Documentation for the Task

November 2025

1 High Level Reasoning

- The first task we need to solve is to generate some data from the given system. The second part is to solve a synthesis problem using the data from we generated. **Assumption:** System 1 has 2 variables $x(t), y(t)$ and system 2 has three $x(t), y(t), z(t)$.

- The idea is that since we know that the equations are some polynomial up to degree 3 try and compute them from the given data directly. The polynomials are finite cases to be checked for both systems and the algorithm we will produce will halt and compute the results after finite steps. Actually our algorithm will compute the coefficients of the polynomials directly. The algorithm using the data will generate a possible trajectory (in the case of system 1 we will consider time step $dt = 1$ for 11 discrete steps starting from $t = 0$, while for system 2 we will do the same approach but for 20 time steps). This will produce two linear systems to be solved separately for system 1 and 3 linear systems for system 2. For the solution of the above linear systems we used the numpy library of Python for Linear Algebra. The second case is the bottom up enumeration approach which we describe later in detail.

  Comment: We have made two separate algorithms for our systems using the same approach, with slight modifications because system 1 has two equations and system 2 has three equations, but the steps are the same.

  Algorithms for systems 1 and 2
  Step 1: Generate data
  Step 2: Approximate derivatives from data
  Step 3: Build linear systems of the form A·x = b
  Step 4: Solve for coefficients using numpy
  Step 5: Return the coefficients of the polynomials

  Comment: Below we will explain for system 1. For system 2 the approach is similar

2 Lower Level Explanation

1. Since we know that the problem is up to degree 3 polynomial, then we can reduce the program space of the problem to all possible polynomials up to degree 3, this means that each equation is some polynomial of the following form:

   $$\dot{x}(t) = a_0+a_1x(t)+a_2y(t)+a_3x(t)^2+a_4x(t)y(t)+a_5y(t)^2+a_6x(t)^3+a_7x(t)^2y(t)+a_8x(t)y(t)^2+a_9y(t)^3$$

   $$\dot{y}(t) = b_0+b_1x(t)+b_2y(t)+b_3x(t)^2+b_4x(t)y(t)+b_5y(t)^2+b_6x(t)^3+b_7x(t)^2y(t)+b_8x(t)y(t)^2+b_9y(t)^3$$

   For convenience we set,

   $$\dot{x}(t) = p_1(x,\, y) \qquad \dot{y}(t) = p_2(x,\, y)$$

   Comment: We reduce our original synthesize problem to the problem of find the coefficients for the polynomials p_1, p_2.

2. Now we can get some data from the system of the following form where the values x_0, y_0 are the initial conditions that we will set in the Python Code. The values x_n, y_n are the values we get for the initial conditions. The time step (which we set at $dt = 1$ and the time from 0 to 10 for system 1).

   Comment: We took on purpose 11 points as this will later generate a square matrix 10 x 10.

x_n	y_n	t_n
x_0	y_0	0
x_1	y_1	1
x_2	y_2	2
x_3	y_3	3
x_4	y_4	4
x_5	y_5	5
x_6	y_6	6
x_7	y_7	7
x_8	y_8	8
x_9	y_9	9
x_{10}	y_{10}	10

   Comment: x_i is actually $x(t_i)$ the value at time t_i.

3. Now we use the definitions of the derivative, for the data points computed in the previous step.

   $$\dot{x}(t_n) = p_1(x_n, y_n) = \frac{x_{n+1} - x_n}{dt}, \dot{x}(t_n) = p_2(x_n, y_n) = \frac{y_{n+1} - y_n}{dt}.$$

We know the values of the derivatives at 10 points of the form (x_n, y_n) and from the last two equations above we get
$\dot{x} = p_1(x, y)$ and $\dot{y} = p_2(x, y)$ and so we get for every $n \in [0, 1, ..., 9]$

$$\dot{x}(t_n) = a_0 + a_1 x_n + a_2 y_n + a_3 x_n^2 + a_4 x_n y_n + a_5 y_n^2 + a_6 x_n^3 + a_7 x_n^2 y_n + a_8 x_n y_n^2 + a_9 y_n^3$$

$$\dot{y}(t_n) = b_0 + b_1 x_n + b_2 y_n + b_3 x_n^2 + b_4 x_n y_n + b_5 y_n^2 + b_6 x_n^3 + b_7 x_n^2 y_n + b_8 x_n b_n^2 + a_9 y_n^3$$

4. The unknowns now are the coefficients of the polynomial. Since there are 10 coefficients and we have 10 equations we have produced two systems of the form $A_1 x = b_1$ and $A_2 x = b_2$, where A_i is a 10 x 10 matrix, x are the coefficients and b_i are the approximations of the form $\dot{x}(t_n), \dot{y}(t_n)$. Each row of the matrix has the form:

$$(1, x_n, y_n, x_n^2, x_n y_n, y_n^2, x_n^3, x_n^2 y_n, x_n y_n^2, y_n^3)$$

Comment: We reduce the problem of finding the coefficients to the problem of solving the above linear systems.

5. In order to solve the above systems we use numpy library of Python in our code for linear algebra, which is possible because A is square matrix.

3 Bottom-Up Polynomial Synthesis Approach

In the second part of the implementation, we take a different perspective and treat the discovery of the dynamical system as a small program-synthesis problem. Instead of directly assuming that the governing equations are polynomials of degree 3, we define a *language of polynomials* and search over this language in increasing order of complexity.

Polynomial Language and Search Space

We restrict the space of possible right-hand sides to multivariate polynomials in the system variables. Since the task guarantees that the true equations have degree at most 3, the language contains exactly the monomials:

$$1,\ x,\ y,\ x^2,\ xy,\ y^2,\ x^3,\ x^2 y,\ xy^2,\ y^3.$$

Rather than using the full basis from the start, we organise the search space into three levels:

- **Size 1:** Polynomials of degree ≤ 1, generated by $\{1, x, y\}$.
- **Size 2:** Polynomials of degree ≤ 2, generated by $\{1, x, y, x^2, xy, y^2\}$.
- **Size 3:** Polynomials of degree ≤ 3, i.e. as.

At each size level, we build a corresponding design matrix whose rows evaluate the selected monomials on the trajectory points generated earlier.

Least-Squares Reconstruction

For each model size, we solve a linear system of the form

$$Ax = b,$$

where b contains the finite-difference approximations of the derivatives and A contains the evaluations of the selected monomials at each time step.

Unlike the first implementation—where the matrix was square and solved via exact linear algebra, we now use the least-squares method implemented using `numpy.linalg.lstsq`. This allows us to compare how well each polynomial size fits the dynamics.

If the error for a small model (e.g., degree 1) is already sufficiently low, we accept that model and do not move to more complex polynomials. Otherwise, we continue to size 2 and finally to size 3.