e) = \frac{1}{2}k(x_f - x_i)^2$	Yes	1.000
$I = mr^2$	Yes	1.000
$F_{drag} = \frac{1}{2}CpAv^2$	No	0.359
$W = F(x_f - x_i)$	Yes	1.000
$T = \frac{2\pi}{w}$	Yes	1.000
$F_c = \frac{mv^2}{r}$	No	0.148
$j = \sigma T^4$	No	0.503
$P = \frac{nRT}{V}$	No	0.997
$F = mg \cdot cos(\theta)$	No	0.079
Stress Strain Relationship: $\frac{x_0}{x_1} + (\frac{x_0}{x_3})^{x_3}$	No	1.000

APPENDIX B

Equation	**Noise Level**	**Able to find a roughly correct relation?**	**R^2**
Total Mechanical Energy (TME) = $U_g + U_s + K$	0	Yes	1.000
	0.2	Yes	0.998
	0.4	Yes	0.993
	0.6	No	0.685
Kinetic Energy (KE) = $\frac{1}{2}mv^2$	0	Yes	1.000
	0.2	Yes	0.996
	0.4	No	0.045
	0.6	No	0.464
Gravitational Potential Energy (U_g) = mgh	0	Yes	1.000
	0.2	Yes	0.996
	0.4	No	0.005
	0.6	No	0.158
$v_f = v_i + at$	0	Yes	1.000
	0.2	Yes	0.998
	0.4	Yes	0.990
	0.6	No	0.846
Power (P) = $\frac{Fd}{t}$	0	No	0.995
	0.2	No	-0.385
	0.4	No	-0.405
	0.6	No	-0.675
$x_f = x_i + v_it + \frac{1}{2}at^2$	0	No	0.936
	0.2	No	0.933

	0.4	No	0.923
	0.6	No	0.907
Elastic Potential Energy $(U_e) = \frac{1}{2}k(x_f - x_i)^2$	0	Yes	1.000
	0.2	No	0.997
	0.4	No	0.644
	0.6	No	0.633
$I = mr^2$	0	Yes	1.000
	0.2	Yes	0.996
	0.4	No	0.629
	0.6	No	0.002
$F_{drag} = \frac{1}{2}CpAv^2$	0	No	0.359
	0.2	No	0.011
	0.4	No	0.005
	0.6	No	0.006
$W = F(x_f - x_i)$	0	Yes	1.000
	0.2	Yes	0.998
	0.4	Yes	0.990
	0.6	No	-0.033
$T = \frac{2\pi}{w}$	0	Yes	1.000
	0.2	No	0.003
	0.4	No	0.003
	0.6	No	0.0004
$F_c = \frac{mv^2}{r}$	0	No	0.148
	0.2	No	-0.032
	0.4	No	-0.259
	0.6	No	-0.271

$j = \sigma T^4$	0	No	0.503
	0.2	No	0.503
	0.4	No	0.494
	0.6	No	0.004
$P = \frac{nRT}{V}$	0	No	0.997
	0.2	No	-0.387
	0.4	No	-6.881
	0.6	No	-0.332
F = mg $\cdot$ $cos(\theta)$	0	No	0.079
	0.2	No	0.068
	0.4	No	0.059
	0.6	No	0.081
Stress Strain Relationship: $\frac{x_0}{x_1} + (\frac{x_0}{x_3})^{x_3}$	0	No	1.000
	0.2	No	0.214
	0.4	No	0.042
	0.6	No	0.017

APPENDIX C

Equation	Independent Variable Range (Uniform Distribution)	Detect Symmetries with Constant Alpha	Detect Symmetries with Adaptive Alpha
Generic Symmetries			
$(x_0 - x_1)^2 + (x_2 - x_3)^2 + (x_4 - x_5)^2$	-100 to 100	Yes	Yes
	-50 to 50	Yes	Yes
	-10 to 10	Yes	Yes
	-1 to 1	No	Yes
	-0.5 to 0.5	No	Yes

$x_0 \cdot (x_1 - x_2)^2$	-100 to 100	Yes	No
	-50 to 50	Yes	No
	-10 to 10	No	No
	-1 to 1	No	Yes
	-0.5 to 0.5	No	Yes
$x_0 \cdot (x_1 - x_2)^2 + x_3 \cdot (x_4 - x_5)^2$	-100 to 100	Yes	Yes
	-50 to 50	Yes	Yes
	-10 to 10	Yes	Yes
	-1 to 1	No	Yes
	-0.5 to 0.5	No	Yes
$(x_0 \cdot x_1) - (x_2 \cdot x_3)$	-100 to 100	No	No
	-50 to 50	No	No
	-10 to 10	No	No
	-1 to 1	No	Yes
	-0.5 to 0.5	No	Yes
$(x_0 \cdot x_1) \cdot (x_2 \cdot x_3)$	-100 to 100	Yes	No
	-50 to 50	Yes	No
	-10 to 10	Yes	Yes
	-1 to 1	No	Yes
	-0.5 to 0.5	No	Yes
Physics Equations			
$x_0^2 + 2x_1(x_2-x_3)$	-100 to 100	Yes	Yes
	-50 to 50	Yes	Yes
	-10 to 10	No	Yes
	-1 to 1	Yes	Yes

	-0.5 to 0.5	Yes	Yes
$x_0 \cdot x_1 \cdot (x_2 - x_3)$	-100 to 100	Yes	No
	-50 to 50	Yes	No
	-10 to 10	Yes	Yes
	-1 to 1	No	Yes
	-0.5 to 0.5	No	Yes
$\sin(x_0 \cdot x_1)$	-100 to 100	No	No
	-50 to 50	No	No
	-10 to 10	No	No
	-1 to 1	No	Yes
	-0.5 to 0.5	No	Yes
$\frac{1}{2}x_0 \cdot (x_1^2 - x_2^2)$	-100 to 100	No	No
	-50 to 50	No	No
	-10 to 10	No	No
	-1 to 1	No	No
	-0.5 to 0.5	No	No
Fake Symmetry Tests			
$(x_0 - x_1) + x_1^2 + x_2$	-100 to 100	Yes	Yes
	-50 to 50	Yes	Yes
	-10 to 10	Yes	Yes
	-1 to 1	No	Yes
	-0.5 to 0.5	No	Yes
$x_0 + x_0^2 + x_1 - x_2$	-100 to 100	Yes	Yes
	-50 to 50	Yes	Yes
	-10 to 10	Yes	Yes

	-1 to 1	No	Yes
	-0.5 to 0.5	No	Yes
$x_0 \cdot (x_0 - x_1 - 5)$	-100 to 100	Yes	No
	-50 to 50	Yes	No
	-10 to 10	No	No
	-1 to 1	No	No
	-0.5 to 0.5	No	No

APPENDIX D: A BRIEF OVERVIEW OF THE BINOMIAL COEFFICIENT

The binomial coefficient of two integers *n* and *k* can be calculated in the following way:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad \text{(Equation 16)}$$

where $n!$ is the factorial operation, or $n(n - 1)(n - 2)...(3)(2)(1)$. The calculations for equations 13 and 14 can be found below:

$$\binom{7}{4} = \frac{7!}{4!(7-4)!} = \frac{7!}{4!3!} = \frac{7\cdot6\cdot5\cdot4\cdot3\cdot2\cdot1}{4\cdot3\cdot2\cdot1\cdot3\cdot2\cdot1} = \frac{7\cdot6\cdot5}{3\cdot2} = 35 \qquad \text{(Equation 17)}$$

$$\binom{11}{6} = \frac{11!}{6!5!} = \frac{11\cdot10\cdot9\cdot8\cdot7\cdot6\cdot5\cdot4\cdot3\cdot2\cdot1}{6\cdot5\cdot4\cdot3\cdot2\cdot1\cdot5\cdot4\cdot3\cdot2\cdot1} = \frac{11\cdot10\cdot9\cdot8\cdot7}{5\cdot4\cdot3\cdot2\cdot1} = 462 \qquad \text{(Equation 18)}$$

APPENDIX E: CODE

Here is the link to our Jupyter notebook, containing all of our code, unit tests, and documentation.

https://colab.research.google.com/drive/1nsy8Q_pyYPyvLleGMXwKgnh4ovB7TtKh?usp=sharing