

Electronics and Computer Science
Faculty of Engineering and Physical Sciences
University of Southampton

Ashwinkrishna Azhagesh

25/03/2025

An AI Approach to Chaotic Physical Systems:

Project supervisor: **Adam Peugeot**

Second examiner: **David Millard**

Progress report submitted for the award of
Bachelors of Science

Abstract

Empirical laws are mathematical generalisations found through observing the physical world. It has taken us centuries of gathering data, keen research along with repeated experiments, and no doubt plenty of talented scientists to discover these laws. Leading us to understand everything from the mysteries that govern the collision of two objects to the shape of the path planets thread upon.

Recent advances in neural networks including increases in computational power permit us to train models, that replicate, fasten and automate our discovery of empirical laws. This extends to even noisy chaotic systems such as the double pendulum. Combined with white box models, symbolic regression and explainable A.I., we can peer into the "mind," of how such models, process data and conclude their observations. Human cognition is inherently finite in its capacity for thought and observational ability, has been historically overcome through the development of new tools such as the microscope. Similarly, cognitive biases can be mitigated, by utilising artificial intelligence, which is a rapidly emerging technology capable of expanding our perception and analysis.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.

- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.

- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

I have not used any resources produced by anyone else.

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else.

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for

this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

ECS Statement of Originality Template, updated August 2018, Alex Weddell aiofficer@ecs.soton.ac.uk

Abstract

I would like to thank my supervisors, Professor Adam Peugeot and Professor David Millard, for all the help and advise I received throughout this project.

Contents

Abstract	7
Statement of Originality	7
1 Introduction:	7
1.1 Motivation:	7
2 Previous Work:	7
2.1 Literature Review:	7
3 Noise:	7
3.1 How noise affects the model:	7
3.2 How to mitigate noise in data:	7
3.3 Modelling the noise:	8
4 Writing my own symbolic regressor from scratch:	8
4.1 The core;	8
4.1.1 exploiting physical properties	8
4.1.2 Dealing with constants:	8
4.1.3 Dealing with powers:	9
4.1.4 Chaining powers and constants:	9
4.1.5 Loading data:	9
4.1.6 Evaluating expressions:	10
4.2 Polynomial Fit module:	10
4.2.1 Data Loading:	11
4.2.2 Generating polynomial expressions:	11
4.2.3 Filtering the Polynomial expressions:	11
4.2.4 Evaluating expressions:	11
4.3 Dimensional Analysis:	12
4.3.1 Handling Units:	12
4.3.2 Construct Matrix and Target Vector:	12
4.3.3 Solving Dimension and Basis Units:	13
4.3.4 Data Transformation Function:	13
4.3.5 Symbolic Transformation Generator	13
4.4 Testing:	13
5 Applying the model to Biological Data:	14
6 Broder Use Cases:	14
7 Project Management:	14
8 Conclusion:	14
References	14
Appendix:	14
9 Project Planning:	14

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 section saying 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 Previous Work:

2.1 Literature Review:

3 Noise:

In this section, I aimed to explore how noise affects the model, and potential ways to mitigate it. Continuing onwards from the previous model, in the data generation step, noise was artificially added, and the results were observed.

So in order to model the noise, I used the python random library, and generated random numbers between 0 and an ever increasing amount of randomness, in order to gauge the accuracy as noise increased for the model. I was also part

3.1 How noise affects the model:

So in order to add noise to the generated data set, I imported random, and used the randn.int function. In order to vary the inputs, another function was created that incrementally passes in higher numbers as parameters to the random function, allowing each set of generated data to incrementally become more and more noisy. Then the symbolic regression model is run on these new data sets, and the resulting equations levels of noise are then plotted in a graph. Furthermore using the Time library to measure the amount of time it takes to run the model as the amount of random error increases.

3.2 How to mitigate noise in data:

Ways to mitigate the noise and its 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 average 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 algorithm.

The second denoise algorithm 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 pseudo 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."

3.3 Modelling the noise:

4 Writing my own symbolic regressor from scratch:

4.1 The core;

The core and essential part of any symbolic regression model, lies in the way it at the simplest level, generates and traverses the search tree of possible equations and expressions that may fit the data presented to it.

In order to save time, and to test if my expression generation was working as intended, i have started with simple 2 variable equations, and also pass in the specific operations used in the equation. Furthermore this is also extended to handle constants and more later on.

enter in the pseudo code here.

Then I further improved this, by designing a recursive way to generate these expressions, to allow to generate more robust equations from the given variables. Also this is dynamic, so it can

enter in pseudo code

4.1.1 exploiting physical properties

The next step is to then start to truncate these generated expressions as much as possible to prune the search tree. One of the ways you can do this is through exploiting the symmetrical property of physical equations and how they are mathematically equivalent. Such as removing duplicate expressions.

This is how I achieved that.

Give pseudo code here.

4.1.2 Dealing with constants:

Another way i further pruned the amount of expression, is through filtering all the expressions generated through the newer recursive generator, by removing all the expressions that did not contain all the specified variables. This is in order to save further time later on during the evaluation section.

Insert in pseudo code:

Then later on i designed it next to work with nested expressions and i wanted it to work with more than one constant.

insert in some code that has changed.

I had then filtered out any of the expressions generated that did not have both of the constants. This is mainly to save on some computation time in evaluating some of the redundant expressions. This is how I managed to do that.

Insert in pseudo code.

4.1.3 Dealing with powers:

So applying powers to expressions, I applied the power to the expression. This also allows you to prune the search tree further by not needed to generate redundant expressions with powers. This is the pseudo code.

Then I filtered based on if the expression contained the power, this allows me to further prune the tree. In a more robust model, this is derived from scratch, however for the sake of computation time, and flexibility, I decided to proceed with this approach as it saves some time.

this is the pseudo code.

4.1.4 Chaining powers and constants:

The next step is to chain together powers and constants, such that both are applied to the expressions. It can already be chained as with the design it already has. However it needs to be filtered properly in order to maintain the least amount of expressions possible.

The constants filter can be used, but the powers are inside the constants, and therefore the older power filter does not work as intended. Therefore i needed to redesign it such that it will function recursively.

This is the code.

However, sometimes this gives off constants that are chained, such as $\sin(\sin())$, and so to filter this further, I want to filter out expressions with more than one instance of the constant that is chained.

this is the code - filter single constant

4.1.5 Loading data:

Next I needed a way to load the data I have typed up. At this point I was focused on testing as quick as possible and in order to proceed in a prompt manner I made up some dummy data values. Afterwards I made the decision to keep the data as a numpy array, because this will be faster than a text file, there are some various reasons for this, such as numpy arrays being stored in memory, the efficiency of the underlying data format it is stored in (binary), and finally numpy uses C, and so it vectorises operations, making it far faster.

Insert in pseudo code.

As you can see I check if the number of variables entered matches the shape of the array in X, which here is the input data, and y being the target data, as in the final result. I.e x contains the mass and acceleration values, and y is the array of the result of the equation $f = ma$, so it only contains the value of f in it. This is a basic check to make sure the number of columns all have a corresponding variable.

4.1.6 Evaluating expressions:

Next I evaluate the expressions that I had generated, and I assign the variables to a column of the data, in increasing order. Then this is substituted into the equation, and the expressions are run, and there is an array of outputs of the expression. This essentially evaluates every generated expressions that has been pruned, and returns a np array of the results of those expressions based on the input data.

insert in pseudo code here.

Then like the paper suggested, insert in paper here, I used a medium error description length loss function, and have implemented it in the same way as in the paper. Using error squared, making all the errors positive, and added 1 as a constant to ensure that all the errors are greater than the value, when taking the log.

Insert in pseudo code.

Then furthermore I also implemented 2 other loss algorithms, specifically root mean squared loss as well as mean absolute error.

insert in pseudo code.

This was to help bridge and improve upon the loss algorithm used in the paper, as these two have their own advantages, and a combined hybrid approach seemed smarter.
Explain why later.

4.2 Polynomial Fit module:

Now that the simple, core of the algorithm works, and is adapted to take care of constants, powers, variables, generate expressions, and filter out the redundancy using physical properties of the world such as symmetry, I now aimed to further extend the program by writing a polynomial fit module. The aim of the polynomial fit technique is to.

Why: Many functions in physics (or parts of them) are low-order polynomials.

Why: It's a computationally very cheap method for this specific function class.

How: It attempts to fit the given data to a sum of polynomial terms.

How: It generates all possible polynomial terms up to a specified low degree (e.g., degree 4).

How: For each data point, this creates a linear equation where the unknowns are the polynomial coefficients.

How: It solves the resulting system of linear equations using standard methods like least squares.

How: The Root Mean Squared Error (RMSE) of the fit is calculated.

How: If the RMSE is below a predefined tolerance (ϵ_{pol}), the polynomial is accepted as a solution.

Effect: It acts as a fast base case in the recursive algorithm, quickly solving problems that are simple polynomials.

Effect: It can also solve sub-problems that are transformed into polynomials by other modules (e.g., dimensional analysis, inverting the function).

4.2.1 Data Loading:

So to start, I began by creating the data loading function. The aim was to take in a numpy array, with the data, along with a list of variables. Then comparing the shape of the data column and the number of variables in order to make sure the input is sufficient.

This is the pseudo code.

4.2.2 Generating polynomial expressions:

Then the next step is to generate polynomial expressions, and then it will return a list of polynomial expressions on the list.

Pseudo code.

This function, `generate_polynomial_expressions`, creates all possible symbolic polynomial expressions by combining a list of coefficients.

4.2.3 Filtering the Polynomial expressions:

The `filter_expressions` function programmatically filters symbolic expressions using structural and semantic constraints. Leveled for large – scales symbolic filtering tasks where strict mathematical structure must be enforced.

insert in pseudo code.

This initial version only worked for symbolic constants, ie sin, cos etc, and didn't work for numbers, or integer coefficients, I caught this error during testing and I rewrote the function so that it works for integer coefficients.

insert code:

4.2.4 Evaluating expressions:

Now I need to take the filtered expressions, and try fit the model to the dataset np array. The model fitting logic function fits polynomial expressions to input data by finding the best set of coefficients that minimize the error between the predicted and actual output values. It tests multiple polynomial degrees (up to a specified maximum) and selects the one that provides the lowest error, ensuring an optimal balance between accuracy and complexity.

I use root mean squared error to calculate the loss, and the function returns a list of loss, per expression.

So i take an expression, then I substitute in the variables using the input data, calculate the predicted y value of the said equation, then take it away from the true value of y the target and then use that to calculate the rmse.

pseudo code.

Best polynomial fit:

Now that I have the list of expressions and the corresponding rmse values, i pick the lowest rmse as the most accurate polynomial fit for the data.

insert in pseudo code

4.3 Dimensional Analysis:

Physics Constraint: Physical equations must be dimensionally consistent (units on both sides must match).

Strong Simplification: This dimensional constraint severely limits the possible forms of the unknown function.

First Step: AI Feynman applies dimensional analysis as the very first attempt to simplify the problem.

Unit Representation: Units of variables (like mass, length, time) are represented as vectors of integer powers.

Linear System: A linear system is set up based on the unit vectors of the input variables and the target variable.

Dimensionless Combinations: Solving this system and finding the null space reveals combinations of variables that are dimensionless.

Problem Transformation: The original problem is transformed into finding a function of these new, dimensionless variables.

Reduced Variables: This process typically reduces the number of independent variables the algorithm needs to search over.

Search Space Reduction: A smaller number of variables drastically shrinks the combinatorial search space for subsequent steps.

Efficiency Boost: It makes Polynomial Fit, Brute Force, and Neural Network-guided searches significantly faster and more likely to succeed.

4.3.1 Handling Units:

So i went to the ai feynman database website, and downloaded their units.csv to get a better idea of all the units in the dataset, I was dealing with. Then i had a look at all the required units, and then made a unit table, array, so that each unit corresponds to a unique power of the basic si units which I also implemented, as an array/list.

code:

4.3.2 Construct Matrix and Target Vector:

This function constructs the dimensional matrix M and target vector b essential for dimensional analysis. It accepts lists of independent and dependent variable names and a dictionary mapping variable names (keys) to their unit vectors (values).

Technical Implementation: Unit vectors for independent variables are retrieved via dictionary lookup, using lowercase variable names (`var.lower()`) to ensure case-insensitivity. These vectors are efficiently assembled into the columns of matrix M using `numpy.column_stack`. The dependent variable's unit vector forms vector b. `try...except KeyError` handles missing units.

Design Rationale: Case-insensitivity enhances usability. `numpy.column_stack` offers performance. Explicit error handling prevents silent failures.

code:

4.3.3 Solving Dimension and Basis Units:

This function determines the exponents for dimensional scaling and dimensionless groups by solving the linear systems $Mp = b$ and $MU = 0$. It takes the dimensional matrix M and target vector b as input. The function first converts these NumPy arrays into SymPy matrices (`sp.Matrix`) to leverage symbolic computation capabilities. It then attempts to find an exact particular solution p for $Mp = b$ using SymPy's LU-solve method, chosen for its ability to yield rational solutions. Robust error handling via `try...except` addresses potential issues like inconsistent systems. Finally, it calculates the null space basis U of M using `M_sym.nullspace()`, which identifies the combinations forming dimensionless groups. The function returns the symbolic solution p and the null space basis U .

Think of it like this: you have a target physical quantity (b , like Force) that depends on several input quantities (M , like mass, length, time).

Finding the "Unit-Fixing" Part ($p = M_{sym}.LU\text{solve}(b_{sym})$) :

The function first figures out the specific combination of powers (p) of your input variables (x) that you need to multiply together (x^p) so that the result has the exact same physical units as your target variable (b).

For example, if the target is Force ($[M L T^2]$) and inputs are mass ($[M]$), length ($[L]$), and time ($[T]$), it would find p corresponding to $\text{mass}^1 * \text{length}^1 * \text{time}^2$.

It uses `LUsolve` from SymPy to try and find an exact (often simple fraction or integer) solution for these powers p .

Finding the "Dimensionless Combinations" ($U = M_{sym}.nullspace()$) :

After accounting for the basic units, any remaining relationship must involve combinations of input variables that have no units at all (they are dimensionless numbers, like Reynolds number).

The function finds all the fundamental ways (U) you can combine powers of the input variables (x^u) such that the units completely cancel out.

In essence, the function:

Separates the part of the formula responsible for getting the units right (p).

Identifies all the core dimensionless building blocks (U) that the rest of the formula must be made from.

This allows the main algorithm to later focus on finding the relationship between these dimensionless quantities, which is a simpler problem than the original one involving various physical units.

4.3.4 Data Transformation Function:

This function converts original physical data ($data_x, data_y$) into a dimensionless form using previously calculated exponents (p, U).

How it Works: First, it calculates a scaling factor for each data point by raising input variables ($data_x$) to the powers specified in p and dividing the result by the target variable ($data_y$).

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_vars`) and the exponent vectors for scaling (p) and dimensionless combinations (U).

How it Works: It first creates SymPy symbolic objects for each input variable name using `sp.symbols`. Utilizing these symbols and the scaling exponents p , it constructs the symbolic expression `symbolic_p` representing the unit-fixing scaling factor (x^p) via `sp.Mul`. Subsequently, it iterates through each exponent vector u in U , building the corresponding dimensionless combination `symbolic_u` as `symbolic_p * (x^u)`.

insert code:

4.4 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.

5 Applying the model to Biological Data:

6 Broder Use Cases:

7 Project Management:

8 Conclusion:

9 Project Planning: